SANDIA REPORT

SAND2010-6313
Unlimited Release
Printed September 2010

Adagio 4.18 User's Guide

SIERRA Solid Mechanics Team
Computational Solid Mechanics and Structural Dynamics Department
Engineering Sciences Center

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

Sandia National Laboratories is a multi-program laboratory operated by Sandia Corporation,
a wholly owned subsidiary of Lockheed Martin Corporation, for the United States Department of Energy's
National Nuclear Security Administration under Contract DE-AC04-94AL85000.

Sandia National Laboratories

SAND2010-6313
Unlimited Release
Printed September 2010

Adagio 4.18 User's Guide

SIERRA Solid Mechanics Team
Computational Solid Mechanics and Structural Dynamics Department
Engineering Sciences CenterSandia National Laboratories
Box 5800
Albuquerque, NM 87185-0380

Abstract

Adagio is a Lagrangian, three-dimensional, implicit code for the analysis of solids and structures. It uses a multi-level iterative solver, which enables it to solve problems with large deformations, nonlinear material behavior, and contact. It also has a versatile library of continuum and structural elements, and an extensive library of material models. Adagio is written for parallel computing environments, and its solvers allow for scalable solutions of very large problems. Adagio uses the SIERRA Framework, which allows for coupling with other SIERRA mechanics codes. This document describes the functionality and input structure for Adagio.

Acknowledgments

This document is the result of the collective effort of a number of individuals. The current development team responsible for Adagio and Presto, the SIERRA Solid Mechanics codes, includes Nathan K. Crane, Jason D. Hales, Martin W. Heinstein, Alex J. Lindblad, David J. Littlewood, Kendall H. Pierson, Vicki L. Porter, Nathaniel S. Roehrig, Timothy R. Shelton, Gregory D. Sjaardema, Benjamin W. Spencer, and Jesse D. Thomas. This document is written and maintained by this team.

Outside the core development team, there are number of other individuals who have contributed to this manual. Nicole L. Breivik, J. Franklin Dempsey, Jeffery D. Gruda, and Chi S. (David) Lo have provided valuable input from the user community as Presto and Adagio Product Managers.

Many others have contributed to this document, either directly or by providing suggestions. These include, but are not limited to Manoj K. Bhardwaj, James V. Cox, Arne S. Gullerud, Daniel C. Hammerand, J. Richard Koteras, Jakob T. Ostien, Rhonda K. Reinert, William M. Scherzinger, and Gerald W. Wellman.

Contents

List of Figures

List of Tables

Adagio 4.18 Release Notes

Following is a list of new features and syntax changes made to Adagio since the 4.16 release.

Material Model Documentation

Documentation has been added for a number of material models. These include Thermoelastic (Section 4.2.2), Neo-Hookean (Section 4.2.3), Power Law Creep (Section 4.2.12), K&C Concrete (Section 4.2.14), Low Density Foam (Section 4.2.16), and Wire Mesh (Section 4.2.18).

Linear Shell

A fully linear shell element is available. See Section 5.2.4.

New Methods for Defining Beam/Shell Offset

Shell offset can now be defined based on a mesh variable, and beam offset can now be defined in the global coordinate system either directly in the input file or by a mesh variable. See Sections 5.2.4 and 5.2.6.

Point Mass for Rigid Bodies

Mass may now be added to a rigid body at a user-specified location. See Section 5.3.1.

Element Disconnection For Shells

The element disconnection method for handling failure with element death is now available for shell elements. See Section 5.5.3.3.

Initialization of Variables with Weibull Distribution

Field variables can be initialized with a spatially varying random variable conforming to the Weibull probability distribution function. See Section 6.2.4.

MPCs for Volumetric Constraints

The capability to use MPCs for tied contact has been extended to allow for volumetric constraints between slave nodes and the nodes on an enclosing surface. This is primarily intended to be used to constrain volumes of void elements to an enclosing solid. See Section 6.10.2.

Interaction Defaults

Adagio can now select rational contact interaction defaults when using the `GENERAL CONTACT = ON` command line in the `INTERACTION DEFAULTS` contact command block. Contact will automatically select a rational master/slave arrangement for all block-to-block interactions and use default tolerances and friction models for these interactions. See Section 7.

Hourglass Energy by Block

The hourglass energy sums are now available on each block as global variables. See Section 8.2.1.8.

Output Synchronization

The results, restart, heartbeat, and history output can be synchronized with databases of the same type in other regions and output from coupled applications. See Sections 8.2.1.16, 8.3.10, 8.4.10, and 8.5.12.

User Output Data Filtering

Frequency based data filters can now be automatically computed off of nodal and element quantities. See Section 8.2.2.6.

Surface Normal Data Transfer

The `VARIABLE INTERPOLATION` command block may now be used to transfer only component of a variable normal to a surface. See Section 8.7.

Global Energy Reporting

Options for global energy reporting have been introduced to adjust accuracy and performance. See Section 8.8.

Global Rigid Body Variable Output

The default global rigid body variable output may now be modified. See Section 8.8.

RVE Usage Enhancements

A representative volume element (RVE) with nonmatching opposing surfaces can be used by including a block of membranes surrounding the RVE. This membrane block is required to have matching opposing surfaces and must be tied to the underlying RVE. See Section 9.1.

Adagio 4.18 Known Issues

Section 3.3: Deactivation of element blocks (see Section 5.1.5.8) does not currently work in conjunction with the full tangent preconditioner in Adagio. To use this capability, one of the nodal preconditioners must be used.

Section 5.2.8: Superelements are not compatible with several modeling capabilities. They cannot be used with element death. They cannot be used with node-based, power method, or Lanczos critical time step estimation methods. They are also not compatible with some preconditioners (such as FETI) for implicit solutions.

Section 6.3.2.2: If a prescribed displacement with the `CYLINDRICAL AXIS` option is applied to nodes that fall on the axis, it will have no effect. Separate boundary conditions should be applied to those nodes to fix them in the plane normal to the axis.

Section 6.3.3.2: If a prescribed velocity with the `CYLINDRICAL AXIS` option is applied to nodes that fall on the axis, it will have no effect. Separate boundary conditions should be applied to those nodes to fix them in the plane normal to the axis.

Section 8.4: User defined variables (see Section 10.2.4) are not currently supported with heartbeat output.

Section 9.2: Currently, the J-Integral evaluation capability is based on assumptions of elastostatics and a stationary crack, and is only implemented for uniform gradient hex elements.

Chapter 1

Introduction

This document is a user's guide for the code Adagio. Adagio is a three-dimensional, implicit solid mechanics code with a versatile element library, nonlinear material models, and capabilities for modeling large deformation and contact. Adagio is a parallel code, and its nonlinear solver and contact capabilities enable scalable solutions of large problems. It is built on the SIERRA Framework [1, 3]. SIERRA provides a data management framework in a parallel computing environment that allows the addition of capabilities in a modular fashion.

The *Adagio 4.18 User's Guide* provides information about the functionality in Adagio 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, the command structure related to the use of various element types is grouped in one chapter; descriptions of material models are grouped in another chapter.

The input and usage of Adagio is similar to that of the code Presto [2]. Presto, like Adagio, is a solid mechanics code built on the SIERRA Framework. The primary difference between the two codes is that Presto uses explicit time integration for transient dynamics analysis, whereas Adagio is an implicit code.

Because of the similarities in input and usage between Adagio and Presto, the user's guides for the two codes are structured in the same manner and share common material. (Once you have mastered the input structure for one code, it will be easy to master the syntax structure for the other code.) To maintain the commonality between the two user's guides, we have used a variety of techniques. For example, references to Presto may be found in the Adagio user's guide and vice versa, and the chapter order across the two guides is the same.

On the other hand, each of the two user's guides is expressly tailored to the features of the specific code and documents the particular functionality for that code. For example, though both Presto and Adagio have contact functionality, the content of the chapter on contact in the two guides differs.

Important references for both Adagio and Presto are given in the references section at the end of this chapter. Adagio was preceded by the codes JAC and JAS3D; JAC is described in Reference 4; JAS3D is described in Reference 5. Presto was preceded by the code Pronto3D. Pronto3D is described in References 6 and 7. Some of the fundamental nonlinear technology used by both Presto and Adagio are described in References 8, 9, and 10. Currently, both Presto and Adagio use the Exodus II database and the XDMF database; Exodus II is more commonly used than XDMF. (Other options may be added in the future.) The Exodus II database format is described in Reference 11, and the XDMF database format is described in Reference 12. Important information about contact is provided in the reference document for ACME [13]. ACME is a third-party library for contact.

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 Adagio. 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 [11]. 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 more detail in Chapter 3.) Other capabilities documented in this chapter are available for calculating element distortion and for activating and deactivating functionality at different times throughout an analysis.
- Chapter 3 discusses the multilevel, nonlinear iterative solver in Adagio. This chapter also describes how to set start time, end time, and time blocks for an analysis.
- Chapter 4 describes material models that can be used in conjunction with the elements in Presto and Adagio. Most of the material models have an interface that allows the models to be used by the elements in both codes. Even though a material model can be used by both codes, it may be that the use of the material model is better suited for one code rather than for the other code. For example, a material model set up to characterize behavior over a long time would be better suited for use in Adagio than in Presto. If a material model is better suited for one of the two codes, this information will be noted for the material model. In some cases, a material model may only be included in one of the two user's guides. Chapter 4 also discusses the application of temperature to a mesh and the computation of thermal strains (isotropic and anisotropic).
- Chapter 5 lists the elements in Presto and Adagio and describes how to set up commands to use the various options for the elements. Most elements can be used in either Presto or Adagio. If an element is available in one code but not the other, this information will be noted

for the element. In some cases, an element may only be included in one of the two user's guides. For example, Presto has a special element implementation referred to as smoothed particle hydrodynamics (SPH). The Presto user's guide contains a section on SPH, but the Adagio user's guide does not. Chapter 5 also includes descriptions of the commands for mass property calculations, element death, and rigid bodies.

- Chapter 6 documents how to use kinematic boundary conditions, force boundary conditions, initial conditions, and specialized boundary conditions.
- Chapter 7 discusses how to define interactions of contact surfaces.
- Chapter 8 details the various options for obtaining output.
- Chapter 9 documents special modeling techniques.
- Chapter 10 provides an overview of the user subroutine functionality.
- Appendix A provides a sample input file from an analysis of an eraser being pulled across a surface. This problem emphasizes large deformation and contact.
- Appendix B lists all the permissible Adagio input lines in their proper scope.
- The index allows you to find information about command blocks and command lines. 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 each chapter are listed at the end of the respective chapters rather than in a separate references chapter. The reference sections in the chapters are not necessarily edited so that they are specific to Adagio or Presto. Some chapters will have exactly the same set of references (even if not all are cited for a particular user's guide), and some chapters will have the references tailored to the specific user's guide.

1.2 Overall Input Structure

Adagio is one of many mechanics codes built on the SIERRA Framework. The SIERRA Framework provides the capability to perform multiphysics analyses by coupling together SIERRA codes appropriate for the mechanics of interest. Input files may be set up for analyses using only Adagio, or they may be set up to couple Adagio and one or more other SIERRA analysis codes. For example, you might run Adagio to compute a stress state, and then use the results of this analysis as initial conditions for a 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 Adagio reflects the fact that it 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 Adagio.)

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 SIERRA scope. The SIERRA 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 coupled 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 SIERRA 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 SIERRA scope. The procedure scope controls the overall analysis from the start time to the end time; the region scope controls a single time step. For a multiphysics analysis, the SIERRA scope could contain several different procedures and several different regions.

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

Inside the region scope for Adagio 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 Adagio 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 SIERRA scope, for example, is defined by a command block that begins with an input line of the following form:

```
  BEGIN SIERRA my_problem
```

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

```
END SIERRA my_problem
```

In the above input line, END and SIERRA are the key words to end this command block. The SIERRA scope can also be terminated simply by using the following key word:

```
END
```

The above 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 either of the following forms:

```
END PRESCRIBED DISPLACEMENT
```

```
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 following 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 this form:

```
COMPONENT = X
```

The form above 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 this 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 Adagio 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 Adagio 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 Adagio. 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 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 follows:

```
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 follows:

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

In the above example, 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 the following:

```
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 as follows:

```
COMPONENTS = Z

COMPONENTS = Z X

COMPONENTS = Y X Z

COMPONENTS = Z Y X
```

An example of an invalid expression would be the following:

```
COMPONENTS = Y Y Z
```

1.3.6 Known Issues and Warnings

Where there are known issues with the code, these are documented in the following manner:

⚠ **Known Issue:** A description of the known issue with the code would be provided here.

Similarly, warnings regarding usage of code features that are not defective, but must be used with care because of their nature, are documented as follows:

⚠ **Warning:** A description of the warning related to the usage of a code feature would be provided here.

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 `$` symbol and the end-of-line. If the first non-blank character in a line is a `#` or `$`, 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 the character strings in the input lines are case insensitive. For example, the `BEGIN SIERRA` key words could appear as one of the following:

```
BEGIN SIERRA
begin sierra
Begin Sierra
```

You could specify a `SIERRA` command block with:

```
BEGIN SIERRA BEAM
```

and terminate the command block with this input line:

```
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 the following forms:

```
node set= nodelist_10
```

```
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 following line:

```
BEGIN SIERRABEAM
```

However, it is harder to check this line for the correct spelling of the key words and the intended SIERRA scope name than this 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 following correct form:

```
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 following expected name:

```
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 following 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 = separates key words from input values in a command line. Consider the following command line:

```
COMPONENTS = X
```

Here, the key word `COMPONENTS` is separated from its value, a string in this case, by the =. 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 = provides a strong visual cue for separating key words from values. By using the = 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 accidentally delete =, it is much easier to detect than accidentally 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, 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. 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 the following specification:

```
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. 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 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, but not required, 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 the command blocks in the correct structure.

1.4.11 Including Files

External text files containing input commands can be included at any point in the Adagio input file using the `INCLUDEFILE` command. This command can be used in any context in the input file. To use this command, simply use the command `INCLUDEFILE` followed by the name of the file to be included. For example, the command:

```
INCLUDEFILE displacement_history.i
```

would include the `displacement_history.i` as if the contents of that file were places in the position that it is included in the input file. The included file is contained in the standard echo of the input that is provided at the beginning of the log file.

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 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 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 identifier is 125, the value of `surface_name` would be `surface_125`. It is also possible to generate an alias for the side set[1] 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. It is also possible to name a surface in some mesh generation programs and that name can be used in the input file.

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. It is also possible to name a nodelist in some mesh generation programs and that name can be used in the input file.

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. It is also possible to name an element block in some mesh generation programs and that name can be used in the input file.

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

[1] See the `ALIAS` command in Section 5.1.2

1.6 Major Scope Definitions for an Input File

The typical input file will have the structure shown below. The major scopes—SIERRA, 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 SIERRA
  # scope appear here. The PROCEDURE ADAGIO command
  # block is the beginning of the next scope.
  #
  # function definitions
  # material descriptions
  # description of mesh file
  #
  BEGIN ADAGIO PROCEDURE <string>procedure_name
    #
    # time step control
    #
    BEGIN ADAGIO REGION <string>region_name
      #
      # All command blocks and command lines in the
      # region scope appear here
      #
      # solver commands
      # specification for output of result
      # specification for restart
      # boundary conditions
      # definition of contact
      #
    END [ADAGIO REGION <string>region_name]
  END [ADAGIO PROCEDURE <string>procedure_name]
END [SIERRA <string>some_name]
```

1.7 Input/Output Files

The primary user input to Adagio 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 that Adagio 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 Adagio. Both Adagio and Presto use the same file structure. Therefore, in Figure 1.1, we indicate that the code (graphically represented by the central cylinder) can be either Presto or Adagio.

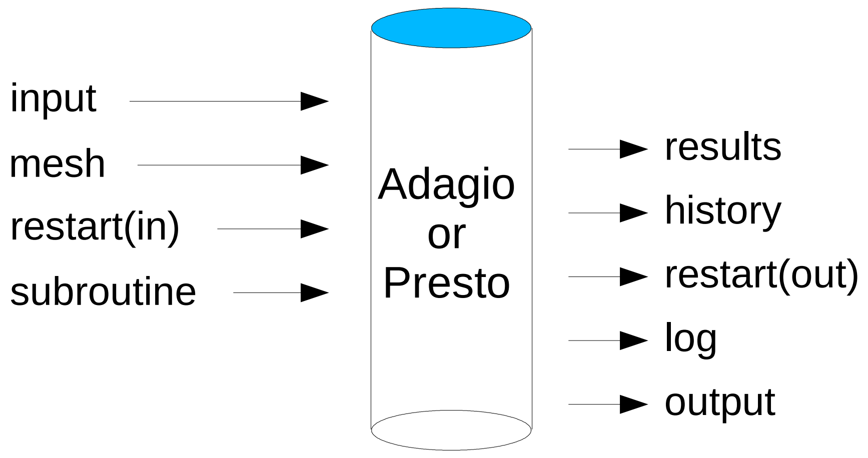


Figure 1.1: Input/output files

As shown in Figure 1.1, Adagio uses the input file, mesh files, restart files, and user subroutine files. The input file, which is required, is a set of valid Adagio 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 Adagio 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 Adagio (Chapter 10).

As also shown in Figure 1.1, Adagio 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 Adagio 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 n^{th} 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 Obtaining Support

Support for all SIERRA Mechanics codes, including Adagio, can be obtained by contacting the SIERRA Mechanics user support hotline by email at sierra-help@sandia.gov, or by telephone at (505)845-1234.

1.9 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. Koteras, J. R., A. S. Gullerud, V. L. Porter, W. M. Scherzinger, and K. H. Brown. "PRESTO: Impact Dynamics with Scalable Contact Using the SIERRA Framework." In *First MIT Conference on Computational Fluid and Solid Mechanics*, edited by K. J. Bathe, 294–296. Amsterdam: Elsevier, 2001.

3. 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.

4. Biffle, J. H. *JAC – A Two-Dimensional Finite Element Computer Program for the Non-Linear Quasi-Static Response of Solids with the Conjugate Gradient Method*, SAND81-0998. Albuquerque, NM: Sandia National Laboratories, April 1984. pdf.

5. Blanford, M. L., M. W. Heinstein, and S. W. Key. *JAS3D – A Multi-Strategy Iterative Code for Solid Mechanics Analysis Users' Instructions, Release 2.0*, Draft SAND report. Albuquerque, NM: Sandia National Laboratories, September 2001.

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

7. Attaway, S. W., K. H. Brown, F. J. Mello, M. W. Heinstein, J. W. Swegle, J. A. Ratner, and R. I. Zadoks. *PRONTO3D User's Instructions: A Transient Dynamic Code for Nonlinear Structural Analysis*, SAND98-1361. Albuquerque, NM: Sandia National Laboratories, June 1998. pdf.

8. Laursen, T. A., S. W. Attaway, and R. I. Zadoks. *SEACAS Theory Manuals: Part I. Problem Formulation in Nonlinear Solid Mechanics*, SAND98-1760/1. Albuquerque, NM: Sandia National Laboratories, August 1998. pdf.

9. Laursen, T. A., S. W. Attaway, and R. I. Zadoks. *SEACAS Theory Manuals: Part II. Nonlinear Continuum Mechanics*, SAND98-1760/2. Albuquerque, NM: Sandia National Laboratories, September 1998. pdf.

10. 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, March 1999. pdf.

11. Larry A. Schoof, Victor R. Yarberry, *EXODUS II: A Finite Element Data Model*, SAND92-2137, Sandia National Laboratories, September 1994. pdf. See also documentation available at EXODUS II Sourceforge page. link.

12. The eXtensible Data Model and Format (XDMF). link.

13. 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*. SAND2001-3318. Albuquerque, NM: Sandia National Laboratories, October 2001. pdf.

Chapter 2

General Commands

The commands described in this section appear in the SIERRA or procedure scope or control general functionality in Adagio.

2.1 SIERRA Scope

These commands are used to set up some of the fundamentals of the Adagio input. The commands are physics independent, or at least can be shared between physics. The commands lie in the SIERRA 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 SIERRA scope defined by
  # begin/end sierra.
  #
END [SIERRA <string>name]
```

All input commands 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 as follows:

```
END [SIERRA <string>name]
```

In these input lines, `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 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 Adagio allows a user to run an analysis up to a certain time, stop the analysis at this time, and then restart the analysis from this 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 a 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. However, not all parameters can be reset. Users should exercise care in resetting parameters in the input file for a restart. You will 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 SIERRA scope and the region scope**. One of two restart command lines, `RESTART` or `RESTART TIME`, appears in the SIERRA 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 8.5. This section gives a brief discussion of the command lines that appear in the SIERRA scope. For a full discussion of all the command lines used for restart, consult 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 SIERRA 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 **SIERRA 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 (e.g., the `RESTART` command line in Section 2.1.3.2) prevents the overwriting of restart, results, and history files. See Section 8.5 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. See Section 8.5 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 10.

2.1.5 Functions

```
BEGIN DEFINITION FOR FUNCTION <string>function_name
  TYPE = <string>CONSTANT|PIECEWISE LINEAR|PIECEWISE CONSTANT|
    ANALYTIC
  ABSCISSA = <string>abscissa_label
    [scale = <real>abscissa_scale(1.0)]
    [offset = <real>abscissa_offset(0.0)]
  ORDINATE = <string>ordinate_label
    [scale = <real>ordinate_scale(1.0)]
    [offset = <real>ordinate_offset(0.0)]
  X SCALE = <real>x_scale(1.0)
  X OFFSET = <real>x_offset(0.0)
  Y SCALE = <real>y_scale(1.0)
  Y OFFSET = <real>y_offset(0.0)
  BEGIN VALUES
    <real>x_1   <real>y_1
    <real>x_2   <real>y_2
    ...
    <real>x_n   <real>y_n
  END [VALUES]
  AT DISCONTINUITY EVALUATE TO <string>LEFT|RIGHT(LEFT)
  EVALUATE EXPRESSION = <string>analytic_expression1;
    analytic_expression2; ...
  DEBUG = ON|OFF(OFF)
END [DEFINITION FOR FUNCTION <string>function_name]
```

A number of Adagio 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 SIERRA 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 four options to define the type of function. The value of this string can be `CONSTANT`, `PIECEWISE LINEAR`, `PIECEWISE CONSTANT`, 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 user can optionally add a scale factor and/or an offset which has the following effect: $abscissa_{scaled} = scale * (abscissa + offset)$.

- 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 user can also optionally add a scale factor and/or an offset which has the following effect: $ordinate_{scaled} = scale * (ordinate + offset)$.

- The `X SCALE` command line sets the scale factor value for the abscissa and has the same effect as if the optional `SCALE` command were used in the `ABSCISSA` command line.

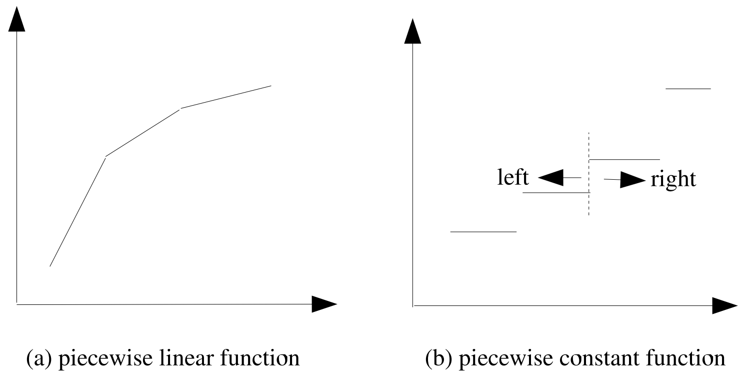
- The `X OFFSET` command line sets the offset value for the abscissa and has the same effect as if the optional `OFFSET` command were used in the `ABSCISSA` command line.

- The `Y SCALE` command line sets the scale factor value for the ordinate and has the same effect as if the optional `SCALE` command were used in the `ORDINATE` command line.

- The `Y OFFSET` command line sets the offset value for the ordinate and has the same effect as if the optional `OFFSET` command were used in the `ORDINATE` command line.

- The `DEBUG command` line prints functions to the log file if they were scaled and/or offset. This command line is optional. The generated function function name is the original function name concatenated with `_with_scale_and_offset_applied`. The generated function is a valid function and could be placed into the input file and used.

- The `VALUES` command block consists of the real value pairs (`x_1`,`y_1`) through (`x_n`, `x_n`), which describe the function. This command block must be used if the value on the `TYPE` command line is `CONSTANT`, `PIECEWISE LINEAR`, or `PIECEWISE CONSTANT`. For a `CONSTANT` function, only one value is needed. For a `PIECEWISE LINEAR` or `PIECEWISE CONSTANT` function, the values are (x, y) pairs of data that describe the function. The values are nested inside the `VALUES` command block.



(a) piecewise linear function (b) piecewise constant function

Figure 2.1: Piecewise linear and piecewise constant functions

A PIECEWISE LINEAR function performs linear interpolations between the provided value pairs; a PIECEWISE CONSTANT function is constant valued between provided value pairs. Figure 2.1 (a) shows an example of a piecewise linear function, and Figure 2.1 (b) shows an example of a piecewise constant function.

For functions that are based on tabular values, such as the PIECEWISE LINEAR and PIECEWISE CONSTANT functions, there is a possibility that the function may be evaluated for abscissa values that fall outside the range of the tabulated values. If the abscissa for which the function is to be evaluated is lower than the lowest tabulated abscissa, the function returns the ordinate corresponding to the smallest tabulated abscissa. Likewise, if the function is evaluated for an abscissa higher than the highest tabulated abscissa, the function returns the ordinate corresponding to the highest tabulated abscissa. For example, consider the following function:

```
begin definition for function my_func
  type = piecewise linear
  begin values
    5.0  0.0
    10.0 50000.0
  end values
end definition for function my_func
```

For values less than 5, this function returns 0, and for values greater than 10, it returns 50000.

- For a piecewise constant function, a constant valued segment ends on the left hand side of an abscissa value and a new constant value segment begins on the right hand side of the same abscissa value. (This transition from one constant value to another is indicated by the dotted line in Figure 2.1 (b).) When the value of the function is to be evaluated at a discontinuity, where there two potential values for the ordinate, the default behavior is to use the ordinate from the value pair that has the lower-valued abscissa, or in other words, to use the value on the left hand side of the discontinuity. The AT DISCONTINUITY EVALUATE TO command line can be used to override this default behavior at an abscissa with two ordinate values. The command line can have a value of either LEFT or RIGHT. If LEFT (the default) is specified, the ordinate value to the left of the abscissa is used; if RIGHT is specified, the ordinate value to the right of the abscissa is used.

- 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
```

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 Adagio 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, Adagio input permits the definition of points, directions, and axes. Definition of these geometric entities occurs in the SIERRA 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(2) = <real>theta(0.0)
END [ORIENTATION <string>orientation_name]
```

The `ORIENTATION` command block is currently used in Adagio to define a local *co-rotational* coordinate system at each shell element centroid for output of shell in-plane stresses and strains. Each `BEGIN SHELL SECTION` command block has an orientation that generates this local coordinate system for each element in the associated element blocks. The generated local coordinate system rotates with the shell element, thus making it *co-rotational*.

Stresses and strains for shell elements are computed in the shell element's local coordinate system, which typically varies from one element to the next, and can be seen in Figure 2.2. The local **R** axis of element 1 does not align with the local **R** axis of element 2. Thus without the use of an orientation to rotate the local coordinate system in to a consistent coordinate system, stress and strain results would be difficult to decipher. The use of an orientation generates a transformation matrix from a shell element's local **R**, **S**, **T** coordinate system to the final coordinate system, **R′**, **S′**, **T′**, by use of orientation and element-specific basis vectors and a rotation about an axis, all defined below.

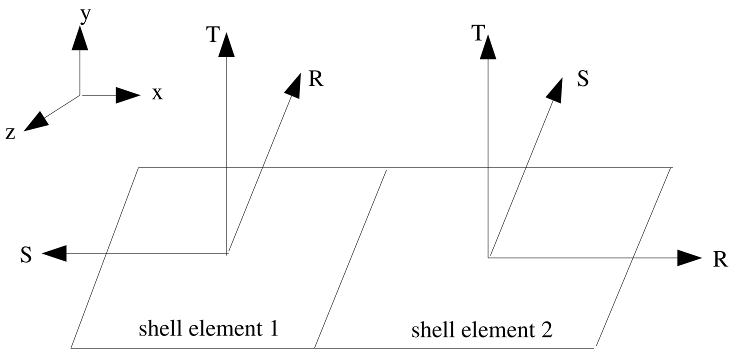


Figure 2.2: Adjacent shell elements with nonaligned local coordinate systems

There are four orientation systems available in Adagio to better align the stress and strain output of shell elements. These orientations are defined via the optional `SYSTEM` command line within the `ORIENTATION` command block and are: `RECTANGULAR`, `Z_RECTANGULAR`, `CYLINDRICAL` and `SPHERICAL` with the default being `RECTANGULAR`. If the `ORIENTATION` command line within a `SHELL SECTION` block is not used, the default orientation is a `RECTANGULAR` system with axes aligned with the global X,Y and Z axes.

Each of the four systems produces a unique set of bases vectors, $\mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3$, computed using the centroid of the shell element and the two required inputs, `POINT A` and `POINT B`. These four methods of computing $\mathbf{g}_1, \mathbf{g}_2, \mathbf{g}_3$ are described below.

- `RECTANGULAR`: `POINT A` defines a point, P_A, in the local coordinate system of the shell element that lies in the direction of the the first basis vector, $\mathbf{g}_1$, from the centroid, C with local coordinate of $(0, 0, 0)$, see Figure 2.3. Thus, the vector from the centroid of the element to point P_A is defined as $\mathbf{p}_A = P_A - C$. Similarly, a vector from the centroid to point P_B, defined via `POINT B`, can be computed as $\mathbf{p}_B = P_B - C$. Using these two vectors, the rectangular basis vectors are defined as: $\mathbf{g}_1 = \frac{\mathbf{p}_A}{\|\mathbf{p}_A\|}$, $\mathbf{g}_3 = \frac{\mathbf{g}_1 \times \mathbf{p}_B}{\|\mathbf{g}_1 \times \mathbf{p}_B\|}$, $\mathbf{g}_2 = \frac{\mathbf{g}_3 \times \mathbf{g}_1}{\|\mathbf{g}_3 \times \mathbf{g}_1\|}$.

- `Z RECTANGULAR`: This set of basis vectors is defined very similarly to the `RECTANGULAR` system, however point P_A now defines a point that lies in the direction of the third basis vector, $\mathbf{g}_3$ from the centroid of the element, see Figure 2.4. Using the same definition for $\mathbf{p}_A$

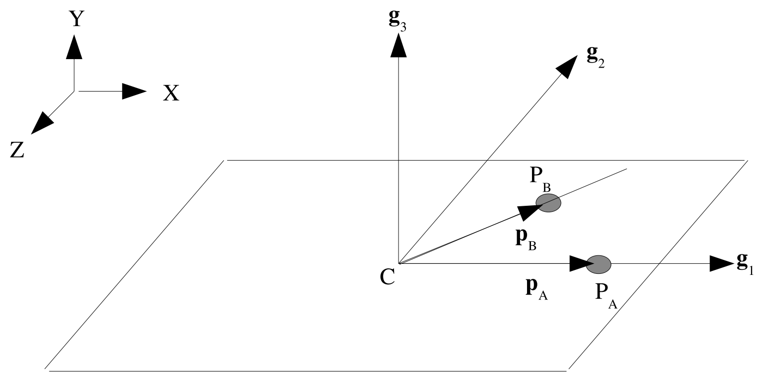


Figure 2.3: Rectangular coordinate system

and $\mathbf{p}_B$ as above we can define the three basis vectors as: $\mathbf{g}_3 = \frac{\mathbf{p}_A}{\|\mathbf{p}_A\|}$, $\mathbf{g}_2 = \frac{\mathbf{g}_3 \times \mathbf{p}_B}{\|\mathbf{g}_3 \times \mathbf{p}_B\|}$, $\mathbf{g}_1 = \frac{\mathbf{g}_2 \times \mathbf{g}_3}{\|\mathbf{g}_2 \times \mathbf{g}_3\|}$.

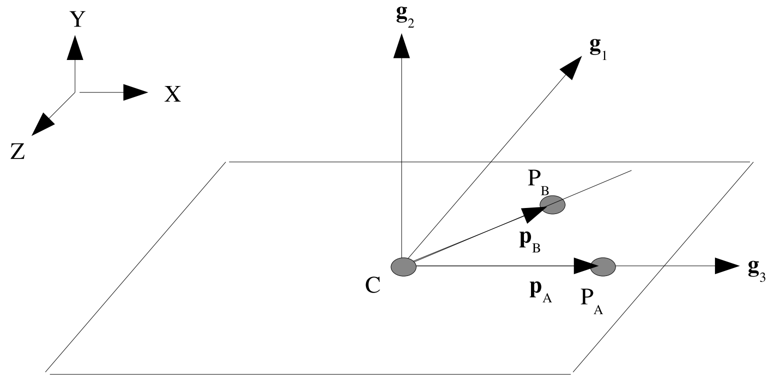


Figure 2.4: Z-Rectangular coordinate system.

- `CYLINDRICAL`: `POINT A`, P_A, and `POINT B`, P_B, define the direction of the third basis vector, $\mathbf{g}_3$ such that $\mathbf{g}_3 = \frac{P_B - P_A}{\|P_B - P_A\|}$, as seen in Figure 2.5. Defining a vector, $\mathbf{v}$, from P_A to the centroid of the element, C in the global coordinate system, the second basis vector, $\mathbf{g}_2$ is defined as: $\mathbf{g}_2 = \frac{\mathbf{v} \times \mathbf{g}_3}{\|\mathbf{v} \times \mathbf{g}_3\|}$. Therefore the first basis vector is defined as $\mathbf{g}_1 = \frac{\mathbf{g}_2 \times \mathbf{g}_3}{\|\mathbf{g}_2 \times \mathbf{g}_3\|}$.

- `SPHERICAL`: The point P_A, from the `POINT A` command line, defines the center of a sphere. The point P_B, from the `POINT B` command line, defines a polar axis for the sphere. (See Figure 2.6.) The first basis vector is defined by the vector that passes from P_A to the centroid, C in the global coordinate system, of the element: $\mathbf{g}_1 = \frac{C - P_A}{\|C - P_A\|}$. The third basis vector is parallel to the vector from P_A to P_B such that $\mathbf{g}_3 = \frac{P_B - P_A}{\|P_B - P_A\|}$. From these two vector, the

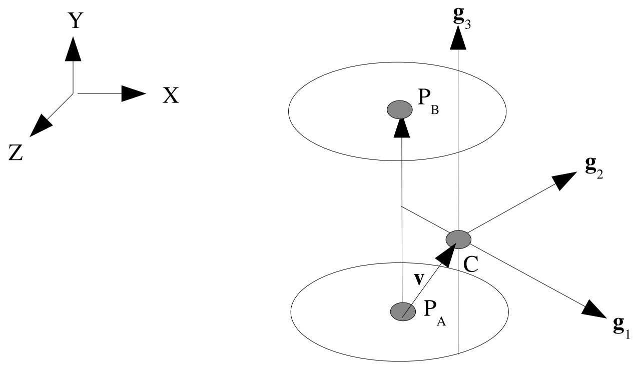


Figure 2.5: Cylindrical coordinate system.

second basis vector is defined as $\mathbf{g}_2 = \frac{\mathbf{g}_3 \times \mathbf{g}_1}{\|\mathbf{g}_3 \times \mathbf{g}_1\|}$. If $\mathbf{g}_1$ and $\mathbf{g}_3$ are collinear, the second basis vector is taken to be the normal of the element, $\mathbf{T}$, and the third basis is $\mathbf{g}_3 = \frac{\mathbf{g}_1 \times \mathbf{g}_2}{\|\mathbf{g}_1 \times \mathbf{g}_2\|}$.

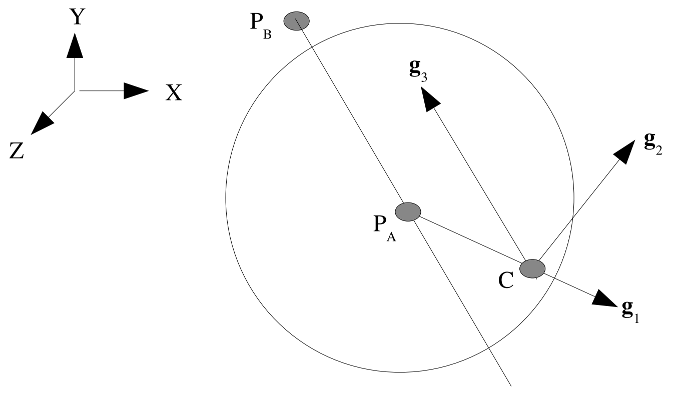


Figure 2.6: Spherical coordinate system.

It is possible, using the `ROTATION ABOUT` command line, to specify an angle and an axis to rotate the basis vectors, $\mathbf{g}_1$, $\mathbf{g}_2$ and $\mathbf{g}_3$ about in order to produce a second set of basis vectors, $\mathbf{g}'_1$, $\mathbf{g}'_2$ and $\mathbf{g}'_3$, that are then used to compute the final transformation matrix. The syntax for this command is as follows: `ROTATION ABOUT 1|2|3(2) = <real>theta(0.0)`

Where the 1,2,3 refers to which basis vector the rotation will occur about. The default value is 2 with an angle theta of 0.0, thus implying a rotation of 0.0 radians about $\mathbf{g}_2$. How $\mathbf{g}'_1$, $\mathbf{g}'_2$ and $\mathbf{g}'_3$ are computed for each of the three cases is described below. In each case, the second order rotation tensor, **Q**, is defined as:

$$\mathbf{Q} = \begin{bmatrix} 2(q_0^2 + q_1^2) - 1 & 2(q_1 q_2 - q_0 q_3) & 2(q_1 q_3 - q_0 q_3) \\ 2(q_1 q_2 + q_0 q_3) & 2(q_o^2 + q_2^2) - 1 & 2(q_2 q_3 - q_0 q_1) \\ 2(q_1 q_3 - q_0 q_2) & 2(q_2 q_3 + q_0 q_1) & 2(q_0^2 + q_3^2) - 1 \end{bmatrix}$$

Where: $q_0 = cos(\frac{\theta}{2})$, $q_1 = a_1 sin(\frac{\theta}{2})$, $q_2 = a_2 sin(\frac{\theta}{2})$, and $q_3 = a_3 sin(\frac{\theta}{2})$, where the vector $\mathbf{a} = [a_1, a_2, a_3]$ is the basis vector we are rotating about.

- Rotation about $\mathbf{g}_1$ (`ROTATION ABOUT 1`): The prime set of basis vectors are defined as: $\mathbf{g}_1' = \mathbf{Q}\mathbf{g}_3$, $\mathbf{g}_2' = \mathbf{Q}\mathbf{g}_2$, $\mathbf{g}_3' = -\mathbf{g}_1$. Which can graphically be seen in Figure 2.7

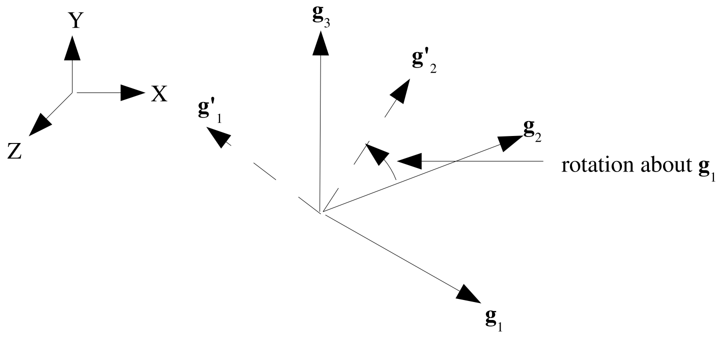


Figure 2.7: Rotation about 1

- Rotation about $\mathbf{g}_2$ (`ROTATION ABOUT 2`), default: The prime set of basis vectors are defined as: $\mathbf{g}_1' = \mathbf{Q}\mathbf{g}_1$, $\mathbf{g}_2' = \mathbf{Q}\mathbf{g}_3$, $\mathbf{g}_3' = -\mathbf{g}_2$.

- Rotation about $\mathbf{g}_3$ (`ROTATION ABOUT 3`): The prime set of basis vectors are defined as: $\mathbf{g}_1' = \mathbf{Q}\mathbf{g}_2$, $\mathbf{g}_2' = \mathbf{Q}\mathbf{g}_1$, $\mathbf{g}_3' = -\mathbf{g}_3$.

The next step is to project $\mathbf{g}_1'$ onto the shell face as follows: $\mathbf{R}' = \mathbf{g}_1' - (\mathbf{g}_1' \cdot \mathbf{T})\mathbf{T}$, which defines a rotated **R** vector. If $\mathbf{R}'$ is perpendicular to the face of the element, we then redefine $\mathbf{R}'$ to be: $\mathbf{R}' = \mathbf{g}_2' - (\mathbf{g}_2' \cdot \mathbf{T})\mathbf{T}$, the projection of $\mathbf{g}_2'$ onto the element face. Once we have $\mathbf{R}'$ we can define $\mathbf{S}'$ as $\frac{\mathbf{T}\times\mathbf{R}'}{\|\mathbf{T}\times\mathbf{R}'\|}$. Hence, the final transformation matrix taking local stress and strain (in the **R**, **S** coordinate system) to the $\mathbf{R}'$, $\mathbf{S}'$ coordinate system is:

$$\mathbf{Q}' = \begin{bmatrix} \mathbf{R}\cdot\mathbf{R}' & \mathbf{R}\cdot\mathbf{S}' & 0 \\ \mathbf{S}\cdot\mathbf{R}' & \mathbf{S}\cdot\mathbf{S}' & 0 \\ 0 & 0 & 1 \end{bmatrix}$$

2.1.8 Coordinate System Block Command

```
BEGIN COORDINATE SYSTEM <string>coordinate_system_name
  TYPE = <string>RECTANGULAR|CYLINDRICAL|SPHERICAL (RECTANGULAR)
  #
  ORIGIN = <string>origin_point_name
  VECTOR = <string>z_vector_point_name
  POINT = <string>x_vector_point_name
  ORIGIN NODE = <string>origin_nodelist_name
  VECTOR NODE = <string>z_vector_nodelist_name
  POINT NODE = <string> x_vector_nodelist_name
  #
END [COORDINATE SYSTEM <string>coordinate_system_name]
```

The `COORDINATE SYSTEM` command block is used to define a local coordinate system for transforming a stress tensor from components in the global xyz coordinate system to the local system for output. This command block cannot be used for in-plane stresses and strains of shell elements because the shell integration point stresses and strains are computed in a local element system that varies element to element and rotates with the element. For output of shell stresses, the `BEGIN ORIENTATION` command must be used as explained in the previous section.

The `COORDINATE SYSTEM` block is also used to define a local coordinate system on the elements of a block for use in a representative volume analysis. In these analyses, elements of each representative volume will be aligned with the local system defined on the parent element.

A local element coordinate system may be defined using points defined in the input file with the `DEFINE POINT` command or using nodelists in the mesh file that contain exactly one node each. By using nodes in the mesh file, the defined coordinate system may translate and rotate during the analysis.

The `TYPE` command line allows three options for constructing a local coordinate system: `RECTANGULAR`, `CYLINDRICAL`, and `SPHERICAL`. The type defaults to `RECTANGULAR` if the `TYPE` command line is not present.

- `RECTANGULAR`: The command line `ORIGIN` or `ORIGIN NODE` specifies the origin of the local Cartesian coordinate system. The local Z' axis is a vector from the origin to the point given in the `VECTOR` of `VECTOR NODE` command. The local X' axis is the component of the vector from the origin to the `POINT` or `POINT NODE` that is orthogonal to the Z' axis. Finally, the local Y' axis is obtained from the cross product of Z' and X'.

- `CYLINDRICAL`: The command line `ORIGIN` or `ORIGIN NODE` specifies one point on the axis of the cylinder. The local Z' axis, the axis of the cylinder, is a vector from the origin to the point given in the `VECTOR` or `VECTOR NODE` command. The local X' axis is constructed as the vector normal to the cylindrical axis and passing through the location at which the local system is desired. Depending on the context, this local point may be a node, an element centroid, or an element integration point. If this local point lies on the Z' axis, then the point defined by the `POINT` or `POINT NODE` command is instead used to define the X' axis.

Finally, the Y' axis is obtained from the cross product of Z' and X'. Thus at the desired point not on the cylinder axis, the X' axis is through the cylinder thickness, Y' is tangent to the cylinder, and Z' is parallel to the cylinder axis.

- `SPHERICAL`: The command line `ORIGIN` or `ORIGIN NODE` specifies the location of the center of the sphere. The local X' axis is constructed as the vector from the center through the location at which the local system is desired. Depending on the context, this local point may be a node, an element centroid, or an element integration point. The local Z' axis is the component of the vector from the origin to the point given in the `VECTOR` or `VECTOR NODE` command that is normal to the X' axis. If this Z' is parallel the X' axis, then the Z' axis is defined along the vector from the point defined in `POINT` or `POINT NODE` to the origin. Finally, the Y' axis is obtained from the cross product of Z' and X'. Thus at a desired point, the X' axis is through the sphere thickness and the Y' and Z' axes lie in the tangent plane.

2.1.9 Define Coordinate System Line Command

```
DEFINE COORDINATE SYSTEM <string>coord_sys_name <string>coord_sys_type
  WITH POINT <string>point_1 POINT <string>point_2
  POINT <string>point_3
```

The line command `DEFINE COORDINATE SYSTEM` can also be used to define the axis directions of a local coordinate system to be located at nodes, element centroids, or element integration points. In this command

- The string `coord_sys_name` is a name for this coordinate system. This name must be unique to all other coordinate systems defined in the input file.
- The string `coord_sys_type` states the type of the coordinate system to be used. Three options are allowed for constructing a local coordinate system: `RECTANGULAR`, `CYLINDRICAL`, and `SPHERICAL`.
- The strings `point_1`, `point_2`, and `point_3` are the names for three points defined in the input file via `DEFINE POINT` command lines. These three points are used to define two of the coordinate system axes as described below for the different types of systems.

For a rectangular system, the local Z' axis is parallel to the vector from the `point_1` to `point_2`. The local X' axis is the component of the vector from `point_1` to `point_3` that is orthogonal to the Z' axis. Finally, the local Y' axis is obtained from the cross product of Z' and X'.

Likewise, for a cylindrical system, the local Z' axis is parallel to the vector from the `point_1` to `point_2` and defines the axis of the cylinder. The local X' axis is constructed as the vector normal to the cylindrical axis and passing through the location at which the local system is desired. This local point may be a node, an element centroid, or an element integration point. If this local point lies on the Z' axis, then the point `point_3` is instead used to define the X' axis. Finally, the Y' axis is obtained from the cross product of Z' and X'. Thus at the desired point not on the cylinder axis, the X' axis is through the cylinder thickness, Y' is tangent to the cylinder, and Z' is parallel to the cylinder axis.

For a spherical system, `point_1` specifies the center of a sphere. The local X' axis is constructed as the vector from `point_1` through the location at which the local system is desired. This local point may be a node, an element centroid, or an element integration point. The local Z' axis is the component of the vector from the `point_1` to `point_3` that is normal to the X' axis. If this Z' is parallel the X' axis, then `point_3` is used instead. Finally, the Y' axis is obtained from the cross product of Z' and X'. Thus at a desired point, the X' axis is through the sphere thickness and the Y' and Z' axes lie in the tangent plane.

2.2 Procedure and Region

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

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

The `TIME CONTROL` command block also appears within the `ADAGIO PROCEDURE` command block but outside of the `ADAGIO 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 Procedure

The analysis time, from the initial time to the termination time, is controlled within the procedure scope defined by the `ADAGIO PROCEDURE` command block. The command block begins with an input line of the form

```
BEGIN ADAGIO PROCEDURE <string>adagio_procedure_name
```

and is terminated with an input line of the following form:

```
END [ADAGIO PROCEDURE <string>adagio_procedure_name]
```

The string `adagio_procedure_name` is the name for the Adagio 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 allows the user to control the size of the time step.

In an implicit code such as Adagio, a solver is used to find a converged solution at every time step. The time steps can be arbitrarily large, although errors due to time discretization are larger with larger time steps. In addition, for nonlinear problems, it is often more difficult to obtain a converged solution with large time steps. The time step size is controlled by the user, although Adagio has a capability to optionally modify the time step based on the effort required by the nonlinear solver.

The `TIME CONTROL` block contains the basic commands used to control the time step size. To use adaptive time stepping, an `ADAPTIVE TIME STEPPING` command block is placed in the Adagio region scope. In addition, for all Adagio analyses, commands to control the solver are required in the Adagio region scope. The details of the commands related to time stepping and the nonlinear solver are documented in Chapter 3.

2.2.3 Region

Individual time steps are controlled within the region scope. The region scope is defined by a `ADAGIO REGION` command block that begins with an input line of the form

```
  BEGIN ADAGIO REGION <string>adagio_region_name
```

and is terminated with an input line of the following form:

```
  END [ADAGIO REGION <string>adagio_region_name]
```

The string `adagio_region_name` is the name for the Adagio 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). For Adagio, the command blocks for the solver are in the region. These blocks are discussed in Chapter 3.

The region makes use of information in the procedure and the SIERRA scope. For example, the specific element type used for an element block in the analysis model is defined in the SIERRA scope. 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 SIERRA scope.

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 scope, then we would enter the command line as follows:

```
  USE FINITE ELEMENT MODEL penetrator
```

2.4 Element Distortion Metrics

Adagio can compute a number of element distortion metrics useful for assessing the quality of the solution as the mesh evolves over time. These metrics are computed as element variables, and can be used for output or as criteria for element death, just like any other element variable. The solution quality generally deteriorates as elements approach inversion, and if they do invert, the analysis aborts. The distortion metrics measure how close an element is to inversion. For all metrics a value of 1.0 is an ideal element and a value of 0.0 is a degenerate element. The following distortion metrics are available:

- `NODAL_JACOBIAN_RATIO` is currently available only for 8 node hexahedra. This metric evaluates the Jacobian function at each node of each element, and then computes the nodal Jacobian ratio as the smallest nodal Jacobian divided by the largest nodal Jacobian. An element having right angles between all adjacent edges has a nodal Jacobian ratio of 1.0. A negative nodal Jacobian ratio indicates that the element is becoming either convex or locally inverted. See Figure 2.8 for examples of quadrilateral elements with positive, zero, and negative nodal Jacobian ratios. The element calculations on poorly shaped elements (those with negative nodal Jacobian ratios) will generally be less accurate than those on well shaped elements (those with positive nodal Jacobian ratios). In addition, contact surfaces may become tangled and non-physical if elements start to invert.

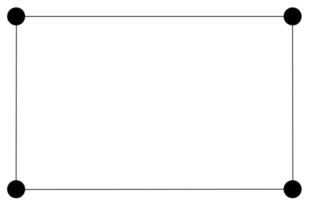
- `ASPECT_RATIO` is available only for tetrahedral elements. A perfect equilateral tetrahedron has an aspect ratio of 1.0. A degenerate zero-volume tetrahedron has an aspect ratio of zero. An inverted tetrahedron has a negative aspect ratio. A very thin element can have a very small aspect ratio.

- `SOLID_ANGLE` computes the minimum or maximum angle between edges of an element as compared to optimal angles. The optimal solid angle for tetrahedra and triangles is 60 degrees; for hexahedra and quadrilaterals, it is 90 degrees. This error metric is 1.0 for an element in which all angles are optimal. Severely distorted or twisted elements have values of this metric near 0.0, and it is negative for inverted elements. This metric is 0 for degenerate elements. The `SOLID ANGLE` metric functions with hex, tet, triangle, and quad elements.

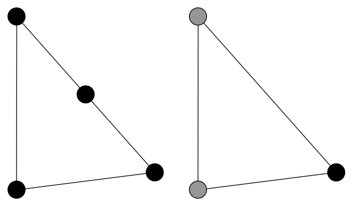
- `PERIMETER_RATIO` measures the the ratio of the deformed perimeter of an element to the undeformed perimeter of the element. This metric initially has a value of 1.0, and assumes values either greater than or lower than 1 as the mesh deforms. The `PERIMETER_RATIO` metric only works with triangle and quad elements.

- `DIAGONAL_RATIO` measures the the ratio of the deformed max diagonal of an element to the undeformed max diagonal of the element. This metric initially has a value of 1.0, and assumes values either greater than or lower than 1 as the mesh deforms. The `DIAGONAL_RATIO` metric only works with hex and quad elements.

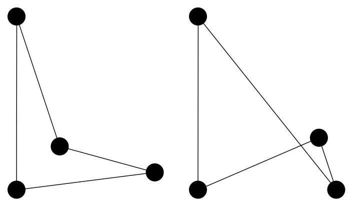
The results from any of these distortion metrics can be requested by specifying an `ELEMENT VARIABLES` command line (Section 8.2.1.4) in the `RESULTS OUTPUT` command block (Section 8.2.1) for each metric for which the results are of interest. For example, to request the



a) Nodal Jacobian = 1.0



b)Nodal Jacobian = 0.0



c) Nodal Jacobian < 0.0

Figure 2.8: Examples of elements with varying nodal Jacobians

`ASPECT_RATIO` metric to be output as an element variable named `aspect` in the results output file, the following syntax can be included in a `RESULTS OUTPUT` command block:

```
ELEMENT ASPECT_RATIO as aspect
```

2.5 Activation/Deactivation of Functionality

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

The `ACTIVE PERIODS` or `INACTIVE 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. Command blocks that support this capability are documented accordingly in the sections of this manual where they are described.

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 3.11) 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, e.g., `time_block_name`. Each defined time period runs from that period's start time to the next period's start time.

Only one of `ACTIVE PERIODS` or `INACTIVE PERIODS` can be used in any given command block. If the `ACTIVE PERIODS` command line is present, the functionality will be treated as active for all of the named periods, and inactive for any time periods that are not listed. If the `INACTIVE PERIODS` command line is present, the functionality will be treated as inactive for all of the named periods and active for any period not listed. If neither of these command lines is present, by default the functionality is active during all time periods.

2.6 Error Recovery

```
ERROR TOLERANCE = ON|OFF (ON)
```

This command controls how aggressively Adagio will attempt to recover from a non-physical or ill-defined condition and continue with the analysis.

Under the default setting of `ON`, the code may invoke methods to keep the analysis moving forward despite the presence of non-physical or ambiguous conditions. For example, if the contact algorithm determines that a facet is potentially in two contact surfaces (an ambiguous condition), Adagio will assign the facet to one of the surfaces and continue with the analysis. A second example pertains to SPH particles: if the deformation gradient of an SPH particle is inverted, the particle will be ignored rather than causing the analysis to terminate. Non-physical and ambiguous conditions will generally result in a warning message being printed.

If `ERROR TOLERANCE` is `OFF`, Adagio will terminate an analysis in the presence of ambiguous or non-physical conditions. This may make it easier for an analyst to address non-fatal but potentially problematic issues.

2.7 Manual Job Control

The output of a running job can be controlled externally through the use of "shutdown" and "control" files. This mechanism allows for additional output of restart, results, history, and/or heartbeat data to be requested, as well as an optional graceful shutdown of the job.

A graceful shutdown is requested by inserting a "shutdown" file in the working directory of a running Adagio job. The name of this file can be any of the following:

```
sierra.shutdown
<application_name>.shutdown
<base_name>.shutdown
```

If the `<application_name>.shutdown` variant is used, `<application_name>` is the name of the Sierra application being run. For example, `adagio.shutdown` would be used to shut down Adagio. If the `<base_name>.shutdown` variant is used, `<base_name>` is the basename of the input file. For example, if the input file name is `my_analysis.i`, then the shutdown file would be `my_analysis.shutdown`.

If the code detects the existence of a shutdown file, it will dump an output step to any open restart, results, history, or heartbeat file and then gracefully terminate the job. An entry will be written to the log file specifying that a shutdown file was detected and the file will be deleted. The contents of the file are not important; the shutdown file is only checked for existence.

The control file capability is provided to give more control over output and execution than is possible with the shutdown file. The name of the control file is the same as that of the shutdown file except that the filename suffix is `.control` instead of `.shutdown`. The contents of the file consist of a single line, and are case insensitive. The syntax is:

```
DUMP [RESTART] [RESULTS] [HISTORY] [HEARTBEAT] [STEP] [TIME]
  [STOP|ABORT|SHUTDOWN|CONTINUE]
```

The optional strings `RESTART`, `RESULTS`, `HISTORY`, `HEARTBEAT`, `STEP`, and `TIME` are used to specify the type of output that is written before an action is taken. If any output blocks of the specified type were defined in the input file for the model, output will be written to the files. The `STEP` and `TIME` options result in the last step and time being written to the log file. Multiple output types may be requested. If no output type is requested, all types of output specified in the input file will be written.

In addition to controlling the type of output that occurs, the `.control` file also specifies whether the job should be terminated or allowed to continue. If `STOP`, `ABORT`, or `SHUTDOWN` is specified at the end of the line, the job will be gracefully terminated. If `CONTINUE` is specified or no option is specified, the job will continue.

Several examples of the contents of continue files are shown below. The following examples all result in output being written to all types of output file and the job continuing:

```
dump
```

```
dump continue
dump restart results history heartbeat step continue
```

In both of the following examples, the current step and time will be written to the the log file, but no additional output will be written and the job will continue:

```
dump step
dump time continue
```

This example would result in a write to all output types and a graceful shutdown:

```
dump stop
```

In the following example, no output would be written to any files, but the current step and time would be written to the log file before a graceful shutdown:

```
dump step abort
```

An alternate abbreviated syntax is also supported. The abbreviated commands are shown below along with the full commands to which they are mapped:

```
sw1          dump restart shutdown
sw2          dump step continue
sw3          dump restart continue
sw4          dump results continue
```

If either the shutdown or control files are used, a message is output to the log file listing the name of that file, and in the case of the control file, the contents of that file. The control or shutdown file is then deleted.

Chapter 3

Solver, Time Stepping, and Implicit Dynamics

Nonlinear solvers are a core part of any implicit finite element code. Adagio's solution strategy is based on iterative solution techniques, and builds on the capabilities pioneered in the JAS3D code [1]. Adagio uses a nonlinear preconditioned conjugate gradient (CG) algorithm [2] to iteratively find a solution that satisfies equilibrium within a user-specified error tolerance at each load step.

The nonlinear CG algorithm iterates to find a solution by computing a series of search direction vectors that are successively added to the trial velocity vector. In each iteration, the residual is multiplied by a preconditioning matrix to obtain a gradient direction. The gradient direction is used to compute the next search direction in a way that ensures that each new search direction is orthogonal to the previous search directions. A line search is used to compute a scaling factor that when applied to the search direction results in a minimized residual.

The performance of the CG algorithm is highly dependent on the conditioning of the problem, which is affected by the choice of preconditioner. Using the inverse of the full tangent stiffness matrix as a preconditioner minimizes the number of CG iterations required, but this can be computationally expensive. An alternative is to use the inverse of a diagonalized stiffness matrix as a preconditioner. This requires much lower computational and memory resources, per iteration, but may require many more iterations.

Adagio provides several preconditioner options that can be categorized in two general types. These enable the efficient solution of a wide variety of problems using available resources. The first type of preconditioner is the nodal preconditioner. There are several different variants of nodal preconditioners available. These include the diagonal preconditioners that were available in JAS3D, in addition to other options. The second type of preconditioner provided by Adagio is the full tangent preconditioner. This option uses the FETI parallel scalable linear solver to solve the full tangent matrix for use as a preconditioner in the CG algorithm.

In addition to using a preconditioner, conditioning of a problem can be improved by using a multi-level solver. Rather than directly solving the potentially ill-posed full problem with the CG solver, Adagio forms a series of "model problems" and solves them with the core CG solver. Features of the model that make it ill-posed are either adjusted or held constant for a model problem. After each model problem is solved, the multilevel solver updates the quantities that were controlled. The process of forming model problems and updating is iterative, as with the core solver, and convergence must be achieved both in the core solver and in the multilevel controls.

Adagio provides three types of controls for the multilevel solver: "control contact," "control stiffness," and "control failure". Control contact must be used for all problems that have sliding contact, but is not needed for tied contact. Control stiffness is used to solve problems that are difficult because of extreme differences in the stiffness of various modes of material response. For instance, control stiffness is beneficial for nearly incompressible materials where the bulk response is much stiffer than the shear response. Control stiffness is also useful for oriented materials where the material response is much stiffer in some directions than it is in others. In addition, using control stiffness can greatly speed the iterative solution process when a model contains several materials that have extreme differences in stiffness. Control failure is used on problems that involve element death.

Adagio also provides a geometric multigrid algorithm called "control modes," which uses a coarse grid to solve for the low mode response. To use the control modes algorithm, a user must supply a coarse mesh that overlays the actual problem mesh. This algorithm works well for solving problems that are dominated by bending of slender members.

This chapter discusses the commands in Adagio for specifying solver options, for controlling time stepping, and for solving implicit dynamic problems. Adagio's CG solver can either be used alone or as the core solver within a multilevel solver. Section 3.1 presents the characteristics and structure of the `SOLVER` command block, which contains all solver-related commands within the `ADAGIO REGION` command block. Section 3.2 discusses the `CG` command block, which controls Adagio's nonlinear preconditioned CG solver. An option in the `CG` command block is to use a full tangent preconditioner. The command block to control the behavior of this option is discussed in Section 3.3. The recommended linear solver for use as a full tangent preconditioner is FETI, whose command block is presented in Section 3.4. The three types of controls, control contact, control stiffness, and control failure, that can be used within the multilevel solver are presented in Sections 3.5, 3.6, and 3.7. Section 3.8 explains how to activate the multigrid control modes solution method. This feature greatly increases the effectiveness of the CG algorithm for solving slender, bending-dominated problems. Section 3.9 describes the predictors, which are used to generate an initial trial solution for a load step or for a multilevel-solver model problem. As explained in Section 3.10, Adagio also has an option that allows it to use the same algorithms as the JAS3D solver. Adagio's approaches for controlling time stepping and for performing implicit solutions on quasi-static and dynamic problems are described in Sections 3.11 and 3.12, respectively. Finally, Section 3.13 provides references for this chapter.

3.1 Multilevel Solver

Adagio allows the conjugate gradient (CG) solver to be used either by itself or as the core solver within a multilevel solver. The multilevel solver improves the ability of the core solver to solve poorly conditioned problems. This is done by holding fixed some variables that would ordinarily be free to change in the fully nonlinear iteration. The multilevel solver algorithm can be used recursively to treat multiple variables by assigning them to multiple solution levels. CG iterations are performed while a variable is held fixed. After convergence is obtained with the controlled variable fixed, a new residual calculation is performed, and the controlled variable is updated. Convergence on a particular level requires that all levels leading up to that level must be converged.

Once all control variables have been established, CG iterations are performed until convergence is obtained on the core "model problem." At this point, the variable controlled at level 1 is adjusted to minimize its residual. This process is known as a "level 1 update." After the level 1 update, the model problem is solved again, and this process is repeated until the core solver and the variable controlled at level 1 have converged. Once a solution has been obtained at level 1, the variable controlled at level 2 is updated. After each level 2 update, the level 1 problem must be solved again. This process can be repeated recursively for many levels. For convergence in the multilevel solver, all updates starting with the lowest level and proceeding to the highest level must be in equilibrium. These concepts are illustrated in Figure 3.1.

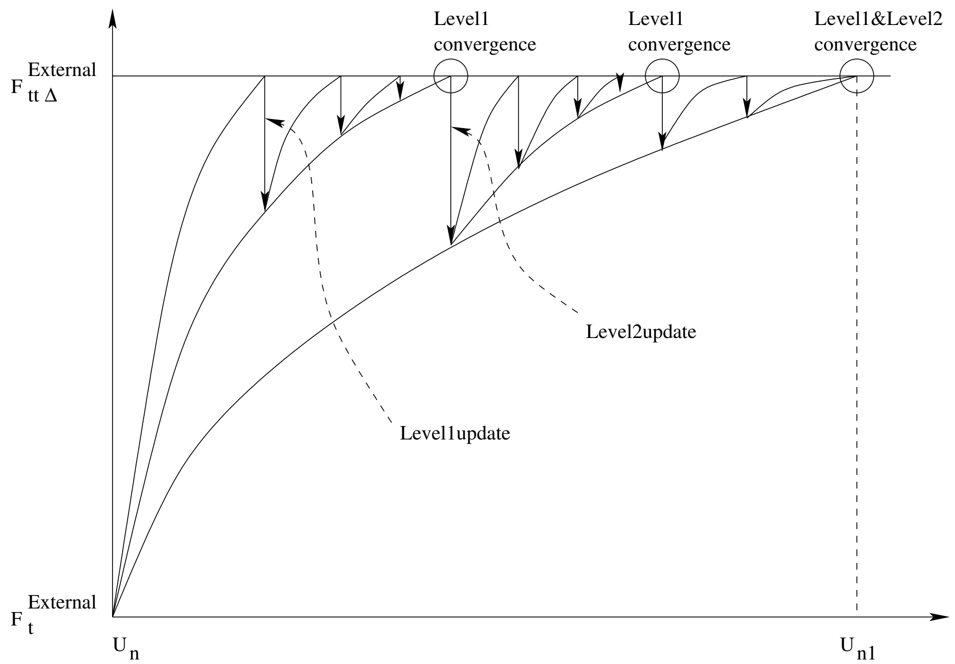


Figure 3.1: Reaching convergence with controls and the multilevel solver.

Adagio has three types of controls that can be used within the multilevel solver: control contact, control stiffness, and control failure. Control contact is used to keep the set of nodes in contact constant during the solution of a model problem. Contact searches are performed during the multilevel solver updates. Control contact must be used for all problems that have sliding contact. It is not required or beneficial for models that only have tied contact. Control contact is documented in detail in Section 3.5.

Control stiffness is used to improve the conditioning of models that contain widely varying stiffnesses in different modes of material response. Such differences naturally exist for models with nearly incompressible materials where the bulk behavior is much stiffer than the shear behavior, for models with orthotropic and/or anisotropic materials that have much higher stiffnesses in preferred material directions, and for models containing several materials that are vastly different in their overall stiffnesses. Control stiffness works by scaling a given material response up (stiffening) or down (softening) to create a set of better conditioned model problems. For nearly incompressible materials, for instance, the bulk stress increments may be softened and/or the shear stress increments may be stiffened. For oriented materials, the stress increments in certain material directions are softened to create a material response that is more isotropic in nature. Finally, for the case where a material is much stiffer than the surrounding materials, both the bulk and shear stress increments are softened to create a response that has a stiffness much closer to that of the adjacent materials. Ultimately, as the control stiffness model problems progress, the true material response is recovered by building up stresses that correspond to the true material response. There are a number of special material models that are designed to work with this capability.

Control failure is used to improve the conditioning of models that involve element death due to failure defined within a material model or due to a user-defined global variable failure criteria. Currently, failure due to element or nodal variables is not supported in adagio. This control limits the failure only to the single most critical element during each control failure iteration. Iterations of the control failure continue until no additional elements fail.

The following is the basic structure of the `SOLVER` command block:

```
BEGIN SOLVER
  #
  # cg solver commands
  BEGIN CG
    #
    # Parameters for cg
    #
  END [CG]
  #
  # control contact commands
  BEGIN CONTROL CONTACT [<string>contact_name]
    #
    # Parameters for control contact
    #
  END [CONTROL CONTACT <string>contact_name]
  #
  # control stiffness commands
  BEGIN CONTROL STIFFNESS [<string>stiffness_name]
    #
    # Parameters for control stiffness
    #
  END [CONTROL STIFFNESS <string>stiffness_name]
  #
  # control failure commands
  BEGIN CONTROL FAILURE [<string>failure_name]
    #
    # Parameters for control failure
    #
  END [CONTROL FAILURE <string>failure_name]
  #
  # predictor commands
  BEGIN LOADSTEP PREDICTOR
    #
    # Parameters for predictor
    #
  END [LOADSTEP PREDICTOR]
  LEVEL 1 PREDICTOR = <string>NONE|DEFAULT(DEFAULT)
END [SOLVER]
```

The SOLVER command block contains all solver-related commands, and must be present in the ADAGIO REGION command block. It is used regardless of whether the CG solver should be used alone or as a core solver in the multilevel solver. The CG command block (described in Section 3.2) is required for all models. If no multilevel controls are desired, the commands related to those controls are simply omitted from this block.

In addition to the CG command block, the SOLVER command block contains command blocks that describe the controls that will be used. The SOLVER command block can contain any or all of CONTROL CONTACT command block, CONTROL STIFFNESS command block, and CONTROL FAILURE command blocks described in Sections 3.5, 3.6, and 3.7.

Aside from the command blocks for the core solver and the controls, the SOLVER block contains the LOADSTEP PREDICTOR command block, which is described in Section 3.9.1, and the LEVEL 1 PREDICTOR command line, which is described in Section 3.9.2.

Convergence tolerances for the CONTROL CONTACT and CONTROL STIFFNESS solver levels are set within those command blocks while the tolerance of the core CG solver is set within the CG command block. Rather than use a convergence tolerance, the CONTROL FAILURE block is considered converged when no new elements fail during an iteration or when the maximum iterations is reached.

The commands within the CG command block are used in the same way whether or not the multilevel solver is used. It is, however, important to consider the role of tolerances within the multilevel solver. The convergence of the problem is controlled by the convergence of the highest solver level. For the multilevel solver to converge well, it is important for the core solver (or lower level of the solver, in the case of multiple levels) to reduce the residual by a certain amount during each model problem solution. For this reason, it is recommended that the solution tolerances be set an order of magnitude tighter for each lower level solver. The MINIMUM RESIDUAL IMPROVEMENT command in the CG command block is helpful for ensuring that the residual is reduced by the core solver during model problems, even though the residual might already be within the convergence tolerances (see Section 3.2.1).

3.2 Conjugate Gradient Solver

The core nonlinear preconditioned conjugate gradient solver is controlled through the `CG` command block, which must be nested within a `SOLVER` command block, regardless of whether multilevel controls are to be used if it is to be used without multilevel controls.

```
BEGIN CG
  #
  # convergence commands
  TARGET RESIDUAL = <real>target_resid
    [DURING <string list>period_names]
  TARGET RELATIVE RESIDUAL = <real>target_rel_resid
    [DURING <string list>period_names]
  ACCEPTABLE RESIDUAL = <real>accept_resid
    [DURING <string list>period_names]
  ACCEPTABLE RELATIVE RESIDUAL = <real>accept_rel_resid
    [DURING <string list>period_names]
  REFERENCE = EXTERNAL|INTERNAL|RESIDUAL(EXTERNAL)
    [DURING <string list>period_names]
  MINIMUM RESIDUAL IMPROVEMENT  =  <real>resid_improvement
    [DURING <string list>period_names]
  MINIMUM ITERATIONS  = <integer>min_iter(0)
    [DURING <string list>period_names]
  MAXIMUM ITERATIONS  = <integer>max_iter
    [DURING <string list>period_names]
  #
  # preconditioner commands
  PRECONDITIONER = BLOCK|BLOCK_INITIAL|DIAGONAL|
    DIAGSCALING|ELASTIC|IDENTITY|PROBE|SCHUR|TANGENT
    [<real>scaling_factor]
  BALANCE PROBE = <integer>balance_probe
  NODAL PROBE FACTOR = <real>probe_factor(1.0e-6)
  BEGIN FULL TANGENT PRECONDITIONER
    #
    # Parameters for full tangent preconditioner
    #
  END [FULL TANGENT PRECONDITIONER]
  #
  # line search command, default is secant
  LINE SEARCH ACTUAL|TANGENT [DURING <string list>period_names]
  LINE SEARCH SECANT [<real>scale_factor]
  #
  # diagnostic output commands
  ITERATION PRINT = <integer>iter_print
    [DURING <string list>period_names]
  ITERATION PLOT = <integer>iter_plot
    [DURING <string list>period_names]
  ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
  #
  # cg algorithm commands
  ITERATION RESET = <integer>iter_reset(10000)
    [DURING <string list>period_names]
  ORTHOGONALITY MEASURE FOR RESET = <real>ortho_reset(0.5)
    [DURING <string list>period_names]
  RESET LIMITS <integer>iter_start <integer>iter_reset
    <real>reset_growth <real>reset_orthogonality
    [DURING <string list>period_names]
  BETA METHOD = FletcherReeves|PolakRibiere|
    PolakRibierePlus(FletcherReeves)
    [DURING <string list>period_names]
END [CG]
```

Sections 3.2.1 through 3.2.5 describe the components of the `CG` command block.

3.2.1 Convergence Commands

```
TARGET RESIDUAL = <real>target_resid
  [DURING <string list>period_names]
TARGET RELATIVE RESIDUAL = <real>target_rel_resid
  [DURING <string list>period_names]
ACCEPTABLE RESIDUAL = <real>accept_resid
  [DURING <string list>period_names]
ACCEPTABLE RELATIVE RESIDUAL = <real>accept_rel_resid
  [DURING <string list>period_names]
REFERENCE = EXTERNAL|INTERNAL|RESIDUAL(EXTERNAL)
  [DURING <string list>period_names]
MINIMUM RESIDUAL IMPROVEMENT = <real>resid_improvement
  [DURING <string list>period_names]
MINIMUM ITERATIONS = <integer>min_iter(0)
  [DURING <string list>period_names]
MAXIMUM ITERATIONS = <integer>max_iter
  [DURING <string list>period_names]
```

The nonlinear preconditioned CG solver iterates to decrease the residual until the residual is deemed to be sufficiently small, based on user-specified convergence criteria. The command lines listed above are placed in the `CG` command block and used to control these convergence criteria.

Solver convergence is measured by computing the L^2 norm of the residual and comparing that residual norm with target convergence criteria specified by the user. Convergence can be monitored either directly in terms of the residual norm, or in terms of a relative residual, which is the residual norm divided by a reference quantity that is indicative of the current loading conditions on a model. Basing convergence on the relative residual is often helpful because doing so ensures that the convergence target is meaningful for the model. There are some situations, however, when it is better to check for convergence based on the actual residual rather than on the relative residual. This approach is especially helpful when the model passes through a stage during which there are no loads. Note that the actual residual, the relative residual, or both of these residuals can be used for checking convergence at the same time.

All the convergence command lines in the CG command block can have different values for different time periods by using the DURING specification. This specification consists of the key word DURING followed by a list of time periods, and is appended to the end of the command line. Each of the time periods included in the list must correspond to the name of a TIME STEPPING command block (i.e., time_block_name) specified in the TIME CONTROL command block. If the DURING specification is omitted from the command line, the value of the parameter in the command line is applicable for the entire analysis. This default value can be overwritten for specific periods by repeating the same command line and using the DURING specification on each repeated line. In other words, multiple occurrences of the same command line can be included in the command block, one without a DURING specification and each of the others with a unique DURING specification for its applicable time periods.

The TARGET RESIDUAL command line specifies the target convergence criterion in terms of the actual residual norm. The TARGET RELATIVE RESIDUAL command line specifies a convergence criterion in terms of the relative residual. If both command lines are included in the CG command block, the solver will accept the solution as converged if either the target residual or the target relative residual is below the specified values. Note that use of the term “target” in this discussion means “desired.”

The solver also allows the user to define acceptable convergence criteria. If the solution has not converged to the specified targets before the maximum number of iterations of the solver is reached, the residual is checked against the acceptable convergence criteria. These criteria are specified via the ACCEPTABLE RESIDUAL and ACCEPTABLE RELATIVE RESIDUAL command lines. The concepts of residual and relative residual are the same as those used for the target limits, i.e., in the TARGET RESIDUAL and TARGET RELATIVE RESIDUAL command lines. If the solution has not met the target criteria but has met the acceptable criteria, the solution is allowed to proceed. The default value for each acceptable criterion command line is 10 times the value of its corresponding target criterion command line, e.g., if the value of target_resid was 1.0e-6, the value of accept_resid would be 1.0e-5. Solutions that meet only the acceptable criteria are noted in the log file. If the acceptable criteria are not met, the solution at that load step has failed to converge, and the solver exits. If adaptive time stepping, as discussed in Section 3.11.2, is active, a solution of the load step may be attempted with a smaller time step. Otherwise, the code will exit at this point with an error.

If relative residuals are given in the convergence criteria, the REFERENCE command line can be used to select the method for computing the reference load. This command line has three options: EXTERNAL, INTERNAL, and RESIDUAL. When the EXTERNAL option, which is the default, is selected, the reference load is computed by taking the L^2 norm of the current external load. If the model has force boundary conditions, the external force vector used for this norm is composed of the nodal forces resulting from those boundary conditions. If there are no prescribed forces, the reaction forces at all prescribed kinematic boundary conditions are used instead. The INTERNAL option uses the norm of the internal forces as the reference load. This option is helpful in situations where a structure has no externally applied boundary conditions, such as in the case of a thermally loaded structure. Finally, the `RESIDUAL` option denotes that the residual from the initial residual should be used as the reference quantity. The initial residual is computed from the first iteration of the solver.

The `MINIMUM RESIDUAL IMPROVEMENT` command line stipulates that the CG solver must reduce the initial residual by a specified amount to obtain convergence. If this command line is included in the `CG` command block, the improvement condition must be met in addition to the standard target residual criteria. The parameter `resid_improvement` is a real number between 0.0 and 1.0 that specifies the amount by which the residual must be improved as a fraction of the initial residual in the current model problem. Thus, to stipulate that the residual must be smaller than 10% of the initial residual, one would set the value of `resid_improvement` equal to 0.9. The `MINIMUM RESIDUAL IMPROVEMENT` command line is primarily useful in the context of the multilevel solver. The model problems presented to the core solver sometimes begin with very low residuals and require very few iterations to converge. This command line forces the core solver to further improve the residual, which can accelerate the convergence of the other solver levels.

The `MAXIMUM ITERATIONS` and `MINIMUM ITERATIONS` command lines are used to specify the maximum and minimum number of core solver iterations, respectively. These limits apply for a load step if the CG solver is used in the stand-alone mode, or for a model problem if the CG solver is used as the core solver in the multilevel solver. The default value for the minimum number of iterations, `min_iter`, is zero. If a number greater than zero is specified, the solver will iterate at least that many times, regardless of whether the convergence criteria are met.

3.2.2 Preconditioner Commands

```
PRECONDITIONER = BLOCK|BLOCK_INITIAL|DIAGONAL|
  DIAGSCALING|ELASTIC|IDENTITY|PROBE|SCHUR|TANGENT
  [<real>scaling_factor]
BALANCE PROBE = <integer>balance_probe
NODAL PROBE FACTOR = <real>probe_factor(1.0e-6)
BEGIN FULL TANGENT PRECONDITIONER
#
#  Parameters for full tangent preconditioner
#
END [FULL TANGENT PRECONDITIONER]
```

The command lines listed above are used to select the type of preconditioner used by the CG algorithm, as well as the method used to form the preconditioner for some types of preconditioners. The preconditioner is applied to the residual vector to obtain a gradient direction, which in turn is used to find a search direction. The most effective preconditioner is one that closely approximates the effect of multiplying by the inverse of the tangent stiffness matrix.

There are two basic types of preconditioners available: nodal and full tangent. The nodal preconditioners provide only the diagonal terms of the full tangent stiffness matrix or 3-by-3-block diagonal matrices along the diagonal of the stiffness matrix. The diagonal nature of a nodal preconditioner allows it to be inverted very inexpensively and also use very little memory. Nodal preconditioners are so termed because they only account for the stiffness at each node in isolation and ignore the coupling between nodes that is accounted for in the off-diagonal terms of a full stiffness matrix. The result of this approximation is that with a nodal preconditioner, in a single iteration, the equilibration of a residual at a node can only cause movement in nodes that are directly connected by an element to that node. Thus, as the model size increases, the number of iterations also increases. Nodal preconditioners require many iterations, but they are often very efficient because the iterations are very inexpensive, especially if the problem is "blocky" in nature.

With the `PRECONDITIONER` command line, the user selects the nodal preconditioner. As defined below, this command line has several options. Some options are synonyms for other options.

- The `BLOCK` and `ELASTIC` options, which are synonyms, specify the default preconditioner. These options provide a nodal preconditioner with 3-by-3-block matrices that are computed based on the elastic material properties. These matrices are updated at every model problem to account for the current geometry.
- The `BLOCK_INITIAL` option, which is a variant of the `BLOCK` preconditioner, forms the preconditioner at the beginning of the analysis but never recomputes it for efficiency.
- The `DIAGONAL` preconditioner is formed in the same way as the `BLOCK` preconditioner, but only uses the terms on the diagonal.
- The `PROBE` option forms a 3-by-3-block diagonal preconditioner by probing the stiffness of nodes rather than by using the elastic stiffness.
- The `SCHUR` option, a variant of the `PROBE` preconditioner, uses a Schur Complement to approximate the coupling effects between translational and rotational degrees of freedom at a node.
- The `IDENTITY` option uses an identity matrix for the preconditioner, meaning that the residual is used as the gradient direction. This option is only of academic interest and is included for completeness.
- The `DIAGSCALING` option uses the identity matrix multiplied by the parameter `scaling_factor`. This is the only option for which the `scaling_factor` parameter is applicable. This option is also only of academic interest.

The `BALANCE PROBE` and `NODAL PROBE FACTOR` command lines control the behavior of the probing algorithm that is used to obtain the stiffness for the probe preconditioner (selected via the `PROBE` option in the `PRECONDITIONER` command line). For the probe preconditioner, the tangent stiffness K is approximated using finite differences:

$$K_{ij} = \frac{\partial F_i^{int}}{\partial x_j}, \tag{3.1}$$

where F^{int} is the internal force and x is the displacement. The nodal probe preconditioner only computes the 3-by-3-block diagonal version of the stiffness matrix. The `NODAL PROBE FACTOR` command line controls the probing distance, with the value of `probe_factor` defaulting to 1.0e-6. The probe factor controls the probing distance relative to element size. Smaller values may give a better tangent approximation, but these values could result in the generation of round-off errors.

The `BALANCE PROBE` command line selects the type of finite differencing scheme used to probe for the stiffness. The value of `balance_probe` can be set to 0, 1, or 2, as explained below.

- Setting `balance_probe` to 0, the default for the nodal probe preconditioner, causes the preconditioner to be formed by forward finite differencing. Probing occurs in only one direction, resulting in zero-order accuracy in the preconditioner. The stiffness computed when `balance_probe` is set to 0 appears as

$$K_{ij} = \frac{\partial F_i^{int}(x + \delta e_j) - F_i^{int}(x)}{\delta}, \tag{3.2}$$

  where δ is the probing distance, controlled by the `NODAL PROBE FACTOR` command line, and e_j is a unit vector in the jth equation direction.

- Setting `balance_probe` to 1, the default for the full tangent preconditioner, causes the preconditioner to be formed with central differencing. Probing at each element degree of freedom is performed in both positive and negative directions, leading to a first-order accurate estimate of the tangent stiffness. This approach can be necessary in problems with material nonlinearity, where a probe that stretches rather than compresses the element can result in orders of magnitude differences in estimated stiffness. The stiffness computed when `balance_probe` is set to 1 appears as

$$K_{ij} = \frac{\partial F_i^{int}(x + \delta e_j) - F_i^{int}(x - \delta e_j)}{2\delta}. \tag{3.3}$$

- The value of `balance_probe` may also be set to 2, which allows the full tangent preconditioner to obtain central finite differencing with fourth-order error. This approach requires twice the work as central differencing but is more accurate. The stiffness computed when `balance_probe` is set to 2 appears as

$$K_{ij} = \frac{\partial F_i^{int}(x + 2\delta e_j) + 8F^{int}(x + \delta e_j) - 8F^{int}(x + \delta e_j) - F_i^{int}(x - 2\delta e_j)}{12\delta}. \tag{3.4}$$

The `FULL TANGENT PRECONDITIONER` command block enables the full tangent preconditioner and specifies options that apply specifically to that type of preconditioner. See Section 3.3 for a description of the options available. The `FULL TANGENT PRECONDITIONER` command block cannot be used with the `PRECONDITIONER` command line in a `CG` command block. The user must select only one type of preconditioner for an analysis.

