Altair® AcuSolve® 2025

Altair AcuSolve Command Reference Manual

Updated: 11/18/2024

Contents

AcuSolve Command Reference Manual 6

Command Format 7

Functional Commands 8
Declarative Commands 9

Command Qualifier 10

Parameter Format 11

String 12
Enumerated 13
Boolean 14
Integer 15
Real 16
List 17
Array 19

Parameter Operators and Functions 20

File and Data Format/Reading Data From Files 23

ASCII (Text) Files 25
Binary Files 27
Converting Between ASCII and Binary Files 28
Relevance of File Extensions 29

Alternative Command Organization 30

Global Commands 35

ANALYSIS 36
CONSTRAINT 39
DESIGN VARIABLE 41
DESIGN VARIABLES FIELD 42

EQUATION....44
EXTERNAL_CODE....61
FIELD_BOUNDARY_CONDITION....65
FLEXIBLE_BODY....66
IMMERSED_BOUNDARY_SURFACE....78
MESH_MOTION....80
MULTIPLIER_FUNCTION....97
NODAL SHAPES....107
OBJECTIVE....109
OPTIMIZATION....110
PARTICLE_TRACE....113
RADIATION....114
REFERENCE_FRAME....116
RESPONSE VARIABLE....119
SOLAR_RADIATION....126
THERMAL_SHELL....131
TURBULENCE_MODEL_PARAMETERS....133
USER_GLOBAL_DATA....142
WAVE_BOUNDARY_CONDITION....143
WAVE_DAMPING_SOURCE....145

Solution Strategy Commands....146

ALGEBRAIC_MULTIGRID_PARAMETERS....147
AUTO_SOLUTION_STRATEGY....170
CONVERGENCE_CHECK_PARAMETERS....177
LINEAR_SOLVER_PARAMETERS....191
STAGGER....198
TIME_ INCREMENT....210
TIME_INTEGRATION....215
TIME_SEQUENCE....219

Body Force Commands....225

BODY_FORCE....226
GRAVITY....229
MASS_HEAT_SOURCE....236
MASS_SPECIES_SOURCE....242
MOMENTUM_SOURCE....248
ROTATION_FORCE....254
VOLUME_HEAT_SOURCE....256
VOLUME_SPECIES_SOURCE....263

Material Model Commands....269

BATTERY_MODEL....270

CONDUCTIVITY_MODEL 280
CONTACT_ANGLE_MODEL 291
DENSITY_MODEL 293
DIFFUSIVITY_MODEL 300
ELECTRICAL_RESISTIVITY_MODEL 305
EMISSIVITY_MODEL 308
GAS_KINETIC_MODEL 312
FIELD 314
FIELD_INTERACTION_MODEL 315
MATERIAL_MODEL 329
MATERIAL_RADIATION_MODEL 334
MULTI_FIELD_MODEL 336
POROSITY_MODEL 339
SOLAR_RADIATION_MODEL 347
SPECIFIC_HEAT_MODEL 351
SURFACE_TENSION_MODEL 358
VISCOELASTIC_MODEL 361
VISCOSITY_MODEL 365

Nodal Data Commands 376

COORDINATE 377
DERIVED_QUANTITY_OUTPUT 379
ERROR_ESTIMATOR_OUTPUT 380
EXTERNAL_OUTPUT 383
NODAL_INITIAL_CONDITION 389
NODAL_BOUNDARY_CONDITION 402
NODAL_RESIDUAL_OUTPUT 427
NODAL_OUTPUT 429
PERIODIC_BOUNDARY_CONDITION 431
RESTART_OUTPUT 446
RUNNING_AVERAGE_OUTPUT 448
TIME_AVERAGE_OUTPUT 449
TIME_HISTORY_OUTPUT 451

Element Data Commands 454

BATTERY_COMPONENT_MODEL 456
BATTERY_MODULE 457
BATTERY_PACK 470
BATTERY_THERMAL_RUNAWAY_MODEL 473
CAA_ELEMENT_OUTPUT 477
CAA_OUTPUT 479
CAA_SURFACE_OUTPUT 488
ELEMENT_BOUNDARY_CONDITION 490
ELEMENT_OUTPUT 505

