SANDIA REPORT

SAND2007-2734
Unlimited Release
Printed April 24, 2007

Aria 1.5: User Manual

Patrick K. Notz, Samuel R. Subia, Matthew M. Hopkins, Harry K. Moffat, David R. Noble

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

Sandia National Laboratories

SAND2007-2734
Unlimited Release
Printed April 24, 2007

Aria 1.5: User Manual

Patrick K. Notz, Samuel R. Subia, Matthew M. Hopkins, Harry K. Moffat, David R. Noble
Sandia National Laboratories
P.O. Box 5800
Albuquerque, NM 87185

Abstract

Aria is a Galerkin finite element based program for solving coupled-physics problems described by systems of PDEs and is capable of solving nonlinear, implicit, transient and direct-to-steady state problems in two and three dimensions on parallel architectures. The suite of physics currently supported by Aria includes the incompressible Navier-Stokes equations, energy transport equation, species transport equations, nonlinear elastic solid mechanics, and electrostatics as well as generalized scalar, vector and tensor transport equations. Additionally, Aria includes support for arbitrary Lagrangian-Eulerian (ALE) and level set based free and moving boundary tracking. Coupled physics problems are solved in several ways including fully-coupled Newton's method with analytic or numerical sensitivities, fully-coupled Newton-Krylov methods, fully-coupled Picard's method, and a loosely-coupled nonlinear iteration about subsets of the system that are solved using combinations of the aforementioned methods. Error estimation, uniform and dynamic h-adaptivity and dynamic load balancing are some of Aria's more advanced capabilities. Aria is based on the Sierra Framework.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Aria Overview

Aria is a Sierra application implementing the finite element method (FEM) for solving systems of partial differential equations (PDEs). Foremost, Aria's development targets applications which involve incompressible flow (Navier-Stokes). However, the general design of Aria lends itself to the solution of systems of PDEs describing physical processes including energy transport, species tranport with reactions, electrostatics and general transport of scalar, vector and tensor quantities in two and three dimensions both transient and direct to steady state. Moreover, different regions of the physical domain (i.e., the input mesh) may have either different materials and/or different collections of physics (viz., PDEs) defined on them. These systems of equations may be solved alone, in a segregated but coupled algorith ("loosely coupled") or as a single, fully-coupled system. Currently, Aria's loose coupling capabilities are handled by the Arpeggio application which also allows Aria to couple (loosely) to the quasistatic structural mechanics code, Adagio.

Aria is able to accomodate meshes that utilize linear and quadratic elements in two and three dimensions. In two dimensions, Aria supports quadrilateral (4 and 9 node) and trianglar (3 and 6 node) elements. In three dimensions, Aria supports hexahedral (8 and 27 node) and tetrahedran (4 and 10 node) elements. Moreover, meshes may be comprised of combinations of these elements (i.e., both quadrilateral and trianglar elements in two dimensions).

The physical coordinates and mesh displacements are always interpolated in accordance with the input mesh, but other solution degrees of freedom may be interpolated using a lower order basis function. For example, if the input mesh is composed of 9 node (quadratic) quadrilateral elements, then the physical coordinates and mesh displacements (if active) will be interpolated using quadratic basis functions, whereas other degrees of freedom, e.g., temperature or voltage, could use linear shape functions.

Additional information concerning the project may be found at the Aria's home page, Aria Users Homepage Notz (b), and at Aria's sourceforge web site, Aria SourceForge Project Notz (a) . Both of those web sites currently require access to Sandia's internal restricted network.

1.2 Nonlinear Coupling Strategies in Aria

One of the difficulties with writing broadly applicable computational mechanics software is that developers can't take advantage of specific knowledge of the application domain in order to optimize the algorithm. Thus, in providing generality one sometimes sacrifices efficiency. One place this is evident in multiphysics modeling is in the choice of coupling strategies. While it is well understood that a fully coupled system solved with Newton's method utilizing analytic sensitivities is formally the most robust and correct approach to solving multiphysics applications it is also computationally expensive and complex to implement. Furthermore, while Newton's method has the fastest rate of asymptotic convergence it's domain of convergence is often empirically observed to be smaller than other methods. Lastly, in some applications, certain subsets of the physics may be only weakly coupled so that a loosely coupled approach may be more computationally efficient. To address these concerns while remaining general and flexible Aria offers a number of options for nonlinear solution strategies and physics coupling.

In defining a problem in Aria, users configure one or more Regions. Each Region consists of one or more PDEs to be solved on some or all of the input mesh. All of the PDEs in each Region are solved in a tightly coupled (i.e., single matrix) manner using one of several nonlinear solution strategies available. Users may then define loose couplings between two or more Regions. For example, some or all of a solution from one Region may be transferred to another Region where it is treated as a constant, external field. The aggregate nonlinear problem including the contributions from all of the Regions may be iterated to convergence. The particulars of which physics are solved in each Region and the nonlinear solution strategy used within and between Regions is completely specified through the input file. Furthermore, an Aria user may pick a simple, minimal algorithm without needing to fit it into an overly-generalized worst-case scenario that represents the union of all possible algorithms.

Dynamically-specified loose coupling has many potential advantages that users may leverage. First, the resulting linear system is considerably smaller and contains far fewer off-diagonal contributions which can significantly increase the performance of linear solvers. Also, a resulting linear system may have a more attractive form, such as symmetric positive-definite, that permits the use of tailored iterative solutions techniques. Other extensions to loose coupling include subcycling of transient simulations where each Region may advance in time with its own time step size and in-core coupling to other applications based upon the Sierra framework.

1.3 Constraints Equations within Aria

Aria has a unique capability associated with the specification of global constraint equations. These may be used to specify conserved quantities that are not specifically specified as part of the equations set. For example, in some electrochemistry problems where current is specified as a boundary condition, the global conservation of charge neutrality must be imposed as an additional global condition.

Constraint equations have unique issues associated with their solution.

1.4 Level Set Algorithm

Level set algorithms utilize a signed distance function F such that one material, or *phase*, is associated with regions of space where $F > 0$ and a different phase is associated with regions of space where $F < 0$. The curve or surface where $F = 0$ defines the interface between the two phases. In Goma and most other level set codes F is used to partition material property models such that the property has the appropriate values in each phase and, typically, transitions smoothly from one phase to the other. In Aria, however, F is used to partition contributions of the residual equations between the two phases.

In both cases the partitioning is done using a Heaviside function to partition the physical space into two phases which we'll label A and B. The Heaviside function $H_A(F)$ is defined such that $H_A(F) = 1$ in phase A and $H_A(F) = 0$ in phase B; in the vicinity of $F = 0$ the Heaviside function may be defined to be a smooth function that transitions from 0 to 1. Likewise, $H_B(F) = 1$ in phase B and $H_B(F) = 0$ in phase A. In fact $H_B(F)$ is defined as $H_B(F) \equiv 1 - H_A(F)$.

In Goma, this Heaviside function is used to partition the material properties such that a material property σ is defined as

$$\sigma(F, \ldots) = \sigma_A(\ldots)H_A(F) + \sigma_B(\ldots)H_B(F). \tag{1.1}$$

In Aria, however, the integrand of each residual equation is multipled by the sum of Heaviside function so as to decomponse the equation into contributions from each phase,

$$\int\limits_{\mathrm{V}} (\ldots)\ \mathrm{dV} \rightarrow \int\limits_{\mathrm{V}} H_A(F)\,(\ldots)\ \mathrm{dV} + \int\limits_{\mathrm{V}} H_B(F)\,(\ldots)\ \mathrm{dV}. \tag{1.2}$$

This formulation has a number of advantages. Material models are not functions of F so no special models need to be written and the input sytax is the same as well. Secondly, this approach is conservative for conservative governing equations. For example, the MASS and ADVECTION terms of the energy equation (see section 3.3) are proportional to ρC_p and hence, in Goma's formulation, proportional to $H^2(F)$ where as the DIFFUSION term is proportional κ and hence proportional to $H(F)$. Thus, in the vicinity of the interface $F = 0$ energy is not tranported correctly between these tranport modes.

In Aria, each assembly kernel has an arbitrary list of coefficients that multiply the integrand of the kernel (see section 23.12). Thus, the formulation depicted in equation 1.2 is accomplished by simply adding the appropriate Heaviside function to the list of coefficients for each kernel associated with the equation.

1.5 Outline of the Manual

In chapter 2 we will discuss the overall environment for running Aria applications, including the layout for the Aria input deck. In chapter 3 we will present the general equations that are solved by Aria. These should be read by every user.

In later chapters, we will delve down to discuss individual line commands of the input deck. Chapter 4 discusses equation line cards (i.e., EQ), which serve to add individual equations with coupled independent unknowns to a coupled PDE representation of a region. Chapter 5 discusses how to apply initial conditions to the field variables associated with the equation sets. Chapter 6 presents the line commands associated with specifying boundary conditions. Chapter 7 introduces the concepts associated with distinguishing conditions. Chapter 8 introduces line commands associated with source terms.

Chapter 2

Getting Started

2.1 Setting Up Your Environment

To access Sierra/Aria/Arpeggio one additional entry to your path, the location of the SNTools directory, is required. The SNTools team maintains installations on most of the compute resources available to Sandians and sometimes those change from machine to machine. See The SNTools Project for more details about running on specialized machines. On many machines, including the Linux desktops on the 9100 LAN, the path is that shown in this example.

In addition to setting up your path (see below) you should verify that you are using Sandia's version of `ssh` that includes Kerberos authentication support so that you can run parallel jobs without having to supply your password for each additional process spawned by mpi.

2.1.1 Setting up for the `csh` and `tcsh` Shells

Add SNTools to your path. In either your `/.cshrc` or `/.tcshrc` file add the line

```
set path=(/home/sntools/production/current/sntools/engine $path)
```

2.1.2 Setting up for the `bash` Shell

Add SNTools to your path. In either your `/.bashrc` or `/.profile` file add the line

```
export PATH=/home/sntools/production/current/sntools/engine:$PATH
```

2.2 Running Aria

This section includes some very simple examples of how to run Aria. For more information on running on some of Sandia's clusters, etc. see The SNTools Project.

In its simplest form, Aria can be run like this:

```
% sierra aria -i ariarun.i
```

In this example, `ariarun.i` is the Aria input file. The output – nonlinear iterations, time step information, etc. – will be written to a file called `ariarun.log`. So, you can monitor the progress of the simulation by watching the log file. Alternatively, you can have all of the output sent to the screen by using the `-l` *`logfile`* command line option. If you set the log file to be `-` (a single "minus" character) all of the output will be sent to the standard output (usually your screen):

```
%  sierra aria -i ariarun.i -l -
```

If you would like to use `aprepro` in your input file, add the `-a` command line option to have your input file automatically processed:

```
%  sierra aria -i ariarun.i -l - -a
```

Oftentimes we want to run Aria remotely or locally in a batch mode, save any standard output and perhaps even logout from a session. Unfortunately, termination of the session through either voluntary (interactive) or involuntary (timeout) logout out may in effect terminate the Aria job. In this caes one can prevent the job from terminating by using the Unix nohup command in conjunction with the standard execution command line.

```
%  nohup sierra aria -i ariarun.i -l YourLogFile -a
```

2.3 Aria Environment Overview

Aria is a Sierra application implementing the finite element method (FEM) for solving systems of partial differential equations (PDEs). Foremost, Aria's development targets applications which involve incompressible flow (Navier-Stokes). However, the general design of Aria lends itself to the solution of systems of PDEs describing physical processes including energy transport, species tranport with reactions, electrostatics and general transport of scalar, vector and tensor quantities in two and three dimensions both transient and direct to steady state. Moreover, different regions of the physical domain (i.e., the input mesh) may have either different materials and/or different collections of physics (viz., PDEs) defined on them. These systems of equations may be solved alone, in a segregated but coupled algorith ("loosely coupled") or as a single, fully-coupled system. Currently, Aria's loose coupling capabilities are handled by the Arpeggio application which also allows Aria to couple (loosely) to the quasistatic structural mechanics code, Adagio.

Aria's models and algorithms are integrated into the Sierra framework through the architecture illustrated in Figure 2.1. A Sierra-based application has four layers of code: Domain, Procedure, Region, and Model/Algorithm.

The outermost layer of an application is the Domain, or "main" program of the application. THis domain layer is implemented by the Sierra Framework to manage the startup/shutdown of an application, and to orchestrate the execution of an application-proved set of procedures.

Code at the Procedure level is rsponsible for evolving one or more s loosely coupled ses of physics through a sequence of steps. This sequence may be a set of time steps, nonlinear solver iterations, or some combinations of these or other types of steps.

An application mauy define multiple procedures to implement hand-off coupling between physics within the same main program. In hand-off coupling the first (or preceding) procedure completes execution, mesh and field data is transferred to a succeeding procedure, and the succeeding procedure continues the simulation with a different set of physics. For example, the first thermal procedure could calculate a temperature distribution inside a differentially heated fluid, and the second procedure could simulate natural convection of the fluid due to the density gradients set up by the resulting temperature field.

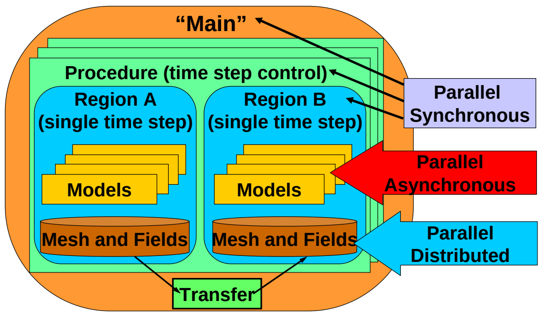


Figure 2.1. Schematic UML class diagram for the Expression subsystem.

Code at the region level is responsible for evolving a tightly coupled set of physics throug a single step. Loose coupling of REgions is supported by the advanced transfer services provided by the Sierra framework.

Each region owns (1) a set of models or algorithms that implement its tightly coupled set of physics and solvers and (2) an in-memory parallel distributed mesh and field database. This mesh and field data is fully distributed among parallel processors via domain decomposition.

2.4 Parallel Processing Runtime Environment

SIMD vs MIMD

mpi

parallel io

exception handling

2.5 Overview of the Input File Structure

An Aria model is described by commands contained in an ASCII input file. The structure of the input file follows a nested hierarchy. The topmost level of this hierarchy is named the domain. Underneath the domain is a level called the procedure, followed by the region level. Figure 1.1 shows this nesting.

The domain level contains one or more procedures. At the domain level, you also find commands associated with describing the finite element mesh, the linear solver set-up, material properties associated with a defined material, and user functions associated with source terms and boundary conditions that are added into Aria's intrinsic set of functions.

The procedure level contains one or more regions. The procedure level is also used to specify the time stepping parameters, and interactions between regions, such as data transfers. Essentially at the procedure level, loose coupling algorithms are specified. Loose coupling here is defined within the context of Aria's implicitly full-coupled paradigm. Whenever an independent variables's interaction with other variables in the solution procedure is not fully represented in the global matrix, the algorithm for loose coupling of that variable and its associated equation will be described at the procedure level. This loose coupling algorithm is given a fancy name called a "solution control description". The procedure level contains a block specifying the solution control procedure. An analogy to this block in simpler codes would be top level loop. For example in time dependent applications, the solution control description block would involve a block to solve the time dependent problem repeated for each time step until the desired solution time is reached.

The region level is used to specify details about the tightly coupled equation system to be solved. The details include boundary conditions and initial conditions, where materials models are applied, and where surface and volumetric source terms are applied. Essentially, meshes and material properties described at the domain level are tied into the problem statement here via their names.

Global constraints equations are also specified at the region level. At the region level, specification of what gets sent to the output file and at what frequency also is made. Additional post-processing associated with the output is specified. For example, additional volumetric fields which are functions of the independent variables may be specified to be added to the output file.

There are two types of commands in the input file. The first type is referred to as a block command. A block command is a grouping mechanism. A block command contains a set of commands made up of other block commands and line commands. A line command is the second type of command. The domain, procedure, and region levels are all parsed as block commands. A block command is defined in the input file by a matching pair of Begin and End lines. For example,

```
Begin SIERRA myJob

 .....

End SIERRA myJob
```

A set of key words for the block command follows the "Begin" and "End" keywords. In most cases a user-specified name is added to the block commands. In the example above the keywords, SIERRA myJob, are added. Optionally, the keyword may be left off of the end of the block.

The second type of command is the line command. A line command is used to specify parameters within a given block command. In the remaining chapters and sections of this manual, the scope of each block and line command is identified, along with summaries of the meanings. Note that the ordering of any commands within a command block is arbitrary. Thus,

```
Begin Finite Element model fluid

Database name is pipeflow2d.g

Use Material water for block1

End Finite Element model fluid
```

will have the same effect as

```
Begin Finite Element model fluid

Use Material water for block1

Database name is pipeflow2d.g
```

```
End Finite Element model fluid
```

And the ordering of command blocks within the domain/procedure/region blocks are arbitrary– allowing you conderable freedom to collect and arrange commands. Note that the terms "command block" and "block command" are interchangeable.

The sierra command block must contain a block for a procedure containing an aria region:

```
Begin procedure myProcedureName

 Begin Aria region name

 End Aria region name

End procedure myProcedureName
```

The procedure command block is used to contain all of the Aria commands that are associated with a solution procedure defined for a set of Aria Regions. The *myProcedureName* and *name* keywords of the procedure and region blocks are left up to you. Note that the Aria procedure command block must be present in the input file and must contain at least one Aria region command block. The procedure command block also contains other important command blocks such as the TIME STEPPING block.

2.5.1 Syntax Conventions for Commands

In this section we describe the conventions used in presenting all the command descriptions in the remainder of this manual. There are four basic kinds of tokens, or words, that Aria expects to find as it parses an input file. These are *keywords, names, parameters* and *delimiters*.

2.5.1.1 Keywords

The words which distinguish one block command, or line command, from another we term keywords. Keywords are denoted in this manual in the monospaced font, for example, `BOUNDARY CONDITION`.

2.5.1.2 Names

The word, or words, that you supply on the same line of the `begin` line of a block command, is the *name*. Many times you may need to supply this *name* as a character parameter in a separate line command. Names are denoted in italics, *name* , as are parameters.

2.5.1.3 Parameters

There are three types of input parameters you may need to supply to line commands: character strings, integers, and real numbers. These are denoted in the documentation as (C), (I), and (R), respectively. Character strings don't have to be delimited by quotation marks. Real numbers may be entered in decimal form or exponential form. For example 0.0001, .1E-3, 10.0d-5 are all equivalent. Furthermore, if a real(R) is expected, an integer can be used. If an integer(I) is expected, however, you must specify it without a decimal point.

2.5.1.4 Multiple Parameters

For the case when a list of one or more paremeters is allowed, or required, for a command, (C,...) denotes a list of character strings, (I,...) a list of integers, and (R, ...) a list of real numbers. For a list of character strings, the separator between the strings must be one or more spaces or tab characters. Therefore, phrases with multiple spaces and words in them are tokenized into multiple character parameters before being processed by the application. For a list of real or integer numbers the comma can also be used as a separator.

2.5.1.5 Enumerated Parameters

Certain commands have predefined parameters, called *enumerations*, which are listed within {}. Each parameter in the list is separated using | . The default parameter for the list of parameters is enclosed by <>.

2.5.1.6 Delimiters

The keywords of a line command are often required to be separated from the parameters by a delimiter. You have a choice of delimiters to use: the equal sign, =, or a word. In this manual, we denote the choices surrounded by {}, and separated by |. You may use any one of the delimiters from those listed. For example, the line command to specify the density within the Property Specificaiton for Material Block command is

`Density` {= |IS} (R)

Examples of valid form syou could write in the input file are

Begin Property Specificaiton for Material water ... Density $\bar{1}$.0E-3 # kg/m$\hat{3}$ at 20C ... End

and

Examples of valid form syou could write in the input file are

Begin Property Specificaiton for Material water ... Density is 1.0E-3 # kg/m$\hat{3}$ at 20C ... End

2.5.1.7 White Space

Command keywords, names, and parameters and delimiters must have spaces around them.

2.5.1.8 Indentation

All leading spaces and/or tab characters are ignored in the input file. Of course, we recommend that you use indentation to improve the readability for yourself and others that may need to see your files.

2.5.1.9 Case Sensitivity

None of the command keywords, parameters, or delimiters read from the input file are case sensitive. For example, the following two lines are equivalent:

```
Use Material water for block_1
```

and

```
USE material wATer for blOCK_1
```

The exception to this rule are file names used for input and output, because the current operating systems on which SIERRA applications are run are based on UNIX, where file names are case sensitive.

2.5.1.10 Comments and Line Continuation

You may place comments in the input file starting with either the $ or # character. All further characters on al ine following a comment character are ignored.

You can continue a command in the input file to the next line by using the line continuation character $, or you may optionally following it with a comment#. All further characters on the same line following a line continuation character $ are ignored, and the characters on the following line are joined and parsing continues. An example is the line command used to specify the title of a thermal model:

```
Begin SIERRA Job_Indentifier

#

$ This thermal model for Aria simulates a convective heat transfer

#

Title \$ The title command is used to set the analysis title

Convective heat transfer to a part.  The analysis \#

makes use of conjugate heat transfer to account for \$

cooling of a part due to flowing water.

 ...

End SIERRA Job_Indentifier
```

2.5.1.11 Checking the Syntax

Errors in the input deck can be checked by adding the command, "-check" to the aria command line. For example,

```
sierra aria -check -i input.i
```

This command will print the code echo of the input deck and any syntax errors within it to the screen.

```
[<Operator>_]<Name>[_<Subindex>][_<Phase>][_<Component>]
```

Figure 2.2. General format of Aria's string-based naming convention for expressions. Fields in square brackets are optional.

2.6 Fields

Fields are defined as variables which are distributed on mesh objects. For example, if the temperature is defined via Q1 interpolation on a 2D mesh consisting of quadrilaterals, then the vector of nodal temperature coefficients that make up the interpolation would be defined as the Temperature field on that mesh. Fields may be defined on any mesh object type (e.g., elements, faces, edges, nodes, node sets, and side sets), not just at nodes.

The mesh object and field data may be distribued among parallel processors via a domain decomposition algorithm. Both fields and meshes are owned at the region level. A particular field may or may not be part of Aria's solution vector for the particular region. However, all fields in Aria's solution vector are fields defined on the mesh for that region.

2.6.1 Field String-Naming Convention

Due to the dynamic nature of fields and variables in Aria a consistent naming convention must be used for sanity sake. This section describes the format of string-names of Aria Expressions. These string forms are used for input and output only; Aria has more efficient internal structures for referencing Expressions.

Briefly, the overall format is described in Figure 2.6.1.

Valid values of the `<Operator>` field are listed in Table 2.6.1. Valid values of the `<Name>` field are too numerous to list here; they include things like degrees of freedom (`VELOCITY`, `SPECIES`) and material properties (`VISCOSITY`, `ELECTRICAL_CONDUCTIVITY`). The `<Subindex>` field can be used to designate multiple instances of a field. This is typically used for species equations. All integer values are valid subindex values but it's best to use values ≥ 1. The `<Phase>` field is used in level set problems. Some fields are present in "all phases" while others, such as material properties, depend on which phase is being referred to. The `<Component>` field allows the user to specify a particular component of vector and tensor fields; valid values are described in Table 2.6.1.

2.7 Equations

Equations are defined within an Aria region to represent an particular continuity equation to be solved. Within the Aria input deck, solution variables are assigned as the independent unknowns to equations. In general, there is a one-to-one correspondence between solution unknowns and equation degrees of freedom.

2.8 Equation String-Naming Convention

Similar to the field string-naming convention, equation names pose a similar requirement. This section describes the format of string-names of Aria equations. These string forms are used for

Operator	**Description**
(none)	"No-Op", no-operator
`DT`	Time derivative
`GRAD`	Gradient
`DIV`	Divergence
`DET`	Determinant of a 2-tensor
`DETJ`	Determinate of the Jacobian of transformation
`SURFACE_DETJ`	Determinate of the Jacobian of transformation
`REF_FRAME`	"No-Op" in the undeformed reference frame
`GRAD_REF_FRAME`	Gradient in the undeformed reference frame
`DIV_REF_FRAME`	Divergence in the undeformed reference frame
`DETJ_REF_FRAME`	Determinate of the Jacobian of transformation in the undeformed reference frame
`SURFACE_DETJ_REF_FRAME`	Determinate of the Jacobian of transformation in the undeformed reference frame
`OLD`	"No-Op" at the previous time step
`GRAD_OLD`	Gradient at the previous time step
`DIV_OLD`	Divergence at the previous time step

Table 2.1. Valid values of of the `<Operator>` prefix.

Phase	**Description**
(none)	A field present in all phases within a material
`A`	Phase A
`B`	Phase B
`C`	Phase C

Table 2.2. Valid values of of the `<Phase>` suffix. Phase lables are used in level set calculations only.

Component	**Description**
(none)	No specified component
`X`	First vector component
`Y`	Second vector component
`Z`	Third vector component
`XX`	(1,1) 2-tensor component
`XY`	(1,2) 2-tensor component
`XZ`	(1,3) 2-tensor component
`YX`	(2,1) 2-tensor component
`YY`	(2,2) 2-tensor component
`YZ`	(2,3) 2-tensor component
`ZX`	(3,1) 2-tensor component
`ZY`	(3,2) 2-tensor component
`ZZ`	(3,3) 2-tensor component

Table 2.3. Valid values of of the `<Component>` suffix. In non-cartesian coordinate systems these may refer to, for example, radial or angular components.

String-Name	Description
`TEMPERATURE`	Just the temperature.
`SPECIES_2`	Species number two
`VELOCITY_X`	The first component of the velocity vector
`DIV_VELOCITY`	The devergence of the velocity field
`DENSITY`	The density
`DENSITY_A`	The density in level set phase A
`GRAD_SPECIES_2_Y_B`	The second component of the gradient of species number 2 in level set phase B

Table 2.4. Examples of well formed string names for Aria Expressions.

```
<Equation_Name>[_<Subindex>][_<Phase>][_<Component>]
```

Figure 2.3. General format of Aria's string-based naming convention for equations. Fields in square brackets are optional.

input and output only; Aira has more efficient internal structures for referencing equations.

Briefly, the overall format is described in Figure 2.8.

Valid values of the `<Equation_Name>` field are numerous and changing in time. Typical values include `MOMENTUM`, `ENERGY`, `SPECIES`, `LEVEL_SET`, `MESH`, `CURRENT` and `VOLTAGE`; see chapter 4 for a complete description of existing equations. All integer values are valid subindex values but it's best to use values ≥ 1 – currently -1 has a special meaning of "no subindex". The `<Phase>` field is used in level set problems. Some fields are present in "all phases" while others, such as material properties, depend on which phase is being referred to. The `<Component>` field allows the user to specify a particular component of vector and tensor equation; valid values are described in Table 2.6.1.

2.9 Example Program Directory

2.10 Aprepro Interface

Chapter 3

Equations Aria Solves

3.1 Generalized Conservation Equation

We first introduce a general conservation equation, as a model for the specific equations that Aria solves, demonstrating how the galerkin finite element method is applied to it, and how the integration by parts is carried out on its individual terms. Following Deen (1998), the conservation of a general scalar quantity $b(\boldsymbol{x}, t)$, with units of amount-per-unit-volume, at a point $\boldsymbol{x}$ and time t can be expressed as

$$\frac{\partial b}{\partial t} + \boldsymbol{\nabla} \cdot (b\boldsymbol{v}) = -\boldsymbol{\nabla} \cdot \boldsymbol{f} + B_{\mathrm{V}} \tag{3.1}$$

where $\boldsymbol{v}$ is the mass average velocity, $\boldsymbol{f}$ is the diffusive flux of b, and B_{V} is the volumetric source of b.

The Galerkin FEM (G/FEM) residual form of 3.1 is formed by bringing the right hand side terms to the left, multiplying by the FEM weight function ϕ^i and integrating over the volume V,

$$R_b^i = \int\limits_{\mathrm{V}} \left(\frac{\partial b}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{\nabla} b + b \boldsymbol{\nabla} \cdot \boldsymbol{v} + \boldsymbol{\nabla} \cdot \boldsymbol{f} - B_{\mathrm{V}} \right) \phi^i \, \mathrm{dV} = 0. \tag{3.2}$$

In many applications $\boldsymbol{\nabla} \cdot \boldsymbol{v} = 0$ so we ignore that term from here on. However, it is straight forward to account for this term via the source term B_{V}. Using the vector identity $(\boldsymbol{\nabla} \cdot \boldsymbol{f})\phi^i = \boldsymbol{\nabla} \cdot (\boldsymbol{f}\phi^i) - \boldsymbol{\nabla}\phi^i \cdot \boldsymbol{f}$ and using the divergence theorem, 3.2 becomes

$$R_b^i = \int\limits_{\mathrm{V}} \left[\left(\frac{\partial b}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{\nabla} b - B_{\mathrm{V}} \right) \phi^i - \boldsymbol{\nabla}\phi^i \cdot \boldsymbol{f} \right] \mathrm{dV} + \int\limits_{\mathrm{S}} \boldsymbol{n} \cdot \boldsymbol{f} \phi^i \, \mathrm{dS} = 0. \tag{3.3}$$

Here $\boldsymbol{n}$ is a unit normal along the boundary S, pointing out of the volume V.

Equation 3.3 embodies the sign convention for sources, fluxes and equation terms used within Aria. For example, scalar flux expressions in Aria provide values for $f_n \equiv \boldsymbol{n} \cdot \boldsymbol{f}$ and should be positive for a flux of b leaving the volume V.

Note also that we have not assigned a units convention to the equation. Any unit system may be employed in the specification of the individual terms in 3.1. However, each term in 3.1 must have overall units of [b] / [time], and the overall residual expression has units of [b] * [L]**3 / [time], where [b] are units of the conserved quantity, b, [L] is the unit of the length scale, and [time] is the unit for time.

3.2 Conservation of Mass

For a material with density ρ, letting $b = \rho$ results in the conservation of mass. Since there is no net flow relative to the mass average velocity $\boldsymbol{f} = \boldsymbol{0}$. Although there are no sources of mass, having such a source can be convenient in modeling and simulation; so, we let the mass source be $B_\mathrm{V} = q_m$. Thus, (3.1) becomes

$$\frac{\partial \rho}{\partial t} + \rho \boldsymbol{\nabla} \cdot \boldsymbol{v} + \boldsymbol{v} \cdot \boldsymbol{\nabla} \rho = q_m. \tag{3.4}$$

For the special but common case of constant density, this reduces to

$$\boldsymbol{\nabla} \cdot \boldsymbol{v} = 0. \tag{3.5}$$

Using equation 3.3, the G/FEM residual form is

$$R_P^i = \int\limits_\mathrm{V} \left(-\frac{\partial \rho}{\partial t} - \rho \boldsymbol{\nabla} \cdot \boldsymbol{v} - \boldsymbol{v} \cdot \boldsymbol{\nabla} \rho + q_m \right) \phi^i \,\mathrm{dV} = 0. \tag{3.6}$$

Important Note: Equation 3.6 has been multiplied by -1 because this form results in a better linear system for the special case of incompressible flow. This is important to remember when defining mass source terms.

In Aria, each term in 3.6 is specified separately as identified in equation 3.7.

$$R_P^i = \underbrace{\int\limits_\mathrm{V} -\frac{\partial \rho}{\partial t} \phi^i \,\mathrm{dV}}_{\text{MASS}} + \underbrace{\int\limits_\mathrm{V} -\left(\boldsymbol{v} \cdot \boldsymbol{\nabla} \rho + \rho \boldsymbol{\nabla} \cdot \boldsymbol{v}\right) \phi^i \,\mathrm{dV}}_{\text{ADV}} + \underbrace{\int\limits_\mathrm{V} q_m \phi^i \,\mathrm{dV}}_{\text{SRC}} = 0 \tag{3.7}$$

For a purely incompressible form, Aria offers the alternative form given in 3.8;

$$R_P^i + \underbrace{\int\limits_\mathrm{V} -\boldsymbol{\nabla} \cdot \boldsymbol{v} \phi^i \,\mathrm{dV}}_{\text{DIV}} + \underbrace{\int\limits_\mathrm{V} q_m \phi^i \,\mathrm{dV}}_{\text{SRC}} = 0 \tag{3.8}$$

3.3 Conservation of Energy

For a material with constant density and specific heat C_p, temperature T, heat flux $\boldsymbol{q}$ and volumetric energy source H_V, letting $b = \rho C_p T$, $\boldsymbol{f} = \boldsymbol{q}$ and $B_\mathrm{V} = H_V$ results in the conservation of energy.

$$\rho C_p \frac{\partial T}{\partial t} + \rho C_p \boldsymbol{v} \cdot \boldsymbol{\nabla} T = -\boldsymbol{\nabla} \cdot \boldsymbol{q} + H_V. \tag{3.9}$$

A common constitutive relationship for $\boldsymbol{q}$ is Fourier's law, $\boldsymbol{q} = -\kappa \boldsymbol{\nabla} T$ where κ is the thermal conductivity. However, we leave the heat flux as an option to be specified as part of the material properties (see section 10.13). Using equation 3.3, the G/FEM residual form is

$$R_T^i = \int\limits_\mathrm{V} \left[\left(\rho C_p \frac{\partial T}{\partial t} + \rho C_p \boldsymbol{v} \cdot \boldsymbol{\nabla} T - H_V \right) \phi^i - \boldsymbol{\nabla} \phi^i \cdot \boldsymbol{q} \right] \mathrm{dV} + \int\limits_\mathrm{S} q_n \phi^i \,\mathrm{dS} = 0 \tag{3.10}$$

where q_n is the heat flux at the boundary. For example, the natural convection boundary condition gives $q_n = h(T - T_\infty)$ where h is the heat transfer coefficient and T_∞ is the bulk temperature away from the surface.

In Aria, each term in 3.10 is specified separately as identified in equation 3.11.

$$R_T^i = \underbrace{\int\limits_V \rho C_p \frac{\partial T}{\partial t}\phi^i \,\mathrm{dV}}_{\text{MASS}} + \underbrace{\int\limits_V \rho C_p \boldsymbol{v}\cdot\boldsymbol{\nabla} T \phi^i \,\mathrm{dV}}_{\text{ADV}} - \underbrace{\int\limits_V H_V \phi^i \,\mathrm{dV}}_{\text{SRC}} - \underbrace{\int\limits_V \boldsymbol{\nabla}\phi^i \cdot \boldsymbol{q}\,\mathrm{dV}}_{\text{DIFF}} + \int\limits_S q_n \phi^i \,\mathrm{dS} = 0 \quad (3.11)$$

More and more often we needing to account for variable density problems and so we need to bring back some of the terms we threw away because we were going to assume $\boldsymbol{\nabla}\cdot\boldsymbol{v} \equiv 0$. Here's a do-over of equation 3.11 that accomodates a variable density through the DIV term:

$$R_T^i = \underbrace{\int\limits_V \rho C_p \frac{\partial T}{\partial t}\phi^i \,\mathrm{dV}}_{\text{MASS}} + \underbrace{\int\limits_V \rho C_p \boldsymbol{v}\cdot\boldsymbol{\nabla} T \phi^i \,\mathrm{dV}}_{\text{ADV}} + \underbrace{\int\limits_V \rho C_p T \boldsymbol{\nabla}\cdot\boldsymbol{v}\phi^i \,\mathrm{dV}}_{\text{DIV}} - \underbrace{\int\limits_V H_V \phi^i \,\mathrm{dV}}_{\text{SRC}} - \underbrace{\int\limits_V \boldsymbol{\nabla}\phi^i \cdot \boldsymbol{q}\,\mathrm{dV}}_{\text{DIFF}} + \int\limits_S q_n \phi^i \,\mathrm{dS} = 0 \quad (3.12)$$

Note, however, that equation 3.12 still assumes a constant specific heat C_p.

3.4 Conservation of Chemical Species

For a material with species k with molar concentration C_k, molar flux $\boldsymbol{J}_k$ relative to the mass average velocity and volumetic reation rate $R_{V,k}$, letting $b = y_k$, $\boldsymbol{f} = \boldsymbol{J}_k$ and $B_V = R_{V,k}$ in (3.1) results in the conservation equation for species k,

$$\frac{\partial C_k}{\partial t} + \boldsymbol{v}\cdot\boldsymbol{\nabla} C_k = -\boldsymbol{\nabla}\cdot\boldsymbol{J}_k + R_{V,k}. \quad (3.13)$$

For liquid mixtures which are dilute in all species except one, Fick's law is often used to approximate $\boldsymbol{J}_k$. In this approximation, D_k represents the diffusion coefficient of species k with respect to the concentrated species and it is assumed that the interactions between dilute species is assumed negligible. Again, however, we choose to leave the governing equation in the more general form and require the particular diffusive flux model as user input (see section 10.30). Using equation 3.3, the G/FEM residual form is

$$R_{C_k}^i = \int\limits_V \left[\left(\frac{\partial C_k}{\partial t} + \boldsymbol{v}\cdot\boldsymbol{\nabla} C_k - R_{V,k}\right)\phi^i - \boldsymbol{\nabla}\phi^i\cdot\boldsymbol{J}_k\right]\mathrm{dV} + \int\limits_S q_{n,k}\phi^i\,\mathrm{dS} = 0 \quad (3.14)$$

where $q_{n,k}$ is the mass flux at the boundary. For example, the natural convection boundary condition gives $q_n = k(C_k - C_{\infty,k})$ where k is the mass transfer coefficient and $C_{\infty,k}$ is the bulk concentration away from the surface.

In Aria, each term in 3.14 is specified separately as identified in equation 3.15.

$$R^i_{C_k} = \underbrace{\int\limits_V \frac{\partial C_k}{\partial t}\phi^i \,\mathrm{dV}}_{\text{MASS}} + \underbrace{\int\limits_V \boldsymbol{v}\cdot\boldsymbol{\nabla} C_k \phi^i \,\mathrm{dV}}_{\text{ADV}} - \underbrace{\int\limits_V R_{V,k}\phi^i \,\mathrm{dV}}_{\text{SRC}} \\ - \underbrace{\int\limits_V \boldsymbol{\nabla}\phi^i \cdot \boldsymbol{J}_k \,\mathrm{dV}}_{\text{DIFF}} + \int\limits_S q_{n,k}\phi^i \,\mathrm{dS} = 0 \quad (3.15)$$

More and more often we needing to account for variable density problems and so we need to bring back some of the terms we threw away because we were going to assume $\boldsymbol{\nabla}\cdot\boldsymbol{v} \equiv 0$. Here's a do-over of equation 3.15 that accomodates a variable density through the DIV term:

$$R^i_{C_k} = \underbrace{\int\limits_V \frac{\partial C_k}{\partial t}\phi^i \,\mathrm{dV}}_{\text{MASS}} + \underbrace{\int\limits_V \boldsymbol{v}\cdot\boldsymbol{\nabla} C_k \phi^i \,\mathrm{dV}}_{\text{ADV}} + \underbrace{\int\limits_V C\boldsymbol{\nabla}\cdot\boldsymbol{v}\phi^i \,\mathrm{dV}}_{\text{DIV}} \\ - \underbrace{\int\limits_V R_{V,k}\phi^i \,\mathrm{dV}}_{\text{SRC}} - \underbrace{\int\limits_V \boldsymbol{\nabla}\phi^i \cdot \boldsymbol{J}_k \,\mathrm{dV}}_{\text{DIFF}} + \int\limits_S q_{n,k}\phi^i \,\mathrm{dS} = 0 \quad (3.16)$$

Often times it is useful to solve for mass, weight or volume fractions of each species rather than for the concentration directly. In that case, an additional condition exists,

$$\sum_k C_k = 1 \quad (3.17)$$

Using this condition, it is only necessary to solve for $N-1$ species fractions where N is the total number of species present in the problem. The final species, then, is simply given as

$$C_j = 1 - \sum_{k\neq j} C_k \quad (3.18)$$

This method can be triggered in Aria by specifing the equation term FRACBAL. In this case, the equation for C_j is not included in the system of unknowns but is instead post-processed on the fly. Aria will automatically detect all other species equations and include them in the fraction balance.

3.5 Conservation of Fluid Momentum

The Cauchy momentum equation is given by

$$\rho\frac{\partial \boldsymbol{v}}{\partial t} + \rho\boldsymbol{v}\cdot\boldsymbol{\nabla}\boldsymbol{v} - \boldsymbol{g} - \boldsymbol{\nabla}\cdot\boldsymbol{T} = \boldsymbol{0} \quad (3.19)$$

where $\boldsymbol{T}$ is the fluid stress tensor and $\boldsymbol{g}$ is a body force. We construct the G/FEM residual form of 3.19 by contracting with the unit coordinate vector in the k-direction, $\boldsymbol{e}_k$, multiplying by the weight function ϕ^i and integrating over the volume. Using the vector identiy $(\boldsymbol{\nabla}\cdot\boldsymbol{T})\cdot\boldsymbol{e}_k\phi^i = \boldsymbol{\nabla}\cdot(\boldsymbol{T}\cdot\boldsymbol{e}_k\phi^i) - \boldsymbol{T}^t : \boldsymbol{\nabla}(\boldsymbol{e}_k\phi^i)$ and integrating by parts gives

$$R^i_{m,k} = \int\limits_V \left[\left(\rho\frac{\partial \boldsymbol{v}}{\partial t} + \rho\boldsymbol{v}\cdot\boldsymbol{\nabla}\boldsymbol{v} - \boldsymbol{g}\right)\cdot\boldsymbol{e}_k\phi^i + \boldsymbol{T}^t : \boldsymbol{\nabla}\left(\boldsymbol{e}_k\phi^i\right)\right]\mathrm{dV} - \int\limits_S \boldsymbol{n}\cdot\boldsymbol{T}\cdot\boldsymbol{e}_k\phi^i \,\mathrm{dS} = 0 \quad (3.20)$$

In Aria, each term in 3.20 is specified separately as identified in equation 3.15.

$$R^i_{m,k} = \underbrace{\int_V \rho \frac{\partial \boldsymbol{v}}{\partial t} \cdot \boldsymbol{e}_k \phi^i \, \mathrm{dV}}_{\text{MASS}} + \underbrace{\int_V \rho \boldsymbol{v} \cdot \boldsymbol{\nabla} \boldsymbol{v} \cdot \boldsymbol{e}_k \phi^i \, \mathrm{dV}}_{\text{ADV}} - \underbrace{\int_V \boldsymbol{g} \cdot \boldsymbol{e}_k \phi^i \, \mathrm{dV}}_{\text{SRC}} + \underbrace{\int_V \boldsymbol{T}^t : \boldsymbol{\nabla} \left(\boldsymbol{e}_k \phi^i \right) \mathrm{dV}}_{\text{DIFF}} - \int_S \boldsymbol{n} \cdot \boldsymbol{T} \cdot \boldsymbol{e}_k \phi^i \, \mathrm{dS} = 0 \qquad (3.21)$$

3.6 Conservation of Solid Momentum

Aria currently solves the quasistatic form of the solid momentum equations. Furthermore, the solid stress is treated as a linear elastic material. In this limit, the Cauchy momentum equation is given by

$$\boldsymbol{\nabla} \cdot \boldsymbol{T} = \boldsymbol{0} \qquad (3.22)$$

where $\boldsymbol{T}$ is the solid stress tensor. We construct the G/FEM residual form of 3.19 by contracting with the unit coordinate vector in the k-direction, $\boldsymbol{e}_k$, multiplying by the weight function ϕ^i and integrating over the volume. Using the vector identiy $(\boldsymbol{\nabla} \cdot \boldsymbol{T}) \cdot \boldsymbol{e}_k \phi^i = \boldsymbol{\nabla} \cdot (\boldsymbol{T} \cdot \boldsymbol{e}_k \phi^i) - \boldsymbol{T}^t : \boldsymbol{\nabla}(\boldsymbol{e}_k \phi^i)$ and integrating by parts gives

$$R^i_{m,k} = \int_V \boldsymbol{T}^t : \boldsymbol{\nabla} \left(\boldsymbol{e}_k \phi^i \right) \mathrm{dV} = 0 \qquad (3.23)$$

Here, the surface contribution, $\int_S \boldsymbol{n} \cdot \boldsymbol{T} \cdot \boldsymbol{e}_k \phi^i \, \mathrm{dS}$, has been dropped because Aria currently only supports dirichlet and natural (homogeneous) boundary conditions for the solid equation.

In Aria, each term in 3.23 is specified separately as identified in equation 3.24.

$$R^i_{m,k} = \underbrace{\int_V \boldsymbol{T}^t : \boldsymbol{\nabla} \left(\boldsymbol{e}_k \phi^i \right) \mathrm{dV}}_{\text{DIFF}} = 0 \qquad (3.24)$$

Currently, Aria does not support direct specification of the more popular stress-strain parametization that utilizes Young's modulus E, Poisson's ratio ν and coefficient of thermal expansion α (note, the shear modulus $G = \mu$). The relationship between these two parametizations is summarized here for convenience.

$$2\mu = \frac{E}{(1+\nu)} \qquad (3.25)$$

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)} = 2\mu \frac{\nu}{(1-2\nu)} \qquad (3.26)$$

$$\beta = \frac{\alpha E}{(1-2\nu)} = \alpha (3\lambda + 2\mu) \qquad (3.27)$$

3.7 Voltage Equation

The electric potential or voltage V is frequently used in determining the electric field, $\boldsymbol{E} = -\boldsymbol{\nabla} V$. While (3.1) cannot be applied to the voltage, the equation governing the voltage – Gauss' law from

Maxwell's equations – has a similar form. Writing the electric displacement $\boldsymbol{D}$ as $\boldsymbol{D} = \epsilon \boldsymbol{E}$, where ϵ is the electric permittivity, Gauss' law is

$$\boldsymbol{\nabla} \cdot \epsilon \boldsymbol{\nabla} V = \rho_e \tag{3.28}$$

where the permittivity is taken to be a constant and ρ_e is the volumetric free charge density.

Using equation 3.3, the G/FEM residual form is

$$R_V^i = \int_{\mathrm{V}} \left(-\rho_e \phi^i + \boldsymbol{\nabla} \phi^i \cdot \epsilon \boldsymbol{\nabla} V \right) \mathrm{dV} + \int_{\mathrm{S}} q_n \phi^i \, \mathrm{dS} = 0 \tag{3.29}$$

In Aria, each term in 3.29 is specified separately as identified in equation 3.30.

$$R_V^i = -\underbrace{\int_{\mathrm{V}} \rho_e \phi^i \, \mathrm{dV}}_{\mathrm{SRC}} + \underbrace{\int_{\mathrm{V}} \boldsymbol{\nabla} \phi^i \cdot \epsilon \boldsymbol{\nabla} V \, \mathrm{dV}}_{\mathrm{DIFF}} + \int_{\mathrm{S}} q_n \phi^i \, \mathrm{dS} = 0 \tag{3.30}$$

3.8 Current Equation

An alternate formulation for solving for the electrical potential (see section 3.7) is to solve the "current" equation which is a conservation equation for electrical charge. The electrical current $\boldsymbol{J}$ is frequently related to the electric field $\boldsymbol{E}$ using Ohm's law as $\boldsymbol{J} = \sigma_e \boldsymbol{E}$ where σ_e is the electrical conductivity. The electric potential or voltage V is used in determining the electric field, $\boldsymbol{E} = -\boldsymbol{\nabla} V$. However, we choose to leave the electrical current as a more general constitutive model to be provided as a material model input (see section 10.4).

$$-\boldsymbol{\nabla} \cdot \boldsymbol{J} = \rho_e \tag{3.31}$$

Using equation 3.3, the G/FEM residual form is

$$R_V^i = \int_{\mathrm{V}} \left(-\rho_e \phi^i - \boldsymbol{\nabla} \phi^i \cdot \boldsymbol{J} \right) \mathrm{dV} + \int_{\mathrm{S}} q_n \phi^i \, \mathrm{dS} = 0 \tag{3.32}$$

In Aria, each term in 3.32 is specified separately as identified in equation 3.33.

$$R_V^i = -\underbrace{\int_{\mathrm{V}} \rho_e \phi^i \, \mathrm{dV}}_{\mathrm{SRC}} - \underbrace{\int_{\mathrm{V}} \boldsymbol{\nabla} \phi^i \cdot \boldsymbol{J} \, \mathrm{dV}}_{\mathrm{DIFF}} + \int_{\mathrm{S}} q_n \phi^i \, \mathrm{dS} = 0 \tag{3.33}$$

3.9 Suspension Equation

In treating the suspension as a continuum, we introduce an evolution equation for particle volume fraction, ϕ, as

$$\frac{\partial \phi}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{\nabla} \phi + \boldsymbol{\nabla} \cdot \boldsymbol{N} = 0. \tag{3.34}$$

The particle volume fraction is defined as the total summed volume of particles per volume of the particle medium. 3.34 represents a balance between the storred particles, the convected particle flux, and the diffusive particle flux, N. Several mechanisms which include Brownian motion, sedimentation, hydrodynamic particle interactions, and gradients in suspension viscosity may contribute to the diffusive particle flux. Specification of the appropriate flux model must then be carried out to close the definition of the conservation equation.

3.10 Stress Tensor Projection Equation

A projection equation is defined as an equation where a derived quantity at the interior gauss points is evaluated and projected to be a solution unknown at the nodal points. The stress tensor projection equation projects the momentum stresses, tau, without the pressure term, to the nodal points. Projecting the momentum stress smoothes out the momentum stress tensor and allows for a dot product to be carried out on the projected field, which is needed for least squares stabilization schemes. The solution variable, τ_{ab}, is calculated from 3.35.

$$R_V^i = -\underbrace{\int_{\mathrm{V}} (\tau_{ab} - src\_\tau_{ab})\phi^i \, \mathrm{dV}}_{\mathrm{DEF}} = 0 \tag{3.35}$$

τ_{ab} is a tensor variable. For 2D problems, ab stands for XX, XY, YX, and YY. For 3D problems ab stands for XX, XY, XZ, YX, YY, YZ, ZX, ZY, and ZZ. The source term in the equation $src_{\tau ab}$ refers to the momentum stress without the pressure diagonal term.

$$src\_\tau_{xx} = 2\mu\frac{du}{dx} - \frac{2}{3}(\mu - \lambda)(\boldsymbol{\nabla} \cdot \boldsymbol{v})$$

$$src\_\tau_{yy} = 2\mu\frac{dv}{dy} - \frac{2}{3}(\mu - \lambda)(\boldsymbol{\nabla} \cdot \boldsymbol{v})$$

$$src\_\tau_{zz} = 2\mu\frac{dw}{dz} - \frac{2}{3}(\mu - \lambda)(\boldsymbol{\nabla} \cdot \boldsymbol{v})$$

$$src\_\tau_{xy} = src\_\tau_{yx} = \mu(\frac{du}{dy} + \frac{dv}{dx})$$

$$src\_\tau_{xz} = src\_\tau_{zx} = \mu(\frac{du}{dz} + \frac{dw}{dx})$$

$$src\_\tau_{yz} = src\_\tau_{zy} = \mu(\frac{dv}{dz} + \frac{dw}{dy})$$

3.11 Notes on Solid Mechanics

Some of the standard references on solid mechanics include Malvern (1969), Mase (1970), Bonet and Wood (1997) and Belytschko et al. (2004). As is often the case, the mathematical notion used through-out these texts is different in many cases and this is often a source of confusion. Here, we'll lay out some basic defintions in our notation and, when possible, give the notation used in these othere texts.

In what follows, $\boldsymbol{x}$ is the position vector of a material particle in the *deformed* or *current* spatial configuration and $\boldsymbol{X}$ is the position vector of a material particle in the *undeformed* or *initial* or *reference* configuration. The displacement vector $\boldsymbol{d}$ is the difference between these to states[1] viz. $\boldsymbol{x} = \boldsymbol{X} + \boldsymbol{d}$.

We will make extensive use of the gradients of these fields and so it is important to distinguish between gradients in the reference configuration and the current configuration. Gradients in the current configuration are denoted $\boldsymbol{\nabla}$ in Gibbs notation or $\partial\ /\partial x_i$ in index notation. Gradients in the reference configuration are denoted $\boldsymbol{\nabla}_X$ in Gibbs notation or $\partial\ /\partial X_i$ in index notation[2].

Next, we define the deformation gradient $\boldsymbol{F}$

$$\begin{aligned}
\boldsymbol{F} &\equiv \boldsymbol{\nabla}_X \boldsymbol{x}^t && (3.36)\\
&= \boldsymbol{\nabla}_X \boldsymbol{X}^t + \boldsymbol{\nabla}_X \boldsymbol{d}^t && (3.37)\\
&= \boldsymbol{I} + \boldsymbol{\nabla}_X \boldsymbol{d}^t && (3.38)
\end{aligned}$$

where the superscript ^{t} denotes the transpose operator[3]. The inverse deformation gradient[4], $\boldsymbol{F}^{-1}$, is also useful and can be computed as

$$\begin{aligned}
\boldsymbol{F}^{-1} &\equiv \boldsymbol{\nabla} \boldsymbol{X}^t && (3.39)\\
&= \boldsymbol{\nabla} \boldsymbol{x}^t - \boldsymbol{\nabla} \boldsymbol{d}^t && (3.40)\\
&= \boldsymbol{I} - \boldsymbol{\nabla} \boldsymbol{d}^t. && (3.41)
\end{aligned}$$

The determinants of $\boldsymbol{F}$ and $\boldsymbol{F}^{-1}$ are denoted J and J^{-1} respectively and are often used in transformations between different stress definitions[5].

It's worth noting at this point that in Aria both gradient operators, $\boldsymbol{\nabla}$ and $\boldsymbol{\nabla}_X$, are available as Expression objects as are $\boldsymbol{F}$, $\boldsymbol{F}^{-1}$, J and J^{-1}.

The Green or Green-Lagrange strain tensor $\boldsymbol{E}$ is defined[6] as

$$\begin{aligned}
\boldsymbol{E} &\equiv \frac{1}{2}\left(\boldsymbol{F}^t \cdot \boldsymbol{F} - \boldsymbol{I}\right) && (3.42)\\
&= \frac{1}{2}\left(\boldsymbol{\nabla}_X \boldsymbol{d} + \boldsymbol{\nabla}_X \boldsymbol{d}^t + \boldsymbol{\nabla}_X \boldsymbol{d} \cdot \boldsymbol{\nabla}_X \boldsymbol{d}^t\right). && (3.43)
\end{aligned}$$

The Green strain is a strain measure in the reference configuration and is suitable for large deformations and large rotations. The analagous Eulerian (or Almansi's) strain tensor $\boldsymbol{E}^*$ is defined[7]

[1] $\boldsymbol{d}$ is denoted $\boldsymbol{u}$ in Malvern (1969), Mase (1970), Bonet and Wood (1997), and Belytschko et al. (2004).

[2] In Belytschko et al. (2004) $\boldsymbol{\nabla}_X$ is denoted $\boldsymbol{\nabla}_0$.

[3] In Mase (1970) this is called the *conjugate dyadic* and is denoted with a subscript c. In Malvern (1969) the quantity $\boldsymbol{\nabla}_X \boldsymbol{x}^t$ is denoted $\boldsymbol{x}\overleftarrow{\boldsymbol{\nabla}}_X$ where the arrow over the gradient operator denotes the direction of the operation.

[4] In Mase (1970) $\boldsymbol{F}^{-1}$ is denoted $\boldsymbol{H}$.

[5] Sometimes J is expressed as a ratio of the densities between the reference and current configurations, $\rho_\circ/\rho$.

[6] In Mase (1970) $\boldsymbol{E}$ is denoted $\boldsymbol{L}_G$ and is called the *Lagrangian strain.*

[7] In Mase (1970) $\boldsymbol{E}^*$ is denoted $\boldsymbol{E}_A$.

as

$$\boldsymbol{E}^* \equiv \frac{1}{2}\left(\boldsymbol{I} - \boldsymbol{F}^{-t} \cdot \boldsymbol{F}^{-1}\right) \tag{3.44}$$

$$= \frac{1}{2}\left(\nabla \boldsymbol{d} + \nabla \boldsymbol{d}^t - \nabla \boldsymbol{d} \cdot \nabla \boldsymbol{d}^t\right). \tag{3.45}$$

The Eulerian strain is also a suitable strain measure for large deformations and rotations but is defined in the current configuration.

The Cauchy stress, $\boldsymbol{\sigma}$, is a stress measure defined in the current configuration as

$$\boldsymbol{\sigma} = \lambda E_{kk} \boldsymbol{I} + 2\mu \boldsymbol{E} \tag{3.46}$$

where E_{kk} denotes the trace of $\boldsymbol{E}$ and λ and μ are the Lamé coefficients. This constitutive equation may also be augmented with some initial residual stress or a thermal stress,

$$\boldsymbol{\sigma} = \lambda E_{kk} \boldsymbol{I} + 2\mu \boldsymbol{E} - \beta \left(T - T_\circ\right) \boldsymbol{I} + \boldsymbol{\sigma}_r. \tag{3.47}$$

Here β is related to the coefficient of thermal expansion, T is the temperature, $T_\circ$ is the reference temperature of the solid and $\boldsymbol{\sigma}_r$ is the residual stress.

For large deformations and large rotations Aria uses the second Piola-Kirchhoff stress which is defined in the reference configuration and is related to the Cauchy stress as

$$\boldsymbol{S} = J \boldsymbol{F}^{-1} \cdot \boldsymbol{\sigma} \cdot \boldsymbol{F}^{-t}. \tag{3.48}$$

The reverse transformation is readily given by

$$\boldsymbol{\sigma} = J^{-1} \boldsymbol{F} \cdot \boldsymbol{S} \cdot \boldsymbol{F}^t. \tag{3.49}$$

Mathematically, the Cauchy stress $\boldsymbol{\sigma}$ is most conveniently expressed in terms of the Lamé coefficients λ, μ and β. In practice, however, it is more common to measure and report a different but related set of parameters: the Young's modulus E, the Poisson's ratio ν and the coefficient of thermal expansion α. (Note, the shear modulus $G = \mu$.) The relationship between these two sets of parameters is

$$2\mu = \frac{E}{(1+\nu)} \tag{3.50}$$

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)} = 2\mu \frac{\nu}{(1-2\nu)} \tag{3.51}$$

$$\beta = \frac{\alpha E}{(1-2\nu)} = \alpha\left(3\lambda + 2\mu\right) \tag{3.52}$$

In Aria, there are separate `Expression_Names` for the Cauchy stress and the second Piola-Kirchhoff stress expressions. In the input files, users provide their choice of constitutive relations in the material model specification, e.g.,

```
Begin Aria Material The_Material
   Density     = Constant rho = 2.33e-15
   Lambda      = Constant lambda = 52810.30445
   Two Mu      = Constant two_mu = 134426.2295
   Mesh Stress = Nonlinear_Elastic Reference_Frame=Moving
   Mesh Stress = Residual Sx=-11 Sy=-11
   Mesh Stress = Isothermal T=800 T_ref=450
End
```

In this Example, there will be three contributions to the stress in the mesh stress: nonlinear elasticity, a planar residual stress and an isotropic linear thermal stress. Internally, Aria contains a separate expression for transforming these Cauchy stresses into Piola-Kirchhoff stresses, viz.

$$\boldsymbol{S} = J\boldsymbol{F}^{-1} \cdot \left(\sum_i \boldsymbol{\sigma}_i \right) \cdot \boldsymbol{F}^{-t}. \tag{3.53}$$

Note that second Piola-Kirchhoff stresses are not specified in the input file – only Cauchy stresses are. Aria automatically creates an Expression to compute the transform in (3.53).

3.12 Units and Unit Conversions

Aria make no a priori specification concerning the units of each term. However, as with all engineering codes, errors associated with unit conversions are quite easy to do. In some situations, with just one or two driving forces playing a role in a calculation, nondimensionalization of the equations can lead to a simplification of the problem statement (and to increased solution robustness due to proper scaling of the terms in the equations). However, in complicated cases with multiple competing forces and rate constants, sticking to unit systems to specify all quantities frequently leads to less errors in engineering calculations, and also leads to the ability to incorporate third party library packages for specification of source terms and transport properties which necessarily presume to employ units systems in their application programming interfaces (API). The next section discusses the SI units system, and its application for reacting flow and electromagnetic applications.

3.12.1 SI Units

Quantity	**Name**	**Abbreviation**
length	meter (metre)	m
mass	kilogram	kg
time	second	s
electric current	ampere	A
thermodynamic temperature	kelvin	K
amount of substance	kmole	kmol
luminous intensity	candela	cd

Table 3.1. Fundamental SI units.

discuss electromagnetic unit specification, e.g., Gauss's law.

Factor	**Prefix**	**Abbreviation**
10^{24}	yotta	Y
10^{21}	zetta	Z
10^{18}	exa	E
10^{15}	peta	P
10^{12}	tera	T
10^{9}	giga	G
10^{6}	mega	M
10^{3}	kilo	k
10^{2}	hecto	h
10^{1}	deca	da
10^{-1}	deci	d
10^{-2}	centi	c
10^{-3}	milli	m
10^{-6}	micro	μ
10^{-9}	nano	n
10^{-12}	pico	p
10^{-15}	femto	f
10^{-18}	atto	a
10^{-21}	zepto	z
10^{-24}	yocto	y

Table 3.2. SI magnitude prefixes.

Quantity	**Unit**	**Definition**
Frequency	hertz	$\mathrm{Hz} = 1/\mathrm{s}$
Force	newton	$\mathrm{N} = \mathrm{m{\cdot}kg/s^2}$
Pressure, stress	pascal	$\mathrm{Pa} = \mathrm{N/m^2} = \mathrm{kg/m{\cdot}s^2}$
Energy, work, quantity of heat	joule	$\mathrm{J} = \mathrm{N{\cdot}m} = \mathrm{m^2{\cdot}kg/s^2}$
Power, radiant flux	watt	$\mathrm{W} = \mathrm{J/s} = \mathrm{m^2{\cdot}kg/s^3}$
Quantity of electricity, electric charge	coulomb	$\mathrm{C} = \mathrm{s{\cdot}A}$
Electric potential	volt	$\mathrm{V} = \mathrm{W/A} = \mathrm{m^2{\cdot}kg/s^3{\cdot}A}$
Capacitance	farad	$\mathrm{F} = \mathrm{C/V} = \mathrm{s^4{\cdot}A^2/m^2{\cdot}kg}$
Electric resistance	ohm	$\mathrm{Omega} = \mathrm{V/A} = \mathrm{m^2{\cdot}kg/s^3{\cdot}A^2}$
Conductance	siemens	$\mathrm{S} = \mathrm{A/V} = \mathrm{s^3{\cdot}A^2/m^2{\cdot}kg}$
Magnetic flux	weber	$\mathrm{Wb} = \mathrm{V{\cdot}s} = \mathrm{m^2\ kg/s^2{\cdot}A}$
Magnetic flux density, magnetic induction	tesla	$\mathrm{T} = \mathrm{Wb/m^2} = \mathrm{kg/s^2{\cdot}A}$
Inductance	henry	$\mathrm{H} = \mathrm{Wb/A} = \mathrm{m^2\ kg/s^2{\cdot}A^2}$
Luminous flux	lumen	$\mathrm{lm} = \mathrm{cd{\cdot}sr}$
Illuminance	lux	$\mathrm{lx} = \mathrm{lm/m^2} = \mathrm{cd{\cdot}sr/m^2}$
Activity (ionizing radiations)	becquerel	$\mathrm{Bq} = 1/\mathrm{s}$
Absorbed dose	gray	$\mathrm{Gy} = \mathrm{J/kg} = \mathrm{m^2/s^2}$
Dynamic viscosity	pascal second	$\mathrm{Pa{\cdot}s} = \mathrm{kg/m{\cdot}s}$
Moment of force	meter newton	$\mathrm{N{\cdot}m} = \mathrm{m^2{\cdot}kg/s^2}$
Surface tension	newton per meter	$\mathrm{N/m} = \mathrm{kg/s^2}$
Heat flux density, irradiance	watt per square meter	$\mathrm{W/m^2} = \mathrm{kg/s^3}$
Heat capacity, entropy	joule per kelvin	$\mathrm{J/K} = \mathrm{m^2{\cdot}kg/s^2{\cdot}K}$
Specific heat capacity, specific entropy	joule per kilogram kelvin	$\mathrm{J/kg\ K} = \mathrm{m^2/s^2{\cdot}K}$
Specific energy	joule per kilogram	$\mathrm{J/kg} = \mathrm{m^2/s^2}$
Thermal conductivity	watt per meter kelvin	$\mathrm{W/m{\cdot}K} = \mathrm{m{\cdot}kg/s^3{\cdot}K}$
Energy density	joule per cubic meter	$\mathrm{J/m^3} = \mathrm{kg/m\ s^2}$
Electric field strength	volt per meter	$\mathrm{V/m} = \mathrm{m{\cdot}kg/s^3{\cdot}A}$
Electric charge density	coulomb per cubic meter	$\mathrm{C/m^3} = \mathrm{s{\cdot}A/m^3}$
Electric displacement, electric flux density	coulomb per square meter	$\mathrm{C/m^2} = \mathrm{s{\cdot}A/m^2}$
Permittivity	farad per meter	$\mathrm{F/m} = \mathrm{s^4{\cdot}A^2/m^3{\cdot}kg}$
Permeability	henry per meter	$\mathrm{H/m} = \mathrm{m{\cdot}kg/s^2{\cdot}A^2}$
Molar energy	joule per kmol	$\mathrm{J/kmol} = \mathrm{m^2kg/s^2{\cdot}kmol}$
Molar entropy, molar heat capacity	joule per kmol kelvin	$\mathrm{J/kmol\ K} = \mathrm{m^2kg/s^2{\cdot}K{\cdot}kmol}$
Exposure (ionizing radiations)	coulomb per kilogram	$\mathrm{C/kg} = \mathrm{s{\cdot}A/kg}$
Absorbed dose rate	gray per second	$\mathrm{Gy/s} = \mathrm{m^2/s^3}$

Table 3.3. SI derived units and their definitions.