3.2.3 Line Search Command

```
LINE SEARCH ACTUAL|TANGENT [DURING <string list>period_names]
LINE SEARCH SECANT [<real>scale_factor]
```

During each CG iteration, after the search direction is computed, a line search is used to find an optimal scaling factor that when applied to the search vector will result in a minimized residual. The line search algorithm is controlled with the `LINE SEARCH` command line. Three line search types are available: `ACTUAL`, `SECANT`, and `TANGENT`. The `SECANT` option is the default, and is used if the `LINE SEARCH` command line is not present.

- The `ACTUAL` line search is a single-step quadratic line search that is equivalent to the corresponding JAS3D line search option. The line search requires an additional evaluation for each iteration of the element internal forces and thus the material response. For a given search direction s, residual r, and velocity v, the step length α is computed as

$$\alpha = -\frac{s^T r(v)}{s^T (F_{int}(v + s) - F_{int}(v))}, \tag{3.5}$$

  where the residual is computed based on the internal force F_{int} and on the external force F_{ext} as $r = F_{int}(v) - F_{ext}(v)$.

- The `SECANT` line search is a slight variation of the `ACTUAL` line search described above. Like the `ACTUAL` line search, this line search requires an additional internal force evaluation for each iteration. The step length α is computed as

$$\alpha = -\frac{s^T r(v)}{s^T (r(v + s) - r(v))}. \tag{3.6}$$

  The optional `scale_factor` can be used with the secant line search to scale the the search vector s in the equation above. If `scale_factor` is omitted, no scaling is done. If `scale_factor` is provided on the command line, s is scaled by the product of `scale_factor` and the dimension of the smallest element. Setting the scale factor to a small value can be helpful to avoid inverting elements during the evaluation of $r(v + s)$. The `SECANT` line search is recommended for problems in which the external force is highly nonlinear (e.g., a beam-bending problem with a pressure distribution).

- The `TANGENT` line search avoids the additional internal force calculation that is inherent in the other line search types by using the tangent modulus computed in the last call to the material subroutine for each element. The step length α is computed as

$$\alpha = -\frac{s^T r(v)}{s^T K s}, \tag{3.7}$$

  where K is the tangent stiffness.

The optional `scale_factor` can be used with all three line search types. This parameter is used to scale the search direction s. If `scale_factor` is omitted, no scaling is done. If `scale_factor` is provided on the command line, s is scaled by the product of `scale_factor` and the dimension of the smallest element. Supplying a small value for `scale_factor` can help to avoid element inversion during the line search.

3.2.4 Diagnostic Output Commands

```
ITERATION PRINT = <integer>iter_print
  [DURING <string list>period_names]
ITERATION PLOT = <integer>iter_plot
  [DURING <string list>period_names]
ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
```

The command lines listed above can be used to control the output of diagnostic information from the CG solver. The `ITERATION PRINT` and `ITERATION PLOT` command lines can be appended with the `DURING` specification, as discussed in Section 3.2.1.

The `ITERATION PRINT` command line controls the frequency at which the convergence information line is printed to the log file. The default value of `iter_print` is 25 if a nodal preconditioner is used, and it is 1 if a full tangent preconditioner is used.

The `ITERATION PLOT` command line allows plots of the current state of the model to be written to the output database during the CG iterations. The value supplied in `iter_plot` specifies the frequency of such plots. The default behavior is not to produce such plots. The plots are generated with a fictitious time step. Every time such a plot is generated, a message is sent to the log file. This feature is useful for debugging, but it should be used with care. Note that the `ITERATION PLOT` command line can be appended with the `DURING` specification, as discussed in Section 3.2.1.

The output produced by iteration plots is sent by default to every active results output file, as specified by a `RESULTS OUTPUT` command block. The `ITERATION PLOT OUTPUT BLOCKS` command line can optionally be used to supply a list of output block names to which iteration plots should be written. Each of the output block names listed in `plot_blocks`, must match the name of a `RESULTS OUTPUT` command block (see Section 8.2.1). The `ITERATION PLOT OUTPUT BLOCKS` command line can be useful for directing iteration plots to separate output files so that these plots are separated from the converged solution output. It can also be used to send output plots for different levels of the multilevel solver to different files.

3.2.5 CG Algorithm Commands

```
ITERATION RESET = <integer>iter_reset(10000)
  [DURING <string list>period_names]
ORTHOGONALITY MEASURE FOR RESET = <real>ortho_reset(0.5)
  [DURING <string list>period_names]
RESET LIMITS <integer>iter_start <integer>iter_reset
  <real>reset_growth <real>reset_orthogonality
  [DURING <string list>period_names]
BETA METHOD = FletcherReeves|PolakRibiere|
  PolakRibierePlus(FletcherReeves)
  [DURING <string list>period_names]
```

The behavior of the CG algorithm can be changed using the command lines listed above. The CG algorithm orthogonalizes the current search direction and the previous search direction at each iteration. During an iteration where a reset occurs, the orthogonalization is skipped, resulting in a search in the steepest descent direction. Note that all of these command lines can be appended with the `DURING` specification, as discussed in Section 3.2.1.

The `ITERATION RESET` command line specifies that the CG algorithm be reset after every `iter_reset` iterations. The default is to reset the algorithm after every 10,000 iterations.

For the CG algorithm to work well, each new search direction must be sufficiently orthogonal to the previous search directions. If the current search direction has a significant component in one of the previous search directions, error can be introduced that will not be corrected in future iterations. At every iteration, the relative lack of orthogonality between the current gradient direction and the previous search direction is computed. The `ORTHOGONALITY MEASURE FOR RESET` command line specifies that a reset will occur if the lack of orthogonality exceeds the value of `ortho_reset`. The default value of 0.5 results in few resets. Setting `ortho_reset` to a value as tight as 0.01 can result in many resets and potentially improved convergence on some problems.

The `RESET LIMITS` command line is also used to control CG resets. This command line is provided to achieve compatibility with JAS3D, and it behaves exactly the same as its JAS3D counterpart. The command line, however, should not be used when either the `ITERATION RESET` command line or the `ORTHOGONALITY MEASURE FOR RESET` command line is included in the `CG` command block. Up to four parameters can optionally be specified in the `RESET LIMITS` command line. The first, `iter_start`, sets the number of iterations to wait before looking for a minimum residual. The second, `iter_reset`, sets the number of iterations to allow between finding a minimum and restarting the iterative algorithm. The third, `reset_growth`, sets the amount of growth in the residual norm that would indicate divergence and thus trigger a reset. Finally, `reset_orthogonality` sets the relative lack of orthogonality between the current gradient direction and the previous search direction that would trigger a reset.

The `BETA METHOD` command line allows different formulas to be used in computing the β scalar in the CG algorithm. If set to `FletcherReeves`, the β scalar is computed as

$$\beta_k = \frac{r_k^T g_k}{r_{k-1}^T g_{k-1}}. \tag{3.8}$$

If set to `PolakRibiere` (the default), β is computed as

$$\beta_k = \frac{r_k^T (g_k - g_{k-1})}{r_{k-1}^T g_{k-1}}. \tag{3.9}$$

With the `PolakRibierePlus` option, β is computed as

$$\beta_k = \max(\beta_k^{PR}, 0). \tag{3.10}$$

In the above equations, r_k is the current iteration's residual vector, r_{k-1} is the previous iteration's residual vector, g_k is the current iteration's gradient vector, and g_{k-1} is the previous iteration's gradient vector. The `PolakRibierePlus` formula forces beta to be positive. A steepest descent direction is taken when beta becomes negative.

3.3 Full Tangent Preconditioner

In addition to the nodal preconditioners, Adagio provides the capability to use a full tangent preconditioner in the CG algorithm. The full tangent preconditioner employs a scalable parallel linear solver to solve for the gradient direction using the full tangent stiffness matrix. Although the full tangent preconditioner is significantly more costly per CG iteration than are the nodal preconditioners, it can solve problems in a very small number of iterations. Full tangent preconditioners are especially effective in solving poorly conditioned problems that are often very difficult to solve using the CG algorithm with nodal preconditioners, such as those problems that involve the bending response of long, slender members.

Using a full tangent preconditioner requires that two command blocks be added to the input file. First, the `FULL TANGENT PRECONDITIONER` command block, described in this section, must be added to the `CG` command block. The `FULL TANGENT PRECONDITIONER` command block is used instead of the `PRECONDITIONER` command line in the `CG` command block. Second, a command block for a linear solver must be defined in the SIERRA scope, as described in Section 3.4. That linear solver command block must be referenced from within the `FULL TANGENT PRECONDITIONER` command block, instructing the CG algorithm to use the specified linear solver. The command block to enable the full tangent preconditioner is as follows:

Known Issue: Deactivation of element blocks (see Section 5.1.5.8) does not currently work in conjunction with the full tangent preconditioner in Adagio. To use this capability, one of the nodal preconditioners must be used.

```
BEGIN FULL TANGENT PRECONDITIONER
  #
  # solver selection commands
  LINEAR SOLVER = <string>linear_solver_name
  NODAL PRECONDITIONER METHOD = ELASTIC|PROBE|DIAGONAL
    (ELASTIC)
  #
  # tangent matrix formation commands
  PROBE FACTOR = <real>probe_factor(1.0e-6)
  BALANCE PROBE = <integer>balance_probe(1)
  CONSTRAINT ENFORCEMENT = SOLVER|PENALTY(PENALTY)
  PENALTY FACTOR = <real>penalty_factor(100.0)
  SHELL DRILLING STIFFNESS =
    <real>shell_drill_stiff(1.0e-4 for quasistatics)
  TANGENT DIAGONAL SCALE = <real>tangent_diag_scale(0.0)
  #
  # reset and iteration commands
  MAXIMUM RESETS FOR MODELPROBLEM = <integer>max_mp_resets
    (100000) [DURING <string list>period_names]
  MAXIMUM RESETS FOR LOADSTEP = <integer>max_ls_resets
```

```
      (100000) [DURING <string list>period_names]
    MAXIMUM ITERATIONS FOR MODELPROBLEM
      = <integer>max_mp_iter(100000)
      [DURING <string list>period_names]
    MAXIMUM ITERATIONS FOR LOADSTEP = <integer>max_ls_iter
      (100000) [DURING <string list>period_names]
    ITERATION UPDATE = <integer>iter_update
      [DURING <string list>period_names]
    SMALL NUMBER OF ITERATIONS = <integer>small_num_iter
      [DURING <string list>period_names]
    NUMBER OF SMOOTHING ITERATIONS
      = <integer>num_smooth_iter(0)
      [DURING <string list>period_names]
    #
    # fall-back strategy commands
    STAGNATION THRESHOLD = <real>stagnation(1.0e-12)
      [DURING <string list>period_names]
    MINIMUM CONVERGENCE RATE = <real>min_conv_rate(1.0e-4)
      [DURING <string list>period_names]
    ADAPTIVE STRATEGY = SWITCH|UPDATE(SWITCH)
      [DURING <string list>period_names]
  END [FULL TANGENT PRECONDITIONER]
```

The FULL TANGENT PRECONDITIONER command block contains all command lines related to the interaction of Adagio with the linear solver. There are command lines to control selection of the linear solver, formation of the matrices passed into the linear solver, CG iteration strategies, updating strategies, and strategies for dealing with poor convergence. Reasonable defaults have been set for most problems. The only required command line in this command block is the LINEAR SOLVER command line, which is used to select the linear solver. The command lines in this command block are described in Sections 3.3.1 through 3.3.4.

3.3.1 Solver Selection Commands

```
LINEAR SOLVER = <string>linear_solver_name
NODAL PRECONDITIONER METHOD = ELASTIC|PROBE|DIAGONAL(ELASTIC)
```

The command lines listed above are used in selecting a linear solver for use with the full tangent preconditioner and in selecting a nodal preconditioner. The LINEAR SOLVER command line is the only required command line in the FULL TANGENT PRECONDITIONER command block. This command line specifies the name of the solver that will be used to compute the action of the preconditioner by solving the linear system. Linear solvers are defined in the SIERRA scope. Although several linear-solver packages are available for use as preconditioners in Adagio, we recommend that the FETI equation solver be used in production analyses. The FETI equation solver is actively maintained and tested by the Adagio development team. The FETI EQUATION SOLVER command block is documented in Section 3.4.

An arbitrary number of equation solver command blocks can be included in the input file to define various sets of solver options. Each equation solver command block must be given a name, which is referenced in the `linear_solver_name` parameter of the `LINEAR SOLVER` command line. This name instructs Adagio to use the specified linear solver as a preconditioner for the CG solver.

When Adagio uses a full tangent preconditioner, it also forms a nodal preconditioner. This nodal preconditioner is used in contact calculations, as a fall-back preconditioner if the full tangent preconditioner does not perform well, and optionally for performing a specified number of smoothing iterations prior to use of the full tangent preconditioner.

The nodal preconditioner is selected with the `NODAL PRECONDITIONER METHOD` command line. Three options are available: `ELASTIC`, `PROBE`, and `DIAGONAL`. The `ELASTIC` option is the default. These options have the same meaning as the corresponding options given for the nodal preconditioner in the `CG` command block, as documented in Section 3.2.2.

3.3.2 Matrix Formation Commands

```
PROBE FACTOR = <real>probe_factor(1.0e-6)
BALANCE PROBE = <integer>balance_probe(1)
CONSTRAINT ENFORCEMENT = SOLVER|PENALTY(PENALTY)
PENALTY FACTOR = <real>penalty_factor(100.0)
TANGENT DIAGONAL SCALE = <real>tangent_diag_scale(0.0)
SHELL DRILLING STIFFNESS =
  <real>shell_drill_stiff(1.0e-4 for quasistatics)
```

The command lines listed above can be used to optionally control the way the stiffness matrix is formed. This matrix is formed by assembling contributions from all active elements. These element stiffness matrices are formed by finite differencing.

The `PROBE FACTOR` command line controls the probing distance relative to element size The default value of `probe_factor` is 1.0e-6. Smaller values may give a better tangent approximation, but they may also generate round-off errors.

The `BALANCE PROBE` command line selects the type of finite-differencing scheme used to probe for the stiffness. The value of `balance_probe` can be set to 0, 1, or 2. Setting `balance_probe` to 0 causes the preconditioner to be formed by forward finite differencing. Probing occurs in only one direction, resulting in zero-order accuracy in the preconditioner. Setting `balance_probe` to 1, the default for the full tangent preconditioner, causes the preconditioner to be formed with central differencing. Probing at each element degree of freedom is performed in both positive and negative directions, leading to a first-order accurate estimate of the tangent stiffness. The value of `balance_probe` may also be set to 2, which allows the full tangent preconditioner to obtain central finite differencing with fourth-order error. This scheme requires twice the work as the central-differencing scheme but is more accurate. See Section 3.2.2 for a more detailed discussion of these options.

The `CONSTRAINT ENFORCEMENT` command line controls the way multipoint constraints (i.e. contact, rigid body, and periodic BC) are enforced in the linear system. Two options are available:

SOLVER and PENALTY. Specifying SOLVER results in the constraints being passed to the solver, thus allowing the solver to handle constraint enforcement internally. Specifying PENALTY results in the constraints being converted to penalty "elements" whose stiffness is contributed to the linear system before it is passed to the linear solver. If the PENALTY option is used, the penalty stiffness is controlled by the PENALTY FACTOR command line.

Although some of the available linear-solver packages support internal constraint enforcement, these packages often perform poorly in parallel. Thus, we recommend that CONSTRAINT ENFORCEMENT be set to PENALTY for all analyses. The errors caused by penalty compliance are corrected in each CG iteration, so the converged solution will not have errors that are due to penalty compliance. This strategy has proven to be very effective for dealing with constraints.

The PENALTY FACTOR command line is used to specify the penalty stiffness. At each constraint, the diagonal tangent stiffness of the slave degree of freedom is multiplied by the value of the penalty_factor parameter. The default value for this parameter is 100.0, which is reasonable for most problems. Assigning a high value (greater than about 1.0e+8) can cause poor conditioning of the linear system and also cause the linear solver to perform poorly. Assigning a low value (less than about 10) could require more CG iterations due to the correction of errors resulting from penalty compliance.

The TANGENT DIAGONAL SCALE command line is used to specify a tangent matrix diagonal scale factor, which is used to scale the diagonal entries of the tangent matrix. This is useful in some situations where a model does not have sufficient boundary conditions to remove all rigid body modes. This command has also been used as a last resort to get convergence when all else fails. The diagonal terms of the tangent matrix are scaled by (1.0 + tangent_diagonal_scale). The default value of tangent_diagonal_scale is 0.0, which means the tangent matrix diagonals are not scaled. A value of 0.1 scales up all tangent matrix diagonal terms by 10 percent.

The SHELL DRILLING STIFFNESS command line is used to add a small rotational stiffness in the normal direction of a shell since this degree of freedom does not exist in the local shell coordinate system. The value specified is multiplied by Young's modulus, the shell thickness, and the time step to get a stiffness which is added to the full tangent preconditioner stiffness matrix for this degree of freedom.

3.3.3 Reset and Iteration Commands

```
MAXIMUM RESETS FOR MODELPROBLEM = <integer>max_mp_resets
  (100000) [DURING <string list>period_names]
MAXIMUM RESETS FOR LOADSTEP = <integer>max_ls_resets
  (100000) [DURING <string list>period_names]
MAXIMUM ITERATIONS FOR MODELPROBLEM = <integer>max_mp_iter
  (100000) [DURING <string list>period_names]
MAXIMUM ITERATIONS FOR LOADSTEP = <integer>max_ls_iter
  (100000) [DURING <string list>period_names]
ITERATION UPDATE = <integer>iter_update
  [DURING <string list>period_names]
SMALL NUMBER OF ITERATIONS = <integer>small_num_iter(0)
```

```
    [DURING <string list>period_names]
  NUMBER OF SMOOTHING ITERATIONS = <integer>num_smooth_iter(0)
    [DURING <string list>period_names]
```

When a linear solver is used as a preconditioner, the efficiency of the solution can often be dramatically affected by the frequency at which the preconditioner is updated. The computational expense of updating the preconditioner, which involves forming and factorizing the stiffness matrix, is often much higher than the cost of applying the preconditioner, which involves only a back-solve, during a CG iteration. If the stiffness does not dramatically change from iteration to iteration, it is often much more efficient to re-use the preconditioner for a number of iterations, or even load steps, than it is to update it at every iteration.

The command lines listed above can be used to control when the preconditioner is updated. As Adagio allows for the nodal preconditioner to be used in place of the full tangent preconditioner in some situations, these command lines also control this behavior. Note that all of these command lines can be appended with the `DURING` specification, as discussed in Section 3.2.1.

The `MAXIMUM RESETS FOR MODELPROBLEM` command line limits the number of times that the preconditioner can be reset (updated) during a model problem. By default, there is no limit on the number of updates that can occur during a model problem. The `MAXIMUM RESETS FOR LOADSTEP` command line limits the updates that can occur during a load step. By default, there is also no limit on the number of updates that can occur during a load step.

The `MAXIMUM ITERATIONS FOR LOADSTEP` and `MAXIMUM ITERATIONS FOR MODELPROBLEM` command lines can be used to limit the number of CG iterations that will be performed using the full tangent preconditioner per model problem or load step, respectively. The values of the parameters `max_mp_iter` and `max_ls_iter` are both unlimited by default. If one of these iteration limits is reached, the CG solver will continue to iterate using the fall-back nodal preconditioner until either the CG solver iteration limit is reached or convergence is achieved. It can be useful to limit the iterations taken by the full tangent preconditioner. If the full tangent preconditioner is performing poorly, the nodal preconditioner may work better in some cases.

The `ITERATION UPDATE` command line controls the frequency at which the full tangent preconditioner is updated. By default, it is only updated at the first iteration of each model problem. If this command line is used, the preconditioner will be updated at the frequency specified by `iter_update`. If a model experiences highly nonlinear behavior over the course of a model problem, it may be useful to use this command line to cause more frequent preconditioner updates. Such problems may converge better when `iter_update` is set to about 10. It is important to realize that frequent updates may reduce the number of CG iterations but dramatically increase the computational cost.

It can often be efficient to update the preconditioner very infrequently. The `SMALL NUMBER OF ITERATIONS` command line is used to "freeze" the preconditioner, or prevent updates, until a model problem requires more iterations than the number specified in `small_num_iter`. The default value of `small_num_iter` is 0, meaning that the preconditioner is never re-used for the next model problem unless the `SMALL NUMBER OF ITERATIONS` command line is used. It is often beneficial to set `small_num_iter` to 10. This can often result in the re-use of the preconditioner over many load steps. When this command line is used, the CG solver often converges in a very small number of iterations immediately following a preconditioner update; however, over the course of several load steps, the iteration counts slowly increase because the preconditioner is out of date. At a certain point, the iteration count exceeds `small_num_iter`, and the preconditioner is reset. This solution strategy is very efficient because it allows re-use of the preconditioner while it is still relatively effective.

In some cases it is beneficial to perform a number of CG iterations using the nodal preconditioner prior to applying the full tangent preconditioner. The iterations with the nodal preconditioner are used as a "smoother" to put the model in a better state for the full tangent preconditioner. This behavior can be enabled using the `NUMBER OF SMOOTHING ITERATIONS` command line. By default, this feature is disabled. If this command line is present, the `num_smooth_iter` parameter specifies the number of smoothing iterations to be taken. This feature can also be used in cases where the nodal preconditioner may be effective enough to achieve convergence with relatively few iterations, but occasionally the nodal preconditioner is not effective. If `num_smooth_iter` is set to a relatively high number, many load steps can be solved without ever using the full tangent preconditioner, but this preconditioner is used on the more difficult steps that require more iterations.

Another use of the `NUMBER OF SMOOTHING ITERATIONS` command line is for switching between the nodal preconditioner and the full tangent preconditioner for certain time periods. This command line, as all the other command lines discussed above, can be set on a period-specific basis by appending it with the `DURING` keyword, followed by a list of periods. To use the nodal preconditioner on some periods, the user can set `num_smooth_iter` to a high number for those periods and leave the default value of 0 for the periods in which the full tangent preconditioner is desired.

3.3.4 Fall-Back Strategy Commands

```
STAGNATION THRESHOLD = <real>stagnation(1.0e-12)
MINIMUM CONVERGENCE RATE = <real>min_conv_rate(1.0e-4)
ADAPTIVE STRATEGY = SWITCH|UPDATE(SWITCH)
```

Occasionally, the full tangent preconditioner may stagnate, failing to significantly reduce the residual from one CG iteration to the next. At each iteration, Adagio checks for slow convergence, and attempts to remedy poor convergence if such is detected. The commands listed above can optionally be added to the `FULL TANGENT PRECONDITIONER` command block.

The `STAGNATION THRESHOLD` and `MINIMUM CONVERGENCE RATE` command lines are used to set the `stagnation` and `min_conv_rate` parameters, respectively. These parameters are used in a strategy that attempts to remedy slow convergence. The method for remedying slow convergence is controlled with the `ADAPTIVE STRATEGY` command line, which can have a value of `SWITCH` or `UPDATE`. The default strategy is `SWITCH`. The convergence rate is computed as the absolute value of the relative difference between residuals after successive CG iterations. In the `SWITCH` strategy, if the convergence rate is less than the value of `min_conv_rate` but greater than the value of `stagnation`, the CG solver will switch to using the fall-back nodal preconditioner. In the `UPDATE` strategy, the full tangent preconditioner would be updated instead. In either case, if the convergence rate is below the value of `stagnation`, the solver will switch to the nodal preconditioner.

3.4 FETI Equation Solver

FETI is a domain-decomposition-based parallel iterative linear solver that can be used as a full tangent preconditioner for Adagio's nonlinear CG solver [3, 4]. FETI uses a direct solver on each domain and iteratively solves for Lagrange multiplier fields at the domain boundaries. Under typical usage, the FETI domains correspond to the portions of the model owned by each processor. If a model is run on a single processor, FETI simply behaves as a direct solver. Because large models are typically run using many processors, FETI uses significantly lower computing resources than a direct solution of the whole problem would require because a large number of small direct solutions are performed in parallel.

Although a number of other linear solvers are available for use as full tangent preconditioners in Adagio's CG solver, it is recommended that FETI be used. FETI is actively maintained and tested by the Adagio development team; its effectiveness as a robust parallel solver has been demonstrated on a wide range of production analyses. The command block for the FETI equation solver is as follows:

```
BEGIN FETI EQUATION SOLVER <string>name
  #
  # convergence commands
  MAXIMUM ITERATIONS = <integer>max_iter(500)
  RESIDUAL NORM TOLERANCE = <real>resid_tol(1.0e-6)
  #
  # memory usage commands
  PARAM-STRING "precision" VALUE <string>"single"|"double"
    ("double")
  PRECONDITIONING METHOD = NONE|LUMPED|DIRICHLET(DIRICHLET)
  MAXIMUM ORTHOGONALIZATION = <integer>max_orthog(500)
  #
  # solver commands
  LOCAL SOLVER = SKYLINE|SPARSE|ITERATIVE(SPARSE)
  COARSE SOLVER = SKYLINE|SPARSE|ITERATIVE(SPARSE)
  NUM LOCAL SUBDOMAINS = <integer>num_local_subdomains
END [FETI EQUATION SOLVER <string>name]
```

The command lines used to control FETI all reside in the `FETI EQUATION SOLVER` command block, where `name` identifies the particular command block. This command block, as with all equation solver command blocks, must be placed in the SIERRA scope and must be referenced by `name` when it is used. Although a number of command lines are available to control the behavior of FETI, the default settings generally work well for the vast majority of problems. Thus, it is recommended that all default settings be used unless special behavior is desired because of the unique features of a specific model. The command lines in the `FETI EQUATION SOLVER` command block are described in Sections 3.4.1 through 3.4.3.

3.4.1 Convergence Commands

```
MAXIMUM ITERATIONS = <integer>max_iter(500)
RESIDUAL NORM TOLERANCE = <real>resid_tol(1.0e-6)
```

The command lines listed above provide controls on the convergence of the FETI iterative solver, and belong in the `FETI EQUATION SOLVER` command block.

The `MAXIMUM ITERATIONS` command line sets the maximum number of iterations allowed per FETI solution. The default value of the parameter `max_iter` is 500. A FETI solution occurs for every CG iteration in which FETI is used as a full tangent preconditioner. The `RESIDUAL NORM TOLERANCE` command line sets the convergence criterion for the FETI solver. The default value of the parameter `resid_tol` is 1.0e-6. If convergence is not reached before the iteration count exceeds `max_iter`, FETI will simply return the current gradient direction to the CG solver, which will continue iterating. The code will not exit with an error. The default settings for both of these command lines are reasonable for most models and typically should not be modified.

3.4.2 Memory Usage Commands

```
PARAM-STRING "precision" VALUE <string>"single"|"double"
  ("double")
PRECONDITIONING METHOD = NONE|LUMPED|DIRICHLET(DIRICHLET)
MAXIMUM ORTHOGONALIZATION = <integer>max_orthog(500)
```

The command lines listed above can be placed in the `FETI EQUATION SOLVER` command block to enable optional memory-saving features of the FETI solver. All these features will adversely affect the performance of this solver to some degree, but they can be useful if the memory requirements of a model exceed the capacity of the machine on which the model is run. Before using these features, it is important to consider that on a distributed memory cluster, spreading the model out over more processors can reduce the memory requirements on each processor. For this reason, it is often better to use more processors rather than use the options described here.

FETI has the option of using either single or double precision for storage of internal variables. The default behavior is to use double precision, and this is typically recommended. To select single precision, the user would specify the command line as follows: `PARAM-STRING "precision" VALUE "single"`. Using single-precision variables within FETI can dramatically reduce the memory requirements. This may, however, slightly degrade the performance of the solver, requiring more iterations within FETI or more CG iterations. Using single precision in FETI does not affect the Adagio data structures, which are always double precision, and therefore does not adversely affect solution accuracy.

The `PRECONDITIONING METHOD` command line selects the preconditioning method that is used internally within FETI. The default option, `DIRICHLET`, typically results in the best convergence rate. The `LUMPED` option uses less memory, but it usually results in more iterations within FETI. This option should only be used if there are constraints on memory usage. The `NONE` option uses no preconditioner and is included only for completeness. This option is not of practical interest.

Like Adagio's CG solver, the FETI equation solver stores a set of search directions to ensure that the search direction used in each iteration is orthogonal to previous search directions. The number of search directions stored is controllable with the `MAXIMUM ORTHOGONALIZATION` command line. The default value of 500 for `max_orthog` provides optimal convergence. Setting `max_orthog` to a lower number can decrease memory usage but, as with the other options discussed above, may require more iterations for convergence.

3.4.3 Solver Commands

```
LOCAL SOLVER = SKYLINE|SPARSE|ITERATIVE(SPARSE)
COARSE SOLVER = SKYLINE|SPARSE|ITERATIVE(SPARSE)
NUM LOCAL SUBDOMAINS = <integer>num_local_subdomains
```

The command lines listed above control the type of solver used by FETI for solving the linear system that arises from the coarse grid and for the local linear system on the actual solution mesh. The default behavior is for FETI to use a sparse direct solver for both of these systems, an approach that works well for most problems. The `LOCAL SOLVER` command line is used to select the solver for the local subdomains. Similarly, the solver for the coarse grid can be selected with the `COARSE SOLVER` command lines. Both of these command lines allow the same three options: `SPARSE`, `SKYLINE`, and `ITERATIVE`, where `SPARSE` is the default.

- The `SPARSE` option uses a sparse matrix storage direct solver. The sparse matrix is factored into an $A = LDL^T$ decomposition, but the code implementation takes advantage of equation orderings to reduce matrix fill-in. The default equation ordering is done by calling METIS's sparse matrix ordering algorithm. This solver is recommended for both speed and memory efficiency.

- The `SKYLINE` option uses a skyline (profile) matrix storage direct solver. The skyline matrix is factored into an $A = LDL^T$ decomposition. This method is very robust, detects rigid-body modes effectively, and includes the reverse Cuthill-McKee (RCM) and Sloan equation orderings. This solver uses much more memory than does the sparse solver.

- The `ITERATIVE` option implements a multiple domain FETI algorithm for the local solver, where the number of domains is either computed based on some heuristics or is input by the user. This option is recommended in special circumstances where memory becomes an issue with the sparse direct solver.

The `NUM LOCAL SUBDOMAINS` command line sets the number of local subdomains for the `ITERATIVE` local solver. This command line is not used for the other local solvers. As the number of local subdomains increases, the size of the local subdomains decreases, thus reducing the memory requirements and the time it takes for the local matrix factorizations. As the number of subdomains increases, the size of the coarse grid increases, thus requiring increased factorization time and memory. A good rule of thumb is to set the number of local subdomains to the total number of elements in the mesh divided by 400. For a serial run, if the `NUM LOCAL SUBDOMAINS` command line is not specified, FETI sets the number of domains to the greater of 2 or the number of elements divided by 400. For parallel runs, FETI uses the same number of domains used by Adagio.

3.5 Control Contact

The multilevel solution control scheme used for contact is referred to as control contact. After a set of nodes in contact (a constraint set) is established, a model problem is solved using the CG solver with this constraint set held constant. For frictional contact, slave nodes are fixed to master faces during the CG iterations. For frictionless contact, the slave nodes are fixed in the normal direction but are allowed to slide during the CG iterations. After model problem convergence, the constraint set is updated to reflect the changing contact conditions. This update gives rise to a force imbalance and to another model problem.

Changing or updating the constraint set is referred to as a contact update. Multiple contact updates are typically required before equilibrium is achieved. The contact update consists of a search, a gap removal, an equilibrium query, and a slip calculation if equilibrium is not satisfied.

New constraints are detected in the search phase based on the current deformed configuration of the model. If a node is found to penetrate a master surface, the node is added to the set of constraints. The slave node is moved to the master surface by moving it along the push-back vector. The penetration may be removed at once (the default behavior), or it may be incrementally removed over a number of model problems.

The gap removal phase involves creating or destroying constraints and calculating push-back vectors for slave nodes. For types of surface mechanics in which contacting surfaces are free to separate, a slave node constraint continues to exist as long as there is a compressive force between the slave node and the master surface. The constraint changes during the gap removal phase to reflect changes in the shape and orientation of the master surface. Constraints are destroyed when a tensile force exceeding a known tolerance exists at the master/slave interface.

During the slip portion of the contact update, a residual force is calculated at each node and resolved into normal and tangential components. This force reflects changes in residual due to gap removal that occurred earlier in the update. For frictional contact, the friction coefficient is used to determine the tangential load capacity. If the tangential loads exceed this capacity, nodes will slip.

The procedure for control contact is illustrated in the following sequence of figures. It is assumed that control contact is on level 1, so the model problems are directly solved by the core solver. In this example (Figure 3.2), the model problem has three constraints that are enforced during iterations of the core solver. Nothing prevents other nodes from penetrating surfaces during solution of the model problem, and in this example, one node does penetrate.

Once the model problem has converged, a contact update is performed. In the gap removal portion of the update (Figure 3.3), one of the constraints accumulated a tensile force large enough that surfaces should separate, thus eliminating this constraint. During the search, one node penetrated the surface, resulting in a new constraint for this node. A push-back vector is calculated for this node to remove the penetration.

In the second phase of the contact update, slip is allowed to occur along the master/slave interface. After the gaps are removed, the external and internal force vectors are recomputed, and forces are partitioned along normal and tangential directions of the master surface. This process is illustrated in Figure 3.4.

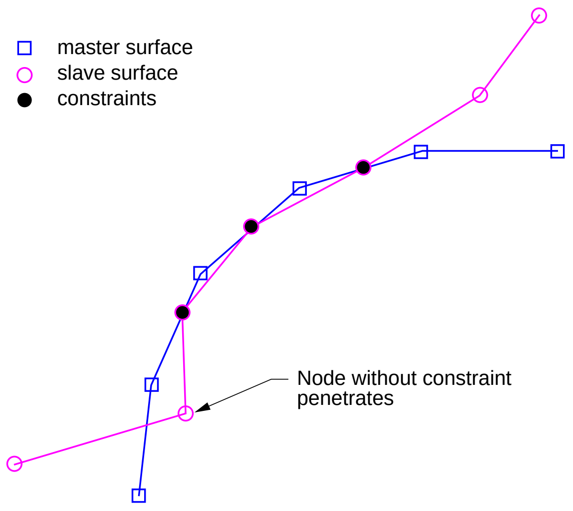


Figure 3.2: Contact configuration at the beginning of the contact update.

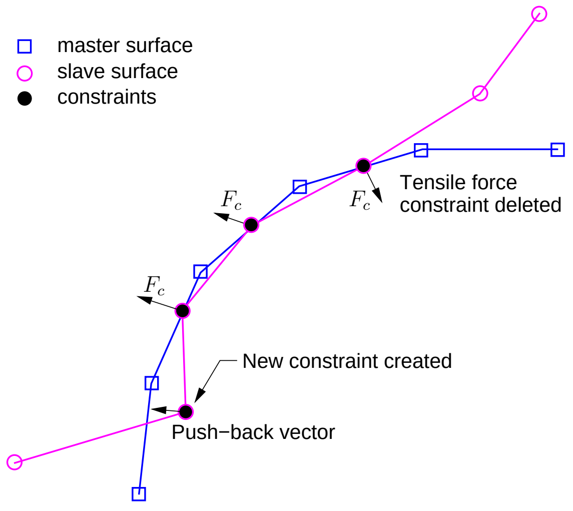


Figure 3.3: Contact gap removal (after contact search).

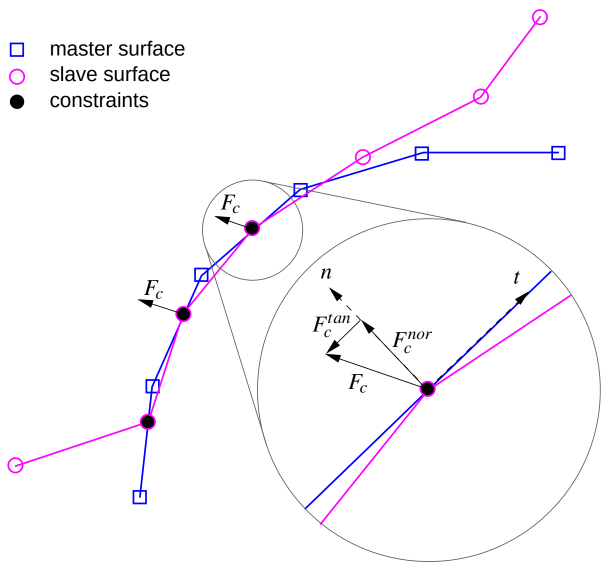


Figure 3.4: Contact slip calculations.

The command block for control contact is as follows:

```
BEGIN CONTROL CONTACT
  #
  # convergence commands
  TARGET RESIDUAL = <real>target_resid
    [DURING <string list>period_names]
  TARGET RELATIVE RESIDUAL = <real>target_rel_resid
    [DURING <string list>period_names]
  TARGET RELATIVE CONTACT RESIDUAL = <real>target_rel_cont_resid
    [DURING <string list>period_names]
  ACCEPTABLE RESIDUAL = <real>accept_resid
    [DURING <string list>period_names]
  ACCEPTABLE RELATIVE RESIDUAL = <real>accept_rel_resid
    [DURING <string list>period_names]
  ACCEPTABLE RELATIVE CONTACT RESIDUAL =
    <real>accept_rel_cont_resid [DURING <string list>period_names]
  REFERENCE = EXTERNAL|INTERNAL|RESIDUAL(EXTERNAL)
    [DURING <string list>period_names]
  MINIMUM ITERATIONS  = <integer>min_iter(0)
    [DURING <string list>period_names]
  MAXIMUM ITERATIONS  = <integer>max_iter
    [DURING <string list>period_names]
  #
  # level selection command
  LEVEL = <integer>contact_level(1)
  #
  # diagnostic output commands
```

```
    ITERATION PLOT = <integer>iter_plot
      [DURING <string list>period_names]
    ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
  END [CONTROL CONTACT]
```

To enable control contact, a CONTROL CONTACT command block must exist in the SOLVER command block. The line commands within the CONTROL CONTACT command block are used to control convergence during the contact updates, select the level for the contact control within the multilevel solver, and output diagnostic information. These commands are described in detail in Section 3.5.1 through Section 3.5.3.

3.5.1 Convergence Commands

The command lines listed in this section are placed in the CONTROL CONTACT command block to control convergence criteria for contact within the multilevel solver.

```
TARGET RESIDUAL = <real>target_resid
  [DURING <string list>period_names]
TARGET RELATIVE RESIDUAL = <real>target_rel_resid
  [DURING <string list>period_names]
TARGET RELATIVE CONTACT RESIDUAL = <real>target_rel_cont_resid
  [DURING <string list>period_names]
ACCEPTABLE RESIDUAL = <real>accept_resid
  [DURING <string list>period_names]
ACCEPTABLE RELATIVE RESIDUAL = <real>accept_rel_resid
  [DURING <string list>period_names]
ACCEPTABLE RELATIVE CONTACT RESIDUAL =
  <real>accept_rel_cont_resid [DURING <string list>period_names]
REFERENCE = EXTERNAL|INTERNAL|RESIDUAL(EXTERNAL)
  [DURING <string list>period_names]
MINIMUM ITERATIONS = <integer>min_iter(0)
  [DURING <string list>period_names]
MAXIMUM ITERATIONS = <integer>max_iter
  [DURING <string list>period_names]
```

Solver convergence is monitored for control contact in much the same way as it is done for the core CG solver. The command lines listed above are used for specifying the convergence criteria used in the level 1 updates. With the exception of two additional command lines used for controlling the residual due to contact, these command lines are the same as those used for the core CG solver (described in Section 3.2.1) and have the same meaning. Here, however, these command lines are applied to the contact control. Note that all of these command lines can be appended with the DURING specification, as discussed in Section 3.2.1.

Contact convergence is measured by computing the L^2 norm of the residual, and comparing that residual norm with target convergence criteria specified by the user. There are two residual norms used as convergence metrics for contact: the norm of the residual on all nodes, and the norm of the residual on the nodes currently in contact. In the discussion here, the residual norm for all nodes is referred to as the residual, while the residual norm for the contact nodes is referred to as the contact residual. Convergence can be monitored directly in terms of the residual norm, the relative residual, or the relative contact residual. The relative residual and relative contact residual are computed by dividing the residual and contact residual, respectively, by a reference quantity that is indicative of the current loading conditions on a model. Either the residual, the relative residual, or both can be used for checking convergence at the same time. In addition, the contact relative residual must be below specified convergence limits.

The `TARGET RESIDUAL` command line specifies the target convergence criterion in terms of the actual residual norm. The `TARGET RELATIVE RESIDUAL` command line specifies a convergence criterion in terms of the relative residual. If both command lines are specified, the multilevel solver will accept the contact solution as converged if either the target residual or the target relative residual is below the specified values. The `TARGET RELATIVE CONTACT RESIDUAL` command line specifies the target convergence criterion for the relative contact residual. This criterion must be satisfied in addition to the target residual or relative residual for convergence. The default value for the target relative contact residual is the target relative residual.

The multilevel solver also allows for acceptable convergence criteria to be input for contact convergence. If the solution has not converged to the specified targets before the maximum number of iterations is reached, the residual is checked against the acceptable convergence criteria. These criteria are specified via the `ACCEPTABLE RESIDUAL`, `ACCEPTABLE RELATIVE RESIDUAL`, and `ACCEPTABLE RELATIVE CONTACT RESIDUAL` command lines. The concepts of residual, relative residual, and relative contact residual are the same as those used for the target limits. If the solution has not met the target criteria but has met the acceptable criteria, the solution is allowed to proceed. The defaults for each of these acceptable criteria are 10 times the corresponding target criteria.

If relative residuals are given in the convergence criteria, the `REFERENCE` command line can be used to select the method that will be used to compute the reference load. This command line has three options: `EXTERNAL`, `INTERNAL`, and `RESIDUAL`. The `EXTERNAL` option, which is the default, is computed by taking the L^2 norm of the current external load. The `INTERNAL` option uses the norm of the internal forces as the reference load. Finally, the `RESIDUAL` option denotes that the initial residual should be used as the reference quantity.

The `MAXIMUM ITERATIONS` and `MINIMUM ITERATIONS` command lines specify the maximum and minimum number of contact updates, respectively. The default minimum number of iterations, `min_iter`, is 0. If a number greater than 0 is specified, the multilevel solver will update contact at least that many times, regardless of whether the convergence criteria have been met.

3.5.2 Level Selection Command

```
    LEVEL = <integer>contact_level(1)
```

The `LEVEL` command line in the `CONTROL CONTACT` command block is used to specify the level in the multilevel solver at which contact is controlled. The `contact_level` parameter can have a value of either 1 or 2, with 1 being the default.

This command is used when multiple controls (i.e. control contact and control stiffness) are active in the multilevel solver. It is permissible for multiple controls to exist at a given level. The default behavior is for all controls to be at level 1. To have a control at level 2, another control must be active at level 1 because a level in the multilevel solver cannot be skipped. The level of other controls is specified using the `LEVEL` command line in the command blocks associated with those controls.

3.5.3 Diagnostic Output Commands

```
ITERATION PLOT = <integer>iter_plot
  [DURING <string list>period_names]
ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
```

The command lines listed above can be used to control the output of diagnostic information from the contact control within the multilevel solver. The `ITERATION PLOT` command line allows plots of the current state of the model to be written to the output database during the contact updates. The value supplied in `iter_plot` specifies the frequency of such plots. The default behavior is not to produce such plots. The plots are generated with a fictitious time step. Every time such a plot is generated, a message is sent to the log file. This feature is useful for debugging, but it should be used with care. Note that the `ITERATION PLOT` command line can be appended with the `DURING` specification, as discussed in Section 3.2.1.

The output produced by iteration plots is sent by default to every active results output file, as specified by a `RESULTS OUTPUT` command block. The `ITERATION PLOT OUTPUT BLOCKS` command line can optionally be used to supply a list of output block names to which iteration plots should be written. Each of the output block names listed in `plot_blocks`, must match the name of a `RESULTS OUTPUT` command block (see Section 8.2.1). The `ITERATION PLOT OUTPUT BLOCKS` command line can be useful for directing iteration plots to separate output files so that these plots are separated from the converged solution output. It can also be used to send output plots for different levels of the multilevel solver to different files.

3.6 Control Stiffness

Control stiffness is an augmented Lagrange iterative solution strategy for solving models that have widely varying stiffnesses in various modes of material response. These differences in stiffness can be between individual materials in a problem or even between the different responses of a single material model. For instance, nearly incompressible materials have a bulk/volumetric response that is much stiffer than the corresponding shear/deviatoric response. A similar case arises for materials that are orthotropic or anisotropic in nature and exhibit widely varying stiffnesses in the principal material directions.

Using control stiffness allows part of the constitutive response of a material to be softened or stiffened to give a tangent response that results in model problems that can be solved more easily by the core solver. This is accomplished by scaling a chosen stress increment up or down in a given model problem and adding it to a saved reference stress to yield a scaled stress that is used for equilibrium evaluations. The reference stress accumulates the model problem stress increments so that the true material response for a time step is obtained after solving a sequence of model problems.

For the case of nearly incompressible materials, a sequence of model problems can be constructed in which the bulk behavior is softened (scaled down) and/or the shear behavior is stiffened (scaled up). For the case of anisotropic materials, the anisotropic part of the material response is scaled down to create a model problem where the material has a more nearly isotropic response. In addition, the isotropic part of the response of some of these orthotropic materials can have its bulk stress increments scaled down and/or its shear/deviatoric stress increments scaled up. Finally, if an isotropic material is much stiffer than its neighbors, all of that material's stress increments can be scaled down.

Scaling a stress increment up or down is akin to using scaled moduli in a model problem. It should be noted that this is only precisely true for the case in which the true material response is linear. For the case of nonlinear materials, the effective scaled moduli are based solely on the difference between the unscaled stresses and the reference stresses. As a result, these effective scaled moduli vary over the course of model problem and core solver iterations. Nevertheless, the control stiffness algorithms are completely general and do not rely on linearity in the true material response.

The degree to which components of a material's response are softened or stiffened to achieve overall minimum computational time is problem dependent and involves a trade-off between the difficulty of solving a model problem and the number of model problems that must be used to achieve the final solution.

For example, consider a nearly incompressible material, such as rubber, for which the model problem solution may be optimized by scaling the bulk and shear behaviors such that the effective tangent response has a bulk modulus that is equal to twice its shear modulus. Adjusting the ratio between the bulk and shear moduli in this manner may minimize the core solver iterations for a given model problem. However, because of the inherently large difference between these two moduli in this material, significant bulk and/or shear scaling would be required to achieve these optimal scaled moduli. Such severe scaling could result in a large number of model problems to achieve the true material response. Choosing less severe scalings (scalings closer to 1), on the other hand, could result in more core solver iterations in each model problem, but fewer model problems to arrive at the true material response.

Experience has shown that once nearly optimal scaling values have been determined for a class of problems, these can be applied successfully to families of problems with similar characteristics.

A partial explanation of scaling the material response will be given for a simple scalar relation between stress and strain. For example, the true pressure-volume relation for a nearly incompressible material is a scalar relation between pressure and volumetric strain. Appropriate generalizations of the softening and stiffening algorithms to tensor relations are easily achieved by applying the same algorithms for all of the stress quantities individually. Only the calculation of the appropriate error measures is modified to deal with the tensorial nature of the stress-strain response that is being scaled.

There are two strain and three stress quantities of interest in the control stiffness algorithm. The stress quantities are the unscaled stress calculated by the unmodified material model, the scaled stress used for equilibrium evaluation, and a reference stress used to build up stresses to achieve convergence such that the scaled model problem stress is nearly equal to the unscaled stress computed from the material model. The strain quantities of interest are the true strain computed from the kinematics and the strain necessary to achieve the scaled stress using the unmodified material model.

In the equations that follow, the subscripts refer to the particular model problem being solved. Let us begin with the true material response in model problem I given by

$$\sigma_I = f(e_I), \tag{3.11}$$

where σ_I is the true stress corresponding to the kinematic strain e_I determined from the nodal displacements, and $f(\cdot)$ is the function representing the constitutive response. Although the true material response has been written for the hyperelastic case employing total stress and strain quantities, the control stiffness algorithms that are being presented can be equally applied to hypoelastic models where the stress rate is written in terms of the strain rate. The important point here is that the unscaled constitutive equation is used to calculate the unscaled/true stress response. It is not necessary for that relation to be linear or for a particular formulation to be used in terms of total strains or incremental strain rates. However, it should be noted that control stiffness is set up in Adagio to work only with certain material models.

The scaled stress response to use for equilibrium evaluations in a model problem is determined as follows:

$$s_I = r_I + \lambda \left(\sigma_I - r_I \right), \tag{3.12}$$

where s_I is the scaled stress, r_I is the reference stress, and λ is the scaling factor. For softening behavior, λ is less than 1, while for stiffening behavior, it is greater than 1. As noted previously, this scaled stress is what the core solver uses for equilibrium evaluation in a given model problem. It should be noted that the core solver typically requires a number of iterations to find the scaled stress that results in equilibrium. The softening and stiffening control stiffness algorithms differ only in terms of what is used for the reference stress. The reference stress in model problem I is given by the following:

$$r_I = \begin{cases} s_{I-1} & \lambda < 1 \quad \text{(softening)} \\ \sigma_{I-1} & \lambda > 1 \quad \text{(stiffening)} \end{cases} \tag{3.13}$$

Figures 3.5 and 3.6 provide a graphical presentation of softening the material response, and Figures 3.7 and 3.8 offer a similar presentation of stiffening the material response.

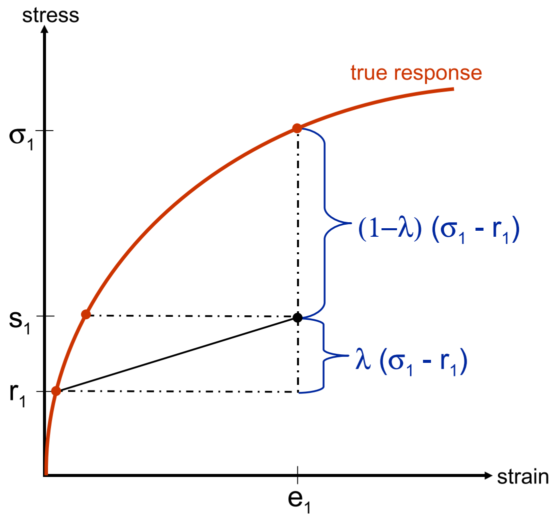


Figure 3.5: Control stiffness softening behavior in the first model problem of a time step.

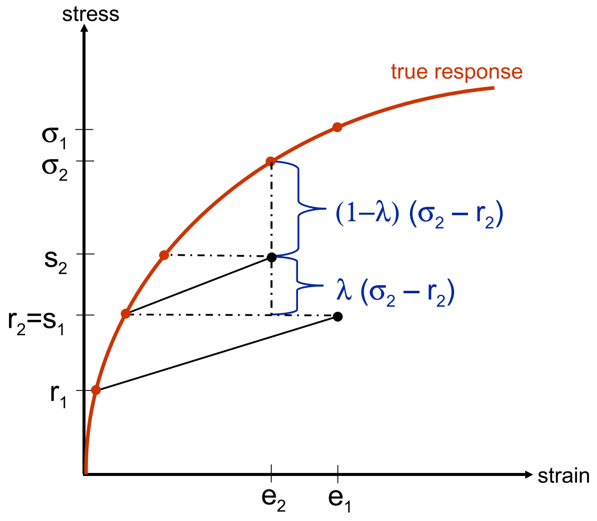


Figure 3.6: Control stiffness softening behavior in the second model problem of a time step.

When equilibrium is achieved in a model problem, it is necessary to do two things. First, an error measure related to the difference between the unscaled and scaled stress must be determined. Second, the reference stress used in the scaled stress calculations must be updated. The reference

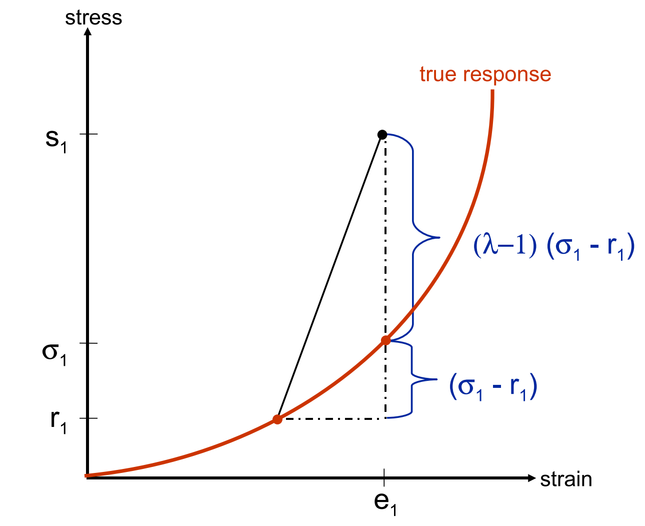


Figure 3.7: Control stiffness stiffening behavior in the first model problem of a time step.

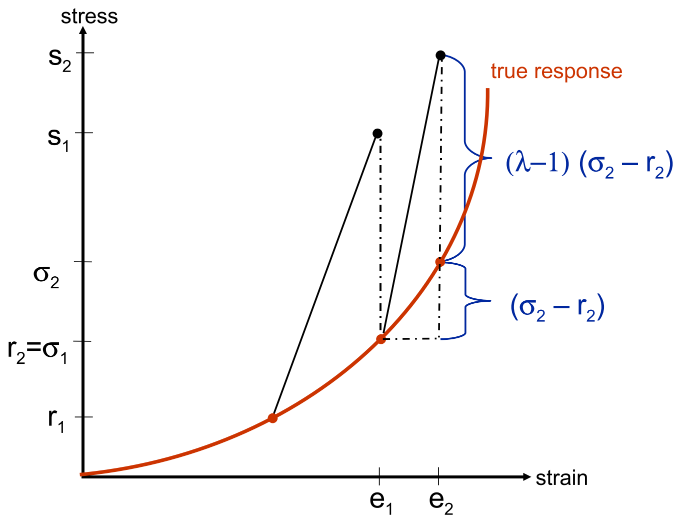


Figure 3.8: Control stiffness stiffening behavior in the second model problem of a time step.

stresses are updated according to Equation (3.13). That is, for the case of softening the material response, the reference stress is updated for the next model problem to be the scaled stress that achieved equilibrium in the current model problem. On the other hand, for the case of stiffening the material response, the reference stress is updated for the next model problem to be the true stress at the end of the current model problem.

The control stiffness updating process is converged when the scaled and unscaled stresses differ by an acceptably small amount. Several different error measures can be used to determine the degree to which the scaled and unscaled stresses differ. These can be categorized into two types: those that consider stress errors, and those that consider strain errors.

The stress convergence measures consider consider the difference between the scaled stress and the unscaled stress. The first of these stress error measures involves taking the L^2 norm of the stress difference:

$$error_I = \|s_I - \sigma_I\|_2 . \tag{3.14}$$

A second stress error measure involves normalizing that quantity:

$$error_I = \frac{\|s_I - \sigma_I\|_2}{\|s_I\|_2}. \tag{3.15}$$

The strain error convergence measure is based on a strain quantity E_I that is computed as the difference between the scaled stress and the unscaled stress and dividing that value by a modulus:

$$E_I = \frac{|s_I - \sigma_I|}{M}, \tag{3.16}$$

where M is an appropriate material modulus that is used to represent the unscaled constitutive response. For the case where the true material response is linear and the scaled stress is computed by stiffening the material response, E_I as defined by Equation (3.16) corresponds exactly to the difference between the kinematic strain e_I and the strain ϵ_I needed in the unscaled constitutive equation to give the scaled stress:

$$\epsilon_I = f^{-1}(s_I) = s_I/M, \tag{3.17}$$

where M would be the real material modulus. Nevertheless, Equation (3.16) is not dependent upon linear behavior, and it can be used in both softening and stiffening types of control stiffness scaling. The strain error quantity used to check convergence is computed by taking a global maximum (an L^∞ norm) as follows:

$$error_I = \left\| \frac{E_I}{E_{ref}} \right\|_\infty , \tag{3.18}$$

where E_{ref} is a reference strain specified in the input parameters for the material whose response is being scaled.

In summary, the user may specify error tolerances to determine convergence of the sequence of model problems solved in the control stiffness algorithm by using one of the error measures defined by Equations (3.14), (3.15), or (3.18). If the user so chooses, error tolerances corresponding to both Equations (3.14) and (3.15) can be specified, and convergence is considered whenever either tolerance is met. Because L^2 norms are computed by summing over all the elements in a mesh, users are cautioned that the tolerance used with Equation (3.14) to achieve a given level of control stiffness convergence is mesh dependent. That is, as the number of elements greatly increases, the L^2 norm of Equation (3.14) naturally increases. In most cases, the relative strain error given in Equation (3.18) is the preferred error measure. In addition to the error measures defined by Equations (3.14), (3.15), or (3.18), it is necessary to determine whether equilibrium is still achieved once the reference stresses have been updated. That is, equilibrium is re-evaluated with σ_{I+1} and s_{I+1} calculated without any additional changes to the fundamental nodal degrees of freedom such that

$$\sigma_{I+1} = \sigma_I \tag{3.19}$$

and

$$s_{I+1} = r_{I+1} + \lambda \left(\sigma_{I+1} - r_{I+1} \right) . \qquad (3.20)$$

If convergence is achieved both for the difference between the scaled stress and the unscaled stress (either as a direct measure or as a strain error) and for the updated equilibrium evaluation, the time step is considered complete. Note that if control contact is also active, it is necessary that the appropriate contact convergence checks be satisfied as well.

Figure 3.9 shows an example of a linear material that has been softened through control stiffness. In this example, the time step is considered converged immediately after the second model problem update. The third model problem involved no iterations and is thus simply a re-evaluation of equilibrium and control stiffness convergence after the reference stress is updated following the second model problem.

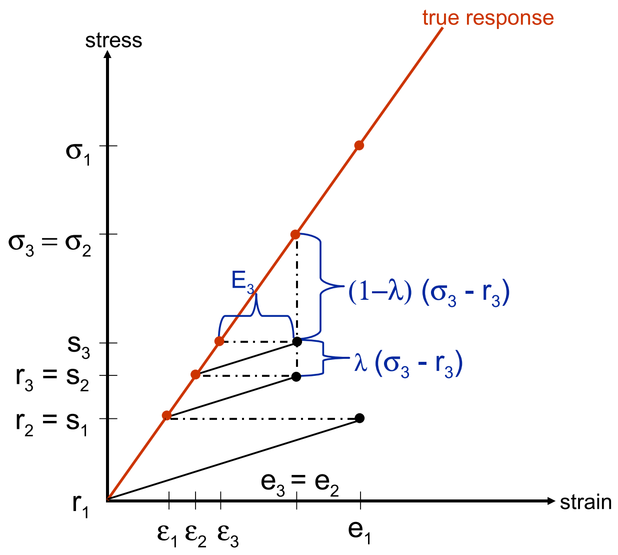


Figure 3.9: Control stiffness softening behavior convergence is achieved after solving two model problems.

The command block for control stiffness is as follows:

```
BEGIN CONTROL STIFFNESS [<string>stiffness_name]
  #
  # convergence commands
  TARGET <string>RESIDUAL|AXIAL FORCE INCREMENT|
    PRESSURE INCREMENT|SDEV INCREMENT|STRESS INCREMENT
    = <real>target [DURING <string list>period_names]
  TARGET RELATIVE <string>RESIDUAL|AXIAL FORCE INCREMENT|
    PRESSURE INCREMENT|SDEV INCREMENT|STRESS INCREMENT|
    STRAIN INCREMENT
    = <real>target_rel [DURING <string list>period_names]
  ACCEPTABLE <string>RESIDUAL|AXIAL FORCE INCREMENT|
    PRESSURE INCREMENT|SDEV INCREMENT|STRESS INCREMENT
    = <real>accept [DURING <string list>period_names]
  ACCEPTABLE RELATIVE <string>RESIDUAL|AXIAL FORCE INCREMENT|
    PRESSURE INCREMENT|SDEV INCREMENT|STRESS INCREMENT|
    STRAIN INCREMENT
    = <real>accept_rel [DURING <string list>period_names]
  REFERENCE = EXTERNAL|INTERNAL|RESIDUAL(EXTERNAL)
    [DURING <string list>period_names]
  MINIMUM ITERATIONS  = <integer>min_iter(0)
    [DURING <string list>period_names]
  MAXIMUM ITERATIONS  = <integer>max_iter
    [DURING <string list>period_names]
  #
  # level selection command
  LEVEL = <integer>stiffness_level
  #
  # diagnostic output commands
  ITERATION PLOT = <integer>iter_plot
    [DURING <string list>period_names]
  ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
END [CONTROL STIFFNESS <string>stiffness_name]
```

To enable control stiffness, a `CONTROL STIFFNESS` command block must exist in the `SOLVER` command block. The command lines within the `CONTROL STIFFNESS` command block are used to control convergence during the control stiffness updates, select the level for control stiffness within the multilevel solver, and output diagnostic information. These command lines are described in detail in Sections 3.6.1 through 3.6.3.

3.6.1 Convergence Commands

The command lines listed in this section are placed in the `CONTROL STIFFNESS` command block to control convergence criteria for stiffness control within the multilevel solver.

```
TARGET <string>RESIDUAL|AXIAL FORCE INCREMENT|
  PRESSURE INCREMENT|SDEV INCREMENT|STRESS INCREMENT
  = <real>target [DURING <string list>period_names]
TARGET RELATIVE <string>RESIDUAL|AXIAL FORCE INCREMENT|
  PRESSURE INCREMENT|SDEV INCREMENT|STRESS INCREMENT|
  STRAIN INCREMENT
  = <real>target_rel [DURING <string list>period_names]
ACCEPTABLE <string>RESIDUAL|AXIAL FORCE INCREMENT|
  PRESSURE INCREMENT|SDEV INCREMENT|STRESS INCREMENT
  = <real>accept [DURING <string list>period_names]
ACCEPTABLE RELATIVE <string>RESIDUAL|AXIAL FORCE INCREMENT|
  PRESSURE INCREMENT|SDEV INCREMENT|STRESS INCREMENT|
  STRAIN INCREMENT
  = <real>accept_rel [DURING <string list>period_names]
```

```
REFERENCE = EXTERNAL|INTERNAL|RESIDUAL(EXTERNAL)
  [DURING <string list>period_names]
MINIMUM ITERATIONS = <integer>min_iter(0)
  [DURING <string list>period_names]
MAXIMUM ITERATIONS = <integer>max_iter
  [DURING <string list>period_names]
```

Convergence for problems using control stiffness requires that two criteria be met. The first criterion is that equilibrium must be achieved. The second criterion is that the scaled stress used for equilibrium must be close enough to the unscaled stress that is determined with the unmodified material model as calculated from either Equation (3.14), (3.15), or (3.18).

The command lines listed above for the equilibrium evaluation are similar to those used for the core CG solver, and have similar meaning. Here, however, the command lines are applied as part of determining convergence for the control stiffness series of model problems. Note that some of the command lines have multiple options, e.g., the command line beginning with `TARGET` has five options. This practice has been used for the command lines beginning with `TARGET`, `TARGET RELATIVE`. `ACCEPTABLE`, and `ACCEPTABLE RELATIVE`. Note also that all of these command lines can be appended with the `DURING` specification, as discussed in Section 3.2.1.

Convergence of equilibrium using scaled stresses is measured by computing the L^2 norm of the residual, and comparing that residual norm with target convergence criteria specified by the user. Convergence can be monitored either directly in terms of the residual norm, or in terms of a relative residual, which is the residual norm divided by a reference quantity that is indicative of the current loading conditions on a model. Either the residual, the relative residual, or both of these can be used for checking convergence at the same time. The `TARGET RESIDUAL` command line specifies the target convergence criterion in terms of the actual residual norm. The `TARGET RELATIVE RESIDUAL` command line specifies a convergence criterion in terms of the relative residual. If both absolute and relative criteria are specified for the residual calculation, the equilibrium is accepted if either criterion is met.

The multilevel solver also allows acceptable convergence criteria to be input for residual convergence. If the solution has not converged to the specified targets before the maximum number of iterations is reached, the residual is checked against the acceptable convergence criteria. These criteria are specified via the `ACCEPTABLE RESIDUAL` and `ACCEPTABLE RELATIVE RESIDUAL` command lines. The concepts of absolute and relative values are the same here as discussed for the target limits.

If relative residuals are given in the convergence criteria, the `REFERENCE` command line can be used to select the method for computing the reference load. This command line has three options: `EXTERNAL`, `INTERNAL`, and `RESIDUAL`. With the `EXTERNAL` option, which is the default, the reference is computed by taking the L^2 norm of the current external load. The `INTERNAL` option uses the norm of the internal forces as the reference load. Finally, the `RESIDUAL` option denotes that the initial residual should be used as the reference quantity.

For control stiffness, unlike other controls, the convergence of the series of model problems to obtain the final solution for a time step is also based on other criteria that measure how far apart the scaled and unscaled stress responses are from each other. Either a direct error measure of the difference of the scaled and unscaled stress is used or a relative strain error computed using the difference between the scaled and unscaled stress and a representative modulus is employed.

There are a number of variants of the TARGET command that can be used for the direct stress differences. The TARGET AXIAL FORCE INCREMENT command is used to base convergence on the increment of axial force in fiber membrane elements in an update. The TARGET PRESSURE INCREMENT and TARGET SDEV INCREMENT commands are used to specify convergence for nearly incompressible materials based on the pressure and deviatoric stress increments, respectively. Also, TARGET STRESS INCREMENT is used to base convergence on the stress increment for materials which have their entire behavior softened as part of a control stiffness approach. If desired, the RELATIVE form of these convergence criteria can be used. If the TARGET RELATIVE STRAIN INCREMENT command is specified, the strain error measure between the scaled and unscaled stress will be computed and used.

Any combination of these criteria can be specified. If more than one is specified, all of the criteria must be satisfied. However, each material block contributes either to the direct stress error measures or to the relative strain error measure. If a material block specifies a positive non-zero REFERENCE STRAIN in its definition, it will contribute to the relative strain error measure. Otherwise, it will contribute to the direct stress error measures. Finally, acceptable convergence criteria can be input for converging the difference between the scaled and unscaled stress. If the solution has not met the target criteria for equilibrium and the difference between the scaled and unscaled stresses, but meets the acceptable criteria, it is allowed to proceed to the next time step.

The MAXIMUM ITERATIONS and MINIMUM ITERATIONS command lines specify the maximum and minimum number of control stiffness updates, respectively. The default minimum number of iterations, min_iter, is 0. If a number greater than 0 is specified for min_iter, the multilevel solver will update stiffness at least that many times, regardless of whether the convergence criteria have been met.

3.6.2 Level Selection Command

```
LEVEL = <integer>stiffness_level
```

The LEVEL command line in the CONTROL STIFFNESS command block is used to specify the level in the multilevel solver at which stiffness is controlled. The stiffness_level parameter can have a value of either 1 or 2, with 1 being the default.

This command is used when multiple controls (i.e. control contact and control stiffness) are active in the multilevel solver. It is permissible for multiple controls to exist at a given level. The default behavior is for all controls to be at level 1. To have a control at level 2, another control must be active at level 1 because a level in the multilevel solver cannot be skipped. The level of other controls is specified using the LEVEL command line in the command blocks associated with those controls.

3.6.3 Diagnostic Output Commands

```
ITERATION PLOT = <integer>iter_plot
  [DURING <string list>period_names]
ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
```

The command lines listed above can be used to control the output of diagnostic information from the stiffness control within the multilevel solver. The `ITERATION PLOT` command line allows plots of the current state of the model to be written to the output database during the stiffness control updates. The value supplied in `iter_plot` specifies the frequency of such plots. The default behavior is not to produce such plots. The plots are generated with a fictitious time step. Every time such a plot is generated, a message is sent to the log file. This feature is useful for debugging, but it should be used with care. Note that the `ITERATION PLOT` command line can be appended with the `DURING` specification, as discussed in Section 3.2.1.

The output produced by iteration plots is sent by default to every active results output file, as specified by a `RESULTS OUTPUT` command block. The `ITERATION PLOT OUTPUT BLOCKS` command line can optionally be used to supply a list of output block names to which iteration plots should be written. Each of the output block names listed in `plot_blocks`, must match the name of a `RESULTS OUTPUT` command block (see Section 8.2.1). The `ITERATION PLOT OUTPUT BLOCKS` command line can be useful for directing iteration plots to separate output files so that these plots are separated from the converged solution output. It can also be used to send output plots for different levels of the multilevel solver to different files.

3.7 Control Failure

Control failure can be used to improve convergence in problems that involve material failure computed by the Multilinear Elastic-Plastic Failure material model or by user-defined global variables. For element death to occur in adagio, the user must define both an element death block (as defined in Section 5.5) and a multilevel solver with control failure.

The command block for control failure is as follows:

```
    BEGIN CONTROL FAILURE [<string>failure_name]
      #
      #  convergence control command
      MAXIMUM ITERATIONS = <integer>max_iter
        [DURING <string list>period_names]
      #
      # level selection command
      LEVEL = <integer>failure_level
      #
      # diagnostic output commands
      ITERATION PLOT = <integer>iter_plot
        [DURING <string list>period_names]
      ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
    END [CONTROL FAILURE <string>failure_name]
```

To enable control failure, a `CONTROL FAILURE` command block must exist in the `SOLVER` command block. The command lines within the `CONTROL FAILURE` command block are used to establish convergence criteria, select the level for control failure within the multilevel solver, and output diagnostic information. These command lines are described in detail in Sections 3.7.1 through 3.7.3.

Convergence of control failure is defined by two criteria. The first criterion is that no new elements have been marked as ready to begin to fail in the previous model problem iteration. Currently this control only works with the Multilinear Elastic-Plastic Failure material model. An optional second criteria is that this first criteria is reached before the maximum number of iterations is reached if this command is specified by the user.

3.7.1 Convergence Command

The only user input convergence criteria that is allowed for control failure is specification of maximum iterations.

```
  MAXIMUM ITERATIONS = <integer>max_iter
     [DURING <string list>period_names]
```

Iterations for control failure will continue until no new elements are failing unless the user specified number of maximum iterations is reached. If a maximum number of iterations is specified in the input, control failure iterations will only continue until this number is reached. If the maximum number of iterations is reached the analysis will terminate in an unconverged state and with an explanatory message in the log file. Note that this command can be appended with the DURING specification, as discussed in Section 3.2.1.

3.7.2 Level Selection Command

```
LEVEL = <integer>failure_level
```

The LEVEL command line in the CONTROL FAILURE command block is used to specify the level in the multilevel solver at which failure is controlled. The failure_level parameter can have a value of either 1 or 2, with 1 being the default.

This command is used when multiple controls (i.e. control contact and control failure) are active in the multilevel solver. The control failure should be the only control at the outermost (highest numerical value) level. To have a control at level 2, another control must be active at level 1 because a level in the multilevel solver cannot be skipped. The level of other controls is specified using the LEVEL command line in the command blocks associated with those controls.

3.7.3 Diagnostic Output Commands

```
ITERATION PLOT = <integer>iter_plot
  [DURING <string list>period_names]
ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
```

The command lines listed above can be used to control the output of diagnostic information from the failure control within the multilevel solver. The ITERATION PLOT command line allows plots of the current state of the model to be written to the output database during the failure control updates. The value supplied in iter_plot specifies the frequency of such plots. The default behavior is not to produce such plots. The plots are generated with a fictitious time step. Every time such a plot is generated, a message is sent to the log file. This feature is useful for debugging, but it should be used with care. Note that the ITERATION PLOT command line can be appended with the DURING specification, as discussed in Section 3.2.1.

The output produced by iteration plots is sent by default to every active results output file, as specified by a RESULTS OUTPUT command block. The ITERATION PLOT OUTPUT BLOCKS command line can optionally be used to supply a list of output block names to which iteration plots should be written. Each of the output block names listed in plot_blocks, must match the name of a RESULTS OUTPUT command block (see Section 8.2.1). The ITERATION PLOT OUTPUT BLOCKS command line can be useful for directing iteration plots to separate output files so that these plots are separated from the converged solution output. It can also be used to send output plots for different levels of the multilevel solver to different files.

3.8 Control Modes

The core conjugate gradient solution algorithm used by Adagio typically works very well on "blocky" problems if the nodal preconditioners are used. As the aspect ratios of the structures modeled increase, however, the performance of the CG solver begins to degrade. Because of their lack of coupling terms, the nodal preconditioners only allow for the effects of the residual to be propagated across a single element during an iteration. Consequently, these preconditioners are effective at minimizing high frequency error, but may require many iterations to solve for the low frequency, structural bending modes in slender structures.

The full tangent preconditioners provided by Adagio greatly improve the effectiveness of the CG algorithm for solving models with slender members. However, the memory usage requirements and the computational effort needed to factorize matrices with the full tangent preconditioners can sometimes be significant.

In addition to its other solution strategies, Adagio provides a multigrid solution method within the CG algorithm. This method, known as control modes, greatly increases the effectiveness of the CG algorithm for solving slender, bending-dominated problems. In addition to the actual mesh of the model to be solved, referred to as the "reference mesh," control modes uses a coarse mesh of the model, known as a "constraint mesh," to solve for the low-frequency response. To use control modes, the user must supply both the reference mesh and the constraint mesh. While it does require some additional effort to generate a suitable coarse mesh, control modes provides very efficient solutions with only slightly higher memory usage than would be required by the standard CG solver with a nodal preconditioner.

Control modes functions somewhat like another level in the multilevel solver, although it does not appear as a control type in the `SOLVER` command block. The CG algorithm operates alternately on the residual on the coarse mesh and the fine mesh. The coarse residual is computed by performing a restriction operation from the fine mesh to the coarse mesh using the shape functions of the coarse elements. The CG solver is used to minimize the coarse residual until convergence is reached, at which point it switches to operate on the fine mesh to minimize that residual. The fine mesh iterations continue until either the fine mesh has converged or the fine residual is 50% of the coarse residual. The solver alternates between the fine and coarse meshes until convergence is obtained on both meshes.

To use control modes, the user should set up the mesh file and the input file as usual, except that the following additional items must be provided:

- A constraint mesh must be generated. The constraint mesh must be in a separate file from the reference mesh, which is the real model. The constraint mesh should be a coarse representation of the reference mesh. If there are node sets or side sets in the reference mesh that are used to prescribe kinematic boundary conditions, similar mesh entities should be provided in the coarse mesh to prescribe similar boundary conditions.

- A second `FINITE ELEMENT MODEL` command block must be provided in addition to the standard definition for the reference finite element model in the input file. This command block is set up exactly as it normally would be (see Section 5.1), except that the mesh file referenced is the constraint mesh instead of the reference mesh. Although the constraint mesh is used purely as a solution tool, and does not use any finite elements or material models, each block in the constraint mesh must still be assigned a material model.

- A `CONTROL MODES REGION` command block must appear alongside the standard `ADAGIO REGION` command block within the `ADAGIO PROCEDURE` command block. The presence of the `CONTROL MODES REGION` command block instructs the CG solver to use the control modes logic. There are no commands within the CG solver for the region associated with the reference mesh related to control modes. The `CONTROL MODES REGION` command block is documented in Section 3.8.1. It contains the same commands used within the standard `ADAGIO REGION` command block, except that the commands in the `CONTROL MODES REGION` command block are used to control the control modes algorithm and the boundary conditions on the coarse mesh.

3.8.1 Control Modes Region

```
BEGIN CONTROL MODES REGION
  #
  # model setup
  USE FINITE ELEMENT MODEL <string>model_name
  CONTROL BLOCKS = <string list>control_blocks
  #
  # solver commands
  BEGIN SOLVER
    BEGIN LOADSTEP PREDICTOR
      #
      # Parameters for loadstep predictor
      #
    END [LOADSTEP PREDICTOR]
    BEGIN CG
      #
      # Parameters for CG
      #
    END [CG]
  END [SOLVER]
  JAS MODE [SOLVER|CONTACT|OUTPUT]
  #
  # kinematic boundary condition commands
  BEGIN FIXED DISPLACEMENT
    #
    # Parameters for fixed displacement
    #
  END [FIXED DISPLACEMENT]
  BEGIN PERIODIC
    #
    # Parameters for periodic
    #
  END [PERIODIC]
END [CONTROL MODES REGION]
```

The CONTROL MODES REGION command block controls the behavior of the control modes algorithm, and is placed alongside a standard ADAGIO REGION command block within the ADAGIO PROCEDURE scope. With the exception of the CONTROL BLOCKS command line, all the commands that can be used in this block are standard commands that appear in the Adagio region. These commands have the same meaning in either context; they simply apply to the constraint mesh or to the reference mesh, depending on the region block in which they appear. Sections 3.8.1.1 through 3.8.1.3 describe the components of the CONTROL MODES REGION command block.

3.8.1.1 Model Setup Commands

```
USE FINITE ELEMENT MODEL <string>model_name
CONTROL BLOCKS = <string list>control_blocks
```

The command lines listed above must appear in the CONTROL MODES REGION command block if control modes is used. The USE FINITE ELEMENT MODEL command line should reference the finite element model for the constraint mesh. This command line is used in the same way that the command line is used for the reference mesh (see Section 2.3).

The CONTROL BLOCKS command line provides a list of blocks in the reference mesh that will be controlled by the constraint mesh in the solver. The block names are listed using the standard method of referencing mesh entities (see Section 1.5). For example, the block with an ID of 1 would be listed as block_1 in this command. Multiple CONTROL BLOCKS command lines may appear to specify a long list of blocks over several lines.

3.8.1.2 Solver Commands

```
BEGIN SOLVER
  BEGIN LOADSTEP PREDICTOR
    #
    # Parameters for loadstep predictor
    #
  END [LOADSTEP PREDICTOR]
  BEGIN CG
    #
    # Parameters for CG
    #
  END [CG]
END SOLVER
JAS MODE [OUTPUT|CONTACT|SOLVER]
```

The constraint mesh must have solver parameters defined using the `LOADSTEP PREDICTOR` and `CG` command blocks, which must be nested within a `SOLVER` block in the `CONTROL MODES REGION`. The constraint mesh does not have material properties or contact, so the `CG` command block is used to define the solver on the constraint mesh. The multilevel solver is not used on the constraint mesh, even though the multilevel solver may be used on the reference mesh in an analysis that uses a constraint mesh. All the command lines that appear in these command blocks in a standard analysis can be used for the parameters of the solver on the constraint mesh. Refer to Section 3.9.1 for a description of the command lines for the load step predictor and to Section 3.2 for a description of the CG solver commands.

The full set of preconditioners that are available on the reference mesh are also available for the constraint mesh. The full tangent preconditioner can be used on the constraint mesh, and provides a very powerful solution strategy. Because the constraint mesh is typically very small compared to the reference mesh, forming and factorizing matrices for the constraint mesh typically requires small computational resources.

The `JAS MODE` command line enables complete compatibility with the JAS3D legacy code. If this mode of operation is desired, the `JAS MODE` command line must be present in both the `CONTROL MODES REGION` command block and the `ADAGIO REGION` command block. See Section 3.10 for more information on this command line.

3.8.1.3 Kinematic Boundary Condition Commands

```
BEGIN FIXED DISPLACEMENT
  #
  # Parameters for fixed displacement
  #
END [FIXED DISPLACEMENT]
BEGIN PERIODIC
  #
  # Parameters for periodic
  #
END [PERIODIC]
```

For the solver to converge well, it is often important to create a node set or a side set on the coarse mesh that coincides geometrically with a node set or a side set on the fine mesh to which a fixed boundary condition is applied. This should be done for fixed boundary conditions, but not for boundary conditions with prescribed nonzero displacements. Any of the kinematic boundary condition command blocks that can appear in a standard Adagio region can also appear in the `CONTROL MODES REGION` command block. However, it is recommended that only `FIXED DISPLACEMENT` and `PERIODIC` boundary conditions be used on the constraint mesh.

The boundary conditions are applied to the specified mesh entities on the constraint mesh. The constraint mesh and the reference mesh can have a node set or a side set with the same identifier. The determination of which entity should be affected by the boundary condition is based on the context of the boundary condition specified. See Section 6.3.1 for details on the `FIXED DISPLACEMENT` command block.

3.9 Predictors

Predictors are an important component of Adagio's nonlinear solution strategy. A predictor is used to generate an initial trial solution for a load step or for a multilevel solver model problem. The goal of the predictor is to generate a trial solution that is as close as possible to the actual converged solution. A good prediction that gives a trial solution close to the actual solution can dramatically reduce the number of iterations required for convergence.

There are two types of predictors used in Adagio: the load step predictor and the level 1 predictor. The load step predictor, documented in Section 3.9.1, estimates a solution at the beginning of a new load step, and is applicable to all types of models. The level 1 predictor estimates the solution at the beginning of a new level 1 model problem in the multilevel solver. This predictor type, documented in Section 3.9.2, is only applicable for the multilevel solver.

3.9.1 Loadstep Predictor

```
BEGIN LOADSTEP PREDICTOR
  TYPE = <string>SCALE_FACTOR|SECANT|EXTERNAL|EXTERNAL_FIRST
    (SECANT)
  SCALE FACTOR = <real>scale_factor(1.0)
    [<real>first_scale_factor]
    [DURING <string list>period_names]
  SLIP SCALE FACTOR = <real>slip_factor(1.0)
    [DURING <string list>period_names]
END [LOADSTEP PREDICTOR]
```

The `LOADSTEP PREDICTOR` command block controls the behavior of the predictor that is used to predict the solution at the beginning of a new load step. This command block is placed in the `SOLVER` scope.

3.9.1.1 Predictor Type

```
TYPE = <string>SCALE_FACTOR|SECANT|EXTERNAL|EXTERNAL_FIRST
  (SECANT)
```

There are several types of load step predictors available in Adagio that are suitable for different types of analysis. The predictor type is selected using the `TYPE` command line in the `LOADSTEP PREDICTOR` command block.

- The scale factor predictor is selected with the `SCALE_FACTOR` option. This type of predictor extrapolates from the solution of the previous time step using the velocity field from that step, multiplied by a scale factor, to obtain a trial solution. The actual scale factor that is applied can be controlled with the `SCALE FACTOR` command line, which also appears in the `LOADSTEP PREDICTOR` command block. This type of predictor works well with models that have smoothly varying loads.

- The secant predictor is selected with the `SECANT` option, which is useful for scenarios in which the loading may not be monotonic. This type of predictor uses a line search to compute an optimal scaling factor for use in extrapolating from the previous solution. This type of predictor is the default, and provides good performance without requiring user intervention to select appropriate scale factors.

  A scale factor can optionally be specified for the `SECANT` predictor using the `SCALE FACTOR` command line. In this case, the user-specified scale factor is used instead of that computed by the line search for the time periods where a scale factor is defined. This permits switching between line search and scale factor predictors over the course of an analysis.

- The external predictor is selected with the `EXTERNAL` option. This type of predictor uses the solution from a file to predict the solution at new load steps. Re-using the work done in previous solutions of similar problems can be very helpful. Using this type of predictor requires that the input mesh file be a results file that contains displacement data.

- The `EXTERNAL_FIRST` option is used to select a special type of predictor that (1) uses the external predictor for the first load step of the solution period and (2) thereafter uses the scale factor predictor for every other load step in the solution period. If this option is chosen, the `SCALE FACTOR` command line must be included in the `LOADSTEP PREDICTOR` command block.

3.9.1.2 Scale Factor

```
SCALE FACTOR = <real>scale_factor(1.0)
  [<real>first_scale_factor]
  [DURING <string list>period_names]
```

The `SCALE FACTOR` command controls the behavior of the `SCALE_FACTOR` and `SECANT` predictor types (and the `EXTERNAL_FIRST` predictor type in all load steps except the first in a time period). The velocity field of the previous step, multiplied by the value of `scale_factor`, is used to extrapolate from the previous solution to obtain a trial solution. The default value of 1.0 provides good performance on models that experience smooth loading. Setting the value of `scale_factor` to 0.0 may improve the convergence of models with discontinuous loading.

The optional `first_scale_factor` parameter is used as the scale factor for the first step of a time period. If `first_scale_factor` is omitted, the default behavior is to use `scale_factor` for all time steps. Models are often subjected to loading that is mostly monotonic, but with discontinuities at the beginning of a new period. Using 1.0 and 0.0 for the values of `scale_factor` and `first_scale_factor`, respectively, often gives good performance in such cases.

The `SCALE FACTOR` command line can be appended with the `DURING` specification, as discussed in Section 3.2.1. This allows for the scale factor to be specified per time period.