ELEMENT_SET....509
EXTERNAL_CODE_SURFACE....526
FAN_COMPONENT....532
FEEDBACK_CONDITION....544
FIELD_BOUNDARY_CONDITION....547
FREE_SURFACE....550
FWH_SURFACE_OUTPUT....554
FWH_OUTPUT....556
GUIDE_SURFACE....557
HEAT_EXCHANGER_COMPONENT....562
INTERFACE_SURFACE....576
INTERPOLATED_MOTION_SURFACE....581
LINE_SOURCE....585
MESH_BOUNDARY_CONDITION....590
PARTICLE_SURFACE....600
RADIATION_SURFACE....604
SIMPLE_BOUNDARY_CONDITION....611
SOLAR_RADIATION_SURFACE....663
SURFACE_INTEGRATED_CONDITION....669
SURFACE_OUTPUT....674
SURFACE_SET....679
TURBULENCE_WALL....682
VAO_ELEMENT_OUTPUT....689
VAO_OUTPUT....691
VAO_SURFACE_OUTPUT....692
VOLUME_SET....694

Functional Commands....698

RUN....699
RESTART....701
INCLUDE....705
ASSIGN....706
QUIT....707

Intellectual Property Rights Notice....708
Technical Support....714

Index....715

AcuSolve Command Reference Manual

1

AcuSolve command descriptions and corresponding examples.

AcuSolve is a general purpose, pressure based incompressible and compressible flow solver. Its technology is based on the Galerkin/Least-Squares (GLS) finite element method. It uses unstructured meshes of tetrahedron, pyramid, wedge and hexahedron elements, with nodal-based field variables. To solve a problem with AcuSolve, you must first generate an unstructured mesh. Any third-party mesh generator may be used for this purpose. Given a mesh, consisting of nodal coordinates, elements, and boundary surfaces and nodes, you can proceed to create an input file.

The input file consists of one or more commands, each having zero or more parameters. These commands define the problem parameters such as nodes, elements, boundary conditions, output, material models and solution strategies.

Command Format

2

The commands have the following general syntax.

This chapter covers the following:

- Functional Commands (p. 8)
- Declarative Commands (p. 9)


```
COMMAND ( qualifier ) {
 parameter1 = value1
 ...
 parameterN = valueN
}
```

where COMMAND is the name of the command, such as ANALYSIS or MATERIAL_MODEL; qualifier, including the surrounding parentheses, is a mandatory qualifier of the command; *parameter1* to *parameterN* are the optional parameters of the command; and *value1* to *valueN* are the corresponding values, which may be integers, reals, strings or arrays, of the parameters. An optional semicolon may be placed between parameters for clarity. If a command is given with no option, the brackets "{" and "}" may be omitted.

For example:

```
RUN
```

The input file is free format. Each command may be specified on a single line or spread across multiple lines. With the exception of double-quoted strings, all arguments are case insensitive. By convention, we write commands in upper case, for example, COORDINATE, parameters and their values in lower case, for example, type=constant_prandtl_number, and capitalize function names, for example, Read(...).

Except within double-quoted strings, all text after and including a pound (hash) sign is treated as a comment, hence ignored. For example,

```
ANALYSIS {
  title = "Example problem # 1"
  mode = static       # run as static for now
  # change to dynamic for final run
}
```

The "#" in "Example problem # 1" is a part of the string value of the title. The other two hash signs and the text that follows them are treated as comments.

There are two types of commands: Functional Commands and Declarative Commands.

Functional Commands

These commands perform operations at the time they are encountered in the file.

These commands include:

- `AUTO_SOLUTION_STRATEGY`
- `RUN`
- `RESTART`
- `INCLUDE`
- `ASSIGN`
- `QUIT`

The placement of these commands in a file is important.

Declarative Commands

These commands are used to define problem parameters. They are order-independent and may be placed anywhere in the file relative to each other.

For example, the `NODAL_BOUNDARY_CONDITION` command may be specified before or after the `COORDINATE` command, even though the former depends on the latter. However, the placement of these commands relative to the functional commands is important. For example in:

```
RESTART {
 from_problem = "channel"
}
NODAL_INITIAL_CONDITION( temperature ) {
 default_value = 20
}
```

the problem starts with temperature values of 20 degrees, whereas in

```
NODAL_INITIAL_CONDITION( temperature ) {
 default_value = 20
}
RESTART {
 from_problem = "channel"
}
```

the problem starts with the temperature values given in the restart file.