Quantity	Unit
erg	1 erg = 10^{-7} J
dyne	1 dyn = 10^{-5} N
poise	1 P = 1 dyn·s/cm^2 = 0.1 Pa·s
stokes	1 St = 1 cm^2/s = 10^{-4} m^2/s
gauss	1 G = 10^{-4} T
oersted	1 Oe = (1000/(4 pi)) A/m
maxwell	1 Mx = 10^{-8} Wb
stilb	1 sb = 1 cd/cm^2 = 10^4 cd/m^2
phot	1 ph = 10^4 lx

Table 3.4. CGS derived units and their definitions.

3.12.2 Common Units and Conversion Factors

1 g·/s^2 = 1 dyn = 10^{-5} kg·m/s^2 = 10^{-5} N
1 g·/s^2 = 7.2330×10^{-5} lb$_\mathrm{m}$·ft/s^2 (poundal)
1 lb$_\mathrm{f}$ = 4.4482 N
1 g·/s^2 = 2.2481×10^{-6}lb$_\mathrm{f}$

Table 3.5. Units and conversion factors for force.

1 bar = 10^5 Pa = 10^5 N/m^2
1 psia = 1 lb$_\mathrm{f}$/in^2
1 psia = 2.0360 in Hg at 0 °C
1 psia = 2.311 ft H_2O at 70 °F
1 psia = 51.715 mm Hg at 0 °C (ρ_{Hg} = 13.5955 g/cm^3)
1 atm = 14.696 psia = 1.01325×10^5N/m^2 = 1.01325 bar
1 atm = 760 mm Hg at 0 °C = 1.01325×10^5 Pa
1 atm = 29.921 in Hg at 0 °C
1 atm = 33.90 ft H_2O at 4 °C

Table 3.6. Units and conversion factors for pressure and stress.

1 cp = 10^{-2} g/cm· s = 10^{-2} Poise
1 cp = 2.4191 lb$_\mathrm{m}$/ft·h
1 cp = 6.7197×10^{-4} lb$_\mathrm{m}$/ft·s
1 cp = 10^{-3} Pa·s = 10^{-3} kg/m·s = 10^{-3} N·/m^2
1 cp = 2.0886×10^{-5} lb$_\mathrm{f}$ · s/ft^2
1 Pa·s = 1 N·s/m^2 = 1 kg/m·s = 1000 cp = 0.67197 lb$_\mathrm{m}$/ft·s

Table 3.7. Units and conversion factors for viscosity.

1 g/cm^3 = 1000 kg/m^3 = 62.43 lb$_m$/ft^3
1 g/cm^3 = 8.345 lb$_m$/U.S. gal
1 lb$_m$/ft^3 = 16.0185 kg/m^3

Table 3.8. Units and conversion factors for density.

1 btu/h·ft·°F = 4.1365×10^{-3} cal/s·cm·°C
1 btu/h·ft·°F = 1.73073 W/m·K

Table 3.9. Units and conversion factors for thermal conductivity.

1 btu/h·ft^2·°F = 1.3571×10^{-4} cal/s·cm^2·°C
1 btu/h·ft^2·°F = 5.6783×10^{-4} W/cm^2·°C
1 btu/h·ft^2·°F = 5.6783 W/m^2·°C
1 kcal/h·m^2·°F = 0.2048 btu/h·ft^2·°F

Table 3.10. Units and conversion factors for heat-transfer coefficients.

Chapter 4

Equation Specification

This chapter will document all of the EQ line commands within the current version of Aria. EQ line commands add equations and independent variables to Aria's specification of the equation set to be solved for within each region. The equation is also associated with a field variable here that becomes part of the solution vector for Aria. EQ cards occur in Aria's input file within Region blocks.

¿¿¿¿Each equation should really point to an equation number in the preceding section, to a section of the manual, or to an external reference¡¡¡¡

The EQ card add an equation to be solved for on a particular *MESH_PART* . The format is as follows

EQ equation FOR *DOF* on *MESH_PART* using *INTERP* with $TERM_0$ . . $TERM_n$

Equation is the string identifier for the individual equations listed in the previous chapter. The format for the equation string identifiers is listed in a subsection of Chapter 2. The *DOF* keyword specifies the independent unknown that is solved for in order to satisfy the equation. Normally, it is a strict function of the equation keyword. In other words, the temperature is the only valid *DOF* entry if the energy equation is being solved. *MESH_PART* is usually the name of an active element block in the finite element model. Unfortunately, if an equation is to be solved on the entire finite element model, this means that there must be multiple EQ keywords for each element block defined in the mesh.

INTERP defines the finite element interpolation to be used. Currently the valid entries for this keyword are P0, P1, Q1, Q2, QS2, T1, and T2. However, in some combinations, various interpolations are not permitted for some variables.

$TERM_n$ refer to the broad categories for the terms in a general advection-diffusion continuity equation. Each term in the equation must be explicitly turned on for it to appear in the conservation equation. Admissible values of TERM are MASS, ADV, DIFF, SRC, and XFER.

XFER refers to the following case.

4.1 EQ CONTINUITY

Syntax	`EQ CONTINUITY[_<Subindex>][_<Phase>] for` *DOF* `ON` *MESH_PART* `USING` *INTERP* `with` $TERM_0$ . . $TERM_n$
Description	Activates the continuity equation 3.4.

Details The only admissible value for ***DOF*** is PRESSURE.

Admissible values of ***INTERP*** are Q_1, Q_2, QS_2, P_1, P_0, T_1 and T_2.

Admissible values of $TERM_n$ are MASS, DIV, ADV, SRC, and XFER.

With the exception of XFER these terms are described in the Equations Aria Solves section 3 of the manual.

The ***MESH_PART*** must be an active element block.

Parent Block(s) `ARIA_REGION`

4.2 EQ CURRENT

Syntax `EQ CURRENT[_<Subindex>][_<Phase>] for` ***DOF*** `ON` ***MESH_PART*** `USING` ***INTERP*** `with` $TERM_0$ . . $TERM_n$

Description Activates the electrical current equation 3.31.

Details The only admissible value for ***DOF*** is VOLTAGE.

Admissible values of ***INTERP*** are Q_1, Q_2, QS_2, T_1 and T_2.

Admissible values of $TERM_n$ are DIFF, SRC, and XFER.

With the exception of XFER these terms are described in the Equations Aria Solves section 3 of the manual.

The ***MESH_PART*** must be an active element block.

Parent Block(s) `ARIA_REGION`

4.3 EQ ENERGY

Syntax `EQ ENERGY[_<Subindex>][_<Phase>] for` ***DOF*** `ON` ***MESH_PART*** `USING` ***INTERP*** `with` $TERM_0$ . . $TERM_n$

Description Activates the energy conservation transport equation 3.9.

Details — Admissible values for ***DOF*** are TEMPERATURE and ENTHALPY.

Admissible values of ***INTERP*** are Q_1, Q_2, QS_2, T_1 and T_2.

Admissible values of $TERM_n$ are MASS, ADV, DIFF, SRC, and XFER.

With the exception of XFER these terms are described in the Equations Aria Solves section[3.3] of the manual.

The ***MESH_PART*** must be an active element block.

Parent Block(s) `ARIA_REGION`

4.4 EQ LEVEL_SET

Syntax — `EQ LEVEL_SET[_<Subindex>] for` ***DOF*** `ON` ***MESH_PART*** `USING` ***INTERP*** `with` $TERM_0$ . . $TERM_n$

Description — Activates the level set (distance function) equation.

Not Ready for General Use

Details — The only admissible value for ***DOF*** is LEVEL_SET.

Admissible values of ***INTERP*** are Q_1, Q_2, QS_2, T_1 and T_2.

Admissible values of $TERM_n$ are MASS, ADV, DIFF, SRC, and XFER.

With the exception of XFER these terms are described in the Equations Aria Solves section 3 of the manual.

The ***MESH_PART*** must be an active element block.

Parent Block(s) `ARIA_REGION`

4.5 EQ MESH

Syntax — `EQ MESH[_<Subindex>] for` ***DOF*** `ON` ***MESH_PART*** `USING` ***INTERP*** `with` $TERM_0$ . . $TERM_n$

Description — Activates the pseudo-solid mesh equation 3.22.

Details The only admissible value for *DOF* is MESH_DISPLACEMENTS.

Admissible values of *INTERP* are Q_1, Q_2, QS_2, T_1 and T_2.

Admissible values of $TERM_n$ are DIFF, TSTRAIN, SRC, and XFER.

With the exception of XFER these terms are described in the Equations Aria Solves section 3 of the manual.

The *MESH_PART* must be an active element block.

Parent Block(s) `ARIA_REGION`

4.6 EQ MOMENTUM

Syntax `EQ MOMENTUM[_<Subindex>][_<Phase>] for` *DOF* `ON` *MESH_PART* `USING` *INTERP* `with` $TERM_0$ . . $TERM_n$

Description Activates the fluid momentum equation 3.19.

Details The only admissible value for *DOF* is VELOCITY.

Admissible values of *INTERP* are Q_1, Q_2, QS_2, T_1 and T_2.

Admissible values of $TERM_n$ are MASS, ADV, DIFF, SRC, and XFER.

With the exception of XFER these terms are described in the Equations Aria Solves section 3 of the manual.

The *MESH_PART* must be an active element block.

Parent Block(s) `ARIA_REGION`

4.7 EQ SHEAR

Syntax `EQ SHEAR[_<Subindex>][_<Phase>] for` *DOF* `ON` *MESH_PART* `USING` *INTERP* `with` $TERM_0$ . . $TERM_n$

Description Activates the shear-rate definition/intermediate equation.

Details The only admissible value for *DOF* is GAMMA_DOT.

Admissible values of *INTERP* are Q_1, Q_2, QS_2, T_1 and T_2.

Admissible values of $TERM_n$ are DEF, SRC, and XFER.

With the exception of XFER these terms are described in the Equations Aria Solves section 3 of the manual.

The *MESH_PART* must be an active element block.

Parent Block(s) `ARIA_REGION`

4.8 EQ SOLID

Syntax `EQ SOLID[_<Subindex>][_<Phase>] for` *DOF* `ON` *MESH_PART* `USING` *INTERP* `with` $TERM_0$ . . $TERM_n$

Description Activates the solid momentum equation 3.22.

Details The only admissible value for *DOF* is SOLID_DISPLACEMENTS.

Admissible values of *INTERP* are Q_1, Q_2, QS_2, T_1 and T_2.

Admissible values of $TERM_n$ are DIFF, TSTRAIN, SRC, and XFER.

With the exception of XFER these terms are described in the Equations Aria Solves section 3 of the manual.

The *MESH_PART* must be an active element block.

Parent Block(s) `ARIA_REGION`

4.9 EQ SPECIES

Syntax `EQ SPECIES[_<Subindex>][_<Phase>] for` *DOF* `ON` *MESH_PART* `USING` *INTERP* `with` $TERM_0$ . . $TERM_n$

Description Activates the species transport equation 3.13.

Details The only admissible value for *DOF* is SPECIES.

Admissible values of *INTERP* are Q_1, Q_2, QS_2, T_1 and T_2.

Admissible values of $TERM_n$ are MASS, ADV, DIFF, SRC, FRACBAL and XFER. With the exception of XFER these terms are described in the Equations Aria Solves section 3 of the manual. The FRACBAL term may not be included with other terms.

Some other things to note about species equations in Aria.

- The FRACBAL term may be assigned to any species number.
- Species numbers in Aria are arbitrary; they may start at any value and need not be continuous.
- For the species fraction balances, Aria will automatically detect all species that are present in the problem and include them in the balance.

The *MESH_PART* must be an active element block.

Parent Block(s) `ARIA_REGION`

4.10 EQ SUSPENSON

Syntax `EQ SUSPENSION[_<Subindex>][_<Phase>] for` *DOF* `ON` *MESH_PART* `USING` *INTERP* `with` $TERM_0$ . . $TERM_n$

Description Activates the suspension transport equation 3.34.

Details The only admissible value for *DOF* is PHI.

Admissible values of *INTERP* are Q_1, Q_2, QS_2, T_1 and T_2.

Admissible values of $TERM_n$ are ADV, DIFF, SRC, and XFER.

With the exception of XFER these terms are described in the Equations Aria Solves section 3 of the manual.

The *MESH_PART* must be an active element block.

Parent Block(s) `ARIA_REGION`

4.11 EQ VOLTAGE

Syntax `EQ VOLTAGE[_<Subindex>][_<Phase>] for` *DOF* `ON` *MESH_PART* `USING` *INTERP* `with` $TERM_0$ . . $TERM_n$

Description Activates the voltage equation (electric-displacement formulation) 3.28. See also the CURRENT equation.

Details The only admissible value for ***DOF*** is VOLTAGE.

Admissible values of ***INTERP*** are Q_1, Q_2, QS_2, T_1 and T_2.

Admissible values of $TERM_n$ are DIFF, SRC, and XFER.

With the exception of XFER these terms are described in the Equations Aria Solves section 3 of the manual.

The ***MESH_PART*** must be an active element block.

Parent Block(s) `ARIA_REGION`

4.12 EQ Stress_Tensor_Projection

Syntax `EQ Stress_Tensor_Projection[_<Subindex>][_<Phase>] for` ***DOF*** `ON` ***MESH_PART*** `USING` ***INTERP*** `with` $TERM_0$ . . $TERM_n$

Description Activates the Stress Tensor Projection equation.

Details The only admissible value for ***DOF*** is Stress_Tensor.

Admissible values of ***INTERP*** are Q_1, Q_2, QS_2, T_1 and T_2.

The only admissible values of $TERM_n$ are DEF.

With the exception of XFER these terms are described in the Equations Aria Solves section 3 of the manual.

The ***MESH_PART*** must be an active element block.

Parent Block(s) `ARIA_REGION`

4.13 ELASTICITY FORMULATION

Syntax `ELASTICITY FORMULATION = PLANE` ***TYPE***

Description Assigns the elasticity formulation ***TYPE*** for two-dimensional problems involving the MESH and SOLID equation 3.22.

Details Allowable formulation PLANE ***TYPE*** is PLANE STRESS or PLANE STRAIN.

Parent Block(s) `ARIA_REGION`

4.14 PRESSURE STABILIZATION

Syntax `PRESSURE STABILIZATION IS` *`TYPE`* `WITH SCALING =` C

Description Prescribe a stabilization technique for solving the `MOMENTUM` 3.19 and `CONTINUITY` 3.4 equations with equal order interpolation.

Details Aria supports both PSPG (Pressure Stabilizide Petrov-Gelerkin) and PSPP (Pressure Stabilized Pressure Projection) stabilization techniques for solving the `MOMENTUM` and `CONTINUITY` equations with equal order interpolation.

Valid options for the *`TYPE`* specification are `NO_STABILIZATION`, `PSPG_CONSTANT`, `PSPG_LOCAL`, `PSPG_GLOBAL` and `PSPP_CONSTANT`.

`NO_STABILIZATION` disables any stabilization.

`PSPP_CONSTANT` results in the recently developed stabilization technique of Dohrmann and Bochev (2004) and Bochev et al. (2006).

In the PSPG forms of stabilization, introduced by Hughes et al. (1986), terms from the momentum equation are added to the continuity equation scaled by a multiplier, α. The exact form of the multipler depend on a global Reynolds number that is defined as

$$Re \equiv \frac{\rho|\boldsymbol{v}|\langle h\rangle}{2\mu} \tag{4.1}$$

Here, ρ is the density, μ is the viscosity, $|\boldsymbol{v}|$ is a velocity scale and $\langle h\rangle$ is an element length scale. Armed with Re, the stabilization multipler α is defined in one of two ways.

$$Re \leq 3 \quad : \quad \alpha \equiv \frac{\tau\langle h\rangle^2}{12\mu} \tag{4.2}$$

$$Re > 3 \quad : \quad \alpha \equiv \frac{\tau\langle h\rangle}{2\rho|\boldsymbol{v}|} \tag{4.3}$$

NB: Currently, Aria always uses the low-Reynolds number for of α, as defined in (4.2). The `PSPG_LOCAL` method computes $|\boldsymbol{v}|$ and $\langle h\rangle$ within each element. The `PSPG_GLOBAL` method computes $|\boldsymbol{v}|$ and $\langle h\rangle$ as averages over all of the elements with the `MOMENTUM` equation defined. The `PSPG_CONSTANT` gives $|\boldsymbol{v}|$ and $\langle h\rangle$ a value of 1 (one) and just uses the scale factor.

Parent Block(s) `ARIA_REGION`

4.15 MESH MOTION

Syntax `MESH MOTION IS` *`TYPE`* `ON` *`MESH_PART`*

Description Defines the type of mesh motion to be used in the simulation.

Details Admissible values of ***TYPE*** are ARBITRARY, LAGRANGIAN, and TOTAL_ALE. When ***TYPE*** is ARBITRARY or TOTAL_ALE the MESH equation must be active. When ***TYPE*** is LAGRANGIAN or TOTAL_ALE the SOLID equation must be active. The ***MESH_PART*** must be an active element block.

Parent Block(s) `ARIA_REGION`

4.16 SAVE RESIDUALS

Syntax `SAVE RESIDUALS =` *`MODE`*

Description Causes Aria to save the residuals to a field with the prefix **`residaul->`**, e.g., **`residual->Temperature`**. This will be done for all fields (though we could make it a per-field option). Valid choices are `OFF` (default), `BEFORE_BCS` and `AFTER_BCS`.

Details Causes Aria to save the residuals to a field with the prefix **`residaul->`**, e.g., **`residual->Temperature`**. This will be done for all fields (though we could make it a per-field option). Valid choices are `OFF` (default), `BEFORE_BCS` and `AFTER_BCS`.

For the choice `BEFORE_BCS` the residuals will be saved at the point in the assembly process where the primary equations have been assembled but prior to the assembly of any boundary conditions or distinguishing conditions. For the choice of `AFTER_BCS` the residuals will be saved after all BCs and distinguishing conditions have been applied. The default, `OFF`, is to not save the residuals.

This feature is only applicable when using the `NEWTON` nonlinear solution strategy.

Parent Block(s) `ARIA_REGION`

4.17 INTEGRATION RULE

Syntax `INTEGRATION RUL for` *`block_name`* `=` *`INT`*

Description Overrides the default integration rule for the equations defined on *`block_name`*.

Details Overrides the default integration rule for the equations defined on *`block_name`*.

Parent Block(s) `ARIA_REGION`

Chapter 5

Initial Conditions

5.1 IC CIRC_X

Syntax IC CIRC_X AT *MESH_PART* *DOF* AMP = Ω

Description Initial boundary condition for the x component of a vector variable with constant tangential magnitude along circles of radius $r(x, y)$ defined on the mesh entity.

Details Sets *DOF* to the provided value on nodeset associated with the name *MESH_PART* according to the relation

$$\mathit{DOF} = -\Omega r(x, y) \sin\theta. \tag{5.1}$$

The rotation Ω is assumed to be about the vector $(0, 0, 1)$ passing through point $(0, 0, 0)$.

Parent Block(s) ARIA_REGION

5.2 IC CIRC_Y

Syntax IC CIRC_Y AT *MESH_PART* *DOF* AMP = Ω

Description Initial boundary condition for the y component of a vector variable with constant tangential magnitude along circles of radius $r(x, y)$ defined on the mesh entity.

Details Sets *DOF* to the provided value on nodeset associated with the name *MESH_PART* according to the relation

$$\mathit{DOF} = \Omega r(x, y) \cos\theta. \tag{5.2}$$

The rotation Ω is assumed to be about the vector $(0, 0, 1)$ passing through point $(0, 0, 0)$.

Parent Block(s) ARIA_REGION

5.3 IC CONSTANT

Syntax `IC CONST AT` *`MESH_PART`* *`DOF`* `=` *`REAL`*

Description Constant initial condition.

Details Sets *DOF* to the provided constant value on mesh entity associated with the name *MESH_PART*.

Parent Block(s) `ARIA_REGION`

5.4 IC COUETTE_X

Syntax `IC COUETTE_X AT` *`MESH_PART`* *`DOF`* `PARAMS =` Ω r_i r_o

Description Initial condition for x component of a vector variable in a Couette device with inner radius r_i, outer radius r_o and driven at angular velocity Ω, that varies spatially over a mesh entity.

Details Sets *DOF* to vary spatially over the nodeset associated with the name *MESH_PART* according to the relation

$$\mathit{DOF} = -\Omega r(x,y)\frac{r_i r_o}{r_o^2 - r_i^2}\sin\theta. \tag{5.3}$$

The rotation Ω is assumed to be about the vector $(0,0,1)$ passing through point $(0,0,0)$.

Parent Block(s) `ARIA_REGION`

5.5 IC COUETTE_Y

Syntax `IC COUETTE_Y AT` *`MESH_PART`* *`DOF`* `PARAMS =` Ω r_i r_o

Description Initial condition for x component of a vector variable in a Couette device with inner radius r_i, outer radius r_o and driven at angular velocity Ω, that varies spatially over a mesh entity.

Details Sets *DOF* to vary spatially over the nodeset associated with the name *MESH_PART* according to the relation

$$\mathit{DOF} = \Omega r(x,y)\frac{r_i r_o}{r_o^2 - r_i^2}\cos\theta. \tag{5.4}$$

The rotation Ω is assumed to be about the vector $(0,0,1)$ passing through point $(0,0,0)$.

Parent Block(s) `ARIA_REGION`

5.6 IC COUETTE_SH

Syntax `IC COUETTE_SH AT` *`MESH_PART`* *`DOF`* `PARAMS =` $\Omega\ r_i\ r_o$

Description Initial condition for generalized shear rate in a Couette device with inner radius r_i, outer radius r_o and driven at angular velocity Ω, that varies spatially over a mesh entity.

Details Sets *`DOF`* to the provided value on nodeset associated with the name *`MESH_PART`* according to the relation

$$\mathit{DOF} = \frac{\Omega}{r^2(x,y)} \frac{r_i r_o}{r_o^2 - r_i^2} \left(\frac{1}{r(x,y)} - \frac{r(x,y)}{r_o^2} \right) \tag{5.5}$$

The rotation Ω is assumed to be about the vector $(0,0,1)$ passing through point $(0,0,0)$.

Parent Block(s) `ARIA_REGION`

5.7 IC READ_FILE

Syntax `IC Read_File` *`DOF`* `=` *`STRING`*

Description This IC command will initialize the field *`DOF`* with values from the field with the name given by the *`STRING`* argument in the input mesh database.

Details This IC command will initialize the field *`DOF`* with values from the field with the name given by the *`STRING`* argument in the input mesh database. For example, if your input mesh database (the file referenced in the `FINITE ELEMENT MODEL`) contains a vector field named `U` then you could initialize a velocity field by setting *`DOF`* to `VELOCITY` and the *`STRING`* argument to `U`.

Parent Block(s) `ARIA_REGION`

5.8 IC LINEAR

Syntax `IC LINEAR AT` *`MESH_PART`* *`DOF`* `COEF =` $C_0\ C_1\ C_2\ C_3$

Description Initial condition that varies spatially over a mesh entity in a linear fashion.

Details Sets *DOF* to the provided value on nodeset associated with the name *MESH_PART* in the following manner

$$DOF = C_0 + C_1x + C_2y + C_3z. \tag{5.6}$$

Parent Block(s) `ARIA_REGION`

5.9 IC PARAB

Syntax `IC PARAB AT` *MESH_PART* *DOF* `COEF =` C_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8 C_9

Description Initial condition that varies spatially over a mesh entity in a parabolic fashion.

Details Sets *DOF* to the provided value on nodeset associated with the name *MESH_PART* in the following manner

$$DOF = C_0 + C_1x + C_2y + C_3z + C_4xy + C_5xz + C_6yz + C_7x^2 + C_8y^2 + C_9z^2. \tag{5.7}$$

Parent Block(s) `ARIA_REGION`

Chapter 6

Boundary Conditions

This chapter documents the boundary condition, BC, line commands within the current version of Aria. In what follows, Dirichlet boundary conditions are first described and are followed by flux boundary condition descriptions.

6.1 BC CIRC_X

Syntax `BC CIRC_X DIRICHLET AT` ***`MESH_PART DOF`*** `AMP =` Ω

Description Dirichlet boundary condition for the x component of a vector variable with constant tangential magnitude along circles of radius $r(x, y)$ defined on the mesh entity.

Details Sets ***`DOF`*** to the provided value on nodeset associated with the name ***`MESH_PART`*** according to the relation

$$\mathit{DOF} = -\Omega r(x, y) \sin\theta. \tag{6.1}$$

The rotation Ω is assumed to be about the vector $(0, 0, 1)$ passing through point $(0, 0, 0)$.

Parent Block(s) `ARIA_REGION`

6.2 BC CIRC_Y

Syntax `BC CIRC_Y DIRICHLET AT` ***`MESH_PART DOF`*** `AMP =` Ω

Description Dirichlet boundary condition for the y component of a vector variable with constant tangential magnitude along circles of radius $r(x, y)$ defined on the mesh entity.

Details Sets ***`DOF`*** to the provided value on nodeset associated with the name ***`MESH_PART`*** according to the relation

$$\mathit{DOF} = \Omega r(x, y) \cos\theta. \tag{6.2}$$

The rotation Ω is assumed to be about the vector $(0, 0, 1)$ passing through point $(0, 0, 0)$.

Parent Block(s) `ARIA_REGION`

6.3 BC CONST

Syntax `BC CONST DIRICHLET AT` ***`MESH_PART DOF`*** `=` ***`REAL`***

Description Constant Dirichlet condition.

Details Sets ***`DOF`*** to the provided constant value on mesh entity associated with the name ***`MESH_PART`***.

Parent Block(s) `ARIA_REGION`

6.4 BC LINEAR

Syntax `BC LINEAR DIRICHLET AT` ***`MESH_PART DOF`*** `COEF =` C_0 C_1 C_2 C_3

Description Dirichlet boundary condition that varies spatially over a mesh entity in a linear fashion.

Details Sets ***`DOF`*** to the provided value on nodeset associated with the name ***`MESH_PART`***in the following manner

$$\mathit{DOF} = C_0 + C_1 x + C_2 y + C_3 z. \tag{6.3}$$

Parent Block(s) `ARIA_REGION`

6.5 BC LINEAR_IN_TIME

Syntax `BC LINEAR_IN_TIME DIRICHLET AT` ***`MESH_PART DOF`*** `=` C_0 C_1

Description Dirichlet boundary condition whose value is a linear function in time.

Details Sets ***`DOF`*** to the provided value on nodeset associated with the name ***`MESH_PART`***according to the relation

$$\mathit{DOF} = C_0 + C_1 t \tag{6.4}$$

Parent Block(s) `ARIA_REGION`

6.6 BC PARAB

Syntax `BC PARAB DIRICHLET AT` ***MESH_PART DOF*** `COEF =` C_0 C_1 C_2 C_3 C_4 C_5 C_6 C_7 C_8 C_9

Description Dirichlet boundary condition for ***DOF*** that varies spatially over a mesh entity in a parabolic fashion.

Details Sets ***DOF*** to the provided value on nodeset associated with the name ***MESH_PART*** in the following manner

$$\textit{DOF} = C_0 + C_1x + C_2y + C_3z + C_4xy + C_5xz + C_6yz + C_7x^2 + C_8y^2 + C_9z^2. \quad (6.5)$$

Parent Block(s) `ARIA_REGION`

6.7 BC PERIODIC_LINEAR_IN_TIME

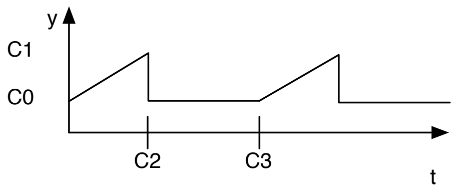
Syntax `BC PERIODIC_LINEAR_IN_TIME DIRICHLET AT` ***MESH_PART DOF*** `COEF =` C_0 C_1 C_2 C_3

Description Dirichlet boundary condition that provides a periodic linear (ramp) function in time over a portion of the local time period $\tau = C_3$.

Details The value of ***DOF*** is given by a periodic linear (ramp) function in time, within a local time interval τ as illustrated in figure below.

$$\textit{DOF} = \begin{cases} C_0 + C_1 t_p & t_p \le t_d \\ C_0 & \textit{otherwise} \end{cases} \quad (6.6)$$

where $t_p = t - \tau \mathrm{Int}(t/\tau)$ is a local time period and $t_d = C_2$ is the dwell time within this time period. Note that $t_d < \tau$ in order for the boundary condition to be uniquely defined.



Parent Block(s) `ARIA_REGION`

6.8 BC PERIODIC_STEP_IN_TIME

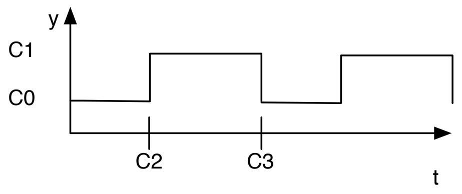
Syntax BC PERIODIC_STEP_IN_TIME DIRICHLET AT *MESH_PART* *DOF* COEF = C_0 C_1 C_2 C_3

Description Dirichlet boundary condition that provides a periodic step function in time over a portion of the local time period $\tau = C_3$.

Details The value of *DOF* is given by a periodic step function in time, as illustrated in figure below.

$$\mathit{DOF} = \begin{cases} C_0 & t_p \le t_d \\ C_1 & otherwise \end{cases} \tag{6.7}$$

where $t_p = t - \tau \text{Int}(t/\tau)$ is a local period time and $t_d = C_2$ is the dwell time within this time period. Note that $t_d < \tau$ in order for the boundary condition to be uniquely defined.



Parent Block(s) ARIA_REGION

6.9 BC ROTATING_X

Syntax BC ROTATING_X DIRICHLET AT *MESH_PART* *DOF* Omega = Ω C_0 C_1 C_2

Description Dirichlet boundary condition that applies the x-component of a rotation in time about the z-axis.

Details Sets *DOF* to the provided value on nodeset associated with the name *MESH_PART* according to the relation

$$\mathit{DOF\_X} = X_0 (\cos \omega t - 1) - Y_0 \sin \omega t \tag{6.8}$$

The rotation Ω is assumed to be about the vector $(0, 0, 1)$ passing through point $(0, 0, 0)$.

Parent Block(s) ARIA_REGION

6.10 BC ROTATING_Y

Syntax BC ROTATING_X DIRICHLET AT *MESH_PART* *DOF* Omega = Ω C_0 C_1 C_2

Description Dirichlet boundary condition that applies the y-component of a rotation in time about the z-axis.

Details Sets *DOF* to the provided value on nodeset associated with the name *MESH_PART* according to the relation

$$\mathit{DOF\_Y} = X_0 \sin \omega t + Y_0 (\cos \omega t - 1) \tag{6.9}$$

The rotation Ω is assumed to be about the vector $(0, 0, 1)$ passing through point $(0, 0, 0)$.

Parent Block(s) ARIA_REGION

6.11 BC TRANSLATE

Syntax BC TRANSLATE DIRICHLET AT *MESH_PART* *DOF* SCALE = C_0

Description Dirichlet boundary condition for translating a value in time.

Details Sets *DOF* to the provided value on nodeset associated with the name *MESH_PART* according to the relation

$$\mathit{DOF} = C_0 t \tag{6.10}$$

Parent Block(s) ARIA_REGION

6.12 BC UNIDIRECTIONAL_FLOW_X

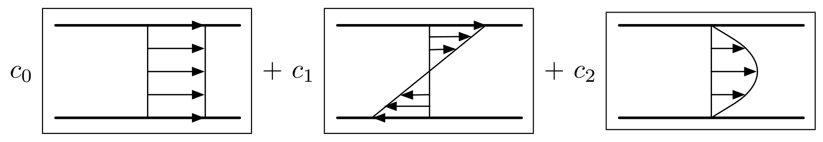
Syntax BC UNIDIRECTIONAL_FLOW_X DIRICHLET AT *MESH_PART* *DOF* coef = c_0 c_1 c_2

Description A specialized boundary condition for conveniently specifying inflow boundary conditions on simple geometries.

Details This boundary condition is really meant for specifying inflow velocity profiles in a convenient way. The three coefficients provide the magnitudes of plug, shear and parabolic flow as a function of the y-coordinate. Specifically,

$$\mathit{DOF} = c_0 + c_1 \frac{y - y_\circ}{H} + c_2 \left[1 - \left(\frac{y - y_\circ}{H} \right)^2 \right] \tag{6.11}$$

Here $y_\circ$ is the y-coordinate of the middle of the surface (sideset) or nodelist (nodeset) and H is the half-width of the surface or nodelist. Both $y_\circ$ and H are automatically calculated by Aria. This combination of flows is represented graphically as



Parent Block(s) `ARIA_REGION`

6.13 BC UNIDIRECTIONAL_FLOW_Y

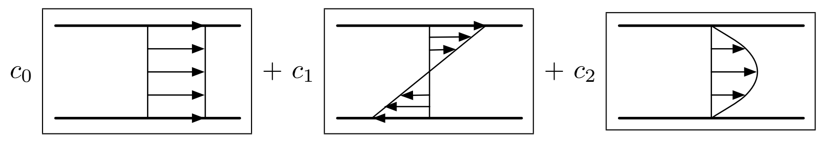
Syntax `BC UNIDIRECTIONAL_FLOW_Y DIRICHLET AT` *`MESH_PART DOF`* `coef =` c_0 c_1 c_2

Description A specialized boundary condition for conveniently specifying inflow boundary conditions on simple geometries.

Details This boundary condition is really meant for specifying inflow velocity profiles in a convenient way. The three coefficients provide the magnitudes of plug, shear and parabolic flow as a function of the x-coordinate. Specifically,

$$\mathit{DOF} = c_0 + c_1 \frac{x - x_\circ}{H} + c_2 \left[1 - \left(\frac{x - x_\circ}{H} \right)^2 \right] \tag{6.12}$$

Here $x_\circ$ is the x-coordinate of the middle of the surface (sideset) or nodelist (nodeset) and H is the half-width of the surface or nodelist. Both $x_\circ$ and H are automatically calculated by Aria. This combination of flows is represented graphically as



Parent Block(s) `ARIA_REGION`

6.14 BC FLUX

Syntax: `BC FLUX FOR` ***`EQNAME`*** `ON` ***`MESH_PART`*** `=` ***`MODEL`*** `[`$\texttt{param}_1$ `=` $\texttt{val}_1$`,` $\texttt{param}_2$ `=` $\texttt{val}_2$ `...]`

Description: Neumann boundary condition that sets the surface normal flux of the degree of freedom associated with equation ***`EQNAME`*** to that provided by the specified ***`MODEL`***.

Details: *`MODEL`* the supplies a diffusive flux $f_n \equiv \boldsymbol{n} \cdot \boldsymbol{f}$ in accordance with equation 3.3. I.e., it adds the surface integral

$$\int_{\mathit{MESH\_PART}} \boldsymbol{n} \cdot \boldsymbol{f} \phi^i \, \mathrm{dS} \tag{6.13}$$

to the residual for equation ***`EQNAME`***. See, e.g., q_n in equation 3.10.

Parent Block(s): `ARIA_REGION`

6.14.1 BC FLUX = CONSTANT

Parameters: `FLUX =` *`REAL`*

Example:

```
BC Flux for Energy on surface_10 = Constant Flux=3.14159
```

Description: `FLUX` is the value of the constant flux where positive values indicate a loss, i.e., positive flux leavse the volume.

6.14.2 BC FLUX = ENCLOSURE_RADIATION

Parameters: `[MULTIPLIER =` *`REAL`*`]`

Example:

```
BC Flux for Energy on surface_10 = Enclosure_Radiation
```

Description: This boundary condition incorporates the heat flux that's computed using Chaparral for enclosure radiation.

See chapter 21 for more information on enclosure radiation.

6.14.3 BC FLUX = LASER

Parameters:

`TON =` *`REAL`*
`TOFF =` *`REAL`*
`X0 =` *`REAL`*
`Y0 =` *`REAL`*
`R0 =` *`REAL`* `W =` *`REAL`*
`FLUX =` *`REAL`*
`RLASER =` *`REAL`*

Example

```
BC Flux for Energy on surface_10 = Laser ton=0 toff=10 x0=0 y0=0 w=1.0
rlaser=0.025 flux=3.14159 r0=0.01
```

Description

This boundary condition imposes an energy flux due to an incident laser. The laser is directed in the z direction and travels in a circular path.

`TON` is the time when the laser is turned on.

`TOFF` is the time when the laser is turned off.

`X0` is the x coordinate of the center of the laser path.

`Y0` is the y coordinate of the center of the laser path.

`W` is the angular velocity of the laser.

`FLUX` is the value of the energy flux *into* the surface. (The usual convention is that positive fluxes indicate loss.)

`RLASER` is the radius of the laser beam.

6.14.4 BC FLUX = LASER_WELD

Parameters

`PATH_FUNCTION` = *STRING*
`FLUX` = *REAL*
`RLASER` = *REAL*
[`NORMAL_TOLERANCE` = *REAL*]

Example

```
# This function lives at the top level, inside the 'Begin Sierra'
block


Begin Definition for Function PATH
  Type is Multicolumn Piecewise Linear
  Column Titles Time X Y Z
  Begin Values
    0.0000000E+00   0.0000000E+00   0.0000000E+00   0.0000000E+00
    0.2250000E+00   0.1500000E+00   0.0000000E+00   0.0000000E+00
    0.2328540E+00   0.1552336E+00  -0.1370490E-03   0.0000000E+00
    ...
   End
End


# This goes in the Aria Region with all of the other BCs
BC Flux for Energy on surface_10 = Laser Path_Function=PATH
Rlaser=0.025 Flux=3.14159
```

Description This boundary condition imposes an energy flux due to an incident laser. The position of the laser is dictated by a user supplied function which contains x, y and z coordinates as a function of time; Aria uses linear piecewise interpolation to obtain the location of the laser at a given time.

`PATH` is the name of the user supplied function which has time as the first column and contains columns titled `X`, `Y` and `Z` containing the x, y and z coordinates.

`FLUX` is the value of the energy flux *into* the surface. (The usual convention is that positive fluxes indicate loss.)

`RLASER` is the radius of the laser beam.

The determination of whether a point in space has an incident laser flux is done as follows. At time t the laser is at a point $\boldsymbol{P}$ and the `Laser_Weld` boundary condition is being evaluated at a point $\boldsymbol{x}$ (typically an integration point). The distance of $\boldsymbol{x}$ from $\boldsymbol{P}$ is defined as $\boldsymbol{r} \equiv \boldsymbol{P} - \boldsymbol{x}$. If a tolerance `NORMAL_TOLERANCE` is supplied and the out-of-surface portion of $\boldsymbol{r}$ is greater than the tolerance then the flux is taken to be zero. Otherwise, the out-of-surface portion of $\boldsymbol{r}$ is subtracted to neglect minor differences between the `PATH_FUNCTION` and the discretized mesh, $\boldsymbol{r}_s \equiv \boldsymbol{r} - (\boldsymbol{n} \cdot \boldsymbol{r})\boldsymbol{n}$ where $\boldsymbol{n}$ is the outward unit normal at $\boldsymbol{x}$. If $|\boldsymbol{r}_s| <$ `RLASER` then the point $\boldsymbol{x}$ is considered to be in the beam and the flux is applied; otherwise, no flux is applied.

6.14.5 BC FLUX = SPOT_WELD

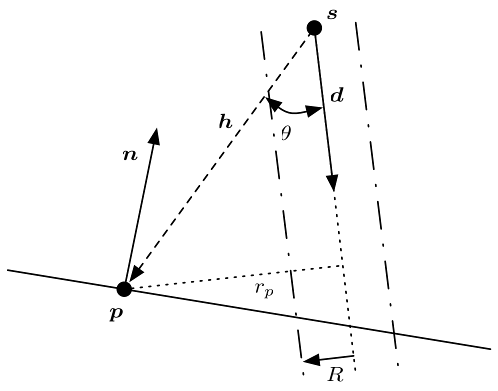
Parameters

```
SRC_X = REAL
SRC_Y = REAL
[SRC_Z = REAL]
DIR_X = REAL
DIR_Y = REAL
[DIR_Z = REAL]
FLUX = REAL
R = REAL
[TON = REAL]
[TOFF = REAL]
```

Example

```
BC Flux for Energy on surface_10 = Spot_Weld Src_X=0 Src_Y=3 Dir_X=0
Dir_Y=-1 Flux=1000 R=0.01
```

Description This boundary condition imposes an energy flux due to an incident laser. The laser source is at the coordinates provided by the `SRC` vector coordinates ($\boldsymbol{s}$ in the figure below) and it is directed from there as specified by the `DIR` vector ($\boldsymbol{d}$ in the figure below).



The radius of the laser is provided by `R` (R in the figure) and the energy flux is provided by the `FLUX` argument (written as f below). A positive `FLUX` is defined as energy input to the surface. That is, the net normal flux is

$$q_n = \frac{(-\boldsymbol{n}) \cdot \boldsymbol{d}}{|\boldsymbol{d}|}(-f) = \frac{\boldsymbol{n} \cdot \boldsymbol{d}}{|\boldsymbol{d}|} f \tag{6.14}$$

Here, the term $-\boldsymbol{n} \cdot \boldsymbol{d}$ accounts for the fact that the surface may not be orthogonal to the laser.

6.14.6 BC FLUX = LATENT_HEAT

Parameters `Y0 = REAL`
`[SPECIES = INT]`

Example

```
BC Flux for Energy on surface_10 = Latent_Heat Y0=0.2 SPECIES=0
BC Flux for Energy on surface_10 = Latent_Heat Y0=0.4 SPECIES=1
BC Flux for Energy on surface_10 = Latent_Heat Y0=0.0 SPECIES=2
```

Description This boundary condition accounts for the heat flux due to the latent heat of vaporization (evaporation).

$$q = H_v \rho (Y_i - Y_{\infty,i}) \tag{6.15}$$

where $H_{v,i}$ is the heat of vaporization of species i, ρ is the density of the material, Y_i is the mass fraction of species i and $Y_{\infty,i}$ is the mass fraction of species i far from the surface.

`SPECIES` is the index of the species to use (in multicomponent systems), $= i$ in equation 6.15.

`Yinf` is the mass fraction of species i far from the surface, $= Y_{\infty,i}$ in equation 6.15.

6.14.7 BC FLUX = NAT_CONV

Parameters `T0 =` *`REAL`*
`H =` *`REAL`*

Example `BC Flux for Energy on surface_10 = Nat_Conv T0=273.15 H=300`

Description This boundary condition accounts for the heat flux due to natural heat convection:

$$q = h(T - T_o) \tag{6.16}$$

`T0` is the temperature of free space.

`H` is the heat transfer coefficient.

6.14.8 BC FLUX = RAD

Parameters `T0 =` *`REAL`*
`CRAD =` *`REAL`*

Example `BC Flux for Energy on surface_10 = Rad T0=273.15 CRAD=1.3e-8`

Description This boundary condition accounts for the heat flux due to radiation in free space:

$$q = c_{rad}(T^4 - T_o^4) \tag{6.17}$$

`T0` is the temperature of free space.

`CRAD` is the coefficient that multiplies the radiation term, viz., the product of the emissivity and the Stefan-Boltzmann constant, which is $5.67 \times 10^{-08}\,\mathrm{W\,m^{-2}\,K^{-4}}$ (SI Units).

6.14.9 BC FLUX = VAPOR_COOLING

Parameters `TBOIL =` *`REAL`*

Example `BC Flux for Energy on surface_10 = Vapor_Cooling Tboil=3000`

Description This boundary condition accounts for the cooling of a material due to vaporization. See Allen Roach (raroach@sandia.gov) for more information.

`TBOIL` is the boiling point of the material.

6.14.10 BC FLUX = NAT_CONV

Parameters `Yinf =` *REAL*
`k =` *REAL*

Example `BC Flux for Species_2 on surface_10 = Nat_Conv k=0.15 Yinf=0.8`

Description This boundary condition accounts for the mass flux due to natural convection:

$$q = k(Y - Y_\infty) \tag{6.18}$$

`Yinf` is the bulk species concentration, Y_∞.

`k` is the mass transfer coefficient.

6.14.11 BC FLUX = CAPILLARY

Parameters `(none)`

Example `BC FluxBP for Momentum on surface_10 = Capillary`

Description This boundary condition implements the capillary (surface tension) contributions to the traction boundary condition Cairncross et al. (2000). Specifically, this boundary condition adds to the k^{th} component of the i^{th} momentum residual

$$\int_S \sigma\left(\boldsymbol{I} - \boldsymbol{n}\boldsymbol{n}\right) : \boldsymbol{\nabla}\left(\phi^i \boldsymbol{e}^k\right)\, \mathrm{dS}. \tag{6.19}$$

where σ is the surface tension and $\boldsymbol{n}$ is the unit outward normal to the interface. This condition accounts for both the curvature and surface tension gradient contributions to the traction condition.

6.14.12 BC FLUX = CONSTANT_TRACTION

Parameters `[X =` *REAL*`]`
`[Y =` *REAL*`]`
`[Z =` *REAL*`]`

Example
```
BC Flux for Momentum on surface_10 = Constant_Traction Y=0.5
or
BC Flux for Mesh on surface_10 = Constant_Traction X=0.1 Y=0.5
or
BC Flux for Solid on surface_10 = Constant_Traction Z=0.5
```

Description This boundary condition integrates a constant and uniform traction over a surface for either fluid momentum, mesh elasticity or solid elasticity. Specifically, this boundary condition adds to the k^{th} component of the i^{th} momentum/mesh/solid residual

$$\int_{\mathrm{S}} \boldsymbol{f}_t \phi^i \,\mathrm{dS}. \tag{6.20}$$

where $\boldsymbol{f}_t$ is the constant traction vector whose components are given by the parameters `X`, `Y` and `Z`.

6.14.13 BC FLUX = ELECTRIC_TRACTION

Parameters `[SIGN = *REAL*]`

Example

```
BC Flux for Momentum on surface_10 = Electric_Traction
BC Flux for Mesh on surface_10 = Electric_Traction
BC Flux for Solid on surface_10 = Electric_Traction
```

Description This boundary condition adds the stress contribution for due to the presence of an electric field. Here, the electric stress tensor is taken to be

$$\boldsymbol{T}_e = \epsilon \boldsymbol{E}\boldsymbol{E} - \frac{1}{2}\epsilon \boldsymbol{E}\cdot\boldsymbol{E}\boldsymbol{I} \tag{6.21}$$

where $\boldsymbol{E} = -\boldsymbol{\nabla} V$ is the electric field and V is voltage and ϵ is the electric permittivity. The electric traction is then defined as

$$\boldsymbol{t} = \epsilon \boldsymbol{n}\cdot\boldsymbol{E}\boldsymbol{E} - \frac{1}{2}\epsilon \boldsymbol{E}\cdot\boldsymbol{E}\boldsymbol{n} \tag{6.22}$$

where $\boldsymbol{n}$ is the outward normal to the boundary.

6.14.14 BC FLUX = FLOW_HYDROSTATIC

Parameters

```
[GX = REAL]
[GY = REAL]
[GZ = REAL]
[P_REF = REAL]
```

Example

```
BC Flux for Momentum on surface_10 = Flow_Hydrostatic Gy=-9.8
```

Description This boundary condition provides a hydrostatic pressure head along a boundary. The acceleration vector g is specified with the parameters `GX`, `GY` and `GZ` (all default to zero). The reference pressure P_{ref} (`P_REF`, defaults to zero) is taken to be the pressure datum at the origin $(0, 0, 0)$. Specifically, this BC adds

$$\boldsymbol{n} \cdot \boldsymbol{T} = -\left(P_{ref} + \sum_{i}^{N_d} \rho g_i x_i \right) \boldsymbol{n} \tag{6.23}$$

to the momentum equation. Here N_d is the spatial dimension of the problem, g_i are the components of the acceleration, x_i are the coordinate components and ρ is the density. **NOTE:** This BC evaluates the density material model for ρ thus the parameters for $\boldsymbol{g}$ should *not* include the density.

6.14.15 BC FLUX = OPEN_FLOW

Parameters `PRESSURE = `*`REAL`*

Example
```
BC Flux for Momentum on surface_10 = Open_Flow Pressure=1000
```

Description This boundary condition adds back the stress contribution for inlet/outlet flows where the pressure is prescribed, and the flow is unidirectional. This command line is used in open-flow applications to set the pressure datum, while attempting to impose the least amount of constraint on the flow profile. A necessary prerequisite is that the flow either be into the domain or out of the domain, not both. The assumption of a fully developed profile is implicit in the expression below where $\boldsymbol{n} \cdot \boldsymbol{\nabla v} = \boldsymbol{0}$ has been used.

$$\boldsymbol{q} = -p_\circ \boldsymbol{n} + \mu \boldsymbol{n} \cdot \boldsymbol{\nabla v}^t \tag{6.24}$$

Here, $p_\circ$ is the pressure provided by the user, μ is the viscosity, $\boldsymbol{\nabla v}^T$ is the transpose of the gradient of the velocity, and $\boldsymbol{n}$ is the outward normal to the boundary.

6.14.16 BC FLUX = PRESSURE

Parameters
```
P = REAL
[C_T = REAL]
```

Example
```
BC Flux for Momentum on surface_10 = Pressure P=101325
or
BC Flux for Mesh on surface_10 = Pressure P=101325
or
BC Flux for Solid on surface_10 = Pressure P=101325
```

Description: This boundary condition integrates a uniform pressure over a surface for either fluid momentum, mesh elasticity or solid elasticity. The optional parameter `C_T` allows the pressure to vary linearly in time. Specifically, this boundary condition adds to the k^{th} component of the i^{th} momentum/mesh/solid residual

$$\int_{\text{S}} -\left(p + c_t t\right) \boldsymbol{n} \phi^i \, \text{dS}. \tag{6.25}$$

where p is the pressure provided by the parameter `P`, t is time and c_t is a constant provided by the `C_T` parameter.

6.14.17 BC FLUX = SLIP

Parameters:

```
BETA = REAL
[VS_X = REAL]
[VS_Y = REAL]
[VS_Z = REAL]
```

Example:

```
BC Flux for Momentum on surface_10 = Slip Beta = 0.01
```

Description: This boundary condition implements the Navier slip boundary condition along a surface wherein the tangential velocity along the surface is proportional to the fluid stress,

$$\boldsymbol{q} = \boldsymbol{n} \cdot \boldsymbol{T} = \frac{1}{\beta}\left(\boldsymbol{v} - \boldsymbol{v}_s\right) \tag{6.26}$$

where β is the Navier slip coefficient, $\boldsymbol{v}$ is the fluid velocity and $\boldsymbol{v}_s$ is the velocity of the surface. The surface velocity is zero by default but a nonzero velocity can be supplied in component form using one or more of the optional `VS_X`, `VS_Y` and `VS_Z` parameters.

6.14.18 BC FLUX = TRANSIENT_TRACTION

Parameters:

```
[A_X = REAL]
[A_Y = REAL]
[A_Z = REAL]
[B_X = REAL]
[B_Y = REAL]
[B_Z = REAL]
```

Example:

```
BC Flux for Momentum on surface_10 = Transient_Traction B_Y=0.5
or
BC Flux for Mesh on surface_10 = Transient_Traction A_X=0.1 A_Y=0.5
or
BC Flux for Solid on surface_10 = Transient_Traction B_Z=0.5
```

Description This boundary condition integrates a uniform but time dependent traction over a surface for either fluid momentum, mesh elasticity or solid elasticity. Specifically, this boundary condition adds to the k^{th} component of the i^{th} momentum/mesh/solid residual

$$\int_S (\boldsymbol{f}_a + t\boldsymbol{f}_b)\, \phi^i \,\mathrm{dS}. \tag{6.27}$$

where t is time and $\boldsymbol{f}_a$ and $\boldsymbol{f}_b$ are is constant traction vectors whose components are given by the parameters `A_X`, `A_Y` and `A_Z` and `B_X`, `B_Y` and `B_Z` respectively.

6.14.19 BC FLUX = WETTING_SPEED_BLAKE_LS

Parameters
```
V_W = REAL
G = REAL
THETA = REAL
WIDTH = REAL
```

Example
```
BC Disting for Momentum_A on surface_3 = Wetting_Speed_Blake_LS
V_w=1e-1 g=1 Width=1 Theta=60
BC Disting for Momentum_B on surface_3 = Wetting_Speed_Blake_LS
V_w=1e-1 g=1 Width=1 Theta=60
```

Description This boundary condition is a distinguishing condition that enforces a slip velocity in accordance with the model provided by Blake and De Coninck (2002).

$$\boldsymbol{v} - f(\phi) v_w \sinh\left(g\left(\cos\theta_s - \cos\theta_a\right)\right) \frac{\boldsymbol{n}_{ls} - \cos\theta_a \boldsymbol{n}_w}{1 - \cos^2\theta_a} = \boldsymbol{0} \tag{6.28}$$

Here θ_s is the static or equilibrium contact angle and is provided by the `THETA` input parameter, θ_a is the actual (or "observered" or "current") contact angle, g is a constant prameter given by the `G` input parameter and v_w is the characteristic slip velocity and is given by the input parameter `V_W`. In their paper, Blake and De Coninck develop g and v_w theoretically.

The function $f(\phi)$ where ϕ is the level set distance function is simply a triangle shape function which causes the velocity to vary from v_w to zero over the distance given by the input parameter `WIDTH`. Taking h_w to half of the input `WIDTH` parameter, $f(\phi)$ is given as

$$f(\phi) = \begin{cases} 0 & : & & & \phi & < & -h_w \\ 1 + \phi/h_w & : & -h_w & \leq & \phi & < & 0 \\ 1 - \phi/h_w & : & 0 & \leq & \phi & < & h_w \\ 0 & : & h_w & \leq & \phi & & \end{cases}$$

This boundary condition is a distinguishing condition which means the momentum equations are discarded at the nodes where this is applied. Also, the velocity provided by this boundary condition is purely tangential. Thus, this boundary condition automatically enforces no-pentration in addition to slip/no-slip.

Note: The interface normal points out of the *negative* phase (negativly signed level set distance ϕ) into the *positive* phase. Thus, the contact angle is measured from the wall, through the *negative* phase and to the level set interface $\phi = 0$.

Chapter 7

Distinguishing Conditions

7.1 An Introduction to Distinguishing Conditions

Aria's *distinguishing condition* (DC) feature is an essential ingredient in solving many coupled physics problems. A distinguishing condition is really just another equation specification except that it typically replaces a regular equation on a subset of the domain such as a surface.

For example, in solving fluids problems with a free surface where the mesh boundary moves with the material, e.g., an ALE simulation, a kinematic condition is used to tie the mesh to the fluid on the free boundary. In this example, one of the mesh coordinates, say `MESH_DISPLACEMENTS_X`, is uknown. So, the equation that is normally used to solve for that component (the $x-$component of the `MESH` equation) is replaced with the kinematic condition: $\boldsymbol{n} \cdot \left(\boldsymbol{v} - \dot{\boldsymbol{d}}\right) - v_\circ = 0$.

An additional feature of distinguishing conditions is that multiple DCs for a given degree of freedom on a given surface are added together. This additive feature allows users to build up their own conditions from primitive ones.

An important thing to know about these conditions is that they are satisfied *weakly.* That is, the DC is multiplied by a finite element weight function and integrated over the surface. Consequently, the condition is only satisfied weakly and to within the tolerance of the nonlinear solver.

The remainder of this chapter contains a description of the primitive DCs that are available in Aria. Using the user plugin feature described in chapter 14 users can add their own, more complicated or specialized conditions.

7.2 BC DISTING

Syntax

```
BC DISTING for EQNAME[_<Subindex>][_<Phase>]
ON MESH_PART = MODEL[param₁ = val₁, param₂ = val₂ ...]
```

Description

Replaces the equations for ***EQNAME*** on ***MESH_PART*** with a distinguishing condition implemented by ***MODEL*** .

Details Prior to the assembly of the distinguishing conditions, the "normal" matrix and RHS entries are zeroed. So, these conditions do not rely on penalty parameters. Also, if multiple distinguishing conditions are supplied the are added together so the *sum* of the conditions is satisfied – that is, they are not individually satisfied. This allows you to construct complex expressions based on the available primitives and/or any plugins. See the `POLYNOMIAL` model below for an example combining two distinguishing conditions. Finally, it is worth noting that these conditions are satisfied *weakly* and so they are only satisfied to within the tolerance of the nonlnear solver.

Parent Block(s) `ARIA REGION`

7.2.1 KINEMATIC

Description This model implements the kinematic condition of the form

$$\boldsymbol{n} \cdot (\boldsymbol{v} - \dot{\boldsymbol{x}}) - v_\circ = 0 \tag{7.1}$$

where $\boldsymbol{n}$ is the outward unit normal to the boundary, $\boldsymbol{v}$ is the velocity, $\dot{\boldsymbol{x}}$ is the time derivative of the mesh boundary position and $v_\circ$ is the mass flux per unit mass across the interface, viz. a "leak" velocity. The leak velocity can be supplied by the `V0` parameter which defaults to zero.

Parameters `V0 = *REAL*`

Example
```
BC Disting For Mesh_X On surface_2 = Kinematic v0=0
```

7.2.2 PLUGIN

Parameters
```
NAME = STRING
[...]
```

Example
```
BC Disting For Temperature On surface_5 = Plugin Name=MyDC alpha=2.3
```

Description `NAME` is the name with which the plugin is registered. See section 14 for more information.

7.2.3 POLYNOMIAL

Parameters
```
VARIABLE = STRING
ORDER = INT
[C0 = REAL]
[C1 = REAL]
...
[CN = REAL]
```

Example
```
BC Disting For Mesh_Y On surface_2 = Polynomial Variable=Time Order=1
C1=0.5

BC Disting For Mesh_Y On surface_2 = Polynomial
Variable=Mesh_Displacement_Y Order=1 C1=-1
```

Description
Arbitrary order polynomial function of a specified scalar variable.

$$\sum_{i=0}^{N} C_i X^i = 0 \tag{7.2}$$

Here, N is the order of the polynomial provided by the `ORDER` parameter and X is the variable supplied by the `VARIABLE` parameter and C_i are the supplied coefficients. Coefficients that are not supplied default to a value of zero. The `VARIABLE` argument can be `TIME` or any internal Expression that evaluates to a scalar. For the latter case, the format of the `VARIABLE` argument is described in section 2.6.1.

In the example given above, two distinguishing conditions are combined to give a composite function. In that example, the resulting conditions is

$$\frac{1}{2}t - \boldsymbol{d}_y = 0 \tag{7.3}$$

or

$$\boldsymbol{d}_y = \frac{1}{2}t \tag{7.4}$$

7.2.4 WETTING_SPEED_BLAKE_LS

Parameters
```
V_W = REAL
G = REAL
THETA = REAL
WIDTH = REAL
[TAU = REAL]
```

Example
```
BC Disting for Momentum_A on surface_3 = Wetting_Speed_Blake_LS
V_w=1e-1 g=1 Width=1 Theta=60
```

Description

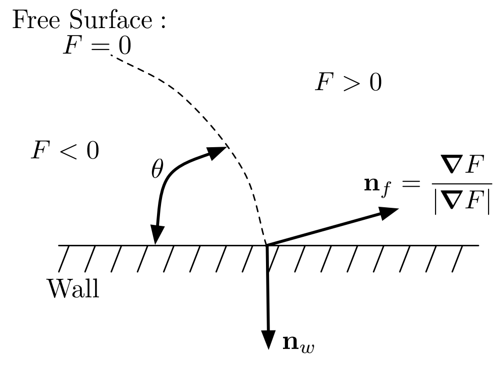
$$\boldsymbol{v} - \delta(F) v_w \sinh\left(g\left(\cos\theta_s - \cos\theta\right)\right) \boldsymbol{t}_w + \delta(F)\tau\dot{\boldsymbol{v}} = 0 \tag{7.5}$$

where $\boldsymbol{v}$ is the fluid velocity and $\boldsymbol{t}_w$ is the tangent to the wall. The function $\delta(F)$, where F is the level set distance field, is given by

$$\delta(F) = \frac{1}{2}\left(1 + \cos\frac{\pi F}{\alpha}\right) \tag{7.6}$$

when $|F| < \alpha$ and zero elsewhwere. Here, α is the half of the `WIDTH` parameter. The term involving τ is a transient relaxation term. By default, $\tau = 0$.

This distinguishing condition is a function of the static and observed contact angles. The *static* contact angle θ_s, supplied by the `THETA` parameter, is fixed. The *observed* contact angle θ is computed from the current state of the solution as illustracted in the following diagram. The important point here is that the contact angle is measured through the *negative* side of the distance function which is denoted `PHASE_A` in Aria.


Free Surface :
$F = 0$
$F > 0$
$F < 0$
θ
$\mathbf{n}_f = \frac{\boldsymbol{\nabla} F}{|\boldsymbol{\nabla} F|}$
Wall
$\mathbf{n}_w$

Chapter 8

Source Terms

8.1 POINT SOURCE FOR ...

Syntax `POINT SOURCE FOR` *`EQNAME`* `ON` *`MESH_PART`* `VALUE = Q X = x [Y = y [Z = z]]`

Description Arbitrary point source contributions.

Details This adds a point source with value `Q` at the specified position. Currently limited to constant and scalar sources. Only supported by the voltage equation (though extending it to other equations is simple, just ask). This line command syntax needs to change since the `ON` *`MESH_PART`* piece doens't really make sense since you already supply the coordinates. You have to supply as many coordinate positions as the problem has dimensions.

Mathematically, the point source is represented as

$$q = Q\delta(\boldsymbol{x} - \boldsymbol{X}) \tag{8.1}$$

where $\delta()$ is the Dirac delta function which is zero everywhwere except at the point $\boldsymbol{x} = \boldsymbol{X}$ where $\boldsymbol{x}$ is the physical coordinate and $\boldsymbol{X}$ is the position provided via the input. In finite elements, this source is integrated

$$\int\limits_{\mathrm{V}} Q\delta(\boldsymbol{x} - \boldsymbol{X})\phi^i \,\mathrm{dV} = Q\phi^i(\boldsymbol{X}) \in \mathrm{Elem}_{\boldsymbol{X}}. \tag{8.2}$$

Here we employ the local support of the basis functions ϕ so that this is only evaluated in the elements containing the point $\boldsymbol{X}$, $\mathrm{Elem}_{\boldsymbol{X}}$.

Parent Block(s) `ARIA_REGION`

Example `POINT SOURCE FOR Voltage ON block_1 Value = 1 X = 0 Y = 0 Z = 0`

8.2 SOURCE FOR ENERGY

Syntax `SOURCE FOR ENERGY ON` *`MESH_PART`* `=` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Arbitrary source contributions for the energy equation.

Details Adds the source provided by *MODEL* to the energy equation.

Parent Block(s) `ARIA_REGION`

8.2.1 SOURCE FOR ENERGY = VISCOUS_DISSIPATION

Parameters `[MULTIPLIER = REAL]`

Example `Source For Energy on block_1 = Viscous_Dissipation`

Description This source accounts for the heat generation due to viscous dissipation:

$$q = m\,\boldsymbol{\tau} : \nabla \boldsymbol{v} \tag{8.3}$$

where $\boldsymbol{\tau}$ is the viscous stress tensor, $\nabla \boldsymbol{v}$ is the gradient of the velocity and m is a multiplier (`MULTIPLIER`) that defaults to 1. The exact form of $\boldsymbol{\tau}$ will depend on the `MOMENTUM_STRESS` model choice(s) for the material.

8.2.2 SOURCE FOR ENERGY = COMPRESSIVE_WORK

Parameters `[MULTIPLIER = REAL]`

Example `Source For Energy on block_1 = Compressive_Work`

Description This source accounts for the heat generation or consumption due to compression:

$$q = -m\,p : \nabla \cdot \boldsymbol{v} \tag{8.4}$$

where p is the pressure, $\nabla \cdot \boldsymbol{v}$ is the divergence of the velocity field and m is a multiplier (`MULTIPLIER`) that defaults to 1.

8.2.3 SOURCE FOR ENERGY = CURING_FOAM_HEAT_OF_RXN

Parameters
```
VFRAC_SUBINDEX = INT
EXTENT_SUBINDEX = INT
H_RXN = REAL
```

Example
```
Source For Energy on block_1 = Curing_Foam_Heat_of_Rxn Vfrac_Subindex=1
Extent_Subindex=2 H_rxn=250
```

Description This source accounts for the heat of reaction of a curing epoxy foam, specifically:

$$q = \rho \left(1 - \phi\right) \Delta H_{rxn} \frac{\partial \xi}{\partial t} \tag{8.5}$$

where ρ is the density of the fluid, ϕ is the volume fraction, ΔH_{rxn} is the heat of reaction and ξ is the extent of reaction.

NOTE: The volume fraction is assumed to be a `SPECIES` field with the subindex provided by the `VFRAC_SUBINDEX` parameter. Likewise, the extent of reaction field is assumed to be a `SPECIES` field with the subindex provided by the `EXTENT_SUBINDEX` parameter.

8.2.4 SOURCE FOR ENERGY = CURING_FOAM_LATENT_HEAT

Parameters
```
VFRAC_SUBINDEX = INT
H_EVAP = REAL
```

Example
```
Source For Energy on block_1 = Curing_Foam_Latent_Heat Vfrac_Subindex=1
H_evap=15
```

Description This source accounts for the loss of energy due to evaporation of a curing epoxy foam, specifically:

$$q = \rho H_{evap} \frac{\partial \phi}{\partial t} \tag{8.6}$$

where ρ is the density of the fluid, H_{evap} is the latent heat of evaporation and ϕ is the volume fraction,

NOTE: The volume fraction is assumed to be a `SPECIES` field with the subindex provided by the `VFRAC_SUBINDEX` parameter.

8.2.5 SOURCE FOR ENERGY = CURING_FOAM_SPECIFIC_HEAT

Parameters
```
VFRAC_SUBINDEX = INT
[CP_FG = REAL]
[CP_E = REAL]
[PHI_ZERO = REAL]
```

Example
```
Source For Energy on block_1 = Curing_Foam_Specific_Heat
Vfrac_Subindex=1 Cp_fG=1 Cp_e=1 phi_zero=0.2
```

Description This source accounts for the loss of energy due to the variable specific heat for the special case where the specific heat material model is `CURING_FOAM`. See 10.32.2. Specifically, this source term is

$$q = -\rho T \left(\frac{\partial C_p}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{\nabla} C_p \right) \tag{8.7}$$

$$= -\rho T b \left(\frac{\partial \phi}{\partial t} + \boldsymbol{v} \cdot \boldsymbol{\nabla} \phi \right) \tag{8.8}$$

where ρ is the density of the fluid, T is the temperature, $\boldsymbol{v}$ is the velocity, ϕ is the volume fraction and b is as defined in `CURING_FOAM` specific heat material model (see 10.32.2).