3.9.1.3 Slip Scale Factor

```
SLIP SCALE FACTOR = <real>slip_factor(1.0)
```

```
        [DURING <string list>period_names]
```

The prediction of slip on slave nodes that are currently active in sliding contact can be controlled separately from other nodes in the model by using the `SLIP SCALE FACTOR` command line. The parameter `slip_factor` controls the scaling of the predicted slip on these nodes. If `slip_factor` is left at its default value of 1.0, contact slip will be predicted using the same scale factor used for the rest of the model. If the value of `slip_factor` is set to 0.0, the predicted solution will not allow any sliding contact nodes to slip. This command line applies to both the `SCALE_FACTOR` and the `SECANT` types of predictors.

The `SLIP SCALE FACTOR` command line can be appended with the `DURING` specification, as discussed in Section 3.2.1.

3.9.2 Level Predictor

```
LEVEL 1 PREDICTOR = <string>NONE|DEFAULT(DEFAULT)
```

The `LEVEL 1 PREDICTOR` command line controls the level 1 predictor. This command line appears in the `SOLVER` command block, and is only applicable if the multilevel solver is used. The level 1 predictor estimates the solution for the next model problem based on the solution of the previous model problem. The default behavior is to enable this predictor, indicated by `DEFAULT`. The predictor can be turned off by setting it to `NONE`.

3.10 JAS3D Compatibility Mode

Adagio's multilevel solver and CG solver are based on the solver used in the legacy JAS3D code [1]. While the solvers in the two codes are very similar, there are some minor implementation differences between the two codes.

```
    JAS MODE [SOLVER|CONTACT|OUTPUT]
```

Adagio provides an option, selectable by inserting the `JAS MODE` command line in the `ADAGIO REGION` command block, to make it use exactly the same algorithms as the JAS3D solver. If control modes is being used, the `JAS MODE` command line must appear both in the `ADAGIO REGION` and in the `CONTROL MODES REGION` command blocks (see Section 3.8.1.2). This option is primarily useful for migrating analyses previously done in JAS3D to Adagio.

The `JAS MODE` command used without any options enables all JAS3D compatibility features available. Optionally, this command can be followed by one of the three keywords `SOLVER`, `CONTACT`, or `OUTPUT` to enable a subset of these features. The command can be repeated on separate lines with different options to enable more than one subset of features.

The subsets of the features enabled by the `JAS MODE` command line are summarized below:

- `SOLVER`: Adagio's CG solver and multilevel solver use exactly the same algorithms. There are a number of algorithmic decisions in the solver that were made one way in JAS3D and another way in Adagio. While both methods are valid, the `JAS MODE` command line forces the code to always use the exact JAS3D algorithms.

- `CONTACT`: Adagio uses the Legacy Contact library instead of the internal Adagio and ACME code for contact search and enforcement. The Legacy Contact library is the same contact code that is used in JAS3D, so contact in Adagio models behaves the same as it would if it were run in JAS3D if this option is chosen.

- `OUTPUT`: The log file output produced by Adagio is formatted in the same manner as JAS3D formats log files, thus facilitating comparisons between the output from JAS3D and Adagio models.

Note that the `JAS MODE` command line only ensures JAS3D compatibility for features that are available in JAS3D. Advanced features such as the full tangent preconditioner are not available in JAS3D. It should also be noted that for those interested in converting JAS3D input files to Adagio input files, there is a program available for this purpose. For more information on this converter program, refer to the Adagio web page or contact an Adagio developer.

3.11 Time Step Control

Time stepping in Adagio is controlled by using a `TIME CONTROL` command block in the procedure scope. In this command block, the user controls the start time, the termination time, and the method by which time is advanced during the analysis. The analysis time can optionally be subdivided into a number of time periods. If the analysis is from time 0 to time T and it is split into three periods, the first period is defined from time 0 to time t_1, the second period is defined from time t_1 to time t_2, and the third period is defined from time t_2 to time T. (The times t_1 and t_2 are set by the user.) The sum of the times for the three periods is T.

There are many reasons for splitting an analysis into multiple time periods. It is usually desirable to start time periods when a change in the model or in the loading conditions occurs that is significant enough to warrant switching to a different set of solver parameters or boundary conditions. Many of Adagio's features can be toggled on or off or changed during different time periods. For example, boundary conditions and contact can be activated during certain time periods. Similarly, many solution parameters can change based on the time period.

Time stepping can be uniform during a solution period, or it can vary during a solution period by using a function to control the time step as a function of analysis time. In Adagio, an automatic time stepping scheme can also be used to automatically take larger time steps at points in the analysis when the solution is relatively easy, and take smaller time steps when it becomes more difficult to solve a time step.

A description of the `TIME CONTROL` command block follows in Section 3.11.1. Section 3.11.2 describes the `ADAPTIVE TIME STEPPING` command block, which allows the user to adapt the size of the time step based on solution difficulty. A simple example of a `TIME CONTROL` command block is presented in Section 3.11.3.

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

Adagio time control resides in a `TIME CONTROL` command block. The command block begins with the input line

```
BEGIN TIME CONTROL
```

and terminates an input line of the following form:

```
END [TIME CONTROL]
```

An arbitrary number of `TIME STEPPING BLOCK` command blocks can be present to define individual time stepping periods within the `TIME CONTROL` command block. 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 period terminates at the start time of the following time period. The start times for the `TIME STEPPING BLOCK` command blocks must appear in increasing order or an error message will result. The example in Section 3.11.3 shows the overall structure of the `TIME CONTROL` command block.

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

- The string `time_block_name` is a name for the time period. Every time period must have a unique name. These names are referenced to control solution parameters or to activate/deactivate functionality.

- The real value `start_time_value` is the start time for this `TIME STEPPING BLOCK` command block. A time period goes from its start time until the start time of the next period or the termination time. The start time may be negative.

- The string `region_name` is the name of the Adagio region affected by the parameters (see Section 2.2).

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

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

Here, `termination_time` is the time at which the analysis will stop. The `TERMINATION TIME` command line appears inside the `TIME CONTROL` command block 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 greater than 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 in the `TIME CONTROL` command block. If more than one of these command lines appears, Adagio gives an error.

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

```
BEGIN PARAMETERS FOR ADAGIO REGION <string>region_name
  TIME INCREMENT = <real>time_increment_value
  NUMBER OF TIME STEPS = <integer>nsteps
  TIME INCREMENT FUNCTION = <string>time_function
END [PARAMETERS FOR ADAGIO REGION <string>region_name]
```

These parameters are specific to an Adagio analysis.

The command block begins with an input line of the form

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

and is terminated with an input line of the following form:

```
END [PARAMETERS FOR ADAGIO REGION <string>region_name]
```

As noted previously, the string `region_name` is the name of the Adagio region to which the parameters apply. The command lines nested inside the `PARAMETERS FOR ADAGIO REGION` command block are used to control the way time stepping occurs, and are described in Sections 3.11.1.1 through 3.11.1.3. Only one of these command lines (`TIME INCREMENT`, `NUMBER OF TIME STEPS`, or `TIME INCREMENT FUNCTION`) may be used in a given time period.

3.11.1.1 Time Increment

```
    TIME INCREMENT = <real>time_increment_value
```

The `TIME INCREMENT` command line directly specifies the size of a uniform time step that will be used during a time period. The size of the time step is specified with the parameter `time_increment_value`.

3.11.1.2 Number of Time Steps

```
    NUMBER OF TIME STEPS = <integer>nsteps
```

The `NUMBER OF TIME STEPS` command line specifies the number of uniform time steps that will be used during a time period. The number of time steps is specified with the parameter `nsteps`. If this command line is used, the time step size is computed by dividing the length of the time period by the specified number of time steps.

3.11.1.3 Time Increment Function

```
    TIME INCREMENT FUNCTION = <string>time_function
```

The `TIME INCREMENT FUNCTION` command line allows the time step to be specified as a function of analysis time. The function used to control the time step is referenced by the parameter `time_function`. The actual function is defined within a `DEFINITION FOR FUNCTION` command block within the SIERRA scope. It is often convenient to use a function of the `PIECEWISE CONSTANT` type (see Section 2.1.5) with this command line. Using a function to control time incrementation allows for the time increment to be conveniently switched many times during an analysis without creating a new time block every time the time increment needs to be changed.

3.11.2 Adaptive Time Stepping

```
BEGIN ADAPTIVE TIME STEPPING
  METHOD = <string>SOLVER|MATERIAL(SOLVER)
    [DURING <string list>period_names]
  TARGET ITERATIONS = <integer>target_iter
    [DURING <string list>period_names]
  ITERATION WINDOW = <integer>iter_window
    [DURING <string list>period_names]
  CUTBACK FACTOR = <real>cutback_factor(0.5)
    [DURING <string list>period_names]
  GROWTH FACTOR = <real>growth_factor(1.5)
    [DURING <string list>period_names]
  MAXIMUM FAILURE CUTBACKS = <integer>max_cutbacks(5)
    [DURING <string list>period_names]
  MAXIMUM MULTIPLIER = <real>max_multiplier
    [DURING <string list>period_names]
  MINIMUM MULTIPLIER = <real>min_multiplier
    [DURING <string list>period_names]
  RESET AT NEW PERIOD = TRUE|FALSE(TRUE)
    [DURING <string list>period_names]
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [ADAPTIVE TIME STEPPING]
```

Adagio has the capability to adapt the time step size during an analysis based on either solution difficulty or on feedback from material models. This capability is enabled by inserting an `ADAPTIVE TIME STEPPING` command block in the region scope. If this command block is not present in the Adagio region, the time stepping specified in the `TIME CONTROL` command block will be used. In addition to adjusting the size of time steps as the analysis proceeds, the adaptive time stepping algorithm can attempt to solve a load step by using a smaller time step if a solution fails to converge.

The adaptive time stepping algorithm works by computing a multiplier that is applied to the time step specified in the `TIME CONTROL` command block. Thus, even if adaptive time stepping is used, the time step must be specified for all loading periods in the `TIME CONTROL` command block. For the initial load step in each solution period, Adagio always uses the prescribed time step. In subsequent steps, the adaptive time stepping algorithm computes a factor that is multiplied by the prescribed time step to adjust the time step based on solution difficulty.

The command lines nested within the `ADAPTIVE TIME STEPPING` command are used to control the behavior of this feature and are described in Sections 3.11.2.1 through 3.11.2.10.

Many of these command lines can optionally be set for a specific time period. To set a parameter for a specific time period, the user can append the `DURING` keyword followed by a list of applicable period names to a command line (see Section 3.2.1. Multiple instances of these command lines can exist to set the parameter differently for different periods. If a command line is not appended with the `DURING` specification, it is used to set the default behavior for all time periods. Setting a default in this way does not preclude setting a period-specific value for that parameter with another instance of the same command line.

3.11.2.1 Method

```
METHOD = <string>SOLVER|MATERIAL(SOLVER)
  [DURING <string list>period_names]
```

The `METHOD` command line is used to select the type of adaptive time stepping to be used. The default `SOLVER` method adapts the time step based on the difficulty of the solution. Solution difficulty is determined based on the number of iterations required to achieve convergence. If the `SOLVER` method is used, the `TARGET ITERATIONS` command line must be used to specify the target iterations used by this method to adapt the time step.

The `MATERIAL` method adapts the time step based on feedback from the material model. Some material models are capable of computing a recommended time step. When material-based adaptive time stepping is used, the minimum time step recommended by all material integration points in the model is used.

3.11.2.2 Target Iterations

```
TARGET ITERATIONS = <integer>target_iter
  [DURING <string list>period_names]
```

The solver method for adaptive time stepping adjusts the time step to achieve a target level of difficulty. The level of difficulty is determined by the total number of core solver iterations required to solve for a time step. The `TARGET ITERATIONS` command is used to specify a target number of iterations per time step. This command line must be provided if the solver based adaptivity method is to be used.

The specified value of `target_iter` is used in conjunction with the value of `iter_window` (specified with the `ITERATION WINDOW` command documented in Section 3.11.2.3) to control the next step size. If the number of iterations for the previous step is greater than (`target_iter-iter_window`), and less than (`target_iter+iter_window`), the step size is kept the same. If the number of iterations is greater than (`target_iter+iter_window`), the step size is decreased, and if it is less than (`target_iter-iter_window`), the step size is increased.

3.11.2.3 Iteration Window

```
ITERATION WINDOW = <integer>iter_window
  [DURING <string list>period_names]
```

The `ITERATION WINDOW` command is used to specify the size of the window for the iteration count used in the solver type of adaptive time stepping. See Section 3.11.2.2 for an explanation of how this is used in conjunction with the specified target iterations to adaptively choose the time step size. The default value of `iter_window` is the value specified in the `TARGET ITERATIONS` command line divided by 10.

3.11.2.4 Cutback Factor

```
CUTBACK FACTOR = <real>cutback_factor(0.5)
  [DURING <string list>period_names]
```

The `CUTBACK FACTOR` command line controls the amount by which the next time step is reduced if the solution is difficult. The parameter `cutback_factor`, which has a default value of 0.5, specifies the multiplier that is applied to the current time step with each cutback. The parameter also controls the amount that the current time step is cut back if a solution fails. Note that the `CUTBACK FACTOR` command line can be appended with the `DURING` specification, as discussed in Section 3.2.1.

3.11.2.5 Growth Factor

```
GROWTH FACTOR = <real>growth_factor(1.5)
  [DURING <string list>period_names]
```

The `GROWTH FACTOR` command line controls the amount by which the next time step is increased if the solution is easy. The parameter `growth_factor`, which has a default value of 1.5, specifies the multiplier (a 50 percent increase) that is applied to the current time step each time the time step is increased. Note that the `GROWTH FACTOR` command line can be appended with the `DURING` specification, as discussed in Section 3.2.1.

3.11.2.6 Maximum Failure Cutbacks

```
MAXIMUM FAILURE CUTBACKS = <integer>max_cutbacks(5)
  [DURING <string list>period_names]
```

The `MAXIMUM FAILURE CUTBACKS` command line controls how many cutbacks to a time step can be taken if a solution fails to converge. The The parameter `max_cutbacks` has a default value of 5 if adaptive time stepping is enabled. If an `ADAPTIVE TIME STEPPING` command block is not present, the code will exit rather than attempting to cut back in the event of an unsuccessful solution. Note that the `MAXIMUM FAILURE CUTBACKS` command line can be appended with the `DURING` specification, as discussed in Section 3.2.1.

3.11.2.7 Maximum Multiplier

```
MAXIMUM MULTIPLIER = <real>max_multiplier
  [DURING <string list>period_names]
```

Time steps are adaptively adjusted by updating a multiplier that is applied to the base time step specified in the `TIME CONTROL` command block. By default, there are no limits on the size of this multiplier. The `MAXIMUM MULTIPLIER` command line can be used to limit the growth of the time step. The parameter `max_multiplier` specifies the upper bound for the multiplier. The largest possible time step in an analysis is the product of `max_multiplier` and the baseline time step size. This baseline size is specified in the `TIME CONTROL` command block. Note that the `MAXIMUM MULTIPLIER` command line can be appended with the `DURING` specification, as discussed in Section 3.2.1.

3.11.2.8 Minimum Multiplier

```
MINIMUM MULTIPLIER = <real>min_multiplier
  [DURING <string list>period_names]
```

Time steps are adaptively adjusted by updating a multiplier that is applied to the base time step specified in the `TIME CONTROL` command block. By default, there are no limits on the size of this multiplier. The `MINIMUM MULTIPLIER` command line can be used to limit the shrinkage of the time step. The parameter `min_multiplier` specifies the lower bound for the multiplier. The smallest possible time step in an analysis is the product of `min_multiplier` and the baseline time step size. Note that the `MINIMUM MULTIPLIER` command line can be appended with the `DURING` specification, as discussed in Section 3.2.1.

3.11.2.9 Reset at New Period

```
RESET AT NEW PERIOD = TRUE|FALSE(TRUE)
  [DURING <string list>period_names]
```

This command controls whether the time step is reset to the user-specified base time step at the beginning of a new time period. If this is set to `TRUE`, which is the default value, the time step is reset. If this parameter is set to `FALSE`, the time step at the end of the previous period is maintained. If the base time step is changed in the new period, the multiplier is adjusted to maintain the same time step. If the resulting multiplier is outside the specified range, it will be adjusted to stay within that range and the time step will change. The value of this parameter can optionally be specified per time period with the `DURING` option (see Section 3.2.1). If adaptive time stepping is inactive for a given period, the time step will always be reset even if it is requested that the adaptive time step not be reset.

3.11.2.10 Active or Inactive Periods

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

If the `ADAPTIVE TIME STEPPING` command block is present in the Adagio region, it is active for all periods by default. Adaptive time stepping can be activated or deactivated for specific time periods with the ACTIVE PERIODS or INACTIVE PERIODS command lines. See Section 2.5 for more information about these optional command lines.

3.11.3 Time Control 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 ADAGIO REGION adagio_region
      NUMBER OF TIME STEPS = 10
    END
  END
  BEGIN TIME STEPPING BLOCK p2
    START TIME = 1.0
    BEGIN PARAMETERS FOR ADAGIO REGION adagio_region
      TIME INCREMENT = 0.2
    END
  END
  TERMINATION TIME = 2.0
END
```

The first TIME STEPPING BLOCK, p1, begins at time 0.0, the initial start time, and terminates at time 1.0. The second TIME STEPPING BLOCK, p2, begins at time 1.0 and terminates at time 2.0, the time listed on the TERMINATION TIME command line. The TIME STEPPING BLOCK names p1 and p2 can be referenced elsewhere to activate boundary conditions or control the solver in different ways during the two periods.

3.12 Implicit Dynamic Time Integration

Adagio has the ability to perform implicit solutions on both quasistatic and dynamic problems. In quasistatic calculations, the solution for static equilibrium is obtained at each step, ignoring the inertial terms of the equations of motion. In dynamic problems, the inertial forces are included, and the HHT time integrator [5] is used to integrate the equations of motion in time. The behavior of the HHT integrator is controlled with three parameters: α, β, and γ. With proper selection of these parameters, the HHT integrator is unconditionally stable. If $\alpha = 0$, this reduces to the Newmark method [7]. The trapezoidal rule is recovered if $\alpha = 0$, $\beta = 0.25$, and $\gamma = 0.5$. For a detailed discussion of the theory of implicit time integrators, see Reference 6.

3.12.1 Implicit Dynamics

```
BEGIN IMPLICIT DYNAMICS
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
  USE HHT INTEGRATION
  ALPHA = <real>alpha(0.0) [DURING <string list>period_names]
  GAMMA = <real>beta(0.5) [DURING <string list>period_names]
  BETA = <real>beta(0.25) [DURING <string list>period_names]
  TIME INTEGRATION CONTROL = <string>ADAPTIVE|COMPUTERESIDUAL|
    IGNORE(IGNORE) [DURING <string list>period_names]
  INCREASE ERROR THRESHOLD = <real>increase_threshold(0.02)
    [DURING <string list>period_names]
  HOLD ERROR THRESHOLD = <real>hold_threshold(0.10)
    [DURING <string list>period_names]
  DECREASE ERROR THRESHOLD = <real>decrease_threshold(0.25)
    [DURING <string list>period_names]
END [IMPLICIT DYNAMICS]
```

The `IMPLICIT DYNAMICS` command block enables implicit dynamics and contains commands to control the behavior of implicit time integration. This command block is placed in the `ADAGIO REGION` command block. Note that prior versions of Adagio required an Andante region to use implicit dynamics. The Andante region has been eliminated, and this capability has all been moved to the Adagio region, which gives the analyst much more flexibility. Formerly, it was necessary to create a new procedure for switching between implicit dynamics and quasistatics in an analysis. Now, these different analysis types can all be done within a single region, and the command lines in the `IMPLICIT DYNAMICS` command block can be used to enable or disable implicit dynamics for specific time periods within a single region. Sections 3.12.1.1 through 3.12.1.4 describe the command lines in this command block.

3.12.1.1 Active or Inactive Periods

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

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

If the `IMPLICIT DYNAMICS` command block is present in the Adagio region, it is active for all periods by default. Implicit dynamics can be activated or deactivated for specific time periods with the `ACTIVE PERIODS` or `INACTIVE PERIODS` command lines. See Section 2.5 for more information about these optional command lines.

3.12.1.2 Use HHT Integration

```
USE HHT INTEGRATION
```

Adagio currently only supports the HHT algorithm for implicit time integration. The `USE HHT INTEGRATION` command line is provided as a placeholder to allow for other time integrators in the future, but currently this command line has no effect.

3.12.1.3 HHT Parameters

```
ALPHA = <real>alpha(0.0) [DURING <string list>period_names]
BETA = <real>beta(0.5) [DURING <string list>period_names]
GAMMA = <real>gamma(0.25) [DURING <string list>period_names]
```

The `ALPHA`, `BETA`, and `GAMMA` command lines specify the HHT integration parameters. These parameters can all be specified for the entire analysis, or they can vary by solution period by appending them with the optional `DURING` specification (see Section 3.2.1).

- The `ALPHA` command line specifies α, which is the dissipation factor applied to the internal force vector. The value of `alpha` controls numerical damping and must always be less than or equal to zero. Its default value of 0.0 results in no numerical damping. To maintain second-order accuracy, the value of `alpha` should not be less than $-\frac{1}{3}$.
- The `GAMMA` command line specifies the dissipation factor γ for Newmark time integrators. The default value of `gamma` is 0.25. It should have a value of $0.5-\alpha$ to maintain second-order accuracy.
- The `BETA` command line specifies the stability parameter β for time integrators in the Newmark family. The default value of `beta` is 0.5. It should have a value of $0.25(\gamma + 0.5)^2$ to maintain second-order accuracy.

3.12.1.4 Implicit Dynamic Adaptive Time Stepping

```
TIME INTEGRATION CONTROL = <string>ADAPTIVE|COMPUTERESIDUAL|
  IGNORE(IGNORE) [DURING <string list>period_names]
INCREASE ERROR THRESHOLD = <real>increase_threshold(0.02)
```

```
    [DURING <string list>period_names]
  HOLD ERROR THRESHOLD = <real>hold_threshold(0.10)
    [DURING <string list>period_names]
  DECREASE ERROR THRESHOLD = <real>decrease_threshold(0.25)
    [DURING <string list>period_names]
```

Adagio provides an adaptive time stepping capability that adjusts the time step based on the error due to time discretization with implicit time integration. If this capability is activated, after a solution is obtained for each time step, an interpolated solution for the half step is computed based on the assumption that acceleration is constant over the time step. The residual is computed using this interpolated solution. The residual is indicative of the error introduced due to the time step, and is used to adjust the time step. The residual is always evaluated relative to the reference load. The size of the next time step is increased if the residual is low, or decreased if the residual is high.

The command lines for adaptive time stepping can all be optionally used in the `IMPLICIT DYNAMICS` command block. These command lines control the behavior of the implicit time integrator's algorithm for adaptive time stepping. All the parameters specified by these command lines can vary by period by appending the command lines with the optional `DURING` specification (see Section 3.2.1).

- The `TIME INTEGRATION CONTROL` command line controls whether the algorithm for adaptive time stepping should be used. Three options are available: `ADAPTIVE`, `COMPUTERESIDUAL`, and `IGNORE`. If the option is set to `ADAPTIVE`, the midstep residuals are computed and the time step is adjusted based on the residuals. If the option is set to `COMPUTERESIDUAL`, the midstep residual is computed and reported in the log file, but the time step is not adjusted. If the option is set to `IGNORE`, the default, no midstep residuals are computed, and the time step is not adjusted.

- The `INCREASE ERROR THRESHOLD` command line controls when the time step is increased. If the relative midstep residual is below the specified value, the next time step will be increased. The default value of `increase_threshold` is 0.02.

- The `HOLD ERROR THRESHOLD` command line controls when the time step is held constant. If the relative midstep residual is below the hold threshold and greater than the increase threshold, the next time step will be held constant. The default value of `hold_threshold` is 0.10, which must be greater than the value of `increase_threshold`.

- The `DECREASE ERROR THRESHOLD` command line controls when the time step is decreased. If the relative midstep residual is below the decrease threshold and greater than the hold threshold, the current time step is accepted but the next time step will be decreased. If the relative midstep residual is greater than the decrease threshold, the current solution is rejected and recomputed with a smaller time step. The default value of `decrease_threshold` is 0.25, which must be greater than `hold_threshold`.

3.13 References

1. Blanford, M. L., M. W. Heinstein, and S. W. Key. *JAS3D – A Multi-Strategy Iterative Code for Solid Mechanics Analysis Users' Instructions, Release 2.0*, Draft SAND report. Albuquerque, NM: Sandia National Laboratories, September 2001.

2. Fletcher, R., and C. M. Reeves. "Function Minimization by Conjugate Gradients." *The Computer Journal* 7 (1964): 149–154.

3. Farhat, C., M. Lesoinne, and K. Pierson. "A Scalable Dual-Primal Domain Decomposition Method." *Numerical Linear Algebra with Applications* 7 (2000): 687–714.

4. Farhat, C., M. Lesoinne, P. LeTallec, K. Pierson, and D. Rixen. "FETI-DP: A Dual-Primal Unified FETI Method – Part I: A Faster Alternative to the Two-Level FETI Method." *International Journal for Numerical Methods in Engineering* 50 (2001): 1523–1544.

5. Hilber, H. M., T. J. R. Hughes, and R. L. Taylor. "Improved Numerical Dissipation for Time Integration Algorithms in Structural Dynamics." *Earthquake Engineering and Structural Dynamics* 5 (1977): 283–292.

6. Hughes, T. J. R. *The Finite Element Method: Linear Static and Dynamic Finite Element Analysis*. Englewood Cliffs, NJ: Prentice-Hall, 1987.

7. Newmark, N. M. "A Method of Computation for Structural Dynamics." *Journal of the Engineering Mechanics Division*, ASCE 85, no. EM3 (1959): 67–94.

Chapter 4

Materials

This chapter describes material models that can be used in conjunction with the elements in Presto and Adagio. Most of the material models have an interface that allows them to be used by the elements in both codes. Even though a material model can be used by both codes, usage of the model may be better suited for the type of problems solved by one code rather than the type of problems solved by the other code. For example, a material model that was built to characterize behavior over a long time would be better suited for use in Adagio than in Presto. If a particular material model is better suited for one code rather than the other, this usage information is provided in the description of that model.

The material models described in this chapter are, in general, applicable to solid elements. The structural elements, such as shells and beams, have a much more limited set of material models. Chapter 5 describes the element library, including which material models are available for the various elements. The introduction to Chapter 5 summarizes all the element types in Presto and Adagio. For each element type, a list of available material models is provided.

When using the nonlinear material models, you may want to output state variables that are associated with these models. See Section 8.9.2 to learn how to output the state variables for the various nonlinear material models.

Most material models for solid elements are available in two libraries. The newer library is the LAME library [17], but it is not the default. The line command to activate the LAME material library for a particular section is described in Section 5.2.1.

General Model Form. `PROPERTY SPECIFICATION FOR MATERIAL` command blocks appear in the SIERRA scope in the general form shown below.

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

PROPERTY SPECIFICATION FOR MATERIAL command blocks are physics independent in the sense that the information in them can be shared by more than one application. For example, some of the PROPERTY SPECIFICATION FOR MATERIAL command blocks contain density information that can be shared among several applications.

The command block begins with the line:

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

and terminates with the line:

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

Here, the string mat_name is a user-specified name for the command block. This name is typically descriptive of the material being modeled, e.g., aluminum_t6061.

Within a PROPERTY SPECIFICATION FOR MATERIAL command block, there will be other command blocks and possibly other general material command lines that are used to describe particular material models. The general material command lines, if present, are listed first, followed by one or more material-model command blocks. The general material command lines may be used to specify the density of the material, the Biot's coefficient, and the application of temperatures and thermal strains to two- or three-dimensional elements. Each material-model command block follows 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.

As noted above, 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-model command block would provide an elastic-plastic model for steel. The general form of a PROPERTY SPECIFICATION FOR MATERIAL command block would be as follows:

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  #
  # General material command lines
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_coefficient_value
  #
  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 <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`. Typically, however, only one material model would be desired for a given block, and the `PROPERTY SPECIFICATION FOR MATERIAL` command block would have only one `PARAMETERS FOR MODEL` command block. A particular material model may only appear once within a given `PROPERTY SPECIFICATION FOR MATERIAL` command block.

Although multiple material models can be defined for one material within a `PROPERTY SPECIFICATION FOR MATERIAL` command block, only one material model is actually used for a given element block during an analysis. The ability to define multiple constitutive models for one material is provided as a convenience to enable the user to easily switch between models. The material name and the model name are both referenced when material models are assigned to element blocks within the `FINITE ELEMENT MODEL` command block, which is described in Section 5.1.

This chapter is organized to correspond to the general form presented for the `PROPERTY SPECIFICATION FOR MATERIAL` command block. Section 4.1 discusses the `DENSITY` command line, the `BIOTS COEFFICIENT` command line, and the command lines used for thermal strains, and also explains how temperatures and thermal strains are applied. Section 4.2 describes each of the material models that are shared by Presto and Adagio. References applicable for both

Presto and Adagio are listed in Section 4.4.

As indicated in the introductory material, not all the material models available are applicable to all 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 available in Presto 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 indicates 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 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. See Section 8.9.2 to learn how to output the state variables for the nonlinear material models.

Note that 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 for most of the material models shared by Presto and Adagio.

4.1 General Material Commands

A PROPERTY SPECIFICATION FOR MATERIAL command block for a particular material may include additional command lines that are applicable to all the material models specified within the command block. These command lines related to density, to Biot's coefficient, and to thermal strain behavior are discussed, respectively, in Section 4.1.1, Section 4.1.2, and Section 4.1.3.

4.1.1 Density Command

```
        DENSITY = <real>density_value
```

This command line specifies the density of the material described in a PROPERTY SPECIFICATION FOR MATERIAL command block. The units of the input parameter density_value are specified as mass per unit volume.

As previously explained, a PROPERTY SPECIFICATION FOR MATERIAL command block can contain one or more PARAMETERS FOR MODEL command blocks. The specified density_value for the material will be used with all of the models described in these PARAMETERS FOR MODEL command blocks.

4.1.2 Biot's Coefficient Command

```
        BIOTS COEFFICIENT = <real>biots_value
```

This command line specifies the Biot's coefficient of the material. Biot's coefficient is used with the pore pressure capability. See Section 6.8 for more information on pore pressure. If not given, the value defaults to 1.0. This parameter is unitless.

As previously explained, a PROPERTY SPECIFICATION FOR MATERIAL command block can contain one or more PARAMETERS FOR MODEL command blocks. The specified biots_value for the material will be used with all of the models described in these PARAMETERS FOR MODEL command blocks.

4.1.3 Thermal Strain Behavior

Isotropic and orthotropic thermal strains may be defined for use by material models listed in a PROPERTY SPECIFICATION FOR MATERIAL command block. Section 4.1.3.1 describes the command lines that are used to define the thermal strain behavior. These command lines must be used in conjunction with other command blocks outside of a PROPERTY SPECIFICATION FOR MATERIAL command block for the calculations of thermal strains to be activated. Section 4.1.3.2 explains the process for activating thermal strains.

4.1.3.1 Defining 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
```

A `PROPERTY SPECIFICATION FOR MATERIAL` command block may include command lines that define thermal strain behavior. 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, the Y-direction, and the Z-direction. For the anisotropic 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 the `THERMAL STRAIN X FUNCTION` command line is applied to the strain in the shell (or membrane) r-direction. (Reference Section 5.2.4 for a discussion of the shell rst coordinate system.) The thermal strain corresponding to the `THERMAL STRAIN Y FUNCTION` command line 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, e.g., XY, YZ, or ZX. The current orthotropic thermal-strain capability has limited use for shells and membranes in the current release of the code. Tying the orthotropic thermal-strain functionality to the shell orientation functionality (Section 5.2.4) 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 `PROPERTY SPECIFICATION FOR MATERIAL` command block, outside of the specifications of any material models in the block. Such placement is necessary because the isotropic thermal strain is a general material property, not a property that is specific to any particular constitutive model, such as `ELASTIC` or `ELASTIC-PLASTIC`. The input value of `thermal_strain_function` is the name of the function that defines thermal strain as a function of temperature for the material model described in this particular `PROPERTY SPECIFICATION FOR MATERIAL` command block. The function is defined within the SIERRA scope using a `DEFINITION FOR FUNCTION` command block. For more information on how to set the input to compute thermal strains and how to apply temperatures, see Section 4.1.3.2.

The specification of an orthotropic thermal-strain field requires that all three of the `THERMAL STRAIN X FUNCTION`, `THERMAL STRAIN Y FUNCTION`, and `THERMAL STRAIN Z FUNCTION` command lines be placed in the `PROPERTY SPECIFICATION FOR MATERIAL` command block. All three command lines must be provided, even when there is no thermal strain in one or more directions. The values of `thermal_strain_x_function`, `thermal_strain_y_function`, and `thermal_strain_z_function` are the names of the functions for thermal strains in the *X*-direction, the *Y*-direction, and the *Z*-direction, respectively. These functions are defined within the SIERRA scope using `DEFINITION FOR FUNCTION` command blocks. To specify that there should be no thermal strain in a given direction, use a function that always evaluates to zero for that direction.

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 several of the material models, as discussed in Section 4.1.3.2. Note that specification of a thermal strain is identified in the descriptions of the material models in Section 4.2 by the notation "`thermal strain option`".

4.1.3.2 Activating Thermal Strains

Adagio 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 SIERRA 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, the material models used by blocks that experience thermal strain must have their thermal strain behavior defined. The command lines for defining isotropic and orthotropic thermal strain are described in Section 4.1.3.1. Materials with isotropic thermal strain use the `THERMAL STRAIN FUNCTION` command line, while those with orthotropic thermal strain must define thermal strain in all three directions using the `THERMAL STRAIN X FUNCTION`, `THERMAL STRAIN Y FUNCTION`, and `THERMAL STRAIN Z FUNCTION` command lines. These inputs can be used with all material models with the exception of the following: elastic three-dimensional orthotropic, elastic laminate, Mooney-Rivlin, NLVE three-dimensional orthotropic, Swanson, and viscoelastic Swanson. These models require their own model-specific inputs to define thermal strain and must not use these standard commands. Information for defining thermal strains is provided in the individual descriptions of these models in Section 4.2.

- Third, a temperature field must be applied to the affected blocks. The command block to specify the application of temperatures is `PRESCRIBED TEMPERATURE`, which is implemented as a standard boundary condition. A description of the `PRESCRIBED TEMPERATURE` command block is given in Section 6.7.

Whenever a temperature field is applied, the temperature is prescribed at the nodes, but thermal strain is computed based on element temperature. Element temperatures are obtained by averaging the temperatures of the nodes connected to a given element. Thermal strains are applied in rate form, so the thermal strain in an element is relative to the thermal strain at the initial temperature. Thus, the initial temperature is the stress-free temperature. If desired, a different stress-free temperature can be used by prescribing the initial temperature with the `INITIAL CONDITION` command block as described in Section 6.2.

4.2 Material Models

This section contains descriptions of the material models that are shared by Presto and Adagio.

4.2.1 Elastic Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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.

The command block for an elastic material starts with the line:

```
BEGIN PARAMETERS FOR MODEL ELASTIC
```

and terminates with the line:

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

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.

- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.

- Any 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.

There are no output variables available for the elastic model. For information about the elastic model, consult Reference 1.

4.2.2 Thermoelastic Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 THERMOELASTIC
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    YOUNGS MODULUS FUNCTION = <string>ym_function_name
    POISSONS RATIO FUNCTION = <string>pr_function_name
  END [PARAMETERS FOR MODEL THERMOELASTIC]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The thermoelastic material model is used to describe the temperature-dependent linear elastic behavior of materials. This model is generally valid for small deformations.

The command block for a thermoelastic material starts with the line:

```
BEGIN PARAMETERS FOR MODEL THERMOELASTIC
```

and terminates with the line:

```
END [PARAMETERS FOR MODEL THERMOELASTIC]
```

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.

- Any 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 `YOUNGS MODULUS FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope that describes a scale factor on 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 SIERRA scope that describes a scale factor on Poisson's ratio as a function of temperature.

There are no output variables available for the thermoelastic model. For information about the thermoelastic model, consult Reference 1.

4.2.3 Neo-Hookean Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 NEO_HOOKEAN
    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 NEO_HOOKEAN]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The neo-Hookean material model is used to describe linear elastic behavior of materials. Unlike the elastic model, this model is valid for both large and small deformations. The neo-Hookean model uses a strain energy density, which makes it a hyperelastic constitutive model. This feature makes it applicable to finite strains. Currently the neo-Hookean model, as implemented in LAME, does not support thermal strains.

The command block for a neo-Hookean material starts with the line:

```
BEGIN PARAMETERS FOR MODEL NEO_HOOKEAN
```

and terminates with the line:

```
END [PARAMETERS FOR MODEL NEO_HOOKEAN]
```

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.

- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.

- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.

- Any 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.

There are no output variables available for the neo-Hookean model. For information about the neo-Hookean model, consult Reference 5.

4.2.4 Elastic Fracture Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 CRACK OPENING 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 that is 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.

The command block for an elastic fracture material starts with the line:

```
BEGIN PARAMETERS FOR MODEL ELASTIC_FRACTURE
```

and terminates with the line:

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

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.

- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.

- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.

- Any 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 CRACK OPENING STRAIN` command line. This component of strain is aligned with the maximum-principal-stress direction at failure.

Output variables available for this model are listed in Table 8.18.

4.2.5 Elastic-Plastic Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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, typically metals, that 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, the model also has the ability to define a yield surface that not only grows, or hardens, but also moves in stress space. This ability 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.

The command block for an elastic-plastic material starts with the line:

```
BEGIN PARAMETERS FOR MODEL ELASTIC_PLASTIC
```

and terminates with the line:

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

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.
- Any 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.

Output variables available for this model are listed in Table 8.19. For information about the elastic-plastic model, consult Reference 1.

4.2.6 Elastic-Plastic Power-Law Hardening Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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.

The command block for an elastic-plastic power-law hardening material starts with the line:

```
BEGIN PARAMETERS FOR MODEL EP_POWER_HARD
```

and terminates with the line:

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

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.

- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.

- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on specifying and applying thermal strains and temperatures.

- Any 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.

Output variables available for this model are listed in Table 8.20. For information about the elastic-plastic power-law hardening model, consult Reference 1.

4.2.7 Ductile Fracture Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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>critical_strain
  END [PARAMETERS FOR MODEL DUCTILE_FRACTURE]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

This model is identical to the elastic-plastic power-law hardening model with the addition of a failure criterion and a post-failure 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

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

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.

The command block for an elastic-plastic power-law hardening material with failure starts with the line:

```
BEGIN PARAMETERS FOR MODEL DUCTILE_FRACTURE
```

and terminates with the line:

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

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains and temperatures.
- Any 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.

Output variables available for this model are listed in Table 8.21. For information about the ductile fracture material model, consult Reference 1.

4.2.8 Multilinear Elastic-Plastic Hardening Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 MULTILINEAR_EP
    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 = <string>hardening_function_name
    YOUNGS MODULUS FUNCTION = <string>ym_function_name
    POISSONS RATIO FUNCTION = <string>pr_function_name
    YIELD STRESS FUNCTION = <string>yield_stress_function_name
  END [PARAMETERS FOR MODEL MULTILINEAR_EP]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

This model is similar to the power-law hardening model except that the hardening behavior is described with a piecewise-linear curve as opposed to a power law.

The command block for a multi-linear elastic-plastic hardening material starts with the line:

```
BEGIN PARAMETERS FOR MODEL MULTILINEAR_EP
```

and terminates with the line:

```
END [PARAMETERS FOR MODEL MULTILINEAR_EP]
```

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.

- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.

- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.

- Any 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 SIERRA scope that describes the hardening behavior of the material as a stress versus equivalent plastic strain. This curve is expressed as the additional increment of stress over the yield stress versus equivalent plastic strain, thus the first point on the curve should be (0.0, 0.0).

- The `YOUNGS MODULUS FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope that describes a scale factor on 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 SIERRA scope that describes a scale factor on 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 SIERRA scope that describes a scale factor on the yield stress as a function of temperature.

Output variables available for this model are listed in Table 8.22.

4.2.9 Multilinear Elastic-Plastic Hardening Model with Failure

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 = <string>hardening_function_name
    YOUNGS MODULUS FUNCTION = <string>ym_function_name
    POISSONS RATIO FUNCTION = <string>pr_function_name
    YIELD STRESS FUNCTION = <string>yield_stress_function_name
    CRITICAL TEARING PARAMETER = <real>crit_tearing
    CRITICAL CRACK OPENING STRAIN = <real>critical_strain
  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 that the hardening behavior is described with a piecewise-linear curve as opposed to a power law. This model incorporates a failure criterion and a post-failure 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:

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

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.

The command block for a multi-linear elastic-plastic hardening material with failure starts with the line:

```
    BEGIN PARAMETERS FOR MODEL ML_EP_FAIL
```

and terminates with the line:

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

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.
- Any 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 SIERRA scope that describes the hardening behavior of the material as a stress versus equivalent plastic strain. This curve is expressed as the additional increment of stress over the yield stress versus equivalent plastic strain, thus the first point on the curve should be (0.0, 0.0).

- The `YOUNGS MODULUS FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope that describes a scale factor on 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 SIERRA scope that describes a scale factor on 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 SIERRA scope that describes a scale factor on 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.

Output variables available for this model are listed in Table 8.23.

4.2.10 Johnson-Cook Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 JOHNSON_COOK
    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
    RHOCV = <real>rho_cv
    RATE CONSTANT = <real>rate_constant
    THERMAL EXPONENT = <real>thermal_exponent
    REFERENCE TEMPERATURE = <real>reference_temperature
    MELT TEMPERATURE = <real>melt_temperature
  END [PARAMETERS FOR MODEL JOHNSON_COOK]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The Johnson-Cook model is used to model materials, typically metals, undergoing plastic deformation at finite strains. The hardening function is defined by:

$$\sigma = (\sigma_y + B(\bar{\epsilon}_p)^N)(1 + C\ln(e^*))(1 - (T^*)^M)\,, \tag{4.5}$$

where σ_y is the yield stress, B is the hardening constant, $\bar{\epsilon}_p$ is the equivalent plastic strain, N is the hardening exponent, C is the rate constant, e^* is the non-dimensional effective total strain rate, and T^* is the homologous temperature. T^* is defined as:

$$T^* = (T - T_{ref})/(T_{melt} - T_{ref})\,, \tag{4.6}$$

where T is the current temperature, T_{ref} is the reference temperature, and T_{melt} is the melt temperature. In the case where $M <= 0$, $(1 - (T^*)^M) = 1$.

Plastic work results in adiabatic heating. The resulting change in temperature is computed according to the function:

$$\Delta T = \frac{0.95\sigma_y \Delta\bar{\epsilon}_p}{\rho c_v}, \tag{4.7}$$

where ρ is the material density and c_v is the specific heat.

For more information about the Johnson-Cook material model, consult Reference 22.

In the command blocks that define the Johnson-Cook model:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.
- Any 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 product ρc_v is defined with the `RHOCV` command line.
- The thermal exponent is defined with the `THERMAL EXPONENT` command line.
- The reference temperature is defined with the `REFERENCE TEMPERATURE` command line.
- The melt temperature is defined with the `MELT TEMPERATURE` command line.

4.2.11 BCJ Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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, the `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains. 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. Note that the parameters for the `SPECIFIC HEAT`, `THETA OPT`, `FACTOR`, `RHO`, and `TEMP0` command lines are used 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.

4.2.12 Power Law Creep

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 POWER_LAW_CREEP
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    CREEP CONSTANT = <real>creep_constant
    CREEP EXPONENT = <real>creep_exponent
    THERMAL CONSTANT = <real>thermal_constant
  END [PARAMETERS FOR MODEL POWER_LAW_CREEP]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The power law creep model is a secondary creep model that can be used to model the time-dependent behavior of metals, brazes or solders at high homologous temperatures. It can also be used as a simple model for the time-dependent behavior of geologic materials such as salt.

In the power law creep model, the effective creep strain rate is related to the effective stress raised to a power

$$\dot{\bar{\varepsilon}}^c = A\bar{\sigma}^m \exp\left(\frac{-Q}{RT}\right), \tag{4.8}$$

where $\dot{\bar{\varepsilon}}^c$ is the effective creep strain rate, $\bar{\sigma}$ is the effective stress, A is the creep constant, m is the creep exponent, Q is the activation energy, R is the universal gas constant (1.987 cal/mole K), and T is the absolute temperature.

If an analysis is run isothermally, then the `THERMAL CONSTANT` is simply Q/RT for the given temperature. If the analysis is a thermal analysis, then the `THERMAL CONSTANT` is Q/R, and T is in general a function of space and time.

The command block for a power law creep material starts with the line:

```
BEGIN PARAMETERS FOR MODEL POWER_LAW_CREEP
```

and terminates with the line:

```
    END [PARAMETERS FOR MODEL POWER_LAW_CREEP]
```

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.
- Any 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 creep constant, A, in Equation (4.8) is defined with the `CREEP CONSTANT` command line.
- The creep exponent, m, in Equation (4.8) is defined with the `CREEP EXPONENT` command line.
- The thermal constant, the particular form depending on if the analysis is isothermal or not, is defined with the `THERMAL CONSTANT` command line.

Output variables available for this model are listed in Table 8.35.

4.2.13 Soil and Crushable Foam Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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:

$$\sigma_{yd} = a_0 + a_1 p + a_2 p^2 , \tag{4.9}$$

where p is positive in compression. The determination of the yield stress from Equation (4.9) 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 coefficients in Equation (4.9) 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:

$$\phi_p = p - f_p(\varepsilon_V) \ , \tag{4.10}$$

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

The command block for a soil and crushable foam material starts with the line:

```
BEGIN PARAMETERS FOR MODEL SOIL_FOAM
```

and terminates with the line:

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

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.
- Any 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, see the PRONTO3D document listed as Reference 6. The soil and crushable foam model is the same as the soil and crushable foam model in PRONTO3D. The PRONTO3D model is based on a material model developed by Krieg [7]. The Krieg version of the soil and crushable foam model was later modified by Swenson and Taylor [8]. The soil and crushable foam model developed by Swenson and Taylor is the model in PRONTO3D and is also the shared model for Presto and Adagio.

Output variables available for this model are listed in Table 8.37.

4.2.14 Karagozian and Case Concrete Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
   DENSITY = <real>density_value
   BIOTS COEFFICIENT = <real>biots_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 KC_CONCRETE
      YOUNGS MODULUS = <real>youngs_modulus
      POISSONS RATIO = <real>poissons_ratio
      BULK MODULUS = <real>bulkmodulus
      SHEAR MODULUS = <real>shearmodulus
      COMPRESSIVE STRENGTH = <real>compressive_strength
      TENSILE STRENGTH = <real>tensile_strength
      LAMBDA = <real>lambda
      LAMBDAM = <real>lambda_m
      LAMBDAZ = <real>lambda_z
      SINGLE RATE ENHANCEMENT = <enum>TRUE|FALSE
      FRACTIONAL DILATANCY = <real>omega
      MAXIMUM AGGREGATE SIZE = <real>max_aggregate_size
      ONE INCH = <real>one_inch
      RATE SENSITIVITY FUNCTION = <string>rate_function_name
      PRESSURE FUNCTION = <string>pressure_function_name
      UNLOAD BULK MODULUS FUNCTION = <string>bulk_function_name
      HARDEN-SOFTEN FUNCTION = <string>harden_soften_function_name
   END PARAMETERS FOR MODEL KC_CONCRETE
END PROPERTY SPECIFICATION FOR MATERIAL name
```

The Karagozian & Case (or K&C) concrete model is an inelasticity model appropriate for approximating the constitutive behavior of concrete. Coupled with appropriate elements for capturing the embedded deformation of reinforcing steel, the K&C concrete model can be utilized effectively for simulating the mechanical response of reinforced concrete structures. The K&C model has several useful characteristics for estimating concrete response, including strain-softening capabilities, some degree of tensile response, and a nonlinear stress-strain characterization that robustly simulates the behavior of plain concrete. This model is described in detail in Reference 9.

The command block for the K&C concrete material model starts with the line:

```
BEGIN PARAMETERS FOR MODEL KC_CONCRETE
```

and terminates with the line:

```
    END PARAMETERS FOR MODEL KC_CONCRETE
```

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.
- Any 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.

In addition to these material moduli, the following constitutive parameters should be defined:

- The compressive strength for a uniaxial compression test is defined with the `COMPRESSIVE STRENGTH` command line.
- The tensile strength for the uniaxial tension test is defined with the `TENSILE STRENGTH` command line.
- The abscissa of the hardening/softening curve where this curve takes on the value of one is termed Lambda-M, and it is defined with the `LAMBDAM` command line (Reference 9, pg. B-3).
- The abscissa of the hardening/softening curve where this curve takes on the value of zero after its peak value has been attained is termed Lambda-Z, and it is defined with the `LAMBDAZ` command line. This parameter should satisfy LAMBDAZ > LAMBDAM (Reference 9, pg. B-3). This input is Sierra-specific, and differs from the previous PRONTO3D definitions.
- The `SINGLE RATE ENHANCEMENT` parameter indicates whether the rate enhancement of the model should be independent of the sign of the deformation. If this parameter is set to `TRUE`, the same enhancement function is used for both compression and tension. If it is set to `FALSE`, the enhancement function must assign values for both positive and negative values of strain rate (Reference 9, pg. B-5). This parameter is also Sierra-specific, and is different from the previous PRONTO3D definitions.

- The `FRACTIONAL DILATANCY` is an estimate of the size of the plastic volume strain increment relative to that corresponding to straining in the hydrostatic plane. This value normally ranges from 0.3 to 0.7, and a value of one-half is commonly utilized in practice.

- The `MAXIMUM AGGREGATE SIZE` parameter provides an estimate of the largest length dimension for the aggregate component of the concrete mix. The American Concrete Institute code [10] includes specifications for maximum aggregate size that are based on member depth and clear spacing between adjacent reinforcement elements.

- The parameter `ONE INCH` provides for conversion to units other than the pounds/inch system commonly used in U.S. concrete venues. This parameter should be set to the equivalent length in the system utilized for analysis, e.g., if centimeters are used, then this parameter is set to 2.54.

The following functions describe the evolution of material coefficients in this model:

- The function characterizing the enhancement of strength with strain rate is described via the `RATE SENSITIVITY FUNCTION` (Reference 9, pg. B-3).

⚠ **Warning:** The `RATE SENSITIVITY FUNCTION` command should be used with caution. The implementation appears to provide unconservative estimates of concrete strength in tension, and users are cautioned to provide rate sensitivity function values that have the value of 1.0 for positive (tensile) values of strain rate. These values correspond to no additional strength in tension due to strain rate, and are both physically realistic and properly conservative.

- The function describing the relationship between pressure and volumetric strain is described via the `PRESSURE FUNCTION`.

- The function characterizing the relationship between bulk modulus and volumetric strain during unloading is described via the `UNLOAD BULK MODULUS FUNCTION`.

- The function describing the hardening and softening functions function eta as a function of the material parameters lambda (see `LAMBDAM` and `LAMBDAZ`) is defined via the `HARDEN-SOFTEN FUNCTION` (Reference 9, pg. B-3).

4.2.15 Foam Plasticity Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 (including polyurethane, polystyrene bead, etc.) subjected to large deformation. This model is not appropriate for flexible foams that return to their undeformed shape after loads are removed.

The command block for a foam plasticity material starts with the line:

```
BEGIN PARAMETERS FOR MODEL FOAM_PLASTICITY
```

and terminates with the line:

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

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.
- Any 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 is defined with the `HYDRO HARDENING` command line.

- The hydrodynamic hardening exponent 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 between 0.0 and 0.95 are recommended.

Output variables available for this model are listed in Table 8.24.

4.2.16 Low Density Foam Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 LOW_DENSITY_FOAM
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    A = <real>A
    B = <real>B
    C = <real>C
    NAIR  = <real>NAir
    P0  = <real>P0
    PHI = <real>Phi
  END [PARAMETERS FOR MODEL LOW_DENSITY_FOAM]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The `LOW_DENSITY_FOAM` material model is a phenomenological model for low density polyurethane foams, where `A`, `B`, and `C` are material constants, `NAIR` is the mole fraction of air, `P0` is the initial air pressure, and `PHI` is the volume fraction of solid material. The yield function for this model has the form

$$\bar{\sigma} = \mathrm{A}\left\langle \mathrm{I}_2' \right\rangle + \mathrm{B}\left(1.0 + \mathrm{C}\mathrm{I}_1\right), \tag{4.11}$$

where $\langle\rangle$ denotes the Heaviside step function, I_1 is the first invariant of the deviatoric strain, and I_2' is the second invariant of the strain.

For more information on the low density foam material model, see [11].

4.2.17 Elastic Three-Dimensional Orthotropic Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_value
  BEGIN PARAMETERS FOR MODEL ELASTIC_3D_ORTHOTROPIC
    # general parameters (any two are required)
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    # required parameters
    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.

The command block for an elastic three-dimensional orthotropic material starts with the line:

```
BEGIN PARAMETERS FOR MODEL ELASTIC_3D_ORTHOTROPIC
```

and terminates with the line:

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

In the above command blocks all of the following are required inputs, including two of the five general elastic material constants:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- Any 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 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 using 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 using 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 13 for more information about the elastic three-dimensional orthotropic model.

4.2.18 Wire Mesh Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 WIRE_MESH
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    YIELD FUNCTION = <string>yield_function
    TENSION = <real>tensile_strength
  END [PARAMETERS FOR MODEL WIRE_MESH]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The wire mesh constitutive model was developed at Sandia National Laboratories to model layers of mesh for analyses of packaging for transportation of hazardous materials.

The wire mesh model decomposes the Cauchy stress tensor into its principal components. It then checks each of the principal stress to see whether they exceed the yield criterion. If a principal stress is negative, i.e. in compression, the yield condition is

$$\psi = \sigma - f(e_v) \tag{4.12}$$

The yield function for this model can be input by the user, and defines the yield strength as a function of the volumetric engineering strain, e_v.

If the principal stress is tensile, a cutoff value of the principal tensile stress is used, and the yield condition is

$$\psi = \sigma - \tau \tag{4.13}$$

The value for τ is given by the value of `TENSION`.

The command block for a wire mesh material starts with the line:

```
    BEGIN PARAMETERS FOR MODEL WIRE_MESH
```

and terminates with the line:

```
    END [PARAMETERS FOR MODEL WIRE_MESH]
```

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.
- Any 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 function in compression is defined with the `YIELD FUNCTION` command line.
- The tensile strength is give by the `TENSION` command line.

Output variables available for this model are listed in Table 8.25.

More information on the model can be found in the report by Neilsen, et. al. [12]

4.2.19 Orthotropic Crush Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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
    EX  = <real>modulus_x_pre_lockup
    EY  = <real>modulus_y_pre_lockup
    EZ  = <real>modulus_z_pre_lockup
    GXY = <real>shear_modulus_xy_pre_lockup
    GYZ = <real>shear_modulus_yz_pre_lockup
    GZX = <real>shear_modulus_zx_pre_lockup
    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
    VMIN = <real>lockup_volumetric_strain
    YOUNGS MODULUS = <real>youngs_modulus_post_lockup
    POISSONS RATIO = <real>poissons_ratio_post_lockup
    SHEAR MODULUS = <real>shear_modulus_post_lockup
    BULK MODULUS = <real>bulk_modulus_post_lockup
    LAMBDA = <real>lambda_post_lockup
    YIELD STRESS = <real>yield_stress_post_lockup
  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 [14]. The orthotropic crush model divides material behavior into three phases:

- orthotropic elastic,
- volumetric crush (partially compacted), and

- elastic–perfectly plastic (fully compacted).

The command block for an orthotropic crush material starts with the line:

```
BEGIN PARAMETERS FOR MODEL ORTHOTROPIC_CRUSH
```

and terminates with the line:

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

In the above command blocks:

- The uncompacted density of the material is defined with the `DENSITY` command line.
- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.
- 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 compressive volumetric strain at lockup or full compaction is defined with the `VMIN` command line.
- The directional stress σ_{xx} as a function of the compressive volumetric strain is defined by the function referenced in the `CRUSH XX` command line.
- The directional stress σ_{yy} as a function of the compressive volumetric strain is defined by the function referenced in the `CRUSH YY` command line.
- The directional stress σ_{zz} as a function of the compressive volumetric strain is defined by the function referenced in the `CRUSH ZZ` command line.
- The directional stress σ_{xy} as a function of the compressive volumetric strain is defined by the function referenced in the `CRUSH XY` command line.
- The directional stress σ_{yz} as a function of the compressive volumetric strain is defined by the function referenced in the `CRUSH YZ` command line.
- The directional stress σ_{zx} as a function of the compressive volumetric strain is defined by the function referenced in the `CRUSH ZX` command line.

- Any 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.

Note that several of the command lines in this command block (those beginning with `CRUSH`) reference functions. These functions must be defined in the SIERRA scope. Output variables available for this model are listed in Table 8.32. For information about the orthotropic crush model, consult Reference 14.

4.2.20 Orthotropic Rate Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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]
```

The orthotropic rate model extends the functionality of the orthotropic crush constitutive model described in Section 4.2.19. The orthotropic rate model has been developed to describe the behavior of an aluminum honeycomb subjected to large deformation. The orthotropic rate model, like the original orthotropic crush model, has six independent yield functions that evolve with volume strain. Unlike the orthotropic crush model, the orthotropic rate model has yield functions that also depend on strain rate. The orthotropic rate 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 orthotropic crush model. A honeycomb orientation capability is included with the orthotropic rate model that allows users to prescribe initial honeycomb orientations that are not aligned with the original global coordinate system.

The command block for an orthotropic rate material starts with the line:

```
BEGIN PARAMETERS FOR MODEL ORTHOTROPIC_RATE
```

and terminates with the line:

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

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.
- 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 of `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 of `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 compressive volumetric strain is given by the `T FUNCTION` command line.

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

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

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

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

- The function describing the WT-normal strength of the honeycomb as a function of compressive volumetric strain 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 SIERRA scope. Output variables available for this model are listed in Table 8.33.

4.2.21 Elastic Laminate Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  BEGIN PARAMETERS FOR MODEL ELASTIC_LAMINATE
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    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 15 and 16.

The command block for an elastic laminate material starts with the line:

```
BEGIN PARAMETERS FOR MODEL ELASTIC_LAMINATE
```

and terminates with the line:

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

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- Two of the five elastic constants must be defined with the `YOUNGS MODULUS`, `POISSONS RATIO`, `SHEAR MODULUS`, `BULK MODULUS`, or `LAMBDA` commands
- The elastic constants are unrelated to the laminate stiffness matrix. They are used along with the shell section thickness (defined in the shell section command block, see section 5.2.4) to calculate drilling and hourglass stiffnesses. Otherwise these values have no physical significance in the elastic laminate material model.
- 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 by functions that are referenced on the `NTH11 FUNCTION`, `NTH22 FUNCTION`, and `NTH12 FUNCTION` command lines.
- The thermal-force-couple resultants are defined by functions that are referenced on the `MTH11 FUNCTION`, `MTH22 FUNCTION`, and `MTH12 FUNCTION` command lines.

4.2.22 Fiber Membrane 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 FIBER_MEMBRANE
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    CORD DENSITY = <real>cord_density
    CORD DIAMETER = <real>cord_diameter
    MATRIX DENSITY = <real>matrix_density
    TENSILE TEST FUNCTION = <string>test_function_name
    PERCENT CONTINUUM = <real>percent_continuum
    EPL = <real>epl
    AXIS X = <real>axis_x
    AXIS Y = <real>axis_y
    AXIS Z = <real>axis_z
    MODEL = <string>RECTANGULAR
    STIFFNESS SCALE = <real>stiffness_scale
    REFERENCE STRAIN = <real>reference_strain
  END [PARAMETERS FOR MODEL FIBER_MEMBRANE]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The fiber membrane model is used for modeling membranes that are reinforced with unidirectional fibers. Through the use of a non-zero `PERCENT CONTINUUM`, a background isotropic material response can also be incorporated and is added in a manner such that the response in the fiber direction is unchanged. The fiber membrane model can be used in both Presto and Adagio. When the fiber membrane model is used in Adagio, the model can be used with or without the control-stiffness option in Adagio's multilevel solver. The control-stiffness option is implemented via the `CONTROL STIFFNESS` command block and is discussed in Chapter 3. If the control-stiffness option is activated in Adagio, the response in the fiber direction is softened by lowering the fiber response. In all cases, the final material behavior that is used for equilibrium corresponds to the real material response. When the fiber membrane model is used in Presto, the fiber scaling, which is controlled by the `STIFFNESS SCALE` command line, is ignored.

The command block for a fiber membrane material starts with the line:

```
    BEGIN PARAMETERS FOR MODEL FIBER_MEMBRANE
```

and terminates with the line:

```
    END [PARAMETERS FOR MODEL FIBER_MEMBRANE]
```

In the above command blocks, the following definitions are applicable. Usage requirements are identified both in this list of definitions and in the discussion that follows the list.

- The density of the material is defined with the `DENSITY` command line. This command line should be included, but its value will be recomputed (and hence replaced) if the `CORD DENSITY`, `CORD DIAMETER`, and `MATRIX DENSITY` command lines are specified.
- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.
- Any two of the following elastic constants are required. These are used to compute values for the elastic preconditioner only.
  - 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 density of the fibers is defined by the `CORD DENSITY` command line. This command line is optional. See the usage discussion below.
- The diameter of the fibers is defined by the `CORD DIAMETER` command line. This command line is optional. See the usage discussion below.
- The density of the matrix is defined by the `MATRIX DENSITY` command line. This command line is optional. See the usage discussion below.
- The `TENSILE TEST FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope that describes the fiber force versus strain data. This command line must be included.
- The fractional fiber stiffness to use in defining the background isotropic response is given by the `PERCENT CONTINUUM` command line. This command line must be included.
- The number of fibers per unit length is defined by the `EPL` command line. This command line must be included.

- The components of the vector defining the initial fiber direction is given by the `AXIS X`, `AXIS Y`, and `AXIS Z` command lines. These command lines must be included. See the usage discussion below.

- The coordinate system for specifying the fiber orientation is given by the `MODEL` command line. Only the option `RECTANGULAR` is available in this release. This command line must be included. See the usage discussion below.

- The fiber scaling is specified by the `STIFFNESS SCALE` command line. If the control-stiffness option is used in Adagio, this command line must be included. When the fiber membrane model is used in Presto, this command line is ignored.

- The reference strain is defined with the `REFERENCE STRAIN` command line. This command line is optional for Adagio and is not used in Presto. If the control-stiffness option is used in Adagio, this command line may be included. See the usage discussion below.

Certain command lines in the `PARAMETERS FOR MODEL FIBER_MEMBRANE` command block also have interdependencies or other factors that may impact their usage in Presto and Adagio, as discussed below.

The `CORD DENSITY`, `CORD DIAMETER`, and `MATRIX DENSITY` command lines are optional. When included, these three command lines are used for computation of the correct density corresponding to the fibers, the number of fibers per unit length, and the chosen matrix. When these three command lines are not included, the density is taken as that specified by the `DENSITY` command line.

The `AXIS X`, `AXIS Y`, and `AXIS Z` command lines must be specified if the value for the `MODEL` command line is `RECTANGULAR`. Currently, these axis-related command lines must be specified.

Specifying a reference strain (via the `REFERENCE STRAIN` command line) implies the use of strains for measuring part of the control-stiffness material constraint violation in Adagio. If this command line is not present, the material constraint violation is determined by comparing the change in the scaled fiber force over the current model problem.

4.2.23 Incompressible Solid Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 INCOMPRESSIBLE_SOLID
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    BULK MODULUS = <real>bulk_modulus
    SHEAR MODULUS = <real>shear_modulus
    LAMBDA = <real>lambda
    K SCALING = <real>k_scaling
    2G SCALING = <real>2g_scaling
    TARGET E = <real>target_e
    MAX POISSONS RATIO = <real>max_poissons_ratio
    REFERENCE STRAIN = <real>reference_strain
    SCALING FUNCTION = <string>scaling_function_name
  END [PARAMETERS FOR MODEL INCOMPRESSIBLE_SOLID]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The incompressible solid model is a variation of the elastic model and can be used in both Presto and Adagio. In Adagio, the incompressible solid model is used with the control-stiffness option in the multilevel solver. The control-stiffness option is implemented via the `CONTROL STIFFNESS` command block and is discussed in Chapter 3. The model is used to model nearly incompressible materials where Poisson's ratio, ν, ≈ 0.5. In the course of solving a series of model problems in Adagio, the material response from this model incorporates scaling the bulk and/or shear behaviors to yield a material response that is more amenable to solution using Adagio's conjugate gradient solver. The final material behavior that is calculated corresponds to the actual moduli that are specified. When this model is used in Presto, the material scalings are ignored, and the model behaves like a linear elastic model.

The command block for an incompressible solid material starts with the line:

```
BEGIN PARAMETERS FOR MODEL INCOMPRESSIBLE_SOLID
```

and terminates with the line:

```
    END [PARAMETERS FOR MODEL INCOMPRESSIBLE_SOLID]
```

In the above command blocks, the following definitions are applicable. Usage requirements are identified both in this list of definitions and in the discussion that follows the list.

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.
- Any two of the following elastic constants are required to define the unscaled material response:
  - 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 following material-scaling command lines are used only in Adagio:
  - The nominal bulk scaling is defined with the `K SCALING` command line. This command line is optional. See the usage discussion below.
  - The nominal shear scaling is defined with the `2G SCALING` command line. This command line is optional. See the usage discussion below.
  - The target Young's modulus is defined with the `TARGET E` command line. This command line is optional. See the usage discussion below.
  - The maximum Poisson's ratio is defined with the `MAX POISSONS RATIO` command line. This command line is optional. See the usage discussion below.
  - The reference strain is defined with the `REFERENCE STRAIN` command line. This command line is optional. See the usage discussion below.
  - The `SCALING FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope that describes the time dependent scaling to be applied. This command line is optional. See the usage discussion below.

As noted previously, only two of the elastic constants are required to define the unscaled material response. This requirement applies to use of the incompressible solid model in Presto and in Adagio. Further, all the material-scaling command lines are only used in Adagio.

Several options exist for defining the bulk and/or shear scalings that can be used with the multilevel solver in Adagio.

- Option 1: You can provide the scalings directly by including both of the K SCALING and 2G SCALING command lines or either of them. When both command lines are input, the user-specified values for their parameters will be used. If only the K SCALING command line is input, the bulk scaling is as specified in the k_scaling parameter, and the value of the shear scaling parameter, 2g_scaling, is set to 1.0. On the other hand, if only the 2G SCALING command line is input, then the shear scaling is as specified in the 2g_scaling parameter, but the value of the bulk-scaling parameter, k_scaling, is not set to 1.0. Instead, the bulk scaling is determined by computing a scaled bulk modulus from the scaled shear modulus and a (scaled) Poisson's ratio of 0.3. Then, the bulk scaling is determined simply as the ratio of the scaled bulk modulus to the actual bulk modulus.

- Option 2: You can specify either or both of the TARGET E and MAX POISSONS RATIO command lines to define the scalings. If only the TARGET E command line is included, the bulk and shear scalings are computed by first finding scaled moduli using the value of the target_e parameter along with a (scaled) Poisson's ratio of 0.3. The bulk and shear scalings are then determined as the ratio of the appropriate scaled to unscaled modulus. If only the MAX POISSONS RATIO command line is included, the shear scaling is set to 1.0, and the bulk scaling is computed by first calculating a scaled bulk modulus from the actual shear modulus and the value of the max_poissons_ratio parameter. The bulk scaling is then calculated simply as the ratio of the scaled bulk modulus to the actual bulk modulus. If both the TARGET E and MAX POISSONS RATIO command lines are included, the bulk scaling (and shear scaling) is determined from the ratio of the bulk scaled modulus (and shear scaled modulus) computed using the values of the target_e and max_poissons_ratio parameters to the unscaled bulk (and shear) modulus.

- Option 3: You can choose not to include any of the K SCALING, 2G SCALING, TARGET E, and MAX POISSONS RATIO command lines. In such case, the shear scaling is set to 1.0, and the bulk scaling is computed as the ratio of the scaled bulk modulus coming from the real shear modulus and a (scaled) Poisson's ratio of 0.3 to the actual bulk modulus.

The function referenced by the value of the parameter scaling_function_name in the SCALING FUNCTION command line can be used to modify the bulk and shear scalings in solution time. The actual scalings used are computed by taking the scalings specified by the parameter values in the K SCALING, 2G SCALING, TARGET E, and MAX POISSONS RATIO command lines and simply multiplying them by the function value at the specified solution time. If the SCALING FUNCTION command line is not included, the bulk and shear scalings are fixed in time.

The REFERENCE STRAIN command line supplies a value for the reference strain that is used to create a normalized material constraint violation based on strains. Specifying a reference strain implies the use of strains for measuring the material constraint violation (or part of the control-stiffness error in Adagio). Otherwise, the material constraint violation is determined by using the change in the scaled stress response over the current model problem.

4.2.24 Mooney-Rivlin Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_value
  #
  BEGIN PARAMETERS FOR MODEL MOONEY_RIVLIN
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    BULK MODULUS = <real>bulk_modulus
    SHEAR MODULUS = <real>shear_modulus
    LAMBDA = <real>lambda
    C10 = <real>c10
    C01 = <real>c01
    C10 FUNCTION = <string>c10_function_name
    C01 FUNCTION = <string>c01_function_name
    BULK FUNCTION = <string>bulk_function_name
    THERMAL EXPANSION FUNCTION = <string>eth_function_name
    TARGET E = <real>target_e
    TARGET E FUNCTION = <string>etar_function_name
    MAX POISSONS RATIO = <real>max_poissons_ratio(0.5)
    REFERENCE STRAIN = <real>reference_strain
  END [PARAMETERS FOR MODEL MOONEY_RIVLIN]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

Mooney-Rivlin is a hyperelastic model that is used to model rubber. The Mooney-Rivlin model incorporates temperature-dependent material moduli and can be used in both Presto and Adagio. When the model is used in Adagio, it can be used with or without the control-stiffness option in Adagio's multilevel solver. The control-stiffness option is implemented via the `CONTROL STIFFNESS` command block and is discussed in Chapter 3. The model is used to model nearly incompressible materials where Poisson's ratio, ν, ≈ 0.5. In the course of solving a series of model problems in Adagio, the material response from this model incorporates scaling the bulk and/or shear behaviors to yield a material response that is more amenable to solution using Adagio's conjugate gradient solver. The final material behavior that is calculated corresponds to the actual moduli that are specified. When this model is used in Presto, the material scalings are ignored.

The command block for a Mooney-Rivlin material starts with the line:

```
BEGIN PARAMETERS FOR MODEL MOONEY_RIVLIN
```

and terminates with the line:

```
END [PARAMETERS FOR MODEL MOONEY_RIVLIN]
```

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- Any two of the following elastic constants are required to define the unscaled bulk behavior:
  - 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 nominal value for C10 is defined with the `C10` command line. This command line is required. See the usage discussion below.
- The nominal value for C01 is defined with the `C01` command line. This command line is required. See the usage discussion below.
- The `C10 FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope that describes the temperature dependence of the C10 material parameter. This command line is optional. If it is not present, there is no temperature dependence in the C10 parameter. See the usage discussion below.
- The `C01 FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope that describes the temperature dependence of the C01 material parameter. This command line is optional. If it is not present, there is no temperature dependence in the C01 parameter. See the usage discussion below.
- The `BULK FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope that describes the temperature dependence of the bulk modulus. This command line is optional. If it is not present, there is no temperature dependence in the bulk modulus. See the usage discussion below.
- The `THERMAL EXPANSION FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope that describes the linear thermal expansion as function of temperature. This command line is optional. If it is not present, there is no thermal expansion. See the usage discussion below.
- The following material-scaling command lines are used only in Adagio:
  - The target Young's modulus is defined with the `TARGET E` command line. This command line is optional. See the usage discussion below.

- The `TARGET E FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope that describes the time variation of the target Young's modulus. This command line is optional. If it is not present, there is no time dependence in the Target E parameter. See the usage discussion below.
- The maximum Poisson's ratio is defined with the `MAX POISSONS RATIO` command line. This command line is optional and will default to 0.5 if not specified. See the usage discussion below.
- The reference strain is defined with the `REFERENCE STRAIN` command line. This command line is optional. See the usage discussion below.

As noted previously, only two of the elastic constants are required to define the unscaled bulk behavior. Together, the values for C10 and C01 determine the shear behavior, and thus the `C10` and `C01` command lines must be included in this model.

The command lines for functions that specify the temperature dependence of C10, C01, and bulk modulus are optional, e.g., the `C10 FUNCTION`, `C01 FUNCTION` and `BULK FUNCTION` command lines. If these command lines are not included, their corresponding material parameters are taken to be independent of temperature. Mooney-Rivlin, like other material models, allows for the specification of thermal strain behavior within the material model itself, via the `THERMAL EXPANSION FUNCTION` command line. This command line, like the other "function-type" command lines in this model requires that a function associated with the name be defined in the SIERRA scope.

The bulk and shear scalings that can be used with the multilevel solver in Adagio are specified via a combination of the `TARGET E`, `TARGET E FUNCTION`, and `MAX POISSONS RATIO` command lines. If the `TARGET E` command line is not included (and the `MAX POISSONS RATIO` command line is included), the shear scaling is set to 1.0, and the bulk scaling is determined from the ratio of the scaled bulk modulus to its unscaled value, where the scaled bulk modulus is computed using the value of the `max_poissons_ratio` parameter along with the unscaled initial shear modulus that is determined from the value of the parameters specified in the `C10` and `C01` command lines. On the other hand, if both the `TARGET E` command line and the `MAX POISSONS RATIO` command line are included, bulk and shear scaling values are computed using scaled moduli that are calculated from the `target_e` and `max_poissons_ratio` parameter values.

Including the `TARGET E FUNCTION` command line allows time-dependent bulk and shear scaling to be used. If this command line is not specified, the bulk and shear scalings remain constant in solution time. If the command line is specified, the target Young's modulus that is used for computing the scaled moduli is multiplied by the function value.

The `REFERENCE STRAIN` command line supplies a value for the reference strain used to create a normalized material constraint violation that is based on strains. Specifying a reference strain implies the use of strains for measuring the material constraint violation (or part of the control-stiffness error in Adagio). Otherwise, the material constraint violation is determined using the change in the scaled stress response over the current model problem.

Brief documentation on the theoretical basis for the Mooney-Rivlin model is given in Reference 17.

4.2.25 NLVE 3D Orthotropic Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_value
  #
  BEGIN PARAMETERS FOR MODEL NLVE_3D_ORTHOTROPIC
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    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
    FICTITIOUS LOGA FUNCTION = <string>fict_loga_function_name
    FICTITIOUS LOGA SCALE FACTOR = <real>fict_loga_scale_factor
    # In each of the five ``PRONY'' command lines and in
    # the RELAX TIME command line, the value of i can be from
    # 1 through 30
    1PSI PRONY <integer>i = <real>psi1_i
    2PSI PRONY <integer>i = <real>psi2_i
    3PSI PRONY <integer>i = <real>psi3_i
    4PSI PRONY <integer>i = <real>psi4_i
    5PSI PRONY <integer>i = <real>psi5_i
    RELAX TIME <integer>i = <real>tau_i
    REFERENCE TEMP = <real>tref
    REFERENCE DENSITY = <real>rhoref
    WLF C1 = <real>wlf_c1
    WLF C2 = <real>wlf_c2
    B SHIFT CONSTANT = <real>b_shift
    SHIFT REF VALUE = <real>shift_ref
    WWBETA 1PSI = <real>wwb_1psi
    WWTAU 1PSI = <real>wwt_1psi
    WWBETA 2PSI = <real>wwb_2psi
    WWTAU 2PSI = <real>wwt_2psi
    WWBETA 3PSI = <real>wwb_3psi
    WWTAU 3PSI = <real>wwt_3psi
    WWBETA 4PSI = <real>wwb_4psi
    WWTAU 4PSI = <real>wwt_4psi
    WWBETA 5PSI = <real>wwb_5psi
    WWTAU 5PSI = <real>wwt_5psi
    DOUBLE INTEG FACTOR = <real>dble_int_fac
    REF RUBBERY HCAPACITY = <real>hcapr
    REF GLASSY HCAPACITY = <real>hcapg
```

```
GLASS TRANSITION TEM = <real>tg
REF GLASSY C11 = <real>c11g
REF RUBBERY C11 = <real>c11r
REF GLASSY C22 = <real>c22g
REF RUBBERY C22 = <real>c22r
REF GLASSY C33 = <real>c33g
REF RUBBERY C33 = <real>c33r
REF GLASSY C12 = <real>c12g
REF RUBBERY C12 = <real>c12r
REF GLASSY C13 = <real>c13g
REF RUBBERY C13 = <real>c13r
REF GLASSY C23 = <real>c23g
REF RUBBERY C23 = <real>c23r
REF GLASSY C44 = <real>c44g
REF RUBBERY C44 = <real>c44r
REF GLASSY C55 = <real>c55g
REF RUBBERY C55 = <real>c55r
REF GLASSY C66 = <real>c66g
REF RUBBERY C66 = <real>c66r
REF GLASSY CTE1 = <real>cte1g
REF RUBBERY CTE1 = <real>cte1r
REF GLASSY CTE2 = <real>cte2g
REF RUBBERY CTE2 = <real>cte2r
REF GLASSY CTE3 = <real>cte3g
REF RUBBERY CTE3 = <real>cte3r
LINEAR VISCO TEST = <real>lvt
T DERIV GLASSY C11 = <real>dc11gdT
T DERIV RUBBERY C11 = <real>dc11rdT
T DERIV GLASSY C22 = <real>dc22gdT
T DERIV RUBBERY C22 = <real>dc22rdT
T DERIV GLASSY C33 = <real>dc33gdT
T DERIV RUBBERY C33 = <real>dc33rdT
T DERIV GLASSY C12 = <real>dc12gdT
T DERIV RUBBERY C12 = <real>dc12rdT
T DERIV GLASSY C13 = <real>dc13gdT
T DERIV RUBBERY C13 = <real>dc13rdT
T DERIV GLASSY C23 = <real>dc23gdT
T DERIV RUBBERY C23 = <real>dc23rdT
T DERIV GLASSY C44 = <real>dc44gdT
T DERIV RUBBERY C44 = <real>dc44rdT
T DERIV GLASSY C55 = <real>dc55gdT
T DERIV RUBBERY C55 = <real>dc55rdT
T DERIV GLASSY C66 = <real>dc66gdT
T DERIV RUBBERY C66 = <real>dc66rdT
T DERIV GLASSY CTE1 = <real>dcte1gdT
T DERIV RUBBERY CTE1 = <real>dcte1rdT
T DERIV GLASSY CTE2 = <real>dcte2gdT
```

```
T DERIV RUBBERY CTE2 = <real>dcte2rdT
T DERIV GLASSY CTE3 = <real>dcte3gdT
T DERIV RUBBERY CTE3 = <real>dcte3rdT
T DERIV GLASSY HCAPACITY = <real>dhcapgdT
T DERIV RUBBERY HCAPACITY = <real>dhcaprdT
REF PSIC = <real>psic_ref
T DERIV PSIC = <real>dpsicdT
T 2DERIV PSIC = <real>d2psicdT2
PSI EQ 2T = <real>psitt
PSI EQ 3T = <real>psittt
PSI EQ 4T = <real>psitttt
PSI EQ XX 11 = <real>psiXX11
PSI EQ XX 22 = <real>psiXX22
PSI EQ XX 33 = <real>psiXX33
PSI EQ XX 12 = <real>psiXX12
PSI EQ XX 13 = <real>psiXX13
PSI EQ XX 23 = <real>psiXX23
PSI EQ XX 44 = <real>psiXX44
PSI EQ XX 55 = <real>psiXX55
PSI EQ XX 66 = <real>psiXX66
PSI EQ XXT 11 = <real>psiXXT11
PSI EQ XXT 22 = <real>psiXXT22
PSI EQ XXT 33 = <real>psiXXT33
PSI EQ XXT 12 = <real>psiXXT12
PSI EQ XXT 13 = <real>psiXXT13
PSI EQ XXT 23 = <real>psiXXT23
PSI EQ XXT 44 = <real>psiXXT44
PSI EQ XXT 55 = <real>psiXXT55
PSI EQ XXT 66 = <real>psiXXT66
PSI EQ XT 1 = <real>psiXT1
PSI EQ XT 2 = <real>psiXT2
PSI EQ XT 3 = <real>psiXT3
PSI EQ XTT 1 = <real>psiXTT1
PSI EQ XTT 2 = <real>psiXTT2
PSI EQ XTT 3 = <real>psiXTT3
REF PSIA 11 = <real>psiA11
REF PSIA 22 = <real>psiA22
REF PSIA 33 = <real>psiA33
REF PSIA 12 = <real>psiA12
REF PSIA 13 = <real>psiA13
REF PSIA 23 = <real>psiA23
REF PSIA 44 = <real>psiA44
REF PSIA 55 = <real>psiA55
REF PSIA 66 = <real>psiA66
T DERIV PSIA 11 = <real>dpsiA11dT
T DERIV PSIA 22 = <real>dpsiA22dT
T DERIV PSIA 33 = <real>dpsiA33dT
```

```
        T DERIV PSIA 12 = <real>dpsiA12dT
        T DERIV PSIA 13 = <real>dpsiA13dT
        T DERIV PSIA 23 = <real>dpsiA23dT
        T DERIV PSIA 44 = <real>dpsiA44dT
        T DERIV PSIA 55 = <real>dpsiA55dT
        T DERIV PSIA 66 = <real>dpsiA66dT
        REF PSIB 1 = <real>psiB1
        REF PSIB 2 = <real>psiB2
        REF PSIB 3 = <real>psiB3
        T DERIV PSIB 1 = <real>dpsiB1dT
        T DERIV PSIB 2 = <real>dpsiB2dT
        T DERIV PSIB 3 = <real>dpsiB3dT
        PSI POT TT = <real>psipotTT
        PSI POT TTT = <real>psipotTTT
        PSI POT TTTT = <real>psipotTTTT
        PSI POT XT 1 = <real>psipotXT1
        PSI POT XT 2 = <real>psipotXT2
        PSI POT XT 3 = <real>psipotXT3
        PSI POT XTT 1 = <real>psipotXTT1
        PSI POT XTT 2 = <real>psipotXTT2
        PSI POT XTT 3 = <real>psipotXTT3
        PSI POT XXT 11 = <real>psipotXXT11
        PSI POT XXT 22 = <real>psipotXXT22
        PSI POT XXT 33 = <real>psipotXXT33
        PSI POT XXT 12 = <real>psipotXXT12
        PSI POT XXT 13 = <real>psipotXXT13
        PSI POT XXT 23 = <real>psipotXXT23
        PSI POT XXT 44 = <real>psipotXXT44
        PSI POT XXT 55 = <real>psipotXXT55
        PSI POT XXT 66 = <real>psipotXXT66
      END [PARAMETERS FOR MODEL NLVE_3D_ORTHOTROPIC]
    END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The NLVE three-dimensional orthotropic model is a nonlinear viscoelastic orthotropic continuum model that describes the behavior of fiber-reinforced polymer-matrix composites. In addition to being able to model the linear elastic and linear viscoelastic behaviors of such composites, it also can capture both “weak” and “strong” nonlinear viscoelastic effects such as stress dependence of the creep compliance and viscoelastic yielding. This model can be used in both Presto and Adagio.

Because the NLVE model is still under active development and also because it has an extensive list of command lines, we have not followed the typical approach in documenting this model.

4.2.26 Stiff Elastic

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 STIFF_ELASTIC
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    BULK MODULUS = <real>bulk_modulus
    SHEAR MODULUS = <real>shear_modulus
    LAMBDA = <real>lambda
    SCALE FACTOR = <real>scale_factor
    REFERENCE STRAIN = <real>reference_strain
  END [PARAMETERS FOR MODEL STIFF_ELASTIC]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The stiff elastic model is a variation of the isotropic elastic model. The stiff elastic model can be used in both Presto and Adagio. When the model is used in Adagio, it is typically used with the control-stiffness option in Adagio's multilevel solver. The control-stiffness option is implemented via the `CONTROL STIFFNESS` command block and is discussed in Chapter 3. The stiff elastic model is used to lower the stiffness of the bulk and shear behaviors of relatively stiff materials to yield a material response more amenable to solution using Adagio's conjugate gradient solver. The final material behavior that is calculated corresponds to the actual moduli that are specified. When this model is used in Presto, the material scalings are ignored.