Command Qualifier

Some commands require a qualifier. The qualifier is used to distinguish one use of the command from another.

For example, two different material models are distinguished by their qualifiers:

```
MATERIAL_MODEL( "air" ) {
 density_model = "air at standard atmosphere"
 viscosity_model = "air"
}
MATERIAL_MODEL( "aluminum" ) {
 density_model = "aluminum"
 conductivity_model = "aluminum"
}
```

The specification of the qualifier is mandatory for commands that accommodate it and is an error for those commands that do not accommodate one. There are two types of qualifiers: user-given name and enumerated. The user-given name qualifier is any user-specified name, as in the above `MATERIAL_MODEL` examples. The user-given name of a command may be referenced by another command. For example:

```
ELEMENT_SET( "channel" ) {
 material_model = "air"
}
```

The enumerated type is for selecting a specific value from a predefined set of options. For example, consider:

```
NODAL_INITIAL_CONDITION( velocity ) {
 default_values = { 1, 0, 0 }
}
```

and

```
NODAL_INITIAL_CONDITION( pressure ) {
 default_value = 0
}
```

In the first command, the initial conditions are applied to velocity and in the second command they are applied to pressure. `NODAL_INITIAL_CONDITION` is currently the only command that uses the enumerated type of qualifier.

Parameter Format

4

Each command may have zero or more parameters. Most command parameters have default values.

This chapter covers the following:

- String (p. 12)
- Enumerated (p. 13)
- Boolean (p. 14)
- Integer (p. 15)
- Real (p. 16)
- List (p. 17)
- Array (p. 19)

The parameter values are persistent, in the sense that they change only if the command is issued with that parameter. For example, the density parameter in `DENSITY_MODEL` has a default value of one. If you issue the following commands:

```
DENSITY_MODEL( "air" ) {
 density = 1.225
}
DENSITY_MODEL( "air" ) {
 expansivity = 0.0039
}
```

the value of the density parameter of the density model "air" is 1.225, the value of expansivity is 0.0039 and the remaining parameters have their respective default values.

Each parameter has one of the following seven types:

- String
- Enumerated
- Boolean
- Integer
- Real
- List
- Array

String

This is a user-given string, which is typically used for referencing another command.

The string is any set of characters enclosed in a pair of double-quotes, for example, "My #1 String". The string may not exceed 1023 characters. For example, in

```
ELEMENT_BOUNDARY_CONDITION( "inflow" ) {
 element_set = "channel"
}
```

element_set references the element set "channel" given above. The quotes surrounding a string may be omitted if the string contains only alphanumeric and underscore characters. However, strings that are not quoted are automatically converted to lower case and thus this practice is not recommended.

Enumerated

This is a set of options available for a given parameter.

For example, the shape parameter of the `ELEMENT_SET` command accepts the following values: *four_node_tet*, *five_node_pyramid*, *six_node_wedge*, and *eight_node_brick*. Hence, you can write:

```
ELEMENT_SET( "channel" ) {
 shape = eight_node_brick
}
```

Many enumerated values have abbreviations. For example, *eight_node_brick* may be abbreviated as hex8.

Boolean

A boolean parameter turns an option on or off.

For example,

```
TIME_INCREMENT {
 auto_time_increment = on
}
```

turns on the automatic time increment options. Boolean values *on*, *yes*, and *true* are equivalent; likewise boolean values *off*, *no*, and *false* are equivalent.

Integer

This is an integer value given to a parameter.

For example,

```
NODAL_OUTPUT {
 output_frequency = 10
}
```

specifies that the nodal values are to be output every 10 time steps. Some integer parameters may be confined to a valid range. For example, the output frequency must be greater than or equal to zero, where zero indicates that this option is inactive.

Real

This is a floating point value given to a parameter.

For example,

```
NODAL_OUTPUT {
 output_time_interval = 0.3
}
```

specifies that the nodal values are to be output every 0.3 time units and at the last time step. Similar to integer parameters, real parameters may be confined to a valid range. For example, the *output_time_interval* must be greater than or equal to zero, where zero indicates that this option is inactive.

List

Some parameters require reference to a list of user-specified commands.

For example,

```
TIME_SEQUENCE {
staggers = { "my flow stagger", "my turbulence stagger" }
}
```

specifies that the user staggers *"my flow stagger"* and *"my turbulence stagger"* should be executed in that order.