NOTE: The volume fraction is assumed to be a `SPECIES` field with the subindex provided by the `VFRAC_SUBINDEX` parameter.

8.2.6 SOURCE FOR ENERGY = JOULE_HEATING

Parameters `[VOLTAGE_SUBINDEX = INT]`

Example `SOURCE FOR Energy ON block_1 = Joule_Heating`

Description This source term adds the volumetric heat source due to Joule heating, a.k.a., Ohmic heating or resitance heating. The volumetric heating is given by (see, Section 3.3)

$$H_V = I^2 R \tag{8.9}$$

$$= (-\sigma_e \boldsymbol{\nabla} V) \cdot (-\sigma_e \boldsymbol{\nabla} V) R \tag{8.10}$$

$$= (-\sigma_e \boldsymbol{\nabla} V) \cdot (-\sigma_e \boldsymbol{\nabla} V) \frac{1}{\sigma_e} \tag{8.11}$$

$$= \sigma_e (\boldsymbol{\nabla} V \cdot \boldsymbol{\nabla} V) \tag{8.12}$$

where I is the current density, R is the resitivity, $\sigma_e = 1/R$ is the electrical conductivity and V is the voltage.

8.2.7 SOURCE FOR ENERGY = POLYNOMIAL

Parameters
```
VARIABLE = STRING
ORDER = INT
[C0 = REAL]
[C1 = REAL]
...
[CN = REAL]
```

Example
```
SOURCE For Energy on block_1 = Polynomial Variable=Temperature Order=1
C0=401.0 C1=88.5
```

Description Arbitrary order polynomial function of a specified scalar variable.

$$H_V = \sum_{i=0}^{N} C_i X^i \tag{8.13}$$

Here, N is the order of the polynomial provided by the `ORDER` parameter and X is the variable supplied by the `VARIABLE` parameter and C_i are the supplied coefficients. Coefficients that are not supplied default to a value of zero. The `VARIABLE` argument can be `TIME` or any internal Expression that evaluates to a scalar. For the latter case, the format of the `VARIABLE` argument is described in section 2.6.1.

8.2.8 SOURCE FOR ENERGY = TBC_JOULE_HEATING

Parameters
```
Ua = REAL
Ub = REAL
V_NS = INT
Ti = REAL
CURRENT_LOAD = REAL
[VOLTAGE_SUBINDEX = INT]
```

Example
```
SOURCE FOR Energy ON block_1 = TBC_Joule_Heating Ua=1.4251 Ub=0.0004785
V_NS=1 Ti=298 CURRENT_LOAD=0.0017
```

Description This source term adds the volumetric heat source due to Joule heating in a thermal battery cell volumetric heating is given by (see, Section 3.3)

$$H_V = \left(U_\circ - V - T_i \frac{\partial U_\circ}{\partial T} \right) I_\circ \tag{8.14}$$

Here, $U_\circ$ is the open circuit potential which is given as a linear function in temperature T and U_a and U_b are the coefficents of that function. V is the cell potential which is taken as the voltage at the node given in the single-node nodeset number `V_NS` and $I_\circ$ is cell load.

For more details, see Chen et al. (2000).

8.3 SOURCE FOR MOMENTUM

Syntax `SOURCE FOR MOMENTUM ON` *`MESH_PART`* `=` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Arbitrary source contributions for the momentum equation.

Details Adds the source provided by *`MODEL`* to the momentum equation.

Parent Block(s) `ARIA_REGION`

8.3.1 SOURCE FOR MOMENTUM = CONSTANT_VECTOR

Parameters	`[X = `*`REAL`*`]`
	`[Y = `*`REAL`*`]`
	`[Z = `*`REAL`*`]`

Example	`SOURCE FOR momentum ON block_1 = CONSTANT_VECTOR Z=-9.8`
Description	This source term only applies to the momentum equation though it is automatically applied to the continuity equation in PSPG formulations. `X`, `Y`, `Z` are the components of the vector source. This vector source is *not* multiplied by the density.

8.3.2 SOURCE FOR MOMENTUM = HYDROSTATIC

Parameters	`GX = `*`REAL`*
	`GY = `*`REAL`*
	`GZ = `*`REAL`*
	`[REF_DENSITY = `*`REAL`*`]`

Example	`SOURCE FOR momentum ON block_1 = HYDROSTATIC gx = 0 gy = 0 gz = -980`
Description	This source term only applies to the momentum equation though it is automatically applied to the continuity equation in PSPG formulations. `GX`, `GY`, `GZ` are the components of the gravity vector $\boldsymbol{g}$ and `REF_DENSITY` is an constant, uniform reference density $\rho_\circ$. With density ρ, the hydrostatic source is defined as $(\rho - \rho_\circ)\boldsymbol{g}$. The default value of the reference density is 0.

8.3.3 SOURCE FOR MOMENTUM = ROTATING_BODY_FORCE

Parameters	`G = `*`REAL`*
	`FREQUENCY = `*`REAL`*
	`[PHASE_SHIFT = `*`REAL`*`]`
	`[REF_DENSITY = `*`REAL`*`]`

Example	`Source for Momentum on block_1 = Rotating_Body_Force g=9.8`
	`frequency=2.5 phase_shift=90`

Description	This source term only applies to the momentum equation though it is automatically applied to the continuity equation in PSPG formulations. `GX`, `GY`, `GZ` are the components of the gravity vector $\boldsymbol{g}$ and `REF_DENSITY` is an constant, uniform reference density $\rho_\circ$. With density ρ, the hydrostatic source is defined as $(\rho - \rho_\circ)\boldsymbol{g}$. The default value of the reference density is 0.

8.3.4 SOURCE FOR MOMENTUM = BOUSSINESQ

Parameters
```
TEMP_REF = REAL
VOL_EXP = REAL
GX = REAL
GY = REAL
GZ = REAL
```

Example
```
SOURCE FOR momentum ON block_1 = BOUSSINESQ vol_exp=0.1 temp_ref=298.15
gx = 0 gy = 0 gz = -980
```

Description This source term only applies to the momentum equation though it is automatically applied to the continuity equation in PSPG formulations.

GX, GY, GZ are the components of the gravity vector $\boldsymbol{g}$. VOL_EXP is the volume expansion coefficient α and TEMP_REF is the reference temperature T_{ref}. With density ρ and temperature T the Boussinesq source is defined as $\rho \boldsymbol{g} \alpha (T - T_{ref})$.

8.4 SOURCE FOR CURRENT

Syntax SOURCE FOR CURRENT ON *MESH_PART* = *MODEL* [param$_1$ = val$_1$, param$_2$ = val$_2$ ...]

Description Arbitrary source contributions for the current equation.

Details Adds the source provided by ***MODEL*** to the current equation.

Parent Block(s) ARIA_REGION

8.4.1 SOURCE FOR CURRENT = BUTLER_VOLMER_SIMPLE

Parameters
```
A = REAL
C_A = REAL
C_C = REAL
U = REAL
Sign = INT
[V1_SUBINDEX = INT]
[V2_SUBINDEX = INT]
```

Example
```
SOURCE FOR CURRENT_1 ON block_1 = Butler_Volmer_Simple A=1.0 C_a=1.0
C_c=-1.0 U=0.2 Sign=-1
SOURCE FOR CURRENT_2 ON block_1 = Butler_Volmer_Simple A=1.0 C_a=1.0
C_c=-1.0 U=0.0 Sign=+1
```

Description This model implements a very simple form of the Butler-Volmer reaction kinetics. It is intended for developmental and demonstrational purposes only.

This source term has the following form (see, also, equation 3.33)

$$R_{V,k} = A\left(e^{c_a(V_1 - V_2 - U)} - e^{-c_c(V_1 - V_2 - U)}\right) \tag{8.15}$$

where V_1 is is the first electric potential field and V_2 is the second electric potential field. By default V_1 is `VOLTAGE_1` and V_2 is `VOLTAGE_2` but the subindexes may be changed using the optional `V1_SUBINDEX` and `V2_SUBINDEX` options.

8.4.2 SOURCE FOR CURRENT = POLYNOMIAL

Parameters
```
VARIABLE = STRING
ORDER = INT
[C0 = REAL]
[C1 = REAL]
...
[CN = REAL]
```

Example
```
SOURCE For Current on block_1 = Polynomial Variable=Temperature
Order=1 C0=401.0 C1=88.5
```

Description Arbitrary order polynomial function of a specified scalar variable.

$$H_V = \sum_{i=0}^{N} C_i X^i \tag{8.16}$$

Here, N is the order of the polynomial provided by the `ORDER` parameter and X is the variable supplied by the `VARIABLE` parameter and C_i are the supplied coefficients. Coefficients that are not supplied default to a value of zero. The `VARIABLE` argument can be `TIME` or any internal Expression that evaluates to a scalar. For the latter case, the format of the `VARIABLE` argument is described in section 2.6.1.

8.5 SOURCE FOR SPECIES

Syntax `SOURCE FOR SPECIES ON` *`MESH_PART`* `=` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Arbitrary source contributions for the species equation.

Details Adds the source provided by *`MODEL`* to the current equation.

Parent Block(s) `ARIA_REGION`

8.5.1 SOURCE FOR SPECIES = CURING_FOAM_EXTENT

Parameters	`K =` *REAL* `E =` *REAL* `R =` *REAL* `N =` *REAL*
Example	`Source For Species_2 on block_1 = Curing_Foam_Extent k=1.145e5 E=10 R=8.314472E3 n=1.3`
Description	This source accounts for the reaction of a curing epoxy foam, specifically:

$$q = ke^{E/RT}\left(1-\xi\right)^{n} \tag{8.17}$$

where T is the temperature and ξ is the extent of reaction.

8.5.2 SOURCE FOR SPECIES = CURING_FOAM_VFRAC

Parameters	`A =` *REAL* `B =` *REAL* `C =` *REAL* `T_BOILING =` *REAL*
Example	`Source For Species_2 on block_1 = Curing_Foam_Vfrac a=1 b=0 c=2e-3 T_BOILING=473.15`
Description	This source accounts for the change in volume fraction with temperature.

$$q = \left(a + bT + cT^2\right)_{\exp} \frac{\partial T}{\partial t} \tag{8.18}$$

where T is the temperature. This source term is only active when $T >= T_{boiling}$.

8.5.3 SOURCE FOR SPECIES = POLYNOMIAL

Parameters	`VARIABLE =` *STRING*
	`ORDER =` *INT*
	`[C0 =` *REAL*`]`
	`[C1 =` *REAL*`]`
	`...`
	`[CN =` *REAL*`]`

Example	`Source For Species on block_1 = Polynomial Variable=Temperature Order=1 C0=401.0 C1=88.5`

Description Arbitrary order polynomial function of a specified scalar variable.

$$H_V = \sum_{i=0}^{N} C_i X^i \tag{8.19}$$

Here, N is the order of the polynomial provided by the `ORDER` parameter and X is the variable supplied by the `VARIABLE` parameter and C_i are the supplied coefficients. Coefficients that are not supplied default to a value of zero. The `VARIABLE` argument can be `TIME` or any internal Expression that evaluates to a scalar. For the latter case, the format of the `VARIABLE` argument is described in section 2.6.1.

8.6 SOURCE FOR VOLTAGE

Syntax `SOURCE FOR VOLTAGE ON` *`MESH_PART`* `=` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Arbitrary source contributions for the voltage equation.

Details Adds the source provided by *`MODEL`* to the current equation.

Parent Block(s) `ARIA_REGION`

8.6.1 SOURCE FOR VOLTAGE = POLYNOMIAL

Parameters
```
VARIABLE = STRING
ORDER = INT
[C0 = REAL]
[C1 = REAL]
...
[CN = REAL]
```

Example
```
Source For Voltage on block_1 = Polynomial Variable=Temperature
Order=1 C0=401.0 C1=88.5
```

Description Arbitrary order polynomial function of a specified scalar variable.

$$H_V = \sum_{i=0}^{N} C_i X^i \tag{8.20}$$

Here, N is the order of the polynomial provided by the `ORDER` parameter and X is the variable supplied by the `VARIABLE` parameter and C_i are the supplied coefficients. Coefficients that are not supplied default to a value of zero. The `VARIABLE` argument can be `TIME` or any internal Expression that evaluates to a scalar. For the latter case, the format of the `VARIABLE` argument is described in section 2.6.1.

Chapter 9

Constraint Conditions

9.1 CONSTRAIN

Syntax `CONSTRAIN AVERAGE_VOLUME_FRACTION phi =` C_0

Description Integral constraint for the average particle volume fraction in a SUSPENSION problem.

Details Constrains ϕ throughout the problem domain to achieve the specified value of average volume particle fraction in accordance with the relation

$$C_0 = \left[\int_{\mathrm{V}_\phi} \mathrm{dV} \right]^{-1} \int_{\mathrm{V}_\phi} \phi \mathrm{dV} \tag{9.1}$$

over element blocks where the SUSPENSION equation is defined.

Parent Block(s) `ARIA_REGION`

Chapter 10

Material Properties

10.1 BETA

Syntax `BETA =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the coefficient for thermal stress.

Details The solid stress $\boldsymbol{T}$ is given by

$$\boldsymbol{T} = \lambda E_{kk}\boldsymbol{I} + 2\mu\boldsymbol{E} - \beta\left(T - T_{ref}\right)\boldsymbol{I} \tag{10.1}$$

where λ and μ are the Lamé coefficients, $\boldsymbol{E} = \frac{1}{2}\left(\boldsymbol{\nabla d} + \boldsymbol{\nabla d}^T\right)$ is the deformation tensor, β is the coefficient of thermal stress, T is temperature and T_{ref} is the solid stress reference temperature.

These Lamé coefficients are related to the more standard Young's modulus, Poisson's ratio and CTE (α) as follows:

$$2\mu = \frac{E}{(1+\nu)} \tag{10.2}$$

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)} = 2\mu\frac{\nu}{(1-2\nu)} \tag{10.3}$$

$$\beta = \frac{\alpha E}{(1-2\nu)} = \alpha\left(3\lambda + 2\mu\right) \tag{10.4}$$

When a user supplies the Young's modulus, Poisson's ratio and CTE properties Aria internally convertes them into the Lamé coefficients.

Parent Block(s) `ARIA MATERIAL`

10.1.1 BETA = CONSTANT

Parameters `BETA =` *`REAL`*

Example `BETA = CONSTANT BETA = 1.0`

Description BETA is the value of β.

10.1.2 BETA = CONVERTED

Parameters (None)

Example BETA = Converted

Description Aria will use Young's modulus, Poisson ratio and CTE to compute the Lamé β coefficient. Supplying the Lamé coefficients is more computationally efficient but perhaps less convenient, especially if the material properties are varying (e.g., temperature dependent in a non-isothermal problem).

10.1.3 BETA = LINEAR

Parameters A = *REAL*
B = *REAL*

Example BETA = LINEAR A = 1.0 B = -.005

Description β is a inear function of temperature T,

$$\beta = A + BT \tag{10.5}$$

10.2 BULK VISCOSITY

Syntax BULK VISCOSITY = *MODEL* [param$_1$ = val$_1$, param$_2$ = val$_2$ ...]

Description Specifies the material model for the fluid bulk viscosity.

Details Specifies the material model for the fluid bulk viscosity.

Parent Block(s) ARIA MATERIAL

10.2.1 BULK VISCOSITY = CONSTANT

Parameters KAPPA = *REAL*

Example `BULK VISCOSITY = CONSTANT KAPPA = 1.0e-5`

Description `KAPPA` is the value of the constant fluid bulk viscosity.

10.2.2 BULK VISCOSITY = CURING_FOAM

Parameters
```
VFRAC_SUBINDEX = INT
EXTENT_SUBINDEX = INT
PHI_ZERO = REAL
[A = REAL]
[B = REAL]
[C = REAL]
[KSI_C = REAL]
```

Example
```
Bulk Viscosity = Curing_Foam Vfrac_Subindex=1 Extent_Subindex=2
Phi_Zero=0.45
```

Description For a curing expoxy with volume fraction ϕ and extent of reaction ξ the viscosity is given by

$$\kappa = \frac{4}{3}\mu_\circ \frac{\phi_\circ - \phi - 1}{\phi_\circ - \phi} \tag{10.6}$$

where $\mu_\circ$ is given by

$$\mu_\circ = (a - bT)\left(\frac{\xi_c^2 - \xi^2}{\xi_c^2}\right)^c \tag{10.7}$$

where T is the tempature. The remaining parameters a, b, c and ξ_c have default values of $a = 20$, $b = 0.22$, $c = -4/3$ and $\xi_c = 0.45$ though they can be overridden with the optional model parameters.

NOTE: The volume fraction is assumed to be a `SPECIES` field with the subindex provided by the `VFRAC_SUBINDEX` parameter. Likewise, the extent of reaction field is assumed to be a `SPECIES` field with the subindex provided by the `EXTENT_SUBINDEX` parameter.

10.3 CTE

Syntax `CTE =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the coefficient of thermal expansion.

Details The solid stress $\boldsymbol{T}$ is given by

$$\boldsymbol{T} = \lambda E_{kk}\boldsymbol{I} + 2\mu\boldsymbol{E} - \beta\left(T - T_{ref}\right)\boldsymbol{I} \tag{10.8}$$

where λ and μ are the Lamé coefficients, $\boldsymbol{E} = \frac{1}{2}\left(\boldsymbol{\nabla d} + \boldsymbol{\nabla d}^T\right)$ is the deformation tensor, β is the coefficient of thermal stress, T is temperature and T_{ref} is the solid stress reference temperature.

These Lamé coefficients are related to the more standard Young's modulus, Poisson's ratio and CTE (α) as follows:

$$\begin{aligned} 2\mu &= \frac{E}{(1+\nu)} && && (10.9) \\ \lambda &= \frac{\nu E}{(1+\nu)(1-2\nu)} &&= 2\mu\frac{\nu}{(1-2\nu)} && (10.10) \\ \beta &= \frac{\alpha E}{(1-2\nu)} &&= \alpha\left(3\lambda + 2\mu\right) && (10.11) \end{aligned}$$

When a user supplies the Young's modulus, Poisson's ratio and CTE properties Aria internally convertes them into the Lamé coefficients.

Supplying the Lamé coefficients is more computationally efficient but perhaps less convenient, especially if the material properties are varying (e.g., temperature dependent in a non-isothermal problem).

Parent Block(s) `ARIA MATERIAL`

10.3.1 CTE = CONSTANT

Parameters `CTE =` *`REAL`*

Example `CTE = CONSTANT cte = 1.0`

Description `CTE` is the value of the coefficient of thermal expansion.

10.4 CURRENT DENSITY

Syntax `CURRENT DENSITY =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the material (constitutive) model for the current density in the bulk.

Details Specifies the material (constitutive) model for the current density in the bulk.

Parent Block(s) `ARIA MATERIAL`

10.4.1 CURRENT DENSITY = BASIC

This is an alias for `OHMS_LAW`.

Example `Current Density = Basic`

10.4.2 CURRENT DENSITY = OHMS_LAW

Parameters `(none)`

Example `Current Density = Ohms_Law`

Description The current density $\boldsymbol{J}$ is given by Ohm's Law,

$$\boldsymbol{J} = -\sigma_e \boldsymbol{\nabla} V \tag{10.12}$$

where σ_e is the electrical conducitivity and V is the voltage (electric potential).

10.5 DENSITY

Syntax `DENSITY =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the material model for the density.

Details Specifies the material model for the density.

Parent Block(s) `ARIA MATERIAL`

10.5.1 DENSITY = CONSTANT

Parameters `RHO =` *`REAL`*

Example `DENSITY = CONSTANT RHO = 1.0`

Description `RHO` is the value of the constant density.

10.5.2 DENSITY = CURING_FOAM

Parameters	`R = REAL`
	`RHO_E = REAL`
	`RHO_F = REAL`
	`PHI_ZERO = REAL`
	`VFRAC_SUBINDEX = INT`

Example	`Molecular Weight = Constant Subindex=1 M = 17.0`
	`Density = Curing_Foam R=8.314472E3 RHO_E=1 RHO_F=1.5 PHI_ZERO=0.2`
	`VFRAC_SUBINDEX=1`

Description: The density curing epoxy foam with volume fraction ϕ, molecular weight M, temperature T, and pressure p is given by

$$\rho = (\phi_\circ - \phi)\frac{pM}{RT} + (1 - \phi_\circ)\,\rho_e + \phi\rho_f \tag{10.13}$$

where R is the gas constant, $\phi_\circ$ is the reference volume fraction in the flourinert ρ_e is the pure expoxy density and ρ_f is the pure flourinert density.

NOTE: The volume fraction is assumed to be a `SPECIES` field with the subindex provided by the `VFRAC_SUBINDEX` parameter.

10.5.3 DENSITY = EXP_DECAY

Parameters	`RHO_INITIAL = REAL`
	`RHO_FINAL = REAL`
	`K = REAL`

Example	`Density = Exp_Decay K=1.2 RHO_INITIAL=1.0 RHO_FINAL=0.2`

Description: This model supplies a density that is an exponential decay,

$$\rho = \rho_f + (\rho_i - \rho_f)\,e^{-kt} \tag{10.14}$$

where ρ_i is the initial density (`RHO_INITIAL`), ρ_f is the final density (`RHO_FINAL` and k (`K`) is the decay constant.

10.5.4 DENSITY = IDEAL_GAS

Parameters	`R = REAL`
	`[P_REF = REAL]`
	`[T_REF = REAL]`

Example	`Molecular Weight = Constant Subindex=1 M = 17.0`
	`Molecular Weight = Constant Subindex=2 M = 23.0`
	`Density = Ideal_Gas R=8.314472E3 T_ref=273.15 P_ref=101325.0`

Description The density of a multicomponent ideal gas in kg m^{-3} may be written as

$$\rho = \frac{P_{ref} + P}{R\,(T_{ref} + T)} \sum_{i}^{N} M_i y_i \tag{10.15}$$

where N is the number of species, P is the pressure in Pascals, P_{ref} is a reference pressure, R is the gas constant, T is the temperature, T_{ref} is a reference temperature, M_i is the molecular weight of species i in kg kmol^{-1} and y_i is the mole fraction of species i. Note, the units given here and on the density card are si units; any units may be used as long as internal consistency is maintained.

The optional reference values for the temperature and pressure allow you to solve for the temperature and pressure using relative units (e.g., Celcius temperature and gauge pressure) and but still use absolute values as required by this material model.

10.5.5 DENSITY = INCOMPRESSIBLE_IDEAL_GAS

Parameters

```
R = REAL
P_REF = REAL
[T_REF = REAL]
```

Example

```
Molecular Weight = Constant Subindex=1 M = 17.0
Molecular Weight = Constant Subindex=2 M = 23.0
Density = Incompressible_Ideal_Gas R=8.314472E3 T_ref=273.15
P_ref=101325.0
```

Description The density of a multicomponent ideal gas in kg m^{-3} may be written as

$$\rho = \frac{P_{ref}}{R\,(T_{ref} + T)} \sum_{i}^{N} M_i y_i \tag{10.16}$$

where N is the number of species, P_{ref} is a reference pressure in pascal, R is the gas constant in J kmol^{-1} K^{-1}, T is the temperature, T_{ref} is a reference temperature, M_i is the molecular weight of species i in kg kmol^{-1}, and y_i is the mole fraction of species i. Note, the units given here and on the density card are si units; any units may be used as long as internal consistency is maintained.

The optional reference value for the temperature allow you to solve for the temperature using relative units (e.g., Celcius temperature) and but still use absolute values as required by this material model.

10.5.6 DENSITY = POLYNOMIAL

Parameters

```
VARIABLE = STRING
ORDER = INT
[C0 = REAL]
[C1 = REAL]
...
[CN = REAL]
```

Example

```
Density = Polynomial Variable=Temperature Order=1 C0=401.0 C1=88.5
```

Description Arbitrary order polynomial function of a specified scalar variable.

$$\rho = \sum_{i=0}^{N} C_i X^i \tag{10.17}$$

Here, N is the order of the polynomial provided by the `ORDER` parameter and X is the variable supplied by the `VARIABLE` parameter and C_i are the supplied coefficients. Coefficients that are not supplied default to a value of zero. The `VARIABLE` argument can be `TIME` or any internal Expression that evaluates to a scalar. For the latter case, the format of the `VARIABLE` argument is described in section 2.6.1.

10.5.7 DENSITY = THERMAL

Parameters
```
[A = REAL]
[B = REAL]
[C = REAL]
[D = REAL]
```

Example
```
DENSITY = THERMAL A = 1.0 B = -.005
```

Description Cubic polynomial function of temperature for the density.

$$\rho = A + BT + CT^2 + DT^3 \tag{10.18}$$

10.5.8 DENSITY = USER_FUNCTION

Parameters
```
NAME = STRING
X = STRING
```

Example
```
Begin Definition for Function Water_Density
   # Source Appendix 2 from "Transport Processes and
   # Unit Operations" by C. J. Geankoplis
   Type is Piecewise Linear
   Begin Values
      # K     kg * m^-3
      273.15   999.87
      277.15  1000.00
      283.15   999.73
      293.15   998.23
      298.15   997.08
      303.15   995.68
      313.15   992.25
      323.15   988.07
      333.15   983.24
      343.15   977.81
      353.15   971.83
      363.15   965.34
      373.15   958.38
   End
End
...

Begin Aria Material Foo
   ...
   Density = User_Function Name=Water_Density X=Temperature
   ...
End Aria Material Foo
```

Description A look-up function is used to compute the values of the density as a function of some other variable, i.e. $f(x)$. The function type ("piecewise linear" in the example above) must support the `differentiate()` method for Newton's method.

Here `NAME` is the name of the user-defined function (`Water_Density` in the example) and `X` is the Aria name of the abcissa variable (`TEMPERATURE` in the example). Note that `X` is not necessarily the same name as the abcissa variable identified in the user-defined function (`T` in the example).

10.6 ELECTRICAL CONDUCTIVITY

Syntax `ELECTRICAL CONDUCTIVITY =` *MODEL* [param$_1$ = val$_1$, param$_2$ = val$_2$ ...]

Description Specifies the material model for the electrical conductivity.

Details Specifies the material model for the electrical conductivity.

Parent Block(s) `ARIA MATERIAL`

10.6.1 ELECTRICAL CONDUCTIVITY = CONSTANT

Parameters `SIGMA =` *REAL*

Example `ELECTRICAL CONDUCTIVITY = CONSTANT SIGMA = 1.0`

Description `SIGMA` is the value of the constant electrical conductivity.

10.6.2 ELECTRICAL CONDUCTIVITY = EXPONENTIAL

Parameters

```
VARIABLE = STRING
[CONSTANT = REAL]
[MULTIPLIER = REAL]
EXPONENT = REAL
```

Example

```
Electrical Conductivity = Exponential Variable=Temperature
Multiplier=1.0 Exponent=-0.3
```

Description Exponential function of in specified scalar variable. The electrical conductivity is computed as

$$\sigma_e = C + Me^{EX} \tag{10.19}$$

Here, C is the constant term supplied by the `CONSTANT` parameter which defaults to zero, M is the value supplied by the `MULTIPLIER` parameter which defaults to unity, X is the variable supplied by the `VARIABLE` parameter and E is the exponential multiplier provided by the `EXPONENT` parameter.

10.6.3 ELECTRICAL CONDUCTIVITY = FROM_RESISTANCE

Parameters *None.*

Example `ELECTRICAL CONDUCTIVITY = FROM_RESISTANCE`

Description The conductivity is computed as the inverse of the electrical resistance which must be provided separately.

10.6.4 ELECTRICAL CONDUCTIVITY = POLYNOMIAL

Parameters

```
VARIABLE = STRING
ORDER = INT
[C0 = REAL]
[C1 = REAL]
...
[CN = REAL]
```

Example

```
Electrical Conductivity = Polynomial Variable=Temperature Order=1
C0=401.0 C1=88.5
```

Description Arbitrary order polynomial function of a specified scalar variable.

$$\sigma_e = \sum_{i=0}^{N} C_i X^i \tag{10.20}$$

Here, N is the order of the polynomial provided by the `ORDER` parameter and X is the variable supplied by the `VARIABLE` parameter and C_i are the supplied coefficients. Coefficients that are not supplied default to a value of zero. The `VARIABLE` argument can be `TIME` or any internal Expression that evaluates to a scalar. For the latter case, the format of the `VARIABLE` argument is described in section 2.6.1.

10.6.5 ELECTRICAL CONDUCTIVITY = TBC

Parameters
`Ki = `*`REAL`*
`Ti = `*`REAL`*
`E = `*`REAL`*
`R = `*`REAL`*

Example `ELECTRICAL CONDUCTIVITY = TBC Ki=1.0 Ti=273 R=8.314 E=1e-3`

Description Thermal battery electrical conductivity model (see Ken Chen).

$$\kappa(T) = \kappa_i \frac{T_i}{T} e^{-\frac{E}{R}\left(\frac{1}{T}-\frac{1}{T_i}\right)} \tag{10.21}$$

Here, T is temperature, T_i is the initial temperature provided by `Ti`, κ_i is the electrical conductivity at T_i provided by `Ki`, R is the universal gas constant provided by `R` and E is the energy provided by `E`.

10.6.6 ELECTRICAL CONDUCTIVITY = THERMAL

Parameters
`[A = `*`REAL`*`]`
`[B = `*`REAL`*`]`
`[C = `*`REAL`*`]`
`[D = `*`REAL`*`]`

Example `ELECTRICAL CONDUCTIVITY = THERMAL A = 1.0 B = -0.01`

Description Cubic polynomial function of temperature for the conductivity.

$$\sigma_e = A + BT + CT^2 + DT^3 \tag{10.22}$$

10.7 ELECTRIC DISPLACEMENT

Syntax `ELECTRIC DISPLACEMENT = `*`MODEL`*` [param`$_1$` = val`$_1$`, param`$_2$` = val`$_2$` ...]`

Description Specifies the material (constitutive) model for the electric displacement

Details Specifies the material (constitutive) model for the electric displacement

Parent Block(s) `ARIA MATERIAL`

10.7.1 ELECTRIC DISPLACEMENT = BASIC

This is an alias for `LINEAR`.

Example `Electric Displacement = Basic`

10.7.2 ELECTRIC DISPLACEMENT = LINEAR

Parameters `(none)`

Example `Electric Displacement = Linear`

Description The electric displacement $\boldsymbol{D}$ is linearly proportional to the electric field ($\boldsymbol{E} = -\boldsymbol{\nabla} V$)

$$\boldsymbol{D} = -\epsilon \boldsymbol{\nabla} V \tag{10.23}$$

where ϵ is the electrical permittivity and V is the voltage (electric potential).

10.8 ELECTRICAL PERMITTIVITY

Syntax `ELECTRICAL PERMITTIVITY =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the material model for the electrical permittivity.

Details Specifies the material model for the electrical permittivity.

Parent Block(s) `ARIA MATERIAL`

10.8.1 ELECTRICAL PERMITTIVITY = CONSTANT

Parameters `E =` *`REAL`*

Example `ELECTRICAL PERMITTIVITY = CONSTANT E = 1.0`

Description `E` is the value of the constant electrical permittivity.

10.9 ELECTRICAL RESISTANCE

Syntax `ELECTRICAL RESISTANCE =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the material model for the electrical resistance.

Details Specifies the material model for the electrical resistance.

Parent Block(s) `ARIA MATERIAL`

10.9.1 ELECTRICAL RESISTANCE = CONSTANT

Parameters *`None.`*

Example `ELECTRICAL RESISTANCE = CONSTANT R = 1.0`

Description `R` is the value of the constant electrical resistance.

10.9.2 ELECTRICAL RESISTANCE = EXPONENTIAL

Parameters
```
VARIABLE = STRING
[CONSTANT = REAL]
[MULTIPLIER = REAL]
EXPONENT = REAL
```

Example
```
Electrical Resistance = Exponential Variable=Temperature
Multiplier=1.0 Exponent=-0.3
```

Description Exponential function of in specified scalar variable. The electrical resistance is computed as

$$R = C + Me^{EX} \tag{10.24}$$

Here, C is the constant term supplied by the `CONSTANT` parameter which defaults to zero, M is the value supplied by the `MULTIPLIER` parameter which defaults to unity, X is the variable supplied by the `VARIABLE` parameter and E is the exponential multiplier provided by the `EXPONENT` parameter.

10.9.3 ELECTRICAL RESISTANCE = FROM_CONDUCTIVITY

Parameters `R =` *`REAL`*

Example `ELECTRICAL RESISTANCE = FROM_CONDUCTIVITY`

Description The resistance is computed as the inverse of the electrical conductivity which must be provided separately.

10.9.4 ELECTRICAL RESISTANCE = POLYNOMIAL

Parameters
```
VARIABLE = STRING
ORDER = INT
[C0 = REAL]
[C1 = REAL]
...
[CN = REAL]
```

Example
```
Electrical Resistance = Polynomial Variable=Temperature Order=1
C0=401.0 C1=88.5
```

Description Arbitrary order polynomial function of a specified scalar variable.

$$R = \sum_{i=0}^{N} C_i X^i \tag{10.25}$$

Here, N is the order of the polynomial provided by the `ORDER` parameter and X is the variable supplied by the `VARIABLE` parameter and C_i are the supplied coefficients. Coefficients that are not supplied default to a value of zero. The `VARIABLE` argument can be `TIME` or any internal Expression that evaluates to a scalar. For the latter case, the format of the `VARIABLE` argument is described in section 2.6.1.

10.9.5 ELECTRICAL RESISTANCE = USER_FUNCTION

Parameters
```
NAME = STRING
X = STRING
```

Example

```
Begin definition for function RESISTANCE_DATA
   Type is piecewise linear
   Abscissa is T
   Ordinate is Electrical_Resistance
   Begin Values
      # [K]      [Ohm-um]
      273        1.00E-9
      323        7.99E-10
      ...
      873        1.09E-11
   End Values
End definition for function RESISTANCE_DATA

...

Begin Aria Material Foo
   ...
   Electrical Resistance = User_Function Name=RESISTANCE_DATA X=Temperature
   ...
End Aria Material Foo
```

Description A look-up function is used to compute the values of the resistance as a function of some other variable, i.e. $f(x)$. The function type ("piecewise linear" in the example above) must support the `differentiate()` method for Newton's method.

Here `NAME` is the name of the user-defined function (`RESISTANCE_DATA` in the example) and `X` is the Aria name of the abcissa variable (`TEMPERATURE` in the example). Note that `X` is not necessarily the same name as the abcissa variable identified in the user-defined function (`T` in the example).

10.10 EMISSIVITY

Syntax `EMISSIVITY =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the material model for the emissivity.

Details Specifies the material model for the emissivity.

Parent Block(s) `ARIA MATERIAL`

10.10.1 EMISSIVITY = CONSTANT

Parameters `E =` *`REAL`*

Example `Emissivity = Constant E = 0.8`

Description E is the value of the constant emissivity.

10.11 ENTHALPHY

Syntax ENTHALPHY = *MODEL* [param$_1$ = val$_1$, param$_2$ = val$_2$ ...]

Description Specifies a model for the enthalphy of a material.

Details Specifies a model for the enthalphy of a material.

Parent Block(s) ARIA MATERIAL

10.11.1 ENTHALPHY = CONSTANT

Parameters H = *REAL*

Example `Enthalphy = Constant H=1e-4`

Description The value is constant in space and time.

10.12 EQUATION OF STATE

Syntax EQUATION OF STATE = *MODEL* [param$_1$ = val$_1$, param$_2$ = val$_2$ ...]

Description Specifies the equation of state for gas dynamics problems.

Details Specifies the equation of state for gas dynamics problems.

Parent Block(s) ARIA MATERIAL

10.12.1 EQUATION OF STATE = IDEAL_GAS

Parameters R = *REAL*
GAMMA = *REAL*

Example `Equation of State = Ideal_Gas R=8.314 Gamma=1.4`

Description R is the gas constant and GAMMA is the ratio of heat capacities.

This model is used for gas dynamics problems where the density is an unknown. In this case, the pressure is given by the ideal gas law,

$$p = RT\rho \tag{10.26}$$

where T is the temperature and ρ is the density. Activating this model supplies several quantities that are related to this equation of state such as the pressure, temperature, and other gas dynamics related quantities.

10.13 HEAT CONDUCTION

Syntax `HEAT CONDUCTION =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the material (constitutive) model for the heat conduction (diffusive flux) in the bulk.

Details Specifies the material (constitutive) model for the heat conduction (diffusive flux) in the bulk.

Parent Block(s) `ARIA MATERIAL`

10.13.1 HEAT CONDUCTION = BASIC

This is an alias for `FOURIERS_LAW`.

Example `Heat Conduction = Basic`

10.13.2 HEAT CONDUCTION = CONVECTED_ENTHALPY

Parameters `(none)`

Example `Heat Conduction = Convected_Enthalpy`

Description The heat conduction (flux) $\boldsymbol{q}$ is given by,

$$\boldsymbol{q} = -h\rho\boldsymbol{v} \tag{10.27}$$

where h is the enthalpy, ρ is the density and $\boldsymbol{v}$ is the velocity.

10.13.3 HEAT CONDUCTION = FOURIERS_LAW

Parameters `(none)`

Example `Heat Conduction = Fouriers_Law`

Description The heat conduction (flux) $\boldsymbol{q}$ is given by Fourier's Law,

$$\boldsymbol{q} = -\kappa \boldsymbol{\nabla} T \tag{10.28}$$

where κ is the thermal conductivity and T is the temperature.

10.14 HEAT OF VAPORIZATION

Syntax `HEAT OF VAPORIZATION =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the heat of vaporization for a material or for a particular species.

Details Quantifies the amount of energy consumed during evaporation per unit mass.

Parent Block(s) `ARIA MATERIAL`

10.14.1 HEAT OF VAPORIZATION = CONSTANT

Parameters `Hv =` *`REAL`*
`[SUBINDEX =` *`INT`*`]`

Example
```
HEAT OF VAPORIZATION = CONSTANT SUBINDEX=0 Hv = 1.0
HEAT OF VAPORIZATION = CONSTANT SUBINDEX=1 Hv = 2.0
HEAT OF VAPORIZATION = CONSTANT SUBINDEX=3 Hv = 3.14
```

Description `HV` is the value of the constant heat of vaporization and `SUBINDEX` is the optioinal species index (used in multicomponent systems).

10.15 INTRINSIC PERMEABILITY

Syntax `INTRINSIC PERMEABILITY =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the material model for the intrinsic permeability tensor for porous flow in Darcy's Law.

Details Specifies the material model for the intrinsic permeability tensor for porous flow in Darcy's Law. In general, the permeability may be nonisotropic in porour media. In that case, Darcy's law may be written as,

$$\rho \boldsymbol{v}_d = \boldsymbol{f} = -\frac{\rho k_r}{\mu} \boldsymbol{K} \cdot (\boldsymbol{\nabla} P - \rho \boldsymbol{g}) \tag{10.29}$$

where $\boldsymbol{K}$ is the intrinsic permeability tensor, ρ is the density, $\boldsymbol{v}_d$ is the Darcy velocity, $\boldsymbol{f}$ is the mas flux, k_r is the relative permeability, P is pressure and $\boldsymbol{g}$ is gravity.

Parent Block(s) `ARIA MATERIAL`

10.15.1 INTRINSIC PERMEABILITY = CONSTANT

Parameters

```
[XX = REAL]
[XY = REAL]
[XZ = REAL]
[YX = REAL]
[YY = REAL]
[YZ = REAL]
[ZX = REAL]
[ZY = REAL]
[ZZ = REAL]
```

Example `Intrinsic Permeability = Constant XX=1 YY=2 ZZ=1`

Description All components default to zero and all values are constant in space and time.

10.16 INTERNAL ENERGY

Syntax `INTERNAL ENERGY = MODEL [param`$_1$` = val`$_1$`, param`$_2$` = val`$_2$` ...]`

Description Specifies a model for the internal energy of a material.

Details Specifies a model for the internal of a material.

Parent Block(s) `ARIA MATERIAL`

10.16.1 INTERNAL ENERGY = GAS_PHASE

Parameters `(none)`

Example `Internal Energy = Gas_Phase`

Description The internal energy e is computed using thermodynamic relation

$$e = h - P/\rho \tag{10.30}$$

where h is the enthalpy, P is the (partial) pressure and ρ is the density.

10.17 LAMBDA

Syntax `LAMBDA = MODEL [param1 = val1, param2 = val2 ...]`

Description Specifies the material model for the λ Lamé coefficient.

Details The solid stress $\boldsymbol{T}$ is given by

$$\boldsymbol{T} = \lambda E_{kk}\boldsymbol{I} + 2\mu\boldsymbol{E} - \beta\left(T - T_{ref}\right)\boldsymbol{I} \tag{10.31}$$

where λ and μ are the Lamé coefficients, $\boldsymbol{E} = \frac{1}{2}\left(\boldsymbol{\nabla d} + \boldsymbol{\nabla d}^T\right)$ is the deformation tensor, β is the coefficient of thermal stress, T is temperature and T_{ref} is the solid stress reference temperature.

These Lamé coefficients are related to the more standard Young's modulus, Poisson's ratio and CTE (α) as follows:

$$2\mu = \frac{E}{(1+\nu)} \tag{10.32}$$

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)} = 2\mu\frac{\nu}{(1-2\nu)} \tag{10.33}$$

$$\beta = \frac{\alpha E}{(1-2\nu)} = \alpha\left(3\lambda + 2\mu\right) \tag{10.34}$$

When a user supplies the Young's modulus, Poisson's ratio and CTE properties Aria internally convertes them into the Lamé coefficients.

Parent Block(s) `ARIA MATERIAL`

10.17.1 LAMBDA = CONSTANT

Parameters `L = REAL`

Example `LAMBDA = CONSTANT L = 1.0`

Description `L` is the value of the constant λ.

10.17.2 LAMBDA = CONVERTED

Parameters `(None)`

Example `LAMBDA = Converted`

Description Aria will use Young's modulus and Poisson ratio to compute the Lamé λ coefficient. Supplying the Lamé coefficients is more computationally efficient but perhaps less convenient, especially if the material properties are varying (e.g., temperature dependent in a non-isothermal problem).

10.18 LEVEL SET HEAVISIDE

Syntax `LEVEL SET HEAVISIDE = MODEL [param₁ = val₁, param₂ = val₂ ...]`

Description Specifies the functional form of the Heaviside function used with level set algorithms. This also implies the Dirac delta function used for level set algorithms.

Details Specifies the functional form of the Heaviside function used with level set algorithms. This also implies the Dirac delta function used for level set algorithms.

Parent Block(s) `ARIA MATERIAL`

10.18.1 LEVEL SET HEAVISIDE = SMOOTH

Parameters `(none)`

Example `LEVEL SET HEAVISIDE = SMOOTH`

Description The Heaviside function in this case is given as

$$H(f) = \frac{1}{2}\left[1 + f/w + \sin\left(\pi f/w\right)/\pi\right] \tag{10.35}$$

Here f is the level set distance function and w is half of the level set width (see 10.19).

10.19 LEVEL SET WIDTH

Syntax `LEVEL SET WIDTH = MODEL [param₁ = val₁, param₂ = val₂ ...]`

Description Specifies the total width of the level set interface. Half of this width falls on the positive side of the zero level set and half falls on the negative side.

Details Specifies the total width of the level set interface. Half of this width falls on the positive side of the zero level set and half falls on the negative side.

Parent Block(s) `ARIA MATERIAL`

10.19.1 LEVEL SET WIDTH = CONSTANT

Parameters `WIDTH = ` *`REAL`*

Example `LEVEL SET WIDTH = CONSTANT WIDTH=0.1`

Description This is what you'd expect it to be – a uniform constant everywhere for all time.

10.20 MASS FLUX

Syntax `MASS FLUX = ` *`MODEL`* `[param`$_1$` = val`$_1$`, param`$_2$` = val`$_2$` ...]`

Description Specifies a constitutive model for the mass flux for porous flow applications.

Details Specifies a constitutive model for the mass flux for porous flow applications.

Parent Block(s) `ARIA MATERIAL`

10.20.1 MASS FLUX = DARCY

Parameters `[GX = ` *`REAL`*`]`
`[GY = ` *`REAL`*`]`
`[GZ = ` *`REAL`*`]`

Example `Mass Flux = Darcy GY=-9.8`

Description The macroscopic, convective mass flux in phase β, $\rho\boldsymbol{v}$ is obtained from the extended Darcy's Law,

$$\boldsymbol{F} = \rho\boldsymbol{f} = -\frac{\rho k_r}{\mu}\boldsymbol{K} \cdot (\boldsymbol{\nabla} P - \rho\boldsymbol{g}) \tag{10.36}$$

10.21 MESH STRESS

Syntax `MESH STRESS = ` *`MODEL`* `[param`$_1$` = val`$_1$`, param`$_2$` = val`$_2$` ...]`

Description Specifies a contribution to the mesh (pseudo-solid) stress tensor. Multiple stresses are combined additvely and may be specified by using this line command multiple times.

Details Specifies a contribution to the mesh (pseudo-solid) stress tensor. The total stress $\boldsymbol{T}$ is given by

$$\boldsymbol{T} = \sum_j \boldsymbol{T}_j \tag{10.37}$$

Parent Block(s) `ARIA MATERIAL`

10.21.1 MESH STRESS = ISOTHERMAL

Parameters `T = `*`REAL`*
`T_REF = `*`REAL`*

Example `MESH STRESS = Isothermal T=500 T_ref=325`

Description This stress accounts for the mechanical stresses due to thermally induced strains.

$$\boldsymbol{T} = -\beta\,(T - T_{ref})\,\boldsymbol{I} \tag{10.38}$$

where β is the Lamé coefficient of thermal stress (related to the coefficient of thermal expansion, α), T is the temperature and T_{ref} is the temperature of the undeformed reference state of the mesh (pseudo-solid). This is a specialization of the `THERMAL` model that uses uniform, fixed temperature and reference temperature.

10.21.2 MESH STRESS = LINEAR_ELASTIC

Parameters `REFERENCE_FRAME = ‘‘MOVING’’ | ‘‘UNDEFORMED’’`

Example `MESH STRESS = Linear_Elastic Reference_Frame = undeformed`

Description Supplies the linear elasticity stress tensor,

$$\boldsymbol{T} = \lambda \operatorname{trace} \boldsymbol{E}\boldsymbol{I} + 2\mu\boldsymbol{E} \tag{10.39}$$

where λ and μ and the Lamé coefficients and $\boldsymbol{E}$ is the strain tensor. When the choice of refernce frame is "UNDEFORMED" then the strain is computed in the undeformed reference state; this is commonly referred to as small strain theory. When the reference frame is "MOVING" then the strain is computed with respect to the deformed coordinates.

Specifically, the strain tensor is given by

$$\boldsymbol{E} = \frac{1}{2}\left(\boldsymbol{\nabla d} + \boldsymbol{\nabla d}^t\right) \tag{10.40}$$

where $\boldsymbol{d}$ is the mesh displacement field. The choice of reference frame determines whether the $\boldsymbol{\nabla}$ operator is computed in the undeformed or moving reference frames.

10.21.3 MESH STRESS = NEOHOOKEAN_ELASTIC

Parameters `(none)`

Example `MESH STRESS = Neohookean_Elastic`

Description Supplies a nonlinear hyperelastic stress of the form,

$$\boldsymbol{T} = \frac{\mu}{J}(\boldsymbol{b} - \boldsymbol{I}) + \frac{\lambda}{J}\ln J\boldsymbol{I} \tag{10.41}$$

where λ and μ and the Lamé coefficients, $\boldsymbol{b} \equiv \boldsymbol{F} \cdot \boldsymbol{F}^t$ is the left Cauchy-Green tensor, $\boldsymbol{F}$ is the deformation gradient and $J \equiv \det \boldsymbol{F}$. See, e.g., Bonet and Wood (1997) or Belytschko et al. (2004).

10.21.4 MESH STRESS = NONLINEAR_ELASTIC

Parameters `(none)`

Example `MESH STRESS = Nonlinear_Elastic`

Description Supplies a nonlinear elastic stress,

$$\boldsymbol{T} = \lambda \operatorname{trace} \boldsymbol{E}\boldsymbol{I} + 2\mu\boldsymbol{E} \tag{10.42}$$

where λ and μ and the Lamé coefficients and $\boldsymbol{E}$ is the strain tensor. The particular choice of strain tensor chosen depends on where the configuration (reference frame) which is set via the MESH MOTION command line. See section 3.11 for more information. When the MESH MOTION is set to ARBITRARY then the Green strain is used. Otherwise, the Almansi strain is used.

10.21.5 MESH STRESS = RESIDUAL

Parameters
```
[SXX | SX = REAL]
[SYY | SY = REAL]
[SZZ | SZ = REAL]
[SXY = REAL]
[SXZ = REAL]
[SYZ = REAL]
```

Example
```
SOLID STRESS = Residual Sxx=0.02 Syy=0.02
```

Description This stress accounts for the initial residual stress in a solid that is constant and uniform everywhere. The components of the residual stress tensor are supplied by the (up-to) six components `SXX`, `SYY`, `SZZ`, `SXY`, `SXZ`, and `SYZ`.

This is directly analogous to the `ISTRESS` condition in ANSYS. To that end, the diagonal components can be specified as either `SXX` or SX etc.

10.21.6 MESH STRESS = THERMAL

Parameters
```
(none)
```

Example
```
MESH STRESS = Thermal
```

Description This stress accounts for the mechanical stresses due to thermally induced strains.

$$\boldsymbol{T} = -\beta \left(T - T_{ref}\right) \boldsymbol{I} \tag{10.43}$$

where β is the Lamé coefficient of thermal stress (related to the coefficient of thermal expansion, α), T is the temperature and T_{ref} is the temperature of the undeformed reference state of the mesh (pseudo-solid).

10.22 MOLECULAR WEIGHT

Syntax `MOLECULAR WEIGHT = MODEL [param`$_1$` = val`$_1$`, param`$_2$` = val`$_2$` ...]`

Description Specifies the molecular weight for a species.

Details Specifies the molecular weight for a species.

Parent Block(s) `ARIA MATERIAL`

10.22.1 MOLECULAR WEIGHT = CONSTANT

Parameters
```
M = REAL
[SUBINDEX = INT]
```

Example
```
Molecular Weight = Constant Subindex=1 M = 17.0
Molecular Weight = Constant Subindex=2 M = 23.0
Molecular Weight = Constant Subindex=5 M = 34.0
```

Description `M` is the value of the molecular weight.

`SUBINDEX` is the species subindex.

10.23 MOMENTUM STRESS

Syntax `MOMENTUM STRESS = MODEL [param` $_1$ `= val` $_1$`, param` $_2$ `= val` $_2$ `...]`

Description Specifies a contribution to the fluid stress tensor. Multiple stresses are combined additvely and may be specified by using this line command multiple times.

Details Specifies a contribution to the fluid momemtnum stress tensor. The total stress $\boldsymbol{T}$ is given by

$$\boldsymbol{T} = \sum_j \boldsymbol{T}_j \tag{10.44}$$

Parent Block(s) `ARIA MATERIAL`

10.23.1 MOMENTUM STRESS = LS_CAPILLARY

Parameters `(none)`

Example `MOMENTUM STRESS = Newtonian MOMENTUM STRESS = LS_Capillary`

Description This adds the capillary boundary condition contributions in the vicinity of the level set interface.

$$\boldsymbol{T} = \sigma\delta(F)\,(\boldsymbol{I} - \boldsymbol{N}\boldsymbol{N}) \tag{10.45}$$

where σ is the surface tension, $\delta(F)$ is the level set delta function, F is the level set distance function and $\boldsymbol{N}$ is the level set normal field.

10.23.2 MOMENTUM STRESS = INCOMPRESSIBLE_NEWTONIAN

Parameters `(none)`

Example `MOMENTUM STRESS = Incompressible_Newtonian`

Description Supplies the incompressible Newtonian stress tensor,

$$\boldsymbol{T} = -p\boldsymbol{I} + \mu\left(\boldsymbol{\nabla v} + \boldsymbol{\nabla v}^t\right) \tag{10.46}$$

where μ is the fluid viscosity, p is the pressure and $\boldsymbol{v}$ is the fluid velocity. The viscosity μ is provided with the viscosity line command as described in section 10.40.

Note that this model does not include stress contributions that are proporational to the divergence of the velocity. To incorporate those contributions, use the NEWTONIAN_DILATIONAL stress model in addition to this model, or use the FORMAL_NEWTONIAN stress model instead of this model.

10.23.3 MOMENTUM STRESS = FORMAL_NEWTONIAN

Parameters `(none)`

Example `MOMENTUM STRESS = Formal_Newtonian`

Description Supplies the complete Newtonian stress tensor,

$$\boldsymbol{T} = -p\boldsymbol{I} + \mu\left(\boldsymbol{\nabla v} + \boldsymbol{\nabla v}^t\right) + \left(\kappa - \frac{2}{3}\mu\right)\boldsymbol{\nabla}\cdot\boldsymbol{vI} \tag{10.47}$$

where μ is the fluid viscosity, p is the pressure and $\boldsymbol{v}$ is the fluid velocity. The viscosity μ is provided with the viscosity line command as described in section 10.40. The bulk viscosity κ is provided with the bulk viscosity line command as described in section 10.2.

10.23.4 MOMENTUM STRESS = NEWTONIAN_DILATIONAL

Parameters `(none)`

Example `MOMENTUM STRESS = Newtonian_Dilational`

Description Adds the dilational stress contribution for Newtonian fluids,

$$\boldsymbol{T} = \left(\kappa - \frac{2}{3}\mu\right)\boldsymbol{\nabla}\cdot\boldsymbol{vI} \tag{10.48}$$

where κ is the bulk viscosity of the fluid, μ is the fluid (dynamic) viscosity and $\boldsymbol{v}$ is the fluid velocity. The viscosity μ is provided with the viscosity line command as described in section 10.40. The bulk viscosity κ is provided with the bulk viscosity line command as described in section 10.2.

See, also, the NEWTONIAN momentum stress model.

10.23.5 MOMENTUM STRESS = NEWTONIAN_VISCOUS

Parameters `(none)`

Example `MOMENTUM STRESS = Newtonian_Viscous`

Description Supplies only the viscous contribution of the Newtonian stress tensor,

$$\boldsymbol{T} = \mu \left(\boldsymbol{\nabla v} + \boldsymbol{\nabla v}^t \right) \tag{10.49}$$

where μ is the fluid viscosity and $\boldsymbol{v}$ is the fluid velocity. The viscosity μ is provided with the viscosity line command as described in section 10.40.

10.23.6 MOMENTUM STRESS = NEWTONIAN_PRESSURE

Parameters `(none)`

Example `MOMENTUM STRESS = Newtonian_Pressure`

Description Supplies only the pressure contribution of the Newtonian stress tensor,

$$\boldsymbol{T} = -p\boldsymbol{I} \tag{10.50}$$

where p is the pressure.

10.24 POISSONS RATIO

Syntax `POISSONS RATIO = `*`MODEL`*` [param`$_1$` = val`$_1$`, param`$_2$` = val`$_2$` ...]`

Description Specifies the material model for the Poisson's ratio.

Details

The solid stress $\boldsymbol{T}$ is given by

$$\boldsymbol{T} = \lambda E_{kk}\boldsymbol{I} + 2\mu\boldsymbol{E} - \beta\left(T - T_{ref}\right)\boldsymbol{I} \tag{10.51}$$

where λ and μ are the Lamé coefficients, $\boldsymbol{E} = \frac{1}{2}\left(\boldsymbol{\nabla d} + \boldsymbol{\nabla d}^T\right)$ is the deformation tensor, β is the coefficient of thermal stress, T is temperature and T_{ref} is the solid stress reference temperature.

These Lamé coefficients are related to the more standard Young's modulus, Poisson's ratio and CTE (α) as follows:

$$\begin{aligned} 2\mu &= \frac{E}{(1+\nu)} && && (10.52)\\ \lambda &= \frac{\nu E}{(1+\nu)(1-2\nu)} &&= 2\mu\frac{\nu}{(1-2\nu)} && (10.53)\\ \beta &= \frac{\alpha E}{(1-2\nu)} &&= \alpha\left(3\lambda + 2\mu\right) && (10.54) \end{aligned}$$

When a user supplies the Young's modulus, Poisson's ratio and CTE properties Aria internally convertes them into the Lamé coefficients.

Supplying the Lamé coefficients is more computationally efficient but perhaps less convenient, especially if the material properties are varying (e.g., temperature dependent in a non-isothermal problem).

Parent Block(s) `ARIA MATERIAL`

10.24.1 POISSONS RATIO = CONSTANT

Parameters `PR =` *`REAL`*

Example `POISSONS RATIO = CONSTANT PR = 1.0`

Description `PR` is the value of the constant Poisson's ratio.

10.25 POROSITY

Syntax `POROSITY =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the material model for the porosity for porous flow applications.

Details Specifies the material model for the porosity for porous flow applications.

Parent Block(s) `ARIA MATERIAL`

10.25.1 POROSITY = CONSTANT

Parameters `PHI =` *`REAL`*

Example `Porosity = Constant PHI=0.1`

Description The value is constant in space and time.

10.25.2 POROSITY = POLYNOMIAL

Parameters

```
VARIABLE = STRING
ORDER = INT
[C0 = REAL]
[C1 = REAL]
...
[CN = REAL]
```

Example `Porosity = Polynomial Variable=Coordinates_X Order=1 C0=0.1 C1=0.01`

Description Arbitrary order polynomial function of a specified scalar variable.

$$\phi = \sum_{i=0}^{N} C_i X^i \tag{10.55}$$

Here, N is the order of the polynomial provided by the `ORDER` parameter and X is the variable supplied by the `VARIABLE` parameter and C_i are the supplied coefficients. Coefficients that are not supplied default to a value of zero. The `VARIABLE` argument can be `TIME` or any internal Expression that evaluates to a scalar. For the latter case, the format of the `VARIABLE` argument is described in section 2.6.1.

10.26 RELATIVE PERMEABILITY

Syntax `RELATIVE PERMEABILITY =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the material model for the relative permeability (scalar) for porous flow in Darcy's Law.

Details Specifies the material model for the relative permeability (scalar) for porous flow in Darcy's Law. In general, the permeability may be nonisotropic in porour media. In that case, Darcy's law may be written as,

$$\rho \boldsymbol{v}_d = \boldsymbol{f} = \frac{\rho k_r}{\mu} \boldsymbol{K} \cdot (\boldsymbol{\nabla} P + \rho \boldsymbol{g}) \tag{10.56}$$

where $\boldsymbol{K}$ is the intrinsic permeability tensor, ρ is the density, $\boldsymbol{v}_d$ is the Darcy velocity, $\boldsymbol{f}$ is the mass flux, k_r is the relative permeability, μ is the dynamic viscosity, P is pressure and $\boldsymbol{g}$ is gravity.

Parent Block(s) `ARIA MATERIAL`

10.26.1 RELATIVE PERMEABILITY = CONSTANT

Parameters

```
K = REAL
```

Example

```
Relative Permeability = Constant K=1e-3
```

Description The value is constant in space and time.

10.26.2 RELATIVE PERMEABILITY = POLYNOMIAL

Parameters

```
VARIABLE = STRING
ORDER = INT
[C0 = REAL]
[C1 = REAL]
...
[CN = REAL]
```

Example

```
Relative Permeability = Polynomial Variable=Temperature Order=1
C0=401.0 C1=88.5
```

Description Arbitrary order polynomial function of a specified scalar variable.

$$k_r = \sum_{i=0}^{N} C_i X^i \tag{10.57}$$

Here, N is the order of the polynomial provided by the `ORDER` parameter and X is the variable supplied by the `VARIABLE` parameter and C_i are the supplied coefficients. Coefficients that are not supplied default to a value of zero. The `VARIABLE` argument can be `TIME` or any internal Expression that evaluates to a scalar. For the latter case, the format of the `VARIABLE` argument is described in section 2.6.1.

10.27 SKELETON DENSITY

Syntax `SKELETON DENSITY = MODEL [param₁ = val₁, param₂ = val₂ ...]`

Description Specifies a model for the porous skeleton density of a material.

Details Specifies a model for the porous skeleton density of a material.

Parent Block(s) `ARIA MATERIAL`

10.27.1 SKELETON DENSITY = CONSTANT

Parameters `RHO =` *`REAL`*

Example `Skeleton Density = Constant Rho=1e3`

Description The value is constant in space and time.

10.28 SKELETON INTERNAL ENERGY

Syntax `SKELETON INTERNAL ENERGY =` *`MODEL`* [param$_1$ = val$_1$, param$_2$ = val$_2$ ...]

Description Specifies a model for the porous skeleton internal energy of a material.

Details Specifies a model for the porous skeleton internal energy of a material.

Parent Block(s) `ARIA MATERIAL`

10.28.1 SKELETON INTERNAL ENERGY = CONSTANT

Parameters `E =` *`REAL`*

Example `Skeleton Internal Energy = Constant E=2.3e-4`

Description The value is constant in space and time.

10.28.2 SKELETON INTERNAL ENERGY = LINEAR

Parameters `CP =` *`REAL`*
`T_REF =` *`REAL`*

Example `Skeleton Internal Energy = Linear Cp=13.7 T_ref=298.15`

Description The internel energy of the porous skeleton, e_s, is given by the simple relation,

$$e_s = C_p\,(T - T_{ref}) \tag{10.58}$$

where C_p is the specific heat supplied by the `CP` parameter and T_{ref} is a reference temperature supplied by the `T_REF` parameter. This model also supplies the time derivative of e_s,

$$\frac{\partial e_s}{\partial t} = C_p \frac{\partial T}{\partial t} \tag{10.59}$$

10.29 SOLID STRESS

Syntax `SOLID STRESS =` *`MODEL`* `[param`$_1$` = val`$_1$`, param`$_2$` = val`$_2$` ...]`

Description Specifies a contribution to the solid stress tensor. Multiple stresses are combined additvely and may be specified by using this line command multiple times.

Details Specifies a contribution to the solid stress tensor. The total stress $\boldsymbol{T}$ is given by

$$\boldsymbol{T} = \sum_j \boldsymbol{T}_j \tag{10.60}$$

Parent Block(s) `ARIA MATERIAL`

10.29.1 SOLID STRESS = ISOTHERMAL

Parameters `T =` *`REAL`*
`T_REF =` *`REAL`*

Example `SOLID STRESS = Isothermal T=500 T_ref=325`

Description This stress accounts for the mechanical stresses due to thermally induced strains.

$$\boldsymbol{T} = -\beta\,(T - T_{ref})\,\boldsymbol{I} \tag{10.61}$$

where β is the Lamé coefficient of thermal stress (related to the coefficient of thermal expansion, α), T is the temperature and T_{ref} is the temperature of the undeformed reference state of the mesh (pseudo-solid). This is a specialization of the `THERMAL` model that uses uniform, fixed temperature and reference temperature.

10.29.2 SOLID STRESS = LINEAR_ELASTIC

Parameters `REFERENCE_FRAME = ‘‘MOVING’’ | ‘‘UNDEFORMED’’`

Example `SOLID STRESS = Linear_Elastic Reference_Frame=Moving`

Description Supplies the linear elasticity stress tensor,

$$\boldsymbol{T} = \lambda \,\mathrm{trace}\, \boldsymbol{E}\boldsymbol{I} + 2\mu\boldsymbol{E} \tag{10.62}$$

where λ and μ and the Lamé coefficients and $\boldsymbol{E}$ is the strain tensor. When the choice of refernce frame is "UNDEFORMED" then the strain is computed in the undeformed reference state; this is commonly referred to as small strain theory. When the reference frame is "MOVING" then the strain is computed with respect to the deformed coordinates.

Specifically, the strain tensor is given by

$$\boldsymbol{E} = \frac{1}{2}\left(\boldsymbol{\nabla}\boldsymbol{d} + \boldsymbol{\nabla}\boldsymbol{d}^t\right) \tag{10.63}$$

where $\boldsymbol{d}$ is the solid displacement field. The choice of reference frame determines whether the $\boldsymbol{\nabla}$ operator is computed in the undeformed or moving reference frames.

10.29.3 SOLID STRESS = NEOHOOKEAN_ELASTIC

Parameters `(none)`

Example `SOLID STRESS = Neohookean_Elastic`

Description Supplies a nonlinear hyperelastic stress of the form,

$$\boldsymbol{T} = \frac{\mu}{J}\left(\boldsymbol{b} - \boldsymbol{I}\right) + \frac{\lambda}{J}\ln J\boldsymbol{I} \tag{10.64}$$

where λ and μ and the Lamé coefficients, $\boldsymbol{b} \equiv \boldsymbol{F}\cdot\boldsymbol{F}^t$ is the left Cauchy-Green tensor, $\boldsymbol{F}$ is the deformation gradient and $J \equiv \det \boldsymbol{F}$. See, e.g., Bonet and Wood (1997) or Belytschko et al. (2004).

10.29.4 SOLID STRESS = NONLINEAR_ELASTIC

Parameters `(none)`

Example `SOLID STRESS = Nonlinear_Elastic`

Description Supplies a nonlinear elastic stress,

$$\boldsymbol{T} = \lambda \,\mathrm{trace}\, \boldsymbol{E}\boldsymbol{I} + 2\mu \boldsymbol{E} \tag{10.65}$$

where λ and μ and the Lamé coefficients and $\boldsymbol{E}$ is the strain tensor. The particular choice of strain tensor chosen depends on where the configuration (reference frame) which is set via the MESH MOTION command line. See section 3.11 for more information. When the MESH MOTION is set to ARBITRARY then the Green strain is used. Otherwise, the Almansi strain is used.