The command block for a stiff elastic material starts with the line:

```
BEGIN PARAMETERS FOR MODEL STIFF_ELASTIC
```

and terminates with the line:

```
END [PARAMETERS FOR MODEL STIFF_ELASTIC]
```

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.

- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.

- The `thermal strain option` is used to define thermal strains. See Section 4.1.3.1 and Section 4.1.3.2 for further information on defining and activating thermal strains.

- Any two of the following elastic constants are required to define the unscaled material response:

  - 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 following command lines are used only in Adagio:

  - The material scaling is defined with the `SCALE FACTOR` command line.
  - The reference strain is defined with the `REFERENCE STRAIN` command line.

As noted previously, only two of the elastic constants are required to define the unscaled material response.

The scaled bulk and shear moduli are computed using a Young's modulus scaled by the value given by the `SCALE FACTOR` line command.

The `REFERENCE STRAIN` command line supplies a value for the reference strain used to create a normalized material constraint violation that is based on strains. Specifying a reference strain implies the use of strains for measuring the material constraint violation (or part of the control-stiffness error in Adagio). Otherwise, the material constraint violation is determined using the change in the scaled stress response over the current model problem.

4.2.27 Swanson Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_value
  #
  BEGIN PARAMETERS FOR MODEL SWANSON
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    BULK MODULUS = <real>bulk_modulus
    SHEAR MODULUS = <real>shear_modulus
    LAMBDA = <real>lambda
    A1 = <real>a1
    P1 = <real>p1
    B1 = <real>b1
    Q1 = <real>q1
    C1 = <real>c1
    R1 = <real>r1
    CUT OFF STRAIN = <real>ecut
    THERMAL EXPANSION FUNCTION = <string>eth_function_name
    TARGET E = <real>target_e
    TARGET E FUNCTION = <string>etar_function_name
    MAX POISSONS RATIO = <real>max_poissons_ratio(0.5)
    REFERENCE STRAIN = <real>reference_strain
  END [PARAMETERS FOR MODEL SWANSON]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The Swanson model is a hyperelastic model that is used to model rubber. The Swanson model can be used in both Presto and Adagio. When the model is used in Adagio, it can be used with or without the control-stiffness option in Adagio's multilevel solver for nearly incompressible materials where Poisson's ratio, ν, $\approx$ 0.5. The control-stiffness option is implemented via the `CONTROL STIFFNESS` command block and is discussed in Chapter 3. In the course of solving a series of model problems in Adagio, the material response from this model incorporates scaling the bulk and/or shear behaviors to yield a material response that is more amenable to solution using Adagio's conjugate gradient solver. The final material behavior that is calculated corresponds to the actual moduli that are specified. When this model is used in Presto, the material scalings are ignored.

The command block for a Swanson material starts with the line:

```
BEGIN PARAMETERS FOR MODEL SWANSON
```

and terminates with the line:

```
END [PARAMETERS FOR MODEL SWANSON]
```

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- Any two of the following elastic constants are required to define the unscaled bulk behavior:
  - 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 following command lines are required:
  - The material constant A1 is defined with the `A1` command line.
  - The material constant P1 is defined with the `P1` command line.
  - The material constant B1 is defined with the `B1` command line.
  - The material constant Q1 is defined with the `Q1` command line.
  - The material constant C1 is defined with the `C1` command line.
  - The material constant R1 is defined with the `R1` command line.
  - The small-strain value used for computing the initial shear modulus is defined with the `CUT OFF STRAIN` command line.
- The `THERMAL EXPANSION FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope that describes the linear thermal expansion as function of temperature. This command line is optional. If it is not present, there is no thermal expansion. See the usage discussion below.
- The following material-scaling command lines are used only in Adagio:
  - The target Young's modulus is defined with the `TARGET E` command line. This command line is optional. See the usage discussion below.
  - The `TARGET E FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope that describes the time variation of the target Young's modulus. This command line is optional. If it is not present, there is no time dependence in the Target E parameter. See the usage discussion below.
  - The maximum Poisson's ratio is defined with the `MAX POISSONS RATIO` command line. This command line is optional and will default to 0.5 if not specified. See the usage discussion below.
  - The reference strain is defined with the `REFERENCE STRAIN` command line. This command line is optional. See the usage discussion below.

As noted previously, only two of the elastic constants are required to define the unscaled bulk behavior. Together, the values for parameters in the `A1`, `P1`, `B1`, `Q1`, `C1`, and `R1` command lines define the unscaled shear behavior, so these command lines must be present. The initial unscaled shear modulus is determined from those parameter values along with the value of the parameter in the `CUT OFF STRAIN` command line, so this command line must also be present.

The Swanson model, like a few of the material models, allows for the specification of thermal strain behavior within the material model itself, via the `THERMAL EXPANSION FUNCTION` command line. This command line, like the other "function-type" command lines in this model, requires that a function associated with the name be defined in the SIERRA scope.

The bulk and shear scalings that can be used with the multilevel solver in Adagio are specified via a combination of the `TARGET E`, `TARGET E FUNCTION`, and `MAX POISSONS RATIO` command lines. If the `TARGET E` command line is not included (and the `MAX POISSONS RATIO` command line is included), the shear scaling is set to 1.0, and the bulk scaling is determined from the ratio of the scaled bulk modulus to its unscaled value, where the scaled bulk modulus is computed using the value of the `max_poissons_ratio` parameter along with the unscaled shear modulus. On the other hand, if both the `TARGET E` command line and the `MAX POISSONS RATIO` are included, bulk and shear scaling values are computed using scaled moduli that are calculated from the `target_e` and `max_poissons_ratio` parameter values.

Including the `TARGET E FUNCTION` command line allows time-dependent bulk and shear scaling to be used. If this command line is not specified, the bulk and shear scalings remain constant in solution time. If the command line is specified, the target Young's modulus that is used for computing the scaled moduli is multiplied by the function value.

The `REFERENCE STRAIN` command line supplies a value for the reference strain used to create a normalized material constraint violation that is based on strains. Specifying a reference strain implies the use of strains for measuring the material constraint violation (or part of the control-stiffness error in Adagio). Otherwise, the material constraint violation is determined using the change in the scaled stress response over the current model problem.

Output variables available for this model are listed in Table 8.38. Brief documentation on the theoretical basis for the Swanson model is given in Reference 17.

4.2.28 Viscoelastic Swanson Model

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_value
  #
  BEGIN PARAMETERS FOR MODEL VISCOELASTIC_SWANSON
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    BULK MODULUS = <real>bulk_modulus
    SHEAR MODULUS = <real>shear_modulus
    LAMBDA = <real>lambda
    A1 = <real>a1
    P1 = <real>p1
    B1 = <real>b1
    Q1 = <real>q1
    C1 = <real>c1
    R1 = <real>r1
    CUT OFF STRAIN = <real>ecut
    THERMAL EXPANSION FUNCTION = <string>eth_function_name
    PRONY SHEAR INFINITY = <real>ginf
    PRONY SHEAR 1 = <real>g1
    PRONY SHEAR 2 = <real>g2
    PRONY SHEAR 3 = <real>g3
    PRONY SHEAR 4 = <real>g4
    PRONY SHEAR 5 = <real>g5
    PRONY SHEAR 6 = <real>g6
    PRONY SHEAR 7 = <real>g7
    PRONY SHEAR 8 = <real>g8
    PRONY SHEAR 9 = <real>g9
    PRONY SHEAR 10 = <real>g10
    SHEAR RELAX TIME 1 = <real>tau1
    SHEAR RELAX TIME 2 = <real>tau2
    SHEAR RELAX TIME 3 = <real>tau3
    SHEAR RELAX TIME 4 = <real>tau4
    SHEAR RELAX TIME 5 = <real>tau5
    SHEAR RELAX TIME 6 = <real>tau6
    SHEAR RELAX TIME 7 = <real>tau7
    SHEAR RELAX TIME 8 = <real>tau8
    SHEAR RELAX TIME 9 = <real>tau9
    SHEAR RELAX TIME 10 = <real>tau10
    WLF COEF C1 = <real>wlf_c1
    WLF COEF C2 = <real>wlf_c2
    WLF TREF = <real>wlf_tref
    NUMERICAL SHIFT FUNCTION = <string>ns_function_name
    TARGET E = <real>target_e
    TARGET E FUNCTION = <string>etar_function_name
```

```
      MAX POISSONS RATIO = <real>max_poissons_ratio(0.5)
      REFERENCE STRAIN = <real>reference_strain
    END [PARAMETERS FOR MODEL VISCOELASTIC_SWANSON]
  END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The viscoelastic Swanson model is a finite strain viscoelastic model that has an initial elastic response that matches the Swanson material model. The bulk response is elastic, while the shear response is viscoelastic. This model is commonly employed in simulating the response of rubber materials. The viscoelastic Swanson model can be used in both Presto and Adagio. When the model is used in Adagio, it can be used with or without the control-stiffness option in Adagio's multilevel solver for nearly incompressible materials where Poisson's ratio, ν, ≈ 0.5. The control-stiffness option is implemented via the `CONTROL STIFFNESS` command block and is discussed in Chapter 3. In the course of solving a series of model problems in Adagio, the material response from this model incorporates scaling the bulk and/or shear behaviors to yield a material response that is more amenable to solution using Adagio's conjugate gradient solver. The final material behavior that is calculated corresponds to the actual moduli that are specified. When this model is used in Presto, the material scalings are ignored.

The command block for a viscoelastic Swanson material starts with the line:

```
BEGIN PARAMETERS FOR MODEL VISCOELASTIC_SWANSON
```

and terminates with the line:

```
END [PARAMETERS FOR MODEL VISCOELASTIC_SWANSON]
```

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line.
- The Biot's coefficient of the material is defined with the `BIOTS COEFFICIENT` command line.
- Any two of the following elastic constants are required to define the unscaled bulk behavior:
  - 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 following command lines are required:
  - The material constant A1 is defined with the `A1` command line.

- The material constant P1 is defined with the `P1` command line.
- The material constant B1 is defined with the `B1` command line.
- The material constant Q1 is defined with the `Q1` command line.
- The material constant C1 is defined with the `C1` command line.
- The material constant R1 is defined with the `R1` command line.
- The small-strain value used for computing the glassy shear modulus is defined with the `CUT OFF STRAIN` command line.

- The `THERMAL EXPANSION FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope that describes the linear thermal expansion as function of temperature. This command line is optional. If it is not present, there is no thermal expansion. See the usage discussion below.

- `PRONY SHEAR INFINITY` command line. This command line is required.

- The normalized relaxation spectra coefficients are specified with the `PRONY SHEAR I` command lines, where the value of `I` varies sequentially from 1 to 10. These command lines are optional.

- The normalized relaxation spectra time constants are specified with the `SHEAR RELAX TIME I` command lines, where the value of `I` varies sequentially from 1 to 10. These command lines are optional.

- `WLF COEF C1` command line. This command line is required.

- `WLF COEF C2` command line. This command line is required.

- `WLF TREF` command line. This command line is required.

- `NUMERICAL SHIFT FUNCTION` command line. This command line is optional.

- The following material-scaling command lines are used only in Adagio:

  - The target Young's modulus is defined with the `TARGET E` command line. This command line is required. See the usage discussion below.
  - The `TARGET E FUNCTION` command line references the name of a function defined in a `DEFINITION FOR FUNCTION` command line in the SIERRA scope that describes the time variation of the target Young's modulus. This command line is optional. If it is not present, there is no time dependence in the Target E parameter. See the usage discussion below.
  - The maximum Poisson's ratio is defined with the `MAX POISSONS RATIO` command line. This command line is optional and will default to 0.5 if not specified. See the usage discussion below.
  - The reference strain is defined with the `REFERENCE STRAIN` command line. This command line is required. See the usage discussion below.

As noted previously, only two of the elastic constants are required to define the unscaled bulk behavior. Together, the values for parameters in the A1, P1, B1, Q1, C1, and R1 command lines define the unscaled glassy shear behavior, so these command lines must be present. The unscaled glassy shear modulus is determined from those parameter values along with the value of the parameter in the CUT OFF STRAIN command line, so this command line must also be present.

The viscoelastic Swanson model, like a few of the material models, allows for the specification of thermal strain behavior within the material model itself, via the THERMAL EXPANSION FUNCTION command line. This command line, like the other "function-type" command lines in this model requires that a function associated with the name be defined in the SIERRA scope.

The bulk and shear scalings that can be used with the multilevel solver in Adagio are specified via a combination of the TARGET E, TARGET E FUNCTION, and MAX POISSONS RATIO command lines. If the TARGET E command line is not included (and the MAX POISSONS RATIO command line is included), the shear scaling is set to 1.0, and the bulk scaling is determined from the ratio of the scaled bulk modulus to its unscaled value, where the scaled bulk modulus is computed using the value of the max_poissons_ratio parameter along with the unscaled shear modulus. On the other hand, if both the TARGET E command line and the MAX POISSONS RATIO command line are included, bulk and shear scaling values are computed using scaled moduli that are calculated from the target_e and max_poissons_ratio parameter values.

Including the TARGET E FUNCTION command line allows time-dependent bulk and shear scaling to be used. If this command line is not specified, the bulk and shear scalings remain constant in solution time. If the command line is specified, the target Young's modulus that is used for computing the scaled moduli is multiplied by the function value.

The REFERENCE STRAIN command line supplies a value for the reference strain used to create a normalized material constraint violation that is based on strains. Specifying a reference strain implies the use of strains for measuring the material constraint violation (or part of the control-stiffness error in Adagio). Otherwise, the material constraint violation is determined using the change in the scaled stress response over the current model problem.

Output variables available for this model are listed in Table 8.39. Brief documentation on the theoretical basis for the viscoelastic Swanson model is given in References 17, 18, 19, and 20.

4.3 Cohesive Zone Material Models

Several material models are available for use with cohesive zone elements, and are described in this section.

Traction separation models used for cohesive surface elements are input within `BEGIN PROPERTY SPECIFICATION FOR MATERIAL` blocks in the same manner as continuum models. Although density is not a property used by cohesive zone elements, because of their specification within this block, all of these models currently require a density to be provided as part of their input.

4.3.1 Traction Decay

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  #
  DENSITY = <real>density_value
  #
  BEGIN PARAMETERS FOR MODEL TRACTION_DECAY
    NORMAL DECAY LENGTH = <real>
    TANGENTIAL DECAY LENGTH = <real>
  END [PARAMETERS FOR MODEL TRACTION_DECAY]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The Traction Decay cohesive model is a simple model that get initialized with a traction upon activation or insertion of the cohesive element, and decays that traction to zero over specified values of normal and tangential separation. This model is only valid to use in conjunction with dynamic cohesive zone activation through MPC deactivation or dynamic insertion of cohesive surface elements.

The command block for a traction decay material starts with the line:

```
BEGIN PARAMETERS FOR MODEL TRACTION_DECAY
```

and terminates with the line:

```
END [PARAMETERS FOR MODEL TRACTION_DECAY]
```

In the above command block:

- The density of the material is defined with the `DENSITY` command line. Although this is not used in the model, it is currently a required input parameter.
- The separation length over which the normal traction decays to zero is set by the `NORMAL DECAY LENGTH` command line.
- The separation length over which the tangential traction decays to zero is set by the `TANGENTIAL DECAY LENGTH` command line.

4.3.2 Tvergaard Hutchinson

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  #
  DENSITY = <real>density_value
  #
  BEGIN PARAMETERS FOR MODEL TVERGAARD_HUTCHINSON
    INIT TRACTION METHOD = IGNORE|ADD (IGNORE)
    LAMBDA_1 = <real>
    LAMBDA_2 = <real>
    NORMAL LENGTH SCALE = <real>
    TANGENTIAL LENGTH SCALE = <real>
    PEAK TRACTION = <real>
    PENETRATION STIFFNESS MULTIPLIER = <real>
    NORMAL INITIAL TRACTION DECAY LENGTH = <real>
    TANGENTIAL INITIAL TRACTION DECAY LENGTH = <real>
    USE ELASTIC UNLOADING = NO|YES (YES)
  END [PARAMETERS FOR MODEL TVERGAARD_HUTCHINSON]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The Tvergaard Hutchinson cohesive model combines the normal and tangential separation into a single normalized separation and calculates a traction per unit length based on this value. This model then calculates normal and tangential traction based on the ratio of the normal and tangential length scales.

For a Tvergaard Hutchinson surface model, a Tvergaard Hutchinson command block starts with the input line: The command block for a traction decay material starts with the line:

```
BEGIN PARAMETERS FOR MODEL TVERGAARD_HUTCHINSON
```

and terminates with the line:

```
END [PARAMETERS FOR MODEL TVERGAARD_HUTCHINSON]
```

In the above command blocks:

- The density of the material is defined with the `DENSITY` command line. Although this is not used in the model, it is currently a required input parameter.

- For dynamically activated or dynamically inserted cohesive surface elements, an initial traction can be added to the calculated traction based on element properties specified in the `ELEMENT DEATH` block and is set via the `INIT TRACTION METHOD`. The default behavior is to ignore any initial tractions and let the traction-separation law dictate the behavior.

- `LAMBDA_1` indicates the normalized separation at which the traction response flattens with an additional increase in separation.

- `LAMBDA_2` indicates the normalized separation at which the traction begins to degrade with an additional increase in separation.

- The separation at which failure occurs in the normal direction is prescribed using the `NORMAL LENGTH SCALE` command.

- The maximum traction is specified through the `PEAK TRACTION` command.

- `NORMAL INITIAL TRACTION DECAY LENGTH` and `TANGENTIAL INITIAL TRACTION DECAY LENGTH` specify the length over which the initial traction will decay to zero in the normal and tangential direction respectively if the cohesive elements are initialized during element death. This decay length is independent of the `NORMAL LENGTH SCALE` and `TANGENTIAL LENGTH SCALE` specified for the calculated traction.

- The separation at which failure occurs in the tangential direction is prescribed using the `TANGENTIAL LENGTH SCALE` command.

- To help prevent interpenetration of the cohesive faces, use the `PENETRATION STIFFNESS MULTIPLIER` command to artificially increase the normal traction when penetration occurs. WARNING: cohesive elements are not well equipped to handle compression. This is an ad-hoc method to handle contact.

- Set the `USE ELASTIC UNLOADING` command to `YES` to force this model to unload elastically.

4.3.3 Thouless Parmigiani

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value

  BEGIN PARAMETERS FOR MODEL THOULESS_PARMIGIANI
    INIT TRACTION METHOD = IGNORE|ADD (IGNORE)
    LAMBDA_1_N = <real>
    LAMBDA_1_T = <real>
    LAMBDA_2_N = <real>
    LAMBDA_2_T = <real>
    NORMAL LENGTH SCALE = <real>
    PEAK NORMAL TRACTION = <real>
    TANGENTIAL LENGTH SCALE = <real>
    PEAK Tangential TRACTION = <real>
    PENETRATION STIFFNESS MULTIPLIER = <real>
    USE ELASTIC UNLOADING = NO|YES (YES)
  END [PARAMETERS FOR MODEL THOULESS_PARMIGIANI]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

The Thouless Parmigiani model support mixed-mode fracture more accurately than the Tvergaard Hutchinson model by separating the normal and tangential components, allowing one to fail independently of the other. Failure of this model is dependent on the energy release of both normal and tangential components. The shape of the traction-separation curve for this model is hardening, followed by a plateau, followed by softening.

The command block for a Thouless Parmigiani material starts with the line:

```
BEGIN PARAMETERS FOR MODEL THOULESS_PARMIGIANI
```

and terminates with the line:

```
END [PARAMETERS FOR MODEL THOULESS_PARMIGIANI]
```

In the above command block:

- The density of the material is defined with the `DENSITY` command line. Although this is not used in the model, it is currently a required input parameter.
- For dynamically activated or dynamically inserted cohesive surface elements, an initial traction can be added to the calculated traction based on element properties specified in the `ELEMENT DEATH` block and is set via the `INIT TRACTION METHOD`. The default behavior is to ignore any initial tractions and let the traction-separation law dictate the behavior.
- `LAMBDA_1_N` indicates the normalized normal separation at which the traction response flattens with an additional increase in separation.
- `LAMBDA_1_T` indicates the normalized tangential separation at which the traction response flattens with an additional increase in separation.

- `LAMBDA_2_N` indicates the normalized normal separation at which the traction begins to decrease with an additional increase in separation.

- `LAMBDA_2_T` indicates the normalized tangential separation at which the traction begins to decrease with an additional increase in separation.

- The separation at which failure occurs in the normal direction is prescribed using the `NORMAL LENGTH SCALE` command.

- The maximum normal traction is specified through the `PEAK NORMAL TRACTION` command.

- The separation at which failure occurs in the tangential direction is prescribed using the `TANGENTIAL LENGTH SCALE` command.

- The maximum tangential traction is specified through the `PEAK TANGENTIAL TRACTION` command.

- To help prevent interpenetration of the cohesive faces, use the `PENETRATION STIFFNESS MULTIPLIER` command to artificially increase the normal traction when penetration occurs. WARNING: cohesive elements are not well equipped to handle compression. This is an ad-hoc method to handle contact.

- Set the `USE ELASTIC UNLOADING` command to `YES` to force this model to unload elastically.

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. pdf.

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. doi.

5. Simo, J. C., and T. J. R. Hughes. *Computational Inelasticity*, Springer-Verlag, New York, NY, 1998.

6. Taylor, L. M., and D. P. Flanagan. *PRONTO3D: A Three-Dimensional Transient Solid Dynamics Program*, SAND87-1912. Albuquerque, NM: Sandia National Laboratories, March 1989. pdf.

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

8. 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. doi.

9. Attaway, S. W., R. V. Matallucci, S. W. Key, K. B. Morrill, L. J. Malvar, and J. E. Crawford. *Enhancements to PRONTO3D to Predict Structural Response to Blast*, SAND2000-1017. Albuquerque, NM: Sandia National Laboratories, 2000.

10. *ACI318-08: Building Code Requirements for Structural Concrete and Commentary*. Farmington Hills, MI: American Concrete Institute, 2008.

11. Neilsen, M. K., and Morgan, H. S., and Krieg, R. D. *A Phenomenological Constitutive Model for Low Density Polyurethane Foams*, SAND86-2927. Albuquerque, NM: Sandia National Laboratories, April 1987. pdf.

12. Neilsen, M. K., and Pierce, J. D., and Krieg, R. D. *A Constitutive Model for Layered Wire Mesh and Aramid Cloth Fabric*, SAND91-2850. Albuquerque, NM: Sandia National Laboratories, 1993. pdf.

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

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

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

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

17. Scherzinger, W. M., and D. C. Hammerand. *Constitutive Models in LAME*, SAND2007-5873. Albuquerque, NM: Sandia National Laboratories, September 2007. pdf.

18. HKS. *ABAQUS Version 6.6, Theory Manual*. Providence, RI: Hibbitt, Karlsson and Sorensen, 2006.

19. Hammerand, D. C. "ABAQUS Style Finite Strain Viscoelasticity in Adagio." Memo. Albuquerque, NM: Sandia National Laboratories, March 2003.

20. Hammerand, D. C. "Finite Strain Viscoelasticity in Adagio and ABAQUS." Memo. Albuquerque, NM: Sandia National Laboratories, July 2003.

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

22. Johnson, G. R., and Cook, W. H. "A constitutive model and data for metals subjected to large strains, high strain rates and high temperatures" *Proc. 7th. Int. Symp. on Ballistics, The Hague, The Netherlands* (1983): 541–547.

Chapter 5

Elements

This chapter explains how material, geometric, and other properties are associated 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 input file. The specifics of a material type associated with these four node facets are also set in the input file.

Most elements can be used in either Presto or Adagio. If an element is available in one code but not the other, this information will be noted for the element. There are special element implementations in Presto for peridynamics and for smoothed particle hydrodynamics (SPH). These sections are included in the Presto manual but not the Adagio manual. This chapter also includes descriptions of the commands for mass property calculations, element death, and rigid bodies.

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.

Section 5.3.1 explains the use of rigid bodies. Section 5.4 describes the `MASS PROPERTIES` command block, which lets the user compute the total mass of the model or the mass of sub-parts 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. A command block for derived quantities that are to be used with transfers or error estimation is discussed in Section 5.6.

Most of the command blocks and command lines described next appear within the SIERRA 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
    OMIT BLOCK <string>block_list
    COMPONENT SEPARATOR CHARACTER = <string>separator
    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 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 SIERRA 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 5.1.5.

The current element library is 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.

- 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.

- 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. The node-based tetrahedron can be used with any of the material models described in Chapter 4. When using node-based tetrahedron it is important that nodal quantities are used where other element types use element quantities. For example, you evaluate element variable stress with a regular four-node tetrahedron but in node-based tetrahedron one should use nodal variable element_stress_## where ## is a number that increments from 1. Nodal variables for node-based tets will live on all nodes but are zero where they are not used. Element variables for node-based elements serve as intermediate variables and should not be used in post processing in the same way as other regular element variables.

- 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.

- 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.

- 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 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 hardening without failure
  – Multilinear elastic-plastic hardening with 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 hardening (no failure)

- Linear elastic shell element: The linear elastic shell element is linear in both a material and geometric sense. The linear elastic shell element can be used with any material, however it will use only the density and elastic material constants of that material. Use of the linear elastic shell element is specified with the `FORMULATION = NQUAD` command in the section command block.

- 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. The beam element is based on an incremental kinematic formulation that is accurate for large strains and rotations (e.g. this element exactly agrees with a logarithmic strain formulation under large strain axial loading). Thinning of the cross section is taken into account through a constant volume assumption. 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 spring: The two-node spring element computes a uniaxial resistance force based on a non-linear force-engineering strain function. This element can handle preloads, mass per unit length, resetting of the initial length after preload and any arbitrary loading function.

- Two-node damper: (Code Usage: Presto only) 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: The point mass element allows the user to put a specified mass and/or rotational inertia at a node. This element requires input for density, but does not make use of any other material properties.

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.

Two metacharacters can appear in the name of the mesh file. If the %P character is found in the name, it will be replaced with the number processors being used for the run. For example, if you are running on 1024 processors and use the name mesh-%P/job.g, then the name would be expanded to mesh-1024/job.g and the actual mesh files would be mesh-1024/job.g.1024.0000 to mesh-1024/job.g.1024.1023. The other recognized metacharacter is %B which is replaced with the base name of the input file containing the input commands. For example, if the commands are in the file my_analysis_run.i and the mesh database name is specified as %B.g, then the mesh would be read from the file my_analysis_run.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 format is supported by Presto and Adagio for mesh input. Other options may be added in the future.

5.1.2 Alias

It is possible to associate a user-defined name with some mesh entity. The mesh entity names for Exodus II entities are typically the concatenation of the entity type (for example, “block”, “nodelist”, or “surface”), an underscore (“_”), and the entity id. This generated name can be aliased to a more descriptive name 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.

5.1.3 Omit Block

If the finite element mesh contains element blocks that should be omitted from the finite element analysis, the `OMIT BLOCK` line command is used.

```
OMIT BLOCK <string>block_list
```

The element blocks listed in the command are removed from the model. Any nodesets or surfaces only existing on nodes or elements in the omitted element blocks are also omitted. Note that if this command is used in a parallel analysis, it is possible for the resulting model to become unbalanced if, for example, the omitted element blocks make up a large portion of the elements on one or more processors.

Examples of omitting element blocks are:

```
Omit Block block_1 block_2
Omit Block block_10
```

5.1.4 Component Separator Character

A variable defined on the mesh database can be used as an initial condition, or a prescribed temperature with the `READ VARIABLE` command. If the variable is a vector or a tensor, then the base name of the variable will be separated from the suffixes with a separator character. The default separator character is an underscore, but it can be changed with the `COMPONENT SEPARATOR CHARACTER` command.

```
COMPONENT SEPARATOR CHARACTER = <string>character|NONE
```

For example, the variable `displacement` can have the suffixes `x`, `y`, etc. By default, the base name is separated from the suffixes with an underscore character so that we have `displacement_x`, `displacement_y`, etc. in the mesh file. The underscore can be replaced as the default separator by using the above command line. If the data used the period as the separator, then the command would be

```
COMPONENT SEPARATOR CHARACTER = .
```

For the displacement example the components would then appear in the mesh file as `displacement.x`, `displacement.y`, etc.

The separator can be eliminated with an empty string or `NONE`.

5.1.5 Descriptors of Element Blocks

```
BEGIN PARAMETERS FOR BLOCK [<string list>block_names]
  MATERIAL <string>material_name
  SOLID MECHANICS USE MODEL <string>model_name
  INCLUDE ALL BLOCKS
  REMOVE BLOCK = <string list>block_names
  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)
  MEMBRANE HOURGLASS STIFFNESS =
    <real>memb_hour_glass_stiff_value(0.0)
  MEMBRANE HOURGLASS VISCOSITY =
    <real>memb_hour_glass_visc_value(0.0)
  BENDING HOURGLASS STIFFNESS =
    <real>bend_hour_glass_stiff_value(0.0)
  BENDING HOURGLASS VISCOSITY =
    <real>bend_hour_glass_visc_value(0.0)
  TRANSVERSE SHEAR HOURGLASS STIFFNESS =
    <real>tshr_hour_glass_stiff_value(0.0)
  TRANSVERSE SHEAR HOURGLASS VISCOSITY =
    <real>tshr_hour_glass_visc_value(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
  INACTIVE FOR PROCEDURE <string>proc_name DURING PERIODS
    <string list>period_names
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 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 element library is 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]
```

Here `block_names` is a list of element blocks assigned to the element-block command block. Such a list must be included on the `BEGIN PARAMETERS FOR BLOCK` input line if the `INCLUDE ALL BLOCKS` line command is not used. 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 (or all the element blocks included using the line commands `INCLUDE ALL BLOCKS` and `REMOVE BLOCK`) 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`, `COHESIVE SECTION`, `SHELL SECTION`, `MEMBRANE SECTION`, `BEAM SECTION`, `TRUSS SECTION`, and `SUPERELEMENT 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, the code assumes the element block is a block of eight-node hexahedra using mean quadrature and the midpoint-increment formulation.

All the command lines that can be used for the element-block command block are described in Section 5.1.5.1 through Section 5.1.5.8.

5.1.5.1 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.

5.1.5.2 Include All Blocks

The INCLUDE ALL BLOCKS line command is used to associate all element blocks with the same element parameters (which minimizes input).

```
INCLUDE ALL BLOCKS
```

5.1.5.3 Remove Block

The REMOVE BLOCK command line allows you to delete blocks from the set specified in the PARAMETERS FOR BLOCK command block or INCLUDE ALL BLOCKS command line(s) through the string list block_names.

```
REMOVE BLOCK <string>block_list
```

5.1.5.4 Section

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

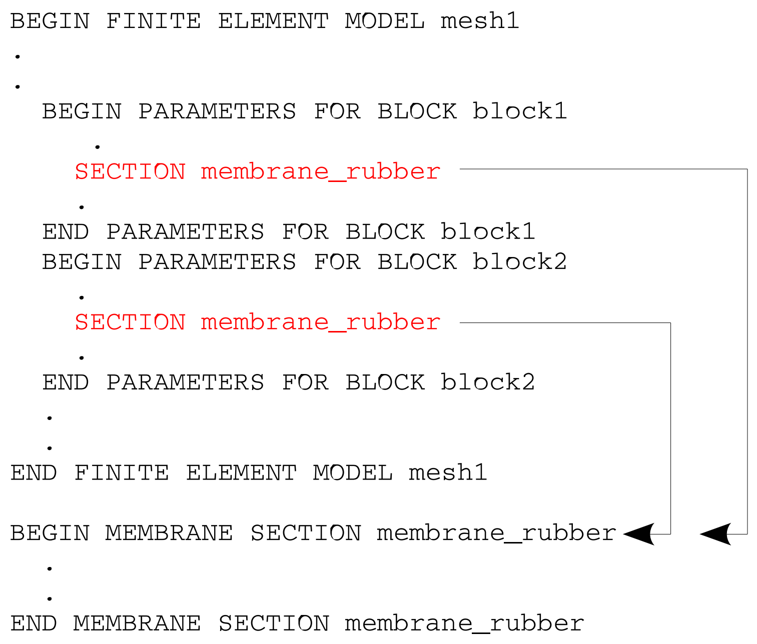
The section specification for an element-block command block is done by using the above command line. The section_id is a string associated with a section command block. The various section command blocks are described in Section 5.2.

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.



```
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.

5.1.5.5 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.

5.1.5.6 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)
MEMBRANE HOURGLASS STIFFNESS =
  <real>memb_hour_glass_stiff_value(0.0)
MEMBRANE HOURGLASS VISCOSITY =
  <real>memb_glass_visc_value(0.0)
BENDING HOURGLASS STIFFNESS =
  <real>bend_hour_glass_stiff_value(0.0)
BENDING HOURGLASS VISCOSITY =
  <real>bend_glass_visc_value(0.0)
TRANSVERSE SHEAR HOURGLASS STIFFNESS =
  <real>tshr_hour_glass_stiff_value(0.0)
TRANSVERSE SHEAR HOURGLASS VISCOSITY =
  <real>tshr_glass_visc_value(0.0)
```

These 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.

Hourglass stiffness and viscosity parameters for hexahedral and tetrahedral elements are set using the `HOURGLASS STIFFNESS` and `HOURGLASS VISCOSITY` commands, respectively. If either of these commands are used for shell elements, they set the hourglass stiffness or viscosity for all three modes (membrane, bending, and transverse shear).

Hourglass parameters for the membrane, bending, and transverse shear modes can be set individually for shell elements. The membrane hourglass stiffness and viscosity can be set with the `MEMBRANE HOURGLASS STIFFNESS` and `MEMBRANE HOURGLASS VISCOSITY` commands. These membrane hourglass commands can also be used with membrane elements. The bending hourglass stiffness and viscosity are set with the `BENDING HOURGLASS STIFFNESS` and `BENDING HOURGLASS VISCOSITY` commands, and transverse shear hourglass stiffness and viscosity are set with the `TRANSVERSE SHEAR HOURGLASS STIFFNESS` and `TRANSVERSE SHEAR HOURGLASS VISCOSITY` commands. All of these commands will override either the default values and any value set in the generic `HOURGLASS STIFFNESS` and/or `HOURGLASS VISCOSITY` commands for the particular mode that is specified.

The hourglass stiffness parameter defaults to 0.05 for solids using hourglass control; it defaults to 0.0 for shell and membrane elements. A reasonable user defined hourglass stiffness (if needed) for shells and membranes is 0.005 (approximately an order of magnitude lower than for solid 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 5.1.5.7 selects the method for computing the effective moduli.

5.1.5.7 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.

5.1.5.8 Activation/Deactivation of Element Blocks by Time

```
ACTIVE FOR PROCEDURE <string>proc_name DURING PERIODS
  <string list>period_names
INACTIVE 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 3.11.1) within a specific procedure named in an `ADAGIO PROCEDURE` command block (Section 2.2.1).

The `ACTIVE FOR PROCEDURE` or `INACTIVE FOR PROCEDURE` command lines can optionally be used to deactivate element blocks for a portion of the analysis. If the `ACTIVE FOR PROCEDURE` command is used, the element block is active for all periods listed for the named procedure, and is deactivated for all time periods that are absent from the list. If the `INACTIVE FOR PROCEDURE` command is used, the element block is deactivated for all periods listed for the named procedure. The element block is active for all time periods that are absent from the list. If neither command line is used, by default the block is active during all time periods. This command line controls the activation and deactivation of all elements in a block. Alternatively, individual elements can be deactivated with the `ELEMENT DEATH` command block (see Section 5.5).

Known Issue: Deactivation of element blocks does not currently work in conjunction with the full tangent preconditioner (see Section 3.3) in Adagio. To use this capability, one of the nodal preconditioners must be used.

5.2 Element Sections

Element sections are defined by section command blocks. There are currently nine different types of section command blocks. The section command blocks appear in the SIERRA 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, COHESIVE, SHELL, MEMBRANE, BEAM, TRUSS, or SUPERELEMENT. 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. The method used to associate these names with individual SECTION command lines in PARAMETERS FOR BLOCK command blocks is discussed in Section 5.1.5.4.

5.2.1 Solid Section

```
BEGIN SOLID SECTION <string>solid_section_name
  COORDINATE SYSTEM = <string>Coordinate_system_name
  FORMULATION = <string>MEAN_QUADRATURE|
    SELECTIVE_DEVIATORIC|VOID(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)
  HOURGLASS FORMULATION = <string>TOTAL|INCREMENTAL(INCREMENTAL)
  HOURGLASS INCREMENT = <string>ENDSTEP|MIDSTEP (ENDSTEP)
  HOURGLASS ROTATION = <string> APPROXIMATE|SCALED (APPROXIMATE)
  RIGID BODY = <string>rigid_body_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id
    TO <integer>last_id
  USE LAME|STRUMENTO(LAME)
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 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 COORDINATE SYSTEM command line specifies the name of a coordinate system definition block command that will be used to define a local coordinate system on each element of a block that uses this SOLID SECTION. The coordinate system can then be used to transform element stresses for solid elements from the global coordinate system to this local element coordinate system through the use of a BEGIN USER OUTPUT command block as described in Section out:user. It is also used for defining a local element coordinate system for representative volume analyses (see Section spec:rve).

The FORMULATION command line specifies whether the element will use a single-point integration rule (mean quadrature), use a selective-deviatoric rule, or act as a void element. 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.

The VOID formulation is valid for 8-node hexahedral and 4-node tetrahedral element blocks. Void elements only compute volume. They do not contribute internal forces to the model. The material model and density associated with void elements are ignored. The volume and the first and second derivatives of the volume for each element are stored in the element variables volume, volume_first_derivative, and volume_second_derivative. The volume derivatives are computed using least squares fits of the volume history, which is stored for the previous five time steps.

In addition to the per-element volume and derivatives, the total volume and derivatives of that total volume for all elements in each void element block are written to the results file as global variables. The names for these variables are voidvol_blockID, voidvol_first_deriv_blockID, and voidvol_second_deriv_blockID. In these global variable names, blockID is the ID of the block. For example, the void volume for block 8 would be stored in voidvol_8.

Some of the solid elements support different strain-incrementation formulations. See the element summary at the beginning of Section 5.1 to determine which strain-incrementation formulations are available for which 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, while 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.

The `HOURGLASS FORMULATION` command is used to switch between total and incremental forms of hourglass control. This option can only be used with eight-noded uniform-gradient hexahedral elements using strongly objective strain incrementation (`STRAIN INCREMENTATION = STRONGLY_OBJECTIVE`). One of the following two arguments can be used with this command: `TOTAL` or `INCREMENTAL`. The total formulation performs stiffness hourglass force updates based on the rotation tensor from the polar decomposition of the total deformation gradient. The incremental formulation is the default and performs stiffness hourglass force updates based on the hourglass velocities and the incremental rotation tensor. The viscous hourglass forces and the hourglass parameters are unchanged by this command. Consult the element documentation [6] for a description of the hourglass forces and the incremental hourglass formulation.

The `HOURGLASS INCREMENT` and `HOURGLASS ROTATION` commands control the speed and accuracy of the hourglass control computation. These commands are only applicable to the uniform gradient hex with midpoint strain incrementation (`STRAIN INCREMENTATION = MIDPOINT_INCREMENT`). The `HOURGLASS INCREMENT` line command specifies whether the hourglass resistance increment is to be computed at the middle or end of the time step. The endstep calculation has a slightly reduced computational cost while the midstep computation is more accurate. The default is `ENDSTEP`. The `HOURGLASS ROTATION` command controls whether the hourglass resistance will be scaled after rotation to preserve the magnitude. Scaling requires additional computation time but will be more accurate, particularly when very large rotations are present in the analysis. The default is `APPROXIMATE`, meaning no scaling is done.

Rigid elements in a section are indicated by including the `RIGID BODY` command line. The `RIGID BODY` command line specifies an identifier that maps to a rigid body command block. See Section 5.3.1 for a full discussion of how to create rigid bodies and Section 5.3.1.1 for information on the use of the `RIGID BODIES FROM ATTRIBUTES` command.

You can request that the material model that will be used with this solid section come from the legacy Strumento material model library by using the `USE STRUMENTO` line command. LAME is the default material model library for all solid sections [9] but can be explicitly requested with the

`USE LAME` line command. The versions of the material models in the Strumento library will be removed in a future release of Adagio, so it is advised to switch to the LAME versions as soon as possible.

5.2.2 Cohesive Section

```
BEGIN COHESIVE SECTION <string>cohesive_section_name
  NUMBER OF INTEGRATION POINTS = <integer>num_int_points(1)
END [COHESIVE SECTION <string>cohesive_section_name]
```

The `COHESIVE SECTION` command block is used to specify the properties for cohesive zone elements (quadrilateral and triangular). The name of this block (given by `cohesive_section_name`) is referenced by the element block for cohesive elements. If the option for adaptive insertion of cohesive zone elements is used, the name of this block is referenced by the `COHESIVE SECTION` command defined in Section 5.5.4.

```
NUMBER OF INTEGRATION POINTS = <integer>num_int_points(1)
```

The default number of integration points for a cohesive element is one. However, it should be noted that with a single integration point, spurious hour-glass like modes can be introduced to the deformation of the cohesive element. Currently, the quadrilateral cohesive element supports one and four integration points while the triangular cohesive element supports one and three integration points.

5.2.3 Localization Section

```
BEGIN LOCALIZATION SECTION <string>localization_section_name
  MEAN DILATATION FORMULATION
  NUMBER OF INTEGRATION POINTS = <integer>num_int_points
  THICKNESS = <real>thickness
END [LOCALIZATION SECTION <string>cohesive_section_name]
```

Localization elements are planar elements that lie between bulk (volumetric) elements and can employ the same underlying bulk material model. Topologically, localization elements are identical to cohesive surface elements. The reason this 2D element can access 3D material models is due to the multiplicative decomposition of the deformation gradient $\boldsymbol{F}$ such that $\boldsymbol{F} = \boldsymbol{F}^{\parallel}\boldsymbol{F}^{\perp}$ where $\boldsymbol{F}^{\parallel}$ encapsulates in-plane stretching and $\boldsymbol{F}^{\perp}$ is defined to be

$$\boldsymbol{F}^{\perp} = \boldsymbol{I} + \frac{\boldsymbol{\Delta}}{h} \otimes \boldsymbol{N} \tag{5.1}$$

where $\boldsymbol{N}$ is the normal to the mid-plane, $\boldsymbol{\Delta}$ is the gap vector, and h is the element "thickness" or the length scale governing the sub-grid separation progress. We note that because the length scale h is independent of the discretization, the methodology is regularized and ideal for employing local, softening material models to simulate the failure process. For full details on the theory, please see 15.

The `LOCALIZATION SECTION` command block is used to specify the properties for localization elements (quadrilateral and triangular). The name of this block (given by `localization_section_name`) is referenced by the element block for localization elements.

`MEAN DILATATION FORMULATION` This command line will yield a constant pressure formulation. The average pressure is obtained at all integration points through a modification of the kinematic quantities. For hypoelastic materials, we volume average the tr[$\boldsymbol{D}$]. For uncoupled hyperelastic models, we volume average det[$\boldsymbol{F}$]. After voluming averaging tr[$\boldsymbol{D}$] and det[$\boldsymbol{F}$], we remove local dilatational contributions and additively (hypoelastic) or multiplicatively (hyperelastic) include the average response. This results in an average pressure (by construction) and has been shown to be effective in avoiding element locking during isochoric deformations. Note that one can use a single integration point to achieve a constant pressure but we do not provide any hourglass control to suppress spurious modes. If isochoric deformations are of concern, we recommend using a fully-integrated element with `MEAN DILATATION FORMULATION`.

`NUMBER OF INTEGRATION POINTS = <integer>num_int_points` The default number of integration points for a localization element depends on the topology, but is always sufficient for full integration. For an 8 noded hex (planar 4 node quad) the default is 4 integration points while for a 6 noded wedge (planar 3 node tri) the default is 1 integration point. However, it should be noted that with a single integration point, spurious hourglass modes can be introduced into the deformation of the quadrilateral localization element. Currently, the quadrilateral localization element supports one and four integration points while the triangular localization element supports one integration point.

`THICKNESS = <real>thickness` The `THICKNESS` command line sets the localization element thickness h. In many respects, h should be considered a material parameter as it governs the evolution of surface separation. We note that the introduction of h generates the true length scale in the problem, the process zone size. Care must be taken to adequately resolve the process zone size or the methodology will not be regularized.

5.2.4 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)
  FORMULATION = MEAN_QUADRATURE|NQUAD (MEAN_QUADRATURE)
  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)
    OFFSET MESH VARIABLE = <string>var_name
    ORIENTATION = <string>orientation_name
    DRILLING STIFFNESS FACTOR = <real>stiffness_factor(0.0)
    RIGID BODY = <string>rigid_body_name
    RIGID BODIES FROM ATTRIBUTES = <integer>first_id
      TO <integer>last_id
    USE LAME|STRUMENTO(LAME)
  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 shell elements in the block will have their thickness initialized to the value shell_thickness.

Adagio 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
```

Here, 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.

The shell formulation can be selected via the FORMULATION command line. The default formulation is MEAN_QUADRATURE. A fully elastic formulation may be selected with the NQUAD option.

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 mid-surface of a shell element relative to the geometric location of the shell element. The lofting factor must be greater than or equal to 0.0 and less than or equal to 1.0. By default, the geometric location of a shell element in a mesh represents the mid-surface 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. For a lofting factor of 0.0, the geometric surface defined by the shell corresponds to the top surface of the shell element. A lofting factor of 1.0 puts the geometric surface at the bottom surface of the shell element. The geometric surface is midway between the top and bottom surfaces for a lofting factor of 0.5, which is the default. Lofting factors greater than 1.0 or less than 0.0 would put the geometric surface outside the shell element and are not allowed.

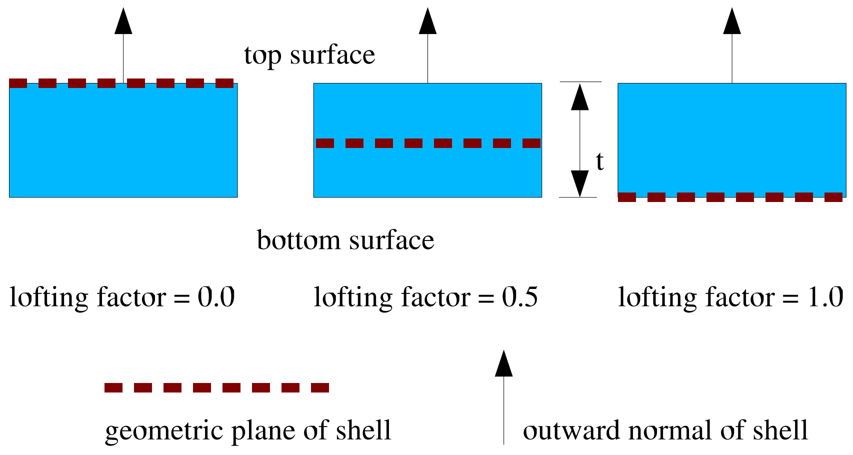


Figure 5.2: Location of geometric plane of shell for various lofting factors.

Consider the example of a lofting factor set to 1.0 for a shell with thickness of 0.2 inch. In this case, the top surface of the shell will be located at a distance of 0.2 inch from the geometric surface (measuring in the direction of the positive shell normal), and the bottom surface will be located at the geometric surface.

Both the shell mechanics and contact use shell lofting. See Section 7.3 for a discussion of lofting surfaces for shells and contact. It is recommended that shell lofting values other than 0.5 not be used if the shell is thick. If the shell is thicker than its in-plane width, the shell lofting algorithms may become unstable.

Lofting as described above may also be implemented through

```
OFFSET MESH VARIABLE = <string>var_name.
```

This command allows an offset value to be read from an attribute (or variable) on the mesh file. This attribute (or variable) must be named but may have any name, and this name is specified in place of `var_name` in the command. An offset is a dimensional value (i.e. not scaled by the shell thickness) that gives the shell mid-plane shift in the positive shell normal direction. Internal to Adagio the offset value is converted to an equivalent lofting factor by dividing by the initial thickness and adding 0.5. Thus, an offset of zero gives a lofting factor of 0.5, an offset of one half of the thickness gives a lofting factor of one, and an offset of negative one half of the thickness gives a lofting factor of zero. There is no check that given offset values produce lofting values between zero and one, which allows any offset to be specified but requires that care be exercised to avoid unstable lofted shells.

Warning: An offset and a lofting factor may not be specified in the same shell section. Both determine the shell lofting and may conflict.

The `ORIENTATION` command line lets you select a coordinate system for output of in-plane stresses and strains. 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.

The `ORIENTATION` command line in the `SHELL SECTION` command block references an `ORIENTATION` command block that appears in the SIERRA scope. As described in Chapter 2 of this document, the `ORIENTATION` command block can be used to define a local co-rotational 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 in-plane stresses and strains. 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
```

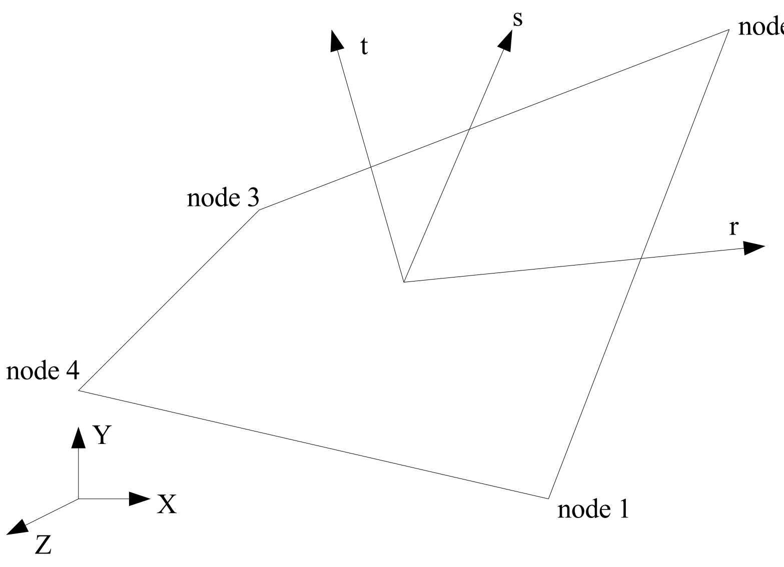


Figure 5.3: Local *rst* coordinate system for a shell element.

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^{ip}_{xx} will be in the Y''-direction; what will be output as the in-plane stress σ^{ip}_{yy} will be in the Z''-direction. The in-plane stress σ^{ip}_{xy} 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.

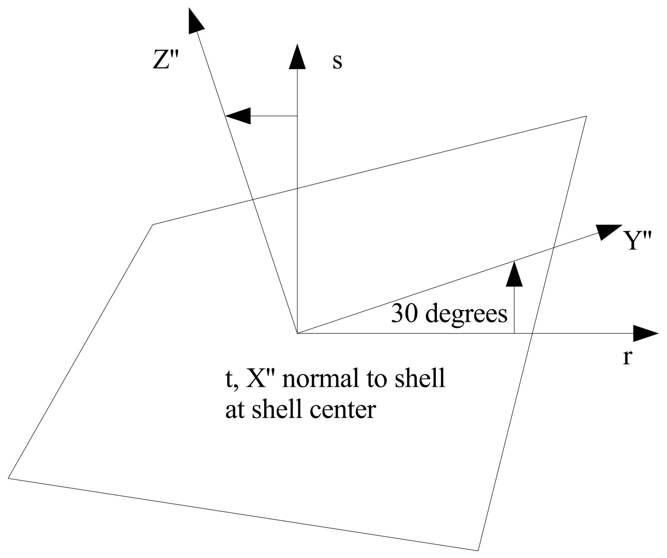


Figure 5.4: Rotation of 30 degrees about the 1-axis (X'-axis).

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

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

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

If no orientation is specified, the in-plane stresses and strains are output in the consistent axes from a default orientation, which is a rectangular system with the X' and Y' axes taken as the global X and Y axes, respectively, and `ROTATION ABOUT 1 = 0.0`.

The command line

```
DRILLING STIFFNESS FACTOR = <real>stiffness_factor
```

adds stiffness in the drilling degrees of freedom to quadrilateral shells. Drilling degrees of freedom are rotational degrees of freedom in the direction orthogonal to the plane of the shell at each node. The formulation used for the quadrilateral shells has no rotational stiffness in this direction. This can lead to spurious zero-energy modes of deformation similar in nature to hourglass deformation. This makes obtaining a solution difficult in quasistatic problems and can result in singularities when using the full tangent preconditioner.

The `stiffness_factor` should be chosen as a quantity small enough to add enough stiffness to allow the solve to be successful without unduly affecting the solution. The default value for `stiffness_factor` is 0. If singularities are encountered in the solution or hourglass-like deformation is observed in the drilling degrees of freedom, it is recommend to try using a small amount of drilling stiffness. A suggested trial value for `stiffness_factor` is 1.0e-4.

Elements in a section can be made rigid by including the `RIGID BODY` command line. The `RIGID BODY` command line specifies an indenter that maps to a rigid body command block. See Section 5.3.1 for a full discussion of how to create rigid bodies and Section 5.3.1.1 for information on the use of the `RIGID BODIES FROM ATTRIBUTES` command.

You can request that the material model that will be used with this shell section come from the Strumento material model library by using the `USE STRUMENTO` line command. LAME is the default material model library for all shell sections [9], but can be explicitly requested with the `USE LAME` line command. The versions of the material models in the Strumento library will be removed in a future release of Adagio, so it is advised to switch to the LAME versions as soon as possible.

5.2.5 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)
  RIGID BODY = <string>rigid_body_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id
    TO <integer>last_id
END [MEMBRANE SECTION <string>membrane_section_name]
```

The `MEMBRANE SECTION` command block is used to specify the properties for a membrane element. If a section defined by this command block is referenced in the parameters for a block of four-noded elements, the elements in that block will be treated as membranes. 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`.

Adagio 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)
```

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 elements, when used with a parameter greater than 0.0, provide hourglass control without artificial hourglass parameters. 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 command line

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

allows the user to shift the location of the mid-surface 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 mid-surface 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. For a lofting factor of 0.0, the geometric surface defined by the membrane corresponds to the top surface of the membrane element. A lofting factor of 1.0 puts the geometric surface at the bottom surface of the membrane element. The geometric surface is midway between the top and bottom surfaces for a lofting factor of 0.5, which is the default.

Consider the example of a lofting factor set to 1.0 for a membrane with thickness of 0.2 inch. In this case, the top surface of the membrane will be located at a distance of 0.2 inch from the geometric surface (measuring in the direction of the positive shell normal), and the bottom surface will be located at the geometric surface.

Both the membrane mechanics and contact use membrane lofting. See Section 7.3 for a discussion of lofting surfaces for membranes and contact.

Elements in a section can be made rigid by including the `RIGID BODY` command line. The `RIGID BODY` command line specifies an indenter that maps to a rigid body command block. Consult Section 5.3.1 for a full description of how to create rigid bodies and Section 5.3.1.1 for information on the use of the `RIGID BODIES FROM ATTRIBUTES` command.

5.2.6 Beam Section

```
BEGIN BEAM SECTION <string>beam_section_name
  SECTION = <string>ROD|TUBE|BAR|BOX|I
  WIDTH = <real>section_width
  WIDTH VARIABLE = <string>width_var
  HEIGHT = <real>section_width
  HEIGHT VARIABLE= <string>height_var
  WALL THICKNESS = <real>wall_thickness
  WALL THICKNESS VARIABLE = <string>wall_thickness_var
  FLANGE THICKNESS = <real>flange_thickness
  FLANGE THICKNESS VARIABLE = <string>flange_thickness_var
  T AXIS = <real>tx <real>ty <real>tz
  T AXIS VARIABLE = <string>t_axis_var
  REFERENCE AXIS = <string>CENTER|RIGHT|
    TOP|LEFT|BOTTOM(CENTER)
  AXIS OFFSET = <real>s_offset <real>t_offset
  AXIS OFFSET GLOBAL = <real>x_offset <real>y_offset <real>z_offset
  AXIS OFFSET VARIABLE = <string>axis_offset_var
  RIGID BODY = <string>rigid_body_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id
    TO <integer>last_id
  USE LAME|STRUMENTO(LAME)
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 or WIDTH VARIABLE
  HEIGHT or HEIGHT VARIABLE
  ```

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

  ```
  WIDTH or WIDTH VARIABLE
  HEIGHT or HEIGHT VARIABLE
  WALL THICKNESS or WALL THICKNESS VARIABLE
  ```

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

```
WIDTH or WIDTH VARIABLE
HEIGHT or HEIGHT VARIABLE
```

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

```
WIDTH or WIDTH VARIABLE
HEIGHT or HEIGHT VARIABLE
WALL THICKNESS or WALL THICKNESS VARIABLE
```

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

```
WIDTH or WIDTH VARIABLE
HEIGHT or HEIGHT VARIABLE
WALL THICKNESS or WALL THICKNESS VARIABLE
FLANGE THICKNESS or FLANGE THICKNESS VARIABLE
```

Most of the sections require the `T AXIS` or `T AXIS VARIABLE` command line. If the beam has a circular or tube cross section, and the width of the beam exactly equals the height, then the `T_AXIS` need not be specified. If the `T_AXIS` is not specified for one of these circularly symmetric cross sections the code will arbitrarily pick a t axis at each beam that is perpendicular to the the beam.

Beam section parameters can be specified as constant for all beams in the section with commands such as `WIDTH` or `T AXIS`. Alternatively a set of beam parameters that vary from element to element can be specified with variants of these commands with `VARIABLE` at the end, such as `WIDTH VARIABLE` or `T AXIS VARIABLE`. When the `VARIABLE` variants of commands are used, the command specifies the name of an attribute field on the input mesh that contains the parameter. For `WIDTH VARIABLE`, `HEIGHT VARIABLE`, `WALL THICKNESS VARIABLE`, and `FLANGE THICKNESS VARIABLE`, the field should contain one entry per element. For `AXIS OFFSET VARIABLE` the field should contain either two or three entries per element. If the field for the axis offset contains two entries, it specifies the offset in the local s, t coordinate system. If it contains three entries, it specifies the offset in the global x, y, z coordinate system. For `T AXIS VARIABLE` the field should contain three entries per element.

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 as a vector in the global coordinate system. The s-axis is computed from the cross product of the t-axis and the r-axis. The t-axis is then recomputed as the cross product of the r-axis and the s-axis to ensure that the t-axis is orthogonal to the r-axis. These local direction vectors are all normalized, so the user-input vectors do not have to be unit vectors.

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 `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.

Alternatively, the axis offset can be specified in global coordinates using the `AXIS OFFSET GLOBAL` command. This command takes three parameters, which are the *x*, *y*, and *z* components of the offset in the global coordinate system.

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.

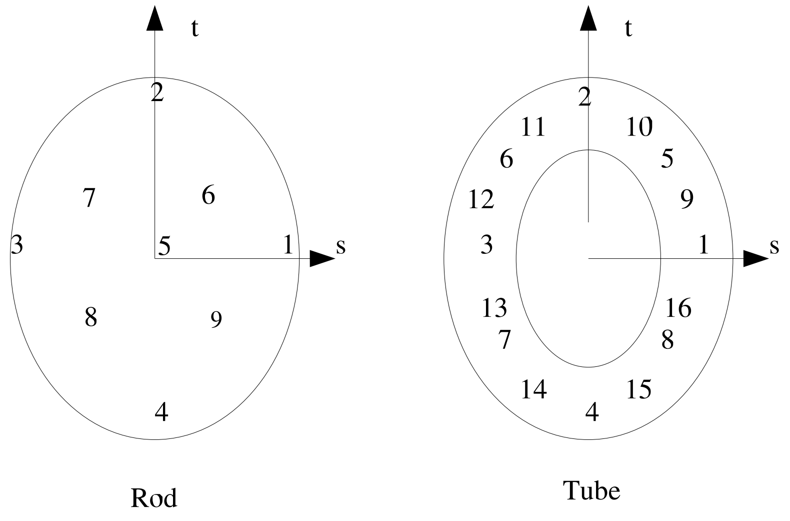


Figure 5.5: Integration points for rod and tube

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 variable that will let users access the strain at the beam integration points is

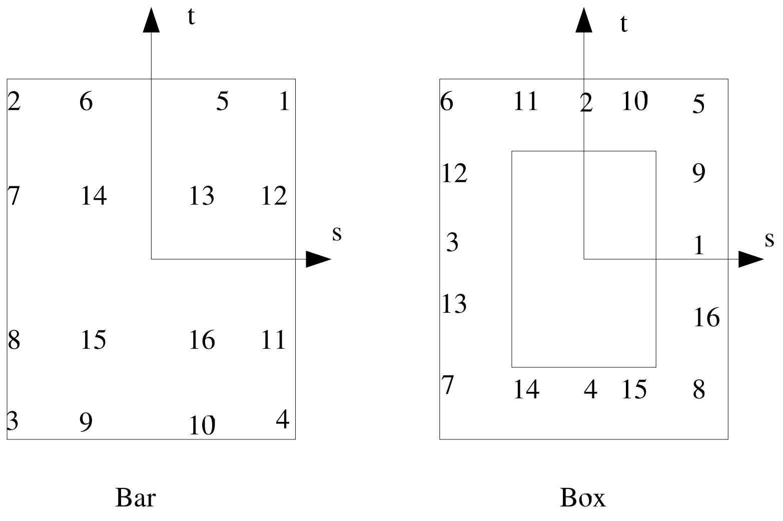


Figure 5.6: Integration points for bar and box.

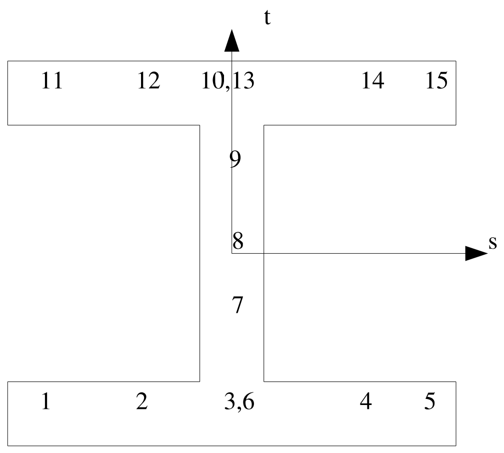


Figure 5.7: Integration points for I-section.

`beam_strain_inc`, and the variable that will let users access the stress at the beam integration points is `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 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. Output of stress is slightly different than the output of strain because the stress is stored as a symmetric tensor that contains six components, although four of these components are never used. The axial stress at the first integration point is designated by stress_xx_01 and the shear stress at the first integration point is stress_xy_01. The other four components, stress_yy_01, stress_zz_01, stress_yz_01 and stress_xz_01, are unused and are set to zero. The stresses at other integration points are named stress_xx_NN and stress_xy_NN, where NN is a number from 01 to 16.

As an alternative for the stress output, you may use the 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 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 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.

Elements in a section can be made rigid by including the RIGID BODY command line. The RIGID BODY command line specifies an identifier that maps to a rigid body command block. See Section 5.3.1 for a full discussion of how to create rigid bodies and Section 5.3.1.1 for information on the use of the RIGID BODIES FROM ATTRIBUTES command.

You can request that the material model that will be used with this beam section come from the Strumento material model library with the USE STRUMENTO line command. LAME is the default material model library for all beam sections [9] but can be explicitly requested by using the USE LAME line command. The versions of the material models in the Strumento library will be removed in a future release of Adagio, so it is advised to switch to the LAME versions as soon as possible.

5.2.7 Truss Section

```
BEGIN TRUSS SECTION <string>truss_section_name
  AREA = <real>cross_sectional_area
  INITIAL LOAD = <real>initial_load
  PERIOD = <real>period
  RIGID BODY = <string>rigid_body_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id
    TO <integer>last_id
  USE LAME|STRUMENTO(LAME)
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

$$\dot{\varepsilon} = \frac{ap}{2}\left[1 - \cos\left(\pi t/p\right)\right], \tag{5.2}$$

where

$$a = \frac{2F_i}{EAp}. \tag{5.3}$$

In Equation (5.3), 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.

Elements in a section can be made rigid by including the RIGID BODY command line. The RIGID BODY command line specifies an indenter that maps to a rigid body command block. See Section 5.3.1 for a full discussion of how to create rigid bodies and Section 5.3.1.1 for information on the use of the RIGID BODIES FROM ATTRIBUTES command.

It should be noted that the axial stress, which is the only stress component for a truss element, is output as a symmetric tensor with six components to be compatible with the notion of volumetric stress. Because of this, for every truss element, six values of stress are stored and output, but only the fist value is ever used. The axial stress is therefore output as stress_xx and the other five stress components, stress_yy, stress_zz, stress_xy, stress_yz, stress_zx are all unused and set to zero.

You can request that the material model that will be used with this truss section come from the Strumento material model library with the `USE STRUMENTO` line command. LAME is the default material model library for all truss sections [9] but can be explicitly requested with the `USE LAME` line command. The versions of the material models in the Strumento library will be removed in a future release of Adagio, so it is advised to switch to the LAME versions as soon as possible.

5.2.8 Superelement Section

```
BEGIN SUPERELEMENT SECTION <string>section_name
  BEGIN MAP
    <integer>node_index_1 <integer>component_index_1
    <integer>node_index_2 <integer>component_index_2
    ...
    <integer>node_index_n <integer>component_index_n
  END
  BEGIN STIFFNESS MATRIX
    <real>k_1_1 <real>k_1_2 ...  <real>k_1_n
    <real>k_2_1 <real>k_2_2 ...  <real>k_2_n
    ...         ...         ...  ...
    <real>k_n_1 <real>k_n_2 ...  <real>k_n_n
  END
  BEGIN DAMPING MATRIX
    <real>c_1_1 <real>c_1_2 ...  <real>c_1_n
    <real>c_2_1 <real>c_2_2 ...  <real>c_2_n
    ...         ...         ...  ...
    <real>c_n_1 <real>c_n_2 ...  <real>c_n_n
  END
  BEGIN MASS MATRIX
    <real>m_1_1 <real>m_1_2 ...  <real>m_1_n
    <real>m_2_1 <real>m_2_2 ...  <real>m_2_n
    ...          ...        ...  ...
    <real>m_n_1 <real>m_n_2 ...  <real>m_n_n
  END
  FILE = <string>netcdf_file_name
END [SUPERELEMENT SECTION <string>section_name]
```

A superelement allows definition of an element with a user defined stiffness, damping, and mass matrix. The superelement stiffness is linear and remains constant in time.

The superelement must be represented by an element in the mesh file. A block of elements used to define a superelement must contain exactly one finite element. The finite element that represents the superelement may have any topology. The topology may either be a valid geometric topology (hex, rod, tet, etc.) or may be be an arbitrary topology as defined in the input mesh file. The nodes of a superelement can be shared with other elements, or can be attached only to the superelement. In addition, the superelement can have additional internal degrees of freedom that are not present in the mesh file. If output values are desired on a node, that node must be present in the input mesh file.

Superelement nodes have all the same variables as regular nodes (mass, displacement, velocity, etc.) Only the element time step is defined as an output variable on the superelement itself.

5.2.8.1 Input Commands

```
BEGIN MAP
```

The `MAP` command block defines the mapping from nodal degrees of freedom to the local degrees of freedom in the stiffness and mass matrix for the superelement. The map should contain *N* pairs of integers, where *N* is the number of nodes in the superelement. The first integer of each pair is the index of the node in the superelement in the range $0 \ldots N$. The second integer is the component in the range $0 \ldots 6$.

A node index of 0 is a special value and marks the degree of freedom that is internal to the superelement. A superelement may have any number of internal degrees of freedom. Internal degrees of freedom are created internal to the code do not correspond to any nodes actually present in the mesh. A node index greater than zero represents that node index in the element. For example if the superelement was represented by an eight node hex element then node indexes could vary from one to eight and would match the first through eighth nodes in the hex element.

If an internal degree of freedom is used, the component index should be set to 0 along with the node index. If a regular node is used, components 1, 2, and 3 correspond to the X, Y, and Z translational degrees of freedom. Components 4, 5, and 6 correspond to the X, Y, and Z rotational degrees of freedom.

The following is an example superelement definition for a three degree of freedom truss element lying along the x-axis. The superelement is defined in the mesh file using a two node rod element. Degrees of freedom 1 and 3 are mapped to x degrees of freedom of the end nodes of the rod element. Degree of freedom 2 is internal to the superelement.

```
BEGIN SUPERELEMENT SECTION truss_x3
  BEGIN MAP
    1 1
    0 0
    2 1
  END
  BEGIN STIFFNESS MATRIX
    100 -100    0
   -100  200 -100
      0 -100  100
  END
  BEGIN DAMPING MATRIX
    1 -1  0
   -1  2 -1
    0 -1  1
  END
  BEGIN MASS MATRIX
    0.25 0.00 0.00
    0.00 0.50 0.00
    0.00 0.00 0.025
  END
END [SUPERELEMENT SECTION <string>section_name]
```

```
BEGIN STIFFNESS MATRIX
```

The `STIFFNESS MATRIX` command block defines the NxN stiffness matrix for the superelement. The number of rows and columns in the stiffness matrix must be the same as the number of rows in the `MAP` command block. The stiffness matrix should be symmetric. If the input matrix is not symmetric, it will be made symmetric by $K_{sym} = 0.5 * (K_{input} + K_{input}^T)$. To guarantee stability for explicit dynamics and solution convergence for implicit statics/dynamics, the stiffness matrix should be positive definite.

```
BEGIN DAMPING MATRIX
```

The `DAMPING MATRIX` command block defines the NxN damping matrix for the superelement. The number of rows and columns in the damping matrix must be the same as the number of rows in the `MAP` command block. The damping matrix should generally be symmetric. This command block is optional. If a damping matrix is not defined, no damping will be used by default.

```
BEGIN MASS MATRIX
```

The `MASS MATRIX` command block defines the NxN mass matrix for the superelement. The mass matrix must have the same dimensions as the stiffness matrix. The mass matrix need not be symmetric, however to guarantee stability for explicit dynamics and solution convergence for implicit statics/dynamics the mass matrix should be positive definite.

```
FILE = <string>netcdf_file_name
```

As an alternative to defining stiffness and mass matrices in the input file, the stiffness and mass matrices may be imported from a NetCDF file. External codes (such as Salinas) are able to compress larger models into superelements and output the matrices in the NetCDF format. The `netcdf_file_name` defines the path to the file that contains the mass, damping, and stiffness matrix definitions. All superelement matrices should be defined either in the input deck or in the NetCDF file. The damping matrix is optional, so if it is not defined in the NetCDF file, no damping will be used by default. The connectivity map needs to be specified in the input file in either case.

In the netcdf file the stiffness matrix must be named `Kr`, the mass matrix `Mr`, and the damping matrix `Cr`.

⚠ **Known Issue:** Superelements are not compatible with several modeling capabilities. They cannot be used with element death. They cannot be used with node-based, power method, or Lanczos critical time step estimation methods. They are also not compatible with some preconditioners (such as FETI) for implicit solutions.

5.3 Element-like Functionality

This section describes the rigid body functionality in Adagio. The rigid body functionality described in this section is specified through command blocks that appear in the SIERRA scope.

5.3.1 Rigid Body

```
BEGIN RIGID BODY <string>rb_name
  MASS = <real>mass
  POINT MASS = <real>mass [AT <real>X <real>Y <real>Z]
  REFERENCE LOCATION = <real>X <real>Y <real>Z
  INERTIA = <real>Ixx <real>Iyy <real>Izz <real>Ixy
    <real>Iyz <real>Izx
  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
  INCLUDE NODES IN <string>surface_name
    [IF <string>field_name <|<=|=|>=|> <real>value]
END [RIGID BODY <string>rb_name]
```

A rigid body can consist of any combination of elements—solid elements, structural elements, and point masses. All nodes associated with a rigid body maintain their 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 can translate and rotate through space, but they cannot deform. Element blocks defining the rigid body do not have to be contiguous. They can also adjoin deformable element blocks. Multiple rigid bodies are allowed in a model.

The non-deformable nature of the rigid body can put it in conflict with other constraints. Refer to Appendix D for information on how conflicting constraints are handled.

Warning: Kinematic boundary conditions prescribed on a rigid body must be applied to the rigid body reference node. Kinematic boundary conditions prescribed on a rigid body that are not prescribed on the rigid body reference node will be unenforced. To apply kinematic boundary conditions to a rigid body, use the `BLOCK =` or the `RIGID BODY =` line command with the rigid body's block id or name, respectively, in the appropriate boundary condition command block. Refer to Chapter 6.

Adagio creates a new node for each rigid body in the analysis. The new nodes are true nodes in that they are associated with solution fields such as displacement, velocity, and rotational velocity. These nodes will appear in a results file along with other nodes. The global node number given to the new nodes is simply the total number of nodes in the mesh plus one, repeated for each new rigid body node.

Specification of a rigid body requires the above command block, which appears in the SIERRA scope, plus the `RIGID BODY` command line that appears in the various `SECTION` command blocks described in this chapter. 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 a `SECTION` command line. In each case, the Section must contain a line such as:

```
  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 (through the Section chosen), 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 SIERRA scope with the form

```
  BEGIN RIGID BODY rigidbody_1
  END RIGID BODY rigidbody_1
```

for our example.

The `RIGID BODY` command block has several optional command lines, composing four groups of commands. One group consists of the `MASS`, `POINT MASS`, `REFERENCE LOCATION`, `POINT INERTIA`, and `INERTIA` command lines, a second group consists of the paired `MAGNITUDE` and `DIRECTION` command lines, a third group consists of the paired `ANGULAR VELOCITY` and `CYLINDRICAL AXIS` command lines, and a final group consists of the `INCLUDE NODES IN` command line. The command block can include none or any combination of these groups. If none of these commands is included, the command block simply supplies the value for `rb_name`.

Input to the `MASS` command line consists of a single real number that defines the total mass of the rigid body. If this line command is not present, the mass of the rigid body will be computed using the elements in the rigid body and their densities. This is the translational mass at the reference location. This command does not change the inertia terms.

The `POINT MASS` command line requires a real number specifying the amount of mass added to the rigid body. Optionally, the location of that mass may be specified with three real numbers. If the location is not given, the additional mass will be placed at the reference location (and will not affect the moments and products of inertia).

The `REFERENCE LOCATION` command line requires three real numbers defining the reference location for the rigid body. If this line command is not present, the center of mass will be used. The center of mass is calculated from the mesh and the element densities.

Input to the `INERTIA` command line consists of six real numbers. If present, this command line will set the inertia for the rigid body. If it is not present, moments and products of inertia are computed for the rigid body based on the reference location of the rigid body and the element masses.

Input to the POINT INERTIA command line consists of six real numbers that define moments (Ixx, Iyy, Izz) and products (Ixy, Iyx, Izx) of inertia to be added to the inertia tensor of the rigid body. 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.

It should be noted, if a rigid body contains an element that has rotational resistance, i.e. shell, beam, etc, the nodal moment of inertia of that element is added to the diagonal components of the rigid body's inertia tensor.

An initial, translational-only velocity for the rigid body reference location should be specified with the MAGNITUDE and DIRECTION command lines. The MAGNITUDE command line gives the magnitude of the initial velocity applied to the reference location, and the DIRECTION command line gives a defined direction.

An initial rotational and translational velocity for a rigid body can be specified with the ANGULAR VELOCITY and CYLINDRICAL AXIS command lines. The ANGULAR VELOCITY command line gives the initial translational velocity of and rotational velocity about the reference location due to an angular velocity about some defined axis given on the CYLINDRICAL AXIS command line. If the defined cylindrical axis passes through the reference location, this command will impart only an initial rotational velocity to the rigid body reference node. If the axis does not pass through the reference location, this command will impart an initial translational velocity on the rigid body node as well as the initial rotational velocity.

The INCLUDE NODES IN command line allows a rigid body to include nodes of a surface or block of the mesh. Optionally, the nodes in the surface or block will be included in the rigid body only if the value of a field on the nodes meets a given criterion. For example, consider the rigid body block below.

```
begin rigid body rigid1
  include nodes in block_1
  include nodes in surface_3 if height = -1.0
end
```

Rigid body rigid1 will include all nodes in block_1 and those nodes on surface_3 whose value of the height field are equal to -1.0. The optional field (height in this case) must be a field known in the analysis. This field may be read in from restart, be a native field initialized upon startup, or be an initialized user-defined field. Since the entire set of nodes in block_1 will be in the rigid body, internal forces will not be computed for elements in block_1.

Adagio automatically outputs quantities such as displacement for the reference location of the rigid body. The name assigned to a rigid body will be used to construct variable names that give the quantities. This lets you identify the output associated with a rigid body based on the name you assigned for the rigid body.

In summary, if you use a rigid body in an analysis, you will do one or more of the following steps:

- Include a RIGID BODY command block in the SIERRA scope. If desired, set reference location, mass, point mass, etc.

- Create a rigid body using one or more element blocks (except PARTICLE or PERIDYNAMICS element blocks). A RIGID BODY command line must appear in the SECTION command block used 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. To include point mass element blocks in a rigid body, a RIGID BODY command line must appear in the SECTION command block used in the PARAMETERS FOR BLOCK command block for those point mass element blocks.

- Associate an initial velocity or initial rotation about an axis with the rigid body, if appropriate. 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.

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, SECTION command block, and the element blocks we want to associate with the rigid body will be as follows:

```
begin solid section hex_section
  rigid body = part_a
end
begin point mass section pm_section
  rigid body = part_a
  volume = 0.1
end

begin parameters for block block_100
  material steel
  solid mechanics model use elastic
  section = hex_section
end
begin parameters for block block_535
  material = aluminum
  solid mechanics model use elastic
  section = hex_section
end
begin parameters for block block_600
  material = mass_for_pointmass
  solid mechanics model use elastic
  section = pm_section
end
```

Suppose we want to have the rigid body initially rotating 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 initial angular velocity specification in the RIGID BODY command block is as follows:

```
begin rigid body part_a
  cylindrical axis = body_axis
  angular velocity = 600
end rigid body part_a
```

Adagio automatically generates and outputs global data associated with the rigid body (e.g. displacement and quaternion). See Section 8.8 for details on global variable output. Individual field components may be output as discussed in Section 8.1.4 and Table 8.2. In general the field component fieldi of the rigid body named part_a will be written to the output file(s) with the name FIELDI_PART_A.

5.3.1.1 Multiple Rigid Bodies from a Single Block

Typically, all of the elements in a block are assigned to a single rigid body. However, it is sometimes necessary to define a very large number of rigid bodies. It would be unwieldy to create a separate block for each rigid body. To avoid this, it is possible to create an element attribute in the input mesh file that contains an integer ID used to denote the rigid body in which each element should belong. The RIGID BODIES FROM ATTRIBUTES command is used to associate the attribute with the rigid body IDs. This command must be used in conjunction with the RIGID BODY command. These commands are used in the context of the section block. They are shown here in the SOLID SECTION block, but they can be used with any section type that supports rigid bodies.

```
BEGIN SOLID SECTION <string>sec_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id to
    <integer>last_id
  RIGID BODY = <string>rigid_body_name
END
```

If the RIGID BODIES FROM ATTRIBUTES command is used, a series of rigid bodies will be created. These rigid bodies are named using the specified rigid_body_name, followed by an underscore (_), and then by an integer ID, which ranges between the specified value of first_id and last_id. Elements having an attribute value equal to a given ID are added to the corresponding rigid body.

When referencing the rigid body name for to apply boundary conditions, it is important to specify the entire name of the rigid body, including the underscore and ID. For example, a name such as part_a_1 would be used in a boundary condition.

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]
```

Adagio 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.) Adagio also automatically gives mass property information for each element block.

In addition to the mass property information that is generated, Adagio 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, Adagio reports the mass and the center of mass in the global coordinate system. It also reports the moments and products of inertia, as computed in the global coordinate system about the center of mass.

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 6.1.1 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 would 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. For example, if two element blocks, `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
  #
  # criterion commands
  CRITERION IS GLOBAL VALUE OF
    <string>var_name <|<=|=|>=|> <real>tolerance
  MATERIAL CRITERION
    = <string list>material_model_names [KILL WHEN
    <integer>num_intg INTEGRATION POINTS REMAIN]

  SUMMARY OUTPUT STEP INTERVAL = <integer>output_step_interval
  SUMMARY OUTPUT TIME INTERVAL = <real>output_time_interval

  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
  #
  # cohesive zone setup commands
  COHESIVE SECTION = <string>sect_name
  COHESIVE MATERIAL = <string>mat_name
  COHESIVE MODEL = <string>model_name
  COHESIVE ZONE INITIALIZATION METHOD = <string>NONE|
    ELEMENT STRESS AVG(NONE)

  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.This command block is located within the `ADAGIO REGION` scope. 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.

When an element is killed, the contribution of that element's mass to the attached nodal mass is removed from the attached nodes. 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 all of the elements in a region are killed, the analysis will terminate.

In Adagio elements may be killed based off a global variable or a material state variable and must be used in conjunction with control failure in a multilevel solver block. Control failure is described in Section 3.7. Element death will not be activated unless control failure is specified.

The `ELEMENT DEATH` command block contains five groups of commands—block set, criteria, evaluation, miscellaneous, and cohesive zone setup. The command block must contain commands from the block set and criteria groups. Command lines from the evaluation and miscellaneous groups are optional, as are the cohesive zone commands.

Following are descriptions of the different command groups, 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 6.1.1.

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 `INCLUDE ALL BLOCKS` command line adds all the element blocks present in the region to the current element death definition. 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 Criterion Commands

Both `CRITERION IS GLOBAL` and `MATERIAL CRITERION` can be specified within a single `ELEMENT DEATH` command block. If multiple death criteria are specified for a given element, it will be killed when the first of those criteria are met. In other words, element death with multiple criteria is an `OR` condition rather than an `AND` condition.

5.5.2.1 Global Death Criterion

```
CRITERION IS GLOBAL VALUE OF
```

```
<string>var_name[(<integer>component_num)]
<|<=|=|>=|> <real>tolerance
```

Any global variable may be used in 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 or element criterion command line.

The input parameters are described as follows:

- The string `var_name` gives the name of the global variable. See Section 8.9 for a listing of the global variables.
- Parenthesis syntax may be used in the variable name to specify specific components of the variable, see 8.1
- 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`.

5.5.2.2 Material Death Criterion

```
MATERIAL CRITERION = <string list>material_model_names [KILL WHEN
  <integer>num_intg INTEGRATION POINTS REMAIN]
```

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 can speed computations, enhance visualization, and prevent spurious inversion of these elements that may stop the analysis.

The material models currently supported for use with the `MATERIAL CRITERION` command line are:

- `ELASTIC_FRACTURE` (Solid only, see Section 4.2.4)
- `DUCTILE_FRACTURE` (Solid only, see Section 4.2.7)
- `ML_EP_FAIL` (Solid and Shell, see Section 4.2.9)

Element death will kill an element based on a material criterion when the material model indicates that the element is failed and can carry no more load. For a single integration point element, this occurs when the one integration point in the element fails. For elements with multiple integration points, the default behavior is for the element to be killed when all but one of the integration points has failed. This behavior is the default because for multi-integration point elements, particularly shells, if there is only a single integration point left, the element is severely under-integrated. The final integration point will generally not attract more load and will never fail. This default behavior can be changed by using the optional `KILL WHEN num_intg INTEGRATION POINTS REMAIN` command. In this command, `num_intg` specifies the number of remaining integration points when the element is to be killed.

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 4.2.7). 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 4.2.9). 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 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.

5.5.3.1 Summary Output Commands

At the end of a run, a summary of all element death blocks is output to the log file. The `SUMMARY OUTPUT STEP INTERVAL` or `SUMMARY OUTPUT STEP INTERVAL` commands can be used to request that the summary be output to the log file at regular intervals during the run. The `SUMMARY OUTPUT TIME INTERVAL` command results in the summary being printed every `output_step_interval` steps, while the `SUMMARY OUTPUT TIME INTERVAL` results in the summary being printed once every `output_time_interval` time units. These two commands can be supplied individually, together or not at all. If neither are used, the summary is printed only at the end of execution.

It should be noted that this command applies to all element death blocks. If these commands appear in multiple element death blocks, the values specified in the last instance of each of these commands prevails.

5.5.3.2 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.

5.5.3.3 Death Method

```
DEATH METHOD = <string>DEACTIVATE ELEMENT|DEACTIVATE NODAL MPCS|
  DISCONNECT ELEMENT|INSERT COHESIVE ZONES(DEACTIVATE ELEMENT)
```

The DEATH METHOD command specifies what happens when an element meets the death criterion. The following strings can be used as arguments to this command: DEACTIVATE ELEMENT, DEACTIVATE NODAL MPCS, DISCONNECT ELEMENT, and INSERT COHESIVE ZONES. The behavior controlled by these options is described below.

- With the default option, DEACTIVATE ELEMENT, the element is deactivated, effectively removing it from the mesh.

- The DEACTIVATE NODAL MPCS option removes the nodes of a killed element from any multi-point constraints. This only has an effect if the element has nodes that are in multi-point constraints. The multi-point constraint deactivation option can be used to break an element away from the mesh, allowing it to move independently. It can also be used to activate cohesive zones.

- The DISCONNECT ELEMENT option disconnects the element from the mesh, allowing the element to move independently. The disconnected element will no longer share any nodes with neighboring elements, and will only interact with the remainder of the mesh through contact.

- The INSERT COHESIVE ZONE option disconnects the element from the mesh and places a cohesive zone between the element and each face-adjacent neighbor. If this option is used, the commands COHESIVE SECTION, COHESIVE MATERIAL, and COHESIVE MODEL must be used to define the type of cohesive zone to be inserted, and are documented in Section 5.5.4.

When using element disconnection, it is necessary to use at least some hourglass stiffness and/or viscosity to prevent large deformations of the disconnected elements in zero energy modes. In addition, it is recommended that a secondary shape-based criterion be used in a separate element death command block to fully remove the disconnected elements if they begin to deform too much. The following example of an additional element death block to be used together with element disconnection shows recommended shape criteria:

```
begin element death
  include all blocks
  #Kill an element if it has negative volume
  death on inversion = on
  #Kill an element if it has become locally inverted (i.e., concave)
  criterion is element value of nodal_jacobian_ratio <= 0.0
  #Kill an element that is becoming very large (shells only)
  criterion is element value of perimeter_ratio > 10.0
end
```

5.5.3.4 Active Periods

The following command lines can optionally be used in the `ELEMENT DEATH` command block:

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

These command lines determine when element death is active. See Section 2.5 for more information about these optional command lines.

5.5.4 Cohesive Zone Setup Commands

The commands listed here are all related to adaptive insertion or activation of cohesive zones by element death.