Some commands, such as `SIMPLE_BOUNDARY_CONDITION`, require a list of references that can be rather long. For example, a complex model may have many surface sets that belong to a single wall boundary condition:

```
SURFACE_SET("wall - face 1"){
 type = tri3
 surfaces = Read("wall-face_1.srf")
...
}
SURFACE_SET("wall - face 2"){
 type = tri3
 surfaces = Read("wall-face_2.srf")
...
}
SURFACE_SET("wall - face 3"){
 type = tri3
 surfaces = Read("wall-face_3.srf")
...
}
SURFACE_SET("wall - face 4"){
 type = tri3
 surfaces = Read("wall-face_4.srf")
...
}
SIMPLE_BOUNDARY_CONDITION("wall"){
 surface_sets = {  "wall - face 1",
        "wall - face 2",
        "wall - face 3",
        "wall - face 4 }
 type = wall
...
}
```

To improve the readability of the input file, the list of surface sets can be placed in a file, say *"wall_surface_sets.srfst"*:

```
wall - face 1
wall - face 2
wall - face 3
wall - face 4
```

Then, this file can be referenced in the `SIMPLE_BOUNDARY_CONDITION` command as follows:

```
SIMPLE_BOUNDARY_CONDITION("wall"){
 surface_sets = Read("wall_surface_sets.srfst"
```

```
  type = wall
...
}
```

Each line in *"wall_surface_sets.srfst"* represents a different surface set. It is not necessary to place quotation marks around the strings that are entered in the file.

Array

Some parameters require an array of integers, floating point numbers, or a mixture of both.

For example, in

```
GRAVITY( "simple gravity" ) {
 gravity = { 0, -9.81, 0 }
}
```

the *gravity* parameter requires a three-component vector of floating point numbers. This parameter requires exactly three numbers separated by commas. An array may be two dimensional. For example, in

```
PERIODIC_BOUNDARY_CONDITION( "a PBC" ) {
 rotation_axis = { 0, 0, 0 ; 0, 0, 1 }
}
```

rotation_axis is defined by two points in space. The axis of rotation is in the z-direction, passing through the points (0,0,0) and (0,0,1). The rows of a two dimensional array are separated by semicolons. In the above example, *rotation_axis* is a 2x3 array. The array may also have mixed integer and floating point values. For example,

```
COORDINATE {
 coordinates = { 1, 0.0, 0.0, 0.0 ;
      2, -1.5, -1.5, -1.5 ;
 ... }
}
```

defines the nodal coordinates of the problem. Here the first column is the node number and columns two, three, and four define the x, y, and z components of the coordinates. The number of columns must be at least four and the number of rows defines the number of nodes in the system. The values of an array may be read from an ASCII file:

```
COORDINATE {
 coordinates = Read( "channel.crd" )
}
```

where each line of the file `"channel.crd"` must contain an integer followed by three floating point numbers separated by white space (one or more spaces or tabs). No commas or semicolons may be embedded in the file. For example, the file `"channel.crd"` may contain:

```
1 0.0 0.0 0.0
2 -1.5 -1.5 -1.5
...
```

Parameter Operators and Functions

5

Parameter types of integer, real, and array may be any valid arithmetic expression. Valid expressions consist of the operators +, -, *, /, ^ (for power), parentheses, a set of built-in functions and a set of built-in or user-assigned variables, specified by the `ASSIGN` command.

The built-in functions are:

Abs(x)	Absolute value of x
Acos(x)	Principal value of arc cosine of x
Asin(x)	Principal value of arc sine of x
Atan(x)	Principal value of arc tangent of x
Atan2(x,y)	Principal value of arc tangent of y/x
Ceil(x)	Smallest integer greater than or equal to x
Cos(x)	Cosine of x
Cosh(x)	Hyperbolic cosine of x
Env("s")	Import environment variables
Exp(x)	Exponential of x
Floor(x)	Largest integer not greater than x
Log(x)	Natural log of x
Log10(x)	Base-ten log of x
Max(x,y)	Maximum of x and y
Min(x,y)	Minimum of x and y
Mod(x,y)	Remainder of x/y
Pow(x,y)	Value of x raised to the power of y (equivalent to x^y)
Sin(x)	Sine of x
Sinh(x)	Hyperbolic sine of x
Sqrt(x)	Square root of x
Tan(x)	Tangent of x