10.29.5 SOLID STRESS = RESIDUAL

Parameters
```
[SXX | SX = REAL]
[SYY | SY = REAL]
[SZZ | SZ = REAL]
[SXY = REAL]
[SXZ = REAL]
[SYZ = REAL]
```

Example
```
SOLID STRESS = Residual Sxx=0.02 Syy=0.02
```

Description This stress accounts for the initial residual stress in a solid that is constant and uniform everywhere. The components of the residual stress tensor are supplied by the (up-to) six components `SXX`, `SYY`, `SZZ`, `SXY`, `SXZ`, and `SYZ`.

This is directly analogous to the `ISTRESS` condition in ANSYS. To that end, the diagonal components can be specified as either `SXX` or SX etc.

10.29.6 SOLID STRESS = THERMAL

Parameters `(none)`

Example
```
SOLID STRESS = Thermal
```

Description This stress accounts for the mechanical stresses due to thermally induced strains.

$$\boldsymbol{T} = -\beta\,(T - T_{ref})\,\boldsymbol{I} \tag{10.66}$$

where β is the Lamé coefficient of thermal stress (related to the coefficient of thermal expansion, α), T is the temperature and T_{ref} is the temperature of the undeformed reference state of the solid.

10.30 SPECIES DIFFUSION

Syntax `SPECIES DIFFUSION = MODEL [param`$_1$` = val`$_1$`, param`$_2$` = val`$_2$` ...]`

Description Specifies the material (constitutive) model for the species diffusion (diffusive flux) in the bulk.

Details Specifies the material (constitutive) model for the species diffusion (diffusive flux) in the bulk.

Parent Block(s) `ARIA MATERIAL`

10.30.1 SPECIES DIFFUSION = BASIC

This is an alias for `FICKS_LAW`.

Example `Species Diffusion = Basic`

10.30.2 SPECIES DIFFUSION = FICKS_LAW

Parameters `(none)`

Example `Species Diffusion = Ficks_Law`

Description The diffusive species flux $\boldsymbol{q}$ is given by Fick's Law,

$$\boldsymbol{q} = -D\boldsymbol{\nabla}C \tag{10.67}$$

where D is the species diffusivity and C is the species concentration.

10.31 SPECIES DIFFUSIVITY

Syntax `SPECIES DIFFUSIVITY = MODEL [param1 = val1, param2 = val2 ...]`

Description Specifies the material model for the species diffusivity.

Details Specifies the material model for the species diffusivity.

Parent Block(s) `ARIA MATERIAL`

10.31.1 SPECIES DIFFUSIVITY = CONSTANT

Parameters `D = REAL`

Example `SPECIES DIFFUSIVITY = CONSTANT D = 1.0`

Description D is the value of the constant species diffusivity.

10.32 SPECIFIC HEAT

Syntax `SPECIFIC HEAT =` *MODEL* `[param`$_1$` = val`$_1$`, param`$_2$` = val`$_2$` ...]`

Description Specifies the material model for the specific heat.

Details Specifies the material model for the specific heat.

Parent Block(s) `ARIA MATERIAL`

10.32.1 SPECIFIC HEAT = CONSTANT

Parameters `CP =` *REAL*

Example `SPECIFIC HEAT = CONSTANT CP = 1.0`

Description `CP` is the value of the constant specific heat.

10.32.2 SPECIFIC HEAT = CURING_FOAM

Parameters
```
VFRAC_SUBINDEX = INT
[CP_FL = REAL]
[CP_FG = REAL]
[CP_E = REAL]
[PHI_ZERO = REAL]
```

Example
```
Specific Heat = Curing_Foam Vfrac_Subindex=1 Cp_fL=1 Cp_fG=1 Cp_e=1
phi_zero=0.2
```

Description For a curing expoxy with volume fraction ϕ the specific heat is given by

$$\begin{aligned} C_p &= C_{p,f_L}\phi + C_{p,f_G}(\phi_\circ - \phi) + C_{p,e}(1-\phi_\circ) && (10.68)\\ &= a + b\phi && (10.69)\end{aligned}$$

where C_{p,f_L} is the specific heat of the liquid phase flourinert, C_{p,f_G} is the specific heat of the gas phase flourinert, $C_{p,e}$ is the specific heat of the epoxy and $\phi_\circ$ is the reference volume fraction in the flourinert. In the latter form of this relationship

$$\begin{aligned} a &= C_{p,f_L} - C_{p,f_G} && (10.70)\\ b &= C_{p,f_G}\phi_\circ + C_{p,e}(1-\phi_\circ). && (10.71)\end{aligned}$$

NOTE: The volume fraction is assumed to be a `SPECIES` field with the subindex provided by the `VFRAC_SUBINDEX` parameter.

10.32.3 SPECIFIC HEAT = EXPONENTIAL

Parameters
```
VARIABLE = STRING
[CONSTANT = REAL]
[MULTIPLIER = REAL]
EXPONENT = REAL
```

Example
```
Specific Heat = Exponential Variable=Temperature Multiplier=1.0
Exponent=-0.3
```

Description Exponential function of in specified scalar variable. The specific heat is computed as

$$C_p = C + Me^{EX} \qquad (10.72)$$

Here, C is the constant term supplied by the `CONSTANT` parameter which defaults to zero, M is the value supplied by the `MULTIPLIER` parameter which defaults to unity, X is the variable supplied by the `VARIABLE` parameter and E is the exponential multiplier provided by the `EXPONENT` parameter.

10.32.4 SPECIFIC HEAT = POLYNOMIAL

Parameters
```
VARIABLE = STRING
ORDER = INT
[C0 = REAL]
[C1 = REAL]
...
[CN = REAL]
```

Example
```
Specific Heat = Polynomial Variable=Temperature Order=1 C0=401.0
C1=88.5
```

Description Arbitrary order polynomial function of a specified scalar variable.

$$C_p = \sum_{i=0}^{N} C_i X^i \tag{10.73}$$

Here, N is the order of the polynomial provided by the `ORDER` parameter and X is the variable supplied by the `VARIABLE` parameter and C_i are the supplied coefficients. Coefficients that are not supplied default to a value of zero. The `VARIABLE` argument can be `TIME` or any internal Expression that evaluates to a scalar. For the latter case, the format of the `VARIABLE` argument is described in section 2.6.1.

10.32.5 SPECIFIC HEAT = USER_FUNCTION

Parameters

```
NAME = STRING
X = STRING
```

Example

```
begin definition for function Water_Heat_Capacity
   # Source Appendix 2 from "Transport Processes and
   # Unit Operations" by C. J. Geankoplis
   type is piecewise linear
   begin values
      # K     J / kg K
      273.15  4220
      283.15  4195
      293.15  4185
      298.15  4182
      303.15  4181
      313.15  4181
      323.15  4183
      333.15  4187
      343.15  4192
      353.15  4199
      363.15  4208
      373.15  4219
   end
end
...

Begin Aria Material Foo
   ...
   Specific Heat = User_Function X=Temperature Name=Water_Heat_Capacity
   ...
End Aria Material Foo
```

Description A look-up function is used to compute the values of the specific heat as a function of some other variable, i.e. $f(x)$. The function type ("piecewise linear" in the example above) must support the `differentiate()` method for Newton's method.

Here `NAME` is the name of the user-defined function (`Water_Heat_Capacity` in the example) and `X` is the Aria name of the abcissa variable (`TEMPERATURE` in the example). Note that `X` is not necessarily the same name as the abcissa variable identified in the user-defined function (`T` in the example).

10.33 SURFACE TENSION

Syntax `SURFACE TENSION = MODEL [param_1 = val_1, param_2 = val_2 ...]`

Description Specifies the model to use for the surface (interfacial) tension.

Details Specifies the model to use for the surface (interfacial) tension.

Parent Block(s) `ARIA MATERIAL`

10.33.1 SURFACE TENSION = CONSTANT

Parameters `SIGMA = REAL`

Example `Surface Tension = Constant Sigma = 72.0`

Description `SIGMA` is the value of the surface tension.

10.33.2 SURFACE TENSION = LINEAR_T

Parameters
```
SIGMA0 = REAL
DSIGMADT = REAL
T_REF = REAL
```

Example `Surface Tension = Linear_T sigma0=72.  dsigmadT = -.15 T_ref = 298.`

Description `SIGMA0` is the value of the surface tension at the reference temperature `T_REF` and `DSIGMADT` is the derivative of the surface temperature with respect to temperature, i.e.,

$$\sigma = \sigma_0 + m(T - T_{ref}) \tag{10.74}$$

where m is `DSIGMADT`.

10.34 SUSPENSION FLUX

Syntax `SUSPENSION FLUX = MODEL [param_1 = val_1, param_2 = val_2 ...]`

Description Specifies the parameters for the suspension flux model.

Details Specifies the suspension flux model and its parameters for this material.

Parent Block(s) `ARIA MATERIAL`

10.34.1 SUSPENSION FLUX = PHILLIPS

Parameters `K_mu = *REAL*`
`K_c = *REAL*`
`phi_max = *REAL*`
`beta = *REAL*`
`particle_radius = *REAL*`

Example `SUSPENSION FLUX = Phillips K_mu=0.62 K_c=0.41 phi_max=0.68 beta=-1.82 particle_radius=0.01`

Description The Phillips diffusive flux model is intended to be used in conjunction with the Krieger viscosity model (10.40.8). Here, the flux is given by

$$\boldsymbol{q} = \left(K_c a^2 - K_\mu a^2 \beta \frac{\phi}{\phi_m - \phi} \right) \dot{\gamma} \phi \boldsymbol{\nabla} \phi + K_c a^2 \phi^2 \boldsymbol{\nabla} \dot{\gamma} \tag{10.75}$$

where $\dot{\gamma}$ is the shear rate, ϕ is the suspension concentration, ϕ_m is the maximum suspension concentration and a is the particle radius.

10.35 THERMAL CONDUCTIVITY

Syntax `THERMAL CONDUCTIVITY = *MODEL* [param`$_1$` = val`$_1$`, param`$_2$` = val`$_2$` ...]`

Description Specifies the material model for the thermal conductivity.

Details Specifies the material model for the thermal conductivity that appears in the diffusion term of the energy equation for temperature.

Parent Block(s) `ARIA MATERIAL`

10.35.1 THERMAL CONDUCTIVITY = CONSTANT

Parameters `K = *REAL*`

Example `THERMAL CONDUCTIVITY = CONSTANT K = 1.0`

Description `K` is the value of the constant thermal conductivity.

10.35.2 THERMAL CONDUCTIVITY = CURING_FOAM

Parameters
```
RHO_E = REAL
K_F = REAL
K_E = REAL
```

Example
```
Thermal Conductivity = Curing_Foam rho_e=1.3 k_e=14 k_f=2.7
```

Description For a curing expoxy with mixture density ρ

$$\begin{aligned} \kappa &= \frac{2}{3}\left(\frac{\rho}{\rho_e}\right)\kappa_e + \left(1 - \frac{\rho}{\rho_e}\right)\kappa_f \qquad (10.76) \\ &= a + b\rho \qquad (10.77) \end{aligned}$$

where ρ_e is the density of the expoxy, κ_e is the thermal conductivity of the epoxy and κ_f is the thermal conductivity of the fourinert. In the latter form of this relationship

$$\begin{aligned} a &= \kappa_f \qquad (10.78) \\ b &= \frac{1}{\rho_e}\left(\frac{2}{3}\kappa_e - \kappa_f\right) \qquad (10.79) \end{aligned}$$

10.35.3 THERMAL CONDUCTIVITY = POLYNOMIAL

Parameters
```
VARIABLE = STRING
ORDER = INT
[C0 = REAL]
[C1 = REAL]
...
[CN = REAL]
```

Example
```
Thermal Conductivity = Polynomial Variable=Temperature Order=1
C0=401.0 C1=88.5
```

Description Arbitrary order polynomial function of a specified scalar variable.

$$\kappa = \sum_{i=0}^{N} C_i X^i \qquad (10.80)$$

Here, N is the order of the polynomial provided by the `ORDER` parameter and X is the variable supplied by the `VARIABLE` parameter and C_i are the supplied coefficients. Coefficients that are not supplied default to a value of zero. The `VARIABLE` argument can be `TIME` or any internal Expression that evaluates to a scalar. For the latter case, the format of the `VARIABLE` argument is described in section 2.6.1.

10.35.4 THERMAL CONDUCTIVITY = THERMAL

Parameters

```
[A = REAL]
[B = REAL]
[C = REAL]
[D = REAL]
```

Example

```
THERMAL CONDUCTIVITY = THERMAL A = 401.0 B = 88.5
```

Description

Cubic polynomial function of temperature for the conductivity.

$$\kappa = A + BT + CT^2 + DT^3 \tag{10.81}$$

10.35.5 THERMAL CONDUCTIVITY = USER_FUNCTION

Parameters

```
NAME = STRING
X = STRING
```

Example

```
begin definition for function SI_K
  type is piecewise linear
  begin values
     20.0     5.50e7
    100.0     4.60e7
    ...
    800.0     1.30e7
   2000.0     1.30e7
  end values
end definition for function SI_K

...

Begin Aria Material Foo
   ...
   Thermal Conductivity = User_Function Name=SI_K X=Temperature
   ...
End Aria Material Foo
```

Description

A look-up function is used to compute the values of the thermal conductivity as a function of some other variable, i.e. $f(x)$. The function type ("piecewise linear" in the example above) must support the `differentiate()` method for Newton's method.

Here `NAME` is the name of the user-defined function (`RESISTANCE_DATA` in the example) and `X` is the Aria name of the abcissa variable (`TEMPERATURE` in the example). Note that `X` is not necessarily the same name as the optional abcissa variable identified in the user-defined function.

10.36 THERMAL DIFFUSIVITY

Syntax `THERMAL DIFFUSIVITY =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the material model for the thermal diffusivity.

Details Specifies the material model for the thermal diffusivity.

Parent Block(s) `ARIA MATERIAL`

10.36.1 THERMAL DIFFUSIVITY = CONSTANT

Parameters `D =` *`REAL`*

Example `THERMAL DIFFUSIVITY = CONSTANT D = 1.0`

Description `D` is the value of the constant thermal diffusivity.

10.37 TOTAL INTERNAL ENERGY

Syntax `TOTAL INTERNAL ENERGY =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies a model for the total internal energy of a material.

Details Specifies a model for the total internal of a material.

Parent Block(s) `ARIA MATERIAL`

10.37.1 TOTAL INTERNAL ENERGY = POROUS

Parameters `(none)`

Example `Total Internal Energy = Porous`

Description The total internal energy e is computed as

$$e = (1 - \phi)\rho_s e_s + \phi * \rho_f e_f \tag{10.82}$$

where ϕ is the porosity, ρ_s is the density of the solid porous skeleton, e_s is the internal energy of the solid porous skeleton, ρ_f is the density of the fluid phase and e_f is the internal energy of the fluid phase.

10.38 TWO MU

Syntax `TWO MU = MODEL [param₁ = val₁, param₂ = val₂ ...]`

Description Specifies the material model for twice the μ Lamé coefficient.

Details The solid stress $\boldsymbol{T}$ is given by

$$\boldsymbol{T} = \lambda E_{kk}\boldsymbol{I} + 2\mu\boldsymbol{E} - \beta\left(T - T_{ref}\right)\boldsymbol{I} \tag{10.83}$$

where λ and μ are the Lamé coefficients, $\boldsymbol{E} = \frac{1}{2}\left(\boldsymbol{\nabla d} + \boldsymbol{\nabla d}^T\right)$ is the deformation tensor, β is the coefficient of thermal stress, T is temperature and T_{ref} is the solid stress reference temperature.

These Lamé coefficients are related to the more standard Young's modulus, Poisson's ratio and CTE (α) as follows:

$$2\mu = \frac{E}{(1+\nu)} \tag{10.84}$$

$$\lambda = \frac{\nu E}{(1+\nu)(1-2\nu)} = 2\mu\frac{\nu}{(1-2\nu)} \tag{10.85}$$

$$\beta = \frac{\alpha E}{(1-2\nu)} = \alpha\left(3\lambda + 2\mu\right) \tag{10.86}$$

When a user supplies the Young's modulus, Poisson's ratio and CTE properties Aria internally convertes them into the Lamé coefficients.

Parent Block(s) `ARIA MATERIAL`

10.38.1 TWO MU = CONSTANT

Parameters `TWO_MU = REAL`

Example `TWO MU = CONSTANT TWO_MU = 1.0`

Description `TWO_MU` is the value of 2μ.

10.38.2 TWO MU = CONVERTED

Parameters (None)

Example `TWO MU = Converted`

Description Aria will use Young's modulus and Poisson ratio to compute the Lamé μ coefficient. Supplying the Lamé coefficients is more computationally efficient but perhaps less convenient, especially if the material properties are varying (e.g., temperature dependent in a non-isothermal problem).

10.39 VALENCE

Syntax `VALENCE =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the valence (net charge) for a species.

Details Specifies the valence (net charge) for a species.

Parent Block(s) `ARIA MATERIAL`

10.39.1 VALENCE = CONSTANT

Parameters `Z =` *`REAL`*
`[SUBINDEX =` *`INT`*`]`

Example
```
Valence = Constant Subindex=1 Z = 1
Valence = Constant Subindex=2 Z = -1
Valence = Constant Subindex=5 Z = -2
```

Description `Z` is the value of the species valence.

`SUBINDEX` is the species subindex.

10.40 VISCOSITY

Syntax `VISCOSITY =` *`MODEL`* `[param`$_1$ `= val`$_1$`, param`$_2$ `= val`$_2$ `...]`

Description Specifies the material model for the fluid viscosity.

Details Specifies the material model for the fluid viscosity.

Parent Block(s) `ARIA MATERIAL`

10.40.1 VISCOSITY = ARRHENIUS

Parameters `mu0 = REAL`
`E = REAL`

Example `VISCOSITY = Arrhenius mu0=16.4 E=5000.`

Description This model provides a viscosity with an Arrhenius temperature dependence:

$$\mu = \mu_0 e^{-E/T} \tag{10.87}$$

where T is the temperature.

10.40.2 VISCOSITY = BINGHAM_WLF

Parameters

```
MU_ZERO = REAL
MU_INF = REAL
F = REAL
N = REAL
A = REAL
LAMBDA = REAL
TAU_Y = REAL
```

Example `VISCOSITY = Bingham_WLF ...`

Description

$$\mu = \mu_\infty + \left(\mu_\circ - \mu_\infty + \tau_y \frac{1 - e^{-\dot{\gamma} F}}{\dot{\gamma}}\right) \left(1 + (\lambda\dot{\gamma})^a\right)^{\frac{n-1}{a}} \tag{10.88}$$

where $\dot{\gamma}$ is the shear rate.

10.40.3 VISCOSITY = BINGHAM_WLFT

Parameters

```
MU_ZERO = REAL
MU_INF = REAL
F = REAL
N = REAL
A = REAL
LAMBDA = REAL
TAU_Y = REAL
C_1 = REAL
C_2 = REAL
T_REF = REAL
```

Example `VISCOSITY = Bingham_WLFT ...`

Description

$$\mu = a_T \left(\mu_\infty + \left(\mu_\circ - \mu_\infty + \tau_y \frac{1 - e^{-a_T \dot{\gamma} F}}{a_T \dot{\gamma}} \right) (1 + (a_T \lambda \dot{\gamma})^a)^{\frac{n-1}{a}} \right) \tag{10.89}$$

where

$$a_T = e^{\frac{c_1 (T_\circ - T)}{c_2 + T - T_\circ}} \tag{10.90}$$

and T is the temperature and $\dot{\gamma}$ is the shear rate.

10.40.4 VISCOSITY = CARREAU

Parameters

```
MU_ZERO = REAL
[MU_INF = REAL]
[A = REAL]
N = REAL
LAMBDA = REAL
```

Example

```
Viscosity = Carreau ...
```

Description

$$\frac{\mu - \mu_\infty}{\mu_\circ - \mu_\infty} = (1 + (\lambda \dot{\gamma})^a)^{\frac{n-1}{a}} \tag{10.91}$$

or

$$\mu = \mu_\infty + (\mu_\circ - \mu_\infty)(1 + (\lambda \dot{\gamma})^a)^{\frac{n-1}{a}} \tag{10.92}$$

where μ_∞ is the infinite shear viscosity (`MU_INF`, defaults to zero), $\mu_\circ$ is the zero shear viscosity (`MU_ZERO`), n (`N`) and a (`A`, defaults to 2) are model parameters, $\dot{\gamma}$ is the shear rate and λ (`LAMBDA`) is a time constant.

10.40.5 VISCOSITY = CARREAU_T

Parameters

```
MU_ZERO = REAL
[MU_INF = REAL]
[A = REAL]
N = REAL
K = REAL
```

Example

```
Viscosity = Carreau_T ...
```

Description

$$\frac{\mu - \mu_\infty}{\mu_\circ - \mu_\infty} = \left(1 + \left(e^{k/T} \dot{\gamma}\right)^a\right)^{\frac{n-1}{a}} \tag{10.93}$$

or

$$\mu = \mu_\infty + (\mu_\circ - \mu_\infty)\left(1 + \left(e^{k/T} \dot{\gamma}\right)^a\right)^{\frac{n-1}{a}} \tag{10.94}$$

where μ_∞ is the infinite shear viscosity (`MU_INF`, defaults to zero), $\mu_\circ$ is the zero shear viscosity (`MU_ZERO`), n (`N`) and a (`A`, defaults to 2) are model parameters and $\dot{\gamma}$ is the shear rate. The quantity $e^{k/T}$, where T is temperature and k (`K`) is a reference temperature, is a temperature dependent time scale; it takes the place of the constant λ time scale in the `CARREAU` model.

10.40.6 VISCOSITY = CONSTANT

Parameters `MU = `*`REAL`*

Example `VISCOSITY = CONSTANT MU = 1.0`

Description `MU` is the value of the constant fluid viscosity.

10.40.7 VISCOSITY = CURING_FOAM

Parameters

```
VFRAC_SUBINDEX = INT
EXTENT_SUBINDEX = INT
PHI_ZERO = REAL
[A = REAL]
[B = REAL]
[C = REAL]
[KSI_C = REAL]
```

Example

```
Viscosity = Curing_Foam Vfrac_Subindex=1 Extent_Subindex=2
Phi_Zero=0.45
```

Description For a curing expoxy with volume fraction ϕ and extent of reaction ξ the viscosity is given by

$$\mu = \mu_\circ \exp \frac{\phi_\circ - \phi}{1 - \phi_\circ + \phi} \tag{10.95}$$

where $\mu_\circ$ is given by

$$\mu_\circ = (a - bT) \left(\frac{\xi_c^2 - \xi^2}{\xi_c^2} \right)^c \tag{10.96}$$

where T is the tempature. The remaining parameters a, b, c and ξ_c have default values of $a = 20$, $b = 0.22$, $c = -4/3$ and $\xi_c = 0.45$ though they can be overridden with the optional model parameters.

NOTE: The volume fraction is assumed to be a `SPECIES` field with the subindex provided by the `VFRAC_SUBINDEX` parameter. Likewise, the extent of reaction field is assumed to be a `SPECIES` field with the subindex provided by the `EXTENT_SUBINDEX` parameter.

10.40.8 VISCOSITY = KRIEGER

Parameters

```
BETA = REAL
PHI_MAX = REAL
MU_S = REAL
```

Example `VISCOSITY = KRIEGER BETA = -1.65, PHI_MAX = 1.0, MU_S = 1.0`

Description In the viscosity model of Krieger (1972)

$$\mu = \mu_s \left(1 - \frac{\phi}{\phi_m}\right)^{\beta} \tag{10.97}$$

BETA is the Krieger exponent, PHI_MAX is the maximum suspension concentration and MU_S is the solvent viscosity.

10.40.9 VISCOSITY = POLYNOMIAL

Parameters

```
VARIABLE = STRING
ORDER = INT
[C0 = REAL]
[C1 = REAL]
...
[CN = REAL]
```

Example

```
Viscosity = Polynomial Variable=Temperature Order=1 C0=401.0 C1=88.5
```

Description Arbitrary order polynomial function of a specified scalar variable.

$$\mu = \sum_{i=0}^{N} C_i X^i \tag{10.98}$$

Here, N is the order of the polynomial provided by the ORDER parameter and X is the variable supplied by the VARIABLE parameter and C_i are the supplied coefficients. Coefficients that are not supplied default to a value of zero. The VARIABLE argument can be TIME or any internal Expression that evaluates to a scalar. For the latter case, the format of the VARIABLE argument is described in section 2.6.1.

10.40.10 VISCOSITY = POWER_LAW

Parameters

```
K = REAL
N = REAL
```

Example

```
Viscosity = Power_Law K=0.8 N=0.5
```

Description The viscosity is proportional to the shear rate, $\dot{\gamma}$ raised to some power, e.g.,

$$\mu = k\dot{\gamma}^n \tag{10.99}$$

where k (K) and n (N) are model parameters.

10.40.11 VISCOSITY = THERMAL

Parameters
```
[A = REAL]
[B = REAL]
[C = REAL]
[D = REAL]
```

Example
```
VISCOSITY = THERMAL A=1750 C=0.12 D=0
```

Description This model is simply a cubic polynomial in temperature where the viscosity is given by

$$\mu = A + BT + CT^2 + DT^3 \tag{10.100}$$

where T is the temperature.

10.40.12 VISCOSITY = USER_FUNCTION

Parameters
```
NAME = STRING
X = STRING
```

Example

```
begin definition for function Water_Viscosity
   # Source Appendix 2 from "Transport Processes and
   # Unit Operations" by C. J. Geankoplis
   type is piecewise linear
   begin values
      # K      Pa*s (or cP)
      273.15  1.7921
      275.15  1.6728
      277.15  1.5674
      279.15  1.4728
      281.15  1.3860
      283.15  1.3077
      285.15  1.2363
      287.15  1.1709
      289.15  1.1111
      291.15  1.0559
      293.15  1.0050
      293.35  1.0000
      295.15  0.9579
      297.15  0.9142
      299.15  0.8737
      301.15  0.8360
      303.15  0.8007
      305.15  0.7679
      307.15  0.7371
      309.15  0.7085
      311.15  0.6814
      313.15  0.6560
      315.15  0.6321
      317.15  0.6097
      319.15  0.5883
      321.15  0.5683
      323.15  0.5494
      325.15  0.5315
      327.15  0.5146
      329.15  0.4985
      331.15  0.4832
      333.15  0.4688
      335.15  0.4550
      337.15  0.4418
      339.15  0.4293
      341.15  0.4174
      343.15  0.4061
      345.15  0.3952
      347.15  0.3849
      349.15  0.3750
      351.15  0.3655
      353.15  0.3565
      355.15  0.3478
      357.15  0.3395
      359.15  0.3315
      361.15  0.3239
      363.15  0.3165
      365.15  0.3095
      367.15  0.3027
      369.15  0.2962
      371.15  0.2899
      373.15  0.2838
   end
end
...
```

Description A look-up function is used to compute the values of the viscosity as a function of some other variable, i.e. $f(x)$. The function type ("piecewise linear" in the example above) must support the `differentiate()` method for Newton's method.

Here `NAME` is the name of the user-defined function (`Water_Viscosity` in the example) and `X` is the Aria name of the abcissa variable (`TEMPERATURE` in the example). Note that `X` is not necessarily the same name as the abcissa variable identified in the user-defined function (`T` in the example).

10.40.13 VISCOSITY = WELD

Parameters `[BETA =` *`REAL`*`]`
`C0 =` *`REAL`*
`C1 =` *`REAL`*
`C2 =` *`REAL`*
`C3 =` *`REAL`*
`T_LIQ =` *`REAL`*`T_90 =` *`REAL`*`T_MAX =` *`REAL`*

Example
```
Viscosity = Weld C0=1 C1=-1e-2 C2=0 C3=0 T_LIQ=920 T_MAX=1400
T_90=1000
```

Description This is an emprical model that emulates the melting of a solid metal during the laser welding process.

$$\mu = \begin{cases} \mu_{90} + (\mu_{liq} - \mu_{90}) \frac{T - T_{90}}{T_{liq} - T_{90}} & : \quad T < T_{liq} \\ c_0 + c_1 \hat{T} + c_2 \hat{T}^2 + c_3 \hat{T}^3 & : \quad T >= T_{liq} \end{cases} \tag{10.101}$$

where μ_{liq} is given by

$$\mu_{liq} = c_0 + c_1 T_{liq} + c_2 T_{liq}^2 + c_3 T_{liq}^3, \tag{10.102}$$

$\mu_{90} = \beta \mu_{liq}$ and $\hat{T} = \min(T, T_{max})$. The default value of `BETA` is 10^{11}.

10.41 YOUNGS MODULUS

Syntax `YOUNGS MODULUS =` *`MODEL`*`[param`$_1$` = val`$_1$`, param`$_2$` = val`$_2$` ...]`

Description Specifies the material model for the Young's modulus.

Details The solid stress $\boldsymbol{T}$ is given by

$$\boldsymbol{T} = \lambda E_{kk}\boldsymbol{I} + 2\mu\boldsymbol{E} - \beta\left(T - T_{ref}\right)\boldsymbol{I} \tag{10.103}$$

where λ and μ are the Lamé coefficients, $\boldsymbol{E} = \frac{1}{2}\left(\boldsymbol{\nabla d} + \boldsymbol{\nabla d}^T\right)$ is the deformation tensor, β is the coefficient of thermal stress, T is temperature and T_{ref} is the solid stress reference temperature.

These Lamé coefficients are related to the more standard Young's modulus, Poisson's ratio and CTE (α) as follows:

$$\begin{aligned} 2\mu &= \frac{E}{(1+\nu)} && (10.104) \\ \lambda &= \frac{\nu E}{(1+\nu)(1-2\nu)} = 2\mu\frac{\nu}{(1-2\nu)} && (10.105) \\ \beta &= \frac{\alpha E}{(1-2\nu)} = \alpha\left(3\lambda + 2\mu\right) && (10.106) \end{aligned}$$

When a user supplies the Young's modulus, Poisson's ratio and CTE properties Aria internally convertes them into the Lamé coefficients.

Supplying the Lamé coefficients is more computationally efficient but perhaps less convenient, especially if the material properties are varying (e.g., temperature dependent in a non-isothermal problem).

Parent Block(s) `ARIA MATERIAL`

10.41.1 YOUNGS MODULUS = CONSTANT

Parameters `YM = *REAL*`

Example `YOUNGS MODULUS = CONSTANT YM = 1.0`

Description `YM` is the value of the constant Young's modulus.

Chapter 11

Solution Control Reference

11.1 TRANSFER

```
Begin TRANSFER transfer_name

    COPY { VOLUME | SURFACE } { ELEMENTS | NODES | CONSTRAINTS } FROM from_region_name
    TO to_region_name

    INTERPOLATE { VOLUME | SURFACE } { ELEMENTS | NODES | CONSTRAINTS } FROM from_region_name
    TO to_region_name

    SEND BLOCK from_blocks TO to_blocks

    SEND FIELD source_field_name STATE { NONE | NEW | OLD | NM1 | NM2 | NM3 | NM4 } TO
    destination_field_name STATE { NONE | NEW | OLD | NM1 | NM2 | NM3 | NM4 } [ LOWER BOUND
    lower_bound UPPER BOUND upper_bound ]

    SEARCH TYPE { = | IS | ARE } [ { PARALLEL | PROXIMITY | DETAILED } { PARALLEL |
    PROXIMITY | DETAILED } { PARALLEL | PROXIMITY | DETAILED } ]

    NODES OUTSIDE REGION { = | IS | ARE } { IGNORE | EXTRAPOLATE }

    SEARCH COORDINATE FIELD source_field_name STATE { NONE | NEW | OLD | NM1 | NM2 | NM3
    | NM4 } TO destination_field_name STATE { NONE | NEW | OLD | NM1 | NM2 | NM3 | NM4 }

    SEARCH SURFACE GAP TOLERANCE { = | IS | ARE } surface_gap_tolerance

    SEARCH GEOMETRIC TOLERANCE { = | IS | ARE } geometric_tolerance

    FROM { ELEMENTS | NODES | CONSTRAINTS } TO { ELEMENTS | NODES | CONSTRAINTS | GAUSS_POINTS }

    INTERPOLATION FUNCTION User_Subroutine

    ALL FIELDS

    EXCLUDE GHOSTED

    USE PREDEFINED TRANSFER predefined_transfer_name FROM from_region TO to_region

End
```

Details transfer region/mesh information. the mechanics/variables information will get sorted out by the calling procedure.

11.1.1 COPY

Syntax

`COPY { VOLUME | SURFACE } { ELEMENTS | NODES | CONSTRAINTS } FROM` *from_region_name* `TO` *to_region_name*

from_region_name : *no description* (C)

to_region_name : *no description* (C)

Details

transfer from region/block to region/block

11.1.2 INTERPOLATE

Syntax

`INTERPOLATE { VOLUME | SURFACE } { ELEMENTS | NODES | CONSTRAINTS } FROM` *from_region_name* `TO` *to_region_name*

from_region_name : *no description* (C)

to_region_name : *no description* (C)

Details

transfer from region/block to region/block

11.1.3 SEND BLOCK

Syntax

`SEND BLOCK` *from_blocks* `TO` *to_blocks*

from_blocks : *no description* (C [, ...])

to_blocks : *no description* (C [, ...])

Details

Add element blocks to a particular same mesh element copy transfer operator.

11.1.4 SEND FIELD

Syntax

`SEND FIELD` *source_field_name* `STATE { NONE | NEW | OLD | NM1 | NM2 | NM3 | NM4 } TO` *destination_field_name* `STATE { NONE | NEW | OLD | NM1 | NM2 | NM3 | NM4 } [ LOWER BOUND` *lower_bound* `UPPER BOUND` *upper_bound* `]`

source_field_name : *no description* (C)

destination_field_name : *no description* (C)

lower_bound : *no description* (R)

upper_bound : *no description* (R)

Details

Specifies the mapping between source and destination field names. example SEND FIELD velocity TO velocity SEND FIELD temp TO temperature lower bound 0 SEND FIELD x TO y lower bound 10 upper bound 100

11.1.5 SEARCH TYPE

Details

11.1.6 NODES OUTSIDE REGION

Details

11.1.7 SEARCH COORDINATE FIELD

Syntax `SEARCH COORDINATE FIELD` *source_field_name* `STATE` { `NONE | NEW | OLD | NM1 | NM2 | NM3 | NM4` } `TO` *destination_field_name* `STATE` { `NONE | NEW | OLD | NM1 | NM2 | NM3 | NM4` }

source_field_name : *no description* (C)

destination_field_name : *no description* (C)

Details

11.1.8 SEARCH SURFACE GAP TOLERANCE

Syntax `SEARCH SURFACE GAP TOLERANCE` { `= | IS | ARE` } *surface_gap_tolerance*

surface_gap_tolerance : *no description* (R)

Details

11.1.9 SEARCH GEOMETRIC TOLERANCE

Syntax `SEARCH GEOMETRIC TOLERANCE` { `= | IS | ARE` } *geometric_tolerance*

geometric_tolerance : *no description* (R)

Details

11.1.10 FROM

Details Allows the send/receive mesh objects to be different.

11.1.11 INTERPOLATION FUNCTION

Syntax `INTERPOLATION FUNCTION` *User_Subroutine*

User_Subroutine : *no description* (C)

Details Allows an application defined subroutine to be used for the interpolation.

11.1.12 ALL FIELDS

Details Select all fields for transfer that have same name and state for source and destination regions.

11.1.13 EXCLUDE GHOSTED

Details exclude ghosted nodes from a copy transfer

11.1.14 USE PREDEFINED TRANSFER

Syntax `USE PREDEFINED TRANSFER` *predefined_transfer_name* `FROM` *from_region* `TO` *to_region*

predefined_transfer_name : *no description* (C)

from_region : *no description* (C)

to_region : *no description* (C)

Details Use predefine transfer semantics provided by the specified name.

11.2 SOLUTION CONTROL DESCRIPTION

```
Begin SOLUTION CONTROL DESCRIPTION name


    USE SYSTEM name Begin SYSTEM name


    End
    Begin SUBSYSTEM name


    End
    Begin INITIALIZE name


    End
    Begin PARAMETERS FOR type-name


    End
```

```
End
```

Details Contains the commands needed to execute an analysis using the Calagio procedure that utilizes Solver Control.

11.2.1 USE SYSTEM

Syntax `USE SYSTEM` *name*

name : *no description* (C [, ...])

Details This set the name of which system to use.

11.3 SYSTEM

```
Begin SYSTEM name


    EVENT name [ WHEN when-expression ]

    SIMULATION START TIME { = | IS } number

    SIMULATION TERMINATION TIME { = | IS } number

    SIMULATION MAX GLOBAL ITERATIONS { = | IS } number

    TRANSFER name [ WHEN when-expression ]

    USE INITIALIZE name

    OUTPUT name [ WHEN when-expression ] Begin TRANSIENT name


    End
    Begin SEQUENTIAL name


    End

End
```

Details This block wraps a solver system for a given name. The NAME parameter is the name used to define the system. There can be more than one system block in the Solver Control Description block. The "use system NAME" line commmand controls which one is to be used.

11.3.1 EVENT

Syntax `EVENT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

11.3.2 SIMULATION START TIME

Syntax `SIMULATION START TIME` { = | IS } *number*

number : *no description* (R)

Details Simulation starting time. (by default 0.0)

11.3.3 SIMULATION TERMINATION TIME

Syntax `SIMULATION TERMINATION TIME` { = | IS } *number*

number : *no description* (R)

Details The drop dead time.

11.3.4 SIMULATION MAX GLOBAL ITERATIONS

Syntax `SIMULATION MAX GLOBAL ITERATIONS` { = | IS } *number*

number : *no description* (I)

Details The Total number of Solves.

11.3.5 TRANSFER

Syntax `TRANSFER` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

11.3.6 USE INITIALIZE

Syntax `USE INITIALIZE` *name*

name : *no description* (C [, ...])

Details This set the name of which initialization to use.

11.3.7 OUTPUT

Syntax `OUTPUT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])
when-expression : *no description* (Q)

Details A Solver Control Output line command which execute a perform I/O on the region.

11.4 TRANSIENT

```
Begin TRANSIENT name


    ADVANCE name [ WHEN when-expression ]
    EVENT name [ WHEN when-expression ]
    TRANSFER name [ WHEN when-expression ]
    OUTPUT name [ WHEN when-expression ]
    INVOLVE name Begin NONLINEAR name


    End
    Begin SUBCYCLE name


    End
    Begin MATRIX FREE NONLINEAR name


    End

End
```

Details This block is used to wrap a time loop.

11.4.1 ADVANCE

Syntax `ADVANCE` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

11.4.2 EVENT

Syntax `EVENT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

11.4.3 TRANSFER

Syntax `TRANSFER` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

11.4.4 OUTPUT

Syntax `OUTPUT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Output line command which execute a perform I/O on the region.

11.4.5 INVOLVE

Syntax `INVOLVE` *name*

name : *no description* (C)

Details Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

11.5 NONLINEAR

```
Begin NONLINEAR name

    ADVANCE name [ WHEN when-expression ]
    EVENT name [ WHEN when-expression ]
    TRANSFER name [ WHEN when-expression ]
    OUTPUT name [ WHEN when-expression ]
    INVOLVE name Begin NONLINEAR name

    End
    Begin SUBCYCLE name

    End

End
```

Details This block is used to wrap a nonlinear solve loop.

11.5.1 ADVANCE

Syntax `ADVANCE` *name* `[ WHEN` *when-expression* `]`

name : no description (C [, ...])

when-expression : no description (Q)

Details Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

11.5.2 EVENT

Syntax `EVENT` *name* `[ WHEN` *when-expression* `]`

name : no description (C [, ...])

when-expression : no description (Q)

Details Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

11.5.3 TRANSFER

Syntax `TRANSFER` *name* `[ WHEN` *when-expression* `]`

`name` `:` *`no description`* `(C [, ...])`

`when-expression` `:` *`no description`* `(Q)`

Details A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

11.5.4 OUTPUT

Syntax `OUTPUT` *name* `[ WHEN` *when-expression* `]`

`name` `:` *`no description`* `(C [, ...])`

`when-expression` `:` *`no description`* `(Q)`

Details A Solver Control Output line command which execute a perform I/O on the region.

11.5.5 INVOLVE

Syntax `INVOLVE` *name*

`name` `:` *`no description`* `(C)`

Details Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

11.6 NONLINEAR

```
Begin NONLINEAR name


End
```

Details This block is used to wrap a nonlinear solve loop.

11.7 SUBCYCLE

```
Begin SUBCYCLE name

    ADVANCE name [ WHEN when-expression ]

    EVENT name [ WHEN when-expression ]

    TRANSFER name [ WHEN when-expression ]

    OUTPUT name [ WHEN when-expression ]

    INVOLVE name Begin SUBCYCLE name

    End

End
```

Details This block is used to wrap a subcycle time loop.

11.7.1 ADVANCE

Syntax `ADVANCE` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

11.7.2 EVENT

Syntax `EVENT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

11.7.3 TRANSFER

Syntax `TRANSFER` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

11.7.4 OUTPUT

Syntax `OUTPUT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Output line command which execute a perform I/O on the region.

11.7.5 INVOLVE

Syntax `INVOLVE` *name*

name : *no description* (C)

Details Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

11.8 SUBCYCLE

```
Begin SUBCYCLE name


End
```

Details This block is used to wrap a subcycle time loop.

11.9 SUBCYCLE

```
Begin SUBCYCLE name
```

```
    ADVANCE name [ WHEN when-expression ]
    EVENT name [ WHEN when-expression ]
    TRANSFER name [ WHEN when-expression ]
    OUTPUT name [ WHEN when-expression ]
    INVOLVE name Begin SUBCYCLE name

    End

End
```

Details This block is used to wrap a subcycle time loop.

11.9.1 ADVANCE

Syntax `ADVANCE name [ WHEN when-expression ]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

11.9.2 EVENT

Syntax `EVENT name [ WHEN when-expression ]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

11.9.3 TRANSFER

Syntax `TRANSFER name [ WHEN when-expression ]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

11.9.4 OUTPUT

Syntax `OUTPUT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Output line command which execute a perform I/O on the region.

11.9.5 INVOLVE

Syntax `INVOLVE` *name*

name : *no description* (C)

Details Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

11.10 SUBCYCLE

```
Begin SUBCYCLE name


End
```

Details This block is used to wrap a subcycle time loop.

11.11 MATRIX FREE NONLINEAR

```
Begin MATRIX FREE NONLINEAR name


    ADVANCE name [ WHEN when-expression ]
    EVENT name [ WHEN when-expression ]
    USE SUBSYSTEM name
    TRANSFER name [ WHEN when-expression ]
    REGISTER REGION name
    REGISTER TRANSFER name
    USE COUPLER coupler_name
    INVOLVE name Begin MATRIX FREE NONLINEAR name
    End

End
```

Details This block is used to wrap a nonlinear solve loop.

11.11.1 ADVANCE

Syntax `ADVANCE` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

11.11.2 EVENT

Syntax `EVENT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

11.11.3 USE SUBSYSTEM

Syntax `USE SUBSYSTEM` *name*

name : *no description* (C [, ...])

Details This set the name of which subsystem to include.

11.11.4 TRANSFER

Syntax `TRANSFER` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

11.11.5 REGISTER REGION

Syntax `REGISTER REGION` *name*

name : *no description* (C [, ...])

Details Register 1 to many regions to participate in a Matrix Free coupled solve.

11.11.6 REGISTER TRANSFER

Syntax `REGISTER TRANSFER` *name*

name : *no description* (C [, ...])

Details Register 1 to many regions to participate in a Matrix Free coupled solve.

11.11.7 USE COUPLER

Syntax `USE COUPLER` *coupler_name*

coupler_name : *no description* (C)

Details Specifiy which coupler solver block to use for setting solver parameters.

11.11.8 INVOLVE

Syntax `INVOLVE` *name*

name : *no description* (C)

Details Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

11.12 MATRIX FREE NONLINEAR

`Begin MATRIX FREE NONLINEAR` *name*

```
End
```

Details This block is used to wrap a nonlinear solve loop.

11.13 SEQUENTIAL

```
Begin SEQUENTIAL name

    ADVANCE name [ WHEN when-expression ]

    EVENT name [ WHEN when-expression ]

    TRANSFER name [ WHEN when-expression ]

    OUTPUT name [ WHEN when-expression ]

    INVOLVE name Begin NONLINEAR name


    End
    Begin MATRIX FREE NONLINEAR name


    End

End
```

Details This block is used to wrap a sequential solution. It is used to wrap a sequence of Non-Linear or pseudo time solve step solves.

11.13.1 ADVANCE

Syntax `ADVANCE` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

11.13.2 EVENT

Syntax `EVENT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

11.13.3 TRANSFER

Syntax `TRANSFER` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

11.13.4 OUTPUT

Syntax `OUTPUT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Output line command which execute a perform I/O on the region.

11.13.5 INVOLVE

Syntax `INVOLVE` *name*

name : *no description* (C)

Details Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

11.14 NONLINEAR

`Begin NONLINEAR` *name*

```
    ADVANCE name [ WHEN when-expression ]
    EVENT name [ WHEN when-expression ]
    TRANSFER name [ WHEN when-expression ]
    OUTPUT name [ WHEN when-expression ]
    INVOLVE name Begin NONLINEAR name

    End
    Begin SUBCYCLE name

    End

End
```

Details This block is used to wrap a nonlinear solve loop.

11.14.1 ADVANCE

Syntax `ADVANCE` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

11.14.2 EVENT

Syntax `EVENT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

11.14.3 TRANSFER

Syntax `TRANSFER` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

11.14.4 OUTPUT

Syntax `OUTPUT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Output line command which execute a perform I/O on the region.

11.14.5 INVOLVE

Syntax `INVOLVE` *name*

name : *no description* (C)

Details Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

11.15 NONLINEAR

```
Begin NONLINEAR name


End
```

Details This block is used to wrap a nonlinear solve loop.

11.16 SUBCYCLE

```
Begin SUBCYCLE name


    ADVANCE name [ WHEN when-expression ]
    EVENT name [ WHEN when-expression ]
    TRANSFER name [ WHEN when-expression ]
    OUTPUT name [ WHEN when-expression ]
    INVOLVE name Begin SUBCYCLE name


    End

End
```

Details This block is used to wrap a subcycle time loop.

11.16.1 ADVANCE

Syntax `ADVANCE` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

11.16.2 EVENT

Syntax `EVENT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

11.16.3 TRANSFER

Syntax `TRANSFER` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

11.16.4 OUTPUT

Syntax `OUTPUT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Output line command which execute a perform I/O on the region.

11.16.5 INVOLVE

Syntax `INVOLVE` *name*

name : *no description* (C)

Details Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

11.17 SUBCYCLE

```
Begin SUBCYCLE name


End
```

Details This block is used to wrap a subcycle time loop.

11.18 MATRIX FREE NONLINEAR

```
Begin MATRIX FREE NONLINEAR name


    ADVANCE name [ WHEN when-expression ]
    EVENT name [ WHEN when-expression ]
    USE SUBSYSTEM name
    TRANSFER name [ WHEN when-expression ]
    REGISTER REGION name
    REGISTER TRANSFER name
    USE COUPLER coupler_name
    INVOLVE name Begin MATRIX FREE NONLINEAR name
    End

End
```

Details This block is used to wrap a nonlinear solve loop.

11.18.1 ADVANCE

Syntax `ADVANCE` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

11.18.2 EVENT

Syntax `EVENT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

11.18.3 USE SUBSYSTEM

Syntax `USE SUBSYSTEM` *name*

name : *no description* (C [, ...])

Details This set the name of which subsystem to include.

11.18.4 TRANSFER

Syntax `TRANSFER` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

11.18.5 REGISTER REGION

Syntax `REGISTER REGION` *name*

name : *no description* (C [, ...])

Details Register 1 to many regions to participate in a Matrix Free coupled solve.

11.18.6 REGISTER TRANSFER

Syntax `REGISTER TRANSFER` *name*

name : *no description* (C [, ...])

Details Register 1 to many regions to participate in a Matrix Free coupled solve.

11.18.7 USE COUPLER

Syntax `USE COUPLER` *coupler_name*

coupler_name : *no description* (C)

Details Specifiy which coupler solver block to use for setting solver parameters.

11.18.8 INVOLVE

Syntax `INVOLVE` *name*

name : *no description* (C)

Details Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

11.19 MATRIX FREE NONLINEAR

`Begin MATRIX FREE NONLINEAR` *name*

```
End
```

Details This block is used to wrap a nonlinear solve loop.

11.20 SUBSYSTEM

```
Begin SUBSYSTEM name

    ADVANCE name [ WHEN when-expression ]
    EVENT name [ WHEN when-expression ]
    USE SUBSYSTEM name
    TRANSFER name [ WHEN when-expression ]
    OUTPUT name [ WHEN when-expression ]
    INVOLVE name Begin MATRIX FREE NONLINEAR name

    End

End
```

Details This block wraps a solver subsystem for a given name. The NAME parameter is the name used to define the system. There can be more than one system block in the Solver Control Description block. The "use subsystem NAME" line commmand controls where it will be included in a solver system.

11.20.1 ADVANCE

Syntax `ADVANCE` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

11.20.2 EVENT

Syntax `EVENT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details — Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

11.20.3 USE SUBSYSTEM

Syntax — `USE SUBSYSTEM` *name*

name : *no description* (C [, ...])

Details — This set the name of which subsystem to include.

11.20.4 TRANSFER

Syntax — `TRANSFER` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details — A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

11.20.5 OUTPUT

Syntax — `OUTPUT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details — A Solver Control Output line command which execute a perform I/O on the region.

11.20.6 INVOLVE

Syntax — `INVOLVE` *name*

name : *no description* (C)

Details — Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

11.21 MATRIX FREE NONLINEAR

```
Begin MATRIX FREE NONLINEAR name

    ADVANCE name [ WHEN when-expression ]
    EVENT name [ WHEN when-expression ]
    USE SUBSYSTEM name
    TRANSFER name [ WHEN when-expression ]
    REGISTER REGION name
    REGISTER TRANSFER name
    USE COUPLER coupler_name
    INVOLVE name Begin MATRIX FREE NONLINEAR name

    End

End
```

Details This block is used to wrap a nonlinear solve loop.

11.21.1 ADVANCE

Syntax `ADVANCE name [ WHEN when-expression ]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

11.21.2 EVENT

Syntax `EVENT name [ WHEN when-expression ]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

11.21.3 USE SUBSYSTEM

Syntax `USE SUBSYSTEM` *name*

name : *no description* (C [, ...])

Details This set the name of which subsystem to include.

11.21.4 TRANSFER

Syntax `TRANSFER` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

11.21.5 REGISTER REGION

Syntax `REGISTER REGION` *name*

name : *no description* (C [, ...])

Details Register 1 to many regions to participate in a Matrix Free coupled solve.

11.21.6 REGISTER TRANSFER

Syntax `REGISTER TRANSFER` *name*

name : *no description* (C [, ...])

Details Register 1 to many regions to participate in a Matrix Free coupled solve.

11.21.7 USE COUPLER

Syntax `USE COUPLER` *coupler_name*

coupler_name : *no description* (C)

Details Specifiy which coupler solver block to use for setting solver parameters.

11.21.8 INVOLVE

Syntax `INVOLVE` *name*

name : *no description* (C)

Details Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

11.22 MATRIX FREE NONLINEAR

```
Begin MATRIX FREE NONLINEAR name


End
```

Details This block is used to wrap a nonlinear solve loop.

11.23 INITIALIZE

```
Begin INITIALIZE name


    ADVANCE name [ WHEN when-expression ]
    EVENT name [ WHEN when-expression ]
    TRANSFER name [ WHEN when-expression ]
    INVOLVE name
End
```

Details This block wraps a initializer for a given name. The NAME parameter is the name used to define the initialization block. There can be more than one initialize block in the Solver Control Description block. The ”use initialize NAME” line commmand controls which one is to be used.

11.23.1 ADVANCE

Syntax `ADVANCE` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that advances the solution. The name is that matches the physics.

11.23.2 EVENT

Syntax `EVENT` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details Used within a Solver Control block to indicate a single step that has no time associated with it. It can cause a solution transfer between regions or cause something to print.

11.23.3 TRANSFER

Syntax `TRANSFER` *name* `[ WHEN` *when-expression* `]`

name : *no description* (C [, ...])

when-expression : *no description* (Q)

Details A Solver Control Transfer line command which executes all transfers defined from the specified region. All transfers with a send region of 'name' will be executed.

11.23.4 INVOLVE

Syntax `INVOLVE` *name*

name : *no description* (C)

Details Specifiy a physics participant to a coupled problem solved using matrix-free nonlinear.

11.24 PARAMETERS FOR

`Begin PARAMETERS FOR` *type-name*

`TARGET ERROR FOR` *region-name* `MeshObjectType` *field-name* { `=` | `IS` } *target-number*

`INITIAL DELTAT` { `=` | `IS` } *number*

`TERMINATION TIME` { `=` | `IS` } *number*

`TOTAL CHANGE IN TIME` { `=` | `IS` } *number*

`NUMBER OF STEPS` { `=` | `IS` } *number*

`START TIME` { `=` | `IS` } *number*

```
    CONVERGED WHEN convergence-expression
    TIME STEP STYLE TimeStepStyle Begin PARAMETERS FOR ARIA REGION RegionName

    End

End
```

Details A Solver Control PARAMETERS block to set up control data for the SC_type parameter. Inside this block one sets the time step parameters or nonlinear parameters.

11.24.1 TARGET ERROR FOR

Syntax `TARGET ERROR FOR` *region-name* `MeshObjectType` *field-name* { `=` | `IS` } *target-number*

region-name : *no description* (C)

field-name : *no description* (C)

target-number : *no description* (R)

Details Assign a target number for a mesh object in a region for convergence.

11.24.2 INITIAL DELTAT

Syntax `INITIAL DELTAT` { `=` | `IS` } *number*

number : *no description* (R)

Details Assign an initial delta T

11.24.3 TERMINATION TIME

Syntax `TERMINATION TIME` { `=` | `IS` } *number*

number : *no description* (R)

Details Assign a final time to stop

11.24.4 TOTAL CHANGE IN TIME

Syntax `TOTAL CHANGE IN TIME` { `=` | `IS` } *number*

number : *no description* (R)

Details Use this number and the initial time to compute termination time.

11.24.5 NUMBER OF STEPS

Syntax `NUMBER OF STEPS` { `=` | `IS` } *number*

`number` `:` *`no description`* `(I)`

Details The number steps to run the time or nonlinear loop

11.24.6 START TIME

Syntax `START TIME` { `=` | `IS` } *number*

`number` `:` *`no description`* `(R)`

Details Assign a start time.

11.24.7 CONVERGED WHEN

Syntax `CONVERGED WHEN` *convergence-expression*

`convergence-expression` `:` *`no description`* `(Q [, ...])`

Details Set the convergence expression.

11.24.8 TIME STEP STYLE

Syntax `TIME STEP STYLE` *TimeStepStyle*

`TimeStepStyle` `:` *`no description`* { `NOSNAP` | `NOCLIP` | `SNAP` | `CLIP` }

Details Set the time stepping style.

Enums `TimeStepStyle`

- `NOSNAP` - *no description*
- `NOCLIP` - *no description*
- `SNAP` - *no description*
- `CLIP` - *no description*

11.25 PARAMETERS FOR ARIA REGION

```
Begin PARAMETERS FOR ARIA REGION RegionName

     INITIAL TIME STEP SIZE { = | IS } dt
     MINIMUM TIME STEP SIZE { = | IS } dt
     TIME STEP VARIATION { = | IS } time_step_variation
     PREDICTOR-CORRECTOR TOLERANCE { = | IS } predictor_corrector_tolerance
     COURANT LIMIT { = | IS } courant_limit
     MAXIMUM TIME STEP SIZE { = | IS } dt
End
```

Details Defines region specific time stepping data

11.25.1 INITIAL TIME STEP SIZE

Syntax `INITIAL TIME STEP SIZE { = | IS }` *dt*

dt : *no description* (R)

Details Specifies the initial time step size. This may remain constant over the run.

11.25.2 MINIMUM TIME STEP SIZE

Syntax `MINIMUM TIME STEP SIZE { = | IS }` *dt*

dt : *no description* (R)

Details Specifies the minimum time step size. Default is 0.0.

11.25.3 TIME STEP VARIATION

Syntax `TIME STEP VARIATION { = | IS }` *time_step_variation*

time_step_variation : *no description* (C)

Details Specifies how the time step sizes are to be derived. It's nice that this can vary from time block to time block, n'est ce pas?

11.25.4 PREDICTOR-CORRECTOR TOLERANCE

Syntax `PREDICTOR-CORRECTOR TOLERANCE` { `=` | `IS` } *predictor_corrector_tolerance*

predictor_corrector_tolerance : *no description* (R)

Details The Predictor-Corrector difference tolerance. Default is 0.001.

11.25.5 COURANT LIMIT

Syntax `COURANT LIMIT` { `=` | `IS` } *courant_limit*

courant_limit : *no description* (R)

Details The Courant Number limit. Default is 0 (INACTIVE).

11.25.6 MAXIMUM TIME STEP SIZE

Syntax `MAXIMUM TIME STEP SIZE` { `=` | `IS` } *dt*

dt : *no description* (R)

Details Specifies the maximum time step size. Default is Real_MAX.

Chapter 12

Time Integration Commands

12.1 Setting Up a Transient Problem

Aria uses the *solution control* library from the SIERRA Framework configuring simulations. All Aria input files must include a Solution Control Description block in the Procedure section of the input file. Here's an example of such a block:

```
.
.
.
Begin Procedure My_Aria_Procedure

   Begin Solution Control Description

      Use System Main

      Begin System Main
         Simulation Start Time            = 0.0
         Simulation Termination Time      = 10.0
         Simulation Max Global Iterations = 1000

         Begin Transient Time_Block_1
            Advance My_Aria_Region
         End
         Begin Transient Time_Block_2
            Advance My_Aria_Region
         End

      End

      Begin Parameters For Transient Time_Block_1
         Start Time      = 0.0
         Number of steps = 8
         Begin Parameters For Aria Region My_Aria_Region
            Time Step Variation    = Fixed
            Initial Time Step Size = 0.001
         End
      End

      Begin Parameters For Transient Time_Block_2
         Begin Parameters For Aria Region My_Aria_Region
            Time Step Variation           = Adaptive
```

```
           Initial Time Step Size        = 0.001
           Predictor-Corrector Tolerance = 1e-3
           Minimum Time Step Size        = 1e-6
        End
     End

  End
  .
  .
  .
```

12.2 INITIAL TIME STEP SIZE

Syntax INITIAL TIME STEP SIZE = *REAL*

Parent Block(s) PARAMETERS FOR ARIA REGION

Description Initial time step size for the time block.

Details Initial time step size for the time block.

Example `INITIAL TIME STEP SIZE = 0.001`

12.3 MINIMUM TIME STEP SIZE

Syntax MINIMUM TIME STEP SIZE = *REAL*

Parent Block(s) PARAMETERS FOR ARIA REGION

Description Minimum time step size for this Aria region.

Details Minimum time step size for this Aria region. Regardless of what the adaptive time step selection routine determines and regardless of what other regions request for time step sizes (for loosely coupled simulations) the time step will not be allowed to fall below this value.

Example `Minimum Time Step Size = 1e-8`

12.4 TIME STEP VARIATION

Syntax `TIME STEP VARIATION =` *`STRING`*

Parent Block(s) `PARAMETERS FOR ARIA REGION`

Description Choose between `FIXED` and `ADAPTIVE` time step selection methods.

Details Choose between `FIXED` and `ADAPTIVE` time step selection methods.

Example `Time Step Variation = FIXED`

12.5 PREDICTOR-CORRECTOR TOLERANCE

Syntax `PREDICTOR-CORRECTOR TOLERANCE =` *`REAL`*

Parent Block(s) `PARAMETERS FOR ARIA REGION`

Description Specifies the tolerance for the difference between the predicted solutiona and the implictly solved corrector solution. Used in adaptive time step selection.

Details The adaptive time step selection formula is

$$\Delta t_{n+1} = \Delta t_n \left(b \frac{\epsilon}{d_{n+1}} \right)^m \tag{12.1}$$

where $\Delta t_n \equiv t_{n+1} - t_n$ is the time step size from the most recent solution, $\Delta t_{n+1} \equiv t_{n+2} - t_{n+1}$ is the new time step size, ϵ is the predictor-corrector tolerance and d_{n+1} is the norm of the difference between the predicted and actual solutions at time t_{n+1}. For first order time integration $m = 1/2$ and $b = 2$. For second order time integration $m = 1/3$ and $b = 3(1 + \Delta t_{n-1}/\Delta t_n)$. See Gartling (1986).

Example `Predictor-Corrector Tolerance = 0.001`

12.6 PREDICTOR FIELDS

Syntax `PREDICTOR FIELDS = [NOT]` *`STRING`* `[`*`STRING`*`...]`

Parent Block(s) `ARIA REGION`

Description Specifies which fields to examine or ignore in the algorithm for adaptive time step selection. Fields that are *not* predictor fields will not be predicted solutions for the first two time steps. This is important for fields like pressure and electrostatic potential which may exhibit large jumps in their solutions due to the absence of time dependent terms in their governing equations.

NOTE: Unlike other time control commands, this command is specified in the `ARIA REGION` block of the input file.

Details Explicitly list field names to include in the adaptive time step selection algorithm, if it's active. By default all fields are included. All field names following the optional "NOT" keyword will be excluded from the selection algorithm. The selected fields are those that contribute to the d_{n+1} norm discussed above, in `TIME TRUNCATION ERROR`. This command line can be provided multiple times with cumulative results.

This same set of fields will have their solution predicted for the first two time steps; all fields have their solution predicted for all subsequent time steps. This is done because equations that do not have a time derivative in them may experience a large jump in solution values between the initial conditions and the first solutions; predicting the next solution based on this large jump may adversely affect convergence of the nonlinear solver.

Example

```
PREDICTOR FIELDS = SPECIES_2
```

Example

```
PREDICTOR FIELDS = VELOCITY TEMPERATURE
```

Example

```
PREDICTOR FIELDS = NOT PRESSURE
```

Chapter 13

Nonlinear Solution Specifications

13.1 NONLINEAR SOLUTION STRATEGY

Syntax `NONLINEAR SOLUTION STRATEGY =` *`STRING`*

Description Specifies the nonlinear solution strategy.

Details Nonlinear solution strategy must be NEWTON or PICARD. Default value = NEWTON.

Parent Block(s) `ARIA_REGION`

13.2 NONLINEAR CORRECTION TOLERANCE

Syntax `NONLINEAR CORRECTION TOLERANCE =` *`REAL`*

Description Convergence tolerance of nonlinear correction norm for the solution iteration.

Details Satisfaction of this criterion is sufficient for the iteration to be considered successfully completed. Default value = 1.0e-6.

Parent Block(s) `ARIA_REGION`

13.3 NONLINEAR RESIDUAL TOLERANCE

Syntax `NONLINEAR RESIDUAL TOLERANCE =` *`REAL`*

Description Convergence tolerance of nonlinear residual for the solution iteration.

Details Satisfaction of this criterion is sufficient for the iteration to be considered successfully completed. Default value = 1.0e-6.

Parent Block(s) ARIA_REGION

13.4 NONLINEAR RESIDUAL RATIO TOLERANCE

Syntax	`NONLINEAR RESIDUAL RATIO TOLERANCE =` *REAL*
Description	Convergence tolerance of ratio of the nonlinear residual to the initial nonlinear residual for the solution iteration.
Details	Satisfaction of this criterion is sufficient for the iteration to be considered successfully completed. Default value = 0.0
Parent Block(s)	ARIA_REGION

13.5 NONLINEAR RELAXATION FACTOR

Syntax	`NONLINEAR RELAXATION FACTOR =` *REAL*
Description	Weighting factor for fraction of a new nonlinear solution that will be applied to the linear solution update.
Details	Weighting factor lies in the range 0.0 to 1.0. Default value = 1.0.
Parent Block(s)	ARIA_REGION

13.6 MAXIMUM NONLINEAR ITERATIONS

Syntax	`MAXIMUM NONLINEAR ITERATIONS =` *INT*
Description	Number of allowable nonlinear iterations in one linear solution step.
Details	The solution step will terminate when the number of nonlinear iterations are exceeded. Default value = 20.
Parent Block(s)	ARIA_REGION

13.7 MINIMUM NONLINEAR ITERATIONS

Syntax `MINIMUM NONLINEAR ITERATIONS =` *`INT`*

Description Minimum number of nonlinear iterations required.

Details The nonlinear solver will continue iterating until this minimum number of iterations are performed, including solving the nonlinear matrix system. The default value is one (1) but setting this to zero can be useful for some situations.

Parent Block(s) `ARIA_REGION`

13.8 ACCEPT SOLUTION AFTER MAXIMUM NONLINEAR ITERATIONS

Syntax `ACCEPT SOLUTION AFTER MAXIMUM NONLINEAR ITERATIONS =` *`BOOL`*

Description Determines whether reaching the maximum number of nonlinear iterations is a "success" criterion.

Details Determines whether reaching the maximum number of nonlinear iterations is a "success" criterion. By default, this is false. By the way, valid values of *`BOOL`* are `TRUE` and `FALSE`, case insensitive.

Parent Block(s) `ARIA_REGION`

13.9 FILTER NONLINEAR SOLUTION

Syntax `FILTER NONLINEAR SOLUTION FOR` *`DOF`* `FILTER_TYPE =` *`REAL`*$_1$ *`REAL`*$_2$

Description Restrict the value of a solution variable used in the nonlinear solver iterations of a solution step.

Details The nonlinear solution can be restricted to a range with upper and lower limits using the FILTER_TYPE = RANGE while supplying two bounding values. To set an upper or lower bound use FILTER_TYPE = MAXIMUM or FILTER_TYPE = MINIMUM while supplying a single bounding value.

Parent Block(s) `ARIA_REGION`

Chapter 14

Writing User Plugins

14.1 About Plugins

Users are free to extend Aria's library of material models, constitutive equations, boundary conditions, distinguishing conditions and source terms through the use of *plugins*. In order to do this, a user writes the C++ code to implement an Expression class. Before delving into Aria plugins it's worth reading section 23.1 and scanning section 23.4 for an introduction to Aria's Expression system.