```
COHESIVE SECTION = <string>sect_name
COHESIVE MATERIAL = <string>mat_name
COHESIVE MODEL = <string>model_name
COHESIVE ZONE INITIALIZATION METHOD = <string>NONE|
  ELEMENT STRESS AVG(NONE)
```

The first three of these commands are only applicable to adaptive insertion of cohesive elements by element death. They are used together to fully define the properties of the elements that are adaptively inserted.

The `COHESIVE SECTION` command is used to specify the name of a section used to define the section properties of the cohesive zone elements to be adaptively inserted. See Section 5.2.2 for a description of cohesive sections.

The COHESIVE MATERIAL command is used to specify the name of the material model to be used for the newly-created cohesive elements. This material model name is the user-provide name for the cohesive zone material provided by the parameter mat_name in the BEGIN PROPERTY SPECIFICATION FOR MATERIAL mat_name command block. The material models available for cohesive zones are documented in Section 4.3.

The COHESIVE MODEL command is used to select the material model to be used for the newly-created cohesive elements. This references the name of the material model model_name defined in a BEGIN PARAMETERS FOR MODEL model_name block. The material models available for cohesive zones are documented in Section 4.3.

The COHESIVE ZONE INITIALIZATION METHOD command controls the initialization of cohesive zones that are dynamically inserted or activated through element death. This command should only be used if there are cohesive zones between elements and all nodes of those elements are initially attached together via multi-point constraints, or the INSERT COHESIVE ZONES death method is being used. When element death is used to deactivate the constraints or insert a cohesive element (with the DEATH METHOD = DEACTIVATE NODAL MPCS option or DEATH METHOD = INSERT COHESIVE ZONES option), the exposed cohesive zone can be given an initial state. The options are to either do nothing (NONE) or to initialize the tractions in the cohesive element based on the stresses in the two elements on either side of the cohesive zone (ELEMENT STRESS AVG). How the cohesive zone uses the initial traction will depend on the cohesive surface material model used.

5.5.5 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_STATUS. Including the command line

```
ELEMENT DEATH_STATUS 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_STATUS is as follows: An element with a value of 1.0 for DEATH_STATUS is a living element. An element with a value of 0.0 for DEATH_STATUS is a dead element. A value less than 0.0 indicates that the element was killed due to a code related issue (e.g. an unsupported geometry issue related to ACME). 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_STATUS. Whether or not an element can have a value for DEATH_STATUS 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_STATUS can be some value between 0.0 and 1.0.

If DEATH_STATUS is written to a results file, and the results file is used in a visualization program to examine the mesh for the model, it is possible to use DEATH_STATUS to exclude killed elements from any view of the model. A subset of the mesh showing just the living elements can be created by visualizing only those elements for which DEATH_STATUS = 1.0. The procedure for visualizing results in this way varies for different postprocessing tools.

When an analysis is using element death the log file contains a table of marker values that will be applied to dead elements. The marker values allow determining which elements where killed and by which criterion. The marker variables are stored in the element variable KILLED_BY_CRITERION and are available for output on the mesh results file. Additionally, a global count of how many elements were killed by each criterion is printed at the end of the run log file.

5.6 Explicitly Computing Derived Quantities

```
BEGIN DERIVED OUTPUT
  COMPUTE AND STORE VARIABLE =
    <string>derived_quantity_name
END DERIVED OUTPUT
```

The above command block is used to explicitly compute and store a derived quantity into an internal field. This is useful if the field is needed by an outside capability such as a transfer or error estimation.

For example, to use a derived quantity in a transfer, you must use a `DERIVED OUTPUT` command block. To transfer the von Mises stress norm, you would use the following command block:

```
BEGIN DERIVED OUTPUT
  COMPUTE AND STORE VARIABLE = von_mises
END DERIVED OUTPUT
```

Tables 8.10 through 8.13 in Section 8 list element variables available for different types of elements. Variables that are only computed at user request are designated with a `yes` in the `Derived` column of these tables. These are the quanities that must be listed in a `BEGIN DERIVED OUTPUT` command block if they are to be transferred during a coupled analysis.

5.7 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. pdf.

2. Rashid, M. M. "Incremental Kinematics for Finite Element Applications." *International Journal for Numerical Methods in Engineering* 36 (1993): 3937–3956. doi.

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. doi.

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. doi.

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. doi.

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. pdf.

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. doi.

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

9. Scherzinger, W. M., and D. C. Hammerand. *Constitutive Models in LAME*, SAND2007-5873. Albuquerque, NM: Sandia National Laboratories, September 2007. pdf.

10. 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. pdf.

11. Sjaardema, G. D. *Overview of the Sandia National Laboratories Engineering Analysis Code Access System*, SAND92-2292. Albuquerque, NM: Sandia National Laboratories, January 1993. pdf.

12. 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. pdf.

13. Karen Devine, Erik Boman, Robert Heapby, Bruce Hendrickson, Courtenay Vaughan, "Zoltan Data Management Service for Parallel Dynamic Applications." *Computing in Science and Engineering* 4 (2002): 90–97. doi. See also ZOLTAN web site. link.

14. Monaghan, J. “SPH without a tensile instability.” *Journal of Computational Physics* 12, (2000): 622. doi.

15. Q. Yang, A. Mota, M. Ortiz. “A class of variational strain-localization finite elements.” *International Journal for Numerical Methods in Engineering*, (2005) 62:1013-1037. doi.

Chapter 6

Boundary Conditions and Initial Conditions

Adagio 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. Adagio offers a flexible means to define subsets of mesh entities. Section 6.1.1 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 variables in Adagio. Adagio 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 a solid mechanics code (fixed displacement, prescribed acceleration, etc.) are options in Adagio and described in Section 6.3. Most of these boundary conditions let you specify a time history using a function, a user subroutine, or by reading values from a mesh file.

- Section 6.4 documents a number of initial velocity options available in Adagio.

- Force boundary conditions typical of those you would expect in a solid mechanics code (prescribed force, traction, etc.) are options in Adagio and described in Section 6.5. Most of these force boundary conditions let you specify a time history using a function or a user subroutine.

- Section 6.6 discusses the gravity load option. A gravity load is a body force boundary condition.

- Section 6.7 details a number of options available for describing a temperature field in Adagio.

- Section 6.8 details the options available for describing a pore pressure field.

- Section 6.10 describes a number of specialized boundary conditions.

6.1 General Boundary Condition Concepts

There are general principles that apply to all of the available types of boundary conditions. To apply a boundary condition, a set of mesh entities and the magnitude and/or direction in which it is to be applied must be specified. Adagio provides several methods for both specifying the set of mesh entities and for prescribing how the boundary condition is to be applied. The general concepts on how this is done are applicable to all of the boundary condition types, and are described in the following sections.

6.1.1 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 boundary, initial, or load condition. All these command lines exist within the command blocks for the various prescribed conditions, 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 an initial, boundary, or load condition. A condition 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 or their attached nodes with a boundary, initial, or load condition. A condition may be applied to multiple surfaces by putting multiple surface names on the command line or by repeating the command line multiple times. 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 and its nodes and faces with a boundary condition. A boundary condition 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` must be used for kinematic boundary conditions on rigid bodies. If a block has been defined as a rigid body, the specified kinematic condition will be applied to the rigid body reference node.

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 and hence all elements and nodes in the model with a boundary, initial, or load condition. 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 node or element face may appear in more than one node set, surface or block. However, the prescribed condition is applied to each node or face only once. For example, node 67 may be a part of nodelist 2 and surface 3 but the boundary condition will only be applied to node 67 once.

The set of mesh entities associated with a boundary, initial, or load condition 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 the nodes in the specified node set from the set of nodes used by the condition.

The `REMOVE SURFACE` command line deletes a set of element faces and their associated nodes from the set of element faces used by the prescribed condition.

The `REMOVE BLOCK` command line deletes a set of elements and their associated nodes from the set of elements used by the prescribed condition.

6.1.2 Methods for Specifying Boundary Conditions

There are three main methods which can be used to prescribe most types of boundary conditions available in Adagio.

- The boundary condition can be prescribed using commands in the input file. These commands are categorized as "specification commands" in this document. Depending on the type of the boundary condition, it is be necessary to prescribe its direction and/or magnitude. Boundary conditions can be specified this way when a set of mesh entities is to experience a similar condition with a time variation that can be expressed by a function. One of the following commands is used to specify the direction of the boundary condition: `COMPONENT`, `DIRECTION`, `CYLINDRICAL AXIS`, or `RADIAL AXIS`. The magnitude is defined using one of `MAGNITUDE`, `FUNCTION` or `ANGULAR VELOCITY`. These commands are used in various combinations depending on the type of the boundary condition. The details of how to use them are provided in the descriptions of the various boundary condition types.

- If the nature of the boundary condition is such its variation in time and space can not be described easily by the combination of a function and a direction, it may be necessary to use a user-defined subroutine. User subroutines provide a very general capability to define how kinematic or force boundary conditions are applied. The use of user-defined subroutines does increase the complexity of defining the model, however. The user must write and debug the subroutine and compile and link it in with Adagio. Because of the added complexity, user subroutines should only be used if the needed capability is not provided by the other methods of prescribing boundary conditions.

- For some types of boundary conditions, the values of the field to be prescribed can be read in from an existing output database. This is often used as a method to transfer results from one analysis code to another. One of the common uses for this capability is to compute temperatures using a thermal code, and then transfer the temperature fields to Adagio to study combined mechanical and thermal effects. This capability can be used either to read in initial values or to read in a series of values that vary over time.

In the following sections describing specific types of boundary conditions, the commands are grouped according to these three categories.

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
  #
  # specification command
  MAGNITUDE = <real list>initial_values
  #
  # probability distribution commands
  DISTRIBUTION = WEIBULL PARAMETERS = <real list>dist_values
    SEED = <integer>dist_seed
  DISTRIBUTION REFERENCE = NODE|EDGE|FACE|ELEMENT|GLOBAL
    <string>size_var_name VALUE = <real>size_ref_val
  #
  # external database commands
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_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]
```

Adagio 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. The initialization is performed both before and after the element and material model initialization. This allows the elements and material models to compute other initial variables based on variables specified by the user and also ensures that the variables specified by the user are not overwritten by the elements and material models. However, there is minimal checking in Adagioto 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 6.1.1 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 Adagio. The variable can be any currently defined variable in Adagio or any user-defined variable created with the

USER VARIABLE command block (see Section 10.2.4).

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 Adagio; 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 Specification Command

If the constant magnitude command is used, one or more initial values are specified directly in the command block. This is done using the following command line:

```
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 Probability Distribution Commands

The field to be initialized can optionally be populated with random numbers generated to conform to a specified probability distribution function. This is accomplished by including the following command:

```
DISTRIBUTION = WEIBULL PARAMETERS = <real list>dist_values
  SEED = <integer>dist_seed
```

Currently, the Weibull distribution is the only supported probability distribution, but this command is intended to eventually support other types of probability distributions. The parameters that describe the probability distribution, `dist_values`, are provided as a list of real numbers.

There are two parameters to the Weibull distribution: the shape parameter, k, and the scale parameter λ. In this implementation, only one parameter, k is input in `dist_values`. The scale parameter λ is hardcoded to be 1.0.

The `DISTRIBUTION` command must be used in conjunction with the `MAGNITUDE` command. The specified magnitude is applied as a scale factor to the randomly generated numbers. Thus, the scale parameter for the Weibull distribution is specified through the `MAGNITUDE` command.

If a field is populated with randomly generated values conforming to the Weibull distribution, the realizations of random values at each point can be scaled to account for the size of the mesh at that location. This is accomplished by including the following command:

```
DISTRIBUTION REFERENCE = NODE|EDGE|FACE|ELEMENT|GLOBAL
  <string>size_var_name VALUE = <real>size_ref_value
```

In this command, `size_var_name` is the name of a variable, which can be of either nodal, edge, face, or element, or global type, and which should represent the size of the mesh at each point. The element `volume` field is a good example of such a variable. The reference value: `size_ref_value`, is a reference size for which no scaling is performed.

The randomly generated value for the variable is multiplied by the factor α, which is computed as:

$$\alpha = \left(\frac{v_i}{v_{ref}}\right)^{-\frac{1}{k}} \tag{6.1}$$

where v_i is the value of a field variable specified in `size_var_name` at the current point i, v_{ref} is the reference size, `size_ref_value`, and k is the Weibull shape parameter (also referred to as the Weibull modulus). The result of this scaling is that realizations of the random variable on larger elements are scaled down and those on smaller elements are scaled up. This accounts for the fact that a defect is more likely to be found within the volume represented by an element as the size of that element increases.

6.2.5 External Mesh Database Commands

If the external database option is used, the initial values for a variable are read from an external mesh database. 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.

The values are read from a finite element model defined via the `FINITE ELEMENT MODEL` command block described in Section 5.1. The finite element model can either be the model used by the region for its mesh definition as specified with the `USE FINITE ELEMENT MODEL` command (see Section 2.3), or it can be a different (but compatible) model. The following command lines control the use of an external mesh database to prescribe initial conditions:

```
READ VARIABLE = <string>var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
TIME = <real>time
```

The `READ VARIABLE = <string>var_name` command is used to read the variable from the region's finite element mesh database. The `var_name` string specifies the name of the variable as it appears on the mesh database. 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.

If the analysis will cause topology modifications due to the use of load balancing, remeshing, or other techniques that can result in nodes and/or elements being moved among processors, then the `COPY VARIABLE` command should be used. This command specifies that the variable named `var_name` will be used to specify the initial conditions. The `FROM MODEL <string>model_name` portion of the command is optional. If it is specified, the results are read from the mesh database named `model_name`. Otherwise, the region's finite element mesh database will be used as the model.

The primary difference between the behavior of the `COPY VARIABLE` command and the `READ VARIABLE` command is that the copy command handles topology modifications caused by load balancing and remeshing, which can move nodes and elements to different processors. The `COPY VARIABLE` command will correctly match the nodes from the external database to the corresponding nodes in the analysis and communicate the data between processors if needed. Note that the nodes in the two finite element models must match spatially, but they can be distributed on different processors.

The variable name used on the mesh file can be arbitrary. The name can be identical to or different from the variable name specified on the `INITIALIZE VARIABLE NAME` command line.

The field to be read may be specified at an arbitrary number of different times on the mesh file. The default behavior is to use the value of the variable at the initial time in the analysis to prescribe the initial condition. The time history is interpolated as needed for an initial analysis time that does not correspond exactly to a time on the mesh file. The `TIME` command line can optionally be used to select a specific time to initialize a variable. 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.

6.2.6 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 Adagio 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 10.2.2 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 10.

The application of user subroutines for variable initialization is essentially the same as the application of user subroutines in general. See Section 6.3.9 and Chapter 10 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.

6.2.7 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 Adagio are described in this section. The kinematic boundary conditions are nested inside the region scope.

Kinematic constraints can potentially be in conflict with other constraints. Refer to Appendix D for information on how conflicting constraints are handled.

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
  #
  # specification commands
  COMPONENT = <string>X/Y/Z | COMPONENTS = <string>X/Y/Z
  #
  # additional commands
  ACTIVE PERIODS = <string list>period_names
  INACTIVE 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 are two additional command lines: `ACTIVE PERIODS` and `INACTIVE PERIODS`. These are used to activate or deactivate this kinematic boundary condition for certain time periods.

For rigid bodies, any kinematic boundary condition is applied directly to the rigid body reference node. The kinematics for the nodes attached to the rigid body are applied via the rigid body constraint conditions. Since rigid bodies are defined using element blocks, only the `BLOCK` command may be used to specify fixed displacement conditions on rigid bodies.

Following are descriptions of the different command groups.

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 6.1.1 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 Specification Commands

There are two component specification commands available in the `FIXED DISPLACEMENT` command block:

```
COMPONENT = X/Y/Z | COMPONENTS = 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 Commands

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines can optionally appear in the `FIXED DISPLACEMENT` command block:

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

These command lines determine when the boundary condition is active. See Section 2.5 for more information about these optional command lines.

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
  #
  # specification 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
  #
  # external database commands
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  TIME = <real>time
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE 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 set of nodes. 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 specification commands in the above command block can be used to specify the displacement field. If the displacement field is more complex, a user subroutine is used to specify the displacement field. The displacement field can also be read from an external mesh database. In a given boundary condition command block, commands from only one of the command groups (specification commands, user subroutine commands, or external database commands) may be used.

For rigid bodies, any kinematic boundary condition is applied directly to the rigid body reference node. The kinematics for the mesh nodes attached to the rigid body are applied via the rigid body constraint conditions. Since rigid bodies are defined using element blocks, only the `BLOCK` command may be used to specify prescribed displacement conditions on rigid bodies.

The `PRESCRIBED DISPLACEMENT` command block contains four groups of commands—node set, function, user subroutine, and external database. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are three additional command lines: `SCALE FACTOR`, `ACTIVE PERIODS`, and `INACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with the specification commands the user subroutine commands, or the external database command. The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines are 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 6.1.1 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 Specification Commands

If the specification commands are 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 used to specify the prescribed displacement with a direction and a function:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z |
  CYLINDRICAL AXIS = <string>defined_axis |
  RADIAL AXIS = <string>defined_axis
FUNCTION = <string>function_name
```

The displacement can be specified along an arbitrary user-defined direction, along a component direction (X, Y, or Z), along the azimuthal direction in a cylindrical coordinate system (defined in reference to an axis), or along a radial direction (defined in reference to an axis). Only one of these options (i.e., command lines) is allowed. The displacement is prescribed only in the specified direction. A prescribed displacement boundary condition does not influence the motion in directions orthogonal to the prescribed direction.

- The `DIRECTION` command line is used to prescribe displacement in an arbitrary user-defined direction. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA scope.

- 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 is used to specify that the prescribed displacement is to be applied in the azimuthal direction of a cylindrical coordinate system. The string `defined_axis` refers to the name of the axis of the cylindrical coordinate system, and which is defined via a `DEFINE AXIS` command block in the SIERRA scope. The displacement is prescribed as a rotation in radians about the axis. Nodes with this type of boundary condition are free to move in the radial and height directions in the cylindrical coordinate system. Restraints can be placed on the node set in those directions if desired by applying separate kinematic boundary conditions that contain `RADIAL AXIS` or `DIRECTION` commands that refer to the same axis. Note that this type of boundary condition is not a rotational boundary condition; it only affects translational degrees of freedom.

  ⚠ **Known Issue:** If a prescribed displacement with the `CYLINDRICAL AXIS` option is applied to nodes that fall on the axis, it will have no effect. Separate boundary conditions should be applied to those nodes to fix them in the plane normal to the axis.

- The `RADIAL AXIS` command line requires an axis definition that appears in the SIERRA scope. The string `defined_axis` uses an `axis_name` that is defined in the SIERRA 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 SIERRA scope using 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 6.3.2.5.

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 do the specification commands, 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 Adagio 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 10.2.2 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 10.

The magnitude set in the user subroutine can be scaled by use of the SCALE FACTOR command line, as described in Section 6.3.2.5.

See Section 6.3.9 and Chapter 10 for more details on implementing the user subroutine option.

6.3.2.4 External Mesh Database Commands

If the external database option is used, the displacement vector (or specified components of the vector) is read from an external mesh database. The displacements are read from a finite element model defined via the FINITE ELEMENT MODEL command block described in Section 5.1. The finite element model can either be the model used by the region for its mesh definition as specified with the `USE FINITE ELEMENT MODEL` command (see Section 2.3), or it can be a different (but compatible) model. The following command lines control the use of an external mesh database to prescribe the displacement:

```
    READ VARIABLE = <string>var_name
    COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
    TIME = <real>time
```

The `READ VARIABLE = <string>var_name` command is used to read the displacement vector from the region's finite element mesh database. The `var_name` string specifies the name of the variable as it appears on the mesh database.

If the analysis will cause topology modifications due to the use of load balancing, remeshing, or other techniques that can result in nodes and/or elements being moved among processors, then the `COPY VARIABLE` command should be used. This command specifies that the variable named `var_name` will be used to specify the displacement. The `FROM MODEL <string>model_name` portion of the command is optional. If it is specified, the results are read from the mesh database named `model_name`. Otherwise, the region's finite element mesh database will be used as the model.

The primary difference between the behavior of the `COPY VARIABLE` command and the `READ VARIABLE` command is that the copy command handles topology modifications caused by load balancing and remeshing, which can move nodes and elements to different processors. The `COPY VARIABLE` command will correctly match the nodes from the external database to the corresponding nodes in the analysis and communicate the data between processors if needed. Note that the nodes in the two finite element models must match spatially, but they can be distributed on different processors.

The field to be read may be specified at an arbitrary number of different times on the mesh file. The default behavior is for the time history of the variable read from the file to be used to prescribe the boundary condition. The time history is interpolated as needed for analysis times that do not correspond exactly to times on the mesh file. The `TIME` command line can optionally be used to specify that the boundary condition be constant over time, based on the value of the variable read from the file at the specified time. 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.

6.3.2.5 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
    INACTIVE 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 specification commands or the user subroutine. 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` and `INACTIVE PERIODS` command lines determine when the boundary condition is active. See Section 2.5 for more information about these command lines.

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
  #
  # specification 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
  #
  # external database commands
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  TIME = <real>time
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE 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 specification commands in the above command block can be used to specify the velocity field. If the velocity field is more complex, a user subroutine is used to specify the velocity field. The velocity field can also be read from an external mesh database. In a given boundary condition command block, commands from only one of the command groups (specification commands, user subroutine commands, or external database commands) may be used.

For rigid bodies, any kinematic boundary condition is applied directly to the rigid body reference node. The kinematics for the mesh nodes attached to the rigid body are applied via the rigid body constraint conditions. Since rigid bodies are defined using element blocks, only the `BLOCK` command may be used to specify prescribed velocity conditions on rigid bodies.

The `PRESCRIBED VELOCITY` command block contains four groups of commands—node set, function, user subroutine, and external database. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are three additional command lines: `SCALE FACTOR`, `ACTIVE PERIODS`, and `INACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with either the specification commands or the user subroutine commands. The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines are used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups.

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 6.1.1 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 Specification Commands

If the specification commands are 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 used to specify the prescribed velocity with a direction and a function:

```
    DIRECTION = <string>defined_direction |
    COMPONENT = <string>X|Y|Z |
    CYLINDRICAL AXIS = <string>defined_axis |
    RADIAL AXIS = <string>defined_axis
  FUNCTION = <string>function_name
```

The velocity can be specified along an arbitrary user-defined direction, along a component direction (X, Y, or Z), along the azimuthal direction in a cylindrical coordinate system (defined in reference to an axis), or along a radial direction (defined in reference to an axis). Only one of these options (i.e., command lines) is allowed. The velocity is prescribed only in the specified direction. A prescribed velocity boundary condition does not influence the motion in directions orthogonal to the prescribed direction.

- The `DIRECTION` command line is used to prescribe velocity in an arbitrary user-defined direction. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA scope.

- The `COMPONENT` command line is used to specify that the prescribed 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 is used to specify that the prescribed velocity is to be applied in the azimuthal direction of a cylindrical coordinate system. The string `defined_axis` refers to the name of the axis of the cylindrical coordinate system, and which is defined via a `DEFINE AXIS` command block in the SIERRA scope. The velocity is prescribed as a rotation in radians about the axis. Nodes with this type of boundary condition are free to move in the radial and height directions in the cylindrical coordinate system. Restraints can be placed on the node set in those directions if desired by applying separate kinematic boundary conditions that contain `RADIAL AXIS` or `DIRECTION` commands that refer to the same axis. Note that this type of boundary condition is not a rotational boundary condition; it only affects translational degrees of freedom.

  Known Issue: If a prescribed velocity with the `CYLINDRICAL AXIS` option is applied to nodes that fall on the axis, it will have no effect. Separate boundary conditions should be applied to those nodes to fix them in the plane normal to the axis.

- The `RADIAL AXIS` command line requires an axis definition that appears in the SIERRA scope. The string `defined_axis` uses an `axis_name` that is defined in the SIERRA 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 SIERRA scope using 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 6.3.3.5.

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 do the specification commands, 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 Adagio 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 10.2.2 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 10.

The magnitude set in the user subroutine can be scaled by use of the `SCALE FACTOR` command line, as described in Section 6.3.3.5.

6.3.3.4 External Mesh Database Commands

If the external database option is used, the velocity vector (or specified components of the vector) is read from an external mesh database. The velocities are read from a finite element model defined via the `FINITE ELEMENT MODEL` command block described in Section 5.1. The finite element model can either be the model used by the region for its mesh definition as specified with the `USE FINITE ELEMENT MODEL` command (see Section 2.3), or it can be a different (but compatible) model. The following command lines control the use of an external mesh database to prescribe the velocity:

```
READ VARIABLE = <string>var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
TIME = <real>time
```

The READ VARIABLE = <string>var_name command is used to read the velocity vector from the region's finite element mesh database. The var_name string specifies the name of the variable as it appears on the mesh database.

If the analysis will cause topology modifications due to the use of load balancing, remeshing, or other techniques that can result in nodes and/or elements being moved among processors, then the COPY VARIABLE command should be used. This command specifies that the variable named var_name will be used to specify the velocity. The FROM MODEL <string>model_name portion of the command is optional. If it is specified, the results are read from the mesh database named model_name. Otherwise, the region's finite element mesh database will be used as the model.

The primary difference between the behavior of the COPY VARIABLE command and the READ VARIABLE command is that the copy command handles topology modifications caused by load balancing and remeshing, which can move nodes and elements to different processors. The COPY VARIABLE command will correctly match the nodes from the external database to the corresponding nodes in the analysis and communicate the data between processors if needed. Note that the nodes in the two finite element models must match spatially, but they can be distributed on different processors.

The field to be read may be specified at an arbitrary number of different times on the mesh file. The default behavior is for the time history of the variable read from the file to be used to prescribe the boundary condition. The time history is interpolated as needed for analysis times that do not correspond exactly to times on the mesh file. The TIME command line can optionally be used to specify that the boundary condition be constant over time, based on the value of the variable read from the file at the specified time. 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.

6.3.3.5 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
    INACTIVE 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 specification commands or the user subroutine. 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 and INACTIVE PERIODS command lines determine when the boundary condition is active. See Section 2.5 for more information about these command lines.

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
  #
  # specification 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
  #
  # external database commands
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  TIME = <real>time
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE 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 specification commands 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 is used to specify the acceleration field. The acceleration field can also be read from an external mesh database. In a given boundary condition command block, commands from only one of the command groups (specification commands, user subroutine commands, or external database commands) may be used.

For rigid bodies, any kinematic boundary condition is applied directly to the rigid body reference node. The kinematics for the mesh nodes attached to the rigid body are applied via the rigid body constraint conditions. Since rigid bodies are defined using element blocks, only the `BLOCK` command may be used to specify prescribed acceleration conditions on rigid bodies.

The `PRESCRIBED ACCELERATION` command block contains four groups of commands—node set, function, user subroutine, and external database. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are three additional command lines: `SCALE FACTOR`, `ACTIVE PERIODS`, and `INACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with either the specification commands or the user subroutine commands. The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines are used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups.

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 6.1.1 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 Specification Commands

If the specification commands are 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 used to specify the prescribed acceleration with a direction and a function:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z
FUNCTION = <string>function_name
```

The acceleration can be specified either along an arbitrary user-defined direction or along a component direction (X, Y, or Z), but not both. The acceleration is prescribed only in the specified direction. A prescribed acceleration boundary condition does not influence the motion in directions orthogonal to the prescribed direction.

- The `DIRECTION` command line is used to prescribe acceleration in an arbitrary user-defined direction. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA scope.
- The `COMPONENT` command line is used to specify that the prescribed 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 SIERRA scope using 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 6.3.4.5.

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 do the specification commands, 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 Adagio 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 10.2.2 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 10.

The magnitude set in the user subroutine can be scaled by use of the `SCALE FACTOR` command line, as described in Section 6.3.4.5.

See Section 6.3.9 and Chapter 10 for more details on implementing the user subroutine option.

6.3.4.4 External Mesh Database Commands

If the external database option is used, the acceleration vector (or specified components of the vector) is read from an external mesh database. The accelerations are read from a finite element model defined via the `FINITE ELEMENT MODEL` command block described in Section 5.1. The finite element model can either be the model used by the region for its mesh definition as specified with the `USE FINITE ELEMENT MODEL` command (see Section 2.3), or it can be a different (but compatible) model. The following command lines control the use of an external mesh database to prescribe the acceleration:

```
    READ VARIABLE = <string>var_name
    COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
    TIME = <real>time
```

The `READ VARIABLE = <string>var_name` command is used to read the acceleration vector from the region's finite element mesh database. The `var_name` string specifies the name of the variable as it appears on the mesh database.

If the analysis will cause topology modifications due to the use of load balancing, remeshing, or other techniques that can result in nodes and/or elements being moved among processors, then the `COPY VARIABLE` command should be used. This command specifies that the variable named `var_name` will be used to specify the acceleration. The `FROM MODEL <string>model_name` portion of the command is optional. If it is specified, the results are read from the mesh database named `model_name`. Otherwise, the region's finite element mesh database will be used as the model.

The primary difference between the behavior of the `COPY VARIABLE` command and the `READ VARIABLE` command is that the copy command handles topology modifications caused by load balancing and remeshing, which can move nodes and elements to different processors. The `COPY VARIABLE` command will correctly match the nodes from the external database to the corresponding nodes in the analysis and communicate the data between processors if needed. Note that the nodes in the two finite element models must match spatially, but they can be distributed on different processors.

The field to be read may be specified at an arbitrary number of different times on the mesh file. The default behavior is for the time history of the variable read from the file to be used to prescribe the boundary condition. The time history is interpolated as needed for analysis times that do not correspond exactly to times on the mesh file. The `TIME` command line can optionally be used to specify that the boundary condition be constant over time, based on the value of the variable read from the file at the specified time. 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.

6.3.4.5 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
INACTIVE 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 specification commands or the user subroutine. 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` and `INACTIVE PERIODS` command lines determine when the boundary condition is active. See Section 2.5 for more information about these command lines.

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
  #
  # specification commands
  COMPONENT = <string>X/Y/Z | COMPONENTS = <string>X/Y/Z
  #
  # additional commands
  ACTIVE PERIODS = <string list>periods_names
  INACTIVE PERIODS = <string list>periods_names
END [FIXED ROTATION]
```

The `FIXED ROTATION` command is only applied to nodes that have rotational degrees of freedom such as shell element nodes and rigid body reference nodes. In the case of rigid bodies, the boundary condition must be specified on the block that defines the rigid body using the `BLOCK` line command.

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 are additional command lines: `ACTIVE PERIODS` and `INACTIVE PERIODS`. These command lines are used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups.

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 6.1.1 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 Specification Commands

There are two component specification commands available in the `FIXED ROTATION` command block:

```
COMPONENT = X/Y/Z | COMPONENTS = 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 Commands

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines can optionally appear in the `FIXED ROTATION` command block:

```
ACTIVE PERIODS = <string list>period_names
INACTIVE 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
  #
  # specification commands
  DIRECTION = <string>defined_direction
  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
  #
  # external database commands
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL
    <string>model_name]
  TIME = <real>time
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PRESCRIBED ROTATION]
```

Warning: The `BEGIN PRESCRIBED ROTATION` command is only applied to nodes that have rotational degrees of freedom such as shell element nodes and rigid body reference nodes. Displacements consistent with rotation about a fixed axis for nodes that do not have rotational degrees of freedom, are imposed using the `CYLINDRICAL AXIS` command line in the `PRESCRIBED DISPLACEMENT` command block described in Section 6.3.2.

For nodal rotational degrees of freedom, the rotations applied using the `BEGIN PRESCRIBED ROTATION` command are the three components of the rotational degree of freedom itself. In the case of rigid bodies, the rotations must be the nonlinear components that result in the desired quaternion on the rigid body reference node. For rigid bodies, the boundary condition is applied to the reference node and therefore must be specified on the block that defines the rigid body using the `BLOCK` line command.

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 specification commands 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 is used to specify the rotation field. The rotation field can also be read from an external mesh database. In a given boundary condition command block, commands from only one of the command groups (specification commands, user subroutine commands, or external database commands) may be used.

The `PRESCRIBED ROTATION` command block contains four groups of commands—node set, function, user subroutine, and external database. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are three additional command lines: `SCALE FACTOR`, `ACTIVE PERIODS`, and `INACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with either the specification commands or the user subroutine commands. The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines are used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups.

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 6.1.1 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 Specification Commands

If the specification commands are 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 used to specify the prescribed rotation with a direction and a function:

```
DIRECTION = <string>defined_direction
FUNCTION = <string>function_name
```

The `DIRECTION` command line is used to prescribe rotation in an arbitrary user-defined direction. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA scope. The rotation is prescribed only in the specified direction. A prescribed rotation boundary condition does not influence the rotation in directions orthogonal to the prescribed direction.

The magnitude of the rotation is specified by the `FUNCTION` command line. This references a `function_name` (defined in the SIERRA 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 6.3.6.5.

The magnitude of the rotation, as specified by the product of the function and the scale factor, has units of radians per second.

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 do the specification commands, 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 Adagio 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 10.2.2 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 10.

The magnitude set in the user subroutine can be scaled by use of the SCALE FACTOR command line, as described in Section 6.3.6.5.

See Section 6.3.9 and Chapter 10 for more details on implementing the user subroutine option.

6.3.6.4 External Mesh Database Commands

If the external database option is used, the rotation vector (or specified components of the vector) is read from an external mesh database. The rotations are read from a finite element model defined via the FINITE ELEMENT MODEL command block described in Section 5.1. The finite element model can either be the model used by the region for its mesh definition as specified with the USE FINITE ELEMENT MODEL command (see Section 2.3), or it can be a different (but compatible) model. The following command lines control the use of an external mesh database to prescribe the rotation:

```
READ VARIABLE = <string>mesh_var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
TIME = <real>time
```

The READ VARIABLE = <string>var_name command is used to read the rotation vector from the region's finite element mesh database. The var_name string specifies the name of the variable as it appears on the mesh database.

If the analysis will cause topology modifications due to the use of load balancing, remeshing, or other techniques that can result in nodes and/or elements being moved among processors, then the COPY VARIABLE command should be used. This command specifies that the variable named var_name will be used to specify the rotation. The FROM MODEL <string>model_name portion of the command is optional. If it is specified, the results are read from the mesh database named model_name. Otherwise, the region's finite element mesh database will be used as the model.

The primary difference between the behavior of the COPY VARIABLE command and the READ VARIABLE command is that the copy command handles topology modifications caused by load balancing and remeshing, which can move nodes and elements to different processors. The COPY VARIABLE command will correctly match the nodes from the external database to the corresponding nodes in the analysis and communicate the data between processors if needed. Note that the nodes in the two finite element models must match spatially, but they can be distributed on different processors.

The field to be read may be specified at an arbitrary number of different times on the mesh file. The default behavior is for the time history of the variable read from the file to be used to prescribe the boundary condition. The time history is interpolated as needed for analysis times that do not correspond exactly to times on the mesh file. The `TIME` command line can optionally be used to specify that the boundary condition be constant over time, based on the value of the variable read from the file at the specified time. 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.

6.3.6.5 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
INACTIVE 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 specification commands or the user subroutine. 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` and `INACTIVE PERIODS` command lines determine when the boundary condition is active. See Section 2.5 for more information about these command lines.

6.3.7 Prescribed Rotational Velocity

```
BEGIN PRESCRIBED ROTATIONAL 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
  #
  # specification 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
  #
  # external database commands
  READ VARIABLE = <string>variable_name
  COPY VARIABLE = <string>variable_name [FROM MODEL
    <string>model_name]
  TIME = <real>time
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PRESCRIBED ROTATIONAL VELOCITY]
```

Warning: The `PRESCRIBED ROTATIONAL VELOCITY` command is only applied to nodes that have rotational degrees of freedom such as shell element nodes and rigid body reference nodes. Velocities consistent with rotation about a fixed axis for nodes that do not have rotational degrees of freedom, are imposed with the `CYLINDRICAL AXIS` command line in the `PRESCRIBED VELOCITY` command block described in Section 6.3.3.

This command provides the total rotational velocity of the body about an axis.For rigid bodies, the boundary condition is applied to the reference node and therefore must be specified on the block that defines the rigid body using the BLOCK line command.

The PRESCRIBED ROTATIONAL VELOCITY command block prescribes the rotational velocity about an axis for a given set of nodes. The rotational velocity field associates a vector giving the magnitude and direction of the rotational velocity with each node in the node set. The rotational velocity field may vary over time and space. If the rotational velocity field has only a time-varying component, the specification commands in the above command block can be used to specify the rotational velocity field. If the rotational velocity field has both time-varying and spatially varying components, a user subroutine is used to specify the rotational velocity field. The rotational velocity field can also be read from an external mesh database. In a given boundary condition command block, commands from only one of the command groups (specification commands, user subroutine commands, or external database commands) may be used.

The PRESCRIBED ROTATIONAL VELOCITY command block contains four groups of commands—node set, function, user subroutine, and external database. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are three additional command lines: SCALE FACTOR, ACTIVE PERIODS, and INACTIVE PERIODS. The SCALE FACTOR command line can be used in conjunction with either the specification commands or the user subroutine commands. The ACTIVE PERIODS and INACTIVE PERIODS command lines are used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups.

6.3.7.1 Node Set Commands

The node set commands portion of the PRESCRIBED ROTATIONAL VELOCITY command block defines a set of nodes associated with the prescribed rotational 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 6.1.1 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.

6.3.7.2 Specification Commands

If the specification commands are used, the rotational velocity vector at any given time is the same for all nodes in the node set associated with the particular `PRESCRIBED ROTATIONAL VELOCITY` command block. The direction of the rotational velocity vector is constant for all time; the magnitude of the rotational velocity vector may vary with time, however.

Following are the command lines used to specify the prescribed rotational velocity with a direction and a function:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z
FUNCTION = <string>function_name
```

The rotational velocity can be specified either along an arbitrary user-defined direction or along a component direction (X, Y, or Z), but not both. The rotational velocity is prescribed only in the specified direction. A prescribed rotational velocity boundary condition does not influence the rotational velocity in directions orthogonal to the prescribed direction.

- The `DIRECTION` command line is used to prescribe rotational velocity in an arbitrary user-defined direction. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA scope.
- The `COMPONENT` command line is used to specify that the prescribed rotational 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 magnitude of the rotational velocity is specified by the `FUNCTION` command line. This references a `function_name` (defined in the SIERRA scope in a `DEFINITION FOR FUNCTION` command block) that specifies the magnitude of the rotational velocity vector as a function of time. The magnitude can be scaled by use of the `SCALE FACTOR` command line described in Section 6.3.7.5.

The magnitude of the rotational velocity, as specified by the product of the function and the scale factor, has units of radians per second.

6.3.7.3 User Subroutine Commands

If the user subroutine option is used, the rotational velocity vector may vary spatially at any given time for each of the nodes in the node set associated with the particular `PRESCRIBED ROTATIONAL VELOCITY` command block. The user subroutine option allows for a more complex description of the rotational velocity field than do the specification commands, 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 rotational velocity direction and a magnitude for every node to which the boundary condition will be applied. The subroutine will be called by Adagio at the appropriate time to generate the rotational 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 10.2.2 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 10.

The magnitude set in the user subroutine can be scaled by use of the `SCALE FACTOR` command line, as described in Section 6.3.7.5.

See Section 6.3.9 and Chapter 10 for more details on implementing the user subroutine option.

6.3.7.4 External Mesh Database Commands

If the external database option is used, the rotational velocity vector (or specified components of the vector) is read from an external mesh database. The rotational velocities are read from a finite element model defined via the `FINITE ELEMENT MODEL` command block described in Section 5.1. The finite element model can either be the model used by the region for its mesh definition as specified with the `USE FINITE ELEMENT MODEL` command (see Section 2.3), or it can be a different (but compatible) model. The following command lines control the use of an external mesh database to prescribe the rotational velocity:

```
READ VARIABLE = <string>mesh_var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
TIME = <real>time
```

The `READ VARIABLE = <string>var_name` command is used to read the rotational velocity vector from the region's finite element mesh database. The `var_name` string specifies the name of the variable as it appears on the mesh database.

If the analysis will cause topology modifications due to the use of load balancing, remeshing, or other techniques that can result in nodes and/or elements being moved among processors, then the COPY VARIABLE command should be used. This command specifies that the variable named var_name will be used to specify the rotational velocity. The FROM MODEL <string>model_name portion of the command is optional. If it is specified, the results are read from the mesh database named model_name. Otherwise, the region's finite element mesh database will be used as the model.

The primary difference between the behavior of the COPY VARIABLE command and the READ VARIABLE command is that the copy command handles topology modifications caused by load balancing and remeshing, which can move nodes and elements to different processors. The COPY VARIABLE command will correctly match the nodes from the external database to the corresponding nodes in the analysis and communicate the data between processors if needed. Note that the nodes in the two finite element models must match spatially, but they can be distributed on different processors.

The field to be read may be specified at an arbitrary number of different times on the mesh file. The default behavior is for the time history of the variable read from the file to be used to prescribe the boundary condition. The time history is interpolated as needed for analysis times that do not correspond exactly to times on the mesh file. The TIME command line can optionally be used to specify that the boundary condition be constant over time, based on the value of the variable read from the file at the specified time. 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.

6.3.7.5 Additional Commands

These command lines in the PRESCRIBED ROTATIONAL VELOCITY command block provide additional options for the boundary condition:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
INACTIVE 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 specification commands or the user subroutine. For example, if the magnitude of the rotational 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 rotational velocity from time 1.0 to 2.0 is 0.75. The default value for the scale factor is 1.0.

The ACTIVE PERIODS and INACTIVE PERIODS command lines determine when the boundary condition is active. See Section 2.5 for more information about these command lines.

6.3.8 Reference Axis Rotation

```
BEGIN REFERENCE AXIS ROTATION
  #
  # block command
  BLOCK = <string list>block_names
  #
  # specification commands
  REFERENCE AXIS X FUNCTION = <string>function_name
  REFERENCE AXIS Y FUNCTION = <string>function_name
  REFERENCE AXIS Z FUNCTION = <string>function_name
  #
  # rotation commands
  ROTATION = <string>function_name
  ROTATIONAL VELOCITY = <string>function_name
  #
  # torque command
  TORQUE = <string>function_name
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [REFERENCE AXIS ROTATION]
```

The `REFERENCE AXIS ROTATION` command block is intended to control the rotation of a rigid body. The three `REFERENCE AXIS` line commands are required and together define a time-varying reference axis or vector. The rigid body will rotate to follow the vector. Depending on other line commands in the block, the rigid body will have either zero or one free rotational degree of freedom. If one degree of freedom is present, it is the rotation about the time-varying reference vector.

At most, one of the `ROTATION`, `ROTATIONAL VELOCITY`, or `TORQUE` line commands may be present. If the `ROTATION` or `ROTATIONAL VELOCITY` command line is present, the rigid body will be fully prescribed for rotation. The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines are used to activate or deactivate this kinematic boundary condition for certain time periods.

Use of the `REFERENCE AXIS ROTATION` command block will cause extra global variables to be written to the results file. These variables describe the reaction force, rotational reaction moment, rotational displacement, and rotational velocity associated with the boundary condition and in the local coordinate system described by the boundary condition. The names of the variables are:

```
REACTR_<NAME>
REACTS_<NAME>
REACTT_<NAME>

RREACTR_<NAME>
```

```
    RREACTS_<NAME>
    RREACTT_<NAME>

    ROTDS_<NAME>

    ROTVR_<NAME>
    ROTVS_<NAME>
    ROTVT_<NAME>
```

The variables beginning with `REACT` give the reaction forces in the boundary condition's local coordinate system. Those beginning with `RREACT` give the reaction moments.

Only the `S` component is given for the rotational displacement. This is due to the fact that the other coordinate directions are poorly defined, and therefore, rotational displacements in those directions provide no value.

The variables beginning with `ROTV` give the rotational velocity in the local coordinate system.

For each global variable, `<NAME>` is the name of the rigid body controlled by this boundary condition.

6.3.8.1 Block Command

The `block command` portion of the `REFERENCE AXIS ROTATION` command block defines a block associated with the prescribed rotation field and must include the following command line:

```
BLOCK = <string list>block_names
```

See Section 6.1.1 for more information about the use of this command line.

6.3.8.2 Specification Commands

The three specification commands are required and together define a time-varying vector that gives the orientation of a reference axis for rotation. The magnitude of the vector is ignored.

Following are the command lines used to specify the reference axis rotation with a direction:

```
REFERENCE AXIS X FUNCTION = <string>function_name
REFERENCE AXIS Y FUNCTION = <string>function_name
REFERENCE AXIS Z FUNCTION = <string>function_name
```

At a given time, the function referred to by the `REFERENCE AXIS X FUNCTION` line command gives the x component of a vector defining the reference axis. The pattern holds for the y and z components as well. Since the magnitude of the resulting axis or vector is not used, the three functions do not need to describe a unit vector in time.

6.3.8.3 Rotation Commands

It is possible to prescribe the complete rotation of a rigid body through the use of the `ROTATION` or `ROTATIONAL VELOCITY` line commands. Without these line commands, the rigid body is free to rotate about the reference axis.

Following are the command lines to prescribe the rotation about the reference axis:

```
ROTATION = <string>function_name
ROTATIONAL VELOCITY = <string>function_name
```

The function referred to by the `ROTATION` line command gives the magnitude of rotation in radians about the reference axis as a function of time. This function should not be changed across a restarted analysis.

The function referred to by the `ROTATIONAL VELOCITY` line command gives the rotational velocity in radians per second about the reference axis as a function of time. This function should not be changed across a restarted analysis.

6.3.8.4 Torque Command

A user may specify a moment about the reference axis through the use of the `TORQUE` line command.

```
TORQUE = <string>function_name
```

The function referred to by the `TORQUE` line command gives the torque or moment about the reference axis as a function of time.

6.3.8.5 Additional Commands

The `SCALE FACTOR`, `ACTIVE PERIODS`, and `INACTIVE PERIODS` command lines can optionally appear in the `REFERENCE AXIS ROTATION` command block:

```
SCALE FACTOR = <real>scale_factor(1.0)
ACTIVE PERIODS = <string list>period_names
INACTIVE PERIODS = <string list>period_names
```

The `SCALE FACTOR` command line applies the specified value as a scale factor to the rotation, rotational velocity, or torque functions.

The final two command lines can activate or deactivate the reference axis rotation for certain time periods. See Section 2.5 for more information about these command lines.

6.3.9 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 10 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, Adagio first normalizes the direction vector. Next, Adagio 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 specification commands
  COMPONENT = <string>X|Y|Z |
    DIRECTION = <string>defined_direction
  MAGNITUDE = <real>magnitude_of_velocity
  #
  # angular velocity specification 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 specification, angular velocity specification, or user subroutine.

The `INITIAL VELOCITY` command block contains four groups of commands—node set, direction specification, angular velocity specification, 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 6.1.1 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 Specification Commands

If the direction specification commands are 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 initial velocity can be specified either along an arbitrary user-defined direction or along a component direction (X, Y, or Z), but not both. The velocity is prescribed only in the specified direction. A prescribed velocity boundary condition does not influence the movement in directions orthogonal to the prescribed direction.

- The `DIRECTION` command line is used to prescribe initial velocity in an arbitrary user-defined direction. The name in the string `defined_direction` is a reference to a direction, which is defined using the `DEFINE DIRECTION` command block within the SIERRA scope.
- The `COMPONENT` command line is used to specify that the initial 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 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 specification commands.

6.4.3 Angular Velocity Specification Commands

If the angular velocity specification commands are used, the initial velocity is applied as an initial angular velocity about some axis. Following are the command lines for angular velocity specification:

```
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 SIERRA scope (via a `DEFINE AXIS` command line).

The magnitude of the angular velocity about this axis is specified 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 specification commands.

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 Adagio 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 10.2.2 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 10.

See Section 6.3.9 and Chapter 10 for more details on implementing the user subroutine option.

6.5 Force Boundary Conditions

A variety of force boundary conditions are available in Adagio. 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
  #
  # specification 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
  INACTIVE 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 command 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, user subroutine commands are 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 hexahedra, four-node tetrahedra, eight-node tetrahedra, 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 definition so that the underlying surface geometry is updated as elements are killed. The `SCALE FACTOR` command line can be used in conjunction with either the function command or the user subroutine commands. The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines are used to activate or deactivate this force boundary condition for certain time periods. Following are descriptions of the different command groups.

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 6.1.1 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 Specification Commands

If the function command 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 used to specify the pressure with a function:

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

The pressure is applied 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 SIERRA 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 command, 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 Adagio 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 10.2.2 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 10.

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 10 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
```

Here variable_name is the name of the variable where pressure information is to be stored. The pressure information will be transferred into this 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 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
    INACTIVE PERIODS = <string list>period_names
```

The USE DEATH command line links the pressure boundary condition to an element death definition. The string death_name must match a name used in an ELEMENT DEATH command block. When elements are killed by the named element death definition, 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 command or the user subroutine. 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` and `INACTIVE PERIODS` command lines determine when the boundary condition is active. See Section 2.5 for more information about these command lines.

6.5.2 Traction

```
BEGIN TRACTION
  #
  # surface set commands
  SURFACE = <string list>surface_names
  REMOVE SURFACE = <string list>surface_names
  #
  # specification 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
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE 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 or a user subroutine. If the traction field is constant over the faces and has only a time-varying component, the specification commands 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 is used to specify the traction field.

The traction field can only be controlled by one method. Accordingly, a `TRACTION` command block can only contain one of the options: function or user subroutine.

Currently, the `TRACTION` command block can be used for surfaces that have faces derived from solid elements (eight-node hexahedra, four-node tetrahedra, eight-node tetrahedra, etc.), membranes, and shells.

The `TRACTION` command block contains three groups of commands—surface set and user subroutine. Each of these command groups is basically independent of the others. In addition to the command lines in the four command groups, there are three additional command lines:

SCALE FACTOR, ACTIVE PERIODS, and INACTIVE PERIODS. The SCALE FACTOR command line can be used in conjunction with the specification commands or the user subroutine option. The ACTIVE PERIODS and INACTIVE PERIODS command lines are used to activate or deactivate this force boundary condition for certain time periods. Following are descriptions of the different command groups.

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 6.1.1 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 Specification Commands

If the specification commands are 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 used to specify the traction with a direction and a function:

```
DIRECTION = <string>defined_direction
FUNCTION = <string>function_name
```

The traction is specified in an arbitrary user-defined direction, and is defined using the DIRECTION command line. The name in the string defined_direction is a reference to a direction, which is defined using the DEFINE DIRECTION command block within the SIERRA scope.

The magnitude of the traction is specified by the FUNCTION command line. This references a function_name (defined in the SIERRA 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 Adagio 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 10.2.2 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 10.

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 components 4 through 6 of `output_values` for `inode`. The magnitude of the traction in the specified direction is given by component 1 of `output_values` at `inode`. The total force on each node is found by integrating the local nodal tractions using the associated directions, which are normalized by Adagioover the face areas. The values of the flags array are not used.

See Chapter 10 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
INACTIVE 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 specification commands or the user subroutine. 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` and `INACTIVE PERIODS` command lines determine when the boundary condition is active. See Section 2.5 for more information about these command lines.

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
  #
  # specification 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
  INACTIVE 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 specification commands 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 is used to specify the force field. In a given boundary condition command block, commands from only one of the command groups (specification commands or user subroutine commands) may be used.

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 three additional command lines: `SCALE FACTOR`, `ACTIVE PERIODS`, and `INACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with either the specification commands or the user subroutine commands. The ACTIVE PERIODS and INACTIVE PERIODS command lines are used to activate or deactivate this force boundary condition for certain time periods. Following are descriptions of the different command groups.

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 6.1.1 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 Specification Commands

If the specification commands are 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 used to specify the prescribe force with a direction and a function:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z
FUNCTION = <string>function_name
```

The force can be specified either along an arbitrary user-defined direction or along a component direction (X, Y, or Z), but not both.

- The DIRECTION command line is used to prescribe force in an arbitrary user-defined direction. The name in the string defined_direction is a reference to a direction, which is defined using the DEFINE DIRECTION command block within the SIERRA scope.

- The `COMPONENT` command line is used to specify that the force 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 SIERRA 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 force is applied only in the prescribed direction, and is not applied in any direction orthogonal to that 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 Adagio 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 10.2.2 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 10.

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 10 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
INACTIVE 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 specification commands or the user subroutine. 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` and `INACTIVE PERIODS` command lines determine when the boundary condition is active. See Section 2.5 for more information about these command lines.

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
  #
  # specification 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
  INACTIVE 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 specification commands 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. In a given boundary condition command block, commands from only one of the command groups (specification commands or user subroutine commands) may be used.

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 four command groups, there are three additional command lines: `SCALE FACTOR`, `ACTIVE PERIODS`, and `INACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with either the specification commands or the user subroutine commands. The ACTIVE PERIODS and INACTIVE PERIODS command lines are used to activate or deactivate this force boundary condition for certain time periods. Following are descriptions of the different command groups.

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 6.1.1 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 Specification Commands

If the specification commands are 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 used to specify the prescribed moment with a function and a direction:

```
DIRECTION = <string>defined_direction |
  COMPONENT = <string>X|Y|Z
FUNCTION = <string>function_name
```

The moment can be specified either along an arbitrary user-defined direction or along a component direction (X, Y, or Z), but not both.

- The DIRECTION command line is used to prescribe the moment in an arbitrary user-defined direction. The name in the string defined_direction is a reference to a direction, which is defined using the DEFINE DIRECTION command block within the SIERRA scope.

- 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 SIERRA 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 moment is applied only in the prescribed direction, and is not applied in any direction orthogonal to that 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 do specification commands, 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 Adagio 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 10.2.2 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 10.

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 10 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
INACTIVE 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 specification commands or the user subroutine. 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` and `INACTIVE PERIODS` command lines determine when the boundary condition is active. See Section 2.5 for more information about these command lines.

6.6 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
  INACTIVE PERIODS = <string list>period_names
END [GRAVITY]
```

A gravity load is generally referred to as a body force boundary condition. A gravity load generates a force at a node that is proportional to the mass of the node. This section describes how to apply a gravity load to a body.

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 6.1.1 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 gravity load is specified along an arbitrary user-defined direction, and is defined using the DIRECTION command line. The name in the string defined_direction is a reference to a direction, which is defined using the DEFINE DIRECTION command block within the SIERRA scope.

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 SIERRA 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 and INACTIVE PERIODS command lines provides an additional option for the gravity load condition. These command lines can activate or deactivate the gravity load for certain time periods. See Section 2.5 for more information about these command lines.

6.7 Prescribed Temperature

```
BEGIN PRESCRIBED TEMPERATURE
  #
  # block set commands
  BLOCK = <string list>block_names
  INCLUDE ALL BLOCKS
  REMOVE BLOCK = <string list>block_names
  #
  # specification 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
  # external database commands
  READ VARIABLE = <string>var_name
  COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
  TIME = <real>time
  TEMPERATURE TYPE = SOLID_ELEMENT|SHELL_ELEMENT (SOLID_ELEMENT)
  #
  # coupled analysis commands
  RECEIVE FROM TRANSFER [FIELD TYPE = NODE|ELEMENT(NODE)]
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PRESCRIBED TEMPERATURE]
```

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 command in the above command block can be used to specify the temperature 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 three additional command lines: `SCALE FACTOR`, `ACTIVE PERIODS`, and `INACTIVE PERIODS`. The `SCALE FACTOR` command line can be used in conjunction with the function command, the user subroutine option, or the read variable option. The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines are used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups.

6.7.1 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 6.1.1 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.

6.7.2 Specification Command

If the function command 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 SIERRA scope using 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 6.7.6.

6.7.3 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 command, 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 Adagio 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.

Several other commands control the behavior of user subroutines: `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. These are described in Section 10.2.2. Examples of using these command lines are provided throughout Chapter 10.

The temperature set in the user subroutine can be scaled by use of the `SCALE FACTOR` command line, as described in Section 6.7.6.

See Chapter 10 for more details on implementing the user subroutine option.

6.7.4 External Mesh Database Commands

If the external database option is used, the temperature field is read from an external mesh database. The temperatures are read from a finite element model defined via the `FINITE ELEMENT MODEL` command block described in Section 5.1. The finite element model can either be the model used by the region for its mesh definition as specified with the `USE FINITE ELEMENT MODEL` command (see Section 2.3), or it can be a different (but compatible) model. The following command lines control the use of an external mesh database to prescribe the temperature:

```
READ VARIABLE = <string>mesh_var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
TIME = <real>time
TEMPERATURE TYPE = SOLID_ELEMENT|SHELL_ELEMENT (SOLID_ELEMENT)
```

The `READ VARIABLE` command is used to read the temperature from the region's finite element mesh database. The `var_name` string specifies the name of the variable as it appears on the mesh database.

If the analysis will cause topology modifications due to the use of load balancing, remeshing, or other techniques that can result in nodes and/or elements being moved among processors, then the `COPY VARIABLE` command should be used. This command specifies that the variable named `var_name` will be used to specify the temperature. The `FROM MODEL <string>model_name` portion of the command is optional. If it is specified, the results are read from the mesh database named `model_name`. Otherwise, the region's finite element mesh database will be used as the model.

The primary difference between the behavior of the `COPY VARIABLE` command and the `READ VARIABLE` command is that the copy command handles topology modifications caused by load balancing and remeshing, which can move nodes and elements to different processors. The `COPY VARIABLE` command will correctly match the nodes from the external database to the corresponding nodes in the analysis and communicate the data between processors if needed. Note that the nodes in the two finite element models must match spatially, but they can be distributed on different processors.

The field to be read may be specified at an arbitrary number of different times on the mesh file. The default behavior is for the time history of the variable read from the file to be used to prescribe the boundary condition. The time history is interpolated as needed for analysis times that do not correspond exactly to times on the mesh file. The `TIME` command line can optionally be used to specify that the boundary condition be constant over time, based on the value of the variable read from the file at the specified time. 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.

If temperature is to be prescribed for shell elements, a linear or quadratic thermal gradient can optionally be specified through the thickness of the shells using the `TEMPERATURE TYPE` command line. The `SOLID_ELEMENT` option to this command is the default behavior, and results in a single temperature being read for each node into the `temperature` variable. With the `SHELL_ELEMENT` option, temperatures are read into the `shell_temperature` variable. Shell elements may potentially define a temperature gradient though the thickness, in which case there will be multiple temperatures at a node to describe the temperature gradient through the shell. The `TEMPERATURE_TYPE` command is only valid in conjunction with the `READ VARIABLE` command.

Though-thickness shell temperatures follow the Aria/Calore convention. If there is a single shell temperature defined at a node, the temperature is constant through the thickness.

If there are two shell temperatures defined at a node, the first is the temperature on the bottom of the shell and the second the temperature at the top. The temperature varies linearly between the top and bottom.

If there are three shell temperatures defined at node, the first is the temperature at the bottom of the shell, the second the temperature at the middle of the shell, and the third the temperature at the top of the shell. The temperature varies quadratically through the thickness.

SOLID_ELEMENT and SHELL_ELEMENT temperatures may be defined simultaneously in the same analysis through two different temperature command blocks. If both are defined, the shell element temperature results override any solid element temperature results on the shell elements.

6.7.5 Coupled Analysis Commands

The RECEIVE FROM TRANSFER command provides the ability to set the temperature in a coupled analysis by transferring results from another SIERRA code, such as Aria.

```
RECEIVE FROM TRANSFER [FIELD TYPE = NODE|ELEMENT(NODE)]
```

If this command is used in its default form, it is expected that the temperature is transferred to a nodal field named temperature in the Adagio region. Adagio performs an interpolation from the nodal field to an element field named temperature. The temperature can also be transferred directly to the element temperature field by using the optional FIELD TYPE = ELEMENT argument to this command.

If the RECEIVE FROM TRANSFER command is used, but the appropriate commands to perform the transfer between the two regions are missing, the temperature will be zero during the entire simulation.

It is also possible to use this line command to cause the initial temperatures obtained from a restart file to remain constant in time.

6.7.6 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
INACTIVE 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 command, the user subroutine, 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 and INACTIVE PERIODS command lines determine when the boundary condition is active. See Section 2.5 for more information about these command lines.

6.8 Pore Pressure

```
     BEGIN PORE PRESSURE
       #
       # block set commands
       BLOCK = <string list>block_names
       INCLUDE ALL BLOCKS
       REMOVE BLOCK = <string list>block_names
       #
       # specification command
       FUNCTION = <string>function_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
       # external database commands
       READ VARIABLE = <string>var_name
       COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
       TIME = <real>time
       #
       # coupled analysis commands
       RECEIVE FROM TRANSFER [FIELD TYPE = NODE|ELEMENT(NODE)]
       #
       # additional commands
       SCALE FACTOR = <real>scale_factor(1.0)
       ACTIVE PERIODS = <string list>period_names
       INACTIVE PERIODS = <string list>period_names
    END [PORE PRESSURE]
```

The `PORE PRESSURE` command block prescribes a pore pressure field for a given set of elements. The pore pressure is prescribed for each element in the block. The pore pressure field may vary over time and space. If the pore pressure field has only a time-varying component, the function command in the above command block can be used to specify the pore pressure field. If the pore pressure field has both time-varying and spatially varying components, a user subroutine option can be used to specify the pore pressure field. Finally, the pore pressure can be read 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.

The `PORE PRESSURE` 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 three additional command lines: `SCALE FACTOR`, `ACTIVE PERIODS`, and `INACTIVE PERIODS`. The

SCALE FACTOR command line can be used in conjunction with the function command, the user subroutine option, or the read variable option. The ACTIVE PERIODS and INACTIVE PERIODS command lines are used to activate or deactivate this kinematic boundary condition for certain time periods. Following are descriptions of the different command groups.

Biot's coefficient can be defined when prescribing pore pressure. See Section 4.1.2 for more information on Biot's coefficient.

6.8.1 Block Set Commands

The block set commands portion of the PORE PRESSURE command block defines a set of elements associated with the pore pressure field and can include a 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 elements derived from some combination of element blocks. See Section 6.1.1 for more information about the use of these command lines for creating a set of elements used by the boundary condition. There must be at least one BLOCK or INCLUDE ALL BLOCKS command line in the command block.

6.8.2 Specification Command

If the FUNCTION command is used, the pore pressure at any given time is the same for all elements in the element set associated with the particular PORE PRESSURE command block. The command line

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

references a function_name (defined in the SIERRA scope using a DEFINITION FOR FUNCTION command block) that specifies the pore pressure as a function of time. The pore pressure can be scaled using the SCALE FACTOR command line described in Section 6.8.6.

6.8.3 User Subroutine Commands

If the user subroutine option is used, the pore pressure field may vary spatially at any given time for each of the elements in the element set associated with the particular PORE PRESSURE command block. The user subroutine option allows for a more complex description of the pore pressure field than does the FUNCTION command, but the user subroutine option also requires that a user subroutine be written to implement this capability. The user subroutine will be used to define a pore pressure for every element to which the boundary condition will be applied. The subroutine will be called by Adagio at the appropriate time to generate the pore pressure field.

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.

Several other commands control the behavior of user subroutines: `SUBROUTINE DEBUGGING OFF`, `SUBROUTINE DEBUGGING ON`, `SUBROUTINE REAL PARAMETER`, `SUBROUTINE INTEGER PARAMETER`, and `SUBROUTINE STRING PARAMETER`. These are described in Section 10.2.2. Examples of using these command lines are provided throughout Chapter 10.

The pore pressure set in the user subroutine can be scaled by use of the `SCALE FACTOR` command line, as described in Section 6.8.6.

See Chapter 10 for more details on implementing the user subroutine option.

6.8.4 External Mesh Database Commands

The pore pressure field can be read from an external mesh database. The finite element model from which pore pressures are read is defined via the `FINITE ELEMENT MODEL` command block described in Section 5.1. The finite element model can either be the model used by the region for its mesh definition as specified with the `USE FINITE ELEMENT MODEL` command (see Section 2.3), or it can be a different (but compatible) model. The following command lines control the use of an external mesh database to prescribe the pore pressure:

```
READ VARIABLE = <string>mesh_var_name
COPY VARIABLE = <string>var_name [FROM MODEL <string>model_name]
TIME = <real>time
```

The `READ VARIABLE` command is used to read the pore pressure from the region's finite element mesh database. The `var_name` string specifies the name of the variable as it appears on the mesh database.

If the analysis will cause topology modifications due to the use of load balancing, remeshing, or other techniques that can result in nodes and/or elements being moved among processors, the

COPY VARIABLE command should be used. This command specifies that the variable named var_name will be used to specify the pore pressure. The FROM MODEL <string>model_name portion of the command is optional. If it is specified, the results are read from the mesh database named model_name. Otherwise, the region's finite element mesh database will be used as the model.

The primary difference between the behavior of the COPY VARIABLE command and the READ VARIABLE command is that the copy command handles topology modifications caused by load balancing and remeshing, which can move nodes and elements to different processors. The COPY VARIABLE command will correctly match the nodes from the external database to the corresponding nodes in the analysis and communicate the data between processors if needed. Note that the nodes in the two finite element models must match spatially, but they can be distributed on different processors.

The field to be read may be specified at an arbitrary number of different times on the mesh file. The default behavior is for the time history of the variable read from the file to be used to prescribe the boundary condition. The time history is interpolated as needed for analysis times that do not correspond exactly to times on the mesh file. The TIME command line can optionally be used to specify that the boundary condition be constant over time, based on the value of the variable read from the file at the specified time. 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.

6.8.5 Coupled Analysis Commands

The RECEIVE FROM TRANSFER command provides the ability to set the pore pressure in a coupled analysis by transferring results from another SIERRA code, such as Aria.

```
RECEIVE FROM TRANSFER [FIELD TYPE = NODE|ELEMENT(NODE)]
```

If this command is used in its default form, it is expected that the pore pressure is transferred to a nodal field named pore_pressure in the Adagio region. Adagio performs an interpolation from the nodal field to an element field named pore_pressure. The pore pressure can also be transferred directly to the element pore_pressure field by using the optional FIELD TYPE = ELEMENT argument to this command.

If the RECEIVE FROM TRANSFER command is used, but the appropriate commands to perform the transfer between the two regions are missing, the pore pressure will be zero during the entire simulation.

6.8.6 Additional Commands

These command lines in the PORE PRESSURE command block provide additional options for the boundary condition:

```
    SCALE FACTOR = <real>scale_factor(1.0)
    ACTIVE PERIODS = <string list>period_names
    INACTIVE 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 pore pressure values of the field defined by the function command, the user subroutine, or the read variable option. For example, if the pore pressure 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 pore pressure from time 1.0 to 2.0 is 50.25. The default value for the scale factor is 1.0.

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines determine when the boundary condition is active. See Section 2.5 for more information about these command lines.

6.9 Fluid Pressure

```
BEGIN FLUID PRESSURE
  #
  # surface set commands
  SURFACE = <string list>surface_names
  #
  # specification commands
  DENSITY = <real>fluid_density
  DENSITY FUNCTION = <string>density_function_name
  GRAVITATIONAL CONSTANT = <real>gravitational_acceleration
  FLUID SURFACE NORMAL = <string>global_component_names
  DEPTH = <real>fluid_depth
  DEPTH FUNCTION = <string>depth_function_name
  REFERENCE POINT = <string>reference_point_name
  #
  # additional commands
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PRESSURE]
```

The `FLUID PRESSURE` command block applies a hydrostatic pressure to each node of each face in the associated surfaces. The pressure at any node is determined from

$$P = \rho g h \tag{6.2}$$

where P is the pressure, ρ is the fluid density at the current time, g is the gravitational constant, and h is the current depth of the fluid above the node. The depth of the fluid is computed as the distance from the current fluid surface to the node in the direction of the fluid surface normal. The normal must be specified as one of the three global coordinate directions, x, y, or z. The global location of the fluid surface is found by adding the current depth to the appropriate coordinate component (the direction defined in the `FLUID SURFACE NORMAL` command) of a datum point. The datum point is either the point specified in the `REFERENCE POINT` line command, or, in the absence of this command, the minimum coordinate on the applied pressure surface in the component direction defined in the `FLUID SURFACE NORMAL` command. Once the current location of the fluid surface is computed, the depth at each node on the pressure surface is computed as the distance from the node to the fluid surface in the direction of the fluid surface normal.

Currently, the `FLUID PRESSURE` command block can be used for surfaces that have faces derived from solid elements (eight-node hexahedra, four-node tetrahedra, eight-node tetrahedra, etc.), membranes, and shells.

The `FLUID PRESSURE` command block contains three groups of commands—surface set, specification, and additional optional commands.

6.9.1 Surface Set Commands

The `surface set commands` portion of the `FLUID PRESSURE` command block defines a set of surfaces associated with the pressure field and consists of the following line:

```
SURFACE = <string list>surface_names
```

In the `SURFACE` command line, a series of surfaces can be listed through the string list `surface_names`. There must be at least one `SURFACE` command line in the command block. See Section 6.1.1 for more information about the use of command lines for creating a set of surfaces used by the boundary condition. The force computed from the hydrostatic pressure will be in the opposite direction of the face normal. When using shells or membranes, the analyst must ensure that all face normals composing the pressure application surface are in the correct direction.

6.9.2 Specification Commands

The density and the gravitational acceleration must be input by the user in units consistent with other material properties and the lengths in the mesh. To facilitate convergence of the initial load step, the gradual application of a hydrostatic load may be specified through a time history function for the density. A combination of the `DENSITY` and the `DENSITY FUNCTION` sets the value for the fluid density at each time. Either the `DENSITY` or the `DENSITY FUNCTION` must be input and both can be used together. If the `DENSITY` command is input without the `DENSITY FUNCTION` command, the density will be constant in time at that value. If the `DENSITY FUNCTION` command is input without a `DENSITY` command, the function is used as a time history of the density. If both the `DENSITY` and the `DENSITY FUNCTION` commands are input, the density value is used as a scale factor on the time history function. Finally, the `GRAVITATIONAL CONSTANT` sets the value for the acceleration due to gravity, g.

```
DENSITY = <real>fluid_density
DENSITY FUNCTION = <string>fluid_density_function
GRAVITATIONAL CONSTANT = <real>G
```

The following command lines are used to define the location of the fluid surface at any time during the analysis:

```
FLUID SURFACE NORMAL = <string>normal_component
DEPTH = <real>initial_fluid_depth
DEPTH FUNCTION = <string>depth_function_name
REFERENCE POINT = <string>point_name
```

The `FLUID SURFACE NORMAL` command sets the outward normal of the fluid surface to be one of the global component directions, x, y, or z. The fluid depth is then assumed to be in the direction opposite this global direction. The `DEPTH` command is used with the `DEPTH FUNCTION` command to determine the fluid depth at any time. At least one of these commands must be input. If the DEPTH command is input without the DEPTH FUNCTION command, the fluid depth will be constant in time with that value. If the DEPTH FUNCTION command is input without a DEPTH command, the function is used as a time history of the depth. If both the DEPTH and the DEPTH FUNCTION commands are input, the specified depth is used as a scale factor on the time history function.

The depth and/or depth function are used to determine the current depth, which is added to the appropriate position of a datum point to compute the current location of the fluid surface in the FLUID SURFACE NORMAL component direction. The datum point is assumed to be the minimum coordinate in the component direction on the pressure surface defined in the SURFACE command if the optional REFERENCE POINT command described below is not used.

The REFERENCE POINT command line is used to specify the fluid surface relative to an external datum. When applying an external fluid pressure in a quasistatic analysis, a corresponding stiffness due to the external fluid is added to the diagonal terms of the stiffness matrix for the full tangent preconditioner to enhance convergence of the solver.

6.9.3 Additional Commands

These command lines in the FLUID PRESSURE command block provide additional options for the boundary condition:

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

By default, the FLUID PRESSURE boundary condition will be active throughout an analysis. However, use of the ACTIVE PERIODS and INACTIVE PERIODS commands can be used to limit the action of the boundary condition to specific time periods. The ACTIVE PERIODS command line specifies when the boundary condition is active implying that it is inactive during any periods not included on the command line. Alternatively, the INACTIVE PERIODS determines when the boundary condition will not be active. See Section 2.5 for more information about these commands.

6.10 Specialized Boundary Conditions

Specialized boundary conditions that are provided to enforce kinematic conditions or apply loads are described in this section.

6.10.1 Blast Pressure

```
BEGIN BLAST PRESSURE
  SURFACE = <string list>surface_ids
  REMOVE SURFACE = <string list>surface_ids
  BURST TYPE = <string>SURFACE|AIR
  TNT MASS IN LBS = <real>tnt_mass_lbs
  BLAST TIME = <real>blast_time
  BLAST LOCATION = <real>loc_x <real>loc_y <real>loc_z
  ATMOSPHERIC PRESSURE IN PSI = <real>atmospheric_press
  AMBIENT TEMPERATURE IN FAHRENHEIT = <real>temperature
  FEET PER MODEL UNITS = <real>feet
  MILLISECONDS PER MODEL UNITS = <real>milliseconds
  PSI PER MODEL UNITS = <real>psi
  PRESSURE SCALE FACTOR = <real>pressure_scale(1.0)
  IMPULSE SCALE FACTOR = <real>impulse_scale(1.0)
  POSITIVE DURATION SCALE FACTOR = <real>duration_scale(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [BLAST PRESSURE]
```

The `BLAST PRESSURE` command block is used to apply a pressure load resulting from a conventional explosive blast. This boundary condition is based on Reference 2 and Reference 3, and Sachs scaling is implemented to match the ConWep code (Reference 4). Angle of incidence is accounted for by transitioning from reflected pressure to incident pressure according to:

$$P_{total} = P_{ref} * \cos\theta + P_{inc} * (1 - \cos\theta) \tag{6.3}$$

where θ is the angle between the face normal vector and the direction to the blast from the face, P_{total} is the total pressure, P_{ref} is the reflected portion of the pressure, and P_{inc} is the incident portion of the pressure. P_{ref} and P_{inc} are based on Friedlander's equation, as described in Reference 3.

If θ is greater than 90 degrees (i.e. the face is pointing away from the blast), only P_{inc} is applied to the face. In this case, the face variable `cosa`, which contains $\cos\theta$, is set to zero.

This boundary condition is applied to the surfaces 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.)

Warning: Multiple `BLAST PRESSURE` command blocks may be used in an analysis to apply blast loads at different locations. However, only one should be applied to a given element face. Each instance of this boundary condition should be applied to a different set of surfaces, and those surfaces should not overlap surfaces used by other instances of this boundary condition. This is because face variables are used to store information used by this boundary condition, and those variables would be over-written by another instance of the boundary condition.

Table 8.9 lists the face variables used by the `BLAST PRESSURE` boundary condition. These can be requested for output in the standard manner (see Chapter 8), and can be useful for verifying that this boundary condition is correctly applied.

The type of burst load is specified with the `BURST TYPE` command, which can be `SURFACE` or `AIR`. The `SURFACE` option is used to define a hemispherical burst, while the `AIR` option is used for a spherical burst.

The equivalent TNT mass (in pounds) is defined with the `TNT MASS IN LBS` command. The time of the explosion is defined using the `BLAST TIME` command. This can be negative, and can be used to start the analysis at the time when the blast reaches the structure, saving computational time. The location of the blast is defined with the `BLAST LOCATION` command.

The current ambient pressure and temperature are defined using the `ATMOSPHERIC PRESSURE IN PSI` and `AMBIENT TEMPERATURE IN FAHRENHEIT` commands, respectively. As implied by the command names, these must be supplied in units of pounds per square inch and degrees Fahrenheit.

Because of the empirical nature of this method for computing an explosive load, appropriate conversion factors for the unit system used in the model must be supplied. The commands `FEET PER MODEL UNITS`, `MILLISECONDS PER MODEL UNITS`, and `PSI PER MODEL UNITS` are used to specify the magnitude of one foot, one millisecond, and one pound per square inch in the unit system of the model.

All of the commands listed above are required. Scaling factors can optionally be applied to modify the peak pressure, the impulse, and the duration of the loading. The `PRESSURE SCALE FACTOR` command scales the the peak value of both the reflected and incident portions of the applied pressure. The `IMPULSE SCALE FACTOR` command scales the impulse of the reflected and incident portions of the applied pressure. The `POSITIVE DURATION SCALE FACTOR` command scales the duration of the reflected and incident portions of the applied pressure. Each of these scaling factors only affects the quantity that it modifies, for example, scaling the pressure does not affect the impulse or duration.

The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines can optionally be used to activate or deactivate this boundary condition for certain time periods. See Section 2.5 for more information about these command lines.

6.10.2 General Multi-Point Constraints

```
BEGIN MPC
  #
  # Master/Slave MPC commands
  MASTER NODE SET = <string list>master_nset
  MASTER NODES = <integer list>master_nodes
  MASTER SURFACE = <string list>master_surf
  MASTER BLOCK = <string list>master_block
  SLAVE NODE SET = <string list>slave_nset
  SLAVE NODES = <integer list>slave_nodes
  SLAVE SURFACE = <string list>slave_surf
  SLAVE BLOCK = <string list>slave_block
  #
  # Tied contact search commands
  SEARCH TOLERANCE = <real>tolerance
  VOLUMETRIC SEARCH TOLERANCE = <real>vtolerance
  #
  # Tied MPC commands
  TIED NODES = <integer list>tied_nodes
  TIED NODE SET = <string list>tied_nset
  #
  # DOF subset selection
  COMPONENTS = <enum>X|Y|Z|RX|RY|RZ
END [MPC]

# Control handling of multiple MPCs
RESOLVE MULTIPLE MPCS = ERROR|FIRST WINS|LAST WINS(ERROR)
```

Adagio provides a general multi-point constraint (MPC) capability that allows a code user to specify arbitrary constraints between sets of nodes. The commands to define a MPC are all listed within a `MPC` command block. There are three types of MPCs: master/slave, tied contact, and tied. All of these types of MPCs are defined within the `MPC` command block, but different commands are used within that block for each case. The commands for each of these types of MPCs are described in detail below.

MPCs can potentially be in conflict with other constraints. Refer to Appendix D for information on how conflicting constraints are handled.

6.10.2.1 Master/Slave Multi-Point Constraints

The master/slave type of MPC imposes a constraint between a set of master nodes and a set of slave nodes. The motion of the three translational degrees of freedom of the slave nodes is constrained to be equal to the average motion of the master nodes. This type of MPC is typically most useful if there is either a single master node and one or more slave nodes, or multiple master nodes and a single slave node. If there are multiple slave nodes, they are constrained to move together as a set.

The sets of master and slave nodes used in the MPC can be defined by using a node set on the mesh file, a list of nodes provided in the input file, or a surface on the mesh file from which a list of nodes is extracted. This can be done for the set of master nodes using one or more of the following commands:

```
MASTER NODE SET = <string list>master_nset
MASTER NODES = <integer list>master_nodes
MASTER SURFACE = <string list>master_surf
MASTER BLOCK = <string list>master_block
SLAVE NODE SET = <string list>slave_nset
SLAVE NODES = <integer list>slave_nodes
SLAVE SURFACE = <string list>slave_surf
SLAVE BLOCK = <string list>slave_block
```

The `MASTER NODE SET` and `SLAVE NODE SET` command lines specify the names of node sets in the mesh file. The nodes in these node sets are included in the sets of master or slave nodes for the constraint.

The `MASTER NODES` and `SLAVE NODES` command lines specify lists of integer IDs of nodes to be included in the sets of master or slave nodes for the constraint.

The `MASTER SURFACE` and `SLAVE SURFACE` command lines specify the name of a surface in the mesh file. The nodes contained in this surface are included in the sets of master or slave nodes for the constraint.

The `MASTER BLOCK` and `SLAVE BLOCK` command lines specify the names of a block in the mesh file. The nodes contained in these blocks are included in the sets of master or slave nodes for the constraint.

6.10.2.2 Tied Contact

A proximity search can optionally be performed to create a set of MPCs that act as tied contact constraints. If the MPCs are created in this way, the search is performed at the time of initialization to find pairings of slave nodes to master faces. A separate constraint is created for each slave node. This is equivalent to using pure master/slave tied contact.

```
SEARCH TOLERANCE = <real> tolerance
```

The `SEARCH TOLERANCE` command line is used to request that a search be performed to create node/face constraints. This line must be present to use MPCs for tied contact. The tolerance value given on the line specifies the maximum distance between a node and a face to create an MPC. This has a similar meaning to the search tolerance used in standard tied contact.

Warning: The `SEARCH TOLERANCE` command line must be present to use MPCs for tied contact. If this command is not present in the `MPC` command block, a master/slave MPC as described in Section 6.10.2.1 will result. All slave nodes would be tied to all master nodes, which is very different from tied contact.

```
VOLUMETRIC SEARCH TOLERANCE = <real> vtolerance
```

In addition to creating node/face constraints, a search can be used to create volumetric constraints, in which a slave node is constrained to a volume bounded by master faces. The `VOLUMETRIC SEARCH TOLERANCE` command is used to enable volumetric constraints and sets the tolerance used for the search. The slave node is constrained to all nodes on the master surface that are within the volumetric search tolerance, `vtolerance`.

Currently the primary usage of volumetric constraints is to constrain a meshed void placed inside another mesh. The volumetric search is used in conjunction with the node/face search. A set of node/face constraints is first created for slave nodes within the standard search tolerance of any master face. Next, a volumetric constraint is created for any remaining unconstrained slave node within the volumetric search tolerance of the node/face constrained nodes. The volumetric constraint is formulated in a way that approximates an isoparametric map.

To use MPCs for tied contact, the master and slave surfaces must be defined. These may be defined using the `MASTER SURFACE`, `MASTER BLOCK`, `SLAVE NODE SET`, `SLAVE SURFACE`, and `SLAVE BLOCK` line commands. These are a subset of the commands available to define master/slave MPCs, as described in Section 6.10.2.1. It is important to note that the `MASTER NODE SET` can not be used to use MPCs for tied contact. The master surface must have information about faces, and this is not available with a node set.

The following example demonstrates how to use MPCs for tied contact between two surfaces:

```
BEGIN MPC
  MASTER SURFACE = surface_10
  SLAVE SURFACE = surface_11
  SEARCH TOLERANCE = 0.0001
END MPC
```

The following example demonstrates the usage of MPCs for both tied and volumetric constraints. Assuming that `surface_10` surrounds the exterior surface of `block_1`, this block would result in tied contact between the nodes on the exterior of `block_1` and `surface_10`, and volumetric constraints between the nodes on the interior of `block_1` and `surface_1`.