Tanh(x)	Hyperbolic tangent of x

The built-in variables are:

E	2.7182818284590452
PI	3.1415926535897931
EPS	Machine precision, typically 2.2204460492503131E-16

The expressions are evaluated at the time they are encountered in the input file; they are then replaced by their values. The expressions are not transferred to restart files; only their values are. For integer parameters, the final value is converted to an integer. For example,

```
ASSIGN {
    variable                    = SHIFT
    value                       = Sin (PI/8 + Asin(.3) )
}
MULTIPLIER_FUNCTION( "shifted half sin" ) {
    type                        = cubic_spline
    curve_fit_variable          = time
    curve_fit_values            = {  0.0, SHIFT + Sin(0.0*PI) ;
                                     0.1, SHIFT + Sin(0.1*PI) ;
                                     ...
                                     1.0, SHIFT + Sin(1.0*PI) ; }
}
```

The special function Env is used to import the value of an environment variable and assign it to a parameter, of any type. Environment variables are typically set at the shell prompt prior to executing a command. The syntax of setting an environment variable depends on the shell and operating system. For example, in a Unix cshell an environment variable is set using the setenv command:

```
cshell-prompt> setenv INLET_VELOCITY 20
```

The input file may then contain

```
SIMPLE_BOUNDARY_CONDITION( "inlet" ) {
    ...
    x_velocity = Env( "INLET_VELOCITY" )
```

This feature may be combined with any valid arithmetic expression as described above.

The Env function is particularly useful for performing parametric studies. For example, an investigation of the effect of inlet velocity may be automated with the following cshell loop:

```
foreach vel ( 5 10 15 20 )
    echo "Processing velocity " $vel " ..."
    setenv INLET_VELOCITY $vel
    acuRun
    acuTrans -out -to stats > STATS.$vel
end
```

The operator "." may be used to concatenate strings. This is particularly useful in connection with an Env function. For example, you can define the PROBLEM environment variable with the following command in a bash shell under Linux:

```
bash-prompt> PROBLEM="channel"
```

and then use it in the input file as

```
COORDINATE {
    coordinates = Read( Env("PROBLEM") . ".crd" )
}
```

This is equivalent to

```
COORDINATE {
    coordinates = Read( "channel.crd" )
}
```

There is significant potential for confusing this operator with the string ".". It is important to be very careful with the quotes. A slightly more complex example equivalent to the above is

```
bash-prompt> EXT="crd"
COORDINATE {
    coordinates = Read( Env("PROBLEM") . "." . Env("EXT") )
}
```

File and Data Format/Reading Data From Files

6

AcuSolve input file supports two mechanisms for entering list and array data.

This chapter covers the following:

- ASCII (Text) Files (p. 25)
- Binary Files (p. 27)
- Converting Between ASCII and Binary Files (p. 28)
- Relevance of File Extensions (p. 29)

The The data may be entered directly into the input file, such as in the following example:

```
COORDINATE {
coordinates =  {1, 1.00, 1.25, 1.50;
                2, 1.10, 1.30, 1.55;
                3, 1.15, 1.35, 1.60;
                4, 1.25, 1.40, 1.65;
               }
}
```

Or, the data may be read from an external file.

An external file can be referenced in the input file through the Read command. The path of the file can be absolute or relative to the location of the problem directory. For example, if the file is in the same directory as the input file:

```
COORDINATE {
    coordinates = Read( "channel.crd" )
}
```

If the file is stored inside `MESH.DIR` directory:

```
COORDINATE {
    coordinates = Read( "MESH.DIR/channel.crd" )
}
```

Both of the formats above are an example of relative path location. The full absolute path can also be used.

The data read from files can be any list or array and can have the following data types:

- Integers
- Real
- String

The format of the files on disk are classified into two categories:

- ASCII (text) Files

- Binary Files

ASCII (Text) Files

ASCII files contain data in ASCII code, which is a human readable character representation of text. Files stored in this format can be read and edited via a text editor, such as vi, emacs or Notepad.

The data entries contained in the files are written in the form of rows and columns. The entries in a row are separated by a white space, which can be one or more character spaces or tabs. The rows are separated by use of a line break.