When a user supplies their own plugin it becomes a first-class piece of Aria. Plugins have no performance penalty over Aria's built-in functionality and plugins have no added restrictions over built-in Expression regarding what can and can't be done. In fact, taking a user plugin and adding it to Aria proper (so that it becomes "owned" by Aria) is a piece of cake.

It's worth re-stating here that a lot of Aria's algorithms make use of sensitivities, i.e., Aria often needs to know the derivative of your model with respect to all of the unknowns in the problem. There are three ways to supply the sensitivities in aria: write them by hand, use a numerical finite-difference function, or use (forward) automatic differentiation (FAD). The FAD method is ideal for plugins because the sensitivities are are analytical and exact though there may be a small performance degradation. However, if only a few models use FAD there may be no measurable performance hit. For that reason, we've designed our plugin system to use FAD by default. If you have different needs, contact the developers for help.

14.2 Compiling and Using Plugins

Let's say that we have plugin C++ code for our very own density model in a file named `My_Density.C`. In order to use this plugin we'll first need to build it into Aria and to do that we'll need a project where we can build Aria. Here are the basic steps:

```
% cd ~/projects/

% create_project -s SierraVOTD plugin_project

% cd plugin_project

% checkout --deps aria
```

Now, currently there's a restriction that your plugin has to be stored in the `aria/` subdirectory of your project. We'll fix that eventually but in the mean time we need to put our plugin code there, e.g.,

```
% mkdir aria

% cp /some/path/My_Density.C aria

% build PLUGINS=My_Density.C
```

To compile-in our plugin we use a Makefile variable called `PLUGINS`. We can place this right on the SNTools' `build` command line like this:

```
% build PLUGINS=My_Density.C aria
```

Naturally, other options to `build` can be added to the command line.

Running the *pluginified* Aria executable can be done as normal using the SNTools' `sierra` tool. If your input files are somewhere inside the project where you built your plugin then you can run the `sierra` tool as normal. If your input files are somewhere else, you can use the `-x` ***`project-path`*** option to `sierra` to tell it to use your project. Continuing our example,

```
% cd ~/some/other/path

% sierra aria -i use_my_density.i -x ~/projects/plugin_project
```

14.3 An Important Note About Model Names

In order to avoid name clashes, model names always end with the generic name of the quantity they provide. So, density model names always end in `_DENSITY`, viscosity model names end in `_VISCOSITY`, etc. For material models, it's easy to know what that ending is because it's the same as the left-hand side of the "=" sign in the material input block (with spaces replaced by an underscore "_"). For source terms, it's always the name of the equation plus `_SOURCE`, e.g., `_ENERGY_SOURCE`. Clearly, it's important to keep this in mind when naming and referring to your plugin model or else Aria won't be able to find it properly.

In this example, we'll name our density `MY_MODEL_DENSITY` (recall, it must end in `_DENSITY`) so in the input file we'll have to refer to it as `MY_MODEL` since the ending is automatically added.

14.4 The Input File

Plugins can be referenced in the input file similiar to the way Aria's built-in Expressions are referenced. The name of the model is just whatever your plugin is named without the endign, e.g., `MY_MODEL`,

```
Begin Aria Material Kryptonite
   Density = My_Model a=1.0 b=-0.01
   ...
End
```

It's important to note that the plugin name must not conflict with model names used internally by Aria (the outcome would be, at least to users, ambiguous). Aria will verify this and produce and error if there's a name clash.

14.5 Example Plugin Code: My_Density.C

In this section we'll write the code to supply a density function which is a cubic polynomial in temperature (T),

$$\rho = a + bT + cT^2 + dT^3.$$

The complete source code for this plugin is available online at http://aria.sandia.gov/My_Density.C.

Normally, writing C++ code requires using a header (`.h`) file and an implementation (`.C`) file but since no other code needs to see our class definition, we can skip the header file and place the definition right in the .C file.

The first thing we need to do is include a header file which will give us all we need to write an Expression. Since our plugin will live in isolation, we'll also add a `using` declaration to make life easier on ourselves; without it we'd have to declare the namespaces or prepend `sierra::Aria::` to lots of data types. So far, we have this:

```
#include <Aria_Plugin_Expression.h>
using namespace sierra::Aria;
```

Next we need the basic declaration of our class. This tells the compiler which methods and data members we want to have. Here's ours:

```
// Class definition -- could go in a header file.
class My_Density : public FAD_Expression
{
public:
  My_Density(Expression_Mechanics * const mechanics,
             const sierra::Identifier & expr_model_name,
             const Subdomain_Tag & subdomain_tag,
             const Int & subindex,
             const Phase_Label & phase_label,
             const sierra::String & params);
  virtual ~My_Density() {}
  virtual void compute_FAD_values();
private:
  Real a;
  Real b;
  Real c;
  Real d;

  FAD_MDArray temperature;
};
```

This is mostly boiler-plate code. Here we declare a constructor and an empty destructor and the method needed for computing our density's values. We also add some private data to store our polynomial coefficients.

Now it's time to write some code. First, we write the constructor which tells Aria, in a generic sense, what we provide (density), what we depend on (temperature) and what parameters we require from the user (a, b, c and d).

```
  My_Density::My_Density(Expression_Mechanics * const mechanics,
                         const sierra::Identifier & expr_model_name,
                         const Subdomain_Tag & subdomain_tag,
                         const Int & subindex,
                         const Phase_Label & phase_label,
                         const sierra::String & params) :
  FAD_Expression(mechanics,subdomain_tag,subindex,phase_label,params)
{
  my_tensor_order          = 0;
  my_expression_tag        = Expression_Tag(DENSITY_EXPR,NO_OP,subindex,phase_label);
  my_expression_model_name = expr_model_name;

  // List the expressions that are required for this model.
  const Expression_Tag temperature_tag(TEMPERATURE_EXPR,NO_OP,subindex,phase_label);

  add_prereq(temperature_tag,temperature);

  // Get my model parameters.
  a = b = c = d = 0.0;
  get_optional_param("A",a);
  get_optional_param("B",b);
  get_optional_param("C",c);
  get_optional_param("D",d);

  if(a == 0.0 && b == 0.0 && c == 0.0 && d == 0.0)
    {
      throw sierra::RuntimeUserError() << "ERROR: All MY_MODEL_DENSITY parameters are zero.";
    }

  // Make myself known to the manager.
  register_myself();
}
```

The `my_tensor_order` tells Aria what kind of field your Expresion creates, viz. scalar, vector or tensor. If left unspecified, the default type is scalar (`my_tensor_order == 0`). Your expression also has a variable called `my_tensor_dimension` which is dimensionality of each tensor order. By default, `my_tensor_dimension` is set to the physical dimension of the problem, i.e., 2 for 2D and 3 for 3D. This variable is also set for scalar fields. Combined, these two variables define the number of values required to fully specify your Expression at a point: `pow(my_tensor_dimension,my_tensor_order)`. These also define the expected signature of the `values` data used below.

Next, we write the code that implements our density function. **Important note:** the `FAD_values` arrays are always initialized to zero before this method is called.

```
void My_Density::compute_FAD_values()
{
  for(Int point=0; point < num_points; ++point)
    {
      const FAD_Type & T = temperature(point);
      FAD_values(point)  = a + T*(b + T*(c + T*(d)));
    }
}
```

Note that since our Expression is a scalar in this example, the signature of `FAD_values` is `FAD_values(point)`. If our result was a vector or tensor, the signature would be `FAD_values(point,r)` and `FAD_values(point,r,s)`, respectively, where `0` $\leq$ `r,s < my_tensor_dimension`.

The last thing we need to do is the actual plugin step. This one line of code,

```
ExprPluginFactory<My_Density> my_density_creator("MY_MODEL_DENSITY");
```

takes care of making your Expression known to Aria. The quoted string is the string that is used in your input file (except for the ending part, see section 14.3. If this string has any spaces in it, Aria will automatically replace them with an underscore.

For completeness the whole plugin code is given here.

```
#include <Aria_Plugin_Expression.h>

using namespace sierra::Aria;

// Class definition -- could go in a header file.
class My_Density : public FAD_Expression
{
public:
  My_Density(Expression_Mechanics * const mechanics,
             const sierra::Identifier & expr_model_name,
             const Subdomain_Tag & subdomain_tag,
             const Int & subindex,
             const Phase_Label & phase_label,
             const sierra::String & params);
  virtual ~My_Density() {}
  virtual void compute_FAD_values();
private:
  Real a;
  Real b;
  Real c;
  Real d;

  FAD_MDArray temperature;
};


My_Density::My_Density(Expression_Mechanics * const mechanics,
                       const sierra::Identifier & expr_model_name,
                       const Subdomain_Tag & subdomain_tag,
                       const Int & subindex,
                       const Phase_Label & phase_label,
                       const sierra::String & params) :
  FAD_Expression(mechanics,subdomain_tag,subindex,phase_label,params)
{
  my_tensor_order          = 0;
  my_expression_tag        = Expression_Tag(DENSITY_EXPR,NO_OP,subindex,phase_label);
  my_expression_model_name = expr_model_name;

  // List the expressions that are required for this model.
  const Expression_Tag temperature_tag(TEMPERATURE_EXPR,NO_OP,subindex,phase_label);

  add_prereq(temperature_tag,temperature);
  // Get my model parameters.
  a = b = c = d = 0.0;
  get_optional_param("A",a);
  get_optional_param("B",b);
  get_optional_param("C",c);
  get_optional_param("D",d);

  if(a == 0.0 && b == 0.0 && c == 0.0 && d == 0.0)
    {
      throw sierra::RuntimeUserError() << "ERROR: All MY_MODEL_DENSITY parameters are zero.";
    }

  // Make myself known to the manager.
  register_myself();
}

void My_Density::compute_FAD_values()
{
  for(Int point=0; point < num_points; ++point)
    {
      const FAD_Type & T = temperature(point);
      FAD_values(point)  = a + T*(b + T*(c + T*(d)));
    }
}

ExprPluginFactory<My_Density> my_density_creator("MY_MODEL_DENSITY");
```

14.6 Testing Your Plugin

There are two good tests you can perform to test your plugin. The first is to run Aria in debug mode. To do this, add `-o dbg` to your `build` command line to build a debug executble. Then, add `-d` to your `sierra` command line to run the debug executable. In debug mode Aria will, among other things, perform bounds checking on your `FAD_MDArray` objects like `values(...)`, `dself(...)` and `sens(...)` in this example.

Secondly, if you hand-code your Newton sensitivities (not done in this example), you can test the coding of your sensitivities by adding `-O ‘‘-arialog sens_check’’` to your `sierra` command line. This will cause Aria to compare the computed sensitivity values with numerical approximations. The sensitivity checker will cause your code to run slower but it will work in either debug or optimized mode. Aria's sensitivity checker is designed to only report discrepencies that have a high probability of being true errors so it's possible that it may miss some small errors. However, reported errors are most probably real.

A lot of times the easiest way to debug your code is to just print information to the screen. Aria provides a facility to support this. To print information to the log file, use the `arialog` C++ output stream. For example,

```
for(Int point=0; point < num_points; ++point)
  {
    ...
    FAD_values(point) = ...
    arialog.m(LOG_PLUGIN) << "value(" << point << ") = " << value(point) << endl;
  }
```

Then, you can activate this output by adding the option `-O ‘‘-arialog plugin’’` to your `sierra` command line. If you leave off the `.m(LOG_PLUGIN)` part then it will always write your output to the log file. There's a performance penalty for having this code present (even if out don't turn the logging on) so you probably want to remove it once you're done debugging.

Chapter 15

NOX Nonlinear Solver Reference

See, also, the NOX parameters online reference.

15.1 NOX NONLINEAR EQUATION SOLVER

Begin NOX NONLINEAR EQUATION SOLVER *Nonlinear Solver Name*

SOLUTION METHOD { = | IS | ARE } *NoxAztecSolverMethods*

PRECONDITIONING METHOD { = | IS | ARE } *NoxAztecPreconditionerMethods*

NONLINEAR SOLVER METHOD { = | IS | ARE } *NoxSolverMethods*

NONLINEAR DIRECTION METHOD { = | IS | ARE } *NoxDirectionMethods*

NONLINEAR JACOBIAN OPERATOR { = | IS | ARE } *NoxJacobianOperators* [allowing diagonal correction]

NONLINEAR LINESEARCH METHOD { = | IS | ARE } *NoxLinesearchMethods*

NOX NONLINEAR PRECONDITIONING METHOD { = | IS | ARE } *NoxPreconditionerOperators* [allowing diagonal correction]

NONLINEAR PRECONDITIONING COMPUTE FREQUENCY { = | IS | ARE } *frequency*

RESET COUNTER EACH TIME STEP { = | IS | ARE } { false | true }

MAX AGE OF JACOBIAN { = | IS | ARE } *max_age*

FORCING TERM METHOD { = | IS | ARE } *NoxNewtonSolveOptions*

FORCING TERM INITIAL TOLERANCE { = | IS | ARE } *init_tol*

FORCING TERM MINIMUM TOLERANCE { = | IS | ARE } *min_tol*

FORCING TERM MAXIMUM TOLERANCE { = | IS | ARE } *max_tol*

TYPE 2 FORCING TERM ALPHA { = | IS | ARE } *alpha*

TYPE 2 FORCING TERM GAMMA { = | IS | ARE } *beta*

MAXIMUM ITERATIONS { = | IS | ARE } *max_iters*

RESIDUAL NORM TOLERANCE { = | IS | ARE } *tol*

RESTART ITERATIONS { = | IS | ARE } *restart_iters*

PRECONDITIONING STEPS { = | IS | ARE } *steps*

POLYNOMIAL ORDER { = | IS | ARE } *order*

MAXIMUM NONLINEAR ITERATIONS { = | IS | ARE } *max_iters*

NONLINEAR ABSOLUTE RESIDUAL NORM TOLERANCE { = | IS | ARE } *abs_res_tol*

NONLINEAR RELATIVE RESIDUAL NORM TOLERANCE { = | IS | ARE } *rel_res_tol*

ILL CONDITIONING THRESHOLD { = | IS | ARE } *threshold*

ILU OVERLAP { = | IS | ARE } *overlap*

ILU GRAPH FILL { = | IS | ARE } *fill*

ILUT FILL FACTOR { = | IS | ARE } *fill_factor*

ILUT DROP TOLERANCE { = | IS | ARE } *tolerance*

PRECONDITIONING PACKAGE { = | IS | ARE } *PreconditioningPackages* [using *Teuchos_Param_List_Name*]

MATRIX FREE FINITE DIFFERENCE METHOD { = | IS | ARE } *NoxDifferencingOptions*

FINITE DIFFERENCE METHOD { = | IS | ARE } *NoxDifferencingOptions*

PERTURBATION COEFFICIENT ALPHA { = | IS | ARE } *FDalpha*

PERTURBATION COEFFICIENT BETA { = | IS | ARE } *FDbeta*

MATRIX FREE PERTURBATION COEFFICIENT LAMBDA { = | IS | ARE } *MFlambda*

MATRIX FREE PERTURBATION COEFFICIENT VALUE { = | IS | ARE } *MFepsilon*

RESCUE BAD NEWTON SOLVE { = | IS | ARE } { false | true }

USE RCM REORDERING { = | IS | ARE } { false | true }

DISTINGUISH MATRIX FREE FILLS { = | IS | ARE } { false | true }

STAGNATION TEST TOLERANCE { = | IS | ARE } *stag_test_tol*

STAGNATION TEST STEPS { = | IS | ARE } *stag_test_steps*

LINEAR STAGNATION TEST TOLERANCE { = | IS | ARE } *lin_stag_test_tol*

LINEAR STAGNATION TEST STEPS { = | IS | ARE } *lin_stag_test_steps*

NONLINEAR ABSOLUTE UPDATE NORM TOLERANCE { = | IS | ARE } *abs_update_tol*

USER DEFINED CONVERGENCE { = | IS | ARE } { false | true }

MINIMUM NONLINEAR ITERATIONS { = | IS | ARE } *req_min_nonlin_iters*

LINEAR SOLVER OUTPUT FREQUENCY { = | IS | ARE } *lin_output_freq*

COLORING ALGORITHM { = | IS | ARE } *NoxColoringMethods*

COLORING REORDERING { = | IS | ARE } *NoxColoringReordering*

COLOR GRAPH USING DISTANCE1 { = | IS | ARE } { false | true }

DUMP JACOBIAN MATRIX to *MatrixOutputFormat*

DUMP RESIDUAL VECTOR to *MatrixOutputFormat*

DUMP INITIAL GUESS VECTOR to *MatrixOutputFormat*

DUMP LINEAR SOLUTION VECTOR to *MatrixOutputFormat*

OUTPUT NONLINEAR OBJECTS using *param_list_name*

SETUP NONLINEAR SOLVER using *setup_params_list_name*

NOX OUTPUT LEVEL { = | IS | ARE } *NoxOutputLevels*

NOX OUTPUT VALUE { = | IS | ARE } *intValue*

USE NOX OPERATOR DEBUGGER { = | IS | ARE } *Debugger_Name*

USE NOX LINEAR SYSTEM { = | IS | ARE } *Linear_System_Name*

USE NOX LINESEARCH { = | IS | ARE } *Linesearch_Name*

NONLINEAR ML COARSENING METHOD { = | IS | ARE } *ML_Nox_Coarsening_Schemes*

NONLINEAR ML IS LINEAR PRECONDITIONER { = | IS | ARE } *ML_Nox_Is_LinearPrec*

NONLINEAR ML IS MATRIXFREE { = | IS | ARE } *ML_Nox_Is_Matrixfree*

NONLINEAR ML FINITE DIFFERENCE FINE LEVEL { = | IS | ARE } *ML_Nox_FD_FineLevel*

NONLINEAR ML MAX NLEVEL { = | IS | ARE } *intValue*

NONLINEAR ML MAX COARSE SIZE { = | IS | ARE } *intValue*

NONLINEAR ML COARSENING RATIO OBJECTIVE { = | IS | ARE } *intValue*

NONLINEAR ML USE NLNCG LEVEL FINE { = | IS | ARE } *ML_Nox_Use_nlnCG*

NONLINEAR ML USE NLNCG LEVEL MED { = | IS | ARE } *ML_Nox_Use_nlnCG*

NONLINEAR ML USE NLNCG LEVEL COARSEST { = | IS | ARE } *ML_Nox_Use_nlnCG*

NONLINEAR ML USE BROYDEN UPDATE { = | IS | ARE } *ML_Nox_Use_Broyden*

NONLINEAR ML NUM ITERATIONS LINCG FINE { = | IS | ARE } *intValue*

NONLINEAR ML NUM ITERATIONS LINCG MED { = | IS | ARE } *intValue*

NONLINEAR ML NUM ITERATIONS LINCG COARSEST { = | IS | ARE } *intValue*

NONLINEAR ML PROBLEM DIMENSION { = | IS | ARE } *intValue*

NONLINEAR ML NUMBER PDES PER NODE { = | IS | ARE } *intValue*

NONLINEAR ML DIMENSION NULLSPACE { = | IS | ARE } *intValue*

NONLINEAR ML LINEAR SMOOTHER FINE { = | IS | ARE } *ML_Nox_Linear_Smoother*

NONLINEAR ML LINEAR SMOOTHER MED { = | IS | ARE } *ML_Nox_Linear_Smoother*

NONLINEAR ML LINEAR SMOOTHER COARSEST { = | IS | ARE } *ML_Nox_Linear_Smoother*

NONLINEAR ML LINEAR SMOOTHER SWEEPS FINE { = | IS | ARE } *intValue*

NONLINEAR ML LINEAR SMOOTHER SWEEPS MED { = | IS | ARE } *intValue*

NONLINEAR ML LINEAR SMOOTHER SWEEPS COARSEST { = | IS | ARE } *intValue*

NONLINEAR ML NONLINEAR SWEEPS PRE FINE { = | IS | ARE } *intValue*

NONLINEAR ML NONLINEAR SWEEPS PRE MED { = | IS | ARE } *intValue*

NONLINEAR ML NONLINEAR SWEEPS COARSEST { = | IS | ARE } *intValue*

NONLINEAR ML NONLINEAR SWEEPS POST MED { = | IS | ARE } *intValue*

NONLINEAR ML NONLINEAR SWEEPS POST FINE { = | IS | ARE } *intValue*

NONLINEAR ML MAX NUMBER CYCLES { = | IS | ARE } *intValue*

NONLINEAR ML FINITE DIFFERENCE CENTERED { = | IS | ARE } *ML_Nox_Fd_Centered*

NONLINEAR ML FINITE DIFFERENCE ALPHA { = | IS | ARE } *ml_fd_alpha*

NONLINEAR ML FINITE DIFFERENCE BETA { = | IS | ARE } *ml_fd_beta*

NONLINEAR ML PRINT LEVEL { = | IS | ARE } *intValue*

NONLINEAR ML RECALCULATION OFFSET { = | IS | ARE } *intValue* Begin TEUCHOS PARAMETER BLOCK *Teuchos Parameter Block Name*

End
Begin NOX DEBUGGER BLOCK *Nox Debugger Block Name*

End
Begin NOX EPETRA OPERATOR *Nox Operator Block Name*

End
Begin NOX AZTECOO LINEAR SYSTEM *Nox AztecOO Linear System Block Name*

```
    End
    Begin NOX LINESEARCH BLOCK Nox Linesearch Block Name



    End

End
```

Details A set of solver parameters for the NOX nonlinear equation solver.

15.1.1 SOLUTION METHOD

Syntax `SOLUTION METHOD { = | IS | ARE }` *NoxAztecSolverMethods*

NoxAztecSolverMethods : *no description* { `cg` | `cgs` | `bicgstab` | `gmres` | `tfqmr` | `lu` }

Details Selection of the AztecOO linear solution method.

Enums `NoxAztecSolverMethods`

- `cg` - *no description*
- `cgs` - *no description*
- `bicgstab` - *no description*
- `gmres` - *no description*
- `tfqmr` - *no description*
- `lu` - *no description*

15.1.2 PRECONDITIONING METHOD

Syntax `PRECONDITIONING METHOD { = | IS | ARE }` *NoxAztecPreconditionerMethods*

NoxAztecPreconditionerMethods : *no description* { `none` | `jacobi` | `neumann` | `least-squares` | `dd-ilut` | `dd-ilu` | `user supplied operator` }

Details Selection of the AztecOO preconditioning methods supported by the NOX nonlinear solver.

Enums `NoxAztecPreconditionerMethods`

`none` - *no description*

`jacobi` - *no description*

`neumann` - *no description*

`least-squares` - *no description*

`dd-ilut` - *no description*

`dd-ilu` - *no description*

`user supplied operator` - *no description*

15.1.3 NONLINEAR SOLVER METHOD

Syntax `NONLINEAR SOLVER METHOD` { `=` | `IS` | `ARE` } *NoxSolverMethods*

NoxSolverMethods : *no description* { `line search based` }

Details Selection of the solution method for nonlinear solver.

Enums `NoxSolverMethods`

`line search based` - *no description*

15.1.4 NONLINEAR DIRECTION METHOD

Syntax `NONLINEAR DIRECTION METHOD` { `=` | `IS` | `ARE` } *NoxDirectionMethods*

NoxDirectionMethods : *no description* { `newton` | `nonlinear cg` | `modified newton` | `semi-implicit` }

Details Selection of the direction method to use with a *line search based* nonlinear solution method.

Enums `NoxDirectionMethods`

`newton` - *no description*

`nonlinear cg` - *no description*

`modified newton` - *no description*

`semi-implicit` - *no description*

15.1.5 NONLINEAR JACOBIAN OPERATOR

Syntax `NONLINEAR JACOBIAN OPERATOR` { `=` | `IS` | `ARE` } *NoxJacobianOperators* `[ allowing diagonal correction ]`

NoxJacobianOperators : *no description* { `matrix free` | `finite difference` | `finite coloring` | `user supplied matrix` | `user supplied operator` }

Details Selection of the jacobian operator type

Enums `NoxJacobianOperators`

`matrix free` - *no description*

`finite difference` - *no description*

`finite coloring` - *no description*

`user supplied matrix` - *no description*

`user supplied operator` - *no description*

15.1.6 NONLINEAR LINESEARCH METHOD

Syntax `NONLINEAR LINESEARCH METHOD` { `=` | `IS` | `ARE` } *NoxLinesearchMethods*

NoxLinesearchMethods : *no description* { `full step` | `polynomial` | `quadratic` | `more thunte` | `nonlinear cg` }

Details Selection of the linesearch method for nonlinear solver.

Enums `NoxLinesearchMethods`

`full step` - *no description*

`polynomial` - *no description*

`quadratic` - *no description*

`more thunte` - *no description*

`nonlinear cg` - *no description*

15.1.7 NOX NONLINEAR PRECONDITIONING METHOD

Syntax `NOX NONLINEAR PRECONDITIONING METHOD` { `=` | `IS` | `ARE` } *NoxPreconditioner-Operators* `[ allowing diagonal correction ]`

NoxPreconditionerOperators : *no description* { `none` | `use jacobian` | `finite difference` | `finite coloring` | `user supplied matrix` | `user supplied operator` | `ml` }

Details Selection of NOX nonlinear solver's preconditioning method.

Enums `NoxPreconditionerOperators`

- `none` - *no description*
- `use jacobian` - *no description*
- `finite difference` - *no description*
- `finite coloring` - *no description*
- `user supplied matrix` - *no description*
- `user supplied operator` - *no description*
- `ml` - *no description*

15.1.8 NONLINEAR PRECONDITIONING COMPUTE FREQUENCY

Syntax `NONLINEAR PRECONDITIONING COMPUTE FREQUENCY` { `=` | `IS` | `ARE` } *frequency*

frequency : *no description* (I)

Details Recompute frequency of the nonlinear preconditioner operator.

15.1.9 RESET COUNTER EACH TIME STEP

Details *** This option is currently inactive. ***

15.1.10 MAX AGE OF JACOBIAN

Syntax `MAX AGE OF JACOBIAN` { `=` | `IS` | `ARE` } *max_age*

max_age : *no description* (I)

Details Integer number of nonlinear iterations between recomputations of the Jacobian

15.1.11 FORCING TERM METHOD

Syntax `FORCING TERM METHOD` { `=` | `IS` | `ARE` } *NoxNewtonSolveOptions*

NoxNewtonSolveOptions : *no description* { `Constant` | `Type 1` | `Type 2` }

Details Specification of linear solver adaptive forcing term method.

Enums `NoxNewtonSolveOptions`

`Constant` - *no description*
`Type 1` - *no description*
`Type 2` - *no description*

15.1.12 FORCING TERM INITIAL TOLERANCE

Syntax `FORCING TERM INITIAL TOLERANCE { = | IS | ARE }` *init_tol*

init_tol : *no description* (R)

Details Linear solver adaptive forcing term initial linear solve tolerance.

15.1.13 FORCING TERM MINIMUM TOLERANCE

Syntax `FORCING TERM MINIMUM TOLERANCE { = | IS | ARE }` *min_tol*

min_tol : *no description* (R)

Details Linear solver adaptive forcing term minimum linear solve tolerance.

15.1.14 FORCING TERM MAXIMUM TOLERANCE

Syntax `FORCING TERM MAXIMUM TOLERANCE { = | IS | ARE }` *max_tol*

max_tol : *no description* (R)

Details Linear solver adaptive forcing term maximum linear solve tolerance.

15.1.15 TYPE 2 FORCING TERM ALPHA

Syntax `TYPE 2 FORCING TERM ALPHA { = | IS | ARE }` *alpha*

alpha : *no description* (R)

Details Linear solver adaptive forcing term Type 2 value for α

15.1.16 TYPE 2 FORCING TERM GAMMA

Syntax `TYPE 2 FORCING TERM GAMMA { = | IS | ARE }` *beta*

beta : *no description* (R)

Details Linear solver adaptive forcing term Type 2 value for γ

15.1.17 MAXIMUM ITERATIONS

Syntax `MAXIMUM ITERATIONS` { `=` | `IS` | `ARE` } *max_iters*

max_iters : *no description* (I)

Details Maximum number of solution method iterations.

15.1.18 RESIDUAL NORM TOLERANCE

Syntax `RESIDUAL NORM TOLERANCE` { `=` | `IS` | `ARE` } *tol*

tol : *no description* (R)

Details Iterative solution method residual convergence tolerance.

15.1.19 RESTART ITERATIONS

Syntax `RESTART ITERATIONS` { `=` | `IS` | `ARE` } *restart_iters*

restart_iters : *no description* (I)

Details Number of iterations between GMRES restarts.

15.1.20 PRECONDITIONING STEPS

Syntax `PRECONDITIONING STEPS` { `=` | `IS` | `ARE` } *steps*

steps : *no description* (I)

Details Number of Jacobi, Gauss-Seidel, or other preconditioning methods' applications per iteration.

15.1.21 POLYNOMIAL ORDER

Syntax `POLYNOMIAL ORDER` { `=` | `IS` | `ARE` } *order*

order : *no description* (I)

Details Polynomial order of preconditioning method.

15.1.22 MAXIMUM NONLINEAR ITERATIONS

Syntax `MAXIMUM NONLINEAR ITERATIONS` { `=` | `IS` | `ARE` } *max_iters*

max_iters : *no description* (I)

Details Maximum number of nonlinear solver iterations.

15.1.23 NONLINEAR ABSOLUTE RESIDUAL NORM TOLERANCE

Syntax `NONLINEAR ABSOLUTE RESIDUAL NORM TOLERANCE` { `=` | `IS` | `ARE` } *abs_res_tol*

abs_res_tol : *no description* (R)

Details Nonlinear absolute residual norm convergence tolerance.

15.1.24 NONLINEAR RELATIVE RESIDUAL NORM TOLERANCE

Syntax `NONLINEAR RELATIVE RESIDUAL NORM TOLERANCE` { `=` | `IS` | `ARE` } *rel_res_tol*

rel_res_tol : *no description* (R)

Details Nonlinear relative residual convergence tolerance.

15.1.25 ILL CONDITIONING THRESHOLD

Syntax `ILL CONDITIONING THRESHOLD` { `=` | `IS` | `ARE` } *threshold*

threshold : *no description* (R)

Details Ill-conditioning threshold for linear solver

15.1.26 ILU OVERLAP

Syntax `ILU OVERLAP` { `=` | `IS` | `ARE` } *overlap*

overlap : *no description* (I)

Details Overlap parameter for incomplete factorizations.

15.1.27 ILU GRAPH FILL

Syntax `ILU GRAPH FILL` { `= | IS | ARE` } *fill*

fill : *no description* (I)

Details Graph fill factor for incomplete factorizations.

15.1.28 ILUT FILL FACTOR

Syntax `ILUT FILL FACTOR` { `= | IS | ARE` } *fill_factor*

fill_factor : *no description* (R)

Details Fill factor for incomplete threshold factorizations.

15.1.29 ILUT DROP TOLERANCE

Syntax `ILUT DROP TOLERANCE` { `= | IS | ARE` } *tolerance*

tolerance : *no description* (R)

Details Drop tolerance for incomplete threshold factorizations.

15.1.30 PRECONDITIONING PACKAGE

Syntax `PRECONDITIONING PACKAGE` { `= | IS | ARE` } *PreconditioningPackages* `[ using` *Teuchos_Param_List_Name* `]`

PreconditioningPackages : *no description* { `aztecoo | ifpack | multilevel` }

Teuchos_Param_List_Name : *no description* (C)

Details Specify which package to use for preconditioning.

Enums `PreconditioningPackages`

- `aztecoo` - *no description*
- `ifpack` - *no description*
- `multilevel` - *no description*

15.1.31 MATRIX FREE FINITE DIFFERENCE METHOD

Syntax `MATRIX FREE FINITE DIFFERENCE METHOD` { `=` | `IS` | `ARE` } *NoxDifferencingOptions*

NoxDifferencingOptions : *no description* { `forward difference` | `backward difference` | `centered difference` }

Details Finite differencing method used for matrix-free.

Enums `NoxDifferencingOptions`

- `forward difference` - *no description*
- `backward difference` - *no description*
- `centered difference` - *no description*

15.1.32 FINITE DIFFERENCE METHOD

Syntax `FINITE DIFFERENCE METHOD` { `=` | `IS` | `ARE` } *NoxDifferencingOptions*

NoxDifferencingOptions : *no description* { `forward difference` | `backward difference` | `centered difference` }

Details Finite differencing method.

Enums `NoxDifferencingOptions`

- `forward difference` - *no description*
- `backward difference` - *no description*
- `centered difference` - *no description*

15.1.33 PERTURBATION COEFFICIENT ALPHA

Syntax `PERTURBATION COEFFICIENT ALPHA` { `=` | `IS` | `ARE` } *FDalpha*

FDalpha : *no description* (R)

Details Parameter, α, used in finite differencing perturbation value calculation.

15.1.34 PERTURBATION COEFFICIENT BETA

Syntax `PERTURBATION COEFFICIENT BETA` { `=` | `IS` | `ARE` } *FDbeta*

FDbeta : *no description* (R)

Details Parameter, β, used in finite differencing perturbation value calculation.

15.1.35 MATRIX FREE PERTURBATION COEFFICIENT LAMBDA

Syntax `MATRIX FREE PERTURBATION COEFFICIENT LAMBDA` { `=` | `IS` | `ARE` } *MFlambda*

MFlambda : *no description* (R)

Details Parameter, lambda, used to compute the perturbation size used in matrix-free residual evaluations.

15.1.36 MATRIX FREE PERTURBATION COEFFICIENT VALUE

Syntax `MATRIX FREE PERTURBATION COEFFICIENT VALUE` { `=` | `IS` | `ARE` } *MFepsilon*

MFepsilon : *no description* (R)

Details Direct specification for pertuebation size used in matrix-free residual evaluations.

15.1.37 RESCUE BAD NEWTON SOLVE

Details Flag to specify if an unconverged linear solve solution should be accepted or flagged as failed.

15.1.38 USE RCM REORDERING

Details Flag to specify whether or not to reorder the matrix via the Reverse Cuthill-McGee algorithm.

15.1.39 DISTINGUISH MATRIX FREE FILLS

Details Flag to specify whether or not to treat Matrix-Free residual fills different than fills used as the right-hand-side in Newton methods.

15.1.40 STAGNATION TEST TOLERANCE

Syntax `STAGNATION TEST TOLERANCE` { `=` | `IS` | `ARE` } *stag_test_tol*

stag_test_tol : *no description* (R)

Details Maximum allowed ratio of nonlinear residuals used to define stagnation event.

15.1.41 STAGNATION TEST STEPS

Syntax `STAGNATION TEST STEPS { = | IS | ARE }` *stag_test_steps*

stag_test_steps : *no description* (I)

Details Maximum number of consecutive nonlinear iterations that the residual ratio is allowed to be above its maximum value.

15.1.42 LINEAR STAGNATION TEST TOLERANCE

Syntax `LINEAR STAGNATION TEST TOLERANCE { = | IS | ARE }` *lin_stag_test_tol*

lin_stag_test_tol : *no description* (R)

Details Maximum allowed ratio of linear residuals used to define stagnation event.

15.1.43 LINEAR STAGNATION TEST STEPS

Syntax `LINEAR STAGNATION TEST STEPS { = | IS | ARE }` *lin_stag_test_steps*

lin_stag_test_steps : *no description* (I)

Details Maximum number of consecutive linear iterations that the linear residual ratio is allowed to be above its maximum value.

15.1.44 NONLINEAR ABSOLUTE UPDATE NORM TOLERANCE

Syntax `NONLINEAR ABSOLUTE UPDATE NORM TOLERANCE { = | IS | ARE }` *abs_update_tol*

abs_update_tol : *no description* (R)

Details Absolute update norm convergence tolerance.

15.1.45 USER DEFINED CONVERGENCE

Details Flag to specify whether or not to allow the application to defined convergence.

15.1.46 MINIMUM NONLINEAR ITERATIONS

Syntax `MINIMUM NONLINEAR ITERATIONS { = | IS | ARE }` *req_min_nonlin_iters*

req_min_nonlin_iters : *no description* (I)

Details Specifies a required minimum number of nonlinear iterations.

15.1.47 LINEAR SOLVER OUTPUT FREQUENCY

Syntax `LINEAR SOLVER OUTPUT FREQUENCY` { `=` | `IS` | `ARE` } *lin_output_freq*

lin_output_freq : *no description* (I)

Details Output frequency for iterative linear solvers.

15.1.48 COLORING ALGORITHM

Syntax `COLORING ALGORITHM` { `=` | `IS` | `ARE` } *NoxColoringMethods*

NoxColoringMethods : *no description* { `greedy` | `luby` }

Details Specify which coloring algorithm to use.

Enums `NoxColoringMethods`

`greedy` - *no description*
`luby` - *no description*

15.1.49 COLORING REORDERING

Syntax `COLORING REORDERING` { `=` | `IS` | `ARE` } *NoxColoringReordering*

NoxColoringReordering : *no description* { `largest first` | `smallest first` | `random` }

Details Specify how to reorder during coloring.

Enums `NoxColoringReordering`

`largest first` - *no description*
`smallest first` - *no description*
`random` - *no description*

15.1.50 COLOR GRAPH USING DISTANCE1

Details Flag indicating use of distance1 coloring of matrix graph.

15.1.51 DUMP JACOBIAN MATRIX

Syntax `DUMP JACOBIAN MATRIX to` *MatrixOutputFormat*

`MatrixOutputFormat` : *`no description`* { `ascii | matrix market` }

Details Debugging flag to allow the user to print the Jacobian matrix and then exit.

Enums `MatrixOutputFormat`

`ascii` - *no description*

`matrix market` - *no description*

15.1.52 DUMP RESIDUAL VECTOR

Syntax `DUMP RESIDUAL VECTOR to` *MatrixOutputFormat*

`MatrixOutputFormat` : *`no description`* { `ascii | matrix market` }

Details Debugging flag to allow the user to print the Residual vector and then exit.

Enums `MatrixOutputFormat`

`ascii` - *no description*

`matrix market` - *no description*

15.1.53 DUMP INITIAL GUESS VECTOR

Syntax `DUMP INITIAL GUESS VECTOR to` *MatrixOutputFormat*

`MatrixOutputFormat` : *`no description`* { `ascii | matrix market` }

Details Debugging flag to allow the user to print the initial guess for the nonlinear solution and then exit.

Enums `MatrixOutputFormat`

`ascii` - *no description*

`matrix market` - *no description*

15.1.54 DUMP LINEAR SOLUTION VECTOR

Syntax `DUMP LINEAR SOLUTION VECTOR to` *MatrixOutputFormat*

`MatrixOutputFormat` : *`no description`* { `ascii | matrix market` }

Details Debugging flag to allow the user to first compute and then dump the vector obtained from doing a linear solve at the current nonlinear iteration and then exit.

Enums `MatrixOutputFormat`

> `ascii` - *no description*
> `matrix market` - *no description*

15.1.55 OUTPUT NONLINEAR OBJECTS

Syntax `OUTPUT NONLINEAR OBJECTS using` *param_list_name*

param_list_name : *no description* (C)

Details Specify nonlinear solver object output using a named teuchos parameter list.

15.1.56 SETUP NONLINEAR SOLVER

Syntax `SETUP NONLINEAR SOLVER using` *setup_params_list_name*

setup_params_list_name : *no description* (C)

Details Specify options to use when setting up the nonlinear solver.

15.1.57 NOX OUTPUT LEVEL

Syntax `NOX OUTPUT LEVEL` { `=` | `IS` | `ARE` } *NoxOutputLevels*

NoxOutputLevels : *no description* { `low` | `medium` | `high` }

Details Level specification for amount of output from the NOX nonlinear solver

Enums `NoxOutputLevels`

> `low` - *no description*
> `medium` - *no description*
> `high` - *no description*

15.1.58 NOX OUTPUT VALUE

Syntax `NOX OUTPUT VALUE` { `=` | `IS` | `ARE` } *intValue*

intValue : *no description* (I)

Details Integer specification for amount of output from the NOX nonlinear solver. Values range from 0-256.

15.1.59 USE NOX OPERATOR DEBUGGER

Syntax `USE NOX OPERATOR DEBUGGER { = | IS | ARE }` *Debugger_Name*

`Debugger_Name` `:` *`no description`* `(C)`

Details Specifies which NOX operator (e.g. Jacobian) debugger to use.

15.1.60 USE NOX LINEAR SYSTEM

Syntax `USE NOX LINEAR SYSTEM { = | IS | ARE }` *Linear_System_Name*

`Linear_System_Name` `:` *`no description`* `(C)`

Details Specifies which NOX Linear System to use.

15.1.61 USE NOX LINESEARCH

Syntax `USE NOX LINESEARCH { = | IS | ARE }` *Linesearch_Name*

`Linesearch_Name` `:` *`no description`* `(C)`

Details Specifies which NOX Linesearch to use.

15.1.62 NONLINEAR ML COARSENING METHOD

Details Choose ML coarsening method.

15.1.63 NONLINEAR ML IS LINEAR PRECONDITIONER

Details Choose ML to act as non-/linear preconditioner.

15.1.64 NONLINEAR ML IS MATRIXFREE

Details Choose ML to be matrixfree preconditioner.

15.1.65 NONLINEAR ML FINITE DIFFERENCE FINE LEVEL

Details Choose ML to construct fine level full Jacobian.

15.1.66 NONLINEAR ML MAX NLEVEL

Syntax `NONLINEAR ML MAX NLEVEL` { `=` | `IS` | `ARE` } *intValue*

`intValue` `:` *`no description`* `(I)`

Details Integer maximum number of levels.

15.1.67 NONLINEAR ML MAX COARSE SIZE

Syntax `NONLINEAR ML MAX COARSE SIZE` { `=` | `IS` | `ARE` } *intValue*

`intValue` `:` *`no description`* `(I)`

Details Size of coarse grid where to stop coarsening further.

15.1.68 NONLINEAR ML COARSENING RATIO OBJECTIVE

Syntax `NONLINEAR ML COARSENING RATIO OBJECTIVE` { `=` | `IS` | `ARE` } *intValue*

`intValue` `:` *`no description`* `(I)`

Details Coarsening ratio objective.

15.1.69 NONLINEAR ML USE NLNCG LEVEL FINE

Details Choose ML to use nlnCG on fine level.

15.1.70 NONLINEAR ML USE NLNCG LEVEL MED

Details Choose ML to use nlnCG on all medium levels.

15.1.71 NONLINEAR ML USE NLNCG LEVEL COARSEST

Details Choose ML to use nlnCG on coarsest level.

15.1.72 NONLINEAR ML USE BROYDEN UPDATE

Details Choose ML to use a Broyden update on all levels that use Newton's method.

15.1.73 NONLINEAR ML NUM ITERATIONS LINCG FINE

Syntax `NONLINEAR ML NUM ITERATIONS LINCG FINE { = | IS | ARE }` *intValue*

intValue : *no description* (I)

Details Number of linear CG iterations in Newton step on level fine.

15.1.74 NONLINEAR ML NUM ITERATIONS LINCG MED

Syntax `NONLINEAR ML NUM ITERATIONS LINCG MED { = | IS | ARE }` *intValue*

intValue : *no description* (I)

Details Number of linear CG iterations in Newton step on level med.

15.1.75 NONLINEAR ML NUM ITERATIONS LINCG COARSEST

Syntax `NONLINEAR ML NUM ITERATIONS LINCG COARSEST { = | IS | ARE }` *intValue*

intValue : *no description* (I)

Details Number of linear CG iterations in Newton step on level coarsest.

15.1.76 NONLINEAR ML PROBLEM DIMENSION

Syntax `NONLINEAR ML PROBLEM DIMENSION { = | IS | ARE }` *intValue*

intValue : *no description* (I)

Details Dimension of the problem (3D, 2D, 1D).

15.1.77 NONLINEAR ML NUMBER PDES PER NODE

Syntax `NONLINEAR ML NUMBER PDES PER NODE { = | IS | ARE }` *intValue*

intValue : *no description* (I)

Details Number of PDEs (dofs) per node.

15.1.78 NONLINEAR ML DIMENSION NULLSPACE

Syntax `NONLINEAR ML DIMENSION NULLSPACE` { `=` | `IS` | `ARE` } *intValue*

intValue : *no description* (I)

Details Dimension of the Nullspace of the problem (6 in 3D, 3 in 2D, 1 in 1D).

15.1.79 NONLINEAR ML LINEAR SMOOTHER FINE

Details Choose ML linear smoother.

15.1.80 NONLINEAR ML LINEAR SMOOTHER MED

Details Choose ML linear smoother.

15.1.81 NONLINEAR ML LINEAR SMOOTHER COARSEST

Details Choose ML linear smoother.

15.1.82 NONLINEAR ML LINEAR SMOOTHER SWEEPS FINE

Syntax `NONLINEAR ML LINEAR SMOOTHER SWEEPS FINE` { `=` | `IS` | `ARE` } *intValue*

intValue : *no description* (I)

Details Number of sweeps of linear smoother.

15.1.83 NONLINEAR ML LINEAR SMOOTHER SWEEPS MED

Syntax `NONLINEAR ML LINEAR SMOOTHER SWEEPS MED` { `=` | `IS` | `ARE` } *intValue*

intValue : *no description* (I)

Details Number of sweeps of linear smoother.

15.1.84 NONLINEAR ML LINEAR SMOOTHER SWEEPS COARSEST

Syntax `NONLINEAR ML LINEAR SMOOTHER SWEEPS COARSEST` { `=` | `IS` | `ARE` } *intValue*

intValue : *no description* (I)

Details Number of sweeps of linear smoother.

15.1.85 NONLINEAR ML NONLINEAR SWEEPS PRE FINE

Syntax `NONLINEAR ML NONLINEAR SWEEPS PRE FINE` { `=` | `IS` | `ARE` } *intValue*

intValue : *no description* (I)

Details Number of presmooth sweeps of nonlinear smoother fine.

15.1.86 NONLINEAR ML NONLINEAR SWEEPS PRE MED

Syntax `NONLINEAR ML NONLINEAR SWEEPS PRE MED` { `=` | `IS` | `ARE` } *intValue*

intValue : *no description* (I)

Details Number of presmooth sweeps of nonlinear smoother med.

15.1.87 NONLINEAR ML NONLINEAR SWEEPS COARSEST

Syntax `NONLINEAR ML NONLINEAR SWEEPS COARSEST` { `=` | `IS` | `ARE` } *intValue*

intValue : *no description* (I)

Details Number of sweeps of nonlinear smoother coarsest.

15.1.88 NONLINEAR ML NONLINEAR SWEEPS POST MED

Syntax `NONLINEAR ML NONLINEAR SWEEPS POST MED` { `=` | `IS` | `ARE` } *intValue*

intValue : *no description* (I)

Details Number of postsmooth sweeps of nonlinear smoother med.

15.1.89 NONLINEAR ML NONLINEAR SWEEPS POST FINE

Syntax `NONLINEAR ML NONLINEAR SWEEPS POST FINE` { `=` | `IS` | `ARE` } *intValue*

intValue : *no description* (I)

Details Number of postsmooth sweeps of nonlinear smoother fine.

15.1.90 NONLINEAR ML MAX NUMBER CYCLES

Syntax `NONLINEAR ML MAX NUMBER CYCLES` { `=` | `IS` | `ARE` } *intValue*

intValue : *no description* (I)

Details max Number of cycles for nox_ml as a solver.

15.1.91 NONLINEAR ML FINITE DIFFERENCE CENTERED

Details Choose ML finite differencing method.

15.1.92 NONLINEAR ML FINITE DIFFERENCE ALPHA

Syntax `NONLINEAR ML FINITE DIFFERENCE ALPHA` { `=` | `IS` | `ARE` } *ml_fd_alpha*

ml_fd_alpha : *no description* (R)

Details Finite Differencing perturbation parameter alpha.

15.1.93 NONLINEAR ML FINITE DIFFERENCE BETA

Syntax `NONLINEAR ML FINITE DIFFERENCE BETA` { `=` | `IS` | `ARE` } *ml_fd_beta*

ml_fd_beta : *no description* (R)

Details Finite Differencing perturbation parameter alpha.

15.1.94 NONLINEAR ML PRINT LEVEL

Syntax `NONLINEAR ML PRINT LEVEL` { `=` | `IS` | `ARE` } *intValue*

intValue : *no description* (I)

Details Output level (0-10).

15.1.95 NONLINEAR ML RECALCULATION OFFSET

Syntax `NONLINEAR ML RECALCULATION OFFSET` { `= | IS | ARE` } *intValue*

intValue : *no description* (I)

Details Recalculation of preconditioner offset.

15.2 TEUCHOS PARAMETER BLOCK

`Begin TEUCHOS PARAMETER BLOCK` *Teuchos Parameter Block Name*

`PARAM-STRING` *parameter_name* `VALUE` *string_value*
`PARAM-REAL` *parameter_name* `VALUE` *real_value*
`PARAM-INT` *parameter_name* `VALUE` *integer_value*
`PARAM-BOOL` *parameter_name* `VALUE` { `false | true` }
`PARAM-SUBLIST` *parameter_name* `VALUE` *block_name*

`End`

Details A block to set a Teuchos parameter list.

15.2.1 PARAM-STRING

Syntax `PARAM-STRING` *parameter_name* `VALUE` *string_value*

parameter_name : *no description* (Q)
string_value : *no description* (Q)

Details Key/Value string-pair to be passed to solver.

15.2.2 PARAM-REAL

Syntax `PARAM-REAL` *parameter_name* `VALUE` *real_value*

parameter_name : *no description* (Q)
real_value : *no description* (R)

Details String-Key/Real-Value pair to be passed to solver.

15.2.3 PARAM-INT

Syntax `PARAM-INT` *parameter_name* `VALUE` *integer_value*

parameter_name : *no description* (Q)

integer_value : *no description* (I)

Details String-Key/Integer-Value pair to be passed to solver.

15.2.4 PARAM-BOOL

Syntax `PARAM-BOOL` *parameter_name* `VALUE` { `false` | `true` }

parameter_name : *no description* (Q)

Details String-Key/Boolean-Value pair to be passed to solver.

15.2.5 PARAM-SUBLIST

Syntax `PARAM-SUBLIST` *parameter_name* `VALUE` *block_name*

parameter_name : *no description* (Q)

block_name : *no description* (C)

Details String-Key/String-Value pair to designate another Teuchos block as a sublist to this block.

15.3 NOX DEBUGGER BLOCK

`Begin NOX DEBUGGER BLOCK` *Nox Debugger Block Name*

`PARAM-STRING` *parameter_name* `VALUE` *string_value*

`PARAM-REAL` *parameter_name* `VALUE` *real_value*

`PARAM-INT` *parameter_name* `VALUE` *integer_value*

`NOX DEBUG BASE OPERATOR` { = | IS | ARE } *Base Op Name*

`NOX DEBUG TEST OPERATOR` { = | IS | ARE } *Test Op Name*

`FLOOR VALUE` { = | IS | ARE } *floor_value*

`ABSOLUTE TOLERANCE` { = | IS | ARE } *abs_tol*

```
    RELATIVE TOLERANCE { = | IS | ARE } rel_tol
    MAX REPORTED VALUES { = | IS | ARE } max_to_report
End
```

Details A block to set a NOX debugger parameter list.

15.3.1 PARAM-STRING

Syntax `PARAM-STRING` *parameter_name* `VALUE` *string_value*

parameter_name : *no description* (Q)
string_value : *no description* (Q)

Details Key/Value string-pair to be passed to solver.

15.3.2 PARAM-REAL

Syntax `PARAM-REAL` *parameter_name* `VALUE` *real_value*

parameter_name : *no description* (Q)
real_value : *no description* (R)

Details String-Key/Real-Value pair to be passed to solver.

15.3.3 PARAM-INT

Syntax `PARAM-INT` *parameter_name* `VALUE` *integer_value*

parameter_name : *no description* (Q)
integer_value : *no description* (I)

Details String-Key/Integer-Value pair to be passed to solver.

15.3.4 NOX DEBUG BASE OPERATOR

Syntax `NOX DEBUG BASE OPERATOR { = | IS | ARE }` *Base Op Name*

Base Op Name : *no description* (C)

Details Specifies which NOX operator to use as a base operator for comparisons.

15.3.5 NOX DEBUG TEST OPERATOR

Syntax `NOX DEBUG TEST OPERATOR` { `=` | `IS` | `ARE` } *Test Op Name*

Test Op Name : *no description* (C)

Details Specifies which NOX operator to test against the base operator for comparisons.

15.3.6 FLOOR VALUE

Syntax `FLOOR VALUE` { `=` | `IS` | `ARE` } *floor_value*

floor_value : *no description* (R)

Details Specify a threshold floor value below which values are not compaored.

15.3.7 ABSOLUTE TOLERANCE

Syntax `ABSOLUTE TOLERANCE` { `=` | `IS` | `ARE` } *abs_tol*

abs_tol : *no description* (R)

Details Specify an absolute tolerance for comparing values.

15.3.8 RELATIVE TOLERANCE

Syntax `RELATIVE TOLERANCE` { `=` | `IS` | `ARE` } *rel_tol*

rel_tol : *no description* (R)

Details Specify a relative tolerance for comparing values.

15.3.9 MAX REPORTED VALUES

Syntax `MAX REPORTED VALUES` { `=` | `IS` | `ARE` } *max_to_report*

max_to_report : *no description* (I)

Details Specify the maximum number of values to report. Default is all.

15.4 NOX EPETRA OPERATOR

```
Begin NOX EPETRA OPERATOR Nox Operator Block Name

    FINITE DIFFERENCE METHOD { = | IS | ARE } NoxDifferencingOptions
    PERTURBATION COEFFICIENT ALPHA { = | IS | ARE } FDalpha
    PERTURBATION COEFFICIENT BETA { = | IS | ARE } FDbeta
    MATRIX FREE PERTURBATION COEFFICIENT LAMBDA { = | IS | ARE } MFlambda
    MATRIX FREE PERTURBATION COEFFICIENT VALUE { = | IS | ARE } MFepsilon
    COLOR GRAPH USING DISTANCE1 { = | IS | ARE } { false | true }
    OPERATOR TYPE { = | IS | ARE } NoxJacobianOperators
    FIX EMPTY ROWS { = | IS | ARE } { false | true }
End
```

Details A block to setup a NOX Epetra operator.

15.4.1 FINITE DIFFERENCE METHOD

Syntax `FINITE DIFFERENCE METHOD { = | IS | ARE }` *NoxDifferencingOptions*

NoxDifferencingOptions : *no description* { `forward difference` | `backward difference` | `centered difference` }

Details Finite differencing method.

Enums `NoxDifferencingOptions`

`forward difference` - *no description*

`backward difference` - *no description*

`centered difference` - *no description*

15.4.2 PERTURBATION COEFFICIENT ALPHA

Syntax `PERTURBATION COEFFICIENT ALPHA { = | IS | ARE }` *FDalpha*

FDalpha : *no description* (R)

Details Parameter, α, used in finite differencing perturbation value calculation.

15.4.3 PERTURBATION COEFFICIENT BETA

Syntax `PERTURBATION COEFFICIENT BETA` { `=` | `IS` | `ARE` } *FDbeta*

FDbeta : *no description* (R)

Details Parameter, β, used in finite differencing perturbation value calculation.

15.4.4 MATRIX FREE PERTURBATION COEFFICIENT LAMBDA

Syntax `MATRIX FREE PERTURBATION COEFFICIENT LAMBDA` { `=` | `IS` | `ARE` } *MFlambda*

MFlambda : *no description* (R)

Details Parameter, lambda, used to compute the perturbation size used in matrix-free residual evaluations.

15.4.5 MATRIX FREE PERTURBATION COEFFICIENT VALUE

Syntax `MATRIX FREE PERTURBATION COEFFICIENT VALUE` { `=` | `IS` | `ARE` } *MFepsilon*

MFepsilon : *no description* (R)

Details Direct specification for pertuebation size used in matrix-free residual evaluations.

15.4.6 COLOR GRAPH USING DISTANCE1

Details Flag indicating use of distance1 coloring of matrix graph.

15.4.7 OPERATOR TYPE

Syntax `OPERATOR TYPE` { `=` | `IS` | `ARE` } *NoxJacobianOperators*

NoxJacobianOperators : *no description* { `matrix free` | `finite difference` | `finite coloring` | `user supplied matrix` | `user supplied operator` }

Details Selection of the operator type

Enums	`NoxJacobianOperators`

`matrix free` - *no description*
`finite difference` - *no description*
`finite coloring` - *no description*
`user supplied matrix` - *no description*
`user supplied operator` - *no description*

15.4.8 FIX EMPTY ROWS

Details	Flag whether or not to correct empty matrix rows

15.5 NOX AZTECOO LINEAR SYSTEM

`Begin NOX AZTECOO LINEAR SYSTEM` *Nox AztecOO Linear System Block Name*

`SOLUTION METHOD` { `=` | `IS` | `ARE` } *NoxAztecSolverMethods*

`LINEAR SYSTEM TYPE` { `=` | `IS` | `ARE` } *NoxLinearSystemTypes*

`PRECONDITIONING METHOD` { `=` | `IS` | `ARE` } *NoxAztecPreconditionerMethods*

`PRECONDITION USING JACOBIAN` { `=` | `IS` | `ARE` } { `false` | `true` }

`NONLINEAR PRECONDITIONING COMPUTE FREQUENCY` { `=` | `IS` | `ARE` } *frequency*

`MAX AGE OF JACOBIAN` { `=` | `IS` | `ARE` } *max_age*

`FORCING TERM METHOD` { `=` | `IS` | `ARE` } *NoxNewtonSolveOptions*

`FORCING TERM INITIAL TOLERANCE` { `=` | `IS` | `ARE` } *init_tol*

`FORCING TERM MINIMUM TOLERANCE` { `=` | `IS` | `ARE` } *min_tol*

`FORCING TERM MAXIMUM TOLERANCE` { `=` | `IS` | `ARE` } *max_tol*

`TYPE 2 FORCING TERM ALPHA` { `=` | `IS` | `ARE` } *alpha*

`TYPE 2 FORCING TERM GAMMA` { `=` | `IS` | `ARE` } *beta*

`MAXIMUM ITERATIONS` { `=` | `IS` | `ARE` } *max_iters*

`RESIDUAL NORM TOLERANCE` { `=` | `IS` | `ARE` } *tol*

`RESTART ITERATIONS` { `=` | `IS` | `ARE` } *restart_iters*

`PRECONDITIONING PACKAGE` { `=` | `IS` | `ARE` } *PreconditioningPackages* `[ using` *Teuchos_Param_List_Name* `]`

`RESCUE BAD NEWTON SOLVE` { `=` | `IS` | `ARE` } { `false` | `true` }

`LINEAR SOLVER OUTPUT FREQUENCY` { `=` | `IS` | `ARE` } *lin_output_freq*

`SET JACOBIAN OPERATOR` { `=` | `IS` | `ARE` } *Operator_Name*

`SET PRECONDITIONER OPERATOR` { `=` | `IS` | `ARE` } *Operator_Name*

`End`

Details A block to setup a NOX AztecOO Linear System.

15.5.1 SOLUTION METHOD

Syntax `SOLUTION METHOD` { `=` | `IS` | `ARE` } *NoxAztecSolverMethods*

NoxAztecSolverMethods : *no description* { `cg` | `cgs` | `bicgstab` | `gmres` | `tfqmr` | `lu` }

Details Selection of the AztecOO linear solution method.

Enums `NoxAztecSolverMethods`

`cg` - *no description*

`cgs` - *no description*

`bicgstab` - *no description*

`gmres` - *no description*

`tfqmr` - *no description*

`lu` - *no description*

15.5.2 LINEAR SYSTEM TYPE

Syntax `LINEAR SYSTEM TYPE` { `=` | `IS` | `ARE` } *NoxLinearSystemTypes*

NoxLinearSystemTypes : *no description* { `aztecoo` | `semi-implicit` | `belos` | `generic sierra linear solver` }

Details Selection of the NOX Linear System type.

Enums `NoxLinearSystemTypes`

`aztecoo` - *no description*

`semi-implicit` - *no description*

`belos` - *no description*

`generic sierra linear solver` - *no description*

15.5.3 PRECONDITIONING METHOD

Syntax `PRECONDITIONING METHOD` { `=` | `IS` | `ARE` } *NoxAztecPreconditionerMethods*

NoxAztecPreconditionerMethods : *no description* { `none` | `jacobi` | `neumann` | `least-squares` | `dd-ilut` | `dd-ilu` | `user supplied operator` }

Details Selection of the AztecOO preconditioning methods supported by the NOX nonlinear solver.

Enums `NoxAztecPreconditionerMethods`

`none` - *no description*

`jacobi` - *no description*

`neumann` - *no description*

`least-squares` - *no description*

`dd-ilut` - *no description*

`dd-ilu` - *no description*

`user supplied operator` - *no description*

15.5.4 PRECONDITION USING JACOBIAN

Details Flag whether or not to use tha Jacobian operator as the preconditioner operator

15.5.5 NONLINEAR PRECONDITIONING COMPUTE FREQUENCY

Syntax `NONLINEAR PRECONDITIONING COMPUTE FREQUENCY` { `=` | `IS` | `ARE` } *frequency*

frequency : *no description* (I)

Details Recompute frequency of the nonlinear preconditioner operator.

15.5.6 MAX AGE OF JACOBIAN

Syntax `MAX AGE OF JACOBIAN` { `=` | `IS` | `ARE` } *max_age*

max_age : *no description* (I)

Details Integer number of nonlinear iterations between recomputations of the Jacobian

15.5.7 FORCING TERM METHOD

Syntax `FORCING TERM METHOD` { `=` | `IS` | `ARE` } *NoxNewtonSolveOptions*

`NoxNewtonSolveOptions` : *`no description`* { `Constant` | `Type 1` | `Type 2` }

Details Specification of linear solver adaptive forcing term method.

Enums `NoxNewtonSolveOptions`

`Constant` - *no description*

`Type 1` - *no description*

`Type 2` - *no description*

15.5.8 FORCING TERM INITIAL TOLERANCE

Syntax `FORCING TERM INITIAL TOLERANCE` { `=` | `IS` | `ARE` } *init_tol*

`init_tol` : *`no description`* `(R)`

Details Linear solver adaptive forcing term initial linear solve tolerance.

15.5.9 FORCING TERM MINIMUM TOLERANCE

Syntax `FORCING TERM MINIMUM TOLERANCE` { `=` | `IS` | `ARE` } *min_tol*

`min_tol` : *`no description`* `(R)`

Details Linear solver adaptive forcing term minimum linear solve tolerance.

15.5.10 FORCING TERM MAXIMUM TOLERANCE

Syntax `FORCING TERM MAXIMUM TOLERANCE` { `=` | `IS` | `ARE` } *max_tol*

`max_tol` : *`no description`* `(R)`

Details Linear solver adaptive forcing term maximum linear solve tolerance.

15.5.11 TYPE 2 FORCING TERM ALPHA

Syntax `TYPE 2 FORCING TERM ALPHA` { `=` | `IS` | `ARE` } *alpha*

`alpha` : *`no description`* `(R)`

Details Linear solver adaptive forcing term Type 2 value for α

15.5.12 TYPE 2 FORCING TERM GAMMA

Syntax `TYPE 2 FORCING TERM GAMMA` { `=` | `IS` | `ARE` } *beta*

beta : *no description* (R)

Details Linear solver adaptive forcing term Type 2 value for γ

15.5.13 MAXIMUM ITERATIONS

Syntax `MAXIMUM ITERATIONS` { `=` | `IS` | `ARE` } *max_iters*

max_iters : *no description* (I)

Details Maximum number of solution method iterations.

15.5.14 RESIDUAL NORM TOLERANCE

Syntax `RESIDUAL NORM TOLERANCE` { `=` | `IS` | `ARE` } *tol*

tol : *no description* (R)

Details Iterative solution method residual convergence tolerance.

15.5.15 RESTART ITERATIONS

Syntax `RESTART ITERATIONS` { `=` | `IS` | `ARE` } *restart_iters*

restart_iters : *no description* (I)

Details Number of iterations between GMRES restarts.

15.5.16 PRECONDITIONING PACKAGE

Syntax `PRECONDITIONING PACKAGE` { `=` | `IS` | `ARE` } *PreconditioningPackages* [`using` *Teuchos_Param_List_Name*]

PreconditioningPackages : *no description* { `aztecoo` | `ifpack` | `multilevel` }

Teuchos_Param_List_Name : *no description* (C)

Details Specify which package to use for preconditioning.

Enums `PreconditioningPackages`

- `aztecoo` - *no description*
- `ifpack` - *no description*
- `multilevel` - *no description*

15.5.17 RESCUE BAD NEWTON SOLVE

Details Flag to specify if an unconverged linear solve solution should be accepted or flagged as failed.

15.5.18 LINEAR SOLVER OUTPUT FREQUENCY

Syntax `LINEAR SOLVER OUTPUT FREQUENCY` { `= | IS | ARE` } *lin_output_freq*

lin_output_freq : *no description* (I)

Details Output frequency for iterative linear solvers.

15.5.19 SET JACOBIAN OPERATOR

Syntax `SET JACOBIAN OPERATOR` { `= | IS | ARE` } *Operator_Name*

Operator_Name : *no description* (C)

Details Specifies which NOX Epetra operator to use as the Jacobian.

15.5.20 SET PRECONDITIONER OPERATOR

Syntax `SET PRECONDITIONER OPERATOR` { `= | IS | ARE` } *Operator_Name*

Operator_Name : *no description* (C)

Details Specifies which NOX Epetra operator to use as the Preconditioner.

15.6 NOX LINESEARCH BLOCK

`Begin NOX LINESEARCH BLOCK` *Nox Linesearch Block Name*

```
    NONLINEAR LINESEARCH METHOD { = | IS | ARE } NoxLinesearchMethods
    USE PARAMETER LIST { = | IS | ARE } Teuchos_Param_List_Name
End
```

Details A block to setup a NOX Linesearch.