```
BEGIN MPC
  MASTER SURFACE = surface_10
  SLAVE BLOCK = block_1
  SEARCH TOLERANCE = 0.0001
  VOLUMETRIC SEARCH TOLERANCE = 3.0
END MPC
```

6.10.2.3 Tied Multi-Point Constraints

The tied type of MPC imposes a constraint that ties together the motion of the three translational degrees of freedom for a set of nodes. Nodes are not specified as being masters or slaves for this type of constraint. The set of nodes to be tied together can be specified as either a list of node IDs or with a node set by using the `TIED NODES` or `TIED NODE SET` command.

```
TIED NODES = <integer list>tied_nodes
TIED NODE SET = <string list>tied_nset
```

The `TIED NODES` command line is used to specify an integer list of IDs of the nodes to be tied together. The `TIED NODE SET` can be used to specify the name of a node set that contains the nodes to be tied together. Only one of these commands can be used in a given `MPC` command block.

Warning: The tied MPC described here does not do a contact search. For the MPC to behave like tied contact, use the commands described in Section 6.10.2.2.

6.10.2.4 Resolve Multiple MPCs

The behavior of multi-point constraints is ill-defined when a master node is constrained to more than one set of slave nodes. Adagio's MPC capability can handle chained MPCs, where a master node is a slave node in another constraint, but it cannot simultaneously enforce multiple MPCs that have the same master.

```
RESOLVE MULTIPLE MPCS = ERROR|FIRST WINS|LAST WINS(ERROR)
```

The `RESOLVE MULTIPLE MPCS` command line, used within the region scope, controls how to resolve cases where a slave node is constrained to more than one set of master nodes. Although multiple MPCs cannot be simultaneously enforced, this command provides ways to work around this problem that may be acceptable in many situations. The default option is `ERROR`, which results in an error message and terminates the code if this occurs. Alternatively, this command can be set to `FIRST WINS` to keep the first MPC found for a given slave node or `LAST WINS` to keep the last MPC found. This command line controls the behavior of all MPCs in the model.

6.10.2.5 Constraining a Subset of all DOFs

By default, multi-point constraints are applied to all degrees of freedom of the nodes involved. For nodes that only have translational degrees of freedom, all three components (X, Y and Z) are constrained. Likewise, for nodes that have both translational and rotational degrees of freedom, all six components (X, Y, Z, RX, RY and RZ) are constrained. The `COMPONENTS` command line can be used to enforce the constraint on a subset of the degrees of freedom. If the `COMPONENTS` command line is included in a `MPC` command block, only the components listed would be constrained.

6.10.3 Submodel

```
BEGIN SUBMODEL
  #
  EMBEDDED BLOCKS = <string list>embedded_block
  ENCLOSING BLOCKS = <string list>enclosing_block
END [SUBMODEL]
```

Adagio provides a method to embed a submodel in a larger finite element model. The element blocks for both the submodel and the larger system model should exist in the same mesh file. The space occupied by the embedded blocks should also be occupied by the enclosing blocks.

This capability ties each node of the submodel to an element in the larger finite element model. The code makes no correction for mass due to volume overlap. However, this correction in many cases can be done easily by hand simply by adjusting the density of the submodel block so that it is the difference between the density of the submodel block and the enclosing block.

The embedded blocks (the submodel blocks) and the enclosing blocks (the system model blocks) are specified using the following two line commands:

```
  EMBEDDED BLOCKS = <string list>embedded_block
  ENCLOSING BLOCKS = <string list>enclosing_block
```

For example, to embed `block_7` and `block_8` inside a system model where the embedded blocks are within `block_2`, `block_3`, and `block_5`, the following can be used:

```
BEGIN SUBMODEL
  EMBEDDED BLOCKS = block_7 block_8
  ENCLOSING BLOCKS = block2 block_3 block_5
END
```

6.11 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*. SAND2003-1048. Albuquerque, NM: Sandia National Laboratories, April 2003. pdf.

2. Kingery, C. N. and Bulmash, G. *Airblast Parameters from TNT Spherical Air Burst and Hemispherical Surface Burst,* Technical Report ARBBRL-TR-02555, Aberdeen Proving Ground, MD: Ballistic Research Laboratory, April 1984.

3. Randers-Pehrson, G. and Bannister, K. A. *Airblast Loading Model for DYNA2D and DYNA3D,* ARL-TR-1310, Army Research Laboratory, March 1997.

4. Protective Design Center, United States Army Corps of Engineers, *ConWep 2.1.0.8.* link.

5. 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.

6. 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.

Chapter 7

Contact

This chapter describes the input syntax for defining interactions of contact surfaces in a Adagio analysis. For more information on contact and its computational details, consult References 1 and 2.

Contact constraints can potentially be in conflict with other constraints. Refer to Appendix D for information on how conflicting constraints are handled.

Contact refers to the interaction of 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 Adagio are designed to ensure that surfaces do not interpenetrate in a nonphysical way, and that the interface behavior is computed correctly (e.g., energy dissipation from a friction model). Adagio uses a kinematic approach rather than a penalty approach to eliminate the interpenetration of surfaces. In the kinematic approach, a series of constraint equations that remove interpenetration are satisfied. A penalty approach can be thought of as introducing "stiff" springs between contact surfaces as a means of preventing interpenetration.

Contact between surfaces is enforced as node-face interactions. Consider the two-dimensional contact problem shown in Figure 7.1. Block a is enclosed by surface a, and block b is enclosed by surface b. A surface is defined by a collection of finite element faces. The surface of a block of hexahedral elements, for example, is a collection of quadrilateral faces on the surface of the block. For this two-dimensional drawing, the faces are straight lines between two nodes. Only the faces on the portions of the surfaces that will come into contact are shown.

Figure 7.1 shows the two blocks at time step n. Figure 7.2 shows the blocks at time step $n + 1$ when the blocks have moved and deformed under the influence of external forces. Before contact is taken into account, the two blocks interpenetrate one another. This interpenetration is removed by applying the contact algorithm.

For interpenetration to occur as shown in Figure 7.2, any node on surface a that penetrates surface b must pass through some face on surface b. Likewise, each node on surface b that penetrates surface a must pass through some face on surface a. All the nodes on surface a could be forced to lie on surface b, where surface b has the configuration shown in Figure 7.2. In this case, surface b would be a master surface and surface a would be a slave surface. Or all nodes on surface b could

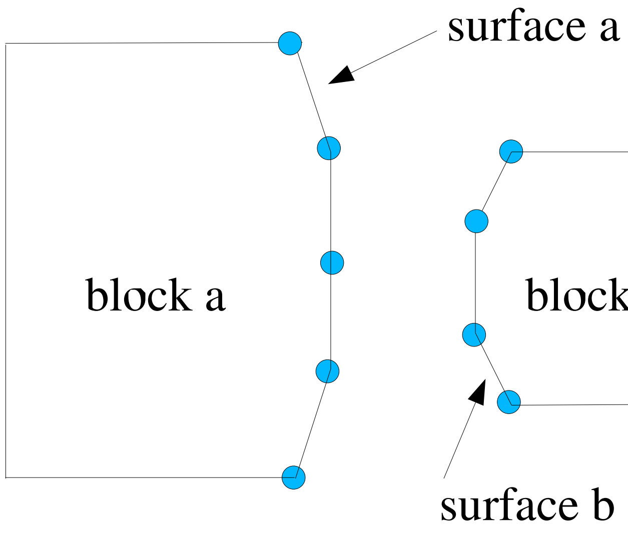


Figure 7.1: Two blocks at time step *n* before contact.

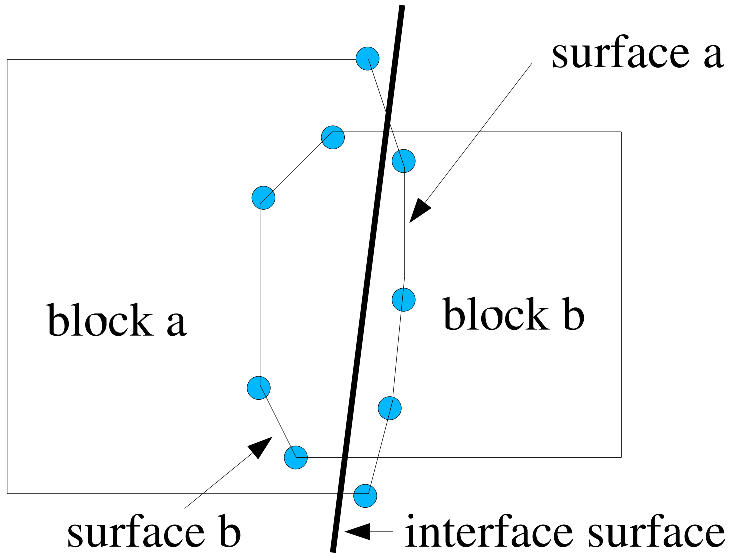


Figure 7.2: Two blocks at time step *n* + 1, after penetration.

be forced to lie on surface *a*, in which case surface *a* is a master surface and surface *b* is the slave. In Adagio one surface of an interaction must be designated as the master and the other as the slave.

The preceding two-dimensional example is analogous to much of the contact that is encountered when contact 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 these surfaces are used to move nodes to account for any interpenetration of the surfaces. Adagio will also handle variations of the node-face contact described as follows:

- 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.

- 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 cannot penetrate the surface.

There are some special considerations for contact with structural elements (i.e. shells) 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.

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 5.2.4.

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 in Adagio 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 Adagio focuses on large-scale global contact in a massively parallel environment. 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 as opposed to a penalty approach. A kinematic approach is more accurate than the penalty approach.

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 Adagio 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 Adagio default parameters for a problem.
2. Accept the Adagio default parameters for some of the contact surfaces. For the rest of the contact surfaces, the user can set values in interactions.
3. Define all surface-pair interactions separately.

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.
- Describe friction models used in the surface interactions for this analysis. Currently, there are 4 types of friction models.
- 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 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
  #
  ENFORCEMENT = <string>TIED|FRICTIONLESS|FRICTIONAL
  #
  CONTACT SURFACE <string>name
    CONTAINS <string list>surface_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
  #
  BEGIN SURFACE NORMAL SMOOTHING
    ANGLE = <real>angle_in_deg
    DISTANCE = <real>distance
    RESOLUTION = <string>NODE|EDGE
  END [SURFACE NORMAL SMOOTHING]
  #
  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 GLUED MODEL <string>name
  END [GLUED MODEL <string>name]
  #
  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
    CAPTURE TOLERANCE = <real>cap_tol
    TENSION RELEASE = <real>ten_release
    SLIP PENALTY = <real>slip_pen
    FACE MULTIPLIER = <real>face_multiplier(0.1)
    SECONDARY DECOMPOSITION = <string>ON|OFF(OFF)
  END [SEARCH OPTIONS <string>name]
  #
  BEGIN INTERACTION DEFAULTS [<string>name]
    CONTACT SURFACES = <string list>surface_names
    GENERAL CONTACT = <string>ON|OFF(OFF)
    FRICTION MODEL = <string>friction_model_name|
      FRICTIONLESS(FRICTIONLESS)
  END [INTERACTION DEFAULTS <string>name]
  #
  BEGIN INTERACTION [<string>name]
    MASTER = <string>surface
    SLAVE = <string>surface
    CAPTURE TOLERANCE = <real>cap_tol
    NORMAL TOLERANCE = <real>norm_tol
    TANGENTIAL TOLERANCE = <real>tang_tol
    FRICTION MODEL = <string>friction_model_name|
      FRICTIONLESS(FRICTIONLESS)
    PUSHBACK FACTOR = <real>pushback_factor(1.0)
    TENSION RELEASE = <real>ten_release
    TENSION RELEASE FUNCTION = <string>ten_release_func
    FRICTION COEFFICIENT = <real>coeff
    FRICTION COEFFICIENT FUNCTION = <string>coeff_func
  END [INTERACTION <string>name]
  #
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 7.1through Section 7.9. Section 7.11 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 Enforcement

```
    ENFORCEMENT = <string>TIED|FRICTIONLESS|FRICTIONAL
```

The `ENFORCEMENT` command line indicates which of three types of contact enforcement available in Adagio should be used in this `CONTACT DEFINITION` block. The first of these, `TIED`, will match slave nodes to master surfaces during initialization. Thereafter, the slave nodes will not be allowed to move relative to their master surfaces.

The `FRICTIONLESS` option allows slave nodes to slide along master surfaces but prevents penetration.

The `FRICTIONAL` option also allows slave nodes to slide along master surfaces. However, the sliding motion is affected by a frictional law. No penetration is allowed.

7.3 Descriptions of Contact Surfaces

Contact determines whether two surfaces, each defined by either an analytic representation or a collection of finite element faces, have interpenetrated. 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.

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. On the other hand, a shell element has both a top face and a bottom face. 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.

The contact enforcement algorithm only allows for a face to be associated with a single contact surface. If a face were allowed to belong to more than one surface involved in contact, ambiguities would arise in determining the interaction properties for that face. This situation could occur if multiple sidesets used in contact overlap. It could also occur if a sideset used in contact included the surface of a skinned block.

If a face is part of more than one contact surface, the face is included in the first contact surface to which it belongs that is listed in the `CONTACT DEFINITION` block. If it belongs to any other contact surface listed later in that block, it is excluded from that surface for the purpose of contact enforcement, and a warning message is generated.

To enforce contact between unassociated nodes and a surface, a contact node set must be defined. The contact of one-dimensional elements (springs, trusses, beams) with a surface can be modeled as unassociated nodes contacting a surface, although, as in the case of shells, there are some limitations.

Surfaces involved in contact can be defined using the `CONTACT SURFACE` command line or the `CONTACT SURFACE` command block. Unassociated nodes involved in contact can be defined using the `CONTACT NODE SET` command line or the `CONTACT SURFACE` command block. The `CONTACT DEFINITION` command block can contain any combination of these command lines and command blocks provided that contact surface names are not duplicated. The `CONTACT DEFINITION` command block must include some type of surface definition. Any element faces or unassociated nodes for use in a contact interaction must be identified as contact surfaces or contact node sets, respectively.

Section 7.3.1 through Section 7.3.3 describe the command lines and command blocks for defining contact surfaces composed of finite element faces and node sets that can contact surfaces.

7.3.1 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 as defined in the exodus file. These are not names of analytic surfaces. 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 7.3.3.

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 7.9.1.

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, a given face may not appear in more than one contact surface.

7.3.2 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 7.3.3 for further information about using a node set that contacts a surface.

There must be at least one BLOCK, SURFACE, or NODE SET command line in the command block.

7.3.3 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 a mesh surface when contact interactions are defined. 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
```

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 Adagio will not currently handle any contact between two one-dimensional elements.

7.4 Surface Normal Smoothing

```
BEGIN SURFACE NORMAL SMOOTHING
  ANGLE = <real>angle_in_deg(60.0)
  DISTANCE = <real>distance(0.01)
  RESOLUTION = <string>NODE|EDGE(NODE)
END SURFACE NORMAL SMOOTHING
```

Finite element discretization often results in models with faceted edges, while the true geometry of the part is actually smoothly curved. If the faces of adjacent finite elements on a surface have differing normals, the discontinuities at the edges between those faces can cause problems with contact. These discontinuities in the face normals are particularly troublesome with an implicit code such as Adagio, which uses an iterative solver to obtain a converged solution at every step. If a node is in contact near an edge with a normal discontinuity, the node may slide back and forth between the two neighboring faces during the iterations. Because the normal directions of the two faces differ, this can make it difficult to converge on a solution to this discontinuous contact problem.

Surface normal smoothing is a technique that creates a smooth variation in the normal near edges. 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 obtain a converged solution in the case where a node has penetrated near the edge of a face.

Presto does not use an iterative solver and thus does not encounter the difficulties associated with face normal discontinuities. Consequently, the `SURFACE NORMAL SMOOTHING` command block is not typically useful for Presto models. It is provided in both Presto and Adagio, however, to provide a consistent transition between the two codes if they are used together in a coupled analysis.

If the `SURFACE NORMAL SMOOTHING` command block is present, this feature is activated. There are three optional commands that can be used within this block to control the behavior of normal smoothing.

- The `ANGLE` command is used to control whether smoothing should occur between neighboring faces. If the angle between two faces is less than the specified angle (given in degrees), smoothing is activated between them. Otherwise, the discontinuity is considered to be a feature of the model rather than an artifact of meshing, and they are not smoothed. The default value for `angle` is 60.

- The `DISTANCE` command specifies the distance as a fraction of the face size over which smoothing should occur. The specified value can vary from 0 to 1. The default value for `distance` is 0.01.

- The `RESOLUTION` command specifies the method used to determine the smoothed normal direction. The default `NODE` option uses a node-based algorithm to fit a smooth curve, while the `EDGE` option uses an edge-based algorithm.

7.5 Contact Output Variables

Contact variables can be output to provide information about enforcement of contact interactions. 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.

Nodal contact variables that can be output are listed in Table 8.7. Where applicable, names of the equivalent variables in JAS3D are given in parentheses at the end of the description. The variables can be output in history files or results files; see Chapter 8 for more information on outputting nodal variables. Note that currently the variables cannot be calculated at output time so the first time they are output a request is made to calculate them. This means that the first output step where they are to appear the data will be all zero. A work around for this is to have at least one output step in which these variables appear before their values are needed.

7.6 Friction Models

To describe the physics of interactions that occur between contact surfaces, the Adagio 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 7.8 and Section 7.9). 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.

The following friction models are currently available: frictionless contact, constant coulomb friction, tied contact, glued contact, spring weld, surface weld, area weld, adhesion, cohesive zone, junction, threaded joint, and pressure-velocity–dependent friction. In addition, models defined by user subroutines can be used as friction models. 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 7.8 and Section 7.9. Adagio 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.

7.6.1 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. A default named frictionless model named `frictionless` can be used without defining this command block.

7.6.2 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.

7.6.3 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. A default named tied model named `tied` can be used without defining this command block.

7.6.4 Glued Model

```
BEGIN GLUED MODEL <string>name
END [GLUED MODEL <string>name]
```

The `GLUED MODEL` command block defines a contact interaction that allows the interacting faces to move independently until they come into contact, but once they come into contact, they behave as a tied contact interaction, with no relative normal or tangential motion for the rest of the analysis. The string `name` is a user-specified name for this friction model that is used to reference this model in the interaction definitions. No command lines are needed inside the command block. A default named glued model named `glued` can be used without defining this command block.

7.7 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
  FACE MULTIPLIER = <real>face_multiplier(0.1)
  CAPTURE TOLERANCE = <real>cap_tol
  TENSION RELEASE = <real>ten_release
  SLIP PENALTY = <real>slip_pen
  #
  # secondary decomposition
  SECONDARY DECOMPOSITION = <string>ON|OFF(OFF)
END [SEARCH OPTIONS <string>name]
```

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 7.7.1, Section 7.7.2, and Section 7.7.3. The values specified by these commands are applied by default to all interaction contact surfaces, unless overridden by a specific interaction definition.

7.7.1 Search Algorithms

```
    GLOBAL SEARCH INCREMENT = <integer>num_steps(1)
    GLOBAL SEARCH ONCE = <string>ON|OFF(OFF)
```

⚠ **Known Issue:** Attempting to use `GLOBAL SEARCH INCREMENT` with a value greater than 1, especially in a problem that contains shells and/or restart, will, in most cases, cause code failure. A `GLOBAL SEARCH INCREMENT` value greater than 1 will, under the best circumstances, give only a marginal improvement in speed.

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.

7.7.2 Search Tolerances

```
SEARCH TOLERANCE = <string>AUTOMATIC|USER_DEFINED(AUTOMATIC)
NORMAL TOLERANCE = <real>norm_tol
TANGENTIAL TOLERANCE = <real>tang_tol
FACE MULTIPLIER = <real>face_multiplier(0.1)
```

As indicated previously, the contact functionality in Adagio 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 7.3). 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, Adagio will automatically calculate normal and tangential tolerances based on the minimum characteristic length multiplied by the value input by the `FACE MULTIPLIER` command. The face multiplier is 0.1. The automatic tolerances add the maximum motion over a time step just like the user defined tolerances. 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 (automatic or user specified) tolerances will be used. For example, suppose you specify a normal tolerance of 1.0×10^{-3} and the automatic tolerancing computes a normal tolerance of 1.05×10^{-3}. Then Adagio 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.

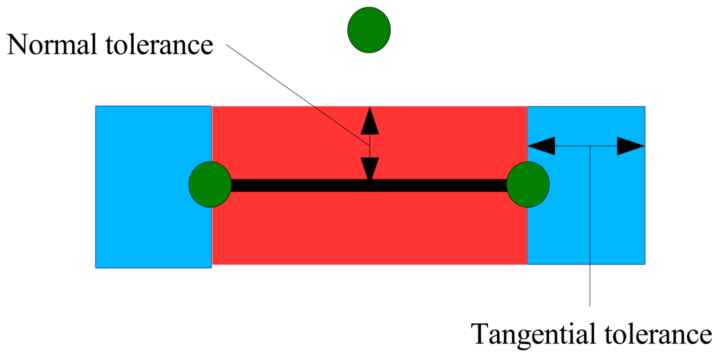


Figure 7.3: Illustration of normal and tangential tolerances.

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 7.9).

7.7.3 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.

7.8 Default Values for Interactions

```
BEGIN INTERACTION DEFAULTS
  CONTACT SURFACES = <string list>surface_names
  GENERAL CONTACT = <string>ON|OFF(OFF)
  FRICTION MODEL = <string>friction_model_name|FRICTIONLESS
    (FRICTIONLESS)

END [INTERACTION DEFAULTS]
```

This section discusses the `INTERACTION DEFAULTS` command block. This command block enables contact enforcement either on all contact surfaces or on a subset of the contact surfaces. It is also used to set default parameters for contact interactions. Those defaults can be overridden for specific interactions by specifying them separately in `INTERACTION` blocks.

It is important to note that unless some combination of the `INTERACTION DEFAULTS` command block and `INTERACTION` command blocks (Section 7.9) exists in the `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 `CONTACT 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 7.9.

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.

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.

7.8.1 Surface Identification

```
CONTACT 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 `CONTACT SURFACES` command line specifies the names of these contact surfaces. The `CONTACT 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`.

7.8.2 General Contact

```
GENERAL CONTACT = <string>ON|OFF(OFF)
```

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 this command line is `OFF`. To enforce general contact between all surfaces specified in the `INTERACTION DEFAULTS` command block, this line must be present:

```
GENERAL CONTACT = ON
```

If no individual contact interactions have been specified with `INTERACTION` command blocks, contact will only be enforced if general contact is enabled.

7.8.3 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 7.6). 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.

7.9 Values for Interactions

```
BEGIN INTERACTION [<string>name]
  MASTER   = <string_list>surfaces [EXCLUDE <string_list>surfaces]
  SLAVE    = <string_list>surfaces [EXCLUDE <string_list>surfaces]
  CAPTURE TOLERANCE = <real>cap_tol
  NORMAL TOLERANCE = <real>norm_tol
  TANGENTIAL TOLERANCE = <real>tang_tol
  FRICTION MODEL = <string>friction_model_name|FRICTIONLESS
    (FRICTIONLESS)
  PUSHBACK FACTOR = <real>pushback_factor(1.0)
  TENSION RELEASE = <real>ten_release
  TENSION RELEASE FUNCTION = <string>ten_release_func
  FRICTION COEFFICIENT = <real>coeff
  FRICTION COEFFICIENT FUNCTION = <string>coeff_func
END [INTERACTION <string>name]
```

The Adagio contact input also permits the setting of values for specific interactions using the `INTERACTION` command block.

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 used only for informational output purposes and is not required.

The valid commands within an `INTERACTION` command block are described in Section 7.9.1 through Section 7.9.8.

7.9.1 Surface Identification

```
MASTER   = <string_list>surfaces [EXCLUDE <string_list>surfaces]
SLAVE    = <string_list>surfaces [EXCLUDE <string_list>surfaces]
```

In Adagio, contact surfaces must be identified using the `MASTER` and `SLAVE` command lines. The nodes of the slave surfaces are searched against the faces of the master surfaces. Each of these command lines takes as input a list of names of contact surfaces defined in the contact block. A master slave interaction will be defined between each surface in the master list and each surface in the slave list. A surface may not be present in both the master and the slave list.

The surface named `all_surfaces` is a special reserved word that is equivalent to typing all contact surfaces known by the contact block into the string list. Optionally, the `EXCLUDE` keyword can be placed on the command line and followed by a list of surface names to exclude from the list. If the `MASTER` or `SLAVE` command lines appear multiple times within a `CONTACT INTERACTION` block, their surface lists will be concatenated. The effect is equivalent to specifying all of the surface names on a single line.

The following examples demonstrate ways to identify contact surfaces involved in an interaction:

These commands define a one-way interaction between the nodes of `s1` and the faces of `m1`:

```
MASTER = m1
SLAVE = s1
```

These commands define a set of one-way interactions between the nodes of `s1` and the faces of `m1`, the nodes of `s1` and the faces of `m2`, the nodes of `s2` and the faces of `m1`, the nodes of `s2` and the faces of `m2`.

```
MASTER = m1 m2
SLAVE = s1 s2
```

These commands define that the nodes of surface `s1` are slaved to all other contact faces in the contact definition block.

```
MASTER = all_surfaces exclude s1
SLAVE = s1
```

7.9.2 Tolerances

```
    CAPTURE TOLERANCE = <real>cap_tol
    NORMAL TOLERANCE = <real>norm_tol
    TANGENTIAL TOLERANCE = <real>tang_tol
    FACE MULTIPLIER = <real>face_multiplier(0.1)
```

You can set tolerances for the interaction for a specific contact surface pair by using the above tolerance-related command lines in an `INTERACTION` command block.

As indicated previously, the contact functionality in Adagio 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 7.3). 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.

The `CAPTURE TOLERANCE`, which should be no larger than the `NORMAL TOLERANCE`, is used to determine which slave nodes near a master surface should be pulled to the master surface and considered for contact. (This applies to slave nodes which have not penetrated the master surface. Slave nodes that have penetrated the master surface will be pushed to the surface regardless of the CAPTURE TOLERANCE.) If later checks determine that a slave node that was pulled to the surface (because it was nearer the surface than the CAPTURE TOLERANCE value) is in fact in tension and should be released, that slave node will not be considered as a potential contact node again during the load step as long as the node remains within the NORMAL TOLERANCE.

7.9.3 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 7.8.3 for a discussion of this command line.

7.9.4 Pushback Factor

```
    PUSHBACK FACTOR = <real>pushback_factor
```

The command line PUSHBACK FACTOR can be used to set the fraction of the gap to be removed in a contact model problem. The default value is 1.0 which removes the entire gap in one contact model problem. Setting the pushback factor to 0.25 will result in the gap being removed in 4 contact model problems.

7.9.5 Tension Release

```
    TENSION RELEASE = <real>ten_release
```

The command line TENSION RELEASE can be used to set a traction threshold below which slave nodes that have come into contact with master faces will not be released. When this value is set and a slave node is in tension, it will only be allowed to pull away from the master surface if the slave node's traction is greater than the TENSION RELEASE tolerance.

7.9.6 Tension Release Function

```
    TENSION RELEASE FUNCTION = <real>ten_release_func
```

The command line TENSION RELEASE FUNCTION provides a way for the tension release threshold to be set using a function from the input file. If the TENSION RELEASE line command is also present, the threshold used by the code will be the TENSION RELEASE value multiplied by the value obtained from the function.

7.9.7 Friction Coefficient

```
FRICTION COEFFICIENT = <real>coeff
```

You can set the coefficient of friction for the interaction for a specific contact surface pair by using the above command line in an `INTERACTION` command block.

7.9.8 Friction Coefficient Function

```
FRICTION COEFFICIENT FUNCTION = <real>coeff_func
```

The command line `FRICTION COEFFICIENT FUNCTION` provides a way for the friction coefficient to be set using a function from the input file. If the `FRICTION COEFFICIENT` line command is also present, the friction coefficient used by the code will be the `FRICTION COEFFICIENT` value multiplied by the value obtained by the function.

7.10 Legacy Contact

This section describes the use of the contact library written for JAS3D. This library is referred to as the Legacy Contact Library or LCLib.

Adagio has two contact search and enforcement options. The first option, the default, is to use ACME for the contact search and Adagio's own enforcement library. The second option is to use JAS3D's contact library for both the search and the enforcement. While the capabilities of the two options are nominally the same, one may perform better than the other for certain problems.

To invoke the use of LCLib, include the line command

```
    JAS MODE
```

in the region.

When using LCLib, only one `CONTACT DEFINITION` block is allowed in the region. The `ENFORCEMENT` line command must invoke the `FRICTIONAL` type. If frictionless contact is desired for an interaction, set the `FRICTION COEFFICIENT` to zero. If tied contact is desired for an interaction, set the `FRICTION COEFFICIENT` to -1.

7.11 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.

7.11.1 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 7.4.

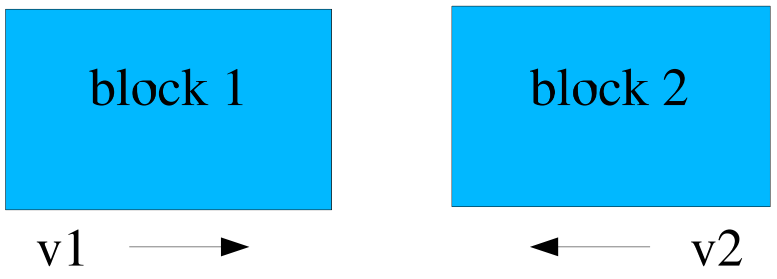


Figure 7.4: Problem with two blocks coming into contact.

The simplest input for this problem will be named `EXAMPLE1` and is shown as follows:

```
BEGIN CONTACT DEFINITION EXAMPLE1

  # enforcement option
  ENFORCEMENT = FRICTIONLESS

  # contact surfaces
  CONTACT SURFACE B1 CONTAINS BLOCK_1
  CONTACT SURFACE B2 CONTAINS BLOCK_2

  # set interactions
  BEGIN INTERACTION EX1
    MASTER = B1
    SLAVE = B2
    CAPTURE TOLERANCE = 1.0E-3
    NORMAL TOLERANCE = 1.0E-3
    TANGENTIAL TOLERANCE = 1.0E-3
  END INTERACTION EX1
END
```

In this example, we define frictionless contact.

Now, let us consider the same problem (two blocks coming into contact) with frictional contact. The input for this variation of our two-block problem will be named `EXAMPLE1A` and is shown as follows:

```
BEGIN CONTACT DEFINITION EXAMPLE1A

  # enforcement option
  ENFORCEMENT = FRICTIONAL

  # contact surfaces
  CONTACT SURFACE B1 CONTAINS BLOCK_1
  CONTACT SURFACE B2 CONTAINS BLOCK_2

  # set interactions
  BEGIN INTERACTION EX1A
    MASTER = B1
    SLAVE = B2
    FRICTION COEFFICIENT = 0.5
    CAPTURE TOLERANCE = 1.0E-3
    NORMAL TOLERANCE = 1.0E-3
    TANGENTIAL TOLERANCE = 1.0E-3
  END INTERACTION EX1A
END
```

For `EXAMPLE1A`, we want to have frictional contact between the two blocks. For the frictional contact, we specify the coefficient of friction as `0.5`.

7.11.2 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 7.5.

The input for this three-block problem will be named `EXAMPLE2` and is shown as follows:

```
BEGIN CONTACT DEFINITION EXAMPLE2

   # enforcement
  ENFORCEMENT = FRICTIONLESS

  # define contact surfaces
  CONTACT SURFACE surface_1 CONTAINS block_1
  CONTACT SURFACE surface_2 CONTAINS block_2
  CONTACT SURFACE surf_3 CONTAINS surface_3
```

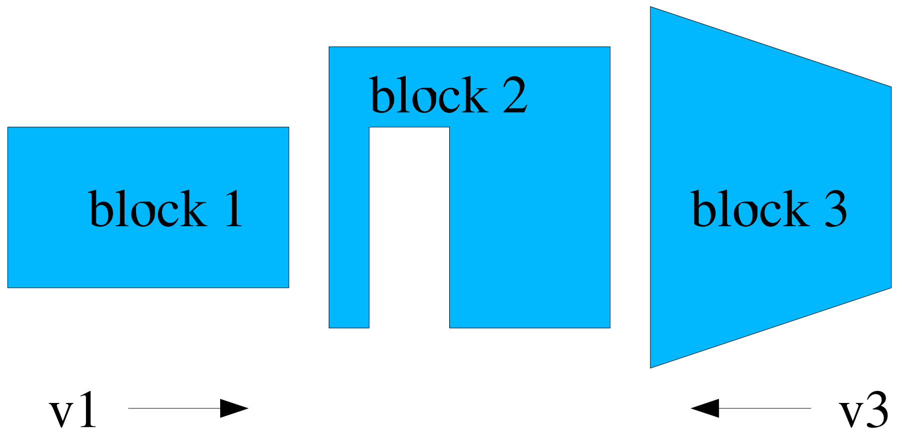


Figure 7.5: Problem with three blocks coming into contact.

```
    # set interaction
    BEGIN INTERACTION S1TOS2
      MASTER = surface_2
      SLAVE = surface_1
      NORMAL TOLERANCE = 1.0E-3
      TANGENTIAL TOLERANCE = 1.0E-3
    END INTERACTION S2TOS3

    # set interaction
    BEGIN INTERACTION S2TOS3
      MASTER = surface_2
      SLAVE = surf_3
      NORMAL TOLERANCE = 0.5E-3
      TANGENTIAL TOLERANCE = 0.5E-3
    END INTERACTION S2TOS3
  END
```

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`.

7.12 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. pdf.

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

Adagio produces a variety of output. This chapter discusses how to control the four major types of output: results output, history output, heartbeat output, and restart output. Results output lets the user select a set of variables (internal, 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 stress for results output, the stress for all elements in the model that calculate this quantity (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 heartbeat output is similar to the history output except that the output is written to a text file instead of to a binary (exodusII [1]) file. The restart output is written so that any calculation 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 8.2 describes the results output. Included in the results output is a description of commands for user-defined output (Section 8.2.2). User-defined output lets the user postprocess analysis results as the code is running to produce a reduced set of output information. Section 8.3 describes the history output, Section 8.4 describes the heartbeat output, and Section 8.5 describes the restart output. All four types of output (results, history, heartbeat, and restart) can be synchronized for analyses with multiple regions. This scheduling functionality is discussed in Section 8.6. In Section 8.9, there is a list of key variables.

Unless otherwise noted, the command blocks and command lines discussed in Chapter 8 appear in the region scope.

8.1 Syntax for Requesting Variables

Variables may be accessed in the code either in whole or by component. Values at specific components or integration points of multi-component variables may be accessed via parenthesis syntax. Parenthesis syntax may be applied to results output, history output, element death, or any other command where variable names are specified. Values of single-component variables indexed in some other way may be accessed with underscore syntax, which is primarily applicable to rigid body fields as discussed in Section 8.1.4.

Parenthesis syntax is a variable name of the form:

```
<string>var_name[(<index>component[,<integer>integration_point)]]
```

For a variable named `var`, a variable name of the form `var(A,B)` asks for the `A` component of the variable at integration point `B`. If a variable is a vector, `x`, `y`, or `z` may be specified as the component. If a variable is 3x3 tensor, `xx`, `yy`, `zz`, `xy`, `xz`, `yz`, `yx`, `zx`, or `zy` may be specified as the component. For other types of variables components of the variable may be requested through an integer index.

The characters : and * are wild cards if used for specifying either the component or the integration point. `var(:,B)` asks for all components of var at integration point B. `var(A,:)` asks for component A of var at all integration points. `var(:,:)` asks for all components of var at all integration points. `var` is shorthand for `var(:,:)`.

`var(A)` will behave slightly differently depending on the nature of the variable. If the variable has multiple components, then `Var(A)` is treated like `var(A,:)`. If a variable has one and only one component then it is assumed that A refers to the integration point number rather than the component number and `Var(A)` is treated like `var(1,A)`.

8.1.1 Example 1

Let stress be a tensor defined on a single integration point element and a displacement vector be defined at all model nodes. The following output variable specification:

```
    element stress as str
    nodal displacement as disp
```

asks for all the components of the stress tensor on elements and all components of the displacement vector on nodes. The code would write the following variables to the output file:

```
str_xx
str_yy
str_zz
str_xy
str_xz
```

```
  str_yz
  disp_x
  disp_y
  disp_z
```

If only the `yy` component of stress is desired, either of the following could be used:

```
  element stress(yy) as my_yy_str1
  element stress(2) as my_yy_str2
```

If only the `z` component of displacement is desired either of the following could be specified:

```
  nodal displacement(z) as my_z_disp1
  nodal displacement(3) as my_z_disp2
```

Note, index 2 of a tensor corresponds to the `yy` component of the tensor and index 3 of a vector corresponds to the `z` component of the vector.

8.1.2 Example 2

Let stress be a tensor defined on each integration point of a three integration point element. Let eqps be a scalar material state variable also defined at each element integration point. To ask for all stress components on all integration points and all eqps data at all integration points the following could be specified:

```
  element stress as str
  element eqps as eqps
```

Which would output the variables:

```
  str_xx_1, str_yy_1, str_zz_1, str_xy_1, str_xz_1, str_yz_1
  str_xx_2, str_yy_2, str_zz_2, str_xy_2, str_xz_2, str_yz_2
  str_xx_3, str_yy_3, str_zz_3, str_xy_3, str_xz_3, str_yz_3
  eqps_1, eqps_2, eqps_2
```

To ask for just the stress tensor and eqps value on the second integration point the following syntax can be used:

```
  element stress(:,2) as str_intg2
  element eqps(2) as eqps_intg2
```

This would output the variables:

```
str_intg2_xx
str_intg2_yy
str_intg2_zz
str_intg2_xy
str_intg2_xz
str_intg2_yz
eqps_intg2
```

To ask for the `xy` component of stress on all integration points any of the following could be used:

```
element  stress(xy,*) as str_xy_all
element  stress(xy,:) as str_xy_all
element  stress(xy)   as str_xy_all
```

Any of the above would output:

```
str_xy_all_1
str_xy_all_2
str_xy_all_3
```

8.1.3 Other command blocks

The parenthesis syntax described above for results output can also be used in most other commands involving variable names. For example, to kill elements based on yy stress or z displacement the following could be specified:

```
begin element death
  criterion is element value of stress(yy) > 1000
  criterion is average nodal value of displacement(z) > 3.0
end
```

8.1.4 Rigid Body Variables

Variables of rigid bodies are provided as separate components of the rigid body fields. To access variables of rigid bodies you must output the desired field component(s), and to access the field component of individual rigid bodies an underscore syntax is employed (the parenthesis syntax applies to multi-component fields and integration points). The underscore syntax simply takes the desired field component and appends it with an underscore and the desired rigid body name.

For example the following lines in a history output block (see Section 8.3)

```
global ax_rb1
global displz_rb4
```

would output to the history file the variables

```
  ax_rb1
  displz_rb4
```

which are the translational acceleration in the x-direction of rigid body rb1 and the translational displacement in the z-direction of rigid body rb4, respectively (see Table 8.2).

8.2 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.

8.2.1 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  <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | NODAL  <string>variable_name
        [AS <string>dbase_variable_name] ...
        <string>variable_name [AS
        <string>dbase_variable_name]
  NODESET  <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | NODESET  <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  <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | FACE  <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  <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | ELEMENT  <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
  GLOBAL  <string>variable_name
```

```
        [AS <string>dbase_variable_name] ...
        <string>variable_name [AS
        <string>dbase_variable_name]
      OUTPUT MESH = EXPOSED_SURFACE|BLOCK_SURFACE
      COMPONENT SEPARATOR CHARACTER = <string>character|NONE
      EXCLUDE = <string>list_of_excluded_element_blocks
      INCLUDE = <string>list_of_included_element_blocks
      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
      SYNCHRONIZE_OUTPUT
      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 is 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 must include the path to that directory.

Two metacharacters can appear in the name of the results file. If the `%P` character is found in the name, it will be replaced with the number processors being used for the run. For example, if you are running on 1024 processors and use the name `results-%P/job.e`, then the name would be expanded to `results-1024/job.e` and the actual results files would be `results-1024/job.e.1024.0000` to `results-1024/job.g.1024.1023`. The other recognized metacharacter is `%B` which is replaced with the base name of the input file containing the input commands. For example, if the commands are in the file `my_analysis_run.i` and the results database name is specified as `%B.e`, then the results would be written to the file `my_analysis_run.e`.

If the results file does not use the Exodus II format [1], 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 [2] are supported in Presto and Adagio. 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 8.5.1.1. 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 8.5.1.2) 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 [3]) 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 8.2.1.1 through Section 8.2.1.18.

8.2.1.1 Output Nodal Variables

```
NODE  <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS <string>dbase_variable_name]
  | NODAL  <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS <string>dbase_variable_name]
```

Any nodal variable in Adagio 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 to output. The string `variable_name` can be either a variable listed in Section 8.9 or a user-defined variable (see Section 8.2.2 and Section 10.2.4).

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 Adagio, which are defined as `force_external`, with the alias `f_ext`. You would then enter the command line:

```
NODE 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`. You can change the component separator, an underscore in this example, by using the `COMPONENT SEPARATOR CHARACTER` command line (see Section 8.2.1.7).

The `NODE` command line can be used an arbitrary 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 defined as `force_internal`.

```
NODE 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 force_external AS f_ext
     force_internal AS f_int
```

The specification of an alias is optional.

8.2.1.2 Output Node Set Variables

```
NODESET  <string>variable_name
  [AS <string>dbase_variable_name] ...
```

```
    <string>variable_name [AS <string>dbase_variable_name]
    | NODESET  <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 defined on a subset of the total set of nodes defining a model. A nodal variable that is defined only on a subset of nodes is referred to as a node set variable. The `NODESET` command line lets you specify a node set variable for output to the results file.

There are two forms of the `NODESET` command line. Either form will let you output a node set variable.

The first form of the command line is as follows:

```
  NODESET  <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` option. Suppose, for example, you wanted to output the node set variable `force_nsetype`, but have that variable have the name `fnsetype` in the results file. You would then enter the command line:

```
NODESET force_nsetype AS fnsetype
```

The `NODESET` command line can be used an arbitrary 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 defined as `force_nsetype2`.

```
NODESET 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 force_nsetype AS fnsetype
     force_nsetype2 AS fnsetype2
```

The specification of an alias is optional.

The second form of the command line is as follows:

```
  NODESET  <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` 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 keyword `INCLUDE` is synonymous with the keyword `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 defined 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` command line as follows:

```
NODESET 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 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.

8.2.1.3 Output Face Variables

```
FACE  <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS <string>dbase_variable_name]
  | FACE  <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 defined on some set of faces that constitute a surface. A variable defined on a set of faces is referred to as a face variable. The `FACE` command line lets you specify a face variable for output to the results file.

There are two forms of the `FACE` command line. Either form will let you output a face variable.

The first form of the command line is as follows:

```
FACE  <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` option. Suppose, for example, you wanted to output a face variable `pressure_face`, but have that variable have the name `pressuref` in the results file. You would then enter the command line:

```
FACE pressure_face AS pressuref
```

The `FACE` command line can be used an arbitrary 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 defined as `scalar_face2`.

```
FACE 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 pressure_face AS pressuref
     scalar_face2 AS scalarf2
```

The specification of an alias is optional.

The second form of the command line is as follows:

```
FACE  <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` 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 keyword `INCLUDE` is synonymous with the keyword `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 defined 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` command line as follows:

```
FACE 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 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.

8.2.1.4 Output Element Variables

```
ELEMENT  <string>variable_name
  [AS <string>dbase_variable_name] ...
  <string>variable_name [AS
  <string>dbase_variable_name]
  | ELEMENT  <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 Adagio can be selected for output in the results file by using the ELEMENT command line.

There are two forms of the ELEMENT command line. Either form will let you output an element variable.

The first form of the command line is as follows:

```
ELEMENT  <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 to output. The string variable_name can be a variable listed in Section 8.9 or a user-defined variable (see Section 8.2.2 and Section 10.2.4).

In the first form of the ELEMENT command line, the element variable is output for all element blocks that have the element variable as a defined variable. For example, all the solid elements have stress as a defined variable. If you had a mesh consisting of hexahedral and tetrahedral elements and requested output of the element variable stress, then 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 Adagio, which is defined as stress, with the alias str. You would then enter the command line:

```
ELEMENT stress AS str
```

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. You can change the tensor component separator, an underscore in this example, by using the COMPONENT SEPARATOR CHARACTER command line (see Section 8.2.1.7).

The ELEMENT command line can be used an arbitrary 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 defined as left_stretch.

```
ELEMENT stress left_stretch
```

Aliases can be specified for each of the variables in a single command line. Thus, If you wanted to output the alias str for element variable stress and also wanted to output the alias strch for face variable lseft_stretch, you would enter the command line:

```
ELEMENT stress AS str
     left_stretch AS strch
```

The specification of an alias is optional.

The second form of the command line is as follows:

```
ELEMENT  <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 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 keyword INCLUDE is synonymous with the keyword ON.) Alternatively, the user can exclude a specific list of element blocks for output by using the EXCLUDE option.

Suppose that the element variable `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` command line as follows:

```
ELEMENT stress AS str
      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 stress AS str
      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.

For multi-integration point elements, quantities from the integration points are appended with a numerical index indicating the integration point. A suffix ranging from 1 to the number of integration points is attached to the 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 quantities `von_mises_01`, `von_mises_02`, ..., `von_mises_15` are output for the shell element.

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 8.2.1.7).

Shell tensor quantities `transform_shell_stress`, `transform_shell_strain` and `transform_shell_rate_of_deformation` may be transformed to a user specified shell local co-rotational coordinate system (i.e. an in-plane coordinate system that rotates with the shell element) for output using the `ORIENTATION` shell section command. If no orientation is specified, these in-plane stresses and strains are output in the default orientation. See Section 5.2.4 for more details.

8.2.1.5 Subsetting of Output Mesh

A specified subset of the element blocks in the mesh can be output to the results database using the `INCLUDE` or `EXCLUDE` commands. The syntax is:

```
INCLUDE = <string>list_of_included_element_blocks
EXCLUDE = <string>list_of_excluded_element_blocks
```

Either command can appear multiple times within the results output block, but the two cannot be mixed within a single results output block. If the `INCLUDE` command is specified, the results database will only contain the listed element blocks; if the `EXCLUDE` command is specified, the results database will contain all element blocks except for the listed element blocks. If the model has surfaces or nodesets, only the portion of the surfaces or nodesets on the selected element blocks will be output.

8.2.1.6 Output Mesh Selection

```
OUTPUT MESH = EXPOSED_SURFACE|BLOCK_SURFACE
```

The `OUTPUT MESH` command provides a way to reduce the amount of data that is written to the results database. There are two options that can be selected:

`EXPOSED_SURFACE` Only output the element faces that make up the "skin" of the finite element model; no internal nodes or elements will be written to the results database. The element results variables will be applied to the skin faces. If the mesh is visualized without any cutting planes, the display should look the same as if the original full mesh were visualized; however, the amount of data written to the output file can be much less than is needed if the full mesh were output.

`BLOCK_SURFACE` This option is similar to the `EXPOSED_SURFACE` option except that the skinning process is done an element block at a time instead of for the full model. In this option, faces shared between element blocks will appear in the output model.

8.2.1.7 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 appear in the results output file as `stress.xx`, `stress.yy`, etc. 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`.

8.2.1.8 Output Global Variables

```
GLOBAL  <string>variable_name
  [AS <string>dbase_variable_name
  <string>variable_name AS <string>dbase_variable_name ...]
```

Any global variable in Adagio can be selected for output in the results file by using the `GLOBAL` command line. The string `variable_name` is the name of the global variable. The string `variable_name` can be either a variable listed in Section 8.9 or a user-defined variable (see Section 8.2.2 and Section 10.2.4).

Kinetic, external, hourglass, and internal energies can be requested as the sum over the entire mesh or as sums over individual element blocks as noted in Section 8.9. For example, if the mesh contains element blocks with IDs 100 and 200, the kinetic and hourglass energy summed over each of these blocks individually can be requested with the commands

```
GLOBAL  ke_block100 as ke100
GLOBAL  ke_block200 as ke200
GLOBAL  hge_block100 as hge100
GLOBAL  hge_block200 as hge200
```

and the total kinetic energy as

```
GLOBAL  kinetic_energy as ke
```

Kinetic, internal, and external energies are computed as nodal values. Since nodes may be shared by element blocks, the total sum of an energy quantity over individual element blocks will not necessarily be equal to the global sum requested using the `kinetic_energy`, `internal_energy`, or `external_energy` variable names.

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 Adagio, which are identified as `timestep`, with the alias `tstep`. You would then enter the command line:

```
GLOBAL timestep AS tstep
```

The GLOBAL command line can be used an arbitrary 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 total number of iterations , which is defined as total_iter, with the alias ti, you would enter the command line:

```
GLOBAL timestep as tstep
    total_iter as ti
```

The specification of an alias is optional.

8.2.1.9 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.

8.2.1.10 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.

8.2.1.11 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.

8.2.1.12 Additional Times for Output

```
ADDITIONAL TIMES = <real>output_time1 <real>output_time2 ...
```

In addition to any times specified by the command line in Section 8.2.1.11, you can use the ADDITIONAL TIMES command line to specify an arbitrary number of additional output times.

8.2.1.13 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`.

8.2.1.14 Additional Steps for Output

```
ADDITIONAL STEPS = <integer>output_step1
  <integer>output_step2 ...
```

In addition to any steps specified by the command line in Section 8.2.1.13, you can use the `ADDITIONAL STEPS` command line to specify an arbitrary number of additional output steps.

8.2.1.15 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.

8.2.1.16 Synchronize Output

```
SYNCHRONIZE OUTPUT
```

In an analysis with multiple regions, it is sometimes desirable to synchronize the output of results data between the regions. This can be done by adding the `SYNCHRONIZE OUTPUT` command line to the results output block. If a results block has this set, then it will write output whenever a previous region writes output. The ordering of regions is based on the order in the input file, algorithmic considerations, or by solution control specifications.

Although the `USE OUTPUT SCHEDULER` command line can also synchronize output between regions, the `SYNCHRONIZE OUTPUT` command line will synchronize the output with regions where the output frequency is not under the direct control of the Sierra IO system. Examples of this are typically coupled applications where one or more of the codes are not Sierra-based applications such as Alegra and CTH. A results block with `SYNCHRONIZE OUTPUT` specified will also synchronize its output with the output of the external code.

The `SYNCHRONIZE OUTPUT` command can be used with other output scheduling commands such as time-based or step-based output specifications.

8.2.1.17 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 SIERRA scope. 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 8.6 for a description of using this command block and the `USE OUTPUT SCHEDULER` command line.

8.2.1.18 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.

8.2.2 User-Defined Output

```
BEGIN FILTER <string>filter_name
  ACOEFF = <real_list>a_coeff
  BCOEFF = <real_list>b_coeff
  INTERPOLATION TIME STEP = <real>ts
END [FILTER]

BEGIN USER OUTPUT
  # 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
  #
  # compute result commands
  COMPUTE GLOBAL <string>result_var_name AS
    SUM|AVERAGE|MAX|MIN OF NODAL|ELEMENT <string>value_var_name
  COMPUTE NODAL <string>result_var_name AS
    MAX OVER TIME|MIN OVER TIME|ABSOLUTE VALUE MAX
    OVER TIME OF NODAL <string>value_var_name
  COMPUTE ELEMENT <string>result_var_name AS
    MAX OVER TIME|MIN OVER TIME|ABSOLUTE VALUE MAX
    OVER TIME OF ELEMENT <string>value_var_name
  #
  # 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

  # variable transformation command
  TRANSFORM NODAL|ELEMENT VARIABLE <string>variable_name
    TO COORDINATE SYSTEM <string>coord_sys_name
```

```
      AS <string> transformed_name
    #
    # Data filtering
    FILTER <string>new_var FROM NODAL|ELEMENT <string>source_var
      USING <string>filter_name


    #
    # additional command
    ACTIVE PERIODS = <string list>period_names
    INACTIVE 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 an external code to produce specialized output results.

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 variables on subsets of the mesh. This option makes use of one of the compute result command lines. For instance, the above example of the contact force is a case 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 Adagio. You can use only one of the three options—compute global result, user subroutine, or copy—in a given command block.

For the compute 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 any of the three compute result command lines. Parenthesis syntax (Section 8.1) may be used to define reductions on specific integration points or components of a variable. By default, a reduction operation operates on each integration point. For example, if the compute global command was used to average values of element stress it would average the values of stress at all integration points of all elements. If the command was used to average `stress(:,1)`, the result would only be the average of stress on the first integration points.

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 10.2.4.

User-defined variables, whether they are generated via the compute 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 8.2.1 and Section 8.3, which describe the output of variables to results files and history files, respectively).

The USER OUTPUT command block contains four groups of commands—mesh-entity set, compute result, user subroutine, and copy. Each of these command groups is basically independent of the others. In addition to the command lines in the four 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.

8.2.2.1 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>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 6.1.1 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.

8.2.2.2 Compute Result Commands

The compute result commands are used to compute new variables by performing operations on an existing variable. Currently three general forms of the compute results command are supported:

```
COMPUTE GLOBAL result_var_name AS SUM|AVERAGE|MAX|MIN
  OF NODAL|ELEMENT source_var_name
COMPUTE NODAL result_var_name AS
  MAX OVER TIME|MIN OVER TIME|ABSOLUTE VALUE MAX OVER TIME
   OF NODAL source_var_name
COMPUTE ELEMENT result_var_name AS
```

```
  MAX OVER TIME|MIN OVER TIME|ABSOLUTE VALUE MAX OVER TIME
   OF ELEMENT source_var_name
```

The compute global result command returns a single global value or a set of global values by examining the current values for a named 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.

The compute nodal and element variable commands compute a new nodal or element field by operating on an existing field. These commands can be used for computing such things as the maximum stress in each element over the course of the analysis.

In the above command lines, the following definitions apply:

- The string `result_var_name` is the name of a new variable in which the computed results are stored. To output this variable in a results file, a heartbeat file, or a history file, you will simply use whatever you have selected for `results_var_name` as the variable name in the output block.

- Four different global reduction methods are available for computing global variables: `SUM`, `AVERAGE`, `MAX`, and `MIN`. `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.

- Three different over time methods are available for computing nodal or element variables: `MAX OVER TIME`, `MIN OVER TIME`, and `ABSOLUTE VALUE MAX OVER TIME`. These options compute the max, min, or absolute max over time of values in the source variable.

- The source variable used to compute a global variable must be either a nodal quantity or an element quantity, as specified by the `NODAL` or `ELEMENT` option. The variable source variable used to compute a nodal or element variable must be of the same type as the result variable.

- The string `source_var_name` is the name of the variable used to compute the result variable. (see Section 8.9 for a listing of available code variables).

- Standard component syntax may also be used in the `source_variable_name` to specify operating on sub-components of a given variable (such as only `stress(xx)`).

The following is an example of using the `COMPUTE` command line to compute the net *x*-direction reaction force:

```
  COMPUTE GLOBAL wall_x_reaction AS SUM OF NODAL reaction(x)
```

The following is an example of using the `COMPUTE` command line to compute the maximum force seen by the spot welds.

```
  COMPUTE NODAL max_spotn AS MAXIMUM OVER TIME OF NODAL spot_weld_
normal_force
```

8.2.2.3 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 Adagio 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 10.

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`. See Chapter 10 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 10.2.2 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 10.

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 and/or a `HEARTBEAT OUTPUT` command block. In the `RESULTS OUTPUT` command block, you would use a `NODAL` command line, an `ELEMENT` command line, or a `GLOBAL` command line, depending on how you defined the variable in the `USER VARIABLE` command block. Similarly, in the `HISTORY OUTPUT` or `HEARTBEAT OUTPUT` command block, you would use the applicable form of the variable command line, depending on how you defined the variable in the `USER VARIABLE` command block.

8.2.2.4 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.

8.2.2.5 Variable Transformation Command

```
TRANSFORM NODAL|ELEMENT VARIABLE <string>variable_name
  TO COORDINATE SYSTEM <string>coord_sys_name
  AS <string> transformed_name
```

The `TRANSFORM NODAL|ELEMENT VARIABLE` command line transforms a nodal vector (displacement, velocity, acceleration, etc) or an element tensor (stress, strain, etc.) from components in the global coordinate system to components in the coordinate system defined in a `BEGIN COORDINATE SYSTEM` command block having the name `coord_sys_name` (see Section 2.1.8). The transformed variables will be output to the results file as `transformed_name`.

Warning: This command cannot be used to transform shell element tensors which are not computed in the global coordinate system. Output of shell stress and strain components in a user-defined, local co-rotational coordinate system are obtained with the element variables `transform_shell_stress` and `transform_shell_strain` as described in Section 8.2.1.

8.2.2.6 Data Filtering Commands

```
BEGIN FILTER <string>filter_name
```

```
    ACOEFF = <real_list>a_coeff
    BCOEFF = <real_list>b_coeff
    INTERPOLATION TIME STEP = <real>ts
  END [FILTER]


  BEGIN USER OUTPUT
    FILTER <string>new_var FROM NODAL|ELEMENT <string>source_var
      USING <string>filter_name
  END
```

The user output `FILTER` command creates a new variable "new_var" by performing an on-the-fly frequency filter of the named element or nodal variable "source_var". The `BEGIN FILTER` command block defines a filter with the name "filter_name". The filter defined by the begin filter command block is then referenced by the `USER OUTPUT` filter command.

In the `BEGIN FILTER` command block the A and B filtering coefficients are defined with the `ACOEFF` and `BCOEFF` command lines. The filter must define at least one A and one B coefficient, there is no maximum number coefficients and the length of A and B do not need to match.

Filter operations assume that the data given at a constant time step. The explicit dynamics time step will tend to vary during the analysis. The `INTERPOLATION TIME STEP` command is used to linearly interpolate the data that is being produced at a non-constant time step down to some specified constant time step. The interpolation time step must be larger than zero and ideally should be specified such that it is smaller than the smallest time step with which the computations will iterate.

One way to obtain the filtering coefficients is with MATLAB. The following is an example of defining a third order Butterworth filter with a pass frequency of 100Hz at data interpolated to a time step of 1.0e-5 seconds. The filter is then used to filter acceleration histories of the nodes to 100Hz. The MATLAB code below will give the desired filtering coefficients. Note that the full 16-digit precision of the coefficients returned by MATLAB should be used. If truncated precision numbers are used, the filters can potentially be unstable.

```
clear;
format long e;
passFrequency = 100;
interp_ts = 1.0e-5;
butterCoeff = 2.0*interp_ts*passFrequency;
[bcoeff,acoeff] = butter(3,butterCoeff);
acoeff
bcoeff
```

The computed filtering coefficients can be used in a `USER OUTPUT` block as show below. If the analysis time step always remains above 1.0e-5 the filter will be valid. If the analysis time step drops below 1.0e-5 there could be aliasing issues, and a smaller interpolation time step should be specified.

```
BEGIN FILTER filt_100Hz
  ACOEFF = 1.000000000000000e+00 -2.987433650055722e+00 $
           2.974946132665442e+00 -9.875122361107358e-01
  BCOEFF = 3.081237301416628e-08  9.243711904249885e-08 $
           9.243711904249885e-08  3.081237301416628e-08
  INTERPOLATION TIME STEP = 1.0e-5
END
BEGIN USER OUTPUT
  FILTER ax100Hz FROM NODAL acceleration(x) USING filt_100Hz
  FILTER ay100Hz FROM NODAL acceleration(y) USING filt_100Hz
  FILTER az100Hz FROM NODAL acceleration(z) USING filt_100Hz
END
```

8.2.2.7 Compute at Every Step Command

```
  COMPUTE AT EVERY TIME STEP
```

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. (Section 10.2.4 discusses user-defined variables.)

8.2.2.8 Additional Command

The `ACTIVE PERIODS` or `INACTIVE PERIODS` command lines can appear as an option in the `USER OUTPUT` command block:

```
ACTIVE PERIODS = <string list>period_names

INACTIVE PERIODS = <string list>period_names
```

These command lines determine when the boundary condition is active. See Section 2.5 for more information about this optional command line.

8.3 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
  GLOBAL <string>variable_name
    [AS <string>history_variable_name]
  #
  # for mesh entity - node, edge, face,
  # element - variables
  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
  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
  SYNCHRONIZE_OUTPUT
  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.

Two metacharacters can appear in the name of the history file. If the %P character is found in the name, it will be replaced with the number processors being used for the run. For example, if you are running on 1024 processors and use the name history-%P/job.h, then the name would be expanded to history-1024/job.h. The other recognized metacharacter is %B which is replaced with the base name of the input file containing the input commands. For example, if the commands are in the file my_analysis_run.i and the history database name is specified as %B.h, then the history would be written to the file my_analysis_run.h.

If the history file does not use the Exodus II format [1], 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 [2] are supported in Presto and Adagio. 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 8.5.1.1. 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 8.5.1.2) 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 [3]) 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 8.3.1 through Section 8.3.12. Note that the command lines for controlling the frequency of history output (in Section 8.3.1 through Section 8.3.12) are the same as those for controlling the frequency of results output. These frequency-related command lines are repeated here for convenience.

8.3.1 Output Variables

The GLOBAL, NODE (or NODAL), EDGE, FACE, or ELEMENT command line is used to select variables for output in the history file. One of several types of variables can be selected for output. The form of the command line varies depending on the type of variable that is selected for output.

8.3.1.1 Global Output Variables

```
GLOBAL <string>variable_name
  [AS <string>history_variable_name]
```

This form of the 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 variable listed in Section 8.9 or a user-defined variable (see Section 8.2.2 and Section 10.2.4).

You can also specify a name, history_variable_name, for the selected entity following the AS keyword. For example, suppose you want to output the total number of iterations (total_iter) as ti. The command line to obtain the total number of iterationsin the history file would be

```
GLOBAL total_iter AS ti
```

The specification of an alias is optional for output of a global variable.

8.3.1.2 Mesh Entity Output Variables

```
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 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 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 command line, the mesh entity type 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 variable listed in Section 8.9 or a user-defined variable (see Section 8.2.2 and Section 10.2.4). If the mesh entity type is set to `ELEMENT`, the string `variable_name` can be a variable listed in Section 8.9 or a user-defined variable (see Section 8.2.2 and Section 10.2.4.

Selection of a specific mesh entity follows the `AT` keyword. 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 must specify a name, `history_variable_name`, for the selected entity following the `AS` keyword. 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:

```
NODE ACCELERATION AT NODE 88 AS accel_88
```

where `accel_88` is the name that will be used for this history variable in the history file.

Note that either the keyword `NODE` or `NODAL` can be used for nodal quantities.

As an example, the command line to obtain the von Mises stress for element 1024 for the history file would be:

```
ELEMENT VON_MISES AT ELEMENT 1024 AS vm_1024
```

where `vm_1024` is the name that will be used for this history variable in the history file.

8.3.1.3 Nearest Point Output Variables

```
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 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 8.3.1.2, for obtaining history output. The command line in this section or the command line in Section 8.3.1.2 produces history files with variable information. The difference in these two command lines (Section 8.3.1.3 and Section 8.3.1.2) is simply in how the variable information is selected.

For the above form of the command line, the mesh entity type 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 variable listed in Section 8.9 or a user-defined variable (see Section 8.2.2 and Section 10.2.4). If the mesh entity type is set to ELEMENT, the string variable_name can be a variable listed in Section 8.9 or a user-defined variable (see Section 8.2.2 and Section 10.2.4).

The specific mesh entity used for output is determined by global coordinates specified by the NEAREST LOCATION keyword 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 must specify a name, history_variable_name, for the selected entity following the AS keyword. 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:

```
NODE ACCELERATION
     NEAREST LOCATION 1012.0, 54.86, 103.3141 AS accel_near
```

where accel_near is the name that will be used for this history variable in the history file.

Note that either the keyword NODE or NODAL can be used for nodal quantities.

8.3.2 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
   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.

8.3.3 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.

8.3.4 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.

8.3.5 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`.

8.3.6 Additional Times for Output

```
 ADDITIONAL TIMES = <real>output_time1 <real>output_time2 ...
```

In addition to any times specified by the command line in Section 8.3.5, you can use the `ADDITIONAL TIMES` command line to specify an arbitrary number of additional output times.

8.3.7 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`.

8.3.8 Additional Steps for Output

```
 ADDITIONAL STEPS = <integer>output_step1
   <integer>output_step2 ...
```

In addition to any steps specified by the command line in Section 8.3.7, you can use the `ADDITIONAL STEPS` command line to specify an arbitrary number of g

8.3.9 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.

8.3.10 Synchronize Output

```
SYNCHRONIZE OUTPUT
```

In an analysis with multiple regions, it is sometimes desirable to synchronize the output of history data between the regions. This can be done by adding the `SYNCHRONIZE OUTPUT` command line to the history output block. If a history block has this set, then it will write output whenever a previous region writes output. The ordering of regions is based on the order in the input file, algorithmic considerations, or by solution control specifications.

Although the `USE OUTPUT SCHEDULER` command line can also synchronize output between regions, the `SYNCHRONIZE OUTPUT` will synchronize the output with regions where the output frequency is not under the direct control of the Sierra IO system. Examples of this are typically coupled applications where one or more of the codes are not Sierra-based applications such as Alegra and CTH. A history block with `SYNCHRONIZE OUTPUT` specified will also synchronize its output with the output of the external code.

The `SYNCHRONIZE OUTPUT` command can be used with other output scheduling commands such as time-based or step-based output specifications.

8.3.11 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 SIERRA scope. 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 8.6 for a description of using this command block and the `USE OUTPUT SCHEDULER` command line.

8.3.12 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.

8.4 Heartbeat Output

```
BEGIN HEARTBEAT OUTPUT <string>heartbeat_name
  # Can also use predefined streams "cout", "stdout",
  # "cerr", "clog", "log", "output", or "outputP0"
  STREAM NAME = <string>heartbeat_file_name
  #
  # Specify whether heartbeat file will be in spyhis (cth)
  # format, or default format
  FORMAT = SPYHIS|DEFAULT
  #
  # for global variables
  GLOBAL <string>variable_name
    [AS <string>heartbeat_variable_name]
  #
  # for mesh entity - node, edge, face,
  # element - variables
  NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
    AT NODE|NODAL|EDGE|FACE|ELEMENT <integer>entity_id
    AS <string>heartbeat_variable_name
  #
  # for nearest point output of mesh entity - node,
  # edge, face, element - variables
  NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
    NEAREST LOCATION <real>global_x,
      <real>global_y>, <real>global_z
    AS <string>heartbeat_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
  SYNCHRONIZE_OUTPUT
  USE OUTPUT SCHEDULER <string>scheduler_name
  OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
    SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
    SIGKILL|SIGILL|SIGSEGV
  PRECISION = <integer>precision
  LABELS = <string>OFF|ON
  LEGEND = <string>OFF|ON
  TIMESTAMP FORMAT <string>timestamp_format
```

```
    MONITOR = <string>RESULTS|RESTART|HISTORY
  END [HEARTBEAT OUTPUT <string>heartbeat_name]
```

The heartbeat output is text output file that 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.

Known Issue: User defined variables (see Section 10.2.4) are not currently supported with heartbeat output.

The output is written as text instead of the binary history output. You can specify a heartbeat file, the results to be included in this file, the formatting of the output, and the frequency at which results are written by using a `HEARTBEAT OUTPUT` command block. The command block appears inside the region scope. For heartbeat output, you will typically work with global, node, and element variables, and, on some occasions, face variables.

More than one heartbeat file can be specified for an analysis. For each heartbeat file, there will be one `HEARTBEAT OUTPUT` command block. The command block for a heartbeat file description begins with

```
BEGIN HEARTBEAT OUTPUT <string>heartbeat_name
```

and is terminated with

```
END [HEARTBEAT OUTPUT <string>heartbeat_name]
```

where `heartbeat_name` is a user-selected name for the command block. Nested within the `HEARTBEAT OUTPUT` command block are a set of command lines, as shown in the block summary given above. The first command line listed (`STREAM NAME`) gives pertinent information about the heartbeat file. The command line

```
STREAM NAME = <string>heartbeat_file_name
```

gives the name of the heartbeat file with the string `heartbeat_file_name`. If the file already exists, it is overwritten. If the heartbeat file is to appear in the current directory and is named `job.h`, this command line would appear as

```
STREAM NAME = job.h
```

If the heartbeat file is to be created in some other directory, the command line would have to show the absolute path to that directory.

In addition to specifying a specific filename, there are several predefined streams that can be specified. The predefined streams are:

- ’cout’ or ’stdout’ specifies standard output;
- ’cerr’, ’stderr’, ’clog’, or ’log’ specifies standard error;
- ’output’ or ’outputP0’ specifies Sierra’s standard output which is redirected to the file specified by the ’-o’ option on the command line.

Two metacharacters can appear in the name of the heartbeat file. If the `%P` character is found in the name, it will be replaced with the number processors being used for the run. For example, if you are running on 1024 processors and use the name `heartbeat-%P/job.h`, then the name would be expanded to `heartbeat-1024/job.h`. The other recognized metacharacter is `%B` which is replaced with the base name of the input file containing the input commands. For example, if the commands are in the file `my_analysis_run.i` and the heartbeat stream name is specified as `%B.h`, then the heartbeat data would be written to the file `my_analysis_run.h`.

The other command lines that appear in the `HEARTBEAT OUTPUT` command block determine the type, frequency, and format of information that is output. Descriptions of these command lines follow in Section 8.4.1 through Section 8.4.14. Note that the command lines for controlling the frequency of heartbeat output (in Section 8.4.3 through Section 8.4.12) are the same as those for controlling the frequency of results and history output. These frequency-related command lines are repeated here for convenience.

8.4.1 Output Variables

The `GLOBAL`, `NODE` (or `NODAL`), `EDGE`, `FACE`, or `ELEMENT` command line is used to select variables for output in the heartbeat file. One of several types of variables can be selected for output. The form of the command line varies depending on the type of variable that is selected for output.

8.4.1.1 Global Output Variables

```
GLOBAL <string>variable_name
  [AS <string>heartbeat_variable_name]
```

This form of the command lets you select any global variable for output in the heartbeat 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 variable listed in Section 8.9 or a user-defined variable (see Section 8.2.2 and Section 10.2.4). The `variable_name` can also specify `time`, `timestep`, or `step` to output the current simulation time, timestep, or execution step, respectively.

You can also specify a name, `heartbeat_variable_name`, for the selected entity following the `AS` keyword. For example, suppose you want to output the total number of iterations (total_iter) as ti. The command line to obtain the total number of iterationsin the heartbeat file would be:

```
GLOBAL total_iter AS ti
```

The specification of an alias is optional for a global variable.

8.4.1.2 Mesh Entity Output Variables

```
    NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
      AT NODE|NODAL|EDGE|FACE|ELEMENT <integer>entity_id
      AS <string>heartbeat_variable_name
```

This form of the command lets you select any nodal, edge, face, or element variable for a specific mesh entity for output in the heartbeat file. For example, this command line will let you pick the displacement at a specific node and output the displacement to the heartbeat file using an alias that you have chosen.

For this form of the command, the mesh entity type 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 variable listed in Section 8.9 or a user-defined variable (see Section 8.2.2 and Section 10.2.4). If the mesh entity type is set to ELEMENT, the string variable_name can be a variable listed in Section 8.9 or a user-defined variable (see Section 8.2.2 and Section 10.2.4).

Selection of a specific mesh entity follows the AT keyword. 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 must specify a name, heartbeat_variable_name, for the selected entity following the AS keyword. For example, suppose you want to output the accelerations at node 88. The command line to obtain the accelerations at node 88 for the heartbeat file would be:

```
NODE ACCELERATION AT NODE 88 AS accel_88
```

where accel_88 is the name that will be used for this heartbeat variable in the heartbeat file.

Note that either the keyword NODE or NODAL can be used for nodal quantities.

As an example, the command line to obtain the von Mises stress for element 1024 for the heartbeat file would be:

```
ELEMENT VON_MISES AT ELEMENT 1024 AS vm_1024
```

where vm_1024 is the name that will be used for this heartbeat variable in the heartbeat file.

8.4.1.3 Nearest Point Output Variables

```
    NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
      NEAREST LOCATION <real>global_x,
        real<global_y>, real<global_z>
      AS <string>heartbeat_variable_name
```

This form of the command lets you select any nodal, edge, face, or element variable for output in the heartbeat 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 8.4.1.2, for obtaining heartbeat output. The command line in this section or the command line in Section 8.4.1.2 produces heartbeat files with variable information. The difference in these two command lines (Section 8.4.1.3 and Section 8.4.1.2) is simply in how the variable information is selected.

For the above form of the command, the mesh entity type 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 variable listed in Section 8.9 or a user-defined variable (see Section 8.2.2 and Section 10.2.4). If the mesh entity type is set to `ELEMENT`, the string `variable_name` can be a variable listed in Section 8.9 or a user-defined variable (see Section 8.2.2 and Section 10.2.4).

The specific mesh entity used for output is determined by global coordinates specified by the `NEAREST LOCATION` keyword 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 must specify a name, `heartbeat_variable_name`, for the selected entity following the `AS` keyword. 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 heartbeat file would be:

```
NODE ACCELERATION
     NEAREST LOCATION 1012.0, 54.86, 103.3141 AS accel_near
```

where `accel_near` is the name that will be used for this heartbeat variable in the heartbeat file.

Note that either the keyword `NODE` or `NODAL` can be used for nodal quantities.

8.4.2 Outputting Heartbeat Data on a Node Set

It is commonly desired to output heartbeat 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 heartbeat 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 heartbeat output
  global disp_ns_1
end
```

If `nodelist_1` contains only a single node, the heartbeat 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.

8.4.3 Set Begin Time for Heartbeat Output

```
START TIME = <real>output_start_time
```

Using the `START TIME` command line, you can write heartbeat variables to the heartbeat file beginning at time `output_start_time`. No heartbeat variables will be written before this time. If other commands set times for heartbeat output (`AT TIME`, `ADDITIONAL TIMES`) that are less than `output_start_time`, those times will be ignored, and heartbeat output will not be written at those times.

8.4.4 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, heartbeat variables are output at times closest to the specified output times.

8.4.5 Output Interval Specified by Time Increment

```
AT TIME <real>time_begin INCREMENT = <real>time_increment_dt
```

At the time specified by `time_begin`, heartbeat variables will be output every time increment given by the real value `time_increment_dt`.

8.4.6 Additional Times for Output

```
 ADDITIONAL TIMES = <real>output_time1 <real>output_time2 ...
```

In addition to any times specified by the command line in Section 8.4.5, you can use the `ADDITIONAL TIMES` command line to specify an arbitrary number of additional output times.

8.4.7 Output Interval Specified by Step Increment

```
AT STEP <integer>step_begin INCREMENT =
  <integer>step_increment
```

At the step specified by `step_begin`, heartbeat variables will be output every step increment given by the integer value `step_increment`.

8.4.8 Additional Steps for Output

```
 ADDITIONAL STEPS = <integer>output_step1
   <integer>output_step2 ...
```

In addition to any steps specified by the command line in Section 8.3.7, you can use the `ADDITIONAL STEPS` command line to specify an arbitrary number of g

8.4.9 Set End Time for Heartbeat Output

```
TERMINATION TIME = <real>termination_time_value
```

Heartbeat output will not be written to the heartbeat file after time `termination_time_value`. If other commands set times for heartbeat output (`AT TIME`, `ADDITIONAL TIMES`) that are greater than `termination_time_value`, those times will be ignored, and heartbeat output will not be written at those times.

8.4.10 Synchronize Output

```
SYNCHRONIZE OUTPUT
```

In an analysis with multiple regions, it is sometimes desirable to synchronize the output of heartbeat data between the regions. This can be done by adding the `SYNCHRONIZE OUTPUT` command line to the heartbeat output block. If a heartbeat block has this set, then it will write output whenever a previous region writes output. The ordering of regions is based on the order in the input file, algorithmic considerations, or by solution control specifications.

Although the `USE OUTPUT SCHEDULER` command line can also synchronize output between regions, the `SYNCHRONIZE OUTPUT` will synchronize the output with regions where the output frequency is not under the direct control of the Sierra IO system. Examples of this are typically coupled applications where one or more of the codes are not Sierra-based applications such as Alegra and CTH. A heartbeat block with `SYNCHRONIZE OUTPUT` specified will also synchronize its output with the output of the external code.

The `SYNCHRONIZE OUTPUT` command can be used with other output scheduling commands such as time-based or step-based output specifications.

8.4.11 Use Output Scheduler

```
USE OUTPUT SCHEDULER <string>scheduler_name
```

In an analysis with multiple regions, it can be difficult to synchronize output such as heartbeat files. To help synchronize output for analyses with multiple regions, you can define an `OUTPUT SCHEDULER` command block at the SIERRA scope. The scheduler can then be referenced in the `HEARTBEAT 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 8.6 for a description of using this command block and the `USE OUTPUT SCHEDULER` command line.

8.4.12 Write Heartbeat On Signal

```
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 heartbeat file when the system encounters the specified signal. The signal can either occur as the result of a system error, or the user can explicitly send the specified signal to the application (See the system documentation man pages for “signal” or “kill” for more information). Only one signal type in the list of signal types should be entered for this command line. Generally, these signals cause the code to terminate before the code can add any current heartbeat output (heartbeat output past the last heartbeat output time step) to the heartbeat file. If the code encounters the specified type of error during execution, a heartbeat file will be written before execution is terminated.

This command line can also be used to force the writing of a heartbeat 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 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.

8.4.13 Heartbeat Output Formatting Commands

There are several command lines for the heartbeat section that modify the formatting of the heartbeat text output. The default output for the heartbeat data consists of a line beginning with a timestamp showing the current wall-clock time followed by multiple columns of data, for example:

```
Begin HeartBeat Region_1_Heartbeat
  Stream Name = output
  At Step 0, Increment = 10

  precision is 5

  global step
  global timestep as dt
  global time
  global total_energy as te
End
```

```
+[12:18:51] step=240, dt=3.13933e-04, time=7.56578e-02, te=4.02795e-06
+[12:18:51] step=250, dt=3.13933e-04, time=7.87971e-02, te=1.32125e-06
+[12:18:51] step=260, dt=3.13933e-04, time=8.19365e-02, te=6.88142e-07
+[12:18:51] step=270, dt=3.13933e-04, time=8.50758e-02, te=3.93574e-06
+[12:18:52] step=280, dt=3.13933e-04, time=8.82151e-02, te=7.46609e-06
+[12:18:52] step=290, dt=3.13933e-04, time=9.13545e-02, te=1.03856e-05
+[12:18:52] step=300, dt=3.13933e-04, time=9.44938e-02, te=1.36822e-05
+[12:18:52] step=310, dt=3.13933e-04, time=9.76331e-02, te=1.64630e-05
```

The above example begins each line with a timestamp followed by five labeled data columns. The precision of the real data is 5. There is no legend in the above example. This format can be modified with the following commands.

8.4.13.1 CTH SpyHis output format

```
FORMAT = SPYHIS|DEFAULT
```

If the `FORMAT=SPYHIS` is specified, then the heartbeat output will be formatted such that it can be processed with the CTH spyhis application which is a post-processor for time-history data.

8.4.13.2 Specify floating point precision

```
PRECISION = <integer>precision
```

By default, the real data is written with a precision of 5 which gives 5 digits following the decimal point. This can be altered with the `PRECISION` command. If the command line `PRECISION = 2` is specified, then the above data would look like:

```
Begin HeartBeat Region_1_Heartbeat
  ...
  precision = 2
  ...
End

+[12:18:51] step=240, dt=3.14e-04, time=7.57e-02, te=4.03e-06
+[12:18:51] step=250, dt=3.14e-04, time=7.88e-02, te=1.32e-06
+[12:18:51] step=260, dt=3.14e-04, time=8.19e-02, te=6.88e-07
```

Note that the precision applies to all real data; it is not possible to specify a different precision for each variable.

8.4.13.3 Specify Labeling of Heartbeat Data

```
LABELS = <string>OFF|ON
```

The above example shows the default output which consists of a label and the data separated by "=". The existence of the labels is controlled with the `LABELS` command. If `LABELS = OFF` is specified, then the above data would look like:

```
Begin HeartBeat Region_1_Heartbeat
  ...
  labels = off
  precision = 2
  ...
End

+[12:17:37] 240, 3.14e-04, 7.57e-02, 4.03e-06
+[12:17:37] 250, 3.14e-04, 7.88e-02, 1.32e-06
+[12:17:38] 260, 3.14e-04, 8.19e-02, 6.88e-07
```

8.4.13.4 Specify Existence of Legend for Heartbeat Data

```
LEGEND = <string>OFF|ON
```

Outputting the data without labels can make it easier to work with the data in a spreadsheet program or other data manipulation program, but with no labels, it is difficult to determine what the data really represents. The `LEGEND` output will print a line at the beginning of the heartbeat output identifying the data in each column. For example:

```
    Begin HeartBeat Region_1_Heartbeat
      ...
      legend = on
      labels = off
      precision = 2
      ...
    End

+[12:17:37] Legend: step, dt, time, te
+[12:17:37] 240, 3.14e-04, 7.57e-02, 4.03e-06
+[12:17:37] 250, 3.14e-04, 7.88e-02, 1.32e-06
+[12:17:38] 260, 3.14e-04, 8.19e-02, 6.88e-07
```

8.4.13.5 Specify format of timestamp

```
TIMESTAMP FORMAT <string>"timestamp_format"
```

Each line of the heartbeat output is preceded by a timestamp which shows the wall-clock time at the time that the line was output. This can be useful to verify that the code is still running and producing output and to determine how fast the code is running. The default timestamp is in the format “[12:34:56]” which is specified by the format ̈[%H:%M:%S]̈. If a different format is desired, it can be specified with the `TIMESTAMP FORMAT` command line. The format must be surrounded by double or single quotes and the format is defined to be the string between the first single or double quote and the last matching quote type. If you want to modify the format, see the documentation for the UNIX `strftime` command for details on how to specify the format. The example below shows a timestamp format delimited by “`{`” and “`}`”. The timestamp consists of a ISO-8601 date format followed by the current time.

```
...
    timestamp format "{%F %H:%M:%S}"
...
+{2008-03-17 09:26:17} 2212, 1.34244e-06, 2.96948e-03, 2.96948e-03
+{2008-03-17 09:26:17} 2213, 1.34244e-06, 2.97082e-03, 2.97082e-03
+{2008-03-17 09:26:17} 2214, 1.34244e-06, 2.97216e-03, 2.97216e-03
+{2008-03-17 09:26:17} 2215, 1.34244e-06, 2.97350e-03, 2.97350e-03
+{2008-03-17 09:26:17} 2216, 1.34244e-06, 2.97485e-03, 2.97485e-03
```

8.4.14 Monitor Output Events

```
MONITOR = <string>RESULTS|RESTART|HISTORY
```

It is sometimes a benefit to know when the code has written a new set of data to one of the other output files (restart output, history output, or results output). The heartbeat output will report this data if the `MONITOR` command line is specified. Each time output is performed to any of the monitored output types, a line will be written to the heartbeat file specifying the timestamp, the simulation time and step, and the label name of the output type. For example:

```
begin results output my_results
  at step 0, increment = 10
...
end results output results

begin heartbeat data hb
  stream name = stdout
  monitor = results
  labels = off
  legend = on
  timestamp format "%F %H:%M:%S "
  at step 0, increment = 2
  global step
  global timestep as dt
  global time
  element spring_engineering_strain at \#
      element 1 as sp1
end
```

Will give the following output:

```
....
+2008-03-17 10:03:22  718, 1.34244e-06, 9.63871e-04, 9.63871e-04
-2008-03-17 10:03:22  Results data written at time = 0.00096656,
step = 720. my_results
+2008-03-17 10:03:22  720, 1.34244e-06, 9.66556e-04, 9.66556e-04
+2008-03-17 10:03:22  722, 1.34244e-06, 9.69241e-04, 9.69241e-04
+2008-03-17 10:03:22  724, 1.34244e-06, 9.71926e-04, 9.71926e-04
+2008-03-17 10:03:22  726, 1.34244e-06, 9.74611e-04, 9.74611e-04
+2008-03-17 10:03:22  728, 1.34244e-06, 9.77296e-04, 9.77296e-04
-2008-03-17 10:03:22  Results data written at time = 0.00097998,
step = 730. my_results
+2008-03-17 10:03:22  730, 1.34244e-06, 9.79981e-04, 9.79981e-04
....
```

8.5 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
  SYNCHRONIZE_OUTPUT
  USE OUTPUT SCHEDULER <string>scheduler_name
  OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
    SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
    SIGKILL|SIGILL|SIGSEGV
  OPTIONAL
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 SIERRA 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 8.5.1, you will learn how to use the `DATABASE NAME`, `INPUT DATABASE NAME`, `OUTPUT DATABASE NAME`, `DATABASE TYPE`, and `OPTIONAL` 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 SIERRA scope.