The number of columns in the file depend on the functionality of the data file. For example, a connectivity file, used in the `ELEMENT_SET` command, with tetrahedral elements looks like:

```
0 2774 2769 2773 2772
1 2771 2770 2768 2767
2 2778 2770 2774 2771
3 2777 2776 2767 2775
4 2786 2781 2785 2784
```

The first column represents the element ID and the following columns represent the node ID associated with that element.

This file is referenced in the input file as:

```
ELEMENT_SET( "Fluid_Vol" ) {
        elements = Read( "MESH.DIR/Fluid_Vol.tet4.cnn" )
        shape = four_node_tet
        ......
        ......}
```

As another example, the element ID file, used in the `ELEMENT_OUTPUT` command, looks like:

```
0
1
2
3
4
```

This file is referenced in the input file as:

```
ELEMENT_OUTPUT( "Fluid_Vol" ) {
        elements = Read( "MESH.DIR/Fluid_Vol.tet4.eid" )
        element_set = "interior tet4"
        .......
        .......}
```

All of the data files need to have a carriage return at the end of the file to indicate the end of the line. This is automatically added in the files created by HyperMesh CFD or SimLab. However, you need to make sure it is present if any changes are made to the data files via a text editor.

Commonly used text editors for Linux based systems, such as vi and emacs, add the carriage return automatically when the file is saved, but for text editors used in Windows systems, such as Notepad and Wordpad, you may need to add this when making changes to the data file.

If a file does not have carriage return after the last entry, it is not completely read by AcuPrep and can lead to errors in the simulation.

When AcuPrep reads an ASCII file that is referenced by an input command, the number of columns required for the specified parameter is known. As such, only the appropriate number of columns of data is retained. If a connectivity file is read with element type specified as four_node_tet, AcuPrep expects five columns of data. If the files have fewer columns, AcuPrep will issue an error while reading the input file. If the ASCII file has additional columns, they are ignored.

AcuPrep can also read an array of strings from an external file. For example, the command `SIMPLE_BOUNDARY_CONDITION` can read surface sets from a file, which has the surfaces listed as strings.

The data file looks like:

```
"Wall tri3 Fluid pyramid5"
"Wall tri3 Fluid wedge6"
"Wall quad4 Fluid wedge6"
"Wall quad4 Fluid brick8"
"Wall tri3 Fluid tet4"
```

This file is referenced in the input file as:

```
SIMPLE_BOUNDARY_CONDITION( "Wall" ) {
        surface_sets = Read( "SBC_Wall.srfst" )
        type = wall
        ...... ..
        ...... ..}
```

Each line is assumed to be a complete string, so it is not necessary to provide quotation marks around the strings. However, AcuPrep supports reading of the strings with surrounding quotation marks as well.

Binary Files

Binary files contain data in binary code which contain both intelligible characters (numbers and text) as well as special characters such as line feed or system bell. AcuSolve utilities store these files with a `.B` extension.

Binary files are smaller in size compared to their ASCII counterparts. They have a compact storage format and can be read and written much quicker. Since binary files contain special characters as well as text, it is not possible to edit them directly. All of the programs in the AcuSolve suite support reading of binary format.

Converting Between ASCII and Binary Files

The utility program, acuFmtArray, can be used to convert binary files to ASCII or vice-versa.

For example,

```
acuFmtArray -crd Channel_acu.crd
```

can be used to convert the ASCII file, `Channel_acu.crd`, into binary format. During conversion, the following message will display in the terminal:

```
acuFmtArray: Coordinates:            Channel_acu.crd -> Channel_acu.crd.B
```

Once the file is converted to binary, it can no longer be edited by text editors. If you want to make changes to the file, it will need to be converted back to ASCII, then saved and again converted to binary.

Note: AcuFmtArray infers the format of the file based on the extension. If a file name ends in `.B`, it is assumed to be binary. Otherwise, it is assumed to be ASCII.

The application automatically removes existing files after converting, unless the option "delete_files"=off is used.

Relevance of File Extensions

AcuSolve utilities store the different data files with their own unique extension.

These extensions do not individually specify a file format, but are used for your convenience. In other words, file extensions can be changed arbitrarily without impacting AcuPrep's ability to read and operate on the file.

The only extension that signifies a specific file format is the `.B` extension, which indicates a binary file. This extension can be added to any file name, and will indicate to AcuPrep that it will be reading a binary file.

The table below shows some of the commonly used file extensions and what they are typically used for. Different pre-processors may use a variation to the conventions shown below.