15.6.1 NONLINEAR LINESEARCH METHOD

Syntax `NONLINEAR LINESEARCH METHOD { = | IS | ARE }` *NoxLinesearchMethods*

NoxLinesearchMethods : *no description* { `full step | polynomial | quadratic | more thunte | nonlinear cg` }

Details Selection of the linesearch method for nonlinear solver.

Enums `NoxLinesearchMethods`

- `full step` - *no description*
- `polynomial` - *no description*
- `quadratic` - *no description*
- `more thunte` - *no description*
- `nonlinear cg` - *no description*

15.6.2 USE PARAMETER LIST

Syntax `USE PARAMETER LIST { = | IS | ARE }` *Teuchos_Param_List_Name*

Teuchos_Param_List_Name : *no description* (C)

Details Specify use of a Teuchos Parameter list

Chapter 16

LOCA Continuation Solver Reference

See, also, the LOCA parameters online reference.

16.1 LOCA CONTINUATION SOLVER

```
Begin LOCA CONTINUATION SOLVER LOCA Solver Name

    USE NOX SOLVER { = | IS | ARE } nox_solver_name

    USE PARAMETER LIST { = | IS | ARE } param_list_name Begin TEUCHOS PARAMETER BLOCK
    Teuchos Parameter Block Name

    End

End
```

Details A set of solver parameters for LOCA continuation solver.

16.1.1 USE NOX SOLVER

Syntax `USE NOX SOLVER { = | IS | ARE }` *nox_solver_name*

nox_solver_name : *no description* (C)

Details Specify a NOX solver to use for the nonlinear solves.

16.1.2 USE PARAMETER LIST

Syntax USE PARAMETER LIST { = | IS | ARE } *param_list_name*

param_list_name : *no description* (C)

Details Specify a Teuchos parameter list to use with the continuation solver.

16.2 TEUCHOS PARAMETER BLOCK

```
Begin TEUCHOS PARAMETER BLOCK Teuchos Parameter Block Name

    PARAM-STRING parameter_name VALUE string_value
    PARAM-REAL parameter_name VALUE real_value
    PARAM-INT parameter_name VALUE integer_value
    PARAM-BOOL parameter_name VALUE { false | true }
    PARAM-SUBLIST parameter_name VALUE block_name
End
```

Details A block to set a Teuchos parameter list.

16.2.1 PARAM-STRING

Syntax PARAM-STRING *parameter_name* VALUE *string_value*

parameter_name : *no description* (Q)
string_value : *no description* (Q)

Details Key/Value string-pair to be passed to solver.

16.2.2 PARAM-REAL

Syntax PARAM-REAL *parameter_name* VALUE *real_value*

parameter_name : *no description* (Q)
real_value : *no description* (R)

Details String-Key/Real-Value pair to be passed to solver.

16.2.3 PARAM-INT

Syntax `PARAM-INT` *parameter_name* `VALUE` *integer_value*

parameter_name : *no description* (Q)

integer_value : *no description* (I)

Details String-Key/Integer-Value pair to be passed to solver.

16.2.4 PARAM-BOOL

Syntax `PARAM-BOOL` *parameter_name* `VALUE` { `false` | `true` }

parameter_name : *no description* (Q)

Details String-Key/Boolean-Value pair to be passed to solver.

16.2.5 PARAM-SUBLIST

Syntax `PARAM-SUBLIST` *parameter_name* `VALUE` *block_name*

parameter_name : *no description* (Q)

block_name : *no description* (C)

Details String-Key/String-Value pair to designate another Teuchos block as a sublist to this block.

Chapter 17

Adaptivity and Error Estimation

17.1 Aria Region-Level Line Commands

17.1.1 ADAPT MESH ON

Syntax `ADAPT MESH ON` *dof EquationDof*

dof : *no description* (C)

EquationDof : *no description* (C [, ...])

Details Causes h-adaptivity to be based upon the specified solution dof using the error estimator that appears in the error estimation controller command block. The estimated global error will be output to the problem log file for each converged solution step. If cases where only error estimates are desired then one should use the ESTIMATE ERROR command instead.

17.1.2 ESTIMATE ERROR FOR

Syntax `ESTIMATE ERROR FOR` *dof EquationDof*

dof : *no description* (C)

EquationDof : *no description* (C [, ...])

Details Causes elemental error to be estimated for specified solution dof using the error estimator that appears in the error estimation controller command block. The estimated global error will be output to the problem log file for each converged solution step.

17.1.3 USE ADAPTIVITY CONTROLLER

Syntax `USE ADAPTIVITY CONTROLLER` *adaptivity_controller_name*

adaptivity_controller_name : *no description* (C)

Details This command is called from the application's region block.

17.1.4 USE ERROR ESTIMATION CONTROLLER

Syntax `USE ERROR ESTIMATION CONTROLLER` *ee_controller_name*

ee_controller_name : *no description* (C)

Details This command is called from the application's region block.

17.1.5 USE UNIFORM REFINEMENT CONTROLLER

Syntax `USE UNIFORM REFINEMENT CONTROLLER` *controller_name*

controller_name : *no description* (C)

Details This command is called from the application's region block.

17.2 ADAPTIVITY CONTROLLER

```
Begin ADAPTIVITY CONTROLLER

    MAX OUTER ADAPT STEPS { = | IS } max_outer_adapt_steps
    MAX INNER ADAPT STEPS { = | IS } max_inner_adapt_steps
    MAX ELEMENTS { = | IS } max_elements
    START TIME { = | IS } start_time
    STOP TIME { = | IS } start_time
    ADAPTIVE STRATEGY { = | IS } AdaptiveStrategyType [ USING REFINE ERROR LIMIT FACTOR
    refine_limit_factor USING UNREFINE ERROR LIMIT FACTOR unrefine_limit_factor ]
    GLOBAL ERROR TOLERANCE { = | IS } global_error_tolerance
End
```

Details Contains the commands needed to set the adaptive strategy and associated parameters, including the stopping criterion.

17.2.1 MAX OUTER ADAPT STEPS

Syntax `MAX OUTER ADAPT STEPS { = | IS }` *max_outer_adapt_steps*

max_outer_adapt_steps : *no description* (I)

Details This parameter specifies how many times the outer adaptive loop will get executed per timestep.

17.2.2 MAX INNER ADAPT STEPS

Syntax `MAX INNER ADAPT STEPS` { `=` | `IS` } *max_inner_adapt_steps*

max_inner_adapt_steps : *no description* (I)

Details This parameter specifies how many times the inner adaptive loop will get executed per timestep.

17.2.3 MAX ELEMENTS

Syntax `MAX ELEMENTS` { `=` | `IS` } *max_elements*

max_elements : *no description* (I)

Details This parameter specifies a stopping criteria based on a maximum allowable number of elements in the mesh.

17.2.4 START TIME

Syntax `START TIME` { `=` | `IS` } *start_time*

start_time : *no description* (R)

Details This command allows you to specify the solution time at which refinement will become active, that is, before this time no elements will be refined.

17.2.5 STOP TIME

Syntax `STOP TIME` { `=` | `IS` } *start_time*

start_time : *no description* (R)

Details This command allows you to specify a solution time at which refinement will become inactive, that is, on and after this time elements will no longer be refined or unrefined.

17.2.6 ADAPTIVE STRATEGY

Syntax `ADAPTIVE STRATEGY` { `=` | `IS` } *AdaptiveStrategyType* `[ USING REFINE ERROR LIMIT FACTOR` *refine_limit_factor* `USING UNREFINE ERROR LIMIT FACTOR` *unrefine_limit_factor* `]`

AdaptiveStrategyType : *no description* { `Default` | `Refine_fixed_fraction` | `NONE` }

refine_limit_factor : *no description* (R)

unrefine_limit_factor : *no description* (R)

Details The optional parameters (with valid values from 0.0 to 1.0) specify the fraction of the (global) maximum element contribution to the global error norm to use as cutoff values for element refinement and unrefinement, respectively.

Enums `AdaptiveStrategyType`

`Default` - *no description*

`Refine_fixed_fraction` - *no description*

`NONE` - *no description*

17.2.7 GLOBAL ERROR TOLERANCE

Syntax `GLOBAL ERROR TOLERANCE { = | IS }` *global_error_tolerance*

global_error_tolerance : *no description* (R)

Details This command adds a stopping criteria based on a maximum allowable value of the global relative error norm.

17.3 ERROR ESTIMATION CONTROLLER

```
Begin ERROR ESTIMATION CONTROLLER CONTROLLER_NAME

    COMPUTE METRIC { = | IS } metricNames
    COMPUTE STEP INTERVAL { = | IS } stepInterval
    COMPUTE AT OUTPUT
    ERROR ESTIMATOR { = | IS } ErrorEstimatorType
    QUANTITY OF INTEREST { = | IS } QuantityOfInterestType [ ON SURFACE sideset_name ]
    TRUTH REFINE LEVEL { = | IS } Truth_refine_level
    COMPUTE OVERKILL SOLUTION USING REFINE LEVEL { = | IS } Overkill_solution_level
    USE DUAL SOLVER DualSolver

End
```

Details Contains the commands needed to set the error estimation scheme.

17.3.1 COMPUTE METRIC

Syntax `COMPUTE METRIC { = | IS }` *metricNames*

metricNames : *no description* (C [, ...])

Details Lists a number of metrics within the error estimation class to compute

17.3.2 COMPUTE STEP INTERVAL

Syntax `COMPUTE STEP INTERVAL { = | IS }` *stepInterval*

stepInterval : *no description* (I)

Details Defines how often the error estimator is computed, by number of steps If negative the estimate will not be computed on a step

17.3.3 COMPUTE AT OUTPUT

Details Flags the error estimator to be computed immediatly prior to the mesh output step. If this option is used with “compute step interval = -1” the error estimate will only be computed prior to output otherwise the error estimate will be computed prior to output in addition to other times.

17.3.4 ERROR ESTIMATOR

Syntax `ERROR ESTIMATOR { = | IS }` *ErrorEstimatorType*

ErrorEstimatorType : *no description* `{ FluxNorm | ZZ | Truth | Distortion | Dual | FluxJump | TransferCopy | DualZZ | NONE }`

Details Contains the commands needed to specify the error estimator.

Enums `ErrorEstimatorType`

- `FluxNorm` - *no description*
- `ZZ` - *no description*
- `Truth` - *no description*
- `Distortion` - *no description*
- `Dual` - *no description*
- `FluxJump` - *no description*
- `TransferCopy` - *no description*
- `DualZZ` - *no description*
- `NONE` - *no description*

17.3.5 QUANTITY OF INTEREST

Syntax `QUANTITY OF INTEREST` { `=` | `IS` } *QuantityOfInterestType* `[ ON SURFACE` *sideset_name* `]`

QuantityOfInterestType : *no description* { `Error_Energy_Norm` | `Average_Displacement` | `Average_Surface_Displacement` | `Average_Value` | `Integrated_Surface_Flux` | `INVALID_Q_OF_I` }

sideset_name : *no description* (C)

Details The quantity of interest must be a valid enumerated type. The types must match exactly with those given by `Apub_Input_ErrEstimator::QofI_Type.`

Enums `QuantityOfInterestType`

- `Error_Energy_Norm` - *no description*
- `Average_Displacement` - *no description*
- `Average_Surface_Displacement` - *no description*
- `Average_Value` - *no description*
- `Integrated_Surface_Flux` - *no description*
- `INVALID_Q_OF_I` - *no description*

17.3.6 TRUTH REFINE LEVEL

Syntax `TRUTH REFINE LEVEL` { `=` | `IS` } *Truth_refine_level*

Truth_refine_level : *no description* (I)

Details The number of refinement levels used for generating the truth mesh. The truth problems are local (elementwise) problems solved on a refinement of each coarse element. The specified level will be used for all coarse elements. The minimum allowable value is 1. In general, a higher level will cost more (i.e., more CPU time required for the error estimation), but will result in a more accurate error estimate.

17.3.7 COMPUTE OVERKILL SOLUTION USING REFINE LEVEL

Syntax `COMPUTE OVERKILL SOLUTION USING REFINE LEVEL` { `=` | `IS` } *Overkill_solution_level*

Overkill_solution_level : *no description* (I)

Details The number of refinement levels used for the overkill solution. The minimum allowable value is 1. The overkill problem is the same as the original problem, but with a (much) finer mesh. In general the overkill solution will NOT be computed (it's expensive!!!). It is useful mainly for regression tests for small problems.

17.3.8 USE DUAL SOLVER

Syntax `USE DUAL SOLVER` *DualSolver*

DualSolver : *no description* (C)

Details This command allows the code to connect a solver block to the dual (Quantity of interest) error estimator in order to solve a different system for the dual/adjoint problem than the primary solver.

17.4 UNIFORM REFINEMENT CONTROLLER

```
Begin UNIFORM REFINEMENT CONTROLLER

    NUMBER OF OUTER STEPS { = | IS } num_outer_steps
    NUMBER OF INNER STEPS { = | IS } num_inner_steps
    INCLUDE MATERIALS [ list of material names ]
    EXCLUDE MATERIALS [ list of material names ]
    REFINE INPUT MESH USING REFINE LEVEL { = | IS } level [ THEN DELETE PARENTS ]
    USE DEPRECATED HANGING NODE TET REFINEMENT
End
```

Details Contains the commands needed to specify uniform refinement.

17.4.1 NUMBER OF OUTER STEPS

Syntax `NUMBER OF OUTER STEPS { = | IS }` *num_outer_steps*

num_outer_steps : *no description* (I)

Details This parameter specifies how many times the outer adaptive loop will get executed per timestep.

17.4.2 NUMBER OF INNER STEPS

Syntax `NUMBER OF INNER STEPS { = | IS }` *num_inner_steps*

num_inner_steps : *no description* (I)

Details This parameter specifies how many times the inner adaptive loop will get executed per timestep.

17.4.3 INCLUDE MATERIALS

Syntax `INCLUDE MATERIALS [` *list of material names* `]`

list of material names : *no description* (C [, ...])

Details List of materials to include in uniform refinement.

17.4.4 EXCLUDE MATERIALS

Syntax `EXCLUDE MATERIALS [` *list of material names* `]`

list of material names : *no description* (C [, ...])

Details List of materials to exclude in uniform refinement.

17.4.5 REFINE INPUT MESH USING REFINE LEVEL

Syntax `REFINE INPUT MESH USING REFINE LEVEL { = | IS }` *level* `[ THEN DELETE PARENTS ]`

level : *no description* (I)

Details The number of refinement levels used for the uninitialized raw input mesh.

17.4.6 USE DEPRECATED HANGING NODE TET REFINEMENT

Details Used to request hanging node tet refinement in place of default Rivara Algorithm.

Chapter 18

Dynamic Load Balancing

See, also, the Zoltan homepage Boman et al. (1999), the Zoltan User's Guide Devine et al. (1999) and an overview of Zoltan Devine et al. (2002).

18.1 ENABLE REBALANCE

Syntax `ENABLE REBALANCE WITH THRESHOLD = `*`REAL`*` USING ZOLTAN PARAMETERS `*`STRING`*

Description Enables load balancing for parallel simulations.

Details This command causes Aria to occassionally redistribute the mesh across the processors in a parallel run in order to (hopefully) balance the work. In order to perform this balancing, Aria must supply a "load" for each element on each processor to the Zoltan library. See the `REBALANCE LOAD MEASURE` commmand (18.2) for load measuring options.

The loads can be output to the results database by adding the following line to the `RESULTS OUTPUT` block in the input file:

```
Element Variables = rebalLoadMeasure as Load
```

The number supplied as a threshold determines how far out of balance the load can become before load balancing is performed and the *`STRING`* argument names the Zoltan parameter block to use. Hence, a `ZOLTAN PARAMETERS` block must also be supplied in the input file.

Parent Block(s) `ARIA_REGION`

18.2 REBALANCE LOAD MEASURE

Syntax `REBALANCE LOAD MEASURE = `*`STRING`*

Description Selects the method for measuring element loads for rebalancing.

Details Valid options are `ELEMENT` (default), `PROCESSOR` and `CONSTANT`.

`ELEMENT` : (default) assigns an element weight equal to the total cost of assembling the element for a timestep divided by the number of nonlinear iterations.

`PROCESSOR` : assigns same weight to all elements on a processor equal to the average cost of assembling an element for a timestep divided by the number of nonlinear iterations.

`CONSTANT` : assigns the same weight to each element, and is useful for regession testing.

Parent Block(s) `ARIA_REGION`

18.3 ZOLTAN PARAMETERS

```
Begin ZOLTAN PARAMETERS parameter_block_model

     Load Balancing Method { = | IS } LoadBalancingMethod
     Deterministic Decomposition { = | IS } { false | true }
     Imbalance Tolerance { = | IS } Tol
     Over Allocate Memory { = | IS } Alloc
     Reuse Cuts { = | IS } { false | true }
     Algorithm Debug Level { = | IS } Level
     Check Geometry { = | IS } { false | true }
     Keep Cuts { = | IS } { false | true }
     Lock RCB Directions { = | IS } { false | true }
     Set RCB Directions { = | IS } RCBSetDirections
     Rectilinear RCB Blocks { = | IS } { false | true }
     Renumber Partitions { = | IS } { false | true }
     Octree Dimension { = | IS } Dim
     Octree Method { = | IS } OctreeMethod
     Octree Min Objects { = | IS } Num
     Octree Max Objects { = | IS } Num
     Zoltan Debug Level { = | IS } Level
     Debug Processor Number { = | IS } Proc
     Timer { = | IS } { wall | cpu }
     Debug Memory { = | IS } Level
     rcb recompute box { = | IS } bool
     rcb max aspect ratio { = | IS } ratio

End
```

Details

The block command line parameter, parameter_block_model, is used to specify a unique block of zoltan parameters for a test or model calculation. example ZOLTAN PARAMETERS model_name

18.3.1 Load Balancing Method

Syntax

`Load Balancing Method` { `=` | `IS` } *LoadBalancingMethod*

`LoadBalancingMethod` `:` *`no description`* { `Recursive Coordinate Bisection` | `Recursive Inertial Bisection` | `Hilbert Space Filling Curve` | `Octree` }

Details

Default: Load Balancing Method = Recursive Coordinate Bisection Zoltan Equivalence: Load Balancing Method = LB_METHOD

Dynamic load rebalancing partitioning decomposition method. Valid values are: Recursive Coordinate Bisection (RCB), Recursive Inertial Bisection (RIB), Octree (OCTPART), or Hilbert Space Filling Curve (HSFC). (Carter Edward's HFFC algorithm)

examples Load Balancing Method = Hilbert Space Filling Curve

Enums

`LoadBalancingMethod`

`Recursive Coordinate Bisection` - *no description*

`Recursive Inertial Bisection` - *no description*

`Hilbert Space Filling Curve` - *no description*

`Octree` - *no description*

18.3.2 Deterministic Decomposition

Details

Default: Deterministic Decomposition = true Zoltan Equivalence: Deterministic Decomposition = DETERMINISTIC

If this value is set to true, Zoltan's computation of the new decomposition is deterministic: i.e. executing the same algorithm with the same input on the same number of processors always produces the same results.

When this parameter is false, message order and other factors may cause variations in decompositions even under identical operating conditions.

It is highly recommended not to change the default of true.

examples Deterministic Decomposition = true Deterministic Decomposition = false

18.3.3 Imbalance Tolerance

Syntax `Imbalance Tolerance { = | IS }` *Tol*

Tol : *no description* (R)

Details Default: Imbalance Tolerance = 1.1 Zoltan Equivalence: Imbalance Tolerance = IMBALANCE_TOL Greater than or equal to 1.0.

The amount of load imbalance the partitioning algorithm should deem acceptable. The load on each processor is computed as the sum of the weights of objects it is assigned. The imbalance is then computed as the maximum load divided by the average load. A value for IMBALANCE_TOL of 1.2 indicates that 20should not exceed 1.2

example Imbalance Tolerance = 1.2

18.3.4 Over Allocate Memory

Syntax `Over Allocate Memory { = | IS }` *Alloc*

Alloc : *no description* (R)

Details Default: Over Allocate Memory = 1.1 Zoltan Equivalence: Over Allocate Memory = RCB_OVERALLOC = RIB_OVERALLOC Greater than or equal to 1.0.

The amount by which to over-allocate temporary storage arrays for objects within the algorithm when additional storage is due to changes in processor assignments. Valid values are: 1.0 (no extra storage allocated), 1.5 (50

examples Over Allocate Memory = 1.0 Over Allocate Memory = 1.5

18.3.5 Reuse Cuts

Details Default: Reuse Cuts = true Zoltan Equivalence: Reuse Cuts = RCB_REUSE

Flag to indicate whether to use previous cuts as initial guesses for the current RCB invocation. Valid values are: false (don't use previous cuts), or true (use previous cuts).

examples Reuse Cuts = false Reuse Cuts = true

18.3.6 Algorithm Debug Level

Syntax `Algorithm Debug Level { = | IS }` *Level*

Level : *no description* (I)

Details Default: Algorithm Debug Level = 1 Zoltan Equivalence: Algorithm Debug Level = RCB_OUTPUT_LEVEL = RIB_OUTPUT_LEVEL = OCT_OUTPUT_LEVEL

Flag controlling the amount of timing and diagnostic output the specific algorithm produces. Valid values are 0 (no output), 1 (print summary), or 2 (print data for each processor).

examples Algorithm Debug Level = 0 Algorithm Debug Level = 1 Algorithm Debug Level = 2

18.3.7 Check Geometry

Details Default: Check Geometry = true Zoltan Equivalence: Check Geometry = CHECK_GEOM

Flag controlling the invocation of input and output error checking. Valid values are false (don't do checking), or true (do checking). Default is true.

examples Check Geometry = false Check Geometry = true

18.3.8 Keep Cuts

Details Default: Keep Cuts = true Zoltan Equivalence: Keep Cuts = KEEP_CUTS

Should information about the cuts determining the RCB, RIB, OCTPART, or HSFC type decomposition be retained? It costs a bit of time to do so, but this information is necessary if application wants to add more objects to the decomposition via calls to Zoltan_Point_Assign or to Zoltan_Box_Assign. Valid values are: false (don't keep cuts), or true (keep cuts).

examples Keep Cuts = false Keep Cuts = true

18.3.9 Lock RCB Directions

Details Default : Lock RCB Directions = false Zoltan Equivalence: Lock RCB Directions = RCB_LOCK_DIRECTIONS

Flag that determines whether the order of the directions of the cuts is kept constant after they are determined the first time RCB is called. Valid values are: false (Don't lock directions), or true (lock directions).

examples Lock RCB Directions = false Lock RCB Directions = true

18.3.10 Set RCB Directions

Syntax

`Set RCB Directions` { `=` | `IS` } *RCBSetDirections*

RCBSetDirections : *no description* { `Do not order cuts` | `xyz` | `xzy` | `yzx` | `yxz` | `zxy` | `zyx` }

Details

Default: Set RCB Directions = Do not order cuts Zoltan Equivalence: Set RCB Directions = RCB_SET_DIRECTIONS

Flag permits the order of cuts to be changed so that all of the cuts in any direction are done as a group. The number of cuts in each direction is determined and then the value of the parameter is used to determine the order that those cuts are made in. When 1D and 2D problems are partitioned, the directions corresponding to unused dimensions are ignored. Valid values are: Do not order cuts, xyz, xzy, yzx, yxz, zxy, or zyx.

examples Set RCB Directions = Do not order cuts Set RCB Directions = xyz Set RCB Directions = xzy Set RCB Directions = yzx Set RCB Directions = yxz Set RCB Directions = zxy Set RCB Directions = zyx

Enums

`RCBSetDirections`

`Do not order cuts` - *no description*

`xyz` - *no description*

`xzy` - *no description*

`yzx` - *no description*

`yxz` - *no description*

`zxy` - *no description*

`zyx` - *no description*

18.3.11 Rectilinear RCB Blocks

Details

Default: Rectilinear RCB Blocks = false Zoltan Equivalence: Rectilinear RCB Blocks = RCB_RECTILINEAR_BLOCKS

Flag controlling the shape of the resulting regions. If this option is specified, then when a cut is made, all of the dots located on the cut are moved to the same side of the cut. The resulting regions are then rectilinear. When these dots are treated as a group, then the resulting load balance may not be as good as when the group of dots is split by the cut. Valid values are: false (move dots individually), or true (move dots in groups).

examples Rectilinear RCB Blocks = false Rectilinear RCB Blocks = true

18.3.12 Renumber Partitions

Details

Default: Renumber Partitions = true Zoltan Equivalence: Renumber Partitions = REMAP

Flag to indicate whether to renumber partitions to mazimize overlap between the old decomposition and the new decomposition (to reduce data movement from old to new decompositions). Requests for remapping are ignored when, in the new decomposition, a partition is spread across multiple processors or partition sizes are specified using "Set Partition Sizes".

examples Renumber Partitions = false Renumber Partitions = true

18.3.13 Octree Dimension

Syntax

`Octree Dimension` { `= | IS` } *Dim*

Dim* : *no description `(I)`

Details

Default: Octree Dimension = 3 Zoltan Equivalence: Octree Dimension = OCT_DIM

Specifies whether the 2D or 3D Octree algorithms should be used. The 3D algorithms can be used for 2D problems, but much memory will be wasted to allow for a non-existent third dimension. Similarly, a 2D algorithm can be used for 3D surface meshes provided that the surface can be projected to the xy-plane without overlapping points. Valid values are: 2 (use 2D algorithm), or 3 (use 3D algorithm).

examples Octree Dimension = 2 Octree Dimension = 3

18.3.14 Octree Method

Syntax

`Octree Method` { `= | IS` } *OctreeMethod*

OctreeMethod* : *no description { `Morton Indexing | Grey Code | Hilbert` }

Details Default: Octree Method = Hilbert Zoltan Equivalence: OCT_METHOD = oct_method

The Space Filling Curve (SFC) to be used. To partition an octree, a traversal of the tree is used to define a global ordering on the leaves of the octree. This global ordering is often referred to as a Space-Filling Curve (SFC). The leaves of the octree can be easily assigned to processors in a manner which equally distributes work by assigning slices of the ordered list to processors. Different tree-traversal algorithms produce different global orderings or SFCs, with some SFCs having better connectivity and partition quality properties than others. Currently, Zoltan supports either Morton Indexing (i.e., Z-curve), Grey Code, or Hilbert SFCs. Morton Indexing and Grey Code SFCs are the simplest (and currently the fastest) of the SFC algorithms, but they produce lower-quality partitions than the Hilbert SFC. Valid values are: Morton Indexing, Grey Code or Hilbert.

examples Octree Method = Morton Indexing Octree Method = Grey Code Octree Method = Hilbert

Enums `OctreeMethod`

`Morton Indexing` - *no description*

`Grey Code` - *no description*

`Hilbert` - *no description*

18.3.15 Octree Min Objects

Syntax `Octree Min Objects { = | IS }` *Num*

`Num` : *`no description`* `(I)`

Details Default: Octree Min Objects = 1 Zoltan Equivalence: Octree Min Objects = OCT_MINOBJECTS

The minimum number of objects to allow in a leaf octant of the octree. These objects will be assigned as a group to a processor, so this parameter helps define the granularity of the load-balancing problem. Values

example Octree Min Objects = 5

18.3.16 Octree Max Objects

Syntax `Octree Max Objects { = | IS }` *Num*

`Num` : *`no description`* `(I)`

Details Default: Octree Max Objects = 1 Zoltan Equivalence: Octree Max Objects = OCT_MAXOBJECTS

The maximum number of objects to allow in a leaf octant of the octree. These objects will be assigned as a group to a processor, so this parameter helps define the granularity of the load-balancing problem. Values greater than or equal to one are allowable.

example Octree Max Objects = 10

18.3.17 Zoltan Debug Level

Syntax `Zoltan Debug Level` { `=` | `IS` } *Level*

Level : *no description* (I)

Details Default: Zoltan Debug Level = 1 Zoltan Equivalence: Zoltan Debug Level = DEBUG_LEVEL 0 Quiet mode; no output unless an error or warning is produced. 1 Values of all parameters set by Zoltan_Set_Param and used by the load-balancing method. 2 Timing information for Zoltan's load-balancing and auto-migration routines. 3 Timing information within load-balancing algorithms (support by algorithms is optional). 4 5 Trace information (enter/exit) for major Zoltan interface routines (printed by the processor specified by the DEBUG_PROCESSOR parameter). 6 Trace information (enter/exit) for major Zoltan interface routines (printed by all processors). 7 More detailed trace information in major Zoltan interface routines. 8 List of objects to be imported to and exported from each processor. 9 10 Maximum debug output; may include algorithm-specific output. example Zoltan Debug Level = 5

18.3.18 Debug Processor Number

Syntax `Debug Processor Number` { `=` | `IS` } *Proc*

Proc : *no description* (I)

Details Default: Debug Processor = 0 Zoltan Equivalence: Debug Processor Number = DEBUG_PROCESSOR

Processor number from which trace output should be printed when the zoltan parameter, DEBUG_LEVEL, is set to 5.

example Debug Processor Number = 2

18.3.19 Timer

Details Default: Timer = wall Zoltan Equivalence: Timer = TIMER

The timer with which you wish to measure time. Valid values are WALL and CPU.

examples Timer = wall Timer = cpu

18.3.20 Debug Memory

Syntax `Debug Memory` { `= | IS` } *Level*

`Level` : *`no description`* `(I)`

Details Default: Debug Memory = 1 Zoltan Equivalence: Debug Memory = DEBUG_MEMORY

Integer indicating the amount of low-level debugging information about memory-allocation should be kept by Zoltan's Memory Management utilities. Valid values are 0, 1, 2, and 3

examples Debug Memory = 0 Debug Memory = 1 Debug Memory = 2 Debug Memory = 3

18.3.21 rcb recompute box

Syntax `rcb recompute box` { `= | IS` } *bool*

`bool` : *`no description`* `(I)`

Details Default: rcb recompute box = 0 Zoltan Equivalence: RCB_RECOMPUTE_BOX = bool

0, leave box computations off 1, turn box computations on

examples rcb recompute box = 1

18.3.22 rcb max aspect ratio

Syntax `rcb max aspect ratio` { `= | IS` } *ratio*

`ratio` : *`no description`* `(I)`

Details Default: rcb max aspect ratio = 10 Zoltan Equivalence: RCB_MAX_ASPECT_RATIO = ratio

The maximum aspect ratio for a given part

examples rcb max aspect ratio = 5

Chapter 19

Linear Solver Reference

19.1 TRILINOS EQUATION SOLVER

`Begin TRILINOS EQUATION SOLVER` *Solver Name*

    `PARAM-STRING` *parameter_name* `VALUE` *string_value*
    `PARAM-REAL` *parameter_name* `VALUE` *real_value*
    `PARAM-INT` *parameter_name* `VALUE` *integer_value*
    `PARAM-BOOL` *parameter_name* `VALUE` { `false` | `true` }
    `PRECONDITIONING METHOD` { = | IS | ARE } *TrilinosPrecondMethods*
    `SOLUTION METHOD` { = | IS | ARE } *TrilinosSolverMethods*
    `MAXIMUM ITERATIONS` { = | IS | ARE } *max_iters*
    `RESIDUAL NORM TOLERANCE` { = | IS | ARE } *tol*
    `RESTART ITERATIONS` { = | IS | ARE } *restart_iters*
    `PRECONDITIONING STEPS` { = | IS | ARE } *steps*
    `POLYNOMIAL ORDER` { = | IS | ARE } *order*
    `ILU FILL` { = | IS | ARE } *fill_level*
    `ILU THRESHOLD` { = | IS | ARE } *threashold*
    `RESIDUAL NORM SCALING` { = | IS | ARE } *AztecResidualNormScaling*
    `DEBUG OUTPUT LEVEL` { = | IS | ARE } *level*
    `DEBUG OUTPUT PATH` { = | IS | ARE } *debugOutput*
    `FEI OUTPUT LEVEL` { = | IS | ARE } *FeiOutputLevels*
    `BC ENFORCEMENT` { = | IS | ARE } *BcEnforcement*
    `MATRIX SCALING` { = | IS | ARE } *MatrixScaling*
    `SHARED OWNERSHIP RULE` { = | IS | ARE } *SharedOwnershipRule*
    `MATRIX FORMAT` { = | IS | ARE } *MatrixFormat*
    `MATRIX REDUCTION` { = | IS | ARE } *AztecReductionType*
    `SOLVE TRANSPOSE` { = | IS | ARE } *true—false*
    `NUM LEVELS` { = | IS | ARE } *num_levels*
    `PRECONDITIONER SUBDOMAIN OVERLAP` { = | IS | ARE } *overlap*
    `MATRIX VIEWER` { = | IS | ARE } *machine:port*
    `select fei` { = | IS | ARE } *WhichFEI*

    FEI ERROR BEHAVIOR { = | IS | ARE } *FeiErrorBehavior* Begin TEUCHOS PARAMETER BLOCK
    Teuchos Parameter Block Name

    End

End

Details A set of solver parameters for Trilinos equation solver.

19.1.1 PARAM-STRING

Syntax PARAM-STRING *parameter_name* VALUE *string_value*

parameter_name : *no description* (Q)
string_value : *no description* (Q)

Details Key/Value string-pair to be passed to solver.

19.1.2 PARAM-REAL

Syntax PARAM-REAL *parameter_name* VALUE *real_value*

parameter_name : *no description* (Q)
real_value : *no description* (R)

Details String-Key/Real-Value pair to be passed to solver.

19.1.3 PARAM-INT

Syntax PARAM-INT *parameter_name* VALUE *integer_value*

parameter_name : *no description* (Q)
integer_value : *no description* (I)

Details String-Key/Integer-Value pair to be passed to solver.

19.1.4 PARAM-BOOL

Syntax PARAM-BOOL *parameter_name* VALUE { false | true }

parameter_name : *no description* (Q)

Details String-Key/Boolean-Value pair to be passed to solver.

19.1.5 PRECONDITIONING METHOD

Syntax `PRECONDITIONING METHOD` { `=` | `IS` | `ARE` } *TrilinosPrecondMethods*

TrilinosPrecondMethods : *no description* { `none` | `jacobi` | `neumann` | `least-squares` | `dd-lu` | `dd-ilut` | `dd-ilu` | `dd-rilu` | `dd-bilu` | `dd-icc` | `multilevel` }

Details Selection of the preconditioning method.

Enums `TrilinosPrecondMethods`

`none` - *no description*

`jacobi` - *no description*

`neumann` - *no description*

`least-squares` - *no description*

`dd-lu` - *no description*

`dd-ilut` - *no description*

`dd-ilu` - *no description*

`dd-rilu` - *no description*

`dd-bilu` - *no description*

`dd-icc` - *no description*

`multilevel` - *no description*

19.1.6 SOLUTION METHOD

Syntax `SOLUTION METHOD` { `=` | `IS` | `ARE` } *TrilinosSolverMethods*

TrilinosSolverMethods : *no description* { `cg` | `cgs` | `bicgstab` | `gmres` | `tfqmr` | `lu` | `amesos-klu` | `amesos-mumps` | `amesos-scalapack` | `amesos-umfpack` | `amesos-dscpack` | `amesos-superludist` }

Details Selection of the linear-system solution algorithm.

Enums `TrilinosSolverMethods`

- `cg` - *no description*
- `cgs` - *no description*
- `bicgstab` - *no description*
- `gmres` - *no description*
- `tfqmr` - *no description*
- `lu` - *no description*
- `amesos-klu` - *no description*
- `amesos-mumps` - *no description*
- `amesos-scalapack` - *no description*
- `amesos-umfpack` - *no description*
- `amesos-dscpack` - *no description*
- `amesos-superludist` - *no description*

19.1.7 MAXIMUM ITERATIONS

Syntax `MAXIMUM ITERATIONS { = | IS | ARE }` *max_iters*

max_iters : *no description* (I)

Details Maximum number of solution method iterations.

19.1.8 RESIDUAL NORM TOLERANCE

Syntax `RESIDUAL NORM TOLERANCE { = | IS | ARE }` *tol*

tol : *no description* (R)

Details Iterative solution method residual convergence tolerance.

19.1.9 RESTART ITERATIONS

Syntax `RESTART ITERATIONS { = | IS | ARE }` *restart_iters*

restart_iters : *no description* (I)

Details Number of iterations between GMRES restarts.

19.1.10 PRECONDITIONING STEPS

Syntax `PRECONDITIONING STEPS` { `=` | `IS` | `ARE` } *steps*

steps : *no description* (I)

Details Number of Jacobi, Gauss-Seidel, or other preconditioning methods' applications per iteration.

19.1.11 POLYNOMIAL ORDER

Syntax `POLYNOMIAL ORDER` { `=` | `IS` | `ARE` } *order*

order : *no description* (I)

Details Polynomial order of preconditioning method.

19.1.12 ILU FILL

Syntax `ILU FILL` { `=` | `IS` | `ARE` } *fill_level*

fill_level : *no description* (I)

Details Fill-in parameter for incomplete factorizations.

19.1.13 ILU THRESHOLD

Syntax `ILU THRESHOLD` { `=` | `IS` | `ARE` } *threashold*

threashold : *no description* (R)

Details Threshold parameter for incomplete factorizations.

19.1.14 RESIDUAL NORM SCALING

Syntax `RESIDUAL NORM SCALING` { `=` | `IS` | `ARE` } *AztecResidualNormScaling*

AztecResidualNormScaling : *no description* { `none` | `RHS` | `R0` | `Anorm` }

Details Scaling method for the residual norm.

Enums `AztecResidualNormScaling`

- `none` - *no description*
- `RHS` - *no description*
- `R0` - *no description*
- `Anorm` - *no description*

19.1.15 DEBUG OUTPUT LEVEL

Syntax `DEBUG OUTPUT LEVEL` { `=` | `IS` | `ARE` } *level*

level : *no description* (I)

Details Output level for debugging. Generally 0 means no solver screen output, and higher values of this parameter correspond to more screen output.

19.1.16 DEBUG OUTPUT PATH

Syntax `DEBUG OUTPUT PATH` { `=` | `IS` | `ARE` } *debugOutput*

debugOutput : *no description* (C)

Details Specify path where debug-logs and other debug output files will be placed.

19.1.17 FEI OUTPUT LEVEL

Syntax `FEI OUTPUT LEVEL` { `=` | `IS` | `ARE` } *FeiOutputLevels*

FeiOutputLevels : *no description* { `none` | `matrix_files` | `stats` | `brief_logs` | `full_logs` | `all` }

Details Control the amount of output produced by FEI.

Enums `FeiOutputLevels`

- `none` - *no description*
- `matrix_files` - *no description*
- `stats` - *no description*
- `brief_logs` - *no description*
- `full_logs` - *no description*
- `all` - *no description*

19.1.18 BC ENFORCEMENT

Syntax `BC ENFORCEMENT` { `=` | `IS` | `ARE` } *BcEnforcement*

BcEnforcement : *no description* { `solver` | `exact` | `remove` | `solver_no_column_mod` | `exact_no_column_mod` }

Details Controls the way Dirichlet BCs are enforced.

Enums `BcEnforcement`

`solver` - *no description*

`exact` - *no description*

`remove` - *no description*

`solver_no_column_mod` - *no description*

`exact_no_column_mod` - *no description*

19.1.19 MATRIX SCALING

Syntax `MATRIX SCALING` { `=` | `IS` | `ARE` } *MatrixScaling*

MatrixScaling : *no description* { `none` | `jacobi` | `block-jacobi` | `row-sum` | `symmetric-diagonal` | `symmetric-row-sum` | `user supplied` }

Details Scaling to be applied to the matrix.

Enums `MatrixScaling`

`none` - *no description*

`jacobi` - *no description*

`block-jacobi` - *no description*

`row-sum` - *no description*

`symmetric-diagonal` - *no description*

`symmetric-row-sum` - *no description*

`user supplied` - *no description*

19.1.20 SHARED OWNERSHIP RULE

Syntax `SHARED OWNERSHIP RULE` { `=` | `IS` | `ARE` } *SharedOwnershipRule*

SharedOwnershipRule : *no description* { `low-numbered-proc` | `proc-with-local-elem` }

Details Controls the way owning processors are chosen for shared nodes in the FEI.

Enums `SharedOwnershipRule`

`low-numbered-proc` - *no description*
`proc-with-local-elem` - *no description*

19.1.21 MATRIX FORMAT

Syntax `MATRIX FORMAT { = | IS | ARE }` *MatrixFormat*

MatrixFormat : *no description* `{ MSR | VBR | CSR }`

Details Storage format for the matrix.

Enums `MatrixFormat`

`MSR` - *no description*
`VBR` - *no description*
`CSR` - *no description*

19.1.22 MATRIX REDUCTION

Syntax `MATRIX REDUCTION { = | IS | ARE }` *AztecReductionType*

AztecReductionType : *no description* `{ fei-remove-slaves }`

Details Remove constraint equations from matrix.

Enums `AztecReductionType`

`fei-remove-slaves` - *no description*

19.1.23 SOLVE TRANSPOSE

Syntax `SOLVE TRANSPOSE { = | IS | ARE }` *true—false*

true/false : *no description* (C)

Details Whether to solve for transpose of system matrix.

19.1.24 NUM LEVELS

Syntax `NUM LEVELS { = | IS | ARE }` *num_levels*

num_levels : *no description* (I)

Details Number of levels for multi-level/multi-grid solvers.

19.1.25 PRECONDITIONER SUBDOMAIN OVERLAP

Syntax `PRECONDITIONER SUBDOMAIN OVERLAP` { `= | IS | ARE` } *overlap*

overlap : *no description* (I)

Details Ovrlap of Schwarz subdomains (eg, 0,1 or 2).

19.1.26 MATRIX VIEWER

Syntax `MATRIX VIEWER` { `= | IS | ARE` } *machine:port*

machine:port : *no description* (C)

Details Host and port-number where matvis is running.

19.1.27 select fei

Syntax `select fei` { `= | IS | ARE` } *WhichFEI*

WhichFEI : *no description* { `old | new` }

Details Selection of old vs new fei.

Enums `WhichFEI`

`old` - *no description*
`new` - *no description*

19.1.28 FEI ERROR BEHAVIOR

Syntax `FEI ERROR BEHAVIOR` { `= | IS | ARE` } *FeiErrorBehavior*

FeiErrorBehavior : *no description* { `returncode | abort` }

Details Set FEI error behavior to abort (rather than the default which is to simply print a message and return an integer error code).

Enums `FeiErrorBehavior`

`returncode` - *no description*
`abort` - *no description*

19.2 TEUCHOS PARAMETER BLOCK

```
Begin TEUCHOS PARAMETER BLOCK Teuchos Parameter Block Name

    PARAM-STRING parameter_name VALUE string_value
    PARAM-REAL parameter_name VALUE real_value
    PARAM-INT parameter_name VALUE integer_value
    PARAM-BOOL parameter_name VALUE { false | true }
    PARAM-SUBLIST parameter_name VALUE block_name

End
```

Details A block to set a Teuchos parameter list.

19.2.1 PARAM-STRING

Syntax `PARAM-STRING` *parameter_name* `VALUE` *string_value*

parameter_name : *no description* (Q)
string_value : *no description* (Q)

Details Key/Value string-pair to be passed to solver.

19.2.2 PARAM-REAL

Syntax `PARAM-REAL` *parameter_name* `VALUE` *real_value*

parameter_name : *no description* (Q)
real_value : *no description* (R)

Details String-Key/Real-Value pair to be passed to solver.

19.2.3 PARAM-INT

Syntax `PARAM-INT` *parameter_name* `VALUE` *integer_value*

parameter_name : *no description* (Q)
integer_value : *no description* (I)

Details String-Key/Integer-Value pair to be passed to solver.

19.2.4 PARAM-BOOL

Syntax PARAM-BOOL *parameter_name* VALUE { false | true }

parameter_name : *no description* (Q)

Details String-Key/Boolean-Value pair to be passed to solver.

19.2.5 PARAM-SUBLIST

Syntax PARAM-SUBLIST *parameter_name* VALUE *block_name*

parameter_name : *no description* (Q)

block_name : *no description* (C)

Details String-Key/String-Value pair to designate another Teuchos block as a sublist to this block.

19.3 AZTEC EQUATION SOLVER

Begin AZTEC EQUATION SOLVER *Solver Name*

PARAM-STRING *parameter_name* VALUE *string_value*

PARAM-REAL *parameter_name* VALUE *real_value*

PARAM-INT *parameter_name* VALUE *integer_value*

SOLUTION METHOD { = | IS | ARE } *AztecSolverMethods*

PRECONDITIONING METHOD { = | IS | ARE } *AztecPrecondMethods*

MAXIMUM ITERATIONS { = | IS | ARE } *max_iters*

RESIDUAL NORM TOLERANCE { = | IS | ARE } *tol*

RESTART ITERATIONS { = | IS | ARE } *restart_iters*

PRECONDITIONING STEPS { = | IS | ARE } *steps*

POLYNOMIAL ORDER { = | IS | ARE } *order*

ILU FILL { = | IS | ARE } *fill_level*

ILU THRESHOLD { = | IS | ARE } *threashold*

RESIDUAL NORM SCALING { = | IS | ARE } *AztecResidualNormScaling*

ILU OMEGA { = | IS | ARE } *value*

DEBUG OUTPUT LEVEL { = | IS | ARE } *level*

DEBUG OUTPUT PATH { = | IS | ARE } *debugOutput*

FEI OUTPUT LEVEL { = | IS | ARE } *FeiOutputLevels*

ORTHOG METHOD { = | IS | ARE } *OrthogMethod*

BC ENFORCEMENT { = | IS | ARE } *BcEnforcement*

    MATRIX SCALING { = | IS | ARE } *MatrixScaling*
    SHARED OWNERSHIP RULE { = | IS | ARE } *SharedOwnershipRule*
    MATRIX FORMAT { = | IS | ARE } *MatrixFormat*
    MATRIX REDUCTION { = | IS | ARE } *AztecReductionType*
    NUM LEVELS { = | IS | ARE } *num_levels*
    PRECONDITIONER SUBDOMAIN OVERLAP { = | IS | ARE } *overlap*
    MATRIX VIEWER { = | IS | ARE } *machine:port*
    select fei { = | IS | ARE } *WhichFEI*
    FEI ERROR BEHAVIOR { = | IS | ARE } *FeiErrorBehavior*

End

Details A set of solver parameters for Aztec equation solver.

19.3.1 PARAM-STRING

Syntax PARAM-STRING *parameter_name* VALUE *string_value*

parameter_name : *no description* (Q)
string_value : *no description* (Q)

Details Key/Value string-pair to be passed to solver.

19.3.2 PARAM-REAL

Syntax PARAM-REAL *parameter_name* VALUE *real_value*

parameter_name : *no description* (Q)
real_value : *no description* (R)

Details String-Key/Real-Value pair to be passed to solver.

19.3.3 PARAM-INT

Syntax PARAM-INT *parameter_name* VALUE *integer_value*

parameter_name : *no description* (Q)
integer_value : *no description* (I)

Details String-Key/Integer-Value pair to be passed to solver.

19.3.4 SOLUTION METHOD

Syntax `SOLUTION METHOD` { `= | IS | ARE` } *AztecSolverMethods*

AztecSolverMethods : ***no description*** { `cg | cgs | bicgstab | gmres | tfqmr | lu` }

Details Selection of the linear-system solution algorithm.

Enums `AztecSolverMethods`

- `cg` - *no description*
- `cgs` - *no description*
- `bicgstab` - *no description*
- `gmres` - *no description*
- `tfqmr` - *no description*
- `lu` - *no description*

19.3.5 PRECONDITIONING METHOD

Syntax `PRECONDITIONING METHOD` { `= | IS | ARE` } *AztecPrecondMethods*

AztecPrecondMethods : ***no description*** { `none | jacobi | neumann | least-squares | symmetric-gauss-seidel | dd-lu | dd-ilut | dd-ilu | dd-rilu | dd-bilu | dd-icc` }

Details Selection of the equation preconditioning method.

Enums `AztecPrecondMethods`

- `none` - *no description*
- `jacobi` - *no description*
- `neumann` - *no description*
- `least-squares` - *no description*
- `symmetric-gauss-seidel` - *no description*
- `dd-lu` - *no description*
- `dd-ilut` - *no description*
- `dd-ilu` - *no description*
- `dd-rilu` - *no description*
- `dd-bilu` - *no description*
- `dd-icc` - *no description*

19.3.6 MAXIMUM ITERATIONS

Syntax `MAXIMUM ITERATIONS` { `=` | `IS` | `ARE` } *max_iters*

max_iters : *no description* (I)

Details Maximum number of solution method iterations.

19.3.7 RESIDUAL NORM TOLERANCE

Syntax `RESIDUAL NORM TOLERANCE` { `=` | `IS` | `ARE` } *tol*

tol : *no description* (R)

Details Iterative solution method residual convergence tolerance.

19.3.8 RESTART ITERATIONS

Syntax `RESTART ITERATIONS` { `=` | `IS` | `ARE` } *restart_iters*

restart_iters : *no description* (I)

Details Number of iterations between GMRES restarts.

19.3.9 PRECONDITIONING STEPS

Syntax `PRECONDITIONING STEPS` { `=` | `IS` | `ARE` } *steps*

steps : *no description* (I)

Details Number of Jacobi, Gauss-Seidel, or other preconditioning methods' applications per iteration.

19.3.10 POLYNOMIAL ORDER

Syntax `POLYNOMIAL ORDER` { `=` | `IS` | `ARE` } *order*

order : *no description* (I)

Details Polynomial order of preconditioning method.

19.3.11 ILU FILL

Syntax `ILU FILL` { `=` | `IS` | `ARE` } *fill_level*

fill_level : *no description* (I)

Details Fill-in parameter for incomplete factorizations.

19.3.12 ILU THRESHOLD

Syntax `ILU THRESHOLD` { `=` | `IS` | `ARE` } *threashold*

threashold : *no description* (R)

Details Threshold parameter for incomplete factorizations.

19.3.13 RESIDUAL NORM SCALING

Syntax `RESIDUAL NORM SCALING` { `=` | `IS` | `ARE` } *AztecResidualNormScaling*

AztecResidualNormScaling : *no description* { `none` | `RHS` | `R0` | `Anorm` }

Details Scaling method for the residual norm.

Enums `AztecResidualNormScaling`

`none` - *no description*
`RHS` - *no description*
`R0` - *no description*
`Anorm` - *no description*

19.3.14 ILU OMEGA

Syntax `ILU OMEGA` { `=` | `IS` | `ARE` } *value*

value : *no description* (R)

Details Omega parameter, dd-rilu uses ILU(k,w), w==omega

19.3.15 DEBUG OUTPUT LEVEL

Syntax `DEBUG OUTPUT LEVEL` { `=` | `IS` | `ARE` } *level*

level : *no description* (I)

Details Output level for debugging. Generally 0 means no solver screen output, and higher values of this parameter correspond to more screen output.

19.3.16 DEBUG OUTPUT PATH

Syntax `DEBUG OUTPUT PATH` { `=` | `IS` | `ARE` } *debugOutput*

debugOutput : *no description* (C)

Details Specify path where debug-logs and other debug output files will be placed.

19.3.17 FEI OUTPUT LEVEL

Syntax `FEI OUTPUT LEVEL` { `=` | `IS` | `ARE` } *FeiOutputLevels*

FeiOutputLevels : *no description* { `none` | `matrix_files` | `stats` | `brief_logs` | `full_logs` | `all` }

Details Control the amount of output produced by FEI.

Enums `FeiOutputLevels`

- `none` - *no description*
- `matrix_files` - *no description*
- `stats` - *no description*
- `brief_logs` - *no description*
- `full_logs` - *no description*
- `all` - *no description*

19.3.18 ORTHOG METHOD

Syntax `ORTHOG METHOD` { `=` | `IS` | `ARE` } *OrthogMethod*

OrthogMethod : *no description* { `modified` | `classical` }

Details User can choose orthogonalization method used by Aztec GMRES.

Enums `OrthogMethod`

- `modified` - *no description*
- `classical` - *no description*

19.3.19 BC ENFORCEMENT

Syntax `BC ENFORCEMENT` { `= | IS | ARE` } *BcEnforcement*

BcEnforcement : ***no description*** { `solver | exact | remove | solver_no_column_mod | exact_no_column_mod` }

Details Controls the way Dirichlet BCs are enforced.

Enums `BcEnforcement`

- `solver` - *no description*
- `exact` - *no description*
- `remove` - *no description*
- `solver_no_column_mod` - *no description*
- `exact_no_column_mod` - *no description*

19.3.20 MATRIX SCALING

Syntax `MATRIX SCALING` { `= | IS | ARE` } *MatrixScaling*

MatrixScaling : ***no description*** { `none | jacobi | block-jacobi | row-sum | symmetric-diagonal | symmetric-row-sum | user supplied` }

Details Scaling to be applied to the matrix.

Enums `MatrixScaling`

- `none` - *no description*
- `jacobi` - *no description*
- `block-jacobi` - *no description*
- `row-sum` - *no description*
- `symmetric-diagonal` - *no description*
- `symmetric-row-sum` - *no description*
- `user supplied` - *no description*

19.3.21 SHARED OWNERSHIP RULE

Syntax `SHARED OWNERSHIP RULE` { `=` | `IS` | `ARE` } *SharedOwnershipRule*

SharedOwnershipRule : *no description* { `low-numbered-proc` | `proc-with-local-elem` }

Details Controls the way owning processors are chosen for shared nodes in the FEI.

Enums `SharedOwnershipRule`

`low-numbered-proc` - *no description*

`proc-with-local-elem` - *no description*

19.3.22 MATRIX FORMAT

Syntax `MATRIX FORMAT` { `=` | `IS` | `ARE` } *MatrixFormat*

MatrixFormat : *no description* { `MSR` | `VBR` | `CSR` }

Details Storage format for the matrix.

Enums `MatrixFormat`

`MSR` - *no description*

`VBR` - *no description*

`CSR` - *no description*

19.3.23 MATRIX REDUCTION

Syntax `MATRIX REDUCTION` { `=` | `IS` | `ARE` } *AztecReductionType*

AztecReductionType : *no description* { `fei-remove-slaves` }

Details Remove constraint equations from matrix.

Enums `AztecReductionType`

`fei-remove-slaves` - *no description*

19.3.24 NUM LEVELS

Syntax `NUM LEVELS` { `=` | `IS` | `ARE` } *num_levels*

num_levels : *no description* (I)

Details Number of levels for multi-level/multi-grid solvers.

19.3.25 PRECONDITIONER SUBDOMAIN OVERLAP

Syntax `PRECONDITIONER SUBDOMAIN OVERLAP { = | IS | ARE }` *overlap*

`overlap` `:` *`no description`* `(I)`

Details Ovrlap of Schwarz subdomains (eg, 0,1 or 2).

19.3.26 MATRIX VIEWER

Syntax `MATRIX VIEWER { = | IS | ARE }` *machine:port*

`machine:port` `:` *`no description`* `(C)`

Details Host and port-number where matvis is running.

19.3.27 select fei

Syntax `select fei { = | IS | ARE }` *WhichFEI*

`WhichFEI` `:` *`no description`* `{ old | new }`

Details Selection of old vs new fei.

Enums `WhichFEI`

- `old` - *no description*
- `new` - *no description*

19.3.28 FEI ERROR BEHAVIOR

Syntax `FEI ERROR BEHAVIOR { = | IS | ARE }` *FeiErrorBehavior*

`FeiErrorBehavior` `:` *`no description`* `{ returncode | abort }`

Details Set FEI error behavior to abort (rather than the default which is to simply print a message and return an integer error code).

Enums `FeiErrorBehavior`

- `returncode` - *no description*
- `abort` - *no description*

Chapter 20

Postprocessing Operations

20.1 POSTPROCESS

The POSTPROCESS line command is used to add extra calculations to a region, that are carried out after the region's main calculations. They also add extra fields to the Exodus II output file.

Syntax `POSTPROCESS` *`OPERATION`*`[[]_SUBINDEX] ON` *`MESH_PART`* `[param`$_1$ `spec`$_1$`]`

Description Performs the postprocessing operation *`OPERATION`* on the mesh entity associated with *`MESH_PART`*.

Details The *`MESH_PART`* may be sideset (e.g., "surface_10") or element block (e.g., "block_1").

Parent Block(s) `ARIA_REGION`

Operation

FLUID_TRACTION

Description Computes the value of $\boldsymbol{n} \cdot \sigma$ on a surface where $\boldsymbol{n}$ is the unit normal to the surface and σ is the fluid stress tensor.

Details The fluid tractions are computed on the surface element nodes of *`MESH_PART`* . The surface element corresponds to a face of the parent element for the fluid MOMENTUM equation.

Operation

FLUID_FORCE

Parameters `[USING INTERP ]`

Description Computes the equivalent nodal point fluid force based upon the integral of $\boldsymbol{n} \cdot \sigma$ on a surface where $\boldsymbol{n}$ is the unit normal to the surface and σ is the fluid stress tensor.

Details By default the equivalent nodal forces are computed on the parent element for the fluid MOMENTUM equation. The USING ***INTERP*** option is provided to allow projection of the equivalent nodal force onto element nodes corresponding to the ***INTERP*** surface element. A valid ***INTERP*** entry can only be the parent element or a sub-element of the parent element. Admissible values of ***INTERP*** are Q_1, Q_2, QS_2, T_1 and T_2.

Operation

ELECTRIC_FIELD

Parameters `[USING INTERP ]`

Description Computes the electric field.

Details Computes the electric field as

$$\boldsymbol{E} = -\boldsymbol{\nabla} V$$

Currently, this only works on the default voltage (subindex=-1)

Operation

CURRENT

Parameters `[USING INTERP ]`

Example `Postprocess Current on block_1`

Description Computes the electrical current.

Details Computes the current as

$$\boldsymbol{J} = -\sigma_e \boldsymbol{\nabla} V$$

where V is the voltage (electric potential) and σ_e is the electrical conductivity.

Operation

PRESSURE

Parameters `[USING` *`INTERP`* `]`

Description Computes the pressure at all of the nodes. This is useful if the pressure degree of freedom uses a lower order finite element interpolation than the mesh, e.g., linear (Q1) pressure on a quadratic (Q2) mesh, or constant-over-the-element (P0) vs. linear (Q1).

Details Currently, the value assigned to a node is the average value based on the number of elements that share that node. There's no slick projection or weighting involved. Therefore, convergence studies based on the pressure will seriously underestimate the fidelity of a calculation, until this is fixed.

Operation

TEMPERATURE

Parameters `[USING` *`INTERP`* `]`

Description Computes the temperature on all of the nodes. This is useful if the temperature degree of freedom uses a lower order interpolation that the mesh, e.g., linear (Q1) temperature on a quadratic (Q2) mesh.

Details Currently, the value assigned to a node is the average value based on the number of elements that share that node. There's no slick projection or weighting scheme involved.

Operation

DENSITY

Parameters `[USING` *`INTERP`* `]`

Description Computes the density material property so that it's available for output.

Details The value assigned to a node is the average value based on the number of elements that share that node. There's no slick projection or weighting involved.

Operation

VISCOSITY

Parameters `[USING` *`INTERP`* `]`

Description Computes the viscosity material property so that it's available for output.

Details No details to speak of. Oh, well there's one detail you may care about: The value assigned to a node is the average value based on the number of elements that share that node. There's no slick projection or weighting involved.

Operation

THERMAL_CONDUCTIVITY

Parameters `[USING` *`INTERP`* `]`

Description Computes the thermal conductivity material property so that it's available for output.

Details No details to speak of. Oh, well there's one detail you may care about: The value assigned to a node is the average value based on the number of elements that share that node. There's no slick projection or weighting involved.

Operation

SPECIFIC_HEAT

Parameters `[USING` *`INTERP`* `]`

Description Computes the specific heat material property so that it's available for output.

Details No details to speak of. Oh, well there's one detail you may care about: The value assigned to a node is the average value based on the number of elements that share that node. There's no slick projection or weighting involved.

Operation

PROJECTED_FLUID_STRESS_XX

Parameters `[USING INTERP ]`

Description Computes one of the stress tensor components, calculated from the stress projection equation, at all of the nodes. This is useful if this degree of freedom uses a lower order finite element interpolation than the mesh, e.g., linear (Q1) pressure on a quadratic (Q2) mesh, or constant-over-the-element (P0) vs. linear (Q1).

Details XX, here, may be equal to XX, XY, YX, and YY for 2D, and XX, XY, XZ, YX, YY, YZ, ZX, ZY, and ZZ for 3D. In other words each component of the tensor must be individually postprocessed and output to an exodus file.

Chapter 21

Enclosure Radiation Reference

21.1 VIEWFACTOR CALCULATION

```
Begin VIEWFACTOR CALCULATION vf_calc

    BSP TREE MAX DEPTH { = | IS } DEPTH AND MIN LIST LENGTH { = | IS } L
    GEOMETRIC TOLERANCE { = | IS } n
    COMPUTE RULE { = | IS } VFComputeRule
    HEMICUBE RESOLUTION { = | IS } n
    HEMICUBE MAX SUBDIVIDES { = | IS } n
    HEMICUBE MIN SEPARATION { = | IS } n
    PAIRWISE NUMBER OF VISIBILITY SAMPLES { = | IS } n
    PAIRWISE VISIBILITY SAMPLE RULE { = | IS } AVSampleRule
    PAIRWISE NUMBER OF MONTE CARLO SAMPLES { = | IS } n
    PAIRWISE MONTE CARLO SAMPLE RULE { = | IS } AMCSampleRule
    PAIRWISE MONTE CARLO TOL1 { = | IS } real_value
    PAIRWISE MONTE CARLO TOL2 { = | IS } real_value
    OUTPUT RULE { = | IS } OutputRule
    NUMBER OF ROTATIONS { = | IS } n
    X-Y PLANE SYMMETRY
    X-Z PLANE SYMMETRY
    Y-Z PLANE SYMMETRY
End
```

Details This block command specifies a radiation enclosure and is used to define a method for calculating view factors. The parameter for this block corresponds to an instance of a radiation enclosure mechanics.

21.1.1 BSP TREE MAX DEPTH

Syntax `BSP TREE MAX DEPTH` { `=` | `IS` } *DEPTH* `AND MIN LIST LENGTH` { `=` | `IS` } *L*

DEPTH : *no description* (I)

L : *no description* (I)

Details This line command sets the BSP tree parameters.

21.1.2 GEOMETRIC TOLERANCE

Syntax `GEOMETRIC TOLERANCE` { `=` | `IS` } *n*

n : *no description* (R)

Details Set the geometric tolerance

21.1.3 COMPUTE RULE

Syntax `COMPUTE RULE` { `=` | `IS` } *VFComputeRule*

VFComputeRule : *no description* { `PAIRWISE` | `HEMICUBE` | `READ` }

Details This line command sets the method for computing the view factors for this enclosure. Default value is HEMICUBE

Enums `VFComputeRule`

`PAIRWISE` - *no description*

`HEMICUBE` - *no description*

`READ` - *no description*

21.1.4 HEMICUBE RESOLUTION

Syntax `HEMICUBE RESOLUTION` { `=` | `IS` } *n*

n : *no description* (I)

Details Set the hemicube resolution

21.1.5 HEMICUBE MAX SUBDIVIDES

Syntax `HEMICUBE MAX SUBDIVIDES` { `=` | `IS` } *n*

n : *no description* (I)

Details Set the upper limit of hemicube subdivides

21.1.6 HEMICUBE MIN SEPARATION

Syntax `HEMICUBE MIN SEPARATION` { `=` | `IS` } *n*

n : *no description* (R)

Details Set the hemicube minimum seperation

21.1.7 PAIRWISE NUMBER OF VISIBILITY SAMPLES

Syntax `PAIRWISE NUMBER OF VISIBILITY SAMPLES` { `=` | `IS` } *n*

n : *no description* (I)

Details Set the pairwise number visibility sample points

21.1.8 PAIRWISE VISIBILITY SAMPLE RULE

Syntax `PAIRWISE VISIBILITY SAMPLE RULE` { `=` | `IS` } *AVSampleRule*

AVSampleRule : *no description* { `RANDOM` | `UNIFORM` | `JITTER` | `HALTON` }

Details Set the pairwise visibility sample rule

Enums `AVSampleRule`

`RANDOM` - *no description*
`UNIFORM` - *no description*
`JITTER` - *no description*
`HALTON` - *no description*

21.1.9 PAIRWISE NUMBER OF MONTE CARLO SAMPLES

Syntax `PAIRWISE NUMBER OF MONTE CARLO SAMPLES` { `=` | `IS` } *n*

n : *no description* (I)

Details Set the pairwise number of Monte Carlo sample points

21.1.10 PAIRWISE MONTE CARLO SAMPLE RULE

Syntax `PAIRWISE MONTE CARLO SAMPLE RULE` { `=` | `IS` } *AMCSampleRule*

AMCSampleRule : *no description* { `RANDOM` | `UNIFORM` | `JITTER` | `HALTON` }

Details

Enums `AMCSampleRule`

`RANDOM` - *no description*

`UNIFORM` - *no description*

`JITTER` - *no description*

`HALTON` - *no description*

21.1.11 PAIRWISE MONTE CARLO TOL1

Syntax `PAIRWISE MONTE CARLO TOL1` { `=` | `IS` } *real_value*

real_value : *no description* (R)

Details Set first of two convergence checks for Monte Carlo integration

21.1.12 PAIRWISE MONTE CARLO TOL2

Syntax `PAIRWISE MONTE CARLO TOL2` { `=` | `IS` } *real_value*

real_value : *no description* (R)

Details Set second of two convergence checks for Monte Carlo integration

21.1.13 OUTPUT RULE

Syntax `OUTPUT RULE` { `=` | `IS` } *OutputRule*

OutputRule : *no description* { `NONE` | `SUMMARY` | `VERBOSE` }

Details Toggle verbose reporting

Enums `OutputRule`

`NONE` - *no description*
`SUMMARY` - *no description*
`VERBOSE` - *no description*

21.1.14 NUMBER OF ROTATIONS

Syntax `NUMBER OF ROTATIONS` { `=` | `IS` } *n*

n : *no description* (I)

Details Set the number of internal rotations for 2D axisymmetric geometry or 3D geometry with rotation symmetry. The default value is 1

21.1.15 X-Y PLANE SYMMETRY

Details Specifies symmetry about the X-Y plane.