Section 8.5.2 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 8.5.3 through Section 8.5.14. 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.

8.5.1 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)
OPTIONAL
```

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
```

To read or create files in some other directory, the command line must include the path to that directory. The directory must exist, it will not be created.

The `DATABASE NAME` command line will let you read restart information and write restart information to the same file. Section 8.5.1.1 through Section 8.5.1.4 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.

Two metacharacters can appear in the name of the restart file. If the `%P` character is found in the name, it will be replaced with the number processors being used for the run. For example, if you are running on 1024 processors and use the name `restart-%P/job.rs`, then the name would be expanded to `restart-1024/job.rs` and the actual restart files would be `restart-1024/job.rs.1024.0000` to `restart-1024/job.rs.1024.1023`. The other recognized metacharacter is `%B` which is replaced with the base name of the input file containing the input commands. For example, if the commands are in the file `my_analysis_run.i` and the restart database name is specified as `%B.rs`, then the restart data would be written to or read from the file `my_analysis_run.rs`.

If the restart file does not use the Exodus II format [1], 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 [2] are supported in Presto and Adagio. Exodus II is more commonly used than XDMF. Other options may be added in the future.

In certain coupled physics analyses in which there are multiple regions, only a subset of the regions may have a restart database associated with them. The `OPTIONAL` command (Section 8.5.1) is used to tell the application that it is acceptable to restart the analysis even though a region does not have an associated restart database. Note that this is only allowed in analyses containing multiple regions; if there is only a single region, it must have a restart database in order to restart.

8.5.1.1 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 SIERRA 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 SIERRA scope of our input file. In a `TIME STEPPING` command block, which is embedded in a `TIME CONTROL` command block (Section 3.11.1) 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 `START TIME` and `TERMINATION TIME` command lines that appear 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 RESTART_DATA
  DATABASE NAME = g.rsout
  AT TIME 0.0 INCREMENT = 0.25E-3
END RESTART DATA RESTART_DATA
```

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 8.5.5).

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.

8.5.1.2 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 SIERRA 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 SIERRA 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 RESTART_DATA
  OUTPUT DATABASE NAME = RS1.rsout
  AT TIME 0.0 INCREMENT = 0.5E-3
END RESTART DATA RESTART_DATA
```

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 SIERRA scope and set the termination time to 5.0×10^{-3} by using the command line

```
TERMINATION TIME = 5.0E-3 \rm
```

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 RESTART_DATA
  INPUT DATABASE NAME = RS1.rsout
  OUTPUT DATABASE NAME = RS2.rsout
  AT TIME 0.0 INCREMENT = 0.5E-3
END RESTART DATA RESTART_DATA
```

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 RESTART_DATA
  INPUT DATABASE NAME = RS2.rsout
  OUTPUT DATABASE NAME = RS3.rsout
  AT TIME 0.0, INCREMENT = 0.5E-3
END RESTART DATA RESTART_DATA
```

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.

8.5.1.3 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 SIERRA 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}.

8.5.1.4 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 8.5.1.2). 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.

8.5.2 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 8.5.1.1 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 8.5.1.2), 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.

8.5.3 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.

8.5.4 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.

8.5.5 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`.

8.5.6 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 8.5.5, you can use the `ADDITIONAL TIMES` command line to specify an arbitrary number of additional restart times.

8.5.7 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`.

8.5.8 Additional Steps for Restart

```
ADDITIONAL STEPS = <integer>output_step1
  <integer>output_step2 ...
```

In addition to any steps specified by the command line in Section 8.5.7, you can use the `ADDITIONAL STEPS` command line to specify an arbitrary number of additional restart steps.

8.5.9 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.

8.5.10 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 8.5.11.

8.5.11 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 8.5.10 with our example of a cycle count of 5 as given in this section (Section 8.5.11). 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.

8.5.12 Synchronize Output

```
SYNCHRONIZE OUTPUT
```

In an analysis with multiple regions, it is sometimes desirable to synchronize the output of restart data between the regions. This can be done by adding the `SYNCHRONIZE OUTPUT` command line to the restart output block. If a restart block has this set, then it will write output whenever a previous region writes output. The ordering of regions is based on the order in the input file, algorithmic considerations, or by solution control specifications.

Although the `USE OUTPUT SCHEDULER` command line can also synchronize output between regions, the `SYNCHRONIZE OUTPUT` will synchronize the output with regions where the output frequency is not under the direct control of the Sierra IO system. Examples of this are typically coupled applications where one or more of the codes are not Sierra-based applications such as Alegra and CTH. A restart block with `SYNCHRONIZE OUTPUT` specified will also synchronize its output with the output of the external code.

The `SYNCHRONIZE OUTPUT` command can be used with other output scheduling commands such as time-based or step-based output specifications.

8.5.13 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 SIERRA scope. 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 8.6 for a description of using this command block and the `USE OUTPUT SCHEDULER` command line.

8.5.14 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.

8.6 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 SIERRA scope. 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 SIERRA 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 8.6.1 describes the `OUTPUT SCHEDULER` command block, and Section 8.6.2 illustrates how this block is referenced with the `USE OUTPUT SCHEDULER` command line.

8.6.1 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 SIERRA 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.

8.6.1.1 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.

8.6.1.2 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.

8.6.1.3 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.

8.6.1.4 Additional Times for Output

```
 ADDITIONAL TIMES = <real>output_time1 <real>output_time2 ...
```

In addition to any times specified by the command line in Section 8.6.1.3, you can use the ADDITIONAL TIMES command line to specify an arbitrary number of additional output times.

8.6.1.5 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`.

8.6.1.6 Additional Steps for Output

```
 ADDITIONAL STEPS = <integer>output_step1
   <integer>output_step2 ...
```

In addition to any steps specified by the command line in Section 8.6.1.5, you can use the `ADDITIONAL STEPS` command line to specify an arbitrary number of additional output steps.

8.6.1.7 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.

8.6.2 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 SIERRA 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
```

8.7 Variable Interpolation

```
BEGIN VARIABLE INTERPOLATION [<string>var_interp_name]
  TRANSFER TYPE = INTERPOLATE FROM NEAREST FACE|
    SUM TO NEAREST ELEMENT
  SOURCE VARIABLE = ELEMENT|GLOBAL|NODAL|
    SURFACE NORMAL NODAL <string>source_var_name
  SOURCE ELEMENT BLOCK = <string>source_block
  SOURCE SURFACE = <string>source_surface
  TARGET VARIABLE = ELEMENT|GLOBAL|NODAL
    <string>target_var_name
  TARGET SURFACE = <string>target_surface
  SEARCH TOLERANCE = <real>search_tol
  PROXIMITY SEARCH TYPE = RAY SEARCH|SPHERE SEARCH
  RAY SEARCH DIRECTION = <real>vecx <real>vecy <real>vecz
  RAY SEARCH DIRECTION = ORTHOGONAL TO LINE <real>p1x
    <real>p1y <real>p1z <real>p2x <real>p2y <real>p2z
  RAY SEARCH DIRECTION = TARGET SURFACE NORMAL

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

END [VARIABLE INTERPOLATION <string>var_interp_name]
```

The command block for a variable interpolation begins with `BEGIN VARIABLE INTERPOLATION [<string>var_interp_name]` and is terminated with `END [VARIABLE INTERPOLATION <string>var_interp_name]` where `var_interp_name` is an optional user-selected name for the command block.

The command `TRANSFER TYPE` can be set to either `INTERPOLATE FROM NEAREST FACE` or `SUM TO NEAREST ELEMENT`. The `TRANSFER TYPE` restricts the commands that can be used in conjunction with it.

When `TRANSFER TYPE` is set to `INTERPOLATE FROM NEAREST FACE` the available commands are restricted to:

```
BEGIN VARIABLE INTERPOLATION [<string>var_interp_name]
  TRANSFER TYPE = INTERPOLATE FROM NEAREST FACE
  SOURCE VARIABLE = NODAL|SURFACE NORMAL NODAL
    <string>source_var_name
  SOURCE SURFACE = <string>source_surface
  TARGET VARIABLE = NODAL <string>target_var_name
  TARGET SURFACE = <string>target_surface
  SEARCH TOLERANCE = <real>search_tol
  PROXIMITY SEARCH TYPE = RAY SEARCH|SPHERE SEARCH
  RAY SEARCH DIRECTION = <real>vecx <real>vecy <real>vecz
  RAY SEARCH DIRECTION = ORTHOGONAL TO LINE <real>p1x
```

```
      <real>p1y <real>p1z <real>p2x <real>p2y <real>p2z
    RAY SEARCH DIRECTION = TARGET SURFACE NORMAL
  END [VARIABLE INTERPOLATION <string>var_interp_name]
```

For TRANSFER TYPE = INTERPOLATE FROM NEAREST FACE, a given point on the target surface finds the closest point on the source surface. At that closet point the source variable is interpolated. The interpolated value is then copied to the given point. Optionally if the SURFACE NORMAL NODAL component is being used and the variable being transferred is a vector then only the surface normal component of that variable will be transferred. This is useful for several applications involving transferring solid mesh quantities such as velocity to a reference mesh modeling a fluid (e.g., air).

When TRANSFER TYPE is set to SUM TO NEAREST ELEMENT the available commands are restricted to:

```
  BEGIN VARIABLE INTERPOLATION [<string>var_interp_name]
    TRANSFER TYPE = SUM TO NEAREST ELEMENT
    SOURCE VARIABLE = ELEMENT <string>source_var_name
    SOURCE ELEMENT BLOCK = <string>source_block
    TARGET VARIABLE = ELEMENT <string>target_var_name
    TARGET SURFACE = <string>target_surface
    SEARCH TOLERANCE = <real>search_tol
    PROXIMITY SEARCH TYPE = RAY SEARCH|SPHERE SEARCH
    RAY SEARCH DIRECTION = <real>vecx <real>vecy <real>vecz
    RAY SEARCH DIRECTION = ORTHOGONAL TO LINE <real>p1x
      <real>p1y <real>p1z <real>p2x <real>p2y <real>p2z
  END [VARIABLE INTERPOLATION <string>var_interp_name]
```

For TRANSFER TYPE = SUM TO NEAREST ELEMENT, a given element in the source block computes its centroid. Then the closest face to that centroid on the target surface is found. The source element value is summed to the element associated with the target face.

The command SOURCE VARIABLE specifies the type of variable, ELEMENT, GLOBAL, or NODAL, and the name of the source variable source_var_name.

The command SOURCE SURFACE specifies the name of the source side set source_surface.

The command TARGET VARIABLE specifies the type of variable, ELEMENT, GLOBAL, or NODAL, and the name of the target variable target_var_name. The target variable must exist. If necessary, a target variable can be created using the BEGIN USER VARIABLE command block (Section 10.2.4).

The command TARGET SURFACE specifies the name of the target side set target_surface.

The command SEARCH TOLERANCE specifies a search distance for use in the proximity search.

The command PROXIMITY SEARCH TYPE can be set to either RAY SEARCH or SPHERE SEARCH. When PROXIMITY SEARCH TYPE = SPHERE SEARCH the SEARCH TOLERANCE is used as the radius for closest point searches. When PROXIMITY SEARCH TYPE = RAY SEARCH the

`SEARCH TOLERANCE` and any `RAY SEARCH DIRECTION` are used together to form rays with lengths equal to the `SEARCH TOLERANCE` in both the positive and negative directions from an associated `RAY SEARCH DIRECTION`. Faces penetrated by the ray get included in the set of faces for determining the closest point.

The command `RAY SEARCH DIRECTION` specifies a search direction for the `PROXIMITY SEARCH = RAY SEARCH DIRECTION` case. A `RAY SEARCH DIRECTION` can be specified directly with three values, `vecx`, `vecy`, and `vecz`. Alternatively, `RAY SEARCH DIRECTION` can be calculated through options `ORTHOGONAL TO LINE` or `TARGET SURFACE NORMAL`. For `ORTHOGONAL TO LINE` the search direction is the vector between a given point and the closest point on the infinite line specified with the two points `p1x, p1y, p1z` and `p2x, p2y, p2z`. For `TARGET SURFACE NORMAL` the search direction is the nodal normal vector calculated from the `TARGET SURFACE`.

8.8 Global Output Options

The following commands exist at the region scope to control the output of global variables:

```
GLOBAL ENERGY REPORTING = EXACT|APPROXIMATE|OFF (EXACT)
EXTENSIVE RIGID BODY VARS OUTPUT = OFF|HISTORY|RESULTS|ALL (ALL)
```

Through the `GLOBAL ENERGY REPORTING` command line Adagio offers three reporting options for global energy variables: `EXACT`, `APPROXIMATE`, and `OFF`. The `EXACT` and `APPROXIMATE` reporting options use different algorithms for tracking the global values of external energy, internal energy, contact energy, and hourglass energy. In many cases, the `APPROXIMATE` reporting option will provide a modest performance improvement with a negligible effect on the reported energy values. The `OFF` option will result in a further performance improvement and will report energy values of zero.

Note that the `GLOBAL ENERGY REPORTING` command has no effect on the analysis itself; the energy values calculated are used only for reporting purposes.

The line command `EXTENSIVE RIGID BODY VARS OUTPUT` controls the default output of global rigid body variables. Regardless of the option choice here, global rigid body variables may be output by name in the history or results output blocks. See Table 8.2 for a list of available variables. The `EXTENSIVE RIGID BODY VARS OUTPUT` options are: `OFF` to specify no default rigid body global variable output; `HISTORY` to specify default rigid body global variable output to the history file(s) only; `RESULTS` to specify default rigid body global variable output to the results file(s) only; `ALL` to specify default rigid body global variable output to both the history and results files. This option defaults to `ALL` so that if this command is not specified both the history and results files will contain the variables listed in Table 8.2 at every output time.

8.9 Variables

This section lists commonly used 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 variables. The second part of this section lists variables associated with material models.

8.9.1 Global, Nodal, Face, and Element Variables

This section lists commonly used global, nodal, and element variables. The variables are presented in tables based on use, as follows:

- Table 8.1 Global Variables for All Analyses
- Table 8.2 Global Variables for Rigid Bodies
- Table 8.3 Global Variables for *J*-Integral
- Table 8.4 Nodal Variables for All Analyses
- Table 8.5 Nodal Variables for Shells
- Table 8.7 Nodal Variables for Contact
- Table 8.8 Nodal Variables for *J*-Integral
- Table 8.9 Face Variables for Blast Pressure Boundary Condition
- Table 8.10 Element Variables for All Elements
- Table 8.11 Element Variables for Solid Elements
- Table 8.12 Element Variables for Membranes
- Table 8.13 Element Variables for Shells
- Table 8.14 Element Variables for Trusses
- Table 8.15 Element Variables for Cohesive Elements
- Table 8.16 Element Variables for *J*-Integral

The tables provide the following information about each variable:

Variable Name. This is the string that will appear on the `GLOBAL`, `NODE`, `FACE`, or `ELEMENT` command line.

Type. This is the variable's type. The various types are denoted with the labels `Integer`, `Integer[]`, `Real`, `Real[]`, `Vector_2D`, `Vector_3D`, `SymTen33`, and `FullTen36`. The type `Integer` indicates the variable is an integer; the type `Integer[]` is an integer array; the type

Real indicates the variable is a real; the type Real[] is a real array. The type Vector_2D indicates the variable type is a two-dimensional vector. The type Vector_3D indicates the 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 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 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 variable 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.

Derived. Any variable designated with a yes in this column must be included in a BEGIN DERIVED OUTPUT command block if it is to be transferred to another procedure or region as described in Section 5.6.

For multi-integration point elements, quantities from the element integration points will be appended with a numerical suffix indicating the integration point. A suffix ranging from 1 to the number of integration points is attached to the 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 von_mises is requested for a shell element with 15 integration points, then the quantities von_mises_01, von_mises_02, ..., von_mises_15 are output for the shell element.

The tables of various types of variables follow.

Table 8.1: Global Variables For All Analyses

Variable Name	**Type**	**Comments**
artificial_energy	Real	
contact_energy	Real	(A component of external energy)
external_energy	Real	
ke_blockblockID	Real	Kinetic energy sum for block blockID
ee_strain_blockblockID	Real	External energy sum for block blockID
ie_strain_blockblockID	Real	Internal energy sum for block blockID
momentum_blockblockID	Vector_3D	Momentum sum for block blockID
hourglass_energy	Real	(A component of internal energy)
hge_blockblockID	Real	Hourglass energy sum for block blockID
internal_energy	Real	
kinetic_energy	Real	
momentum	Vector_3D	Momentum vector
timestep	Real	Current time step
timestep_element	Real	Time step from element estimator
timestep_nodal	Real	Time step from nodal estimator
timestep_material	Real	Time step from material model
timestep_lanczos	Real	Time step from Lanczos estimator
timestep_powermethod	Real	Time step from power method estimator
wall_clock_time	Real	Accumulated wall clock time
wall_clock_time_per_ step	Real	Wall clock time for last time step
cpu_time	Real	Accumulated CPU time
cpu_time_per_step	Real	CPU time for last time step

Table 8.2: Global Variables for Rigid Bodies. (See Section 8.8 for default output options.)

Variable Name	**Type**	**Comments**
ax,ay,az	Real	Translational acceleration
velx,vely,velz	Real	Translational velocity
displx,disply,displz	Real	Translational displacement
rotax,rotay,rotaz	Real	Rotational acceleration
rotvx,rotvy,rotvz	Real	Rotational velocity
rotdx,rotdy,rotdz	Real	Rotational displacement
reactx,reacty,reactz	Real	Translational reaction
rreactx,rreacty, rreactz	Real	Rotational reaction
qvec1,qvec2,qvec3, qvec4	Real	Unit quaternion

Table 8.3: Global Variables for J-Integral (See Section 9.2)

Variable Name	**Type**	**Comments**
`j_average_<jint_name>`	`Real[]`	Average value of the J-integral over the crack. Array sized to number of integration domains and numbered from inner to outer domain. `<jint_name>` is the name of the `J INTEGRAL` block.

Table 8.4: Nodal Variables for All Analyses

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 external_ transferred`	`Vector_3D`	Force transferred from another physics (coupled problems only)
`force_contact`	`Vector_3D`	
`residual`	`Vector_3D`	Force imbalance at current time step
`reaction`	`Vector_3D`	
`mass`	`Real`	
`nodal_time_step`	`Real`	Nodal stable time step (explicit control modes, coarse mesh only)
`hourglass_energy`	`Real`	Nodal integrated energy due to hourglass forces
`quaternion`	`Real`	Current quaternion (rigid body reference nodes only)

Table 8.5: Nodal Variables for Shells and Beams

Variable Name	**Type**	**Comments**
`rotational_displacement`	`Vector_3D`	
`rotational_velocity`	`Vector_3D`	
`rotational_acceleration`	`Vector_3D`	
`moment_internal`	`Vector_3D`	
`moment_external`	`Vector_3D`	
`moment external_ transferred`	`Vector_3D`	Moment transferred from another physics (coupled problems only)
`rotational_reaction`	`Vector_3D`	
`rotational_mass`	`Real`	

Table 8.6: Nodal Variables for Spot Welds

Variable Name	**Type**	**Comments**
`spot_weld_parametric_coordinates`	`Vector_2D`	Coordinates of node on face
`spot_weld_normal_force_at_death`	`Real`	Value of force normal to face when spot weld breaks
`spot_weld_tangential_force_at_death`	`Real`	Value of force tangential to face when spot weld breaks
`spot_weld_death_flag`	`Integer`	alive = 0, dead = FAILURE DECAY CYCLES (default is 10), -1 = no spot weld constructed at this node
`spot_weld_scale_factor`	`Real`	Nodal influence area of current node
`spot_weld_normal_displacement`	`Real`	Current displacement of weld normal to face
`spot_weld_tangential_displacement`	`Real`	Current displacement of weld tangential to face
`spot_weld_normal_force`	`Real`	Current force of weld normal to face
`spot_weld_tangential_force`	`Real`	Current force of weld tangential to face
`spot_weld_stiffness`	`Real`	Current stiffness of weld
`spot_weld_norm_stiffness`	`Real`	Current stiffness of weld normal to face
`spot_weld_tang_stiffness`	`Real`	Current stiffness of weld tangential to face
`spot_weld_initial_offset`	`Vector_3D`	The initial offset of the spot weld node from the spot weld surface. Does not change over time, only output if IGNORE INITIAL OFFSET = YES is specified at input.
`spot_weld_initial_normal`	`Vector_3D`	The initial normal of the spot weld surface at the point of interaction. Only output if IGNORE INITIAL OFFSET = YES is specified at input.

Table 8.7: Nodal Variables for Contact (See Section 7.5)

Variable	Type	Description
`contact_status`	`Real`	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 1.0 = Node is in contact and is slipping -1.0 = Node is in contact and is sticking (`celement`).
`contact_normal_direction`	`Vector_3D`	Direction of the constraint. This is, in general, the normal of the face in the interaction (`cdirnor`).
`contact_tangential_direction`	`Vector_3D`	Direction of the contact tangential force (`cdirtan`).
`contact_normal_force_magnitude`	`Real`	Magnitude of the contact force at the node in the direction normal to the contact face. Magnitude of `contact_normal_direction`.
`contact_tangential_force_magnitude`	`Real`	Magnitude of the contact force at the node in the plane of the contact face. Magnitude of `contact_tangential_direction`.
`contact_normal_traction_magnitude`	`Real`	Traction normal to the contact face. `contact_normal_force_magnitude` divided by `contact_area`. If there are multiple interactions for this node, the traction only for the last interaction is given (`cfnor`).
`contact_tangential_traction_magnitude`	`Real`	Traction in the plane of the contact face. `contact_traction_force_magnitude` divided by `contact_area`. If there are multiple interactions for this node, the traction only for the last interaction is given (`cftan`).

Continued on next page

Table 8.7 – Continued from previous page

Variable	Type	Description
contact_incremental_slip_magnitude	Real	Magnitude of incremental slip over the current time step (cdtan).
contact_incremental_slip_direction	Vector_3D	Normalized direction of incremental slip over the current time step (cdirislp).
contact_accumulated_slip	Real	Magnitude of tangential slip accumulated over the entire analysis. This is the distance along the slip path, and not the magnitude of contact_accumulated_slip_vector (cstan).
contact_accumulated_slip_vector	Vector_3D)	Total accumulated tangential slip over the entire analysis (cdirslp).
contact_frictional_energy	Real	Accumulated amount of frictional energy dissipated over the entire analysis.
contact_frictional_energy_density	Real	Accumulated amount of frictional energy dissipated over the entire analysis, divided by the contact area (cetan).
contact_area	Real	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 (carea).
contact_normal_gap	Real	Magnitude of gap in the direction normal to the face (cgnor).
contact_tangential_gap	Real	Magnitude of gap in the direction tangent to the face (only applicable for compliant friction models) (cgtan).

Table 8.8: Nodal Variables for J-Integral (See Section 9.2)

Variable Name	**Type**	**Comments**
`j_<jint_name>`	`Real[]`	Pointwise value of J-integral along crack. Array sized to number of integration domains and numbered from inner to outer domain. `<jint_name>` is the name of the `J INTEGRAL` block.

Table 8.9: Face Variables for Blast Pressure Boundary Condition (See Section 6.10.1)

Variable Name	**Type**	**Comments**
`pressure`	`Real`	Current total pressure. This is the only field for this boundary condition that varies in time.
`normal`	`Vector_3D`	Face normal vector
`incident_pressure`	`Real`	Peak incident pressure
`reflected_pressure`	`Real`	Peak reflected pressure
`alpha`	`Real`	Decay coefficient α
`beta`	`Real`	Decay coefficient β
`cosa`	`Real`	Cosine of θ
`arrival_time`	`Real`	Time for arrival of blast at face
`positive_duration`	`Real`	Duration of blast at face

Table 8.10: Element Variables for All Elements

Variable Name	**Type**	**Derived** (Sec 5.6)	**Comments**
`diagonal_ratio`	`Real`		See Section 2.4
`element_mass`	`Real`		
`perimeter_ratio`	`Real`		See Section 2.4
`solid_angle`	`Real`		See Section 2.4
`timestep`	`Real`		Critical time step for the element. The element in the model with the smallest time step controls the analysis time step.
`von_mises`	`Real`	yes	Von Mises stress norm
`hydrostatic_stress`	`Real`	yes	One-third the trace of the stress sensor
`fluid_pressure`	`Real`	yes	Negative of `hydrostatic_stress`
`stress_invariant_1`	`Real`	yes	Trace of the stress tensor
`stress_invariant_2`	`Real`	yes	Second invariant of the stress tensor
`stress_invariant_3`	`Real`	yes	Third invariant of the stress tensor
`max_principal_stress`	`Real`	yes	Largest eigenvalue of the stress tensor
`intermediate_principal_stress`	`Real`	yes	Middle eigenvalue of the stress tensor
`min_principal_stress`	`Real`	yes	Smallest eigenvalue of the stress tensor
`max_shear_stress`	`Real`	yes	Maximum shear stress from Mohr's circle
`octahedral_shear_stress`	`Real`	yes	Octahedral shear norm of the stress tensor

Table 8.11: Element Variables for Solid Elements

Variable Name	Type	Derived (Sec 5.6)	Comments
aspect_ratio	Real		Tets only. See Section 2.4
dilmod	Real		
left_stretch	SymTen33		
nodal_jacobian_ratio	Real		Hexes only. See Section 2.4
rate_of_deformation	SymTen33		Hexes and node-based tets only.
rotation	FullTen36		
shrmod	Real		
stress	SymTen33		
unrotated_stress	SymTen33		
log_strain	SymTen33		Log strain tensor
unrotated_log_strain	SymTen33		Log strain tensor in unrotated configuration
effective_log_strain	Real	yes	Effective log strain
log_strain_invariant_1	Real	yes	Trace of the log strain tensor
log_strain_invariant_2	Real	yes	Second invariant of the log strain tensor
log_strain_invariant_3	Real	yes	Third invariant of the log strain tensor
max_principal_log_strain	Real	yes	Largest eigenvalue of the log strain tensor
intermediate_principal_log_strain	Real	yes	Middle eigenvalue of the log strain tensor
min_principal_log_strain	Real	yes	Smallest eigenvalue of the log strain tensor
max_shear_log_strain	Real	yes	Maximum shear log strain from Mohr's circle
octahedral_shear_log_strain	Real	yes	Octahedral strain norm of the log strain tensor
volume	Real		

Table 8.12: Element Variables for Membranes

Variable Name	Type	Comments
memb_stress	SymTen33	
element_area	Real	
element_thickness	Real	

Table 8.13: Element Variables for Shells

Variable Name	**Type**	**Derived** (Sec 5.6)	**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
unrotated_stress	SymTen33		
transform_shell_stress	SymTen21	yes	In-plane shell stress
strain	SymTen33		Integrated strain at midplane in local shell coordinate system
effective_strain	Real	yes	Effective strain norm
strain_invariant_1	Real	yes	Trace of the strain tensor
strain_invariant_2	Real	yes	Second invariant of the strain tensor
strain_invariant_3	Real	yes	Third invariant of the strain tensor
max_principal_strain	Real	yes	Largest eigenvalue of the strain tensor
intermediate_principal_strain	Real	yes	Middle eigenvalue of the strain tensor
min_principal_strain	Real	yes	Smallest eigenvalue of the strain tensor
max_shear_strain	Real	yes	Maximum shear strain from Mohr's circle
octahedral_shear_strain	Real	yes	Octahedral strain norm of the strain tensor
transform_shell_strain	Real	yes	In-plane shell strain
element_area	Real		
element_thickness	Real		
rate_of_deformation	SymTen33		Rate of deformation (stretching) tensor

Table 8.14: Element Variables for Trusses

Variable Name	**Type**	**Comments**
`truss_init_length`	`Real`	
`truss_stretch`	`Real`	
`stress`	`SymTen33`	Axial stress is stored in `stress_xx`. All other components are zero. See Section 5.2.7 for more details.
`truss_strain_incr`	`Real`	
`truss_force`	`Real`	

Table 8.15: Element Variables for Cohesive Elements

Variable Name	**Type**	**Comments**
`cse_traction`	`Vector_3D`	
`cse_separation`	`Vector_3D`	
`cse_initial_trac`	`Vector_3D`	Available only if traction initialization is used
`cse_activated`	`Integer`	For intrinsic elements
`cse_fracture_area`	`Real`	Currently not used

Table 8.16: Element Variables for J-Integral (See Section 9.2)

Variable Name	**Type**	**Comments**
`energy_momentum_tensor`	`FullTen36`	Energy momentum tensor
`integration_domains_<jint_name>`	`Integer[]`	Flag indicating elements in integration domains. Set to 1 if in domain, 0 otherwise. Array sized to number of domains and numbered from inner to outer domain. `<jint_name>` is the name of the `J INTEGRAL` block.

8.9.2 Variables for Material Models

It is possible to output the state variables from the material models. Most of the materials, with the exception of simple models such as the elastic model, have state variables that can be output. The method used to output state variables depends on how the model is implemented. There are currently three cases:

- The Strumento version of most of the solid models for which the entire state variable array can be dumped.
- The Strumento version of a few solid models for which state variables are accessed by name
- The versions of the solid models implemented in the LAME library for which state variables are accessed by name

In the future, the implementation of the solid material models in LAME will be used by default, and all state variables will be accessed by name. The following sections describe the different methods required to output material model variables.

8.9.2.1 State Variable Output by Index for Strumento Solid Material Models

To output all of the state variables for a given material model. Use the `ELEMENT` command line in the `RESULTS OUTPUT` command block of the form:

```
ELEMENT state_material_name
```

where `material_name` is the name of the material model, e.g. `state_elastic_plastic`, `state_power_law_hardening`, `state_foam_plasticity`, or `state_orthotropic_rate`. All of the state variables for the material will be output.

Some of the Strumento material models are implemented in a way such that state variables are accessed directly by name rather than by index. For example, to access the `C10` variable in the Mooney-Rivlin material model, one would simply list the name `C10` to obtain that output. The state variables for the Mooney-Rivlin, Swanson, and Orthotropic Crush material models are accessed in this way.

Section 8.9.2.3 provides tables listing the state variables for all solid material models. Models for which state variable output is requested by name have names entered in a column entitled "Name (Strumento Model)".

8.9.2.2 State Variable Output for LAME Solid Material Models

The state variables for material models in LAME are accessible directly by name. For instance, the equivalent plastic strain variable is accessible by the name `EQPS` for all elastic-plastic material models.

Section 8.9.2.3 provides tables listing the state variables for all solid material models. Models that are implemented in LAME have state variable names listed in the column entitled "Name (LAME Model)". If there are no entries in that column for a given material, then that material is not implemented in LAME.

Available LAME state variables for a material will also be listed in the log file from a run that uses the material model.

8.9.2.3 State Variable Tables for Solid Material Models

As explained in the preceding sections, there are three cases to be considered for state variable output from solid material models: Strumento models, Strumento models for which state output is obtained using the variable name, and LAME models, for which state output is also obtained by variable name. Tables of state variables for commonly used material models are provided in Tables 8.21 through 8.43. These tables contain the indices or names used to access the state variables in the Strumento version of the models, as well as the names used to access the LAME versions of the models. If there are no entries in the "Strumento Model" column for a model, that model is only implemented in LAME. Likewise, if there are no entries in the "LAME Model" column, there is no version of that model in LAME.

Table 8.17: State Variables for ELASTIC Model (Section 4.2.1)

This model has no state variables.

Table 8.18: State Variables for ELASTIC FRACTURE Model (Section 4.2.4)

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
1	`DEATH_FLAG`	flag for element death
2	`CRACK_OPENING_STRAIN`	critical value of opening strain
3	`FAILURE_DIRECTION_X`	crack opening direction - x component
4	`FAILURE_DIRECTION_Y`	crack opening direction - y component
5	`FAILURE_DIRECTION_Z`	crack opening direction - z component
6	`PRINCIPAL_STRESS`	value of maximum principal stress

Table 8.19: State Variables for ELASTIC PLASTIC Model (Section 4.2.5)

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
1	`EQPS`	equivalent plastic strain
2	`BACK_STRESS_XX`	back stress - xx component
3	`BACK_STRESS_YY`	back stress - yy component
4	`BACK_STRESS_ZZ`	back stress - zz component
5	`BACK_STRESS_XY`	back stress - xy component
6	`BACK_STRESS_YZ`	back stress - yz component
7	`BACK_STRESS_ZX`	back stress - zx component
8	`RADIUS`	radius of yield surface

Table 8.20: State Variables for EP POWER HARD Model (Section 4.2.6)

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
1	`EQPS`	equivalent plastic strain
2	`RADIUS`	radius of yield surface

Table 8.21: State Variables for DUCTILE FRACTURE Model (Section 4.2.7)

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
1	`EQPS`	equivalent plastic strain
2	`TEARING_ PARAMETER`	tearing parameter
3	`CRACK_OPENING_ STRAIN`	crack opening strain
4	`FAILURE_ DIRECTION_X`	crack opening direction - x component
5	`FAILURE_ DIRECTION_Y`	crack opening direction - y component
6	`FAILURE_ DIRECTION_Z`	crack opening direction - z component
7	`DEATH_FLAG`	flag for element death
8	`RADIUS`	radius of yield surface

Table 8.22: State Variables for MULTILINEAR EP Model (Section 4.2.8)

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
1	`EQPS`	equivalent plastic strain
2	`RADIUS`	radius of yield surface
3	`BACK_STRESS_XX`	back stress - xx component
4	`BACK_STRESS_YY`	back stress - yy component
5	`BACK_STRESS_ZZ`	back stress - zz component
6	`BACK_STRESS_XY`	back stress - xy component
7	`BACK_STRESS_YZ`	back stress - yz component
8	`BACK_STRESS_ZX`	back stress - zx component

Table 8.23: State Variables for ML EP FAIL Model (Section 4.2.9)

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
1	`EQPS`	equivalent plastic strain
2	`RADIUS`	radius of yield surface
3	`BACK_STRESS_XX`	back stress - xx component
4	`BACK_STRESS_YY`	back stress - yy component
5	`BACK_STRESS_ZZ`	back stress - zz component
6	`BACK_STRESS_XY`	back stress - xy component
7	`BACK_STRESS_YZ`	back stress - yz component
8	`BACK_STRESS_ZX`	back stress - zx component
9	`TEARING_ PARAMETER`	tearing parameter
10	`CRACK_OPENING_ STRAIN`	crack opening strain
11	`FAILURE_ DIRECTION_X`	crack opening direction - x component
12	`FAILURE_ DIRECTION_Y`	crack opening direction - y component
13	`FAILURE_ DIRECTION_Z`	crack opening direction - z component
14	`CRACK_FLAG`	status of the model: 0 for loading, 1 or 2 for initiation of failure, 3 during unloading, 4 for completely unloaded

Table 8.24: State Variables for FOAM PLASTICITY Model (Section 4.2.15)

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
1	`ITER`	iterations
2	`EVOL`	volumetric strain
3	`PHI`	phi
4	`EQPS`	equivalent plastic strain
5	`PA`	A
6	`PB`	B

Table 8.25: State Variables for WIRE MESH Model (Section 4.2.18)

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
1	`EVOL`	engineering volumetric strain
2	`PHI`	current yield strength in compression

Table 8.26: State Variables for HONEYCOMB Model

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
1	`CRUSH`	minimum volume ratio
2	`EQDOT`	effective strain rate
3	`RMULT`	rate multiplier
5	`ITER`	iterations
6	`EVOL`	volumetric strain

Table 8.27: State Variables for HYPERFOAM Model

This model has no state variables.

Table 8.28: State Variables for JOHNSON COOK Model (Section 4.2.10)

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
1	`RADIUS`	radius of yield surface
2	`EQPS`	equivalent plastic strain
3	`THETA`	temperature
4	`EQDOT`	effective total strain rate

Table 8.29: State Variables for LOW DENSITY FOAM Model (Section 4.2.16)

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
	`PAIR`	air pressure

Table 8.30: State Variables for MOONEY RIVLIN Model (Section 4.2.24)

Name (Strumento Model)	**Name (LAME Model)**	**Variable Description**
`C10`	`C10`	
`C01`	`C01`	
`RK`	`K`	
`SFJth`	`SFJTH`	
`RJTH`	`JTH`	
`V_MECH`	`VMECH_XX`	
	`VMECH_YY`	
	`VMECH_ZZ`	
	`VMECH_XY`	
	`VMECH_YZ`	
	`VMECH_ZX`	
	`SFJTH_FLAG`	

Table 8.31: State Variables for NEO HOOKEAN Model (Section 4.2.3)

This model has no state variables.

Table 8.32: State Variables for ORTHOTROPIC CRUSH Model (Section 4.2.19)

Name (Strumento Model)	**Name (LAME Model)**	**Variable Description**
`CRUSH`	`CRUSH`	minimum volume ratio, crush is unrecoverable

Table 8.33: State Variables for ORTHOTROPIC RATE Model (Section 4.2.20)

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
	`CRUSH`	minimum volume ratio, crush is unrecoverable

Table 8.34: State Variables for PIEZO Model

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
	STATE	

Table 8.35: State Variables for POWER LAW CREEP Model (Section 4.2.12)

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
1	ECREEP	equivalent creep strain
2	SEQDOT	equivalent stress rate

Table 8.36: State Variables for SHAPE MEMORY Model

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
	STATE	

Table 8.37: State Variables for SOIL FOAM Model (Section 4.2.13)

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
	EVOL	

Table 8.38: State Variables for SWANSON Model (Section 4.2.27)

Name (Strumento Model)	**Name (LAME Model)**	**Variable Description**
SFJTH	SFJTH	
RJTH	RJTH	
V_MECH	VMECHXX	
	VMECHYY	
	VMECHZZ	
	VMECHXY	
	VMECHYZ	
	VMECHZX	
	SFJTH_FLAG	

Table 8.39: State Variables for VISCOELASTIC SWANSON Model (Section 4.2.28)

Index (Strumento Model)	Name (LAME Model)	Variable Description
	SFJTH	
	JTH	
	VMECHXX	
	VMECHYY	
	VMECHZZ	
	VMECHXY	
	VMECHYZ	
	VMECHZX	
	VSXXDEV1 – VSXXDEV10	
	VSYYDEV1 – VSYYDEV10	
	VSZZDEV1 – VSZZDEV10	
	VSXYDEV1 – VSXYDEV10	
	VSYZDEV1 – VSYZDEV10	
	VSZXDEV1 – VSZXDEV10	
	SOXXDEV	
	SOYYDEV	
	SOZZDEV	
	SOXYDEV	
	SOYZDEV	
	SOZXDEV	

Table 8.40: State Variables for THERMO EP POWER Model

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
1	`EQPS`	equivalent plastic strain
2	`RADIUS`	radius of yield surface
3	`BACK_STRESS_XX`	back stress - xx component
4	`BACK_STRESS_YY`	back stress - yy component
5	`BACK_STRESS_ZZ`	back stress - zz component
6	`BACK_STRESS_XY`	back stress - xy component
7	`BACK_STRESS_YZ`	back stress - yz component
8	`BACK_STRESS_ZX`	back stress - zx component

Table 8.41: State Variables for THERMO EP POWER WELD Model

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
	`EQPS`	equivalent plastic strain
	`RADIUS`	radius of yield surface
	`BACK_STRESS_XX`	back stress - xx component
	`BACK_STRESS_YY`	back stress - yy component
	`BACK_STRESS_ZZ`	back stress - zz component
	`BACK_STRESS_XY`	back stress - xy component
	`BACK_STRESS_YZ`	back stress - yz component
	`BACK_STRESS_ZX`	back stress - zx component
	`WELD_FLAG`	

Table 8.42: State Variables for UNIVERSAL POLYMER Model

Index (Strumento Model)	**Name (LAME Model)**	**Variable Description**
	`AEND`	
	`IGXX1 - IGXX20`	
	`IGYY1 - IGYY20`	
	`IGZZ1 - IGZZ20`	
	`IGXY1 - IGXY20`	
	`IGYZ1 - IGYZ20`	
	`IGZX1 - IGZX20`	
	`IKI11 - IKI120`	
	`IKAT1 - IKAT20`	
	`IF1P1 - IF1P20`	
	`IF2J1 - IF2J20`	
	`EPSXX`	
	`EPSYY`	
	`EPSZZ`	
	`EPSXY`	
	`EPSYZ`	
	`EPSZX`	
	`LOGA`	

Table 8.43: State Variables for VISCOPLASTIC Model

Index (Strumento Model)	Name (LAME Model)	Variable Description
	SVBXX	
	SVBYY	
	SVBZZ	
	SVBXY	
	SVBYZ	
	SVBZX	
	EQDOT	
	COUNT	
	SHEAR	
	BULK	
	RATE	
	EXP	
	ALPHA	
	A1	
	A2	
	A4	
	A5	

8.9.2.4 Variables for Shell/Membrane Material Models

Shell and membrane material models also make their state variables available through direct naming of the variables. Tables 8.44 through 8.47 indicate the names of the state variables for the shell material models.

Table 8.44: State Variables for ELASTIC PLASTIC Model for Shells (Section 4.2.5)

Variable Name	**Variable Description**
`eqps`	Equivalent plastic strain
`back_stress`	Back stress
`radius`	Radius of the yield surface

Table 8.45: State Variables for EP POWER HARD Model for Shells (Section 4.2.6)

Variable Name	**Variable Description**
`eqps`	Equivalent plastic strain
`radius`	Radius of yield surface

Table 8.46: State Variables for MULTILINEAR EP Model for Shells (Section 4.2.8)

Variable Name	**Variable Description**
`eqps`	Equivalent plastic strain
`tensile_eqps`	Equivalent plastic strain only accumulated in tension
`back_stress`	Back stress
`radius`	Radius of the yield surface

Table 8.47: State Variables for ML EP FAIL Model for Shells (Section 4.2.9)

Variable Name	**Variable Description**
`eqps`	Equivalent plastic strain
`back_stress`	Back stress
`radius`	Radius of the yield surface
`tearing_parameter`	The current value of the tearing parameter
`crack_opening_strain`	The value of the crack opening strain during the failure process
`crack_flag`	Status of the model: 0 for loading, 1 or 2 for initiation of failure, 3 during unloading, 4 for completely unloaded

8.9.3 Variables for Surface Models

It is possible to output the state variables from the surface models. The element state variables are output using the variable name by use of following line command:

```
ELEMENT surface_model_state_name
```

Section 8.9.3.1 provides tables listing the state variables for all surface models.

8.9.3.1 State Variable Tables for Surface Models

Table 8.48: State Variables for TRACTION DECAY Surface Model (Section 4.3.1)

Index	Name	Variable Description
0	`MAX_ SEPARATION_S`	maximum separation in the first tangential direction
1	`MAX_ SEPARATION_T`	maximum separation in the second tangential direction
2	`MAX_ SEPARATION_N`	maximum separation in the normal direction

Table 8.49: State Variables for TVERGAARD HUTCHINSON Surface Model (Section 4.3.2)

Index	Name	Variable Description
0	`PEAK_TRACTION`	maximum traction the model can experience
1	`LAMBDA_MAX`	maximum lambda the model has experienced
2	`TRACTION_AT_ LAMBDA_MAX`	traction at LAMBDA_MAX

Table 8.50: State Variables for THOULESS PARMIGIANI Surface Model (Section 4.3.3)

Index	**Name**	**Variable Description**
0	`FRACTION_OF_FAILURE`	current fraction of failure
1	`PEAK_TRACTION_N`	maximum normal traction the model can experience
2	`PEAK_TRACTION_T`	maximum tangential traction the model can experience
3	`LAMBDA_MAX_N`	maximum normal lambda the model has experienced
4	`TRACTION_AT_LAMBDA_MAX_N`	normal traction at LAMBDA_MAX_N
5	`LAMBDA_MAX_T`	maximum tangential lambda the model has experienced
6	`TRACTION_AT_LAMBDA_MAX_T`	tangential traction at LAMBDA_MAX_T
7	`G_AT_LAMBDA_MAX_N`	the area under the normal traction separation curve up to LAMBDA_MAX_N
8	`G_AT_LAMBDA_MAX_T`	the area under the tangential traction separation curve up to LAMBDA_MAX_T

8.10 References

1. Larry A. Schoof, Victor R. Yarberry, *EXODUS II: A Finite Element Data Model*, SAND92-2137, Sandia National Laboratories, September 1994. pdf. See also documentation available at EXODUS II sourceforge page. link.

2. The eXtensible Data Model and Format (XDMF). link.

3. Sjaardema, G. D. *Overview of the Sandia National Laboratories Engineering Analysis Code Access System*, SAND92-2292. Albuquerque, NM: Sandia National Laboratories, January 1993. pdf.

Chapter 9

Special Modeling Techniques

This chapter describes techniques useful for performing special types of analyses:

- Section 9.1 describes the Representative Volume Element (RVE) capability, which is a multiscale technique that uses a separate finite element model to represent the material response at a point.
- Section 9.2 describes the capability to compute J-Integrals as a criterion for fracture growth.

9.1 Representative Volume Elements

The use of representative volume elements (RVEs) is a multiscale technique in which the material response at element integration points in a reference mesh is computed using an RVE that is itself discretized with finite elements. RVEs are typically used to represent local, periodic material inhomogeneities such as fibers or random microstructures to avoid the requirement of a global mesh with elements small enough to capture local material phenomena.

This capability is currently implemented only for uniform gradient hex elements in the reference mesh. In the current implementation of RVEs, periodic boundary conditions are applied to each RVE representing the deformation of a parent element and the stresses are computed in the elements of the RVE. These stresses are then volume-averaged over the RVE and the resulting homogenized stresses are passed back to the parent element.

This chapter explains how to use the RVE capability. Section 9.1.1 gives a detailed description of how RVEs are incorporated into an analysis. Details of the mesh requirements are delineated in Section 9.1.2 and the commands needed in an input file are described in Section 9.1.3.

9.1.1 RVE Processing

The use of the RVE capability requires two regions, each with its own mesh file. One region processes the reference mesh and the other processes all the RVEs. The commands used in the input file for the reference mesh region are the same as any other Adagio region with the exception that a special RVE material model is used for any element blocks that use an RVE. The RVE region is also very similar to an ordinary region. The only differences are that an RVE region has a line command for defining the RVEs' relationship to parent elements in the reference region and has restrictions on the use of boundary conditions.

The processing of an RVE essentially replaces the constitutive model of the parent element in the reference mesh. The steps followed at each iteration/time step of the reference mesh during an analysis using RVEs are:

1. Internal force algorithm is called in the reference region to compute rate of deformation.
2. Each RVE gets the rate of deformation from its parent element in the reference region.
3. The rate of deformation is applied to each RVE as a periodic boundary condition using prescribed velocity.
4. The RVE region is solved to obtain the stress in each element of each RVE.
5. The stresses in the elements of an RVE are volume-averaged over the RVE.
6. Each RVE passes its homogenized (i.e. volume-averaged) stress tensor back to its parent element in the reference mesh.
7. The reference region computes internal force again. Element blocks whose elements have associated RVEs do not compute a stress; they simply use the stress passed to them from their RVE.

9.1.2 Mesh Requirements

Two mesh files, one each for the reference region and the RVE region, are required for an RVE analysis. Figure 9.1 shows an example of the two meshes. The reference mesh of a bar with six elements is shown on the upper left. On the lower right is the mesh for the RVE region containing six RVEs, one for each element of the reference region. In this case, the first five RVEs each consist of two element blocks and the last RVE has four.

In general, each RVE should be a cube with any discretization the user desires. All RVEs must be aligned with the global x, y, and z axes. For stress computations, these axes are rotated into a local coordinate system that can be specified on the reference mesh elements. In other words, if a local coordinate system is specified on a reference mesh element, the RVE global axes will be rotated internally in Adagio to align with the local system on the associated parent element. So the global X axis for an RVE is actually the local X' axis in the parent element.

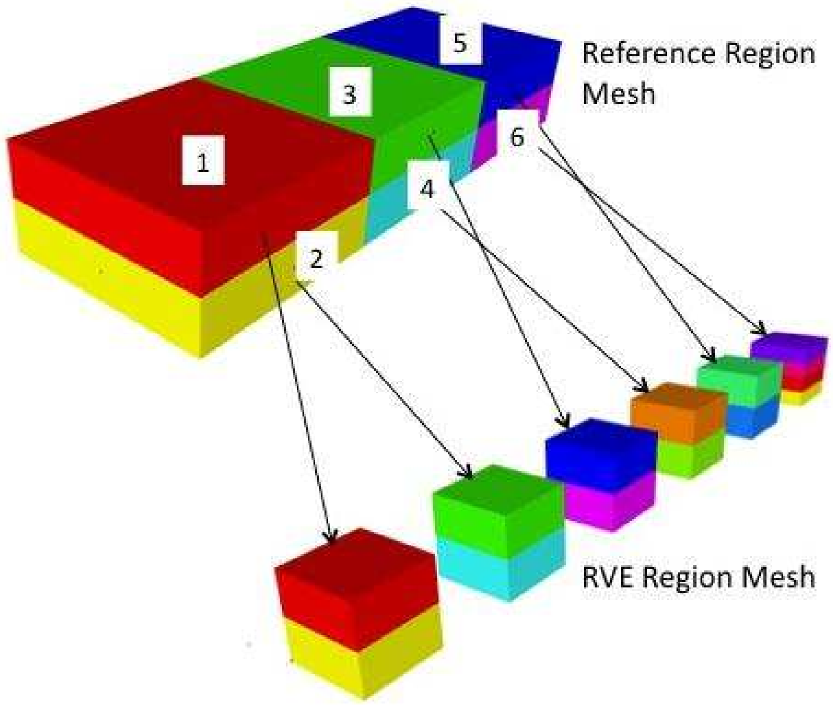


Figure 9.1: Example of meshes for RVE analysis

Additional mesh requirements apply if the mesh does not match across opposing surfaces of the RVE. In this case, the RVE must include a block of membrane elements on the exterior surfaces with matching discretization on opposing surfaces (+x/-x, +y/-y, +z/-z). In order the minimize the effects of this membrane layer on the RVE response, it should be made as thin as possible. This membrane layer then must be tied to the underlying nonmatching RVE surfaces.

The RVE mesh must contain sidesets or nodesets on each surface of every RVE. The RVE may be enclosed with one sideset that spans all six surfaces of the curb, or the user may specify individual sidesets or nodesets on each face. These sidesets/nodesets are used to apply the periodic boundary conditions on the RVE. Adagio generates the boundary conditions internally so the user does not have to include them in the input file. However, this assumes that the sidesets/nodesets exist in the mesh file numbered in a specified order. If individual sidesets/nodesets are used on each face of the RVE, the six sidesets/nodesets must be numbered consecutively, starting with the positive-x face, followed by the negative-x face, positive-y face, negative-y face, positive-z face, and ending with the negative-z face. The beginning sideset id (for the positive-x face) is set by the user in the input file.

9.1.3 Input Commands

There are several input commands that are relevant to RVEs. In the reference region, these commands include a special RVE material model and commands to define and use a local coordinate system along which an associated RVE will be aligned. In addition to the reference region, an RVE region is needed using the `BEGIN RVE REGION` command block. The RVE region command block uses the same nested commands as any other Adagio region (with some restrictions as explained in this section) and an additional line command that relates the RVEs to their parent elements in the reference region.

9.1.3.1 RVE Material Model

In an RVE analysis, any elements of the reference mesh that use an RVE must use the RVE material model. This model is defined similar to other material models as described in Chapter 4 but uses the RVE keyword on the `BEGIN PARAMETERS FOR MODEL` command line as follows:

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  #
  DENSITY = <real>density_value
  #
  BEGIN PARAMETERS FOR MODEL RVE
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
  END PARAMETERS FOR MODEL RVE
  #
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]
```

Currently, the RVE material model tells the reference element not to perform a constitutive evaluation but to instead accept the stress tensor obtained from computation on an RVE. However, the use of an RVE material model still requires the input of Young's modulus and Poissons ratio. These values may be used for time time step estimation and hourglass computations even though they are not used in a constitutive evaluation.

Element blocks in the RVE region can use any material model that is supported in Adagio other than RVE.

9.1.3.2 Embedded Coordinate System

The finite element model of an element block in the reference mesh that uses RVEs can use an embedded coordinate system to orient the RVE relative to the reference element. A coordinate system is defined in the sierra scope as described in Section 2.1.8. A local coordinate system is then associated with an element block through the use of a `COORDINATE SYSTEM` command line within a `BEGIN SOLID SECTION` command block.

```
BEGIN SOLID SECTION <string>section_name
  #
  COORDINATE SYSTEM = <string>coord_sys_name
  #
END [SOLID SECTION <string>section_name]
```

The string coord_sys_name must be a name associated in the input file with a BEGIN COORDINATE SYSTEM command block in the sierra scope. This coordinate system will then be used on all elements of a block associated with a BEGIN PARAMETERS FOR BLOCK command block that includes the command line specifying this solid section.

9.1.3.3 RVE Region

A representative volume element (RVE) region must be a quasistatic region specified with the RVE keyword in the BEGIN RVE REGION command line. The RVE region uses the same block commands and line commands as any other quasistatic region with the addition of line commands that define which element blocks of the reference region are associated with RVEs. There are also some restrictions on boundary conditions as described in Section 9.1.3.6.

```
BEGIN RVE REGION <string>rve_region_name
  #
  # Definition of RVEs
  ELEMENTS <integer>elem_i:<integer>elem_j
    BLOCKS <integer>blk_i:<integer>blk_j
    SURFACE|NODESET <integer>start_id INCREMENT <integer>k
  #
  # Boundary Conditions
  #
  # Results Output Definition
  #
  # Solver Definition
  #
END [RVE REGION <string>rve_region_name]
```

9.1.3.4 Definition of RVEs

One or more ELEMENTS command lines are used to associate elements of the reference region mesh with RVEs in the RVE region. In the

```
ELEMENTS <integer>elem_i:<integer>elem_j
  BLOCKS <integer>blk_i:<integer>blk_j
  SURFACE|NODESET <integer>start_id INCREMENT <integer>incr
```

command line, elements numbered elem_i through elem_j of the reference mesh will be associated with RVEs (for a total number of RVEs equal to elem_j - elem_i + 1), and each RVE will consist of blk_i - blk_j + 1 element blocks. The block ids of the first RVE must be blk_i through blk_j and subsequent RVEs (if elem_j is greater than elem_i) must have consecutively increasing numbers for their block ids.

Similarly start_id gives the surface_id of the first RVE if a single, encompassing surface is used, or the first surface_id or nodelist_id of the first RVE (the positive x surface as explained in Section 9.1.2) if six individual sidesets/nodeset are used. The remaining surfaces (nodesets) of the first RVE and all the surfaces of the following RVEs must be consecutively numbered following `start_id` in the mesh file as explained in Section 9.1.2.

The increment value `incr` indicates the number of sidesets present on the exterior of the RVEs. This is used to determine how to increment the IDs of the sidesets from one RVE to the next, as well as to determine how to prescribe periodic boundary conditions on the RVE. The increment can have a value of either one or six. A value of one indicates that each RVE has one sideset that encompasses all six faces, while a value of six specifies that six sidesets or nodesets are present, one on each face. Note that nodesets are not allowed for the case where `incr` is one.

The following example shows the use of the `ELEMENTS` command line:

```
elements 1:5 blocks 1:2 surface 7 increment 6
elements 6:6 blocks 11:14 nodeset 15 increment 6
```

These commands generate the RVEs shown in Figure 9.1.

The first `ELEMENTS` command line specifies that elements with element ids 1 through 5 in the parent region mesh each have an RVE with two element blocks. The RVE associated with element 1 of the parent region will have two element blocks starting with `block_id` of 1 and ending with a `block_id` of 2. Subsequent RVEs will have consecutively numbered element blocks. That is, parent element 2 will be associated with an RVE that consists of element blocks 3 and 4 in the RVE region, parent element 3 will be associated with the RVE that has element blocks 5 and 6, etc., for the first five elements of the parent region mesh. The keyword `SURFACE` specifies that all the periodic boundary conditions generated by the code for the RVEs for elements 1 to 5 will use sidesets in the RVE region mesh. These sidesets will start with id 7 for the positive-x face of the RVE associated with parent element 1 and continue consecutively for the other faces of the RVE and the RVEs associated with parent elements 2 through 5 in the order specified in Section 9.1.2. In other words, the positive-x face of the RVE for parent element 1 is sideset 7, negative-x is sideset 8, positive-y is sideset 9, negative-y is sideset 10, positive-z is sideset 11, and negative-z is sideset 12. The sidesets for the RVE for parent element 2 will start with id 13 and continue consecutively in the same face order. The process continues for all five RVEs specified in this command line.

The second `ELEMENTS` line specifies that element 6 of the parent region mesh will be associated with the RVE that consists of element blocks 11, 12, 13, and 14. The `NODESET` keyword says this RVE has a nodeset associated with each face of the RVE, starting with nodeset id 15 on the positive-x face, with id's increasing consecutively for the other five faces in the same order described in the paragraph above.

Note that the six elements specified in these command lines must be in element blocks of the reference region mesh that use the RVE material model.

9.1.3.5 Multi-Point Constraints

In the case in which the RVE has nonmatching surfaces, and therefore includes a block of membrane elements on the exterior surfaces, the user must specify a set of multi-point constraints

(MPCs) to tie the membranes to the surface. This is done in the input file through use of an MPC command block:

```
RESOLVE MULTIPLE MPCS = ERROR
BEGIN MPC
  MASTER SURFACE = <string>membrane_surface_id
  SLAVE SURFACE  = <string>RVE_surface_id
  SEARCH TOLERANCE = <real>tolerance
END
```

In this case, the membrane_surface_id corresponds to the single sideset that encompasses the membrane block is the master and the single sideset that encompasses the exterior surfaces of the RVE is the slave. While the underlying RVE may have nonmatching exterior surfaces, the opposing surfaces of the membrane block must have matching discretizations. For more information on the use of MPCs, see Section 6.10.2.1 and the RESOLVE MULTIPLE MPCS command line is discussed in Section 6.10.2.4.

9.1.3.6 RVE Boundary Conditions

Strain rates computed by elements in the reference region are applied through periodic prescribed velocity boundary conditions on the faces of the associated RVEs. These are generated internally by Adagio so the periodic boundary conditions do not need to be in the user's input file. However, because the RVE region is quasistatic, each of the RVEs must be fixed against rigid body motion. This must be done in the input file through use of the prescribed velocity boundary conditions:

```
BEGIN PRESCRIBED VELOCITY pres_vel_name
  NODE SET = <string>nodelist_name
  FUNCTION = <string>function_name
  SCALE FACTOR = <real>scale_factor
  COMPONENT = <string>X|Y|Z
END [PRESCRIBED VELOCITY pres_vel_name]
```

This type of boundary condition is described in detail in Chapter 6 but the use for RVEs is restricted. First, either the function must always evaluate to 0.0 or the scale_factor must have a value of 0. This is essentially a way of using the prescribed velocity boundary condition to fix the nodes in nodelist_name. However, in order for these conditions to work with the periodic boundary conditions which are used to apply the strain rate, PRESCRIBED VELOCITY must be used rather than FIXED DISPLACEMENT or PRESCRIBED DISPLACEMENT boundary conditions.

Generally, three BEGIN PRESCRIBED VELOCITY command blocks will be needed, one each for X, Y, and Z components. In order to eliminate rigid body motion without over constraining the motion, each BEGIN PRESCRIBED VELOCITY block should constrain exactly one node of an RVE in one component direction. (However, nodelist_name may contain nodes from multiple RVEs. Separate boundary condition blocks are not required for each RVE.). To prevent rigid body rotations, the three constrained nodes on each RVE should not be collinear.

9.2 *J*-Integrals

Adagio provides a capability to compute the J-integral via a a domain integral.

Known Issue: Currently, the J-Integral evaluation capability is based on assumptions of elastostatics and a stationary crack, and is only implemented for uniform gradient hex elements.

J is analogous to G from linear elastic fracture mechanics ($-\delta\pi/\delta a$) and is the driving force on the crack tip a [1,2]. Crack propagation occurs when $J \geq R$, where R is the material resistance and is often referred to as the critical energy release rate J_{1c}. In the reference configuration, the vectorial form of the J-integral in finite deformation [4] is

$$\boldsymbol{J} = \int_{\Gamma_0} \boldsymbol{\Sigma} \boldsymbol{N} dA \tag{9.1}$$

where $\boldsymbol{\Sigma} = W\boldsymbol{I} - \boldsymbol{F}^T\boldsymbol{P}$ is referred to as the Eshelby energy-momentum tensor [3]. W is the stored energy density in the reference configuration and $\boldsymbol{F}$ and $\boldsymbol{P}$ are the deformation gradient and first Piola-Kirchhoff stress, respectively. Rice [2] realized that because $\boldsymbol{\Sigma}$ is divergence-free in the absence of body forces, one can examine $\boldsymbol{J}$ in the direction of the defect $\boldsymbol{L}$ (unit vector) and obtain a path-independent integral for traction-free crack faces. J can be written as

$$J = \int_{\Gamma_0} \boldsymbol{L} \cdot \boldsymbol{\Sigma} \boldsymbol{N} dA \tag{9.2}$$

and interpreted as a path-independent driving force in the direction of the defect. We note that one can also express $\boldsymbol{\Sigma}$ in terms of $\bar{\boldsymbol{\Sigma}}$, where $\bar{\boldsymbol{\Sigma}} = W\boldsymbol{I} - \boldsymbol{H}^T\boldsymbol{P}$ and $\boldsymbol{H} = \text{Grad}\,\boldsymbol{u}$. Although $\boldsymbol{\Sigma}$ is symmetric and $\bar{\boldsymbol{\Sigma}}$ is not symmetric, they are equivalent when integrated over the body ($\text{Div}\boldsymbol{P} = \boldsymbol{0}$). In fact, differences in the energy-momentum tensor stem from the functional dependence of the stored energy function W. $\boldsymbol{\Sigma}$ and $\bar{\boldsymbol{\Sigma}}$ derive from $W(\boldsymbol{F})$ and $W(\boldsymbol{H})$, respectively. When integrated, both collapse to the familiar 2-D relation for infinitesimal deformations.

$$J = \int_{\Gamma} \boldsymbol{e}_1 \cdot \boldsymbol{\Sigma} \boldsymbol{n} ds = \int_{\Gamma} (W n_1 - u_{i,1} \sigma_{ij} n_j) \mathrm{d}s \tag{9.3}$$

9.2.1 Technique for Computing *J*

J is often expressed as a line (2D) or surface (3D) integral on a ring surrounding the crack tip. Defining a smooth ring over which to compute this surface integral and performing projections of the required field values onto that ring presents a number of difficulties in the context of a finite element code.

To compute the J-integral in a finite element code, it is more convenient to perform a volume integral over a domain surrounding the crack tip. We can then leverage the information at integration points rather than rely on less accurate projections. To do this, we follow the method described in

[5]. We replace $\boldsymbol{L}$ with a smooth function $\boldsymbol{q}$. On the inner contour of the domain Γ_0, $\boldsymbol{q} = \boldsymbol{L}$. On the outer contour of the domain C_0, $\boldsymbol{q} = \boldsymbol{0}$. Because the outer normal of the domain $\boldsymbol{M}$ is equal and opposite of the normal $\boldsymbol{N}$ on Γ_0, there is a change of sign. For traction-free surfaces, we can apply the divergence theorem, enforce $\text{Div}\boldsymbol{\Sigma} = \boldsymbol{0}$, and find that the energy per unit length $\bar{J}$ is

$$\bar{J} = -\int_{\Omega_0} (\boldsymbol{\Sigma} : \text{Grad}\, \boldsymbol{q}) \mathrm{d}V. \tag{9.4}$$

We note that the all the field quantities are given via simulation and we choose to define $\boldsymbol{q}$ on the nodes of the domain $\boldsymbol{q}^I$ and employ the standard finite element shape functions to calculate the gradient. We can specify the crack direction $\boldsymbol{L}$ or assume that the crack will propagate in the direction normal to the crack front $-\boldsymbol{M}$. For a "straight" crack front, $\boldsymbol{L} = -\boldsymbol{M}$. If $\boldsymbol{S}$ is is tangent to the crack front and $\boldsymbol{T}$ is normal to the lower crack surface, $\boldsymbol{S} \times \boldsymbol{M} = \boldsymbol{T}$. We note that for non-planar, curving cracks, $\boldsymbol{M}$, $\boldsymbol{S}$, and $\boldsymbol{T}$ are functions of the arc length S. For ease, we employ the notation $\boldsymbol{N}$ rather than $-\boldsymbol{M}$. For a crack front S_0, we can define the average driving force J_{avg} as

$$J_{avg} = \frac{\bar{J}}{\int_{S_0} \boldsymbol{L} \cdot \boldsymbol{N} \mathrm{d}S}. \tag{9.5}$$

While the average driving force is useful for interpreting experimental findings and obtaining a macroscopic representation of the driving force, we also seek to examine the local driving force $J(S)$. Using the finite element interpolation functions to discretize $\boldsymbol{L}$ through the smooth function $\boldsymbol{q}$, we find $\boldsymbol{q} = \lambda^I \boldsymbol{q}^I$. For a specific node K, we can define $|\boldsymbol{q}^K| = 1$ and $\boldsymbol{q}^I = \boldsymbol{0}$ for all other $I \neq K$ on S_0. Note that we still need to specify the function $\boldsymbol{q}$ in the $\boldsymbol{S} - \boldsymbol{T}$ plane from the inner contour Γ_0 to the outer contour C_0. The resulting expression for the approximate, pointwise driving force at node K on the crack front is

$$J^K = \frac{\bar{J}}{\int_{S_0} \lambda^K \boldsymbol{q}^K \cdot \boldsymbol{N} \mathrm{d}S}. \tag{9.6}$$

Again, we note that if the direction of propagation $\boldsymbol{L}$ is taken in the direction of the normal $\boldsymbol{N}$, the denominator is $\int_{S_0} \lambda^K \mathrm{d}S$. More information regarding the pointwise approximation of J^K can be found in [6,7].

9.2.2 Input Commands

A user can request that J-integrals be computed during the analysis by including one or more `J INTEGRAL` command blocks in the `REGION` scope. This block can contain the following commands:

```
BEGIN J INTEGRAL <jint_name>
  #
  # integral parameter specification commands
  CRACK DIRECTION = <real>dir_x <real>dir_y <real>dir_z
  CRACK PLANE SIDE SET = <string list>side_sets
  CRACK TIP NODE SET = <string list>node_sets
  INTEGRATION RADIUS = <real>int_radius
```

```
    NUMBER OF DOMAINS = <integer>num_domains
    FUNCTION = PLATEAU|PLATEAU_RAMP|LINEAR(PLATEAU)
    SYMMETRY = OFF|ON(OFF)
    #
    # time period selection commands
    ACTIVE PERIODS = <string list>period_names
    INACTIVE PERIODS = <string list>period_names
  END J INTEGRAL <jint_name>
```

A set of parameters must be provided to define the crack geometry and the integration domains used in the calculation of the J-integral. The model must be set up so that there is a sideset on one surface of the crack plane behind the crack tip and a nodeset containing the nodes on the crack tip. Both the `CRACK PLANE SIDE SET` and `CRACK TIP NODE SET` commands must be used to specify the names of the sideset behind the crack tip and the nodeset on the crack tip, respectively.

By default, the direction of crack propagation is computed from the geometry of the crack plane and tip as provided in the crack plane sideset and crack tip nodeset ($\boldsymbol{L} = \boldsymbol{N}$). The `CRACK DIRECTION` command can optionally be used to override the direction of crack propagation ($\boldsymbol{L}$) computed from the geometry. This command takes three real numbers that define the three components of the crack direction vector as arguments.

To fully define the domains used for the domain integrals, the radius of the domains and the number of domains must also be specified. A series of disc-shaped integration domains are formed with varying radii going out from the crack tip. The `INTEGRATION RADIUS` command is used to specify the radius of the outermost domain. The number of integration domains is specified using the `NUMBER OF DOMAINS` command. The radii of the domains increase linearly going from the innermost to the outermost domain.

The weight function $\boldsymbol{q}$ used to calculate the J-integral is specified by use of the option `FUNCTION` command line. The `LINEAR` function set the weight function to 1.0 on the crack front Γ_0 and 0.0 at the edge of the domain C_0, `int_radius` away from the crack tip. The `PLATEAU` function, which is the default behavior, sets all values of the weight function to 1.0 that lie within the domain of integration and all values outside of the domain are set to 0.0. This allows for integration over a single ring of elements at the edge of the domain. The third option for the `FUNCTION` command is `PLATEAU_RAMP`, which for a single domain will take on the same values as the `LINEAR` function. However, when there are multiple domains over the radius `int_radius`, the n^{th} domain will have weight function values of 1.0 over the inner (n-1) domains and will vary from 1.0 to 0.0 over the outer n^{th} ring of the domain. These functions can be seen graphically in Figure 9.2.

We note that in employing both the `PLATEAU` and the `PLATEAU_RAMP` functions, one is effectively taking a line integral at finite radius (albeit different radii). In contrast, the `LINEAR` option can be viewed as taking the $lim\ \Gamma_0 \to 0^+$. If the model is a half symmetry model with the symmetry plane on the plane of the crack, the optional `SYMMETRY` command can be used to specify that the symmetry conditions be accounted for in the formation of the integration domains and in the evaluation of the integral. The default behavior is for symmetry to not be used.

The user may optionally specify the time periods during which the J-integral is computed. The `ACTIVE PERIODS` and `INACTIVE PERIODS` command lines are used for this purpose. See Sec-

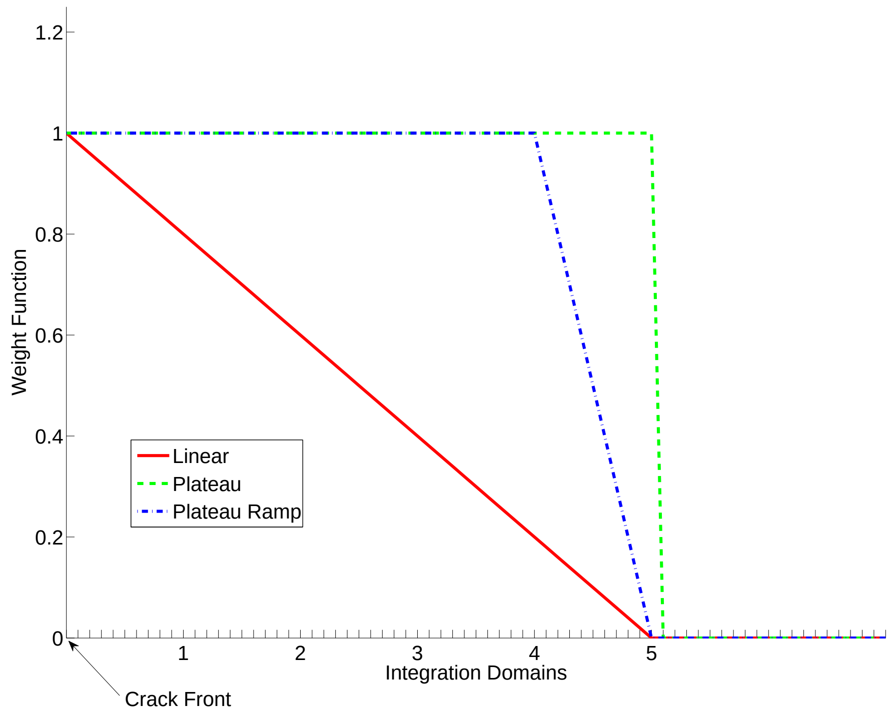


Figure 9.2: Example weight functions for a J-integral integration domain. Weight functions shown for domain 5.

tion 2.5 for more information about these command lines.

9.2.3 Output

A number of variables are generated for output when the computation of the J-integral is requested. The average value of J for each integration domain is available as a global variables, as described in Table 8.3. The pointwise value of J at nodes along the crack for each integration domain is available as a nodal variable, as shown in Table 8.8. Element variables such as the Eshelby energy-momentum tensor and fields defining the integration domains are also available, as listed in Table 8.16.

9.3 References

1. Eshelby, J. D., “The Force on an Elastic Singularity.” *Philosophical Transactions of the Royal Society of London* A244(1951): 87–112. doi.

2. Rice, J. R., “A Path Independent Integral and the Approximate Analysis of Stress Concentration by Notches and Cracks.” *Journal of Applied Mechanics* 35(1968): 379-386.

3. Eshelby, J. D., “Energy Relations and the Energy-Momentum Tensor in Continuum Mechanics.” *Inelastic Behavior of Solids* New York: McGraw-Hill, 1970.

4. Maugin, G. A., *Material Inhomogeneities in Elasticity* New York: Chapman & Hall/CRC, 1993.

5. Li, F. Z., C. F. Shih, and A. Needleman. “A Comparison of Methods for Calculating Energy Release Rates.” *Engineering Fracture Mechanics* 21(1985): 405–421. doi.

6. Shih, C. F., B. Moran, and T. Nakamura. “Energy release rate along a three-dimensional crack front in a thermally stressed body.” *International Journal of Fracture* 30 (1986): 79–102.

7. HKS. *ABAQUS Version 6.7, Theory Manual*. Providence, RI: Hibbitt, Karlsson and Sorensen, 2007.

Chapter 10

User Subroutines

User-defined subroutines is a functionality shared by Adagio and Presto. This chapter discusses when and how to use user-defined subroutines. There are examples of user-defined subroutines in the latter part of this chapter. Some of the examples are code specific, i.e., they are applicable to Presto rather than Adagio or vice versa. All examples, regardless of their applicability, do provide important information about how to use the command options available for user-defined subroutines.

In the introductory part of Chapter 10, we first describe, in general, possible applications for the user subroutine functionality in Adagio. 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 10.1 describes the details of the user subroutine. Section 10.2 describes the command lines associated with user subroutines that appear in the Adagio input file. In Section 10.3, we explain how to build and use a version of Adagio that incorporates your user subroutine. Finally, Section 10.4 provides examples of actual user subroutines, and Section 10.5 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 Adagio capability (boundary conditions, , 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 Adagio 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:

- 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 Adagio. 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 Adagio, you will then need to write the subroutine. The parts of the subroutine that interface to Adagio 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 Adagio input file
   - Access a variety of information—field variables, analysis time, etc.—from Adagio
   - Store computed quantities

   Parameters are values they may be passed from the Adagio 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 Adagio 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 Adagio 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

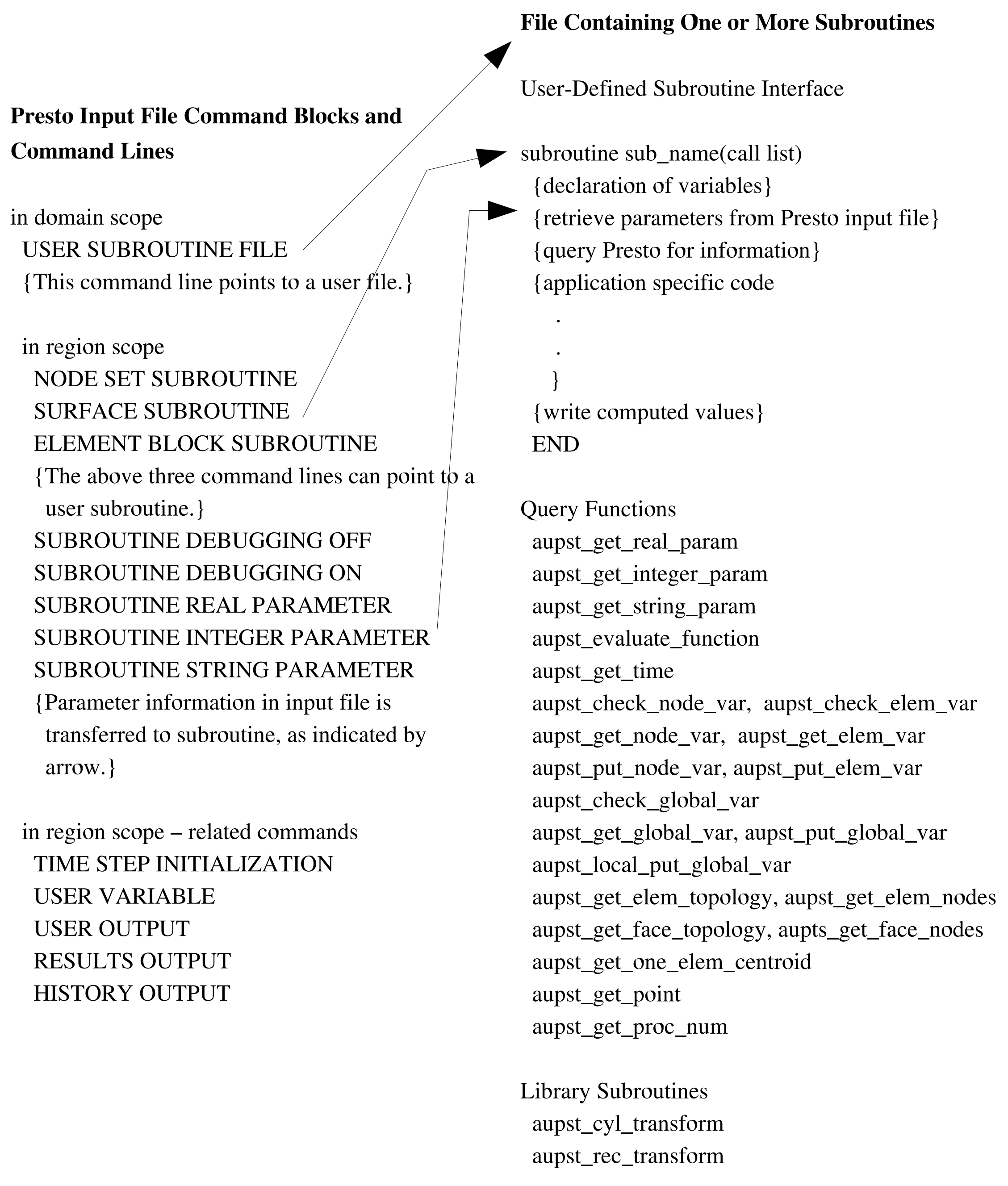
   ```
   NODE SET SUBROUTINE = <string>subroutine_name
   ```

   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 Adagio input file, you can build an executable version of Adagio that will run your user subroutine. Your Adagio run will then incorporate the functionality you have created in your user subroutine.

Figure 10.1 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 Adagio input file and the actual user subroutine, are represented by the two columns in Figure 10.1.



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 10.1: Overview of components required to implement user subroutine functionality, excluding compilation and execution commands.

10.1 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 Adagio 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 Adagio 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 Adagio input file}
{query Adagio 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 Adagio input file for parameters. Additional Adagio information such as field variables or element topology can then be retrieved from Adagio. 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 10.1.1 through Section 10.1.3 describe in detail the format for the interfaces to Adagio that will allow the user to make the subroutine call, retrieve information from Adagio, and write computed values. In these sections, mesh entities can be a node, an element face, or an element.

10.1.1 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 10.1 and Table 10.2.

Table 10.1: 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.
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.

10.1.2 Query Functions

Adagio 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 Adagio.

Table 10.2: 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, Adagio will report the error code and terminate the analysis.

10.1.2.1 Parameter Query

A number of user subroutine parameters may be set as described in Section 10.2.2.3. These subroutine parameters can be obtained from the Adagio 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 Adagio input file. The parameter command lines are described in Section 10.2.2.3. 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 10.3, Table 10.4, and Table 10.5. 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 10.3: 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 Adagio 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 10.4: 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 Adagio 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 10.5: 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 Adagio 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.

10.1.2.2 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 Adagio input file. This query allows the user to directly access information stored in a function defined in the Adagio 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 10.6.

Table 10.6: 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.

10.1.2.3 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 10.1.1, Table 10.1) 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 10.7.

```
aupst_get_time(real time)
```

Table 10.7: aupst_get_time Argument

Parameter	Usage	Data Type	Description
time	Output	Real	Current analysis time.

10.1.2.4 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 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_get_elem_var_offset(int num_elems, int num_components,
                          int offset, 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)

aupst_put_elem_var_offset(int num_elems, int num_components,
                          int offset, 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 Adagio does not exceed the size of the array allocated by the user.

The arguments to field query functions are defined in Table 10.8 through Table 10.15. 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 10.8: aupst_check_node_var Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_nodes	Input	Integer	Number of nodes used to extract field information.
num_components	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 defined Adagio 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 10.9: aupst_check_elem_var Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_elems	Input	Integer	Number of elements used to extract field information.
num_components	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 defined Adagio 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 10.10: aupst_get_node_var Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_nodes	Input	Integer	Number of nodes used to extract field information.
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.
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 defined Adagio 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 10.11: aupst_get_elem_var Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_elems	Input	Integer	Number of elements used to extract field information.
num_components	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 × num_elems containing the field data for each element.
var_name	Input	String	Name of the field variable. The field variable must be a defined Adagio 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_offset(int num_elems, int num_components,
                          int offset, int elem_ids[],
                          real return_data[], string var_name,
                          int error_code)
```

Table 10.12: aupst_get_elem_var_offset Arguments

Parameter	Usage	Data Type	Description
num_elems	Input	Integer	Number of elements used to extract field information.
num_components	Input	Integer	Number of components in the field information. A stress field for an eight-node hexahedron element has six components, for example.
offset	Input	Integer	Offset into var_name field variable at which to get data.
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 defined Adagio 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 10.13: 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_components, 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 defined Adagio 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 10.14: 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_components	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_components, 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 defined Adagio 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_offset(int num_elems, int num_components,
                          int offset, int elem_ids[],
                          real new_data[], string var_name,
                          int error_code)
```

Table 10.15: aupst_put_elem_var_offset Arguments

Parameter	**Usage**	**Data Type**	**Description**
num_elems	Input	Integer	Number of elements for which the user will specify the field data.
num_components	Input	Integer	Number of components in the field information. A stress field for an eight-node hexahedron element has six components, for example.
offset	Input	Integer	Offset into var_name field variable at which to put data.
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_components, num_elems)	Input	Real	Array of size num_components × num_elems containing the new data for the field.
var_name	Input	String	Name of the field variable. The field variable must be a defined Adagio 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.

10.1.2.5 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 8.2.2).

Only user-defined global variables may be modified by the user subroutine (see Section 10.2.4). 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 10.16 through Table 10.19. The call is repeated before each table for easy reference.

```
aupst_check_global_var(int num_comp, string var_name
                       int error_code)
```

Table 10.16: 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 10.17: 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 10.18: 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 10.19: 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 Adagio 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.

10.1.2.6 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 10.20: Topologies Used by Adagio

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

Table 10.20 lists the topologies currently in use by Adagio.

The following topology query functions are available in Adagio:

```
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 10.21 through Table 10.24. 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 10.21: 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 10.22: 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 10.23: 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 10.24: 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.

10.1.3 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 10.25 through Table 10.27. 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 10.25: 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 × 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 10.26: 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 10.27: 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.

10.2 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.

10.2.1 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 SIERRA 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.

10.2.2 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 Adagio command block. User subroutine commands can appear in boundary condition, 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.

10.2.2.1 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 Adagio features may accept one or more types of user subroutines. Only one subroutine is allowed per command block.

10.2.2.2 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 Adagio to crash, but it also has the potential to introduce other very hard-to-trace bugs into the Adagio 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
```

10.2.2.3 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.