Extension	Meaning
`.crd`	Coordinate File
`.cnn`	Connectivity File
`.eid`	Element ID File
`.srf`	Surface File
`.nbc`	Nodal Boundary Condition File
`.pbc`	Periodic Boundary Condition File
`.ebc`	Element Boundary Condition File
`.arm`	AcuSolve Readable Mesh File
`.nic`	Nodal Initial Condition File

As previously mentioned, the binary version of the files listed above have an additional `.B` extension appended to the corresponding extensions displayed in the above table.

Alternative Command Organization

7

In the rest of this manual the commands are grouped into general categories and placed into chapters. In order to make a particular command functionality easier to find, an alternative categorization is presented. Each command may appear in more than one category.

Nodal Data

COORDINATE

NODAL_BOUNDARY_CONDITION

RESTART_OUTPUT

NODAL_OUTPUT

NODAL_RESIDUAL_OUTPUT

Continuum Element Data

ELEMENT_SET

FAN_COMPONENT

HEAT_EXCHANGER_COMPONENT

REFERENCE_FRAME

ELEMENT_OUTPUT

CAA_OUTPUT

CAA_ELEMENT_OUTPUT

FWH_OUTPUT

FWH_SURFACE_OUTPUT

Surface Data

FAN_COMPONENT

HEAT_EXCHANGER_COMPONENT

ELEMENT_BOUNDARY_CONDITION

SIMPLE_BOUNDARY_CONDITION

INTERFACE_SURFACE

GUIDE_SURFACE

EXTERNAL_CODE_SURFACE

SURFACE_INTEGRATED_CONDITION

FREE_SURFACE

TURBULENCE_WALL

PARTICLE_SURFACE

RADIATION_SURFACE

SOLAR_RADIATION_SURFACE

CAA_SURFACE_OUTPUT

FWH_SURFACE_OUTPUT

Boundary Conditions

NODAL_BOUNDARY_CONDITION

PERIODIC_BOUNDARY_CONDITION

ELEMENT_SET

ELEMENT_BOUNDARY_CONDITION

SIMPLE_BOUNDARY_CONDITION

SURFACE_INTEGRATED_CONDITION

FREE_SURFACE

RADIATION_SURFACE

SOLAR_RADIATION_SURFACE

Output

RESTART_OUTPUT

NODAL_OUTPUT

NODAL_RESIDUAL_OUTPUT

TIME_AVERAGE_OUTPUT

TIME_HISTORY_OUTPUT

SURFACE_OUTPUT

ELEMENT_OUTPUT

FAN_COMPONENT

HEAT_EXCHANGER_COMPONENT

RADIATION_SURFACE

DERIVED_QUANTITY_OUTPUT

RUNNING_AVERAGE_OUTPUT

CAA_ELEMENT_OUTPUT

CAA_OUTPUT

CAA_SURFACE_OUTPUT

ERROR_ESTIMATOR_OUTPUT

EXTERNAL_OUTPUT

FWH_OUTPUT

FWH_SURFACE_OUTPUT

Restart

RESTART_OUTPUT

RESTART

Radiation

EQUATION

RADIATION

EMISSIVITY_MODEL

RADIATION_SURFACE

SOLAR_RADIATION

SOLAR_RADIATION_MODEL

SOLAR_RADIATION_SURFACE

Turbulence Modeling

EQUATION

TIME_SEQUENCE

STAGGER

TURBULENCE_WALL

NODAL_BOUNDARY_CONDITION

SIMPLE_BOUNDARY_CONDITION

Component Models

FAN_COMPONENT

HEAT_EXCHANGER_COMPONENT

Material Models

DENSITY_MODEL

SPECIFIC_HEAT_MODEL

VISCOSITY_MODEL

CONDUCTIVITY_MODEL

DIFFUSIVITY_MODEL

POROSITY_MODEL

EMISSIVITY_MODEL

SOLAR_RADIATION_MODEL

SURFACE_TENSION_MODEL

CONTACT_ANGLE_MODEL

VISCOELASTIC_MODEL

Fluid-Solid Interaction

MESH_MOTION

FLEXIBLE_BODY

EXTERNAL_CODE

EXTERNAL_CODE_SURFACE

Body Forces

GRAVITY

REFERENCE_FRAME

MASS_HEAT_SOURCE

VOLUME_HEAT_SOURCE

MASS_SPECIES_SOURCE

VOLUME_SPECIES_SOURCE

BODY_FORCE

LINE_SOURCE

Solution Strategy

AUTO_SOLUTION_STRATEGY

ANALYSIS

TIME_SEQUENCE

STAGGER

TIME_ INCREMENT

CONVERGENCE_CHECK_PARAMETERS

TIME_INTEGRATION

LINEAR_SOLVER_PARAMETERS

PARTICLE_TRACE

Transient/Steady Analysis

ANALYSIS

TIME_SEQUENCE

STAGGER

TIME_ INCREMENT

TIME_INTEGRATION

Auxiliary Functions

USER_GLOBAL_DATA

MULTIPLIER_FUNCTION

MESH_MOTION

GUIDE_SURFACE

ASSIGN

Global Commands

8

This section contains the commands that define the global description of a problem or provide data that can be used globally.