21.1.16 X-Z PLANE SYMMETRY

Details Specifies symmetry about the X-Z plane.

21.1.17 Y-Z PLANE SYMMETRY

Details Specifies symmetry about the Y-Z plane.

21.2 RADIOSITY SOLVER

`Begin RADIOSITY SOLVER` *Boundary condition instance name*

`COUPLING` { `=` | `IS` } *RadCouplingRule*

```
    SOLVER { = | IS } RadSolveRule
    CONVERGENCE TOLERANCE { = | IS } tolerance
    MAXIMUM ITERATIONS { = | IS } M
    OUTPUT RULE { = | IS } OutputRule
End
```

Details Specifies a radiation enclosure. Corresponds to an instance of radiation enclosure mechanics.

21.2.1 COUPLING

Syntax `COUPLING { = | IS }` *RadCouplingRule*

RadCouplingRule : *no description* { `MASON | LAGGED` }

Details Specifies linearization method. Default is MASON.

Enums `RadCouplingRule`

`MASON` - *no description*

`LAGGED` - *no description*

21.2.2 SOLVER

Syntax `SOLVER { = | IS }` *RadSolveRule*

RadSolveRule : *no description* { `CHAPARRAL GMRES | CHAPARRAL CG | COUPLED` }

Details Chaparal solver selection for radiosity system. Default is CHAPARRAL CG.

Enums `RadSolveRule`

`CHAPARRAL GMRES` - *no description*

`CHAPARRAL CG` - *no description*

`COUPLED` - *no description*

21.2.3 CONVERGENCE TOLERANCE

Syntax `CONVERGENCE TOLERANCE { = | IS }` *tolerance*

tolerance : *no description* (R)

Details Sets convergence tolerance. Default value is 1.0e-6.

21.2.4 MAXIMUM ITERATIONS

Syntax `MAXIMUM ITERATIONS { = | IS }` *M*

M : *no description* (I)

Details Sets maximum number of iterations. Default value is 300.

21.2.5 OUTPUT RULE

Syntax `OUTPUT RULE { = | IS }` *OutputRule*

OutputRule : *no description* `{ NONE | SUMMARY | VERBOSE }`

Details Sets the information reporting level. Default is NONE.

Enums `OutputRule`

`NONE` - *no description*
`SUMMARY` - *no description*
`VERBOSE` - *no description*

21.3 ENCLOSURE DEFINITION

```
Begin ENCLOSURE DEFINITION Boundary condition instance name

    ACTIVE PERIODS { = | IS } PeriodNames
    INACTIVE PERIODS { = | IS } PeriodNames
    EMISSIVITY { = | IS } value [ ON surfaceName ]
    EMISSIVITY FUNCTION { = | IS } functionName [ ON surfaceName ]
    EMISSIVITY SUBROUTINE { = | IS } mySub [ ON surfaceName ]
    ADD SURFACE surfaceList
    INTEGRATED POWER OUTPUT variableName
    INTEGRATED FLUX OUTPUT variableName
    NONBLOCKING SURFACES
    BLOCKING SURFACES
    PARTIAL ENCLOSURE AREA { = | IS } A
    PARTIAL ENCLOSURE TEMPERATURE { = | IS } T
```

PARTIAL ENCLOSURE TEMPERATURE TIME FUNCTION { = | IS } *fName*

PARTIAL ENCLOSURE TEMPERATURE SUBROUTINE { = | IS } *fName*

PARTIAL ENCLOSURE EMISSIVITY { = | IS } *E*

PARTIAL ENCLOSURE EMISSIVITY TIME FUNCTION { = | IS } *fName*

PARTIAL ENCLOSURE EMISSIVITY TEMPERATURE FUNCTION { = | IS } *fName*

PARTIAL ENCLOSURE EMISSIVITY SUBROUTINE { = | IS } *fName*

USE VIEWFACTOR CALCULATION

USE VIEWFACTOR SMOOTHING

USE RADIOSITY SOLVER

INPUT DATABASE NAME { = | IS } *filename*

OUTPUT DATABASE NAME { = | IS } *filename* IN *VFfileFormat* FORMAT

DATABASE NAME { = | IS } *filename* IN *VFfileFormat* FORMAT

TOPOLOGY DATABASE NAME { = | IS } *filename*

ROWSUM DATABASE NAME { = | IS } *filename*

RADIOSITY DATABASE NAME { = | IS } *filename*

VIEWFACTOR UPDATE { = | IS } *UpdateMethod* [USING *times*]

End

Details Specifies a radiation enclosure. Corresponds to an instance of radiation enclosure mechanics.

21.3.1 ACTIVE PERIODS

Syntax ACTIVE PERIODS { = | IS } *PeriodNames*

PeriodNames : *no description* (C [, ...])

Details Lists the solution periods during which the given BC, solver, preconditioner, etc. is active. Multiple uses of this line command within a single block will have a cumulative affect.

21.3.2 INACTIVE PERIODS

Syntax INACTIVE PERIODS { = | IS } *PeriodNames*

PeriodNames : *no description* (C [, ...])

Details Lists the solution periods during which the given BC, solver, preconditioner, etc. is inactive. Multiple uses of this line command within a single block will have a cumulative affect.

21.3.3 EMISSIVITY

Syntax `EMISSIVITY { = | IS }` *value* `[ ON` *surfaceName* `]`

value : *no description* (R)

surfaceName : *no description* (C)

Details Sets a constant value of emissivity for a defined surface. If the optional parameters are not included, then this is the default emissivity for the enclosure. Otherwise it is only applied to the indicated surface.

**Note that if a surface is called out more than once, the emissivity definition it is overwritten: last one in wins.

21.3.4 EMISSIVITY FUNCTION

Syntax `EMISSIVITY FUNCTION { = | IS }` *functionName* `[ ON` *surfaceName* `]`

functionName : *no description* (C)

surfaceName : *no description* (C)

Details Sets a emissivity function for a defined surface. If the optional parameters are not included, then this is the default emissivity for the enclosure. Otherwise it is only applied to the indicated surface.

**Note that if a surface is called out more than once, the emissivity definition it is overwritten: last one in wins.

21.3.5 EMISSIVITY SUBROUTINE

Syntax `EMISSIVITY SUBROUTINE { = | IS }` *mySub* `[ ON` *surfaceName* `]`

mySub : *no description* (C)

surfaceName : *no description* (C)

Details Sets a emissivity user subroutine for a defined surface. If the optional parameters are not included, then this is the default emissivity for the enclosure. Otherwise it is only applied to the indicated surface.

Also, the software supports using locally scoped user data for most user subroutines, but I haven't figured out a syntax for it here yet. So it is not yet supported. If you need to get data into this subroutine, use the region's "REAL DATA" and "INTEGER DATA" line commands.

**Note that if a surface is called out more than once, the emissivity definition it is overwritten: last one in wins.

21.3.6 ADD SURFACE

Syntax `ADD SURFACE` *surfaceList*

surfaceList : *no description* (C [, ...])

Details Adds surfaces, by name, to a boundary condition's extent.

21.3.7 INTEGRATED POWER OUTPUT

Syntax `INTEGRATED POWER OUTPUT` *variableName*

variableName : *no description* (C)

Details Calculate the total power associated with this flux boundary condition.

21.3.8 INTEGRATED FLUX OUTPUT

Syntax `INTEGRATED FLUX OUTPUT` *variableName*

variableName : *no description* (C)

Details Calculate the average flux associated with this flux boundary condition.

21.3.9 NONBLOCKING SURFACES

Details Specifies a blocking enclosure

21.3.10 BLOCKING SURFACES

Details Specifies a non-blocking enclosure

21.3.11 PARTIAL ENCLOSURE AREA

Syntax `PARTIAL ENCLOSURE AREA` { = | IS } *A*

A : *no description* (R)

Details Constant value for the partial enclosure area associated with this enclosure radiation flux boundary condition.

21.3.12 PARTIAL ENCLOSURE TEMPERATURE

Syntax `PARTIAL ENCLOSURE TEMPERATURE` { `=` | `IS` } *T*

T : *no description* (R)

Details Constant value for the partial enclosure temperature associated with this enclosure radiation flux boundary condition.

21.3.13 PARTIAL ENCLOSURE TEMPERATURE TIME FUNCTION

Syntax `PARTIAL ENCLOSURE TEMPERATURE TIME FUNCTION` { `=` | `IS` } *fName*

fName : *no description* (C)

Details Time-dependent function name for the partial enclosure temperature associated with this enclosure radiation flux boundary condition.

21.3.14 PARTIAL ENCLOSURE TEMPERATURE SUBROUTINE

Syntax `PARTIAL ENCLOSURE TEMPERATURE SUBROUTINE` { `=` | `IS` } *fName*

fName : *no description* (C)

Details User-defined function name for the partial enclosure temperature associated with this enclosure radiation flux boundary condition.

21.3.15 PARTIAL ENCLOSURE EMISSIVITY

Syntax `PARTIAL ENCLOSURE EMISSIVITY` { `=` | `IS` } *E*

E : *no description* (R)

Details Constant value for the partial enclosure emissivity associated with this enclosure radiation flux boundary condition.

21.3.16 PARTIAL ENCLOSURE EMISSIVITY TIME FUNCTION

Syntax `PARTIAL ENCLOSURE EMISSIVITY TIME FUNCTION` { `=` | `IS` } *fName*

fName : *no description* (C)

Details Time-dependent function name for the partial enclosure emissivity associated with this enclosure radiation flux boundary condition.

21.3.17 PARTIAL ENCLOSURE EMISSIVITY TEMPERATURE FUNCTION

Syntax `PARTIAL ENCLOSURE EMISSIVITY TEMPERATURE FUNCTION` { `=` | `IS` } *fName*

fName : *no description* (C)

Details Temperature-dependent function name for the partial enclosure emissivity associated with this enclosure radiation flux boundary condition.

21.3.18 PARTIAL ENCLOSURE EMISSIVITY SUBROUTINE

Syntax `PARTIAL ENCLOSURE EMISSIVITY SUBROUTINE` { `=` | `IS` } *fName*

fName : *no description* (C)

Details User-defined function name for the partial enclosure emissivity associated with this enclosure radiation flux boundary condition.

21.3.19 USE VIEWFACTOR CALCULATION

Syntax `USE VIEWFACTOR CALCULATION`

unnamed param : *no description* (C)

Details Specifies which view factor calculation to use.

21.3.20 USE VIEWFACTOR SMOOTHING

Syntax `USE VIEWFACTOR SMOOTHING`

unnamed param : *no description* (C)

Details Specifies which view factor smoother to use.

21.3.21 USE RADIOSITY SOLVER

Syntax `USE RADIOSITY SOLVER`

unnamed param : *no description* (C)

Details Specifies which radiosity solver to use.

21.3.22 INPUT DATABASE NAME

Syntax `INPUT DATABASE NAME` { `=` | `IS` } *filename*

filename : *no description* (C)

Details Specifies filename to read viewfactors from

21.3.23 OUTPUT DATABASE NAME

Syntax `OUTPUT DATABASE NAME` { `=` | `IS` } *filename* `IN` *VFfileFormat* `FORMAT`

filename : *no description* (C)

VFfileFormat : *no description* { `ASCII` | `BINARY` }

Details Specify filename to write viewfactors to

Enums `VFfileFormat`

`ASCII` - *no description*

`BINARY` - *no description*

21.3.24 DATABASE NAME

Syntax `DATABASE NAME` { `=` | `IS` } *filename* `IN` *VFfileFormat* `FORMAT`

filename : *no description* (C)

VFfileFormat : *no description* { `ASCII` | `BINARY` }

Details Specifies common filename to read/write viewfactors

Enums `VFfileFormat`

`ASCII` - *no description*

`BINARY` - *no description*

21.3.25 TOPOLOGY DATABASE NAME

Syntax `TOPOLOGY DATABASE NAME` { `=` | `IS` } *filename*

filename : *no description* (C)

Details Specify filename to write enclosure topology to

21.3.26 ROWSUM DATABASE NAME

Syntax `ROWSUM DATABASE NAME` { `=` | `IS` } *filename*

`filename` : *`no description`* `(C)`

Details Specify filename to write enclosure rowsum error results to

21.3.27 RADIOSITY DATABASE NAME

Syntax `RADIOSITY DATABASE NAME` { `=` | `IS` } *filename*

`filename` : *`no description`* `(C)`

Details Specify filename to write enclosure rowsum error and radiosity results to

21.3.28 VIEWFACTOR UPDATE

Syntax `VIEWFACTOR UPDATE` { `=` | `IS` } *UpdateMethod* `[ USING` *times* `]`

`UpdateMethod` : *`no description`* { `STANDARD` | `FREEZE` | `INTERVAL` | `TIMES` }

`times` : *`no description`* `(R [, ...])`

Details Specify the viewfactor re-compute strategy.

Enums `UpdateMethod`

`STANDARD` - *no description*

`FREEZE` - *no description*

`INTERVAL` - *no description*

`TIMES` - *no description*

Chapter 22

IO Reference

22.1 RESTART DATA

```
Begin RESTART DATA Label

    DATABASE NAME { is | = | are } StreamName
    INPUT DATABASE NAME { is | = | are } StreamName
    OUTPUT DATABASE NAME { is | = | are } StreamName
    DATABASE TYPE { is | = | are } DatabaseTypes
    OVERLAY COUNT { is | = | are } count
    CYCLE COUNT { is | = | are } count
    OVERWRITE { is | = } { OFF | ON | TRUE | FALSE | YES | NO }
    ADDITIONAL TIMES { is | = | are } list_of_times
    ADDITIONAL STEPS { is | = | are } list_of_steps
    TIMESTEP ADJUSTMENT INTERVAL { is | = | are } nsteps
    START TIME { is | = | are } start_time
    TERMINATION TIME { is | = | are } final_time
    AT TIME dt { Increment | Interval } { is | = | are } dt
    AT STEP n { Increment | Interval } { is | = | are } m
    OUTPUT ON SIGNAL { is | = | are } signals
    USE OUTPUT SCHEDULER timer_name
End
```

Details Describes the data required to output and input restart data for the enclosing region.

22.1.1 DATABASE NAME

Syntax `DATABASE NAME { is | = | are }` *StreamName*

StreamName : *no description* (C)

Details The database containing the input and/or output restart data. If this analysis is being restarted, restart data will be read from this file. If the analysis is writing restart data, the data will be written to this file. It will be overwritten if it exists (after being read if applicable). If the filename begins with the '/' character, it is an absolute path; otherwise, the path to the current directory will be prepended to the name. See also the 'Input Database' and 'Output Database' commands.

22.1.2 INPUT DATABASE NAME

Syntax `INPUT DATABASE NAME` { `is | = | are` } *StreamName*

`StreamName` : *`no description`* `(C)`

Details The database containing the input restart data. If this analysis is being restarted, restart data will be read from this file. See also the 'Database' and 'Output Database' commands.

22.1.3 OUTPUT DATABASE NAME

Syntax `OUTPUT DATABASE NAME` { `is | = | are` } *StreamName*

`StreamName` : *`no description`* `(C)`

Details The database containing the output restart data. If the analysis is writing restart data, the data will be written to this file. It will be overwritten if it exists. See also the 'Database' and 'Input Database' commands.

22.1.4 DATABASE TYPE

Syntax `DATABASE TYPE` { `is | = | are` } *DatabaseTypes*

`DatabaseTypes` : *`no description`* { `exodusII | SAF | xdmf` }

Details The database type/format used for the restart file.

Enums `DatabaseTypes`

`exodusII` - *no description*

`SAF` - *no description*

`xdmf` - *no description*

22.1.5 OVERLAY COUNT

Syntax `OVERLAY COUNT` { `is | = | are` } *count*

`count` : *`no description`* `(I)`

Details Specify the number of restart outputs which will be overlayed on top of the last written step. For example, if restarts are being output every 0.1 seconds and the overlay count is specified as 2, then restart will write times 0.1 to step 1 of the database. It will then write 0.2 and 0.3 also to step 1. It will then increment the database step and write 0.4 to step 2; overlay 0.5 and 0.6 on step 2... At the end of the analysis, assuming it runs to completion, the database would have times 0.3, 0.6, 0.9, ... However, if there were a problem during the analysis, the last step on the database would contain an intermediate step.

22.1.6 CYCLE COUNT

Syntax `CYCLE COUNT` { `is` | `=` | `are` } *count*

count : *no description* (I)

Details Specify the number of restart steps which will be written to the restart database before previously written steps are overwritten. For example, if the cycle count is 5 and restart is written every 0.1 seconds, the restart system will write 0.1, 0.2, 0.3, 0.4, 0.5 to the database. It will then overwrite the first step with data from time 0.6, the second with time 0.7. At time 0.8, the database would contain data at times 0.6, 0.7, 0.8, 0.4, 0.5. Note that time will not necessarily be monotonically increasing on a database that specifies the cycle count.

22.1.7 OVERWRITE

Details Specify whether the restart database should be overwritten if it exists. The default behavior is to overwrite unless this command is specified in the restart block and either off, false, or no is specified.

22.1.8 ADDITIONAL TIMES

Syntax `ADDITIONAL TIMES` { `is` | `=` | `are` } *list_of_times*

list_of_times : *no description* (R [, ...])

Details Additional simulation times when output should occur.

22.1.9 ADDITIONAL STEPS

Syntax `ADDITIONAL STEPS` { `is` | `=` | `are` } *list_of_steps*

list_of_steps : *no description* (I [, ...])

Details Additional simulation steps when output should occur.

22.1.10 TIMESTEP ADJUSTMENT INTERVAL

Syntax `TIMESTEP ADJUSTMENT INTERVAL` { `is` | `=` | `are` } *nsteps*

nsteps : *no description* (I)

Details Specify the number of steps to 'look ahead' and adjust the timestep to ensure that the specified output times or simulation end time will be hit 'exactly'.

22.1.11 START TIME

Syntax `START TIME` { `is` | `=` | `are` } *start_time*

start_time : *no description* (R)

Details Specify the time to start outputting results from this output request block. This time overrides all 'at time' and 'at step' specifications.

22.1.12 TERMINATION TIME

Syntax `TERMINATION TIME` { `is` | `=` | `are` } *final_time*

final_time : *no description* (R)

Details Specify the time to stop outputting results from this output request block.

22.1.13 AT TIME

Syntax `AT TIME` *dt* { `Increment` | `Interval` } { `is` | `=` | `are` } *dt*

dt : *no description* (R)

dt : *no description* (R)

Details Specify an output interval in terms of the internal simulation time. The first time specifies the time at the beginning of this time interval and the second time specifies the output frequency to be used within this interval.

22.1.14 AT STEP

Syntax `AT STEP` *n* { `Increment` | `Interval` } { `is` | `=` | `are` } *m*

n : *no description* (I)

m : *no description* (I)

Details Specify an output interval in terms of the internal iteration step count. The first step specifies the step count at the beginning of this interval and the second step specifies the output frequency to be used within this interval.

22.1.15 OUTPUT ON SIGNAL

Syntax `OUTPUT ON SIGNAL` { `is` | `=` | `are` } *signals*

```
signals : no description { SIGALRM | SIGFPE | SIGHUP | SIGINT |
    SIGPIPE | SIGQUIT | SIGTERM | SIGUSR1 | SIGUSR2 | SIGABRT |
    SIGKILL | SIGILL | SIGSEGV }
```

Details When the specified signal is raised, the output stream associated with this block will be output.

Enums `signals`

`SIGALRM` - *no description*

`SIGFPE` - *no description*

`SIGHUP` - *no description*

`SIGINT` - *no description*

`SIGPIPE` - *no description*

`SIGQUIT` - *no description*

`SIGTERM` - *no description*

`SIGUSR1` - *no description*

`SIGUSR2` - *no description*

`SIGABRT` - *no description*

`SIGKILL` - *no description*

`SIGILL` - *no description*

`SIGSEGV` - *no description*

22.1.16 USE OUTPUT SCHEDULER

Syntax `USE OUTPUT SCHEDULER` *timer_name*

```
timer_name : no description (C)
```

Details Associates a predefined output scheduler with this output block (results, restart, heartbeat, or history).

22.2 RESULTS OUTPUT

```
Begin RESULTS OUTPUT Label
```

```
    DATABASE NAME { is | = | are } StreamName
    DATABASE TYPE { is | = | are } DatabaseTypes
    TITLE
    GLOBAL VARIABLES { is | = | are } [ variable_list ]
    NODE VARIABLES { is | = | are } [ variable_list ]
    NODAL VARIABLES { is | = | are } [ variable_list ]
    ELEMENT VARIABLES { is | = | are } [ variable_list ]
    OUTPUT MESH { is | = } OutputMesh
    EDGE VARIABLES { is | = | are } [ variable_list ]
    FACE VARIABLES { is | = | are } [ variable_list ]
    NODESET VARIABLES { is | = | are } [ variable_list ]
    COMPONENT SEPARATOR CHARACTER { is | = } separator
    ADDITIONAL TIMES { is | = | are } list_of_times
    ADDITIONAL STEPS { is | = | are } list_of_steps
    TIMESTEP ADJUSTMENT INTERVAL { is | = | are } nsteps
    START TIME { is | = | are } start_time
    TERMINATION TIME { is | = | are } final_time
    AT TIME dt { Increment | Interval } { is | = | are } dt
    AT STEP n { Increment | Interval } { is | = | are } m
    OUTPUT ON SIGNAL { is | = | are } signals
    USE OUTPUT SCHEDULER timer_name
    OVERWRITE { is | = } { OFF | ON | TRUE | FALSE | YES | NO }
End
```

Details Describes the location and type of the output stream used for outputting results for the enclosing region.

22.2.1 DATABASE NAME

Syntax `DATABASE NAME { is | = | are }` *StreamName*

StreamName : *no description* (C)

Details The base name of the database containing the output results. If the filename begins with the '/' character, it is an absolute path; otherwise, the path to the current directory will be prepended to the name.

22.2.2 DATABASE TYPE

Syntax `DATABASE TYPE` { `is` | `=` | `are` } *DatabaseTypes*

DatabaseTypes : *no description* { `exodusII` | `SAF` | `xdmf` }

Details The database type/format to be used for the output results.

Enums `DatabaseTypes`

`exodusII` - *no description*

`SAF` - *no description*

`xdmf` - *no description*

22.2.3 TITLE

Syntax `TITLE` *the_title*

the_title : *no description*

Details Specify the title to be used for this specific output block.

22.2.4 GLOBAL VARIABLES

Syntax `GLOBAL VARIABLES` { `is` | `=` | `are` } `[` *variable_list* `]`

variable_list : *no description* `(C [, ...])`

Details Define the global variables that should be written to the results database. If "variable" is entered, then its name will be used on the output database. If "variable as db_name" is entered, then "db_name" will be the name used on the database for the internal variable "variable". Multiple "variable" or "variable as db_name" entries are allowed on the same line.

22.2.5 NODE VARIABLES

Syntax `NODE VARIABLES` { `is` | `=` | `are` } `[` *variable_list* `]`

variable_list : *no description* `(C [, ...])`

Details Define the nodal variables that should be written to the results database. If "variable" is entered, then its name will be used on the output database. If "variable as db_name" is entered, then "db_name" will be the name used on the database for the internal variable "variable". Multiple "variable" or "variable as db_name" entries are allowed on the same line.

22.2.6 NODAL VARIABLES

Syntax `NODAL VARIABLES` { `is` | `=` | `are` } `[` *variable_list* `]`

`variable_list` : *`no description`* `(C [, ...])`

Details Define the nodal variables that should be written to the results database. If "variable" is entered, then its name will be used on the output database. If "variable as db_name" is entered, then "db_name" will be the name used on the database for the internal variable "variable". Multiple "variable" or "variable as db_name" entries are allowed on the same line.

22.2.7 ELEMENT VARIABLES

Syntax `ELEMENT VARIABLES` { `is` | `=` | `are` } `[` *variable_list* `]`

`variable_list` : *`no description`* `(C [, ...])`

Details Define the variables that should be written to the results database. If "variable" is entered, then its name will be used on the output database. If "variable as db_name" is entered, then "db_name" will be the name used on the database for the internal variable "variable". Multiple "variable" or "variable as db_name" entries are allowed on the same line. The entities that this variable are written to can also be limited or specified with "exclude list_of_entities" or "include list_of_entities"

22.2.8 OUTPUT MESH

Syntax `OUTPUT MESH` { `is` | `=` } *OutputMesh*

`OutputMesh` : *`no description`* { `refined` | `unrefined` | `block surface` | `exposed surface` }

Details Use this command to turn on "unrefined" as the output mesh. The default behavior is "refined", in which field variables are output on the current mesh, which may have been refined (either uniformly or adaptively) or had its topology altered in some way (e.g., dynamic load balancing) with respect to the original mesh read from the the input file. By specifying "Output Mesh = unrefined", all output variables are output only on the original mesh objects read from the input file.

Enums `OutputMesh`

- `refined` - *no description*
- `unrefined` - *no description*
- `block surface` - *no description*
- `exposed surface` - *no description*

22.2.9 EDGE VARIABLES

Syntax `EDGE VARIABLES { is | = | are } [` *variable_list* `]`

`variable_list` `:` *`no description`* `(C [, ...])`

Details Define the variables that should be written to the results database. If "variable" is entered, then its name will be used on the output database. If "variable as db_name" is entered, then "db_name" will be the name used on the database for the internal variable "variable". Multiple "variable" or "variable as db_name" entries are allowed on the same line. The entities that this variable are written to can also be limited or specified with "exclude list_of_entities" or "include list_of_entities". Edge variables are not supported for all database types.

22.2.10 FACE VARIABLES

Syntax `FACE VARIABLES { is | = | are } [` *variable_list* `]`

`variable_list` `:` *`no description`* `(C [, ...])`

Details Define the variables that should be written to the results database. If "variable" is entered, then its name will be used on the output database. If "variable as db_name" is entered, then "db_name" will be the name used on the database for the internal variable "variable". Multiple "variable" or "variable as db_name" entries are allowed on the same line. The entities that this variable are written to can also be limited or specified with "exclude list_of_entities" or "include list_of_entities". Face variables are not supported for all database types.

22.2.11 NODESET VARIABLES

Syntax `NODESET VARIABLES { is | = | are } [` *variable_list* `]`

`variable_list` `:` *`no description`* `(C [, ...])`

Details Define the variables that should be written to the results database. If "variable" is entered, then its name will be used on the output database. If "variable as db_name" is entered, then "db_name" will be the name used on the database for the internal variable "variable". Multiple "variable" or "variable as db_name" entries are allowed on the same line. The entities that this variable are written to can also be limited or specified with "exclude list_of_entities" or "include list_of_entities". Nodeset variables are not supported for all database types.

22.2.12 COMPONENT SEPARATOR CHARACTER

Syntax `COMPONENT SEPARATOR CHARACTER { is | = }` *separator*

`separator` `:` *`no description`* `(C)`

Details The separator is the single character used to separate the output variable basename (e.g. "stress") from the suffices (e.g. "xx", "yy") when displaying the names of the individual variable components. For example, the default separator is "_", which results in names similar to "stress_xx", "stress_yy", ... "stress_zx". To eliminate the separator, specify an empty string ("") or NONE.

22.2.13 ADDITIONAL TIMES

Syntax `ADDITIONAL TIMES { is | = | are }` *list_of_times*

list_of_times : *no description* (R [, ...])

Details Additional simulation times when output should occur.

22.2.14 ADDITIONAL STEPS

Syntax `ADDITIONAL STEPS { is | = | are }` *list_of_steps*

list_of_steps : *no description* (I [, ...])

Details Additional simulation steps when output should occur.

22.2.15 TIMESTEP ADJUSTMENT INTERVAL

Syntax `TIMESTEP ADJUSTMENT INTERVAL { is | = | are }` *nsteps*

nsteps : *no description* (I)

Details Specify the number of steps to 'look ahead' and adjust the timestep to ensure that the specified output times or simulation end time will be hit 'exactly'.

22.2.16 START TIME

Syntax `START TIME { is | = | are }` *start_time*

start_time : *no description* (R)

Details Specify the time to start outputting results from this output request block. This time overrides all 'at time' and 'at step' specifications.

22.2.17 TERMINATION TIME

Syntax `TERMINATION TIME { is | = | are }` *final_time*

final_time : *no description* (R)

Details Specify the time to stop outputting results from this output request block.

22.2.18 AT TIME

Syntax `AT TIME` *dt* { `Increment | Interval` } { `is | = | are` } *dt*

dt : *no description* (R)

dt : *no description* (R)

Details Specify an output interval in terms of the internal simulation time. The first time specifies the time at the beginning of this time interval and the second time specifies the output frequency to be used within this interval.

22.2.19 AT STEP

Syntax `AT STEP` *n* { `Increment | Interval` } { `is | = | are` } *m*

n : *no description* (I)

m : *no description* (I)

Details Specify an output interval in terms of the internal iteration step count. The first step specifies the step count at the beginning of this interval and the second step specifies the output frequency to be used within this interval.

22.2.20 OUTPUT ON SIGNAL

Syntax `OUTPUT ON SIGNAL` { `is | = | are` } *signals*

signals : *no description* { `SIGALRM | SIGFPE | SIGHUP | SIGINT | SIGPIPE | SIGQUIT | SIGTERM | SIGUSR1 | SIGUSR2 | SIGABRT | SIGKILL | SIGILL | SIGSEGV` }

Details When the specified signal is raised, the output stream associated with this block will be output.

Enums signals

SIGALRM - *no description*

SIGFPE - *no description*

SIGHUP - *no description*

SIGINT - *no description*

SIGPIPE - *no description*

SIGQUIT - *no description*

SIGTERM - *no description*

SIGUSR1 - *no description*

SIGUSR2 - *no description*

SIGABRT - *no description*

SIGKILL - *no description*

SIGILL - *no description*

SIGSEGV - *no description*

22.2.21 USE OUTPUT SCHEDULER

Syntax `USE OUTPUT SCHEDULER` *timer_name*

timer_name : *no description* (C)

Details Associates a predefined output scheduler with this output block (results, restart, heartbeat, or history).

22.2.22 OVERWRITE

Details Specify whether the database should be overwritten if it exists. The default behavior is to overwrite unless this command is specified in the output block and either off, false, or no is specified.

22.3 HEARTBEAT

`Begin HEARTBEAT` *Label*

`ADDITIONAL TIMES` { `is` | `=` | `are` } *list_of_times*

`ADDITIONAL STEPS` { `is` | `=` | `are` } *list_of_steps*

`TIMESTEP ADJUSTMENT INTERVAL` { `is` | `=` | `are` } *nsteps*

`START TIME` { `is` | `=` | `are` } *start_time*

`TERMINATION TIME` { `is` | `=` | `are` } *final_time*

`AT TIME` *dt* { `Increment` | `Interval` } { `is` | `=` | `are` } *dt*

AT STEP n { Increment | Interval } { is | = | are } m

OUTPUT ON SIGNAL { is | = | are } *signals*

USE OUTPUT SCHEDULER *timer_name*

STREAM NAME { is | = | are } *StreamName*

PRECISION { is | = | are } *precision*

LABELS { is | = | are } { on | off }

MONITOR { is | = | are | the } { results | restart | history }

TIMESTAMP FORMAT

LEGEND { is | = | are } { on | off }

VARIABLE { is | = | are } { global | node | nodal | element | face | edge } [*variable_list*]

End

Details Describes the location and type of the output stream used for outputting the heartbeat information for the enclosing region.

22.3.1 ADDITIONAL TIMES

Syntax `ADDITIONAL TIMES { is | = | are }` *list_of_times*

list_of_times : *no description* (R [, ...])

Details Additional simulation times when output should occur.

22.3.2 ADDITIONAL STEPS

Syntax `ADDITIONAL STEPS { is | = | are }` *list_of_steps*

list_of_steps : *no description* (I [, ...])

Details Additional simulation steps when output should occur.

22.3.3 TIMESTEP ADJUSTMENT INTERVAL

Syntax `TIMESTEP ADJUSTMENT INTERVAL { is | = | are }` *nsteps*

nsteps : *no description* (I)

Details Specify the number of steps to 'look ahead' and adjust the timestep to ensure that the specified output times or simulation end time will be hit 'exactly'.

22.3.4 START TIME

Syntax `START TIME` { `is` | `=` | `are` } *start_time*

start_time : *no description* (R)

Details Specify the time to start outputting results from this output request block. This time overrides all 'at time' and 'at step' specifications.

22.3.5 TERMINATION TIME

Syntax `TERMINATION TIME` { `is` | `=` | `are` } *final_time*

final_time : *no description* (R)

Details Specify the time to stop outputting results from this output request block.

22.3.6 AT TIME

Syntax `AT TIME` *dt* { `Increment` | `Interval` } { `is` | `=` | `are` } *dt*

dt : *no description* (R)

dt : *no description* (R)

Details Specify an output interval in terms of the internal simulation time. The first time specifies the time at the beginning of this time interval and the second time specifies the output frequency to be used within this interval.

22.3.7 AT STEP

Syntax `AT STEP` *n* { `Increment` | `Interval` } { `is` | `=` | `are` } *m*

n : *no description* (I)

m : *no description* (I)

Details Specify an output interval in terms of the internal iteration step count. The first step specifies the step count at the beginning of this interval and the second step specifies the output frequency to be used within this interval.

22.3.8 OUTPUT ON SIGNAL

Syntax `OUTPUT ON SIGNAL` { `is` | `=` | `are` } *signals*

signals : *no description* { `SIGALRM` | `SIGFPE` | `SIGHUP` | `SIGINT` | `SIGPIPE` | `SIGQUIT` | `SIGTERM` | `SIGUSR1` | `SIGUSR2` | `SIGABRT` | `SIGKILL` | `SIGILL` | `SIGSEGV` }

Details When the specified signal is raised, the output stream associated with this block will be output.

Enums `signals`

- `SIGALRM` - *no description*
- `SIGFPE` - *no description*
- `SIGHUP` - *no description*
- `SIGINT` - *no description*
- `SIGPIPE` - *no description*
- `SIGQUIT` - *no description*
- `SIGTERM` - *no description*
- `SIGUSR1` - *no description*
- `SIGUSR2` - *no description*
- `SIGABRT` - *no description*
- `SIGKILL` - *no description*
- `SIGILL` - *no description*
- `SIGSEGV` - *no description*

22.3.9 USE OUTPUT SCHEDULER

Syntax `USE OUTPUT SCHEDULER` *timer_name*

timer_name : *no description* (C)

Details Associates a predefined output scheduler with this output block (results, restart, heartbeat, or history).

22.3.10 STREAM NAME

Syntax `STREAM NAME { is | = | are }` *StreamName*

StreamName : *no description* (C)

Details The base name of the stream containing the output results. If the filename begins with the '/' character, it is an absolute path; otherwise, the path to the current directory will be prepended to the name. In addition, 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. If the file already exists, it is overwritten.

22.3.11 PRECISION

Syntax `PRECISION` { `is` | `=` | `are` } *precision*

precision : *no description* (I)

Details The precsion to be used for the output of real variables.

22.3.12 LABELS

Details Specifies whether labels will be displayed or just the value of the variable. Labels will be shown if this line is not present.

22.3.13 MONITOR

Details Specifies whether a line will be written to the heartbeat stream when either the results, history, and/or restart data are output.

22.3.14 TIMESTAMP FORMAT

Syntax `TIMESTAMP FORMAT` *FormatString*

FormatString : *no description*

Details The format to be used for the timestamp. See 'man strftime' for more information.

22.3.15 LEGEND

Details Specifies whether a legend will be displayed prior to outputting any variables. The legend will not be shown unless this line is present. The legend shows the names of the variables that will be written to the heartbeat output stream. If the variable has multiple components, then the component count is shown after the variable e.g., velocity(3).

22.3.16 VARIABLE

Syntax `VARIABLE` { `is` | `=` | `are` } { `global` | `node` | `nodal` | `element` | `face` | `edge` } [*variable_list*]

variable_list : *no description* (C [, ...])

Details

Define the variables that should be written to the heartbeat output. The user can request that the values of certain variables be output on the heartbeat line. These variables are limited to region and framework control data currently. The syntax is:

```
variable = {entity_type} {internal_name} at
           {entity_type} {entity_id}     as {external_name}
variable = {entity_type} {internal_name} nearest location
           {x,y,z} as {external_name}
```

For global variables, use:

```
variable = global {internal_name} [as {external_name}]
```

Where:

```
entity_type = node, element, face, edge, global
internal_name = Sierra variable name
entity_id = id of the node, element, face, edge that you want
    the specified variable output at.
external_name = name of variable on the database.
```

The names 'timestep', and 'time' can be specified as variables also. They are the current timestep and simulation time. This line can appear multiple times.

22.4 HISTORY OUTPUT

`Begin HISTORY OUTPUT` *Label*

`ADDITIONAL TIMES` { `is` | `=` | `are` } *list_of_times*

`ADDITIONAL STEPS` { `is` | `=` | `are` } *list_of_steps*

`TIMESTEP ADJUSTMENT INTERVAL` { `is` | `=` | `are` } *nsteps*

`START TIME` { `is` | `=` | `are` } *start_time*

`TERMINATION TIME` { `is` | `=` | `are` } *final_time*

`AT TIME` *dt* { `Increment` | `Interval` } { `is` | `=` | `are` } *dt*

`AT STEP` *n* { `Increment` | `Interval` } { `is` | `=` | `are` } *m*

`OUTPUT ON SIGNAL` { `is` | `=` | `are` } *signals*

`USE OUTPUT SCHEDULER` *timer_name*

`OVERWRITE` { `is` | `=` } { `OFF` | `ON` | `TRUE` | `FALSE` | `YES` | `NO` }

`DATABASE NAME` { `is` | `=` | `are` } *StreamName*

`DATABASE TYPE` { `is` | `=` | `are` } *DatabaseTypes*

`TITLE`

VARIABLE { is | = | are } { global | node | nodal | element | face | edge } [*variable_list*]

End

Details Describes the location and type of the output stream used for outputting history for the enclosing region.

22.4.1 ADDITIONAL TIMES

Syntax `ADDITIONAL TIMES { is | = | are }` *list_of_times*

list_of_times : *no description* (R [, ...])

Details Additional simulation times when output should occur.

22.4.2 ADDITIONAL STEPS

Syntax `ADDITIONAL STEPS { is | = | are }` *list_of_steps*

list_of_steps : *no description* (I [, ...])

Details Additional simulation steps when output should occur.

22.4.3 TIMESTEP ADJUSTMENT INTERVAL

Syntax `TIMESTEP ADJUSTMENT INTERVAL { is | = | are }` *nsteps*

nsteps : *no description* (I)

Details Specify the number of steps to 'look ahead' and adjust the timestep to ensure that the specified output times or simulation end time will be hit 'exactly'.

22.4.4 START TIME

Syntax `START TIME { is | = | are }` *start_time*

start_time : *no description* (R)

Details Specify the time to start outputting results from this output request block. This time overrides all 'at time' and 'at step' specifications.

22.4.5 TERMINATION TIME

Syntax `TERMINATION TIME { is | = | are }` *final_time*

final_time : *no description* (R)

Details Specify the time to stop outputting results from this output request block.

22.4.6 AT TIME

Syntax `AT TIME` *dt* `{ Increment | Interval } { is | = | are }` *dt*

dt : *no description* (R)

dt : *no description* (R)

Details Specify an output interval in terms of the internal simulation time. The first time specifies the time at the beginning of this time interval and the second time specifies the output frequency to be used within this interval.

22.4.7 AT STEP

Syntax `AT STEP` *n* `{ Increment | Interval } { is | = | are }` *m*

n : *no description* (I)

m : *no description* (I)

Details Specify an output interval in terms of the internal iteration step count. The first step specifies the step count at the beginning of this interval and the second step specifies the output frequency to be used within this interval.

22.4.8 OUTPUT ON SIGNAL

Syntax `OUTPUT ON SIGNAL { is | = | are }` *signals*

signals : *no description* { SIGALRM | SIGFPE | SIGHUP | SIGINT | SIGPIPE | SIGQUIT | SIGTERM | SIGUSR1 | SIGUSR2 | SIGABRT | SIGKILL | SIGILL | SIGSEGV }

Details When the specified signal is raised, the output stream associated with this block will be output.

Enums `signals`

- `SIGALRM` - *no description*
- `SIGFPE` - *no description*
- `SIGHUP` - *no description*
- `SIGINT` - *no description*
- `SIGPIPE` - *no description*
- `SIGQUIT` - *no description*
- `SIGTERM` - *no description*
- `SIGUSR1` - *no description*
- `SIGUSR2` - *no description*
- `SIGABRT` - *no description*
- `SIGKILL` - *no description*
- `SIGILL` - *no description*
- `SIGSEGV` - *no description*

22.4.9 USE OUTPUT SCHEDULER

Syntax `USE OUTPUT SCHEDULER` *timer_name*

timer_name : *no description* (C)

Details Associates a predefined output scheduler with this output block (results, restart, heartbeat, or history).

22.4.10 OVERWRITE

Details Specify whether the database should be overwritten if it exists. The default behavior is to overwrite unless this command is specified in the output block and either off, false, or no is specified.

22.4.11 DATABASE NAME

Syntax `DATABASE NAME` { `is` | `=` | `are` } *StreamName*

StreamName : *no description* (C)

Details The base name of the database containing the output history. If the filename begins with the '/' character, it is an absolute path; otherwise, the path to the current directory will be prepended to the name.

22.4.12 DATABASE TYPE

Syntax `DATABASE TYPE` { `is` | `=` | `are` } *DatabaseTypes*

`DatabaseTypes` `:` *`no description`* { `exodusII` | `SAF` | `xdmf` }

Details The database type/format to be used for the output history.

Enums `DatabaseTypes`

`exodusII` - *no description*
`SAF` - *no description*
`xdmf` - *no description*

22.4.13 TITLE

Syntax `TITLE` *the_title*

`the_title` `:` *`no description`*

Details Specify the title to be used for this specific output block.

22.4.14 VARIABLE

Syntax `VARIABLE` { `is` | `=` | `are` } { `global` | `node` | `nodal` | `element` | `face` | `edge` } `[` *variable_list* `]`

`variable_list` `:` *`no description`* `(C [, ...])`

Details Define the variables that should be written to the history database. The syntax is:

```
variable = entity {internal_name} at entity {id} as {DBname}
```

or

```
variable = entity {internal_name} nearest location X, Y, Z as {DBname}
```

or

```
variable = entity {internal_name} at location X, Y, Z as {DBname}.
```

Where {entity} is 'node', 'element', 'face', or 'edge'; {internal_name} is the name of the variable in the Sierra application; and {DBname} is the name as it should appear on the history database.

22.5 RESULTS OUTPUT

```
Begin RESULTS OUTPUT Label

    DATABASE NAME { is | = | are } StreamName
    DATABASE TYPE { is | = | are } DatabaseTypes
    TITLE
    GLOBAL VARIABLES { is | = | are } [ variable_list ]
    NODE VARIABLES { is | = | are } [ variable_list ]
    NODAL VARIABLES { is | = | are } [ variable_list ]
    ELEMENT VARIABLES { is | = | are } [ variable_list ]
    OUTPUT MESH { is | = } OutputMesh
    EDGE VARIABLES { is | = | are } [ variable_list ]
    FACE VARIABLES { is | = | are } [ variable_list ]
    NODESET VARIABLES { is | = | are } [ variable_list ]
    COMPONENT SEPARATOR CHARACTER { is | = } separator
    ADDITIONAL TIMES { is | = | are } list_of_times
    ADDITIONAL STEPS { is | = | are } list_of_steps
    TIMESTEP ADJUSTMENT INTERVAL { is | = | are } nsteps
    START TIME { is | = | are } start_time
    TERMINATION TIME { is | = | are } final_time
    AT TIME dt { Increment | Interval } { is | = | are } dt
    AT STEP n { Increment | Interval } { is | = | are } m
    OUTPUT ON SIGNAL { is | = | are } signals
    USE OUTPUT SCHEDULER timer_name
    OVERWRITE { is | = } { OFF | ON | TRUE | FALSE | YES | NO }

End
```

Details Describes the location and type of the output stream used for outputting results for the enclosing region.

22.5.1 DATABASE NAME

Syntax `DATABASE NAME { is | = | are }` *StreamName*

StreamName : *no description* (C)

Details The base name of the database containing the output results. If the filename begins with the '/' character, it is an absolute path; otherwise, the path to the current directory will be prepended to the name.

22.5.2 DATABASE TYPE

Syntax `DATABASE TYPE` { `is | = | are` } *DatabaseTypes*

DatabaseTypes : *no description* { `exodusII | SAF | xdmf` }

Details The database type/format to be used for the output results.

Enums `DatabaseTypes`

- `exodusII` - *no description*
- `SAF` - *no description*
- `xdmf` - *no description*

22.5.3 TITLE

Syntax `TITLE` *the_title*

the_title : *no description*

Details Specify the title to be used for this specific output block.

22.5.4 GLOBAL VARIABLES

Syntax `GLOBAL VARIABLES` { `is | = | are` } `[` *variable_list* `]`

variable_list : *no description* `(C [, ...])`

Details Define the global variables that should be written to the results database. If "variable" is entered, then its name will be used on the output database. If "variable as db_name" is entered, then "db_name" will be the name used on the database for the internal variable "variable". Multiple "variable" or "variable as db_name" entries are allowed on the same line.

22.5.5 NODE VARIABLES

Syntax `NODE VARIABLES` { `is | = | are` } `[` *variable_list* `]`

variable_list : *no description* `(C [, ...])`

Details Define the nodal variables that should be written to the results database. If "variable" is entered, then its name will be used on the output database. If "variable as db_name" is entered, then "db_name" will be the name used on the database for the internal variable "variable". Multiple "variable" or "variable as db_name" entries are allowed on the same line.

22.5.6 NODAL VARIABLES

Syntax `NODAL VARIABLES { is | = | are } [` *variable_list* `]`

variable_list : *no description* (C [, ...])

Details Define the nodal variables that should be written to the results database. If "variable" is entered, then its name will be used on the output database. If "variable as db_name" is entered, then "db_name" will be the name used on the database for the internal variable "variable". Multiple "variable" or "variable as db_name" entries are allowed on the same line.

22.5.7 ELEMENT VARIABLES

Syntax `ELEMENT VARIABLES { is | = | are } [` *variable_list* `]`

variable_list : *no description* (C [, ...])

Details Define the variables that should be written to the results database. If "variable" is entered, then its name will be used on the output database. If "variable as db_name" is entered, then "db_name" will be the name used on the database for the internal variable "variable". Multiple "variable" or "variable as db_name" entries are allowed on the same line. The entities that this variable are written to can also be limited or specified with "exclude list_of_entities" or "include list_of_entities"

22.5.8 OUTPUT MESH

Syntax `OUTPUT MESH { is | = }` *OutputMesh*

OutputMesh : *no description* `{ refined | unrefined | block surface | exposed surface }`

Details Use this command to turn on "unrefined" as the output mesh. The default behavior is "refined", in which field variables are output on the current mesh, which may have been refined (either uniformly or adaptively) or had its topology altered in some way (e.g., dynamic load balancing) with respect to the original mesh read from the the input file. By specifying "Output Mesh = unrefined", all output variables are output only on the original mesh objects read from the input file.

Enums `OutputMesh`

- `refined` - *no description*
- `unrefined` - *no description*
- `block surface` - *no description*
- `exposed surface` - *no description*

22.5.9 EDGE VARIABLES

Syntax `EDGE VARIABLES` { `is` | `=` | `are` } [*variable_list*]

variable_list : *no description* (C [, ...])

Details Define the variables that should be written to the results database. If "variable" is entered, then its name will be used on the output database. If "variable as db_name" is entered, then "db_name" will be the name used on the database for the internal variable "variable". Multiple "variable" or "variable as db_name" entries are allowed on the same line. The entities that this variable are written to can also be limited or specified with "exclude list_of_entities" or "include list_of_entities". Edge variables are not supported for all database types.

22.5.10 FACE VARIABLES

Syntax `FACE VARIABLES` { `is` | `=` | `are` } [*variable_list*]

variable_list : *no description* (C [, ...])

Details Define the variables that should be written to the results database. If "variable" is entered, then its name will be used on the output database. If "variable as db_name" is entered, then "db_name" will be the name used on the database for the internal variable "variable". Multiple "variable" or "variable as db_name" entries are allowed on the same line. The entities that this variable are written to can also be limited or specified with "exclude list_of_entities" or "include list_of_entities". Face variables are not supported for all database types.

22.5.11 NODESET VARIABLES

Syntax `NODESET VARIABLES` { `is` | `=` | `are` } [*variable_list*]

variable_list : *no description* (C [, ...])

Details Define the variables that should be written to the results database. If "variable" is entered, then its name will be used on the output database. If "variable as db_name" is entered, then "db_name" will be the name used on the database for the internal variable "variable". Multiple "variable" or "variable as db_name" entries are allowed on the same line. The entities that this variable are written to can also be limited or specified with "exclude list_of_entities" or "include list_of_entities". Nodeset variables are not supported for all database types.

22.5.12 COMPONENT SEPARATOR CHARACTER

Syntax `COMPONENT SEPARATOR CHARACTER` { `is` | `=` } *separator*

separator : *no description* (C)

Details The separator is the single character used to separate the output variable basename (e.g. "stress") from the suffices (e.g. "xx", "yy") when displaying the names of the individual variable components. For example, the default separator is "_", which results in names similar to "stress_xx", "stress_yy", ... "stress_zx". To eliminate the separator, specify an empty string ("") or NONE.

22.5.13 ADDITIONAL TIMES

Syntax `ADDITIONAL TIMES { is | = | are }` *list_of_times*

`list_of_times : no description (R [, ...])`

Details Additional simulation times when output should occur.

22.5.14 ADDITIONAL STEPS

Syntax `ADDITIONAL STEPS { is | = | are }` *list_of_steps*

`list_of_steps : no description (I [, ...])`

Details Additional simulation steps when output should occur.

22.5.15 TIMESTEP ADJUSTMENT INTERVAL

Syntax `TIMESTEP ADJUSTMENT INTERVAL { is | = | are }` *nsteps*

`nsteps : no description (I)`

Details Specify the number of steps to 'look ahead' and adjust the timestep to ensure that the specified output times or simulation end time will be hit 'exactly'.

22.5.16 START TIME

Syntax `START TIME { is | = | are }` *start_time*

`start_time : no description (R)`

Details Specify the time to start outputting results from this output request block. This time overrides all 'at time' and 'at step' specifications.

22.5.17 TERMINATION TIME

Syntax `TERMINATION TIME { is | = | are }` *final_time*

`final_time : no description (R)`

Details Specify the time to stop outputting results from this output request block.

22.5.18 AT TIME

Syntax `AT TIME` *dt* { `Increment | Interval` } { `is | = | are` } *dt*

dt : *no description* (R)

dt : *no description* (R)

Details Specify an output interval in terms of the internal simulation time. The first time specifies the time at the beginning of this time interval and the second time specifies the output frequency to be used within this interval.

22.5.19 AT STEP

Syntax `AT STEP` *n* { `Increment | Interval` } { `is | = | are` } *m*

n : *no description* (I)

m : *no description* (I)

Details Specify an output interval in terms of the internal iteration step count. The first step specifies the step count at the beginning of this interval and the second step specifies the output frequency to be used within this interval.

22.5.20 OUTPUT ON SIGNAL

Syntax `OUTPUT ON SIGNAL` { `is | = | are` } *signals*

signals : *no description* { `SIGALRM | SIGFPE | SIGHUP | SIGINT | SIGPIPE | SIGQUIT | SIGTERM | SIGUSR1 | SIGUSR2 | SIGABRT | SIGKILL | SIGILL | SIGSEGV` }

Details When the specified signal is raised, the output stream associated with this block will be output.

Enums signals

- `SIGALRM` - *no description*
- `SIGFPE` - *no description*
- `SIGHUP` - *no description*
- `SIGINT` - *no description*
- `SIGPIPE` - *no description*
- `SIGQUIT` - *no description*
- `SIGTERM` - *no description*
- `SIGUSR1` - *no description*
- `SIGUSR2` - *no description*
- `SIGABRT` - *no description*
- `SIGKILL` - *no description*
- `SIGILL` - *no description*
- `SIGSEGV` - *no description*

22.5.21 USE OUTPUT SCHEDULER

Syntax `USE OUTPUT SCHEDULER` *timer_name*

timer_name : *no description* (C)

Details Associates a predefined output scheduler with this output block (results, restart, heartbeat, or history).

22.5.22 OVERWRITE

Details Specify whether the database should be overwritten if it exists. The default behavior is to overwrite unless this command is specified in the output block and either off, false, or no is specified.

22.6 PARAMETERS FOR BLOCK

```
Begin PARAMETERS FOR BLOCK Blockname

    ACTIVE FOR PROCEDURE procedureName during periods periodNames
    INACTIVE FOR PROCEDURE procedureName during periods periodNames
    MATERIAL MatName
    PHASE Phase_Label { = | IS | ARE } Material_Name
    DEACTIVATE CodeName
    SHELL INTEGRATION POINTS { = | IS | ARE } npoints
    SHELL SCALE THICKNESS { = | IS | ARE } tscale
    HOURGLASS { Stiffness | Viscosity } { = | IS | ARE } hgval
    MEMBRANE SCALE THICKNESS { = | IS | ARE } tscale
    LINEAR BULK VISCOSITY { = | IS | ARE } lbv
    QUADRATIC BULK VISCOSITY { = | IS | ARE } qbv
    SOLID MECHANICS USE MODEL modelname
    DEPOSIT SPECIFIC INTERNAL ENERGY edep [ OVER TIME tdep STARTING AT TIME tinit ]
    ELEMENT NUMERICAL FORMULATION { = | IS | ARE } { Old | New }
    SHELL INTEGRATION SCHEME { = | IS | ARE } { Gauss | Lobatto | Trapezoid }
    LOFTING FACTOR { = | IS | ARE } tscale
    DEVIATORIC PARAMETER { = | IS | ARE } alpha
    TRUSS AREA { = | IS | ARE } TrussArea
    DAMPER AREA { = | IS | ARE } DamperArea
    ELEMENT INITIAL STRAINXX { = | IS | ARE } istrainxx
    ELEMENT INITIAL STRAINYY { = | IS | ARE } istrainyy
    ELEMENT INITIAL STRAINZZ { = | IS | ARE } istrainzz
    ELEMENT INITIAL STRAINXY { = | IS | ARE } istrainxy
    ELEMENT INITIAL STRAINXZ { = | IS | ARE } istrainxz
    ELEMENT INITIAL STRAINYZ { = | IS | ARE } istrainyz
    ELEMENT COORDINATE SYSTEM { = | IS | ARE } { rectangular | cylindrical | spherical }
    POINT AX { = | IS | ARE } ax
    POINT AY { = | IS | ARE } ay
    POINT AZ { = | IS | ARE } az
    POINT BX { = | IS | ARE } bx
    POINT BY { = | IS | ARE } by
    POINT BZ { = | IS | ARE } bz
    BEAM SECTION { = | IS | ARE } { ROD | TUBE | BAR | BOX | I }
    BEAM WIDTH { = | IS | ARE } BeamWidth
    BEAM HEIGHT { = | IS | ARE } BeamHeight
    BEAM WALL THICKNESS { = | IS | ARE } BeamWallThickness
    BEAM FLANGE THICKNESS { = | IS | ARE } BeamFlangeThickness
    BEAM REFERECNE AXIS { = | IS | ARE } { CENTER | RIGHT | TOP | LEFT | BOTTOM }
    SPRING AREA { = | IS | ARE } SpringArea
    RIGID BODY { = | IS | ARE } RigidBodyName
    EFFECTIVE MODULI MODEL { = | IS | ARE } { elastic | current | presto | pronto }
    THICKNESS MESH VARIABLE { = | IS | ARE } var_name
    THICKNESS TIME STEP { = | IS | ARE } time_value
    SECTION { = | IS | ARE } SectionName
End
```

Details Specifies analysis parameters associated with each element block.

22.6.1 ACTIVE FOR PROCEDURE

Syntax `ACTIVE FOR PROCEDURE` *procedureName* `during periods` *periodNames*

procedureName : *no description* (C)

periodNames : *no description* (C [, ...])

Details Lists the solution periods during which the given BC, solver, preconditioner, etc. is active. Multiple uses of this line command within a single block will have a cumulative affect.

22.6.2 INACTIVE FOR PROCEDURE

Syntax `INACTIVE FOR PROCEDURE` *procedureName* `during periods` *periodNames*

procedureName : *no description* (C)

periodNames : *no description* (C [, ...])

Details Lists the solution periods during which the given BC, solver, preconditioner, etc. is inactive. Multiple uses of this line command within a single block will have a cumulative affect.

22.6.3 MATERIAL

Syntax `MATERIAL` *MatName*

MatName : *no description* (C)

Details Associates this element block with its material properties.

22.6.4 PHASE

Syntax `PHASE` *Phase_Label* { `=` | `IS` | `ARE` } *Material_Name*

Phase_Label : *no description* (C)

Material_Name : *no description* (C)

Details Associate phase Phase_Label with material Material_Name on this block.

22.6.5 DEACTIVATE

Syntax `DEACTIVATE` *CodeName*

CodeName : *no description* (C)

Details Deactivate this elemenent block for this mechanics.

22.6.6 SHELL INTEGRATION POINTS

Syntax `SHELL INTEGRATION POINTS { = | IS | ARE }` *npoints*

npoints : *no description* (I)

Details Specify the number of integration points through the thickness of the shells in this block. This is a deprecated command and should be defined in a element section.

22.6.7 SHELL SCALE THICKNESS

Syntax `SHELL SCALE THICKNESS { = | IS | ARE }` *tscale*

tscale : *no description* (R)

Details Supplies a scale factor to be applied to the thickness attribute on the mesh file. This is a deprecated command and should be defined in a element section.

22.6.8 HOURGLASS

Syntax `HOURGLASS { Stiffness | Viscosity } { = | IS | ARE }` *hgval*

hgval : *no description* (R)

Details Supplies the hourglass parameters for this element block.

22.6.9 MEMBRANE SCALE THICKNESS

Syntax `MEMBRANE SCALE THICKNESS { = | IS | ARE }` *tscale*

tscale : *no description* (R)

Details Supplies a scale factor to be applied to the thickness attribute on the mesh file. This is a deprecated command and should be defined in a element section.

22.6.10 LINEAR BULK VISCOSITY

Syntax `LINEAR BULK VISCOSITY { = | IS | ARE }` *lbv*

lbv : *no description* (R)

Details Supplies the linear coefficient for the bulk viscosity computations.

22.6.11 QUADRATIC BULK VISCOSITY

Syntax `QUADRATIC BULK VISCOSITY { = | IS | ARE }` *qbv*

qbv : *no description* (R)

Details Supplies the quadratic coefficient for the bulk viscosity computations.

22.6.12 SOLID MECHANICS USE MODEL

Syntax `SOLID MECHANICS USE MODEL` *modelname*

modelname : *no description* (C)

Details Associates a solid mechanics material model with this element block. The material parameters for this block are specified in the material denoted by the MATERIAL command.

22.6.13 DEPOSIT SPECIFIC INTERNAL ENERGY

Syntax `DEPOSIT SPECIFIC INTERNAL ENERGY` *edep* `[ OVER TIME` *tdep* `STARTING AT TIME` *tinit* `]`

edep : *no description* (R)

tdep : *no description* (R)

tinit : *no description* (R)

Details Defines the amount of specific (per unit mass) internal energy to be deposited in the material. The energy is deposited over time tdep, beginning at time tinit. The optional parameters tdep and tinit both default to zero, so the energy will be deposited instantaneously at time zero if they are not specified. The deposition is uniform in space, so each element in the block has the same amount edep added to its specific internal energy.

22.6.14 ELEMENT NUMERICAL FORMULATION

Details Specifies which element numerical formulation to use for this block.

22.6.15 SHELL INTEGRATION SCHEME

Details Specify the type of integration scheme through the thickness of the shells in this block. This is a deprecated command and should be defined in a element section.

22.6.16 LOFTING FACTOR

Syntax `LOFTING FACTOR { = | IS | ARE }` *tscale*

tscale : *no description* (R)

Details Supplies a lofting factor to be applied to shells or membranes. A value of 0.5 means no lofting (the default), 0.0 means the meshed shell/membrane is the top of the surface, and 1.0 means it is the bottom of the surface. This is a deprecated command and should be defined in a element section.

22.6.17 DEVIATORIC PARAMETER

Syntax `DEVIATORIC PARAMETER { = | IS | ARE }` *alpha*

alpha : *no description* (R)

Details This line command is required for selective deviatoric hexahedron or the selective deviatoric membrane. Its value, which is valid from 0.0 to 1.0, indicates how much of the deviatoric response should be taken from a uniform gradient integration and how much should be taken from a full integration of the element. A value of 0.0 will give a pure uniform gradient response with no hourglass control. Thus, this value is of little practical use. A value of 1.0 will give a fully-integrated deviatoric response. Although any value between 0.0 and 1.0 is perfectly valid, lower values are generally preferred.

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

22.6.18 TRUSS AREA

Syntax `TRUSS AREA { = | IS | ARE }` *TrussArea*

TrussArea : *no description* (R)

Details Specifies the cross-sectional area for a uniform result three-dimensional truss with uniform results (constant stress). This is a deprecated command and should be defined in a element section.

22.6.19 DAMPER AREA

Syntax `DAMPER AREA { = | IS | ARE }` *DamperArea*

DamperArea : *no description* (R)

Details Specifies the cross-sectional area for a damper element. The cross-sectional area is used only for mass calculations. The length times the area and density is used to generate nodal mass if needed. This is a deprecated command and should be defined in a element section.

22.6.20 ELEMENT INITIAL STRAINXX

Syntax `ELEMENT INITIAL STRAINXX` { `=` | `IS` | `ARE` } *istrainxx*

istrainxx : *no description* (R)

Details Specifies element initial strainxx to use for this block.

22.6.21 ELEMENT INITIAL STRAINYY

Syntax `ELEMENT INITIAL STRAINYY` { `=` | `IS` | `ARE` } *istrainyy*

istrainyy : *no description* (R)

Details Specifies element initial strainyy to use for this block.

22.6.22 ELEMENT INITIAL STRAINZZ

Syntax `ELEMENT INITIAL STRAINZZ` { `=` | `IS` | `ARE` } *istrainzz*

istrainzz : *no description* (R)

Details Specifies element initial strainzz to use for this block.

22.6.23 ELEMENT INITIAL STRAINXY

Syntax `ELEMENT INITIAL STRAINXY` { `=` | `IS` | `ARE` } *istrainxy*

istrainxy : *no description* (R)

Details Specifies element initial strainxy to use for this block.

22.6.24 ELEMENT INITIAL STRAINXZ

Syntax `ELEMENT INITIAL STRAINXZ` { `=` | `IS` | `ARE` } *istrainxz*

istrainxz : *no description* (R)

Details Specifies element initial strainxz to use for this block.

22.6.25 ELEMENT INITIAL STRAINYZ

Syntax `ELEMENT INITIAL STRAINYZ` { `=` | `IS` | `ARE` } *istrainyz*

istrainyz : *no description* (R)

Details Specifies element initial strainyz to use for this block.

22.6.26 ELEMENT COORDINATE SYSTEM

Details Provides a model name for the coordinate system specification: (1) xyz (2) cylindrical (3) spherical

22.6.27 POINT AX

Syntax `POINT AX` { `=` | `IS` | `ARE` } *ax*

ax : *no description* (R)

Details Global x-component of vector defining coordinate system direction A. It is not necessary to input A as a unit vector.

22.6.28 POINT AY

Syntax `POINT AY` { `=` | `IS` | `ARE` } *ay*

ay : *no description* (R)

Details Global y-component of vector defining coordinate system direction A. It is not necessary to input A as a unit vector.

22.6.29 POINT AZ

Syntax `POINT AZ` { `=` | `IS` | `ARE` } *az*

az : *no description* (R)

Details Global z-component of vector defining coordinate system direction A. It is not necessary to input A as a unit vector.

22.6.30 POINT BX

Syntax `POINT BX` { `=` | `IS` | `ARE` } *bx*

bx : *no description* (R)

Details Nominally the global x-component of vector defining coord system direction B. However, it is only necessary that the supplied B vector lie in the A-B plane, because C is constructed as (A x B) and B is redefined to be (C x A). It is not necessary to input B as a unit vector.

22.6.31 POINT BY

Syntax `POINT BY { = | IS | ARE }` *by*

by : *no description* (R)

Details Nominally the global y-component of vector defining coord system direction B. However, it is only necessary that the supplied B vector lie in the A-B plane, because C is constructed as (A x B) and B is redefined to be (C x A). It is not necessary to input B as a unit vector.

22.6.32 POINT BZ

Syntax `POINT BZ { = | IS | ARE }` *bz*

bz : *no description* (R)

Details Nominally the global z-component of vector defining coord system direction B. However, it is only necessary that the supplied B vector lie in the A-B plane, because C is constructed as (A x B) and B is redefined to be (C x A). It is not necessary to input B as a unit vector.

22.6.33 BEAM SECTION

Details Specifies the sectional type (rod, tube, bar, box, i) for a three-dimensional beam with uniform results (constant stress along length). This is a deprecated command and should be defined in a element section.

22.6.34 BEAM WIDTH

Syntax `BEAM WIDTH { = | IS | ARE }` *BeamWidth*

BeamWidth : *no description* (R)

Details Specifies the dimension perpendicular to the beam section z-axis (t-axis). This is a deprecated command and should be defined in a element section.

22.6.35 BEAM HEIGHT

Syntax `BEAM HEIGHT { = | IS | ARE }` *BeamHeight*

BeamHeight : *no description* (R)

Details Specifies the dimension parallel to the beam section z-axis (t-axis). This is a deprecated command and should be defined in a element section.

22.6.36 BEAM WALL THICKNESS

Syntax `BEAM WALL THICKNESS { = | IS | ARE }` *BeamWallThickness*

BeamWallThickness : *no description* (R)

Details Specifies the wall thickness for tube and box sections; specifies the flange thickness for an i section. This is a deprecated command and should be defined in a element section.