10.2.3 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
  INACTIVE 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` or `INACTIVE PERIODS`. Following are descriptions of the different command groups and the `ACTIVE PERIODS` or `INACTIVE PERIODS` command line.

10.2.3.1 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.

10.2.3.2 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 10.2.2 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 10.

10.2.3.3 Additional Command

The `ACTIVE PERIODS` or `INACTIVE PERIODS` command line can optionally appear in the `TIME STEP INITIALIZATION` command block:

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

The `ACTIVE PERIODS` or `INACTIVE 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. See Section 2.5 for more information about this optional command line.

10.2.4 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
  INITIAL VARIATION = <real list>values LINEAR DISTRIBUTION
  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 other defined 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.

Known Issue: User defined variables are not currently supported with heartbeat output (see Section 8.4).

The `USER VARIABLE` command block is placed within a Adagio 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 Adagio 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 10.1.2.5) 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.

- The initial value of the user variable can be given some random distribution by use of the the `INITIAL VARIATION` command line. If the `INITIAL VARIATION` command is used the `INITIAL VALUE` command must also be used. In addition the number of values specified in the initial variation command must exactly match the number of values specified in the initial value command. When the initial variation command is used the initial values of the variable will be set as initial value plus or minus some random factor times the initial variation. Currently the only random distribution supported is the linear distribution. With linear distribution the random values will be distributed evenly from initial value minus variation to initial value plus variation.

- 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.

10.3 User Subroutines: Compilation and Execution

Running a code with user subroutines is a two-step process. First, you must create an executable version of Adagio that recognizes the user subroutines. Next, you must use this version of Adagio for an actual Adagio 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 Adagio 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 Adagio input file that makes use of the user subroutine. You will use a system command line of the general form shown below.

```
% sierra adagio -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 SIERRA scope:

```
USER SUBROUTINE FILE = blast_load.F
```

You will also have some subset of the command lines described in the previous section in your Adagio input file. The specific form of the system command line to execute the first step of the user subroutine process is shown below.

`% sierra adagio -i blast_load_1.i --make` The above command will create a local version of Adagio in a local directory named `UserSubsProject`. The system command line to run the local version of Presto is shown below.

```
% sierra adagio -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 adagio -i blast_load_1.i -x UserSubsProject
```

The second command line runs Adagio 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 Adagio is set up at your installation to determine what system command lines are necessary to build Adagio with user subroutines and then use this version of Adagio.

10.4 User Subroutines: Examples

10.4.1 Pressure as a Function of Space and Time

(The following example provides functionality—a blast load on a surface—more applicable to Presto than Adagio. It is included in both manuals as it is instructive in the general use of a user-defined subroutine.)

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 input file as follows:

```
#In the SIERRA 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 10.1.2.6) 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
```

10.4.2 Error Between a Computed and an Analytic Solution

The following code is a user subroutine to compute the error between Adagio-computed results and results from an analytic manufactured solution. 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 Adagio 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 Adagio 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,*) ŠERROR in sphere disp, Ś,
     &  num_nodes exceeds static array sizeŠ
        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
```

10.4.3 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
```

10.5 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 Adagio command lines.) This section describes the usage of each of these subroutines.

10.5.1 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 variable for each element. The variable is created by the following command block in the Adagio 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 Adagio internal input stress tensor variable.
output_stress	String	Name of the Adagio internal output stress tensor variable.

10.5.2 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 variable for each element. The variable is created by the following command block in the Adagio 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 Adagio internal input stress tensor variable.
output_stress	String	Name of the Adagio internal output stress tensor variable.

10.5.3 copy_data

Author: Jason Hales

Purpose:

The purpose of this subroutine is to copy data from one variable to another with offsets given for both variables. This subroutine is generally called by a `USER OUTPUT` command block. For example:

```
begin user output
  block = block_1
  element block subroutine = copy_data
  subroutine integer parameter: source_offset = 4
  subroutine string parameter:  source_name = stress
  subroutine integer parameter: destination_offset = 1
  subroutine string parameter:  destination_name = uservarxy
end user output
```

Requirements:

This subroutine requires that the source and destination fields exist and have lengths at least as great as the values supplied as offsets. The fields used may be defined by the user as variables. In this example, the variable is created by the following command block in the Adagio region:

```
begin user variable uservarxy
  type = element real length = 1
  initial value = 0.0
end user variable
```

Parameters:

Parameter Name	Usage	Description
source_offset	Integer	The offset into the source variable.
source_name	String	The name of the source variable.
destination_offset	Integer	The offset into the destination variable.
destination_name	String	The name of the destination variable.

10.5.4 trace

Author: Jason Hales

Purpose:

The purpose of this subroutine is to compute the trace of a tensor. This subroutine is generally called by a `USER OUTPUT` command block. For example:

```
begin user output
  block = block_1
  element block subroutine = trace
  subroutine string parameter:  source_name = log_strain
  subroutine string parameter:  destination_name = uvarbulkstrain
end user output
```

Requirements:

This subroutine requires that the source and destination fields exist. The source field should have a length of six. The destination field should have a length of one. The destination field will typically be defined by the user as a variable. In this example, the variable is created by the following command block in the Adagio region:

```
begin user variable uvarbulkstrain
  type = element real length = 1
  initial value = 0.0
end user variable
```

Parameters:

Parameter Name	Usage	Description
source_name	String	The name of the source variable.
destination_name	String	The name of the destination variable.

Chapter 11

Transfers

It is sometimes desirable to chain two or more analyses (procedures) together. A common example of this is the need to preload an assembly quasistatically and then subject that assembly to a loading environment best suited to an explicit transient dynamics analysis. The displacements and stresses produced by the quasistatic preload are initial conditions for the transient dynamics event. These displacements and stresses must be transferred from the initial analysis to the subsequent one.

This chapter reviews the concept of transfers in SIERRA and outlines the syntax required to perform a transfer of information between procedures.

11.1 SIERRA Transfers

Applications built on the SIERRA computational framework share underlying data structures. This makes it convenient to couple applications together using transfers.

The coupling available through SIERRA is of two types. The first is what is called intra-procedural coupling. In this case, multiple regions within a single procedure share data. This enables multi-physics analysis such as thermal-mechanical coupling. The details of this type of coupling, along with the syntax to support it, will not be covered here. Coupled codes such as Calagio and Arpeggio use this type of coupling.

The second type of coupling is inter-procedural coupling. Here, the result from one procedure is handed to the next procedure. This is the type of coupling used when moving from one analysis stage to another in Adagio and Presto. When using this type of coupling, the two procedures generally have only one region each.

11.2 Inter-procedural Transfers

The inter-procedural transfers used by Adagio and Presto can transfer data from one or all blocks of the preceding or sending procedure to the subsequent or receiving procedure. The commands to control the transfers should appear at the procedure scope in the second procedure.

When using the inter-procedural transfers, all of the appropriate element data will be transferred from the sending to the receiving elements. Nodal data will be transferred based on the type of analysis done in the two procedures. For example, if the first procedure is a quasistatic analysis and the second is an explicit transient dynamics analysis, the displacements of the nodes will be transferred but not their velocities.

Warning: Transfer of data for node-based tetrahedra is currently not fully supported. A coupled analysis with node-based tetrahedra will only be correct if the tet elements are not deformed in the first (sending) procedure, in which case initialization in the second (receiving) procedure is appropriate.

The set of available commands are below.

```
BEGIN PROCEDURAL TRANSFER <string>name

  BLOCK = <string list>block_name
  INCLUDE ALL BLOCKS
  REMOVE BLOCK = <string list>block_name

  BEGIN INTERPOLATION TRANSFER <string>name

    BLOCK BY BLOCK
    NEAREST ELEMENT COPY
    SEND BLOCKS = <string list>block_name
    SEND COORDINATES = ORIGINAL|CURRENT

    RECEIVE BLOCKS = <string list>block_name
    RECEIVE COORDINATES = original|current

    TRANSFORMATION TYPE = NONE|RIGIDBODY

  END [INTERPOLATION TRANSFER <string>name]
END [PROCEDURAL TRANSFER <string>name]
```

The inter-procedural transfers can be invoked in one of two ways. If the sending and receiving regions use the same finite element model, data can be copied from the sending to the receiving region. In this case, the first three lines of syntax in the transfer block can be used to copy data for all blocks except those that are rigid bodies.

If different finite element models are used by the sending and receiving meshes, or if it is desired to copy data for rigid bodies, the `INTERPOLATION TRANSFER` block must be used, and the first three lines in the `PROCEDURAL TRANSFER` block should not be used. If the lines appropriate for the copy of data are used when they do not apply, their behavior is undefined.

11.2.1 Copying Data with Inter-procedural Transfers

To copy data from one or more blocks in a sending region to the matching blocks in the receiving region, use these line commands:

```
  BLOCK = <string list>block_name
  INCLUDE ALL BLOCKS
  REMOVE BLOCK = <string list>block_name
```

The first of these, the `BLOCK` line command, is used to list blocks that should be copied. If it is desired to copy data for all blocks, use the `INCLUDE ALL BLOCKS` line command. Finally, one or more blocks may be removed from the set of all blocks with the `REMOVE BLOCK` line command. Either the `BLOCK` or `INCLUDE ALL BLOCKS` line command must appear.

11.2.2 Interpolating Data with Interpolation Transfers

When different finite element models are used by the sending and receiving regions, or when it is desired to transfer data for rigid bodies, the `INTERPOLATION TRANSFER` must be used.

If data for a rigid body is to be transferred, use the `TRANSFORMATION TYPE = RIGIDBODY` line command along with the `SEND BLOCKS` and `RECEIVE BLOCKS` line commands. Transfers of rigid body information should be entered for each rigid body separately, which is done simply by listing only one block on each of the `SEND BLOCKS` and `RECEIVE BLOCKS` lines.

When transferring data for non-rigid body blocks, the only required line commands are the `SEND BLOCKS` and `RECEIVE BLOCKS` line commands. These each list one or more blocks to be included in the transfer.

The interpolation transfers move data from sending to receiving meshes by performing searches and interpolating using shape functions. For nodal data, a node in the receiving mesh is located in the sending mesh. The node will be found in (or near) an element in the sending mesh. The parametric coordinates of the point associated with the receiving node will be calculated based on the sending element, and those parametric coordinates will be used in conjunction with the shape functions and the data on the nodes of the sending element to calculate values for the receiving node.

For the transfer of element data, the location of the center of the receiving element will be located in the sending mesh. This point will appear in (or near) an element in the sending mesh. A patch of elements in the sending mesh will be created centered around the element that contains the point associated with the receiving element. The data in the patch of sending elements will be interpolated using a least squares approach to the point associated with the receiving element, and the result will be given to the receiving element.

The interpolation transfer will give the best results when the sending and receiving meshes are very similar. If the meshes do not represent the same volumes in space, for example, the interpolated values will be suspect at best.

On rare occasions, it may be desired to use the current coordinates instead of the original coordinates in performing the search used by the transfers. Use the `SEND COORDINATES = CURRENT` and/or `RECEIVE COORDINATES = CURRENT` line commands for this purpose.

In some instances, the sending and receiving meshes are very similar such that there is a one-to-one correspondence between the list of sending blocks and the list of receiving blocks. In other words, it may be desired to send data from a given block to a corresponding block, from another block to its pair, and so forth. If this is the case, use of the `BLOCK BY BLOCK` line command will cause a separate transfer to be created for each pair of sending and receiving blocks. This is useful to ensure that data from a single block will be sent to one and only one receiving block. This could also be accomplished by listing multiple transfer blocks in the input file.

The `NEAREST ELEMENT COPY` line command changes the behavior of the transfer of element variables. Instead of a least squares approach, the use of this line command will cause data to be sent directly from the nearest sending element to the receiving element.

Appendix A

Example Problem

This appendix provides an example problem to illustrate the construction of an input file for an analysis. The problem is modeled after a pencil/eraser that is pressed against and rubbed across a tablet. The pencil, eraser and tablet are represented by blocks 1, 2 and 3 respectively. Block 4 is used to apply kinematics directly to the eraser via tied contact between block 4 and block 2. Block 1 is really superfluous except that when viewed together, blocks 1, 2 and 3 closely resemble a pencil with an eraser on a tablet. The problem demonstrates the overall structure of an input file and includes the use of both tied and frictional contact simultaneously, the multilevel solver, and a linear solver for preconditioning. A schematic of the problem is shown in Figure A.1 and the mesh is shown in Figure A.2.

The problem kinematics and loading are described briefly here. There are two phases of the loading: preload and sliding. In the preload phase, the tablet (block 3) is kinematically fixed in the x-y plane and a vertical load (in the z-direction) is applied to the tablet which presses the tablet against the eraser (block 2). The eraser is kinematically fixed to block 4 via tied contact (see Figure A.2). Block 4 is held fixed in all three coordinate directions for the compression phase. The initial con-

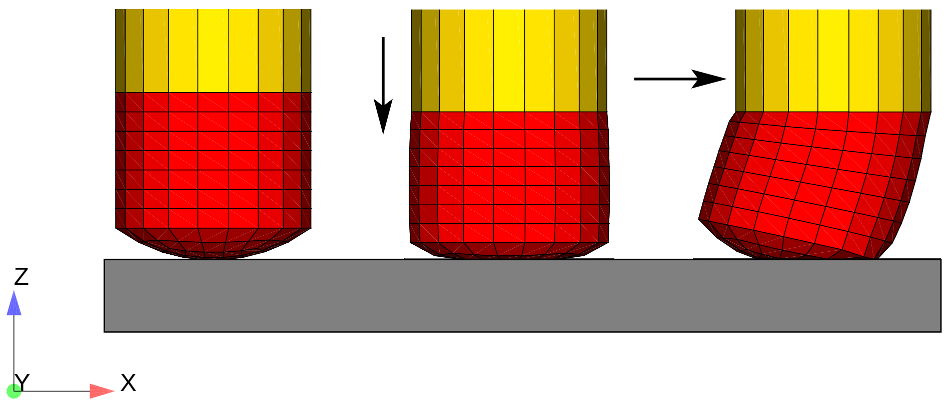


Figure A.1: Eraser schematic; Block 1 (yellow) is pencil, block 2 (red) is eraser, block 3 (gray) is tablet, block 4 (not shown) is tied to eraser and has the kinematics applied to it.

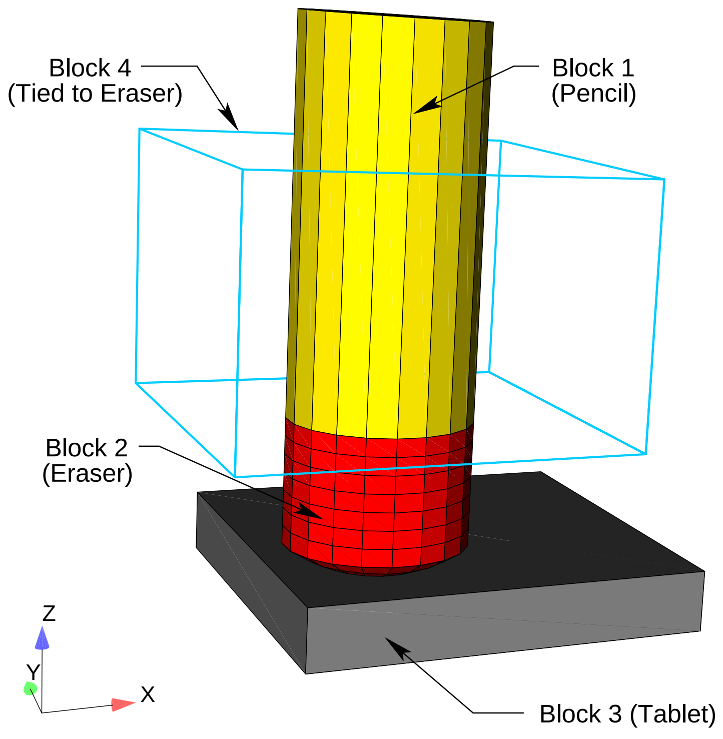


Figure A.2: Complete eraser mesh

figuration and deformations of the preload phase are shown in the first two snapshots on the left in Figure A.1. In the sliding phase, block 4 is kinematically prescribed to move along the *x*-direction, thus sliding or dragging the eraser along the tablet (see Figure A.1) while the tablet force is held constant.

The input file is described below, with comments to explain every few lines. Following the description, the full input file is listed again. Note that all character strings in the input file are presented in lowercase, which is an acceptable format in Adagio.

The input file starts with a begin sierra statement, as is required for all input files:

```
begin sierra eraser
```

We begin by defining vectors corresponding to the coordinate axes. These vectors/directions will be used to define the input for boundary condition blocks that come later.

```
  define direction X with vector 1.0 0.0 0.0
  define direction Y with vector 0.0 1.0 0.0
  define direction Z with vector 0.0 0.0 1.0
```

We now need to define the functions used for this problem. The boundary conditions require a function for the tablet force as well as the sliding motion. Note that both the tablet force and sliding motion are prescribed as functions of time.

```
begin definition for function slide
  type is piecewise linear
  begin values
    0.0      0.0
    1.0      0.0
    2.0      1.0
    3.0      1.0
  end values
end definition for function slide

begin definition for function tablet_force
  type is piecewise linear
  begin values
    0.0      0.0
    1.0      1.0
    3.0      1.0
  end values
end definition for function tablet_force

begin definition for function zero
  type is constant
  begin values
    0.0      0.0
  end values
end definition for function zero
```

Note that the tablet force ramps up over the time interval (0.0,1.0) and then is held constant over the interval (1.0,3.0). The sliding function is zero over the time interval (0.0,1.0) and then it linearly increases over the interval (1.0,2.0). Next, we define the material properties and models that are used in this problem. In this example, we define two sets of material properties: one is called stiff (pencil, tablet, and block 4), and the second is called soft (eraser). We use a linear elastic constitutive model for both cases.

```
begin property specification for material stiff
  density = 1.0
  begin parameters for model elastic
    youngs modulus = 1.e5
    poissons ratio = 0.3
  end parameters for model elastic
end property specification for material stiff

begin property specification for material soft
  density = 1.0
  begin parameters for model elastic
    youngs modulus = 1000.
    poissons ratio = 0.3
  end parameters for model elastic
```

```
        end property specification for material soft
```

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 `eraser.g`. The specification of the database type is optional-ExodusII is the default. Currently, each element block must be defined individually. Note that the tablet, pencil and block 4 all reference the same material description (stiff). The material description is not repeated three times. The material description for stiff appears once and is then referenced three times.

```
        begin finite element model mesh1
          database name = eraser.g
          database type = exodusII
          begin parameters for block block_1 #Pencil
            material stiff
            solid mechanics use model elastic
          end parameters for block block_1 #Pencil
          begin parameters for block block_2 #Eraser
            material soft
            solid mechanics use model elastic
          end parameters for block block_2 #Eraser
          begin parameters for block block_3 #Tablet
            material stiff
            solid mechanics use model elastic
          end parameters for block block_3 #Tablet
          begin parameters for block block_4 #dummy block
            material stiff
            solid mechanics use model elastic
          end parameters for block block_4 #dummy block
        end finite element model mesh1
```

At this point we have finished specifying physics-independent quantities. We now want to set up the Adagio 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. Three time stepping command blocks are used in this analysis. The termination time is set to 1.8. Having multiple time blocks is useful because we can make some solver options a function of the time block.

```
        begin adagio procedure agio_procedure
          begin time control
            begin time stepping block preload
              start time = 0.0
              begin parameters for adagio region adagio
                number of time steps = 5
              end parameters for adagio region adagio
            end time stepping block preload
```

```
    begin time stepping block slide_1
      start time = 1.0
      begin parameters for adagio region adagio
        number of time steps = 1
      end parameters for adagio region adagio
    end time stepping block slide_1
    begin time stepping block slide_2
      start time = 1.1
      begin parameters for adagio region adagio
        number of time steps = 7
      end parameters for adagio region adagio
    end time stepping block slide_2
    termination time = 1.8
  end time control

  begin adagio region agio_region
```

Next we associate the finite element model we defined above (`mesh1`) with this Adagio region.

```
    use finite element model mesh1
```

Now we define the kinematic boundary conditions. First, we prescribe the displacement of surface_200 which is part of block 4. We fix this surface in both the *x* and *z* directions and prescribe the horizontal displacement along the X-direction using the previously defined functions `zero` and `slide`.

```
    ### movement of pencil prescribed by block 4
    begin prescribed displacement
      surface = surface_200
      direction = X
      function = slide
      scale factor = 3.0
    end prescribed displacement
    begin fixed displacement
      surface = surface_200
      components = Y Z
    end fixed displacement
```

Next, we fix the tablet and prevent it from moving in the X-Y plane. We do this on all tablet nodes (nodelist_111 and nodelist_112).

```
    ### Constraints on tablet
    begin fixed displacement
      node set = nodelist_111
      components = X Y
```

```
end fixed displacement
begin fixed displacement
  node set = nodelist_112
  components = X Y
end fixed displacement
```

Finally, we prescribe preload force on the tablet which compresses the tablet against the eraser.

```
### Tablet force
begin prescribed force
  node set = nodelist_111
  direction = Z
  function = tablet_force
  scale factor = 100.0
end prescribed force
```

We now define the contact for this problem. Here we need to define tied contact between block 4 and the eraser. In addition, we have frictional sliding contact between the tablet and the eraser. Two contact block definitions are required; one for the tied contact and one for the frictional contact. The first contact block definition is used for the tied contact between block 4 and the eraser, and the second block is used to define the frictional sliding contact between the eraser and the tablet.

```
### block 4 tied to eraser ###
begin contact definition
  enforcement = tied
  contact surface surf_200 contains surface_200
  contact surface surf_110 contains surface_110
  begin interaction
    master = surf_200
    slave = surf_110
    normal tolerance = 1.0
    tangential tolerance = 0.5e-3
  end interaction
end contact definition

begin contact definition
  enforcement = frictional
  contact surface surf_11 contains surface_11
  contact surface surf_10 contains surface_10
  begin interaction
    master = surf_11
    slave = surf_10
    normal tolerance = 1.0
    tangential tolerance = 0.5e-3
    capture tolerance = 1.0e-2
    friction coefficient = 0.8
  end interaction
end contact definition
```

Now we define what variables we want in the output file, as well as how often we want the output file to be written. The output file will be called eraser.e, and it will be an ExodusII file (the database type command is optional; it defaults to ExodusII). The variables we are requesting are the displacements, velocities, and contact diagnostics at the nodes, and stresses and strains at the elements.

```
begin results output output_adagio
  database name = eraser.e
  database type = exodusII
  at step 0, increment = 1
  nodal displacement as displ
  nodal velocity as vel
  nodal contact_tangential_direction
  nodal contact_normal_direction
  nodal contact_accumulated_slip_vector
  nodal contact_status
  nodal contact_normal_traction_magnitude
  nodal contact_tangential_traction_magnitude
  nodal contact_incremental_slip_magnitude
  nodal contact_accumulated_slip
  nodal contact_frictional_energy_density
  nodal contact_area
  element stress as stress
  element log_strain as strain
end results output output_adagio
```

The final part of the input deck is the Adagio solver commands. Because we have sliding contact, we must use the solver command block. Nested inside the solver block we define the control contact block as well as the nonlinear cg solver block.

```
begin solver
  begin control contact name
    target relative residual = 0.001
    maximum iterations = 2000
  end control contact name
  begin cg
    target residual tolerance = 0.001
    maximum iterations = 2000
    minimum iterations = 1
    orthogonality measure for reset = 0.5
    line search type secant
    begin full tangent preconditioner
      linear solver = feti
```

```
          maximum iterations = 20
          constraint enforcement = penalty
          reset constraint threshold = 0.001
          nodal preconditioner method = elastic
        end full tangent preconditioner
      end cg
    end solver
```

Now we have defined the Adagio region and procedure blocks and so we close them using the following lines:

```
      end adagio region agio_region
    end adagio procedure agio_procedure
```

The final thing that we need to define for this file is the linear solver. In the above solver command block, we included the `full tangent preconditioner` block, which has a nested command line `linear solver = feti`. This command line refers to a linear solver called `feti` that must be defined outside the "Procedure" scope but within the "Sierra" scope and can come at the top of the file prior to the "Procedure" definition or after it as is the case here. The input `feti` on this line command is a user defined string that can have any useful label. The following command block defines the linear solver labeled `feti`. Note that the command block has the label `feti` at the end of the `Begin` line and that this label is referred to from within the above `full tangent preconditioner` command block. The FETI parameters are all set to reasonable values for most problems, so there is no need to set any of them, but if it were necessary to set any of them to non-default values, that would be done within the `feti equation solver` block.

```
  begin feti equation solver feti
  end feti equation solver feti
```

Finally, we finish the input file and close the sierra command block:

```
end sierra eraser
```

Appendix B

Command Summary

This appendix gives all of the Adagio commands in the proper scope.

```
# SIERRA scope 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|PIECEWISE CONSTANT|
      ANALYTIC
    ABSCISSA = <string>abscissa_label
      [scale = <real>abscissa_scale(1.0)]
      [offset = <real>abscissa_offset(0.0)]
    ORDINATE = <string>ordinate_label
      [scale = <real>ordinate_scale(1.0)]
      [offset = <real>ordinate_offset(0.0)]
    X SCALE = <real>x_scale(1.0)
    X OFFSET = <real>x_offset(0.0)
    Y SCALE = <real>y_scale(1.0)
    Y OFFSET = <real>y_offset(0.0)
    BEGIN VALUES
      <real>x_1   <real>y_1
      <real>x_2   <real>y_2
      ...
      <real>x_n   <real>y_n
    END [VALUES]
    AT DISCONTINUITY EVALUATE TO <string>LEFT|RIGHT(LEFT)
    EVALUATE EXPRESSION = <string>analytic_expression1;
      analytic_expression2;...
    DEBUG = ON|OFF(OFF)
  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]

  # Rigid bodies

  BEGIN RIGID BODY <string>rb_name
    MASS = <real>mass
    POINT MASS = <real>mass [AT <real>X <real>Y <real>Z]
    REFERENCE LOCATION = <real>X <real>Y <real>Z
    INERTIA = <real>Ixx <real>Iyy <real>Izz <real>Ixy
      <real>Iyz <real>Izx
    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
  INCLUDE NODES IN <string>surface_name
    [if <string>field_name <|<=|=|>=|> <real>value]
END [RIGID BODY <string>rb_name]

# Elastic material

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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]

# Thermoelastic material

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 THERMOELASTIC
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    YOUNGS MODULUS FUNCTION = <string>ym_function_name
    POISSONS RATIO FUNCTION = <string>pr_function_name
  END [PARAMETERS FOR MODEL THERMOELASTIC]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# neo-Hookean material

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 NEO_HOOKEAN
    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 NEO_HOOKEAN]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Elastic fracture material

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 CRACK OPENING 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
  BIOTS COEFFICIENT = <real>biots_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
  BIOTS COEFFICIENT = <real>biots_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
  BIOTS COEFFICIENT = <real>biots_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>critical_strain
  END [PARAMETERS FOR MODEL DUCTILE_FRACTURE]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Multilinear elastic plastic

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 MULTILINEAR_EP
    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 = <string>hardening_function_name
    YOUNGS MODULUS FUNCTION = <string>ym_function_name
    POISSONS RATIO FUNCTION = <string>pr_function_name
    YIELD STRESS FUNCTION =
      <string>yield_stress_function_name
  END [PARAMETERS FOR MODEL MULTILINEAR_EP]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Multilinear elastic plastic with failure
```

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 = <string>hardening_function_name
    YOUNGS MODULUS FUNCTION = <string>ym_function_name
    POISSONS RATIO FUNCTION = <string>pr_function_name
    YIELD STRESS FUNCTION =
      <string>yield_stress_function_name
    CRITICAL TEARING PARAMETER = <real>crit_tearing
    CRITICAL CRACK OPENING STRAIN = <real>critical_strain
  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
  BIOTS COEFFICIENT = <real>biots_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]

# Power law creep
```

```
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 POWER_LAW_CREEP
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    CREEP CONSTANT = <real>creep_constant
    CREEP EXPONENT = <real>creep_exponent
    THERMAL CONSTANT = <real>thermal_constant
  END [PARAMETERS FOR MODEL POWER_LAW_CREEP]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Soil and crushable foam

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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
  BIOTS COEFFICIENT = <real>biots_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]

# Low density foam

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 LOW_DENSITY_FOAM
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    A = <real>A
    B = <real>B
    C = <real>C
    NAIR  = <real>NAir
    P0  = <real>P0
    PHI = <real>Phi
  END [PARAMETERS FOR MODEL LOW_DENSITY_FOAM]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Elastic three-dimensional orthotropic

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_value
  BEGIN PARAMETERS FOR MODEL ELASTIC_3D_ORTHOTROPIC
    # general parameters (any two are required)
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    # required parameters
    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]

# Wire mesh

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 WIRE_MESH
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    YIELD FUNCTION = <string>yield_function
    TENSION = <real>tensile_strength
  END [PARAMETERS FOR MODEL WIRE_MESH]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Orthotropic crush

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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
  BIOTS COEFFICIENT = <real>biots_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]

# 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]

# Fiber membrane

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 FIBER_MEMBRANE
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    CORD DENSITY = <real>cord_density
    CORD DIAMETER = <real>cord_diameter
    MATRIX DENSITY = <real>matrix_density
    TENSILE TEST FUNCTION = <string>test_function_name
    PERCENT CONTINUUM = <real>percent_continuum
    EPL = <real>epl
    AXIS X = <real>axis_x
    AXIS Y = <real>axis_y
    AXIS Z = <real>axis_z
    MODEL = <string>RECTANGULAR
    STIFFNESS SCALE = <real>stiffness_scale
    REFERENCE STRAIN = <real>reference_strain
  END [PARAMETERS FOR MODEL FIBER_MEMBRANE]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Incompressible solid

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 INCOMPRESSIBLE_SOLID
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    BULK MODULUS = <real>bulk_modulus
    SHEAR MODULUS = <real>shear_modulus
    LAMBDA = <real>lambda
    K SCALING = <real>k_scaling
    2G SCALING = <real>2g_scaling
    TARGET E = <real>target_e
    MAX POISSONS RATIO = <real>max_poissons_ratio
    REFERENCE STRAIN = <real>reference_strain
    SCALING FUNCTION = <string>scaling_function_name
  END [PARAMETERS FOR MODEL INCOMPRESSIBLE_SOLID]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Mooney Rivlin

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_value
  #
  BEGIN PARAMETERS FOR MODEL MOONEY_RIVLIN
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    BULK MODULUS = <real>bulk_modulus
    SHEAR MODULUS = <real>shear_modulus
    LAMBDA = <real>lambda
    C10 = <real>c10
    C01 = <real>c01
    C10 FUNCTION = <string>c10_function_name
    C01 FUNCTION = <string>c01_function_name
    BULK FUNCTION = <string>bulk_function_name
    THERMAL EXPANSION FUNCTION = <string>eth_function_name
    TARGET E = <real>target_e
    TARGET E FUNCTION = <string>etar_function_name
    MAX POISSONS RATIO = <real>max_poissons_ratio
    REFERENCE STRAIN = <real>reference_strain
  END [PARAMETERS FOR MODEL MOONEY_RIVLIN]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# NVLE 3D Orthotropic

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_value
  #
  BEGIN PARAMETERS FOR MODEL NLVE_3D_ORTHOTROPIC
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    SHEAR MODULUS = <real>shear_modulus
    BULK MODULUS = <real>bulk_modulus
    LAMBDA = <real>lambda
    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
    FICTITIOUS LOGA FUNCTION = <string>fict_loga_function_name
    FICTITIOUS LOGA SCALE FACTOR = <real>fict_loga_scale_factor
    # In each of the five ‘‘PRONY’’ command lines and in
    # the RELAX TIME command line, the value of i can be from
    # 1 through 30
    1PSI PRONY <integer>i = <real>psi1_i
    2PSI PRONY <integer>i = <real>psi2_i
    3PSI PRONY <integer>i = <real>psi3_i
    4PSI PRONY <integer>i = <real>psi4_i
    5PSI PRONY <integer>i = <real>psi5_i
    RELAX TIME <integer>i = <real>tau_i
    REFERENCE TEMP = <real>tref
    REFERENCE DENSITY = <real>rhoref
    WLF C1 = <real>wlf_c1
    WLF C2 = <real>wlf_c2
    B SHIFT CONSTANT = <real>b_shift
    SHIFT REF VALUE = <real>shift_ref
    WWBETA 1PSI = <real>wwb_1psi
    WWTAU 1PSI = <real>wwt_1psi
    WWBETA 2PSI = <real>wwb_2psi
    WWTAU 2PSI = <real>wwt_2psi
    WWBETA 3PSI = <real>wwb_3psi
    WWTAU 3PSI = <real>wwt_3psi
    WWBETA 4PSI = <real>wwb_4psi
    WWTAU 4PSI = <real>wwt_4psi
    WWBETA 5PSI = <real>wwb_5psi
    WWTAU 5PSI = <real>wwt_5psi
    DOUBLE INTEG FACTOR = <real>dble_int_fac
    REF RUBBERY HCAPACITY = <real>hcapr
    REF GLASSY HCAPACITY = <real>hcapg
    GLASS TRANSITION TEM = <real>tg
    REF GLASSY C11 = <real>c11g
    REF RUBBERY C11 = <real>c11r
    REF GLASSY C22 = <real>c22g
    REF RUBBERY C22 = <real>c22r
    REF GLASSY C33 = <real>c33g
    REF RUBBERY C33 = <real>c33r
    REF GLASSY C12 = <real>c12g
    REF RUBBERY C12 = <real>c12r
    REF GLASSY C13 = <real>c13g
    REF RUBBERY C13 = <real>c13r
    REF GLASSY C23 = <real>c23g
    REF RUBBERY C23 = <real>c23r
    REF GLASSY C44 = <real>c44g
    REF RUBBERY C44 = <real>c44r
    REF GLASSY C55 = <real>c55g
    REF RUBBERY C55 = <real>c55r
    REF GLASSY C66 = <real>c66g
    REF RUBBERY C66 = <real>c66r
    REF GLASSY CTE1 = <real>cte1g
    REF RUBBERY CTE1 = <real>cte1r
    REF GLASSY CTE2 = <real>cte2g
    REF RUBBERY CTE2 = <real>cte2r
    REF GLASSY CTE3 = <real>cte3g
    REF RUBBERY CTE3 = <real>cte3r
    LINEAR VISCO TEST = <real>lvt
    T DERIV GLASSY C11 = <real>dc11gdT
    T DERIV RUBBERY C11 = <real>dc11rdT
    T DERIV GLASSY C22 = <real>dc22gdT
    T DERIV RUBBERY C22 = <real>dc22rdT
    T DERIV GLASSY C33 = <real>dc33gdT
    T DERIV RUBBERY C33 = <real>dc33rdT
    T DERIV GLASSY C12 = <real>dc12gdT
    T DERIV RUBBERY C12 = <real>dc12rdT
    T DERIV GLASSY C13 = <real>dc13gdT
    T DERIV RUBBERY C13 = <real>dc13rdT
    T DERIV GLASSY C23 = <real>dc23gdT
    T DERIV RUBBERY C23 = <real>dc23rdT
    T DERIV GLASSY C44 = <real>dc44gdT
    T DERIV RUBBERY C44 = <real>dc44rdT
    T DERIV GLASSY C55 = <real>dc55gdT
    T DERIV RUBBERY C55 = <real>dc55rdT
    T DERIV GLASSY C66 = <real>dc66gdT
    T DERIV RUBBERY C66 = <real>dc66rdT
    T DERIV GLASSY CTE1 = <real>dcte1gdT
    T DERIV RUBBERY CTE1 = <real>dcte1rdT
    T DERIV GLASSY CTE2 = <real>dcte2gdT
    T DERIV RUBBERY CTE2 = <real>dcte2rdT
    T DERIV GLASSY CTE3 = <real>dcte3gdT
    T DERIV RUBBERY CTE3 = <real>dcte3rdT
    T DERIV GLASSY HCAPACITY = <real>dhcapgdT
    T DERIV RUBBERY HCAPACITY = <real>dhcaprdT
    REF PSIC = <real>psic_ref
    T DERIV PSIC = <real>dpsicdT
    T 2DERIV PSIC = <real>d2psicdT2
    PSI EQ 2T = <real>psitt
    PSI EQ 3T = <real>psittt
    PSI EQ 4T = <real>psitttt
    PSI EQ XX 11 = <real>psiXX11
    PSI EQ XX 22 = <real>psiXX22
    PSI EQ XX 33 = <real>psiXX33
    PSI EQ XX 12 = <real>psiXX12
    PSI EQ XX 13 = <real>psiXX13
    PSI EQ XX 23 = <real>psiXX23
    PSI EQ XX 44 = <real>psiXX44
    PSI EQ XX 55 = <real>psiXX55
    PSI EQ XX 66 = <real>psiXX66
    PSI EQ XXT 11 = <real>psiXXT11
    PSI EQ XXT 22 = <real>psiXXT22
    PSI EQ XXT 33 = <real>psiXXT33
    PSI EQ XXT 12 = <real>psiXXT12
    PSI EQ XXT 13 = <real>psiXXT13
    PSI EQ XXT 23 = <real>psiXXT23
    PSI EQ XXT 44 = <real>psiXXT44
    PSI EQ XXT 55 = <real>psiXXT55
    PSI EQ XXT 66 = <real>psiXXT66
    PSI EQ XT 1 = <real>psiXT1
    PSI EQ XT 2 = <real>psiXT2
    PSI EQ XT 3 = <real>psiXT3
    PSI EQ XTT 1 = <real>psiXTT1
    PSI EQ XTT 2 = <real>psiXTT2
    PSI EQ XTT 3 = <real>psiXTT3
    REF PSIA 11 = <real>psiA11
    REF PSIA 22 = <real>psiA22
    REF PSIA 33 = <real>psiA33
    REF PSIA 12 = <real>psiA12
    REF PSIA 13 = <real>psiA13
    REF PSIA 23 = <real>psiA23
    REF PSIA 44 = <real>psiA44
    REF PSIA 55 = <real>psiA55
    REF PSIA 66 = <real>psiA66
    T DERIV PSIA 11 = <real>dpsiA11dT
    T DERIV PSIA 22 = <real>dpsiA22dT
    T DERIV PSIA 33 = <real>dpsiA33dT
    T DERIV PSIA 12 = <real>dpsiA12dT
    T DERIV PSIA 13 = <real>dpsiA13dT
    T DERIV PSIA 23 = <real>dpsiA23dT
    T DERIV PSIA 44 = <real>dpsiA44dT
    T DERIV PSIA 55 = <real>dpsiA55dT
    T DERIV PSIA 66 = <real>dpsiA66dT
    REF PSIB 1 = <real>psiB1
    REF PSIB 2 = <real>psiB2
    REF PSIB 3 = <real>psiB3
    T DERIV PSIB 1 = <real>dpsiB1dT
    T DERIV PSIB 2 = <real>dpsiB2dT
    T DERIV PSIB 3 = <real>dpsiB3dT
    PSI POT TT = <real>psipotTT
    PSI POT TTT = <real>psipotTTT
    PSI POT TTTT = <real>psipotTTTT
    PSI POT XT 1 = <real>psipotXT1
    PSI POT XT 2 = <real>psipotXT2
    PSI POT XT 3 = <real>psipotXT3
    PSI POT XTT 1 = <real>psipotXTT1
    PSI POT XTT 2 = <real>psipotXTT2
```

```
    PSI POT XTT 3 = <real>psipotXTT3
    PSI POT XXT 11 = <real>psipotXXT11
    PSI POT XXT 22 = <real>psipotXXT22
    PSI POT XXT 33 = <real>psipotXXT33
    PSI POT XXT 12 = <real>psipotXXT12
    PSI POT XXT 13 = <real>psipotXXT13
    PSI POT XXT 23 = <real>psipotXXT23
    PSI POT XXT 44 = <real>psipotXXT44
    PSI POT XXT 55 = <real>psipotXXT55
    PSI POT XXT 66 = <real>psipotXXT66
  END [PARAMETERS FOR MODEL NLVE_3D_ORTHOTROPIC]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Stiff elastic

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_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 STIFF_ELASTIC
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    BULK MODULUS = <real>bulk_modulus
    SHEAR MODULUS = <real>shear_modulus
    LAMBDA = <real>lambda
    SCALE FACTOR = <real>scale_factor
    REFERENCE STRAIN = <real>reference_strain
  END [PARAMETERS FOR MODEL STIFF_ELASTIC]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Swanson

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_value
  #
  BEGIN PARAMETERS FOR MODEL SWANSON
```

```
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    BULK MODULUS = <real>bulk_modulus
    SHEAR MODULUS = <real>shear_modulus
    LAMBDA = <real>lambda
    A1 = <real>a1
    P1 = <real>p1
    B1 = <real>b1
    Q1 = <real>q1
    C1 = <real>c1
    R1 = <real>r1
    CUT OFF STRAIN = <real>ecut
    THERMAL EXPANSION FUNCTION = <string>eth_function_name
    TARGET E = <real>target_e
    TARGET E FUNCTION = <string>etar_function_name
    MAX POISSONS RATIO = <real>max_poissons_ratio
    REFERENCE STRAIN = <real>reference_strain
  END [PARAMETERS FOR MODEL SWANSON]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Viscoelastic Swanson
BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  BIOTS COEFFICIENT = <real>biots_value
  #
  BEGIN PARAMETERS FOR MODEL VISCOELASTIC_SWANSON
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
    BULK MODULUS = <real>bulk_modulus
    SHEAR MODULUS = <real>shear_modulus
    LAMBDA = <real>lambda
    A1 = <real>a1
    P1 = <real>p1
    B1 = <real>b1
    Q1 = <real>q1
    C1 = <real>c1
    R1 = <real>r1
    CUT OFF STRAIN = <real>ecut
    THERMAL EXPANSION FUNCTION = <string>eth_function_name
    PRONY SHEAR INFINITY = <real>ginf
    PRONY SHEAR 1 = <real>g1
    PRONY SHEAR 2 = <real>g2
    PRONY SHEAR 3 = <real>g3
    PRONY SHEAR 4 = <real>g4
    PRONY SHEAR 5 = <real>g5
    PRONY SHEAR 6 = <real>g6
    PRONY SHEAR 7 = <real>g7
    PRONY SHEAR 8 = <real>g8
    PRONY SHEAR 9 = <real>g9
    PRONY SHEAR 10 = <real>g10
    SHEAR RELAX TIME 1 = <real>tau1
    SHEAR RELAX TIME 2 = <real>tau2
    SHEAR RELAX TIME 3 = <real>tau3
    SHEAR RELAX TIME 4 = <real>tau4
    SHEAR RELAX TIME 5 = <real>tau5
    SHEAR RELAX TIME 6 = <real>tau6
    SHEAR RELAX TIME 7 = <real>tau7
    SHEAR RELAX TIME 8 = <real>tau8
    SHEAR RELAX TIME 9 = <real>tau9
    SHEAR RELAX TIME 10 = <real>tau10
    WLF COEF C1 = <real>wlf_c1
    WLF COEF C2 = <real>wlf_c2
    WLF TREF = <real>wlf_tref
    NUMERICAL SHIFT FUNCTION = <string>ns_function_name
    TARGET E = <real>target_e
    TARGET E FUNCTION = <string>etar_function_name
    MAX POISSONS RATIO = <real>max_poissons_ratio
    REFERENCE STRAIN = <real>reference_strain
  END [PARAMETERS FOR MODEL VISCOELASTIC_SWANSON]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Traction Decay

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  BEGIN PARAMETERS FOR MODEL TRACTION_DECAY
    NORMAL DECAY LENGTH = <real>
    TANGENTIAL DECAY LENGTH = <real>
  END [PARAMETERS FOR MODEL TRACTION_DECAY]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Tvergaard Hutchinson

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  BEGIN PARAMETERS FOR MODEL TVERGAARD_HUTCHINSON
    INIT TRACTION METHOD = IGNORE|ADD (IGNORE)
    LAMBDA_1 = <real>
    LAMBDA_2 = <real>
    NORMAL LENGTH SCALE = <real>
    TANGENTIAL LENGTH SCALE = <real>
    PEAK TRACTION = <real>
    PENETRATION STIFFNESS MULTIPLIER = <real>
    NORMAL INITIAL TRACTION DECAY LENGTH = <real>
    TANGENTIAL INITIAL TRACTION DECAY LENGTH = <real>
    USE ELASTIC UNLOADING = NO|YES (YES)
  END [PARAMETERS FOR MODEL TVERGAARD_HUTCHINSON]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# Thouless Parmigiani

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  BEGIN PARAMETERS FOR MODEL THOULESS_PARMIGIANI
    INIT TRACTION METHOD = IGNORE|ADD (IGNORE)
    LAMBDA_1_N = <real>
    LAMBDA_2_N = <real>
    LAMBDA_1_T = <real>
    LAMBDA_2_T = <real>
    NORMAL LENGTH SCALE = <real>
    TANGENTIAL LENGTH SCALE = <real>
    PEAK NORMAL TRACTION = <real>
    PEAK TANGENTIAL TRACTION = <real>
    PENETRATION STIFFNESS MULTIPLIER = <real>
    USE ELASTIC UNLOADING = NO|YES (YES)
  END [PARAMETERS FOR MODEL THOULESS_PARMIGIANI]
END [PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name]

# RVE

BEGIN PROPERTY SPECIFICATION FOR MATERIAL <string>mat_name
  DENSITY = <real>density_value
  #
  BEGIN PARAMETERS FOR MODEL RVE
    YOUNGS MODULUS = <real>youngs_modulus
    POISSONS RATIO = <real>poissons_ratio
  END [PARAMETERS FOR MODEL RVE]
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
  OMIT BLOCK <string>block_list
  COMPONENT SEPARATOR CHARACTER = <string>separator
  BEGIN PARAMETERS FOR BLOCK [<string list>block_names]
```

```
    MATERIAL <string>material_name
    SOLID MECHANICS USE MODEL <string>model_name
    INCLUDE ALL BLOCKS
    REMOVE BLOCK = <string list>block_names
    SECTION = <string>section_id
    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)
    MEMBRANE HOURGLASS STIFFNESS =
      <real>memb_hour_glass_stiff_value(0.0)
    MEMBRANE HOURGLASS VISCOSITY =
      <real>memb_hour_glass_visc_value(0.0)
    BENDING HOURGLASS STIFFNESS =
      <real>bend_hour_glass_stiff_value(0.0)
    BENDING HOURGLASS VISCOSITY =
      <real>bend_hour_glass_visc_value(0.0)
    TRANSVERSE SHEAR HOURGLASS STIFFNESS =
      <real>tshr_hour_glass_stiff_value(0.0)
    TRANSVERSE SHEAR HOURGLASS VISCOSITY =
      <real>tshr_hour_glass_visc_value(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
    INACTIVE FOR PROCEDURE <string>proc_name DURING PERIODS
      <string list>period_names
  END [PARAMETERS FOR BLOCK <string list>block_names]
END [FINITE ELEMENT MODEL <string>mesh_descriptor]

# Element sections

BEGIN SOLID SECTION <string>solid_section_name
  COORDINATE SYSTEM = <string>Coordinate_system_name
  FORMULATION = <string>MEAN_QUADRATURE|
    SELECTIVE_DEVIATORIC|VOID(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)
  HOURGLASS FORMULATION = <string>TOTAL|INCREMENTAL(INCREMENTAL)
  HOURGLASS INCREMENT = <string>ENDSTEP|MIDSTEP (ENDSTEP)
  HOURGLASS ROTATION = <string> APPROXIMATE|SCALED (APPROXIMATE)
  RIGID BODY = <string>rigid_body_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id
    TO <integer>last_id
  USE LAME|STRUMENTO(LAME)
END [SOLID SECTION <string>solid_section_name]

BEGIN COHESIVE SECTION <string>cohesive_section_name
  NUMBER OF INTEGRATION POINTS = <integer>num_int_points(1)
END [COHESIVE SECTION <string>cohesive_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)
  FORMULATION = MEAN_QUADRATURE|NQUAD (MEAN_QUADRATURE)
  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)
  OFFSET MESH VARIABLE = <string>var_name
  ORIENTATION = <string>orientation_name
  DRILLING STIFFNESS FACTOR = <real>stiffness_factor(0.0)
  RIGID BODY = <string>rigid_body_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id
    TO <integer>last_id
  USE LAME|STRUMENTO(LAME)
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)
  RIGID BODY = <string>rigid_body_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id
    TO <integer>last_id
END [MEMBRANE SECTION <string>membrane_section_name]

BEGIN TRUSS SECTION <string>truss_section_name
  AREA = <real>cross_sectional_area
  INITIAL LOAD = <real>initial_load
  PERIOD = <real>period
  RIGID BODY = <string>rigid_body_name
  RIGID BODIES FROM ATTRIBUTES = <integer>first_id
    TO <integer>last_id
  USE LAME|STRUMENTO(LAME)
END [TRUSS SECTION <string>truss_section_name]

BEGIN SUPERELEMENT SECTION <string>section_name
  BEGIN MAP
    <integer>node_index_1 <integer>component_index_1
    <integer>node_index_2 <integer>component_index_2
    ...
    <integer>node_index_n <integer>component_index_n
  END
  BEGIN STIFFNESS MATRIX
    <real>k_1_1 <real>k_1_2 ...  <real>k_1_n
    <real>k_2_1 <real>k_2_2 ...  <real>k_2_n
    ...         ...         ...  ...
    <real>k_n_1 <real>k_n_2 ...  <real>k_n_n
  END
  BEGIN DAMPING MATRIX
    <real>c_1_1 <real>c_1_2 ...  <real>c_1_n
    <real>c_2_1 <real>c_2_2 ...  <real>c_2_n
    ...         ...         ...  ...
    <real>c_n_1 <real>c_n_2 ...  <real>c_n_n
  END
  BEGIN MASS MATRIX
    <real>m_1_1 <real>m_1_2 ...  <real>m_1_n
    <real>m_2_1 <real>m_2_2 ...  <real>m_2_n
    ...          ...        ...  ...
    <real>m_n_1 <real>m_n_2 ...  <real>m_n_n
  END
  FILE = <string>netcdf_file_name
END [SUPERELEMENT SECTION <string>section_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]

# FETI equation solver

BEGIN FETI EQUATION SOLVER <string>name
  #
  # convergence commands
  MAXIMUM ITERATIONS = <integer>max_iter(500)
  RESIDUAL NORM TOLERANCE = <real>resid_tol(1.0e-6)
  #
  # memory usage commands
  PARAM-STRING "precision" VALUE <string>"single"|"double"
    ("double")
  PRECONDITIONING METHOD = NONE|LUMPED|DIRICHLET(DIRICHLET)
  MAXIMUM ORTHOGONALIZATION = <integer>max_orthog(500)
  #
  # solver commands
  LOCAL SOLVER = SKYLINE|SPARSE|ITERATIVE(SPARSE)
  COARSE SOLVER = SKYLINE|SPARSE|ITERATIVE(SPARSE)
  NUM LOCAL SUBDOMAINS = <integer>num_local_subdomains
END [ FETI EQUATION SOLVER <string>name]

# Begin Procedure scope
BEGIN ADAGIO PROCEDURE <string>adagio_procedure_name

  BEGIN PROCEDURAL TRANSFER <string>name

    BLOCK = <string list>block_name
    INCLUDE ALL BLOCKS
    REMOVE BLOCK = <string list>block_name

    BEGIN INTERPOLATION TRANSFER <string>name

      BLOCK BY BLOCK
      NEAREST ELEMENT COPY
      SEND BLOCKS = <string list>block_name
      SEND COORDINATES = ORIGINAL|CURRENT

      RECEIVE BLOCKS = <string list>block_name
      RECEIVE COORDINATES = ORIGINAL|CURRENT

      TRANSFORMATION TYPE = NONE|RIGIDBODY

    END [INTERPOLATION TRANSFER <string>name]
  END [PROCEDURAL TRANSFER <string>name]

  # Time block
  BEGIN TIME CONTROL
    BEGIN TIME STEPPING BLOCK <string>time_block_name
      START TIME = <real>start_time_value
      BEGIN PARAMETERS FOR ADAGIO REGION
        <string>region_name
        TIME INCREMENT = <real>time_increment_value
        NUMBER OF TIME STEPS = <integer>nsteps
        TIME INCREMENT FUNCTION = <string>time_function
      END [PARAMETERS FOR ADAGIO REGION
        <string>region_name]
    END [TIME STEPPING BLOCK <string>time_block_name]
    TERMINATION TIME = <real>termination_time
  END TIME CONTROL

  # Begin Region scope

  BEGIN ADAGIO REGION <string>adagio_region_name

    USE FINITE ELEMENT MODEL <string>model_name
    GLOBAL ENERGY REPORTING = EXACT|APPROXIMATE|OFF (EXACT)
    EXTENSIVE RIGID BODY VARS OUTPUT = OFF|HISTORY|RESULTS|ALL (ALL)


    # implicit dynamic time integration

    BEGIN IMPLICIT DYNAMICS
      ACTIVE PERIODS = <string list>period_names
      INACTIVE PERIODS = <string list>period_names
      USE HHT INTEGRATION
      ALPHA = <real>alpha(0.0) [DURING <string list>period_names]
      GAMMA = <real>beta(0.5) [DURING <string list>period_names]
      BETA = <real>beta(0.25) [DURING <string list>period_names]
      TIME INTEGRATION CONTROL = <string>ADAPTIVE|COMPUTERESIDUAL|
        IGNORE(IGNORE) [DURING <string list>period_names]
      INCREASE ERROR THRESHOLD = <real>increase_threshold(0.02)
        [DURING <string list>period_names]
      HOLD ERROR THRESHOLD = <real>hold_threshold(0.10)
        [DURING <string list>period_names]
      DECREASE ERROR THRESHOLD = <real>decrease_threshold(0.25)
        [DURING <string list>period_names]
    END [IMPLICIT DYNAMICS]

    # 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
      #
      # criterion commands
      CRITERION IS AVG|MAX|MIN NODAL VALUE OF
        <string>var_name <|<=|=|>=|> <real>tolerance
      CRITERION IS ELEMENT VALUE OF
        <string>var_name <|<=|=|>=|> <real>tolerance [KILL WHEN
        <integer>num_intg INTEGRATION POINTS REMAIN]
      CRITERION IS GLOBAL VALUE OF
        <string>var_name <|<=|=|>=|> <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 [KILL WHEN
        <integer>num_intg INTEGRATION POINTS REMAIN]
      #
      # evaluation commands
      CHECK STEP INTERVAL = <integer>num_steps
      CHECK TIME INTERVAL = <real>delta_t
      DEATH START TIME = <real>time
      #
      # miscellaneous option commands
      SUMMARY OUTPUT STEP INTERVAL = <integer>output_step_interval
      SUMMARY OUTPUT TIME INTERVAL = <real>output_time_interval
      DEATH METHOD = <string>DEACTIVATE ELEMENT|
      DEACTIVATE NODAL MPCS|DISCONNECT ELEMENT|
      INSERT COHESIVE ZONES(DEACTIVATE ELEMENT)
      ACTIVE PERIODS = <string list>period_names
      INACTIVE PERIODS = <string list>period_names
      #
      # cohesive zone setup commands
      COHESIVE SECTION = <string>sect_name
      COHESIVE MATERIAL = <string>mat_name
      COHESIVE MODEL = <string>model_name
      COHESIVE ZONE INITIALIZATION METHOD = <string>NONE|
        ELEMENT STRESS AVG(NONE)
    END [ELEMENT DEATH <string>death_name]

    # Derived output

    BEGIN DERIVED OUTPUT
      COMPUTE AND STORE VARIABLE =
        <string>derived_quantity_name
    END DERIVED OUTPUT

    # 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]
      #
      # specification command
      MAGNITUDE = <real list>initial_values
      #
      # probability distribution commands
      DISTRIBUTION = WEIBULL PARAMETERS = <real list>dist_values
        SEED = <integer>dist_seed
      DISTRIBUTION REFERENCE = NODE|EDGE|FACE|ELEMENT|GLOBAL
        <string>size_var_name VALUE = <real>size_ref_val
      #
      # input mesh commands
      READ VARIABLE = <string>var_name
      COPY VARIABLE = <string>var_name
        [FROM MODEL <string>model_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
      INACTIVE 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
      #
      # external database commands
      READ VARIABLE = <string>var_name
      COPY VARIABLE = <string>var_name
        [FROM MODEL <string>model_name]
      TIME = <real>time
      #
      # additional commands
      SCALE FACTOR = <real>scale_factor(1.0)
      ACTIVE PERIODS = <string list>period_names
      INACTIVE 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
      #
      # external database commands
      READ VARIABLE = <string>var_name
      COPY VARIABLE = <string>var_name
        [FROM MODEL <string>model_name]
      TIME = <real>time
      #
      # additional commands
      SCALE FACTOR = <real>scale_factor(1.0)
      ACTIVE PERIODS = <string list>period_names
      INACTIVE 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
      #
      # external database commands
      READ VARIABLE = <string>var_name
      COPY VARIABLE = <string>var_name
        [FROM MODEL <string>model_name]
      TIME = <real>time
      #
      # additional commands
      SCALE FACTOR = <real>scale_factor(1.0)
      ACTIVE PERIODS = <string list>period_names
      INACTIVE 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
      INACTIVE 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
      #
      # external database commands
      READ VARIABLE = <string>var_name
      COPY VARIABLE = <string>var_name
        [FROM MODEL <string>model_name]
      TIME = <real>time
      #
      # additional commands
      SCALE FACTOR = <real>scale_factor(1.0)
      ACTIVE PERIODS = <string list>period_names
      INACTIVE PERIODS = <string list>period_names
    END [PRESCRIBED ROTATION]

    BEGIN PRESCRIBED ROTATIONAL 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
      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
      #
      # external database commands
      READ VARIABLE = <string>var_name
      COPY VARIABLE = <string>var_name
        [FROM MODEL <string>model_name]
      TIME = <real>time
      #
      # additional commands
      SCALE FACTOR = <real>scale_factor(1.0)
      ACTIVE PERIODS = <string list>period_names
      INACTIVE PERIODS = <string list>period_names
    END [PRESCRIBED ROTATIONAL VELOCITY]

    BEGIN REFERENCE AXIS ROTATION
      #
      # block command
      BLOCK = <string list>block_names
      #
      # specification commands
      REFERENCE AXIS X FUNCTION = <string>function_name
      REFERENCE AXIS Y FUNCTION = <string>function_name
      REFERENCE AXIS Z FUNCTION = <string>function_name
      #
      # rotation commands
      ROTATION = <string>function_name
      ROTATIONAL VELOCITY = <string>function_name
      #
      # torque command
      TORQUE = <string>function_name
      #
      # additional command
      SCALE FACTOR = <real>scale_factor(1.0)
      ACTIVE PERIODS = <string list>period_names
      INACTIVE PERIODS = <string list>period_names
    END [REFERENCE AXIS 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
      INACTIVE 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
      #
      # additional commands
      SCALE FACTOR = <real>scale_factor(1.0)
      ACTIVE PERIODS = <string list>period_names
      INACTIVE 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
      INACTIVE 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
      INACTIVE PERIODS = <string list>period_names
    END [PRESCRIBED MOMENT]

    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
      INACTIVE PERIODS = <string list>period_names
    END [GRAVITY]

    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>mesh_var_name
  COPY VARIABLE = <string>var_name
    [FROM MODEL <string>model_name]
  TIME = <real>time
  TEMPERATURE TYPE = SOLID_ELEMENT|SHELL_ELEMENT(SOLID_ELEMENT)
  #
  # coupled analysis commands
  RECEIVE FROM TRANSFER [FIELD TYPE = NODE|ELEMENT(NODE)]
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PRESCRIBED TEMPERATURE]

BEGIN PORE PRESSURE
  #
  # block set commands
  BLOCK = <string_list>block_names
  INCLUDE ALL BLOCKS
  REMOVE BLOCK
  #
  # function command
  FUNCTION = <string>function_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
  #
  # read variable commands
  READ VARIABLE = <string>mesh_var_name
  COPY VARIABLE = <string>var_name
    [FROM MODEL <string>model_name]
  TIME = <real>time
  #
  # coupled analysis commands
  RECEIVE FROM TRANSFER [FIELD TYPE = NODE|ELEMENT(NODE)]
  #
  # additional commands
  SCALE FACTOR = <real>scale_factor(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [PORE PRESSURE]

BEGIN FLUID PRESSURE
  #
  # surface set commands
  SURFACE = <string list>surface_names
  #
  # specification commands
  DENSITY = <real>fluid_density
  DENSITY FUNCTION = <string>density_function_name
  GRAVITATIONAL CONSTANT = <real>gravitational_acceleration
  FLUID SURFACE NORMAL = <string>global_component_names
  DEPTH = <real>fluid_depth
  DEPTH FUNCTION = <string>depth_function_name
  #
  # additional commands
  REFERENCE POINT = <string>reference_point_name
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [FLUID PRESSURE]

# Specialized boundary conditions

BEGIN BLAST PRESSURE
  SURFACE = <string list>surface_ids
  REMOVE SURFACE = <string list>surface_ids
  BURST TYPE = <string>SURFACE|AIR
  TNT MASS IN LBS = <real>tnt_mass_lbs
  BLAST TIME = <real>blast_time
  BLAST LOCATION = <real>loc_x <real>loc_y <real>loc_z
  ATMOSPHERIC PRESSURE IN PSI = <real>atmospheric_press
  AMBIENT TEMPERATURE IN FAHRENHEIT = <real>temperature
  FEET PER MODEL UNITS = <real>feet
  MILLISECONDS PER MODEL UNITS = <real>milliseconds
  PSI PER MODEL UNITS = <real>psi
  PRESSURE SCALE FACTOR = <real>pressure_scale(1.0)
  IMPULSE SCALE FACTOR = <real>impulse_scale(1.0)
  POSITIVE DURATION SCALE FACTOR = <real>duration_scale(1.0)
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names
END [BLAST PRESSURE]


BEGIN MPC
  #
  # Master/Slave MPC commands
  MASTER NODE SET = <string list>master_nset
  MASTER NODES = <integer list>master_nodes
  MASTER SURFACE = <string list>master_surf
  MASTER BLOCK = <string list>master_block
  SLAVE NODE SET = <string list>slave_nset
  SLAVE NODES = <integer list>slave_nodes
  SLAVE SURFACE = <string list>slave_surf
  SLAVE BLOCK = <string list>slave_block
  #
  # Tied contact search command
  SEARCH TOLERANCE = <real>tolerance
  VOLUMETRIC SEARCH TOLERANCE = <real>vtolerance
  #
  # Tied MPC commands
  TIED NODES = <integer list>tied_nodes
  TIED NODE SET = <string list>tied_nset
END [MPC]

RESOLVE MULTIPLE MPCS = ERROR|FIRST WINS|LAST WINS(ERROR)

BEGIN SUBMODEL
  #
  EMBEDDED BLOCKS = <string list>embedded_block
  ENCLOSING BLOCKS = <string list>enclosing_block
END [SUBMODEL]

# Contact

BEGIN CONTACT DEFINITION <string>name
  #
  ENFORCEMENT = <string>TIED|FRICTIONLESS|FRICTIONAL
  #
  CONTACT SURFACE <string>name
    CONTAINS <string list>surface_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
  #
  BEGIN SURFACE NORMAL SMOOTHING
    ANGLE = <real>angle_in_deg
    DISTANCE = <real>distance
    RESOLUTION = <string>NODE|EDGE
  END [SURFACE NORMAL SMOOTHING]
  #
  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 GLUED MODEL <string>name
  END [GLUED MODEL <string>name]
  #
  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
    CAPTURE TOLERANCE = <real>cap_tol
    TENSION RELEASE = <real>ten_release
    SLIP PENALTY = <real>slip_pen
    FACE MULTIPLIER = <real>face_multiplier(0.1)
    SECONDARY DECOMPOSITION = <string>ON|OFF(OFF)
  END [SEARCH OPTIONS <string>name]
  #
  BEGIN INTERACTION DEFAULTS [<string>name]
    CONTACT SURFACES = <string list>surface_names
    GENERAL CONTACT = <string>ON|OFF(OFF)
    FRICTION MODEL = <string>friction_model_name|
      FRICTIONLESS(FRICTIONLESS)
  END [INTERACTION DEFAULTS <string>name]
  #
  BEGIN INTERACTION [<string>name]
```

```
    MASTER = <string>surface
    SLAVE = <string>surface
    CAPTURE TOLERANCE = <real>cap_tol
    NORMAL TOLERANCE = <real>norm_tol
    TANGENTIAL TOLERANCE = <real>tang_tol
    FRICTION MODEL = <string>friction_model_name|
      FRICTIONLESS(FRICTIONLESS)
    PUSHBACK FACTOR = <real>pushback_factor(1.0)
    TENSION RELEASE = <real>ten_release
    TENSION RELEASE FUNCTION = <string>ten_release_func
    FRICTION COEFFICIENT = <real>coeff
    FRICTION COEFFICIENT FUNCTION = <string>coeff_func
  END [INTERACTION <string>name]
  #
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 <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | NODAL <string>variable_name
        [AS <string>dbase_variable_name] ...
        <string>variable_name [AS
        <string>dbase_variable_name]
  NODESET <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | NODESET <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 <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | FACE <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 <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
    | ELEMENT <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
  GLOBAL <string>variable_name
    [AS <string>dbase_variable_name] ...
    <string>variable_name [AS
    <string>dbase_variable_name]
  INCLUDE = <string>list_of_included_element_blocks
  EXCLUDE = <string>list_of_excluded_element_blocks
  OUTPUT MESH = EXPOSED_SURFACE|BLOCK_SURFACE
  COMPONENT SEPARATOR CHARACTER = <string>character|NONE
  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
  SYNCHRONIZE OUTPUT
  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 FILTER <string>filter_name
  ACOEFF = <real_list>a_coeff
  BCOEFF = <real_list>b_coeff
  INTERPOLATION TIME STEP = <real>ts
END [FILTER]

BEGIN USER OUTPUT
  #
  # 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
  #
  # compute result commands
  COMPUTE GLOBAL <string>result_var_name AS
    <string>SUM | AVERAGE | MAX | MIN OF <string>NODAL |
    ELEMENT <string>value_var_name [(<integer>component_num)]
  COMPUTE NODAL <string>result_var_name AS
    <string>MAX OVER TIME|MIN OVER TIME|ABSOLUTE VALUE MAX
    OVER TIME OF NODAL <string>value_var_name
    [(<integer>component_num)]
  COMPUTE ELEMENT <string>result_var_name AS
    <string>MAX OVER TIME|MIN OVER TIME|ABSOLUTE VALUE MAX
    OVER TIME OF 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
  #
  # variable transformation
  TRANSFORM NODAL|ELEMENT VARIABLE <string>source_variable
     TO COORDINATE SYSTEM <string>coord_sys_name AS target_name
  #
  # compute for element death
  COMPUTE AT EVERY TIME STEP
  #
  # additional command
  ACTIVE PERIODS = <string list>period_names
  INACTIVE PERIODS = <string list>period_names

END [USER OUTPUT]

# 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
  INITIAL VARIATION = <real list>values LINEAR DISTRIBUTION
  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
  GLOBAL <string>variable_name
    [AS <string>history_variable_name]
  NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
    AT NODE|NODAL|EDGE|FACE|ELEMENT <integer>entity_id
    [AS <string>history_variable_name]
  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
  SYNCHRONIZE OUTPUT
  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]

# Heartbeat specification

BEGIN HEARTBEAT OUTPUT <string>heartbeat_name
  STREAM NAME = <string>heartbeat_file_name
  FORMAT = SPYHIS|DEFAULT
  GLOBAL <string>variable_name
    [AS <string>heartbeat_variable_name]
  NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
    AT NODE|NODAL|EDGE|FACE|ELEMENT <integer>entity_id
    [AS <string>heartbeat_variable_name]
  NODE|NODAL|EDGE|FACE|ELEMENT <string>variable_name
    NEAREST LOCATION <real>global_x,
      <real>global_y>, <real>global_z
    [AS <string>heartbeat_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
  SYNCHRONIZE OUTPUT
  USE OUTPUT SCHEDULER <string>scheduler_name
  OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
    SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
    SIGKILL|SIGILL|SIGSEGV
  PRECISION = <integer>precision
  LABELS = <string>OFF|ON
  LEGEND = <string>OFF|ON
  TIMESTAMP FORMAT <string>timestamp_format
  MONITOR = <string>RESULTS|RESTART|HISTORY
END [HEARTBEAT OUTPUT <string>heartbeat_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
  SYNCHRONIZE OUTPUT
  USE OUTPUT SCHEDULER <string>scheduler_name
  OUTPUT ON SIGNAL = <string>SIGALRM|SIGFPE|SIGHUP|SIGINT|
    SIGPIPE|SIGQUIT|SIGTERM|SIGUSR1|SIGUSR2|SIGABRT|
    SIGKILL|SIGILL|SIGSEGV
  OPTIONAL
END [RESTART DATA <string>restart_name]

# Solver commands

BEGIN SOLVER
  #
  # nonlinear conjugate gradient (cg) solver commands
  BEGIN CG
    #
    # convergence commands
    TARGET RESIDUAL = <real>target_resid
      [DURING <string list>period_names]
    TARGET RELATIVE RESIDUAL = <real>target_rel_resid
      [DURING <string list>period_names]
    ACCEPTABLE RESIDUAL = <real>accept_resid
      [DURING <string list>period_names]
```

```
ACCEPTABLE RELATIVE RESIDUAL = <real>accept_rel_resid
  [DURING <string list>period_names]
REFERENCE = EXTERNAL|INTERNAL|RESIDUAL(EXTERNAL)
  [DURING <string list>period_names]
MINIMUM RESIDUAL IMPROVEMENT  =  <real>resid_improvement
  [DURING <string list>period_names]
MINIMUM ITERATIONS  = <integer>min_iter(0)
  [DURING <string list>period_names]
MAXIMUM ITERATIONS  = <integer>max_iter
  [DURING <string list>period_names]
#
# preconditioner commands
PRECONDITIONER = BLOCK|BLOCK_INITIAL|DIAGONAL|
  DIAGSCALING|ELASTIC|IDENTITY|PROBE|SCHUR(ELASTIC)
  [<real>scaling_factor]
BALANCE PROBE = <integer>balance_probe(1.0e-6)
NODAL PROBE FACTOR = <real>probe_factor(1.0e-6)
BEGIN FULL TANGENT PRECONDITIONER [<string>name]
  #
  # solver selection commands
  LINEAR SOLVER = <string>linear_solver_name
  NODAL PRECONDITIONER METHOD = ELASTIC|PROBE|DIAGONAL
    (ELASTIC)
  #
  # tangent matrix formation commands
  PROBE FACTOR = <real>probe_factor(1.0e-6)
  BALANCE PROBE = <integer>balance_probe(1)
  CONSTRAINT ENFORCEMENT = SOLVER|PENALTY(PENALTY)
  PENALTY FACTOR = <real>penalty_factor(100.0)
  SHELL DRILLING STIFFNESS =
    <real>shell_drill_stiff(1.0e-4 for quasistatics)
  TANGENT DIAGONAL SCALE = <real>tangent_diag_scale(0.0)
  #
  # reset and iteration commands
  MAXIMUM RESETS FOR MODELPROBLEM = <integer>max_mp_resets
    (100000) [DURING <string list>period_names]
  MAXIMUM RESETS FOR LOADSTEP = <integer>max_ls_resets
    (100000) [DURING <string list>period_names]
  MAXIMUM ITERATIONS FOR MODELPROBLEM =
    <integer>max_mp_iter(100000)
    [DURING <string list>period_names]
  MAXIMUM ITERATIONS FOR LOADSTEP = <integer>max_ls_iter
    (100000) [DURING <string list>period_names]
  ITERATION UPDATE = <integer>iter_update
    [DURING <string list>period_names]
  SMALL NUMBER OF ITERATIONS = <integer>small_num_iter
    [DURING <string list>period_names]
    NUMBER OF SMOOTHING ITERATIONS = <integer>num_smooth_iter
      (0) [DURING <string list>period_names]
    #
    # fall-back strategy commands
    STAGNATION THRESHOLD = <real>stagnation(1.0e-12)
      [DURING <string list>period_names]
    MINIMUM CONVERGENCE RATE = <real>min_conv_rate(1.0e-4)
      [DURING <string list>period_names]
    ADAPTIVE STRATEGY = SWITCH|UPDATE(SWITCH)
      [DURING <string list>period_names]
  END [FULL TANGENT PRECONDITIONER [<string>name]]
  #
  # line search command, default is secant
  LINE SEARCH ACTUAL|TANGENT [DURING <string list>period_names]
  LINE SEARCH SECANT [<real>scale_factor]
  #
  # diagnostic output commands
  ITERATION PRINT = <integer>iter_print
    [DURING <string list>period_names]
  ITERATION PLOT = <integer>iter_plot
    [DURING <string list>period_names]
  ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
  #
  # cg algorithm commands
  ITERATION RESET = <integer>iter_reset(10000)
    [DURING <string list>period_names]
  ORTHOGONALITY MEASURE FOR RESET = <real>ortho_reset(0.5)
    [DURING <string list>period_names]
  RESET LIMITS <integer>iter_start <integer>iter_reset
    <real>reset_growth <real>reset_orthogonality
    [DURING <string list>period_names]
  BETA METHOD = FletcherReeves|PolakRibiere|
    PolakRibierePlus(PolakRibiere)
    [DURING <string list>period_names]
END [CG]
#
# control contact commands
BEGIN CONTROL CONTACT
  #
  # convergence commands
  TARGET RESIDUAL = <real>target_resid
    [DURING <string list>period_names]
  TARGET RELATIVE RESIDUAL = <real>target_rel_resid
    [DURING <string list>period_names]
  TARGET RELATIVE CONTACT RESIDUAL =
    <real>target_rel_cont_resid
    [DURING <string list>period_names]
  ACCEPTABLE RESIDUAL = <real>accept_resid
    [DURING <string list>period_names]
  ACCEPTABLE RELATIVE RESIDUAL = <real>accept_rel_resid
    [DURING <string list>period_names]
  ACCEPTABLE RELATIVE CONTACT RESIDUAL =
    <real>accept_rel_cont_resid
    [DURING <string list>period_names]
  REFERENCE = EXTERNAL|INTERNAL|RESIDUAL(EXTERNAL)
    [DURING <string list>period_names]
  MINIMUM ITERATIONS  = <integer>min_iter(0)
    [DURING <string list>period_names]
  MAXIMUM ITERATIONS  = <integer>max_iter
    [DURING <string list>period_names]
  #
  # level selection command
  LEVEL = <integer>contact_level(1)
  #
  # diagnostic output commands
  ITERATION PLOT = <integer>iter_plot
    [DURING <string list>period_names]
  ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
END [CONTROL CONTACT]
#
# control stiffness commands
BEGIN CONTROL STIFFNESS [<string>stiffness_name]
  #
  # convergence commands
  TARGET <string>RESIDUAL|AXIAL FORCE INCREMENT|
    PRESSURE INCREMENT|SDEV INCREMENT|STRESS INCREMENT
    = <real>target [DURING <string list>period_names]
  TARGET RELATIVE <string>RESIDUAL|AXIAL FORCE INCREMENT|
    PRESSURE INCREMENT|SDEV INCREMENT|STRESS INCREMENT
    = <real>target_rel [DURING <string list>period_names]
  ACCEPTABLE <string>RESIDUAL|AXIAL FORCE INCREMENT|
    PRESSURE INCREMENT|SDEV INCREMENT|STRESS INCREMENT
    = <real>accept [DURING <string list>period_names]
  ACCEPTABLE RELATIVE <string>RESIDUAL|AXIAL FORCE INCREMENT|
    PRESSURE INCREMENT|SDEV INCREMENT|STRESS INCREMENT
    = <real>accept_rel [DURING <string list>period_names]
  REFERENCE = EXTERNAL|INTERNAL|RESIDUAL(EXTERNAL)
    [DURING <string list>period_names]
  MINIMUM ITERATIONS  = <integer>min_iter(0)
    [DURING <string list>period_names]
  MAXIMUM ITERATIONS  = <integer>max_iter
    [DURING <string list>period_names]
  #
  # level selection command
```

```
    LEVEL = <integer>stiffness_level
    #
    # diagnostic output commands
    ITERATION PLOT = <integer>iter_plot
      [DURING <string list>period_names]
    ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
  END [CONTROL STIFFNESS <string>stiffness_name]
  #
  # control failure commands
  BEGIN CONTROL FAILURE [<string>failure_name]
    #
    #  convergence control command
    MAXIMUM ITERATIONS = <integer>max_iter
      [DURING <string list>period_names]
    #
    # level selection command
    LEVEL = <integer>failure_level
    #
    # diagnostic output commands
    ITERATION PLOT = <integer>iter_plot
      [DURING <string list>period_names]
    ITERATION PLOT OUTPUT BLOCKS  = <string list>plot_blocks
  END [CONTROL FAILURE <string>failure_name]
  #
  # predictor commands
  BEGIN LOADSTEP PREDICTOR
    TYPE = <string>SCALE_FACTOR|SECANT|EXTERNAL|EXTERNAL_FIRST
    SCALE FACTOR = <real>factor(1.0)
      [<real>first_scale_factor]
      [DURING <string list>period_names]
    SLIP SCALE FACTOR = <real>slip_factor(1.0)
      [DURING <string list>period_names]
  END [LOADSTEP PREDICTOR]
  LEVEL 1 PREDICTOR = <string>NONE|DEFAULT(DEFAULT)
END [SOLVER]

# Adaptive time stepping

BEGIN ADAPTIVE TIME STEPPING
  METHOD = <string>SOLVER|MATERIAL(SOLVER)
    [DURING <string list>period_names]
  TARGET ITERATIONS = <integer>target_iter
    [DURING <string list>period_names]
  ITERATION WINDOW = <integer>iter_window
    [DURING <string list>period_names]
  CUTBACK FACTOR = <real>cutback_factor(0.5)
    [DURING <string list>period_names]
    GROWTH FACTOR = <real>growth_factor(1.5)
      [DURING <string list>period_names]
    MAXIMUM FAILURE CUTBACKS = <integer>max_cutbacks(5)
      [DURING <string list>period_names]
    MAXIMUM MULTIPLIER = <real>max_multiplier
      [DURING <string list>period_names]
    MINIMUM MULTIPLIER = <real>min_multiplier
      [DURING <string list>period_names]
    RESET AT NEW PERIOD = TRUE|FALSE(TRUE)
      [DURING <string list>period_names]
    ACTIVE PERIODS = <string list>period_names
    INACTIVE PERIODS = <string list>period_names
  END [ADAPTIVE TIME STEPPING]

  JAS MODE [SOLVER|CONTACT|OUTPUT]

  # J-Integral

  BEGIN J INTEGRAL <jint_name>
    #
    # integral parameter specification commands
    CRACK DIRECTION = <real>dir_x <real>dir_y <real>dir_z
    CRACK PLANE SIDE SET = <string list>side_sets
    CRACK TIP NODE SET = <string list>node_sets
    INTEGRATION RADIUS = <real>int_radius
    NUMBER OF DOMAINSS = <integer>num_domains
    FUNCTION = PLATEAU|PLATEAU_RAMP|LINEAR(PLATEAU)
    SYMMETRY = OFF|ON(OFF)
    #
    # time period selection commands
    ACTIVE PERIODS = <string list>period_names
    INACTIVE PERIODS = <string list>period_names
  END J INTEGRAL <jint_name>

END [ADAGIO REGION <string>adagio_region_name]

# Control modes region

BEGIN CONTROL MODES REGION
  #
  # model setup
  USE FINITE ELEMENT MODEL <string>model_name
  CONTROL BLOCKS [WITH <string>coarse_block] =
    <string list>control_blocks
  #
  # solver commands
  BEGIN SOLVER
```

```
    BEGIN LOADSTEP PREDICTOR
      TYPE = <string>SCALE_FACTOR|SECANT|EXTERNAL|EXTERNAL_FIRST
      SCALE FACTOR = <real>factor(1.0)
        [<real>first_scale_factor(scale_factor)]
        [DURING <string list>period_names]
      SLIP SCALE FACTOR = <real>slip_factor(1.0)
        [DURING <string list>period_names]
    END [LOADSTEP PREDICTOR]
    BEGIN CG
      #
      #      Parameters for CG
      #
    END [CG]
  END SOLVER

  JAS MODE [SOLVER|CONTACT|OUTPUT]
  #
  # kinematic boundary condition commands
  BEGIN FIXED DISPLACEMENT
    #
    #  Parameters for fixed displacement
    #
  END [FIXED DISPLACEMENT]
  BEGIN PERIODIC
    #
    # Parameters for periodic
    #
  END [PERIODIC]
  #
  # output commands
  BEGIN RESULTS OUTPUT <string> results_name
    #
    # Parameters for results output
    #
  END RESULTS OUTPUT <string> results_name
END [CONTROL MODES REGION]

# RVE Region

BEGIN RVE REGION <string>rve_region_name
  #
  # Definition of RVEs
  ELEMENTS <integer>elem_i:<integer>elem_j
    BLOCKS <integer>blk_i:<integer>blk_j
    SURFACE|NODESET <integer>start_id INCREMENT <integer>incr
  #
  # ADAGIO REGION commands valid here with the exceptions
```

```
      # discussed in Section 9.1.  These include but are
      # not limited to:
      #
      # Boundary Conditions
      #
      # Results Output Definition
      #
      # Solver Definition
      #
    END [RVE REGION <string>rve_region_name]

  END [ADAGIO PROCEDURE <string>adagio_procedure_name]

END [SIERRA <string>name]
```

Appendix C

Consistent Units

This appendix describes common consistent sets of units. In using Adagio, it is crucial to maintain a consistent set of units when entering material properties and interpreting results. The only variables that have intrinsic units are rotations, which are in radians. All other variables depend on the consistent set of units that the user uses in inputting the material properties and dimensioning the geometry.

A consistent set of units is made by picking the base units, which when using SI unit systems are length, mass, and time. If English unit systems are used, these base units are length, force, and time. All other units are then derived from these base units. Table C.1 provides several examples of commonly used consistent sets of units. In general, the names of the unit systems in this table are taken from the names of the base units. For example, CGS stands for (centimeters, grams, seconds) and IPS stands for (inches, pounds, seconds).

One of the most common mistakes related to consistent units comes in when entering density. For example, in the IPS system, a common error is to enter the density of stainless steel as 0.289 lb/in^3, when it should be entered as 7.48e-4 $lb \cdot s^2/in$. The weight per unit volume should be divided by the gravitational constant (386.4 in/s^2 in this case) to obtain a mass per unit volume.

Table C.1: Consistent Unit Sets

Unit	Unit System				
	SI	CGS	IPS	FPS	MMTS
Mass	kg	g	$\frac{lb \cdot s^2}{in}$	$slug$	$tonne$
Length	m	cm	in	ft	mm
Time	s	s	s	s	s
Density	$\frac{kg}{m^3}$	$\frac{g}{cm^3}$	$\frac{lb \cdot s^2}{in^4}$	$\frac{slug}{ft^3}$	$\frac{tonne}{mm^3}$
Force	N	$dyne$	lb	lb	N
Pressure	Pa	$\frac{dyne}{cm^2}$	psi	psf	MPa
Moment	$N \cdot m$	$dyne \cdot cm$	$in \cdot lb$	$ft \cdot lb$	$N \cdot mm$
Temperature	K	K	$^\circ R$	$^\circ R$	K
Energy	J	erg	$lb \cdot in$	$lb \cdot ft$	mJ
Velocity	$\frac{m}{s}$	$\frac{cm}{s}$	$\frac{in}{s}$	$\frac{ft}{s}$	$\frac{mm}{s}$
Acceleration	$\frac{m}{s^2}$	$\frac{cm}{s^2}$	$\frac{in}{s^2}$	$\frac{ft}{s^2}$	$\frac{mm}{s^2}$

Appendix D

Constraint Enforcement Hierarchy

When a node has multiple constraints, they are enforced in a specific order. Table D.1 shows the order of enforcement of the various types of constraints.

Table D.1: Constraint Enforcement Order

1	Contact
2	Kinematic
3	MPC
4	Rigid Body

If any of the constraints are in conflict, the last constraint enforced will override previously enforced constraints. For example, if a kinematic boundary condition and a MPC are both active on a node and conflict with each other, the kinematic boundary condition will be enforced first, followed by the MPC. As a result, the MPC will be enforced and will override the kinematic boundary condition.

Index

Distribution

1	0899	Technical Library, 9536 (1 electronic)