22.6.37 BEAM FLANGE THICKNESS

Syntax `BEAM FLANGE THICKNESS { = | IS | ARE }` *BeamFlangeThickness*

BeamFlangeThickness : *no description* (R)

Details Specifies the flange thickness for an i section. This is a deprecated command and should be defined in a element section.

22.6.38 BEAM REFERECNE AXIS

Details Specifies the reference axis location with respect to the beam geometric center line. This is a deprecated command and should be defined in a element section.

22.6.39 SPRING AREA

Syntax `SPRING AREA { = | IS | ARE }` *SpringArea*

SpringArea : *no description* (R)

Details Specifies the cross-sectional area for a spring element. This is a deprecated command and should be defined in a element section.

22.6.40 RIGID BODY

Syntax `RIGID BODY { = | IS | ARE }` *RigidBodyName*

RigidBodyName : *no description* (C)

Details Specifies the rigid body name to use for this element block.

22.6.41 EFFECTIVE MODULI MODEL

Details Specifies the method used to determine the effective moduli. This choice can have a significant effect on the resulting hourglassing behavior. The models are: elastic: use the initial elastic moduli current: use the tangent moduli without correction for edge conditions (e.g. very small stiffnesses, softening materials) presto: use the PRESTO version of the pronto routine, which includes some fairly clear errors. This is included only for backward compatibility with old PRESTO runs. pronto: use the old PRONTO method for computing elastic moduli this approach is straight out of PRONTO, PRESTO's predecessor.

22.6.42 THICKNESS MESH VARIABLE

Syntax `THICKNESS MESH VARIABLE` { `= | IS | ARE` } *var_name*

var_name : *no description* (C)

Details This line defines a mesh variable to read the thicknesses from

22.6.43 THICKNESS TIME STEP

Syntax `THICKNESS TIME STEP` { `= | IS | ARE` } *time_value*

time_value : *no description* (R)

Details This line defines the time step at which to read the thickness variable from

22.6.44 SECTION

Syntax `SECTION` { `= | IS | ARE` } *SectionName*

SectionName : *no description* (C)

Details Specifies the section to use for this element block.

Chapter 23

Developer Documentation

23.1 An Introduction to Aria's Expression System

The following is the conference paper from the Coupled Problems 2005 conference, See Notz et al. (2005).

This section provides a brief overview of a core element of Aria's design called the Expression system. Understanding the Expression system is not essential for using Aria but it is useful in understanding how some of Aria's features. It is also the simplest point of entry for developing and extending Aria with new material properties and boundary conditions. In short, the Expression system is the abstraction of low- to mid-level numerical calculations so as to make the code highly general, reusable, extensible and scalable.

In order to provide a low entry barrier for user extensions we have chosen a design where the essential numerical entities of boundary conditions, material properties, etc., are all implemented the same way. Specifically, these are implemented as C++ classes called Expressions. In addition to supplying the functional evaluation of the numerical quantity, Expression implementers provide information regarding what the Expression provides, what other Expressions it may depend on, the tensorial order of the provided quantity, etc. A higher-level Expression Manager is then able to determine the minimal and sufficient set of Expressions required to perform the simulation and to ensure that they are evaluated in the proper order. In addition to providing a simple extension technique, this makes the core development of Aria very clean and abstracts nonessential details from many of the basic algorithms, while still employing the most efficient numerical kernels and data management techniques at the lowest levels.

To motivate our approach, we start with a simple example. In the FEM solution of the Navier-Stokes equations the following integral, among others, is computed.

$$\int_{\mathrm{V}} \rho \boldsymbol{v} \cdot \boldsymbol{\nabla} \boldsymbol{v} \phi^i \, \mathrm{dV} = \sum_e \int_{\mathrm{V}_e} \rho \boldsymbol{v} \cdot \boldsymbol{\nabla} \boldsymbol{v} \phi^i |j| \, \mathrm{dV}_e \tag{23.1}$$

Here ρ is the density, $\boldsymbol{v}$ is the fluid velocity and ϕ^i is weight function associated with local node i. V is the domain of integration in the physical space and V_e is that in the space of the element with $|j|$ being the Jacobian of transformation from V_e to V. Thus, the assembly kernel responsible for this calculation needs access to all of these quantities. We will show that the kernel, however, does not need to know the particulars of, say, the density model or of the weight function, especially regarding which other quantities they may depend on. Moreover, there are likely to be other assembly kernels that may also require these same quantities, such as the advection term of the energy transport equation,

$$\int_{\mathrm{V}} \rho C_p \boldsymbol{v} \cdot \boldsymbol{\nabla} T \phi^i \, \mathrm{dV} = \sum_e \int_{\mathrm{V}_e} \rho C_p \boldsymbol{v} \cdot \boldsymbol{\nabla} T \phi^i |j| \, \mathrm{dV}_e \tag{23.2}$$

where C_p is the specific heat and T is the temperature. Efficient evaluation of (23.2) should make use of $\rho, |j|, \boldsymbol{v}$ and any other quantities that might have previously been computed in the assembly of (23.1). Keeping track of which quantities are needed and which quantities have already been evaluated in a previous part of the assembly, regardless of the equation system that is being solved, is the purpose of Aria's Expression Manager.

In Aria, all of the quantities presented above, ρ, $\boldsymbol{\nabla v}$, $|j|$, ϕ^i, etc., are implemented as C++ classes derived from the base class Expression. Each Expression has these basic responsibilities:

- **Identification.** Each Expression identifies itself with a generic description, such as density and, when applicable, a specific model description, such as cubic_in_temperature.
- **Prerequisites.** Each Expression (and Expression user) provides a list of other Expressions (using the generic identifiers) that are required for its own calculations. To continue the example, the Expression which implements the cubic_in_temperature model would have temperature as a prerequisite. Arbitrarily complex terms may be built of successively simpler components which ultimately result in a dependency tree with the simplest ingredients being the leaves of the graph.
- **Tensor Properties.** Each Expression states its tensor order (scalar, vector, second order tensor, etc.) and dimension. With this information, Expression users can dynamically determine how to index the values provided by the expression. For example, some thermal conductivities may be scalars while others may be second order tensors as is the case with anisotropic materials.
- **Evaluation.** Each Expression implements a virtual function for computing its values. Depending upon the nonlinear solution strategy, additional methods may be required for computing, for example, Newton sensitivities.

When an Expression is created, it makes itself known to an Expression Manager. The Expression Manager is responsible for establishing the minimal but sufficient list of Expressions required for the calculation. This is done utilizing the lists of prerequisites provided by Expression users and other Expressions. The relationships among Expressions, Expression users, and the Expression Manager is illustrated schematically in figure 23.1 with the use of UML. This figure illustrates two salient features of our Expression subsystem: Expressions are themselves Expression users and Expression users, such as assembly kernels, are the original source for all prerequisites.

The richness of the information provided in these lists of prerequisites quickly becomes apparent. For example, the Expression Manager is able to further utilize the prerequisite information to order the list of Expressions for evaluation. This is done so that when an Expression is evaluated, all of its prerequisite Expressions have already been evaluated. Regardless of how complicated the multiphysics problem happens to be, there are no duplicate evaluations and no overhead costs that would be incurred with lazy-evaluation. Another powerful use for the prerequisite information is in dynamically determining and computing sensitivities for Newton's method. Since primitive variables, or degrees of freedom, are also represented as Expressions, the prerequisites for any given Expression are recursively checked to see if any of them represents a primitive variable. In combination with runtime-queried tensor properties (see above), an Expression developer can implement and evaluate the Expression's sensitivities without ever having direct knowledge of the degrees of freedom in the system. Lastly, Aria utilizes this information for additional purposes such as sparse matrix allocation and automated memory management.

23.2 Nonlinear Coupling Strategies in Aria

One of the difficulties with writing broadly applicable computational mechanics software is that developers can't take advantage of specific knowledge of the application domain in order to optimize

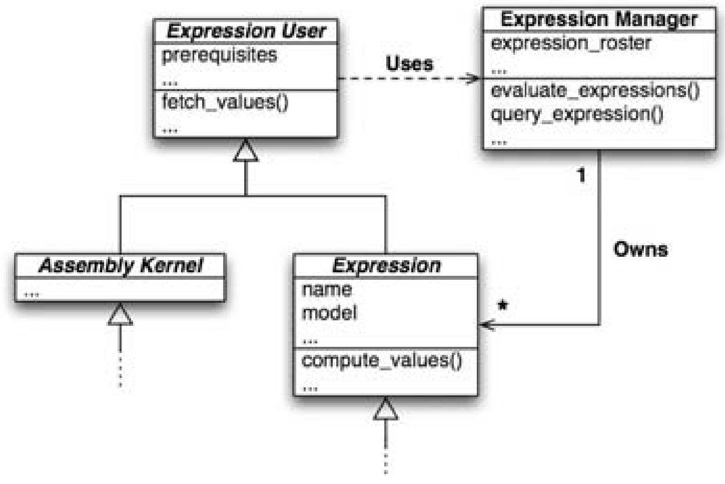


Figure 23.1. Schematic UML class diagram for the Expression subsystem.

the algorithm. Thus, in providing generality one sometimes sacrifices efficiency. One place this is evident in multiphysics modeling is in the choice of coupling strategies. While it is well understood that a fully coupled system solved with Newton's method utilizing analytic sensitivities is formally the most robust and correct approach to solving multiphysics applications it is also computationally expensive and complex to implement. Furthermore, while Newton's method has the fastest rate of asymptotic convergence it's domain of convergence is often empirically observed to be smaller than other methods. Lastly, in some applications, certain subsets of the physics may be only weakly coupled so that a loosely coupled approach may be more computationally efficient. To address these concerns while remaining general and flexible Aria offers a number of options for nonlinear solution strategies and physics coupling.

In defining a problem in Aria, users configure one or more Regions. Each Region consists of one or more PDEs to be solved on some or all of the input mesh. All of the PDEs in each Region are solved in a tightly coupled (i.e., single matrix) manner using one of several nonlinear solution strategies available. Users may then define loose couplings between two or more Regions. For example, some or all of a solution from one Region may be transferred to another Region where it is treated as a constant, external field. The aggregate nonlinear problem including the contributions from all of the Regions may be iterated to convergence. The particulars of which physics are solved in each Region and the nonlinear solution strategy used within and between Regions is completely specified through the input file. Furthermore, an Aria user may pick a simple, minimal algorithm without needing to fit it into an overly-generalized worst-case scenario that represents the union of all possible algorithms.

Dynamically-specified loose coupling has many potential advantages that users may leverage. First, the resulting linear system is considerably smaller and contains far fewer off-diagonal contributions which can significantly increase the performance of linear solvers. Also, a resulting linear system may have a more attractive form, such as symmetric positive-definite, that permits the use of tailored iterative solutions techniques. Other extensions to loose coupling include subcycling of transient simulations where each Region may advance in time with its own time step size and in-core coupling to other applications based upon the Sierra framework.

23.3 Developer Recipes

23.3.1 Adding a New Flux Boundary Condition

1. Add a new name for your model
   (a) Add a new `Identifier` entry to `Aria_Model_Names.h`
   (b) Add the string in `Aria_Model_Names.C`
2. Add your new Expression class.
   (a) Copy-paste-rename a similar Expression definition. See, e.g., `Aria_Material_Models.h`.
   (b) Copy-paste-rename a similar Expression implementation. See, e.g., `Aria_Material_Models.C`.
3. Add an entry to the `Expression_Factory::create_model_expression()`.
   (a) Copy-paste-rename a similar entry in `Aria_Expression_Factory.C`.

23.3.2 Adding a New Material Model

1. Add a new name for your model
   (a) Add a new `Identifier` entry to `Aria_Model_Names.h`
   (b) Add the string in `Aria_Model_Names.C`
2. Add your new Expression class.
   (a) Copy-paste-rename a similar Expression definition. See, e.g., `Aria_Material_Models.h`.
   (b) Copy-paste-rename a similar Expression implementation. See, e.g., `Aria_Material_Models.C`.
3. Add an entry to the `Expression_Factory::create_model_expression()`.
   (a) Copy-paste-rename a similar entry in `Aria_Expression_Factory.C`.

23.4 Expression Reference and API

This section provides a high-level view of how Expressions are run by Aria, from creation to computation. This section also provides a brief description of the Expression methods that a user either must or can optionally supply.

23.4.1 Execution Sequence

1. Initialization Phase (Input Parsing Time)
   (a) `Expression::Expression()` (See 23.4.2) : Constructor. Called at creation time. Defines identity, dependents and parameters. Required.
   (b) `Expression_Manager::preprocess_expressions()` : Called after all expressions have been created and all dependencies have been registered/requested. Unused expressions are destroyed. Dependency based ordering established.
   (c) `Expression::set_nonlinear_method()` : Sets the nonlinear method that will be used.

   (d) `Expression::create_dynamic_storage()` : Initializes storage for values, Newton sensitivies, Picard factors but does not actually allocate memory.

   (e) `Expression::postregistration_setup()` (See 23.4.3) : Called once, after the Expression_Manager has determined that the Expression will indeed be used but before simulation begins. Called in depdendency order. Can get references to dependent's storage (values, Newton sensitivities, Picard factors, etc.).

   (f) `Kernel::postregistration_setup()` : Assembly kernels can interrogate the Expressions available to them and get references to values, Newton sensitivies and Picard factorizations. For Picard's method, Kernels notify Expression_Manager which expressions they will factor.

   (g) `Expression::register_picard_call()` (See 23.4.4) : Called when a Kernel notifies the Expression_Manager that it will factor this Expression for Picard's method. Here, the Expression can register which Expressions it will, in turn, use in its factorization.

2. Simulation Phase

   (a) `Expression::prepare_to_recompute()` (See 23.4.5): Called once for each workset, immediately prior to compute_values(). Can be used to resize or reinitialize storage (usually done by base class), inquire about simulation time, etc.

   (b) `Expression::compute_values()` (See 23.4.6) : Computes the values of the Expression. Required to compile.

   (c) `Expression::compute_sensitivities()` (See 23.4.7) : Computes the Newton sensitivities of the Expression. Required to compile.

   (d) `Expression::compute_picard_factors()` (See 23.4.8) : Computes the Picard factorizations of the Expression. Generally only used for DoF-based Expressions and constitutive models. Required dynamically, if Kernels ask for it.

3. Shutdown Phase

   (a) `Expression:: Expression()` (See 23.4.9) : Destructor. Can be used to free memory allocations (note: Real_MDArray storage is automatically deallocated). Required but normally empty.

23.4.2 Constructor

23.4.3 `postregistration_setup()`

23.4.4 `register_picard_prerequisites()`

23.4.5 `prepare_to_recompute()`

23.4.6 `compute_values()`

23.4.7 `compute_sensitivities()`

23.4.8 `compute_picard_factors()`

23.4.9 Destructor

23.5 Newton Sensitivity Checking for Expressions

When you run Aria you can ask it to look for sensitivity errors in every Expression if you're using Newton's method. If activated, Aria will compare the analytic sensitivities provided by each expression with numerically computed values. This feature is enabled by adding the command line option `-arialog sens_check` to Aria. If you're using the `sierra` tool to run Aria, then add `-O ''-arialog sens_check''` (including quotes) to the `sierra` command line. The same holds if you're running the Arpeggio application.

You can also run the entire test suite with Jacobian checking enabled by adding `-u aria_args = ''-arialog sens_check''` to the `runtest` command line.

23.6 Profiling

It's a good idea to profile the code before spending time trying to optimizing and performance tuning. Fortunately, profiling is pretty easy to do. On Linux, one approach is to use the `gprof` tool. To profile code with gprof, you'll need to compile with the `-pg` option. The easiest way to do this is to set environment variables `USER_CFLAGS` and `USER_LDFLAGS` to `-pg`. In the bash/ksh/zsh shell, use, e.g., `export USER_CFLAGS=-pg` and in tcsh/csh use `setenv USER_CFLAGS -pg`. You'll only get detailed profileing information for code compiled and linked with this flag so you may want to recompile some of the framework too. You can profile debug or optimized code.

With the instrumented executable, run Aria as you normally would. This will generate a file called `gmon.out`. Lastly, run `gprof` to analyze the data: `gprof` *`executable`* > `gprof.txt` where *`executable`* is the executable (probably including an absolute or relative path) used in the run. The output stored in `gprof.txt` will give you some real data telling where the real bottle necks are.

23.7 Purify: Memory Analysis and Debugging

1. Build Aria on Linux. This works for both optimized and debug modes but use debug if you want to see line numbers in Purify.
2. Copy the complete link line into a file.
3. Add `-show` option to `mpiCC`.
4. Remove the `-static` option from the link line arguments.
5. Adjust any path to `Apps_aria.o` if necessary.
6. Put `which mpiCC` on the previous line to verify you're getting `mpiCC` from `/usr/local/mpi/sierra/mpich/1.2.5.2/gcc-3.2.2/bin/`.
7. You might also verify `gcc` is version 3.2.2 (`gcc -v`).
8. Source the script.
9. Copy the link line (start with `g++ ...`) and paste it into a file.
10. Preface `g++` with `purify`, e.g., `purify -always-use-cache-dir -cache-dir=‘pwd‘/ g++ ...`
11. Source this script and it will instrument your executable. It takes a while...
12. Optionally verify that the executable is `purify`ed by running `ldd aria.x`
13. Copy executable to aria/bin directory, removing the `.tmp` extension.
14. Run it standalone (via `sierra`, `runtest` or by hand) or with TotalView.

23.8 Building Against Other Projects

Sometimes it can be very usefull to build against changes that have been made in a different project. For example, if you are waiting for a project's changes to be checked in but would like to start developing against those changes in another project. To do this with the SNTools `build` tool you can use the `-a` option to build. The argument to this option is the XML file for each *product* you want to include in your build. So, if you want to use the version of Krino found in a project located in `/path/to/project/` then you would add `-a /path/to/project/krino/krino_sn.xml` to your normal `build` command line.

23.9 Interfacing with MATLAB

23.9.1 Reading Aria Matrices Into MATLAB

This is sort of an "old" way of getting matrices into MATLAB but it still works just fine.

From Matt Hopkins:

You can read a `.mtx` matrix into MATLAB via the following. Let `my_matrix.mtx` be the matrix sitting in the debug output path you're interested in.

```
>> A_ascii = load('my_matrix.mtx');
>> A = spconvert(A_ascii(2:end,:));
```

And now `A` is in MATLAB's sparse matrix format. It even ignores the zeros in the `.mtx` file (does not allocate space for them).

From there you can do some pretty nifty things. `spy(A)` does sort of what matvis does (graphical view of matrix nonzeros). `condest(A)` estimates the 1-norm condition number of `A` (might take a while for large `A`). Etc.

23.9.2 Interfacing Aria from within MATLAB

Russell Hooper is developing a pretty sophisticated MATLAB interace that allows you to control Aria and probe nonlinear and linear solver data structures directly from within MATLAB. This is rapidly evolving so no details are given here... except for how to build and run Aria/MATLAB.

This recipe is taken from an email exchange between Matt Hopkins and Russell Hooper in early April 2006. It most likely only works on Linux.

```
% setenv LD_LIBRARY_PATH  /usr/local/matlab/7.2/bin/glnx86:\${LD_LIBRARY_PATH}

% build aria -j8 -a /home/rhoope/Trilinos_6.0  \
   CXXFLAGS="-I/usr/local/matlab/7.2/extern/include/" \
   -a  /home/rhoope/projects/noxMatlabStable/equationsolver \
   LDFLAGS="-L/usr/local/matlab/7.2/bin/glnx86  -leng"
```

If you want to build a debug version, add "-g" to both the `CXXFLAGS` and `LDFLAGS`.

To execute aria, first invoke `sierra aria -i input.i --script-only`. Then copy the file `sierra.sh` into another, ie `opt.sh`. Finally, remove the "-o logfilename" argument from `opt.sh` so that your run is truly interactive.

23.10 Error Handling

Traditional techniques for error handling of numerous and varried. Sometimes functions will return special values indicating success or failure of the call. Sometimes global variables or flags are set to indicate an error has occurred. Sometimes abort() is simply called (bad!).

C++ and the SIERRA Framework offer facilities to support error handling. The two primary error handling techniques are *exception handling* and *assertions*. Both of these can be combined to implement *Design by Contract* though this is not formally done in Aria.

Whether one should use an exception or an assertion is sometimes a subtle question. In general, assertions should be used for cases where the error is not likely to occur due merely to user input. However, assertions can also be useful for computationally expensive tests which would adversely affect production calculations.

23.10.1 Exception Handling

C++ offers *exception handling* and the SIERRA Framework utility library provides a parallel-safe layer for exception handling. Because exceptions in C++ are ordinary classes, applications may choose to specialize exceptions for handling and detecting specific failure modes. When SIERRA exceptions are used in conjuction with the SIERRA diagnostic tracing facility, it is possibly to get a stacktrace (or sometimes a partial one) that illustrates the code path the where the exeption was thrown.

In most cases, Aria developers do not need to worry about catching exceptions (they're noramlly fatal errors). The top-level SIERRA Framework routine `run_sierra()` handles the responsibility of catching any uncaught exceptions, converting them into parallel exceptions if necessary, and propogating the exception to any other processors.

If the reader is not already familiar with exception handling in C++, a good place to start is with Stroustrup (2000). In it's simplest form, SIERRA exceptions can be used as follows.

```
#include <util/Exception.h>
// ...
if(something_bad_happened)
  {
    sierra::Exception x("Nice descriptive error message.");
    throw x;
  }
```

The `sierra::Exception` class inherrits from `std::string` and so it supports all of `std::string`'s operations, most notably the + (concatentaion) operator.

```
#include <util/Exception.h>
// ...
if(something_bad_happened)
  {
    sierra::Exception x("Nice descriptive error message.");
    x += "  More info here.";
    throw x;
  }
```

Additionally, the `sierra::Exception` class supports the put-to operator, `<<`, for stream-like handling.

```
#include <util/Exception.h>
// ...
if(something_bad_happened)
  {
    sierra::Exception x;
    x << "Nice descriptive error message."
      << "  More info here.";
    throw x;
  }
```

However, in order to keep the `sierra::Exception` class light weight, it does not inherrit from `std::ostringstream`. Instead, `operator<<` is only defined for the most common objects, such as strings, integers, floats/doubles, etc. The consequence of this is that if you define a class with `operator<<` for `ostream`, that operator will not work with `sierra::Exception` objects. In these cases, it is sometimes useful to use a `std::ostringstream` to construct the error message and then pass the resulting `std::string` to the exception.

```
#include <util/Exception.h>
#include <sstream>
// ...
if(something_bad_happened)
  {
    std::ostringstream os;
    os << "Nice descriptive error message."
       << "  Object foo: ";
       << some_object
       << " is unusable";
    sierra::Exception x;
    x << os.str();
    throw x;
  }
```

Finally, the construction and throwing of the exception can occur on the same line for brevity. In this case, all we need is a temporary object which is unnamed.

```
#include <util/Exception.h>
// ...
if(something_bad_happened)
  {
    throw sierra::Exception() << "Nice descriptive error message."
                              << " Integer out of bounds: j = "
                              << j;
  }
```

23.10.2 Assertions

Assertions are a common way to ensure that certain conditions that are assumed by the code to be true are indeed true. For example, a function or piece of code may require that a pointer be non-NULL, an integer be greater than zero, etc. Using the C++ `assert()` macro is a good way to test such conditions that *should* hold under normal circumstances. By using `assert()`, the enclosed test will only be perfomed if the code is compiled in debug mode; in optomized mode, the macros expand to nothing. Thus, developers can use asserts extensively to test conditions assumed by their code without affecting production-mode performance. Assertions also provide an excellent way to document *and enforce* the assumptions that the developer made in writing the code.

The SIERRA Framework provides an alternate implementation of the `assert()` macro that utilizes the parallel-safe exception handling described above. The macro, `ThrowAssert()`, is defined in the same header file as the exception class.

```
#include <util/Exception.h>
// ...
void
some_function(const MyClass * myclass_ptr)
{
  ThrowAssert(0 != myclass_ptr);
  // It should be OK to use the pointer.
  // ...
```

Because assertions expand to empty code in optimized mode, developers must be careful to never put variable assignments or other state-altering code inside an assert.

```
#include <util/Exception.h>
// ...
// This is a bug:
ThrowAssert(j = i + 2 > 4);
```

A common trick to getting more useful messages from a failed assertion is to add a help string like this:

```
#include <util/Exception.h>
// ...
void
some_function(const Int & num)
{
  ThrowAssert(num > 10 &&
            "This function only works propertly when num is larger than 10");
  // It should be OK to use the pointer.
  // ...
```

Lastly, another macro, `ThrowRequire()` is available. This macro is identical to the `ThrowAssert()` macro except this it is always enabled, even in optimized builds. This macro is currently not used in Aria but use it if you want. The concise and readable form of these macros can help keep code short while still being readable and expressive.

23.11 Outputting User Information (Logging)

The SIERRA Framework utility library supplies a flexible logging facility. The problem with normal `printf` and `cout` output functions is that they get really annoying in parallel. Further, these messages simply roll off the top of the user's screen unless they are careful to redirect the standard output and error streams to a file. Lastly, it's difficult, and may involve recompiling the application, to tailor these messages depending on the kind of information a user is interested in.

23.11.1 The DiagWriter Logging Facility

SIERRA's logging facility, `DiagWriter`, is designed to address these three issues. The `DiagWriter` class, in the simplest sense, is a `std::ostream` class that supports the put-to (`<<`) operator for writing messages.

```
#include <Aria_DiagWriter.h>
//...
namespace Aria
{
  arialog << "This is a message the user will always see." << std::endl;
}
```

Note the use of `std::endl` instead of `std::endl` (the standard and portable newline). This is an unfortunate but important point; using `std::endl` will result in ugly compiler errors.

The most interesting feature of the `DiagWriter` class is that the output can be tagged with a bit field called a *message mask*. These masks are defined in Aria_DiagWriter_fwd.h as enums with meaningful names, e.g., `LOG_EXPRESSION` and `LOG_NONLINEAR`. To mask your output message, you can use the `m()` method on the `DiagWriter` object.

```
#include <Aria_DiagWriter.h>
//...
namespace Aria
{
  arialog.m(LOG_EXPRESSION)
     << "This message is supposedly related to Expressions."
     << std::endl;
  arialog.m(LOG_EXPRESSION | LOG_NONLINEAR)
     << "This message is supposedly related to Expressions "
     << "and/or the nonlinear solver"
     << std::endl;
}
```

Messages that contain a mask are only written if that is enabled via the *print mask* and messages that don't have a specified mask are always written. The user can define a print mask using the "-arialog *string*" command line argument where the provided string maps to one (or more) of the bit masks. The file Aria_DiagWriter.C contains the mapping between the string names and the bit mask values. For example, "expression" is assigned to `LOG_EXPRESSION`. When the user supplies the command line argument "-arialog expression" then the messages written are those masked with `LOG_EXPRESSION` and those with no bit masks.

The `DiagWriter` class contains many more features that are beyond the scope of this section. Explore the header files and the code for more examples.

23.11.2 The Tracing Facility

Tracing is a common and extremely useful debugging utility. With tracing enabled, the code prints the name of each function as it enters and exits the function. C++ doesn't provide any standard way to accomplish this so the `Trace` class is provided by the SIERRA Framework utility library.

The `Trace` class works by creating an instance (object) as the first line of a. The object constructor takes as an argument a string (char *) containing the name of the function. When tracing is enabled the constructor prints this name. When the application leaves this function, the local `Trace` object is automatically destroyed as it goes out of scope. When tracing is enabled, the automatically-called destructor re-prints the function name. The output of nested functions is nested so the output provides a hierarchical view of the flow of control. The most basic usage looks like this:

```
#include <Aria_DiagWriter.h>
//...
namespace Aria
{
  void my_func(const Int & i)
    {
      Trace diag_trace("Aria::my_func(const Int & i)");
      //... normal code follows
    }
}
```

The `Trace` class utilizes the `DiagWriter` class for writing and for determining if tracing is enabled. So, tracing may be enabled using "-arialog trace" on the command line and `Aria::Trace` objects mask all output with `LOG_TRACE`.

Though it's not discussed here, tracing can be turned on and off during the normal execution so that tracing information can be gathered only when it's desired. Read through the header files and the code for examples of doing that. One example of where this is used is in the exception handling. When an exception is thrown, the tracing bit mask is automatically turned on. As the function stack unwinds during the throw, any functions instrumented with `Trace` objects will get destructed and hence they will print their owning function name. The result is a stacktrace, or traceback, which can be extremely useful for debugging.

Adding all of the Trace objects to a code can be a daunting task, especially if you're really anal and want the function arguments and namespace to be a part of the function's printed name (which can be important with polymorphic functions). Aria uses the traceString tool (http://tracestring.sourceforge.net/) to automatically instrument the code with the `Trace` objects. The traceString tool will enclose the `Trace` objects in a pair of special strings so that it can safely update or remove the inserted code later on. Typically, it looks like this:

```
#include <Aria_DiagWriter.h>
//...
namespace Aria
{
  void my_func(const Int & i)
    {
      /* %TRACE[ON]% */ Trace diag_trace("Aria::my_func(const Int & i)"); /* %TRACE% */
      //... normal code follows
    }
}
```

To learn more about traceString, talk to Pat...

23.12 Catalogue of Assembly Kernel Expressions

Aria supplies several generic expressions that can be used for top-level assembly kernels for equation terms. When adding new equations or terms to existing equations, one of the follow generic expressions can often be used.

23.12.1 Scalar_Source_Kernel_Expression

This Expression assembles a source term for a scalar transport equation. Note that the `MASS` and `SRC` terms can both be cast in this form. The general form is:

$$-m\sum_{g=1}^{N_g}\left(\prod_{r=1}^{N_c}c_r(\boldsymbol{x}_g)\right)\left(\sum_{s=1}^{N_s}f_s(\boldsymbol{x}_g)\right)\phi^i(\boldsymbol{x}_g)|j|w_g \tag{23.3}$$

Here, m is a multipler that defaults to 1, N_g is the number of Gauss points in the quadrature rule, N_c is the number of coefficients and N_s is the number of sources provided. As an example, consider the term,

$$\int_{\Omega_e} \rho C_p \frac{\partial T}{\partial t} |j| \phi^i \, \mathrm{d}\Omega_e = \sum_{g=1}^{N_g} \rho C_p \frac{\partial T}{\partial t} |j| \phi^i w_g \tag{23.4}$$

In this example, $N_r = 2$ with $c_1 = \rho$ and $c_2 = C_p$; $N_s = 1$ with $f_1 = \frac{\partial T}{\partial t}$; and $m = -1$. As a side note, in the code the quantities $|j|$ and w_g are treated as additional coefficients for convenience.

23.12.2 Scalar_Advection_Kernel_Expression

This Expression assembles an advection term for a scalar transport equation. The general form is:

$$m \sum_{g=1}^{N_g} \left(\prod_{r=1}^{N_c} c_r(\boldsymbol{x}_g) \right) \boldsymbol{v}_a \cdot \boldsymbol{\nabla} S(\boldsymbol{x}_g) \phi^i(\boldsymbol{x}_g) |j| w_g \tag{23.5}$$

Here, m is a multipler that defaults to 1, N_g is the number of Gauss points in the quadrature rule, N_c is the number of coefficients, $\boldsymbol{v}_a$ is a configurable advection velocity and S is a scalar field. As an example, consider the term,

$$\int_{\Omega_e} \rho C_p \boldsymbol{v} \cdot \boldsymbol{\nabla} T |j| \phi^i \, \mathrm{d}\Omega_e = \sum_{g=1}^{N_g} \rho C_p \boldsymbol{v} \cdot \boldsymbol{\nabla} T |j| \phi^i w_g \tag{23.6}$$

In this example, $N_r = 2$ with $c_1 = \rho$ and $c_2 = C_p$; $S = T$; $\boldsymbol{v}_a = \boldsymbol{v}$; and $m = 1$. As a side note, in the code the quantities $|j|$ and w_g are treated as additional coefficients for convenience.

23.12.3 Scalar_Diffusion_Kernel_Expression

This Expression assembles a diffusion term for a scalar transport equation. The general form is:

$$m \sum_{g=1}^{N_g} \left(\prod_{r=1}^{N_c} c_r(\boldsymbol{x}_g) \right) \left(\sum_{k=1}^{N_F} \boldsymbol{F}_k(\boldsymbol{x}_g) \right) \cdot \boldsymbol{\nabla} \phi^i(\boldsymbol{x}_g) |j| w_g \tag{23.7}$$

Here, m is a multipler that defaults to 1, N_g is the number of Gauss points in the quadrature rule, N_c is the number of coefficients and N_F is the number of flux models provided. As an example, consider the term,

$$\int_{\Omega_e} \boldsymbol{q} \boldsymbol{\nabla} \phi^i |j| \, \mathrm{d}\Omega_e = \sum_{g=1}^{N_g} \boldsymbol{q} \cdot \boldsymbol{\nabla} \phi^i |j| w_g \tag{23.8}$$

In this example, $N_r = 0$; $N_F = 1$ with $F_1 = \boldsymbol{q}$ where $\boldsymbol{q}$ is the Fourier heat flux, $\boldsymbol{q} = -\kappa \boldsymbol{\nabla} T$; and $m = 1$. As a side note, in the code the quantities $|j|$ and w_g are treated as additional coefficients for convenience.

23.12.4 Vector_Source_Kernel_Expression

This Expression assembles a source term for a vector transport equation. Note that the `MASS` and `SRC` terms can both be cast in this form. The general form is:

$$-m \sum_{g=1}^{N_g} \left(\prod_{r=1}^{N_c} c_r(\boldsymbol{x}_g) \right) \left(\sum_{s=1}^{N_s} \boldsymbol{f}_s(\boldsymbol{x}_g) \right) \phi^i(\boldsymbol{x}_g) |j| w_g \tag{23.9}$$

Here, m is a multipler that defaults to 1, N_g is the number of Gauss points in the quadrature rule, N_c is the number of coefficients and N_k is the number of sources provided. In this case, the sources f_s are vector quantities. As an example, consider the term,

$$\int_{\Omega_e} \rho \frac{\partial \boldsymbol{v}}{\partial t} |j| \phi^i \, \mathrm{d}\Omega_e = \sum_{g=1}^{N_g} \rho \frac{\partial \boldsymbol{v}}{\partial t} |j| \phi^i w_g \tag{23.10}$$

In this example, $N_1 = 2$ with $c_1 = \rho$, $N_s = 1$ with $f_1 = \frac{\partial \boldsymbol{v}}{\partial t}$; and $m = -1$. As a side note, in the code the quantities $|j|$ and w_g are treated as additional coefficients for convenience.

23.12.5 Vector_Advection_Kernel_Expression

This Expression assembles an advection term for a vector transport equation. The general form is:

$$m \sum_{g=1}^{N_g} \left(\prod_{r=1}^{N_c} c_r(\boldsymbol{x}_g) \right) \boldsymbol{v}_a \cdot \boldsymbol{\nabla} \boldsymbol{V}(\boldsymbol{x}_g) \phi^i(\boldsymbol{x}_g) |j| w_g \tag{23.11}$$

Here, m is a multipler that defaults to 1, N_g is the number of Gauss points in the quadrature rule, N_c is the number of coefficients, $\boldsymbol{v}_a$ is a configurable advection velocity and $\boldsymbol{V}$ is a vector field. As an example, consider the term,

$$\int_{\Omega_e} \rho \boldsymbol{v} \cdot \boldsymbol{\nabla} \boldsymbol{v} |j| \phi^i \, \mathrm{d}\Omega_e = \sum_{g=1}^{N_g} \rho \boldsymbol{v} \cdot \boldsymbol{\nabla} \boldsymbol{v} |j| \phi^i w_g \tag{23.12}$$

In this example, $N_r = 1$ with $c_1 = \rho$; $\boldsymbol{V} = \boldsymbol{v}$; $\boldsymbol{v}_a = \boldsymbol{v}$; and $m = 1$. As a side note, in the code the quantities $|j|$ and w_g are treated as additional coefficients for convenience.

23.12.6 Vector_Diffusion_Kernel_Expression

This Expression assembles a diffusion term for a vector transport equation. The general form is:

$$m \sum_{g=1}^{N_g} \left(\prod_{r=1}^{N_c} c_r(\boldsymbol{x}_g) \right) \left(\sum_{k=1}^{N_F} \boldsymbol{F}_k^t(\boldsymbol{x}_g) \right) : \boldsymbol{\nabla} \left(\boldsymbol{e}_j \phi^i(\boldsymbol{x}_g) \right) |j| w_g \tag{23.13}$$

Here, m is a multipler that defaults to 1, N_g is the number of Gauss points in the quadrature rule, N_c is the number of coefficients and N_F is the number of flux models provided. As an example, consider the term,

$$\int_{\Omega_e} \boldsymbol{T}^t : \boldsymbol{\nabla} \left(\boldsymbol{e}_j \phi^i \right) |j| \, \mathrm{d}\Omega_e = \sum_{g=1}^{N_g} \boldsymbol{T}^t : \boldsymbol{\nabla} \left(\boldsymbol{e}_j \phi^i \right) |j| w_g \tag{23.14}$$

In this example, $N_r = 0$; $N_F = 1$ with $F_1 = \boldsymbol{T}$ where $\boldsymbol{T}$ is the Newtonian stress, $\boldsymbol{T} = \mu \left(\boldsymbol{\nabla} \boldsymbol{v} + \boldsymbol{\nabla} \boldsymbol{v}^t \right) - p\boldsymbol{I}$; and $m = 1$. As a side note, in the code the quantities $|j|$ and w_g are treated as additional coefficients for convenience.

23.13 Errors and Warnings How-To

Editorial note: This How-To is mostly taken from Dave Bauer's collection of How-Tos. Minor changes include formatting, editorial corrections and possibly additioinal information related to Aria.

23.13.1 Reporting

There are several types of warnings and exceptions that occur in sierra. Warnings are informational messages to the user which may effect the results, but which will allow the execution to complete. Dooms are error conditions which will allow the program to continue to a point, but will not allow it to complete. Often these are within the parser when you want the parser to continue to plod on ahead but not actually execute the code. Exceptions occur when all bets are off.

The Warning and Doom classes send their assembled message to the sierra reporter at destruction. The Exception classes send their assembled message to the sierra reporter within the catch block.

The warning and exception classes will all accept the put-to operator (`<<`) with plain old data. Since it is easy to put useful information to the user/developer in the message, please do so. For warnings directed to developers, it is recommended that the message be terminated with a

```
  << std::endl << WarnTrace;
```

For exceptions, always use

```
    << std::endl << StackTrace;
```

`WarnTrace` and `StackTrace` generate a ”pretty” source file and line number message. If the message is purely informational for the user the std::endl and WarnTrace for the message should be omitted.

These are the include files:

```
  #include <util/Exception.h>// throw sierra
  #exception declarations
  #include <util/ExceptionReport.h>// sierra reporter and
  #RuntimeWarning declarations
```

The `RuntimeUserError` class should be thrown in place of `RuntimeError` when it was the user's fault that the program died. This generally means that the input deck has an error that a `RuntimeDoomed`, the preferred method since it allows the parser, etc to continue, cannot handle since a seg fault or the like is imminent. It kills the output of the traceback since the user really does care and will only cause confusion.

Also, don't put the `WarnTrace` or `StackTrace` to the `RuntimeUserError`.

The following code snippets serve as examples for the warning, doomed and exception conditions.

```
class_tag *
Parser::prsr_handler_x(
const Prsr_CommandValues & value)
{

    if (runtime_warning_condition)
sierra::RuntimeWarning() << "My useful message about " <<
some_data << std::endl << WarnTrace;


    if (runtime_exception_condition)
        throw sierra::RuntimeError() << "My useful message about " <<
        some_data << std::endl << StackTrace;


    if (runtime_exception_condition)
        throw sierra::RuntimeUserError() << "My useful message about "
        << some_data;


    if (parse_handler_warning_condition)
        sierra::RunWarning() << "My useful message about " <<
        some_data << std::endl << WarnTrace;


    if (parse_handler_doomed_condition)
        sierra::RuntimeDoomed() << "My useful message about " <<
        some_data << std::endl << WarnTrace;

}
```

23.13.2 Throttling a Specific Warning or Doom

If a particular warnings is going to be repeated countless times, you can request a unique message id from sierra using

```
  int message_id = sierra::get_next_message_id();
  int message_id = sierra::get_next_message_id(int max_messages_displayed);
```

and pass `message_id` to the message constructor. This value is often stored within the function that generates the message using a static variable. This technique may not be suitable for all situations. Only a limited number of messages of each id are sent to the sierra reporter.

```
    if (runtime_warning_condition) {
        static message_id = get_next_message_id();
sierra::RuntimeWarning(message_id) << "My useful message about
        " << some_data << std::endl << WarnTrace;
    }


    if (parse_handler_warning_condition) {
        static message_id = get_next_message_id();
        sierra::Prsr::ParseWarning(value, message_id) << "My useful
        message about " << some_data << std::endl << WarnTrace;
    }


    if (parse_handler_doomed_condition) {
        static message_id = get_next_message_id();
        sierra::Prsr::ParseDoomed(value, message_id) << "My useful
        message about " << some_data << std::endl << WarnTrace;
    }
```

```
if (parse_handler_exception_condition) {
    static message_id = get_next_message_id();
    sierra::Prsr::ParseError(value, message_id) << "My useful
    message about " << some_data << std::endl << WarnTrace;
}
```

23.13.3 Setting Output Throttles

There are several functions which can throttle the amount of data which is displayed.

To set the maximum number of warnings or dooms before an exception is thrown:

```
void sierra::set_max_warnings(int max_warnings);
void sierra::set_max_dooms(int max_dooms);

int sierra::get_max_warnings();
int sierra::get_max_dooms();
```

To set the maximum number of warnings displayed. Each occurance is still counted, but not displayed. Dooms are always displayed.

```
void sierra::set_max_displayed_warnings(int max_display_warnings);
int sierra::get_max_displayed_warnings();
```

To set the default maximum id messages to display, set `get_next_message_id()` below:

```
int sierra::set_default_max_id_dispaly(int
default_max_id_displayed)
int sierra::set_default_max_id_dispaly(int
default_max_id_displayed)
```

23.13.4 Sierra Exception Reporter

What is this sierra reporter thing you ask? It goes like this: When a Warning or a Doom class is destroyed, `report_exception()` is called with the message. `report_exception()` then calls the registered `report_exception_handler`. The `report_exception_handler` has a signature of

```
(*)(const char *message, int type)
```

and is set with `set_report_exception_handler()`. Sierra sets this to use the `sierra_report_exception_handler()` which writes the message to `sierra::Env::output()`.

If you want special decorations around your messages during executeion, you can change the reporter. The parse, instantiation and commit handlers are set by `run_sierra()`.

23.13.5 The `Output` versus `OutputP0` Dilema

Unfortunately, determining what to do with the output from multiple processors can be an issue. If a warning is going to happen on all processors, you may want to wrapper it with a test for processor zero and only output it there. The default `report_exception_handler` sends the output to `Env::output()`

23.13.6 Getting Counts

To get the number of warnings issued, even if not displayed:

```
get_warning_count()
```

To get the number of dooms issued, even if not displayed:

```
get_doomed_count()
```

23.13.7 Traceback and Tracing

The traceback messages are generated by the Trace and Traceback classes upon there destruction. So, if the source code has had trace objects constructed, the stack trace will show them. If efficiency becomes an issue, the `#define app_TRACE_ENABLED` can be undefined which will cause the Trace and Traceback classes to be empty.

The trace object can be used to extract function signature information. The `getFunctionSpec()`, `getFunctionName()`, `getFunctionShortName()`, `getFunctionClass()`, `getFunctionShortClass()` and `getFunctionNamespace()` functions will all extract appropriate information from the function signature.

The program `/usr/netpub/traceString/traceString` (on Linux) can be used to quickly place the trace objects in your source code. I suggest editing the results and placing `[ON]`, `[TRACEBACK]` or `[NONE]` after the first `%TRACE` as in `%TRACE[NONE]%`. These will instruct `traceString` which object type to create. Then, rerun `traceString`. I set it to `NONE` for functions which will execute many many times or cannot throw an error or not have much value in a stack trace (accessors, etc). I set it to `TRACEBACK` if it will not be useful to see traced during a trace run, but be useful in a traceback. And `ON` otherwise which incurs some overhead that enables the conditional runtime trace.

Here is the `.traceString` file I use and suggest be used throughout sierra.

```
[START_OF_ROUTINE_PATTERNS]
default   : Trace trace__("$(FQNAME)");
off       :
Off       :
OFF       :
none      :
None      :
NONE      :
spec      : static Tracespec trace__("$(FQNAME)");
Spec      : static Tracespec trace__("$(FQNAME)");
SPEC      : static Tracespec trace__("$(FQNAME)");
on        : Trace trace__("$(FQNAME)");
On        : Trace trace__("$(FQNAME)");
ON        : Trace trace__("$(FQNAME)");
traceback : Traceback trace__("$(FQNAME)");
Traceback : Traceback trace__("$(FQNAME)");
TRACEBACK : Traceback trace__("$(FQNAME)");
```

23.13.8 Deriving from a Sierra Exception

When deriving a new exception from the framework exceptions and using the put-to (`<<`) operator, you must include the following template functions in your class or

```
    throw MyError() << "My error message cause of " << x;
```

will certainly fail. It only took me a day to relearn this, but:

If `MyError` does not implement a put-to operator, it uses the base classes put to operator, say `RuntimeError`. Well, the `RuntimeError` put-to operator returns a reference to a `RuntimeError`, so now you are going to be throwing a `RuntimeError` exception not a `MyError` exception. This is only an issue when using the temporaries. By putting the `MyError()` construction on the same line as the throw. I.e.,

```
  MyError x;
  x << "My error message cause of " << x;
  throw x;
```

works fine since we are actually throwing `x`, a `MyError` object.

So, add these if you derive from a sierra exception and wish to throw the temporary object and use the put operator all in one pretty line:

```
    class MyError : public RuntimeError {

    .
    .
    .
      inline MyError& operator<<(ExceptionString& (*f)(ExceptionString
    &)) {
        f(*this);
        return *this;
      }

      template <class T>
      inline MyError &operator<<(const T &t) {
        RuntimeError::operator<<(t);
        return *this;
      }
    };
```

23.13.9 `path_name()`

When generating error messages with object names, use the `path_name()` form rather than just name. This generates a dot (`.`) separated list of names from the `Procedure` on down.

23.13.10 `abort()` – Don't use it

`abort()` does not provide any useful information when the application dies. By replacing abort calls with Warnings, Dooms and Exceptions, the code will be easier to maintain and the user will get better diagnostic output.

23.13.11 `Apub_Parser_Base` – Useful for parsing, not needed for error reporting

This class stashes useful information from within a parser that is likely to be needed after parsing. The runtime error reporting routines handle error reporting making the line number and command value information redundant for error reporting.

23.14 Diagnostic Writer How-To

Editorial note: This How-To is mostly taken from Dave Bauer's collection of How-Tos. Minor changes include formatting, editorial corrections and possibly additioinal information related to Aria.

This document briefly describes the implementation of the diagnostic writer and masked based output. The diagnostic writer is intended to replace debug level output with but masked based diagnostic output. By utilizing the diagnostic writer, your normal output can be separated or interleaved with the diagnostic output, the diagnostic output can the enabled/disabled at specific times during a run, and entire classes can be output by simply putting the object to the diagnostic writer.

23.14.1 Output

Application output falls into three classes. Normal execution output, warning and error output, and diagnostic output. Normal execution output is handled via the `Env::outputP0()` and `Env::output()` functions and by the application diagnostic writer. Warning and error output is handled by the `RuntimeWarning`, `RuntimeDoomed` and runtime exception classes (`RuntimeError`, `LogicError`, etc), which is eventually written to the env output stream and the diagnostic outptu stream. And diagnostic output is for selected operationally descriptive output.

In many cases, the diagnostic output is utilized as the primary output stream when selectable level of user output is desired. With thay in mind, an InfoWriter class has been written. However, additional discussion is required to complete the design and implementation of this class for primary application output.

23.14.2 Diagnostic Writer

The diagnostic writer allows you to write diagnostic information to the diagnostics stream by specifying the content from the command line or the input deck. It a debug level built on a bit mask.

Since there are many applications and libraries, there are several diagnostic writers. Each diagnostic writer has its own bit mask, command line parser and writer. However, they all share a common diagnostic stream. So, output from each diagnostic writer is properly interleaved.

Each application defines it own diagnostic writer. This is generally defined within the `app_DiagWriter_fwd.h`, `app_DiagWriter.h`, `app_DiagWriter.C` files. The `app_DiagWriter_fwd.h` file defines the `LOG_xxx` bit assignments. These values are used to specify the type of message to be written. The `app_DiagWriter.h` rarely needs to be modified. It declares the daignostic writer for the application or library. The `app_DiagWriter.C` files defines the parser which provides names for the `LOG_xxx bit` masks.

23.14.3 Using The Diagnostic Writer

To have your program send output whenever a specified log bit is set, add the following to your code:

```
dwout.m(LOG_xxx) << "description, " << value << std::endl;
```

or, if much computation or MPI is involved:

```
if (dwout.shouldPrint(LOG_xxx)) {
  dout << "really_spendy_function(), " <<
  really_spendy_function() << std::endl;
}
```

Where `dwout` is the name of the diagnostic writer, `LOG_xxx` is the bit which describes the type of message, `description` is a description of the data. value is the value of interest. And, `std::endl` ends the message.

Note that nearly all (please let me know what's missing) containers have output operators. So to write an entire vector, just write the vector variable, the diagnostic writer will take case of the rest.

23.14.4 Turning on the LOG_xxx Bits

Each application has its own command line option and line commands for flipping on the bits. The command line option has limited functionality in that the parameters cannot be switched on and off during an application execution. However, it is useful for a quick look.

For sierra framework, the command line option is `-m`, and each application has it's own option name. Use the `-h` option to display a table of command line options and parameters.

The `Diagnostic Control` command block in the sierra block controls each the diagnostic writers:

```
Begin Diagnostic Control <diagwriter>
   From time t0 to t1 enable <parameters>
   From step s0 to s1 enable <parameters>
   On condition c enable <parameters>
   Enable <parameters>
End Diagnostic Control <diagwriter>
```

Please refer to Diagnostic Control for implementation details.

The diagnostic output can be selectively enabled based on time, step or an application specified condition. During the application's procedure execution loop, the diagnostic controller evaluates the enclosed line commands in the order specified in the input deck. The diagnostic options specified in the first line command that meets its criteria are applied.

Since control parameters are only applied when the criteria is met, it is important to include an `ENABLE` line command with the base settings to be applied as a baseline.

Table 23.14.4 lists some of the options available for the framework diagnostic writer `fmwkout` and options for the Aria diagnostic writer `arialog` are given in table 23.14.4. Other application will likely implement additional diagnostic writers.

Option	Description
`error`	Display error messages
`warning`	Display warning messages
`members`	Display data structure members
`timer`	Display execution time during trace
`trace`	Display execution trace
`contact`	Display contact diagnostic information
`geometry`	Display geometry diagnostic information
`scontrol`	Display solver control diagnostic information
`search`	Display search diagnostic information
`transfer`	Display transfer diagnostic information
`parser`	Display parser diagnostic information
`parameters`	Display parameter diagnostic information
`io`	Display contact I/O information
`plugins`	Display user function and plugin diagnostic information

Table 23.1. Diagnostic writer options for Framework (`fmwkout`).

Option	Description
`bc`	Display boundary condition information
`debug`	Display extra debugging information
`eq`	Display equation information
`expression`	Display Expression information
`hadapt`	Display h-adaptivity information
`linsolve`	Display linear solver information
`nonlinear`	Display nonlinear solver information
`pp`	Display postprocessor information
`sens_check`	Enable the Expression Newton sensitivity checker
`species`	Display species information
`transfer`	Display transfer related information
`plugin`	Display plugin information

Table 23.2. Diagnostic writer options for Aria (`arialog`).

23.14.5 Diagnostic Stream

The disagnostic stream specified the output destination for all the active diagnostic writers. The stream allows the output from the diagnostic writers to be interleaved.

The command line option

```
  -dout <destination>David,
```

and the sierra block line command

```
  diagnostic stream <destination>
```

determines the output. `<destination>` may be `cout`, `cerr`, `outputp0`, `output` or a path. If a path is given, each processor creates its own file suffixing the path with `.n.p` where `n` is the number of processors and `p` is the rank of each processor. See Diagnostic Stream for specifying the output destination.

23.14.6 Coding Objects

To code your own objects to play with the diagnostic writer, you only need to add a `verbose_print()` member function to your class. And, a `operator<<()` function to your namespace. Naturally the implementation for `verbose_print()` is generally in the `.C` file, not the header.

```
  class MyClass : public ParentClass
  {
    public:
    DiagWriter &verbose_print(DiagWriter &dout) const {
      if (dout.shouldPrint()) {
dout << "MyClass " << m_name << push << std::endl;
ParentClass::verbose_print(dout).std::endl();
dout << "m_var1, " << m_var1 << std::endl;
dout << "m_var2, " << m_var2 << std::endl;
dout << "m_ptr1, " << c_ptr(m_ptr1) << std::endl;
dout << "m_ptr2, " << c_ptr_name(m_ptr2) << std::endl;
dout << pop;
      }
      return dout;
    }
  };

  inline DiagWriter &operator<<(Diagwriter &dout, const MyClass
  &my_class) {
    return my_class.verbose_print(dout);
  }
```

If your class is polymorphic, be sure to define `verbose_print()` as `virtual`.

When writing a subclass from your class, the put-to operator (<<) will by do what you expect, eevn when you cast. So you need to use the direct call form.

23.14.7 Writing Containers

You can write an STL container by simple putting it to the diagnostic writer. This is implemented using templates in `utility/include/DiagWriter.h`.

23.14.8 Writing Pointers

Writing of pointers is usually quite ugly since you need to check if the pointer is null first. Instead, use the `c_ptr()` function. If the pointer is null it writes "(pointer), ¡not created¿". Or, you can use `c_ptr_name()` function which will call the `name()` function of the pointed to object if the pointer is not null.

You can have you own pointed object function called by replacing name with your member function name below.

```
template <class T>
inline c_ptr_func_<T, const String &> c_ptr_name(const T *t) {
  return c_ptr_func_<T, const String &>(t, &T::name);
}
```

23.14.9 Diagnostic Control

The diagnostic control block in the sierra command block is handled by the `Fmwk::DiagControl::Control` class. Simply add

```
Fmwk::DiagControl::Control diag_control(step_cntr(),
time(Fmwk::STATE_OLD), 0);
```

at the beginning of your main procedure control loop. This line exists in the Solver Control procedure.

23.15 Timers and Timing How-To

Editorial note: This How-To is mostly taken from Dave Bauer's collection of How-Tos. Minor changes include formatting, editorial corrections and possibly additioinal information related to Aria.

This document briefly describes the time metrics collection features available to sierra applications.

The `DiagTimer` and `Timer` classes provide runtime metric information for properly rigged objects.

The system has a root "System" timer. This timer is started when `run_sierra()` is called and stopped before the successful completion information is displayed.

23.15.1 DiagTimer and Timer

The `DiagTimer` class implements the basic developer interface to the timers. Generally, the framework form, `Timer`, will be used to implement timers within a framework derived class.

Timers are intended to be members of your classes or static objects created within a function or member function. Each timer has a name, a parent and a timer class. The name is used to find a child timer of a parent and to display the collected metrics. The parent is used to build the hierarchy of metrics gathered in a application. And the timer class or type categorizes the timers so they may be enabled, disabled and selected for display.

The timing information is actually maintained in a separate tree. So, during timer construction within a class, the timer is a reference to the real timing metric information in the tree. This design means that timers are not destroyed when an object is destroyed.

The timers are hierarchical by name. So, by giving a timer a unique name among its siblings, each object has it's own timer and it's children timers are also unique.

23.15.2 Add a Timer to a Class

To add a timer to a class, include the header file and add the timer as a member of the class. Then, during construction initialization, specify the timer's name and parent timer. You can also specify a timer class if it is different from the parents.

Then, to start/stop the timer, use the TimeBlock and TimeBlockSynchronized to ensure that a started timer is stopped.

```
#include <util/Timer.h>

class MyClass
{
public:
    // etc.

    Timer &getMyTimer() {
        return m_myTimer;
    }

    Timer &getMySubTimer() {
        return m_mySubTimer;
    }

private:

    Timer m_myTimer;
    Timer m_mySubTimer;
    // etc.
};


MyClass::MyClass(
    Region & region)
  : m_myTimer("My Timer", region.getRegionTimer()),
    m_mySubTimer("My SubTimer", m_myTimer)
{
    // etc.
}
```

```
// When controlling timers using the TimeBlocks, be sure to give the
// object a name.  Some compilers destroy unnamed objects immediately,
// other destroy them at the end of the block. I usually use timer__

MyClass::someFunction()
{
    Timer::TimeBlock timer__(m_myTimer); // Metrics in block are
 //   collected to m_myTimer;
}


MyClass::someFunction()
{
    Timer::TimeBlockSynchronized timer__(m_myTimer);
//  MPI_Barrier then start timer
}


MyClass::someOtherFunction()
{
    m_myTimer.start(); // Works, but is dangerous if stop() is not called.
    m_myTimer.stop();
}
```

23.15.3 Adding a Timer to a Function

To add a timer to a function, create a static Timer in the function, then start/stop it using the Timer::TimeBlock. Note that if there is no parent specified, the root timer "System" will be the timer's parent.

```
int
myFunction()
{
    static Timer my_timer("My Timer");
    Timer::TimerBlock(my_timer);
    // etc.
}
```

23.15.4 Getting Information from a Timer

Each timer has an accumulation time/count, a checkpoint time/count and a recent lap time/count. The accumulation time is the overall time/count since the start of the application. The checkpoint time/count records the current values when set, then the difference from that time/count can be obtained. This is useful for displying delta times for interations. The lap time is the time accumulated during the last `start()`/`stop()` cycle for the timer.

The metrics available are `getCPUTime()`, `getWallTime()`, `getFlopCount()`, `getIOCount()`, `getMsgCount()`. Flop, IO, and Msg are not implemented on most platforms (any really).

```
m_myTimer.getCpuTime().getStart(); // Timer most recent start time
m_myTimer.getCpuTime().getStop(); // Timer most recent stop time
m_myTimer.getCpuTime().getLap(); // Most recent stop - start
m_myTimer.getCpuTime().getTotal(); // Accumulated time
m_myTimer.getCpuTime().getTotal(false); // Accumulated time less checkpoint time
```

I could add `getCheckpoint()` and `getCheckpointTotal()` as variations on a theme, but...

23.15.5 Displaying the Timers

To display a table of the collected timing information, use the `Timer::printTable()` function. The function lets you specify the classes and the metrics to output. Note that only enabled timers and metrics are displayed. Use `TIMER_CHECKPOINT` to display checkpointed time. It also resets the checkpoint time/count after display.

```
std::cout << Timer::printTable(TIMER_CPU | TIMER_ALL);
```

23.15.6 Enabling/Disabling the Timers

Timers are enabled by timer class. They can be enabled using the `Timer::setEnabledTimers()` function, by the `-timer` command line option or by the `ENABLE TIMER` input deck line command.

23.15.7 Timers in the Framework

The framework creates several timers and starts/stops them when it has control of the operations.

header timers are given the name associated with the name of the object automatic timers are handled within framwork and required no additional coding otherwise, instructions are given on how to collect the data for the timer

```
Domain:
  Domain                 header
  LinearSystem           automatic
  Initialize             automatic
  Execute                automatic
  Load                   automatic
  Solve                  automatic

  Procedure:
    Procedure            header
    Initialize           automatic
    Restart              automatic
    Execute              automatic
    MeshInput            automatic
    MeshOutput           automatic
    Transfer             automatic

    Region:
      Region             header
      Initialize          call Timer::TimeBlock timer__(getRegionInitializeTimer()); at beginning
      Execute             call Timer::TimeBlock timer__(getRegionExecuteTimer()); at beginning of

      Mechanics:
        Mechanics         header

        Algorithm:
          Algorithm       header
          Apply           automatic

        WorksetAlgorithm: (sub class of Algorithm)
          Gather          automatic
          ScatterAssemble automatic

      NonLinearCoupler:
        NonLinearSolver   automatic
        Initialize        automatic
        Scatter           automatic
        Solve             automatic

      NonLinearSolver:
        NonLinearSolve    automatic
        Initialize        automatic
        Scatter           automatic
        Solve             automatic

  UserInputFunction:
    UserInputFunction     automatic
```

Glossary

coefficient: Each *field* that is represented by a basis function expansion has a set of coefficients that are used in that expansion. For example, the three dimensional velocity vector represented by an eight node, tri-linear hex element has eight coefficients. In this example, each coefficient is also a vector with three *components.*

This is consistent with the Sierra data model wherein vectors and tensors are single-entity data types.

Although for certain basis function representations the coefficients may be the exact value of the *field* at a point this is not the case in general.

component: The number of values required to describe a *field* at a point. Equal to the tensor dimension raised to the power of the tensor order. For example, temperature has one component, velocity has 3 components (in 3 dimensions) and stress has 9 components.

dof: An entry in the vector of uknowns, i.e. , in the linear solver solution vector.

field: The physical variable of interest, e.g. , temperature or velocity.

multidof: A *field* is a multidof if it contains more than one *component*

References

Ted Belytschko, Wing Kam Liu, and Brian Moran. *Nonlinear Finite Elements for Continua and Structures*. John Wiley and Sons, 2004. 3.11, 1, 2, 10.21.3, 10.29.3

T. D. Blake and J. De Coninck. The influence of solid-liquid interactions on dynamic wetting. *Adv. Colloid and Interface Sci.*, 96:21–36, 2002. 6.14.19

Pavel B. Bochev, Clark R. Rohrmann, and Max D. Gunzburger. Stabilization of low-order mixed finite elements for the Stokes equations. *SIAM J. Numer. Anal.*, 44(1):82–101, 2006. 4.14

Erik Boman, Karen Devine, Robert Heaphy, Bruce Hendrickson, William Mitchell, Matthew St. John, and Courtenay Vaughan. Zoltan home page. `http://www.cs.sandia.gov/Zoltan`, 1999. 18

Javier Bonet and Richard D. Wood. *Nonlinear Continuum Mechanics for Finite Element Analysis*. Cambridge University Press, 1997. 3.11, 1, 10.21.3, 10.29.3

R. A. Cairncross, P. R. Schunk, T. A. Baer, R. R. Rao, and P. A. Sackinger. A finite element method for free surface flows of incompressible fluids in three dimensions. part i. boundary fitted mesh motion. *Int. J. Numer. Methd. Fluids*, 33:375–403, 2000. 6.14.11

Ken S. Chen, Gregory H. Evens, Richard S. Larson, David R. Noble, and William G. Houf. Final report on LDRD project: A phenomenological model for multicomponent transport with simultaneous electrochemical reactions in concentrated solutions. SAND 2000-0207, Sandia National Laboratories, Albuquerque, NM 87185, USA, January 2000. 8.2.8

W. M. Deen. *Analysis of Transport Phenomena*. Topics in Chemical Engineering. Oxford University Press, New York, 1998. 3.1

Karen Devine, Bruce Hendrickson, Erik Boman, Matthew St. John, and Courtenay Vaughan. *Zoltan: A Dynamic Load Balancing Library for Parallel Applications; User's Guide*. Sandia National Laboratories, Albuquerque, NM, 1999. Tech. Report SAND99-1377 `http://www.cs.sandia.gov/Zoltan/ug_html/ug.html`. 18

Karen Devine, Erik Boman, Robert Heaphy, Bruce Hendrickson, and Courtenay Vaughan. Zoltan data management services for parallel dynamic applications. *Computing in Science and Engineering*, 4(2):90–97, 2002. 18

Clark R. Dohrmann and Pavel B. Bochev. A stabilized finite element method for the Stokes problem based on polynomial pressure projections. *Int. J. Num. Meth. Fluids*, 46:183–201, 2004. 4.14

D. K. Gartling. NACHOS II - a finite element computer program for incompressible flow problems. part I - theoretical background. Technical Report SAND86-1816, Sandia National Labs, Albuquerque, NM, USA, April 1986. 12.5

T. Hughes, L. P. Franca, and M. Balestra. A new finite element formulation for computational fluid dynamics: V. circumventing the babuska-brezzi condition: a stable petrov-galerkin formulation of the stokes problem accomodating equal-order interpolations. *Comput. Methods Appl. Mech. Engrg.*, 59:85–99, 1986. 4.14

I. M. Krieger. Rheology of monodisperse latices. *Adv. Colloid Interface Sci.*, 3:111–136, 1972. 10.40.8

Lawrence E. Malvern. *Introduction to the Mechanics of a Continuous Medium*. Series in Engineering of the Physical Sciences. Prentice-Hall, Upper Saddle River, NJ, USA, 1969. 3.11, 1, 3

George E. Mase. *Theory and Problems of Continuum Mechanics*. Schaum's Outline Series. McGraw-Hill, New York, NY, USA, 1970. 3.11, 1, 3, 4, 6, 7

P. K. Notz, S. R. Subia, M. H. Hopkins, and P. A. Sackinger. A novel approach to solving highly coupled equations in a dynamic, extensible and efficient way. In M. Papadrakakis, E. Onate, and B. Schrefler, editors, *Computation Methods for Coupled Problems in Science and Engineering*, page 129, Barcelona, Spain, April 2005. Intl. Center for Num. Meth. in Engng. (CIMNE). 23.1

Patrick K. Notz. SourceForge project page for Aria. `https://sourceforge.sandia.gov/projects/aria/`, a. 1.1

Patrick K. Notz. Aria home page. `http://aria.sandia.gov/`, b. 1.1

Bjarne Stroustrup. *The C++ Programming Language, Special Edition*. Addison Wesley, Reading, MA, 2000. 23.10.1

The SNTools Project. SNTools SourceForge Project. Online. 2.1, 2.2

Index

DISTRIBUTION:

2	MS 9018 Central Technical Files, 8944
2	MS 0899 Technical Library, 4536
2	MS 0619 Review & Approval Desk, 4916