This chapter covers the following:

- ANALYSIS (p. 36)
- CONSTRAINT (p. 39)
- DESIGN VARIABLE (p. 41)
- DESIGN VARIABLES FIELD (p. 42)
- EQUATION (p. 44)
- EXTERNAL_CODE (p. 61)
- FIELD_BOUNDARY_CONDITION (p. 65)
- FLEXIBLE_BODY (p. 66)
- IMMERSED_BOUNDARY_SURFACE (p. 78)
- MESH_MOTION (p. 80)
- MULTIPLIER_FUNCTION (p. 97)
- NODAL SHAPES (p. 107)
- OBJECTIVE (p. 109)
- OPTIMIZATION (p. 110)
- PARTICLE_TRACE (p. 113)
- RADIATION (p. 114)
- REFERENCE_FRAME (p. 116)
- RESPONSE VARIABLE (p. 119)
- SOLAR_RADIATION (p. 126)
- THERMAL_SHELL (p. 131)
- TURBULENCE_MODEL_PARAMETERS (p. 133)
- USER_GLOBAL_DATA (p. 142)
- WAVE_BOUNDARY_CONDITION (p. 143)
- WAVE_DAMPING_SOURCE (p. 145)

These commands are optional. If any of these commands is not given, its default values are used.

ANALYSIS

Specifies global analysis data.

Type

AcuSolve Command

Syntax

ANALYSIS {parameters}

Qualifier

This command has no qualifier.

Parameters

title (string) [=""]
Title of the problem. The title, not to exceed 1023 characters, is an optional description of the problem.

sub-title (string) [=""]
Subtitle of the problem. The subtitle is an optional parameter which may be used to further describe the problem. This parameter may not exceed 1023 characters.

input_version or version (string)[="unknown"]
Version of AcuSolve for which this input file was created.

type or mode (enumerated) [=static]
Type of the analysis.

static or steady	Static (steady state) solution.
dynamic or transient	Dynamic (transient) solution.

Description

This command specifies the global analysis information. The *title* and *sub_title* parameters allow you to provide a description of the problem. These parameters have no impact on the solution of the problem.

The *type* parameter specifies the type of analysis being performed, namely static (steady state) or dynamic (transient). In either case, a time marching scheme is used to solve the problem. In the case of static type, the inertia (mass) terms of the conservation equations are only included in the Galerkin part of the finite element weighted residual formulation. This inclusion adds stability to the nonlinear iterations. Other parts of the finite element formulation, such as the least-squares operator, do not include the inertia terms. This exclusion accelerates nonlinear convergence to steady state at the expense of time accuracy. For dynamic type, the inertia terms are included in all the operators of the finite element formulation, in order to preserve time accuracy.

The *type* parameter alone does not specify the time accuracy of a problem. You also need to set the time integration option via the TIME_INTEGRATION command. In addition, you may need to properly set

the number of nonlinear iterations and the nonlinear convergence tolerances via the `TIME_SEQUENCE` and `STAGGER` commands, and the time increment options via the `TIME_INCREMENT` command.

For example, a steady-state problem may be set up with these commands:

```
ANALYSIS {
  title = "Test steady-state solution"
  sub_title = "8-node brick"
  type = static
}
TIME_SEQUENCE {
  convergence_tolerance = 1.e-3
  min_stagger_iterations = 1
  max_stagger_iterations = 1
  ...
}
TIME_INCREMENT {
  initial_time_increment = 1.e+10
  auto_time_increment = off
  local_time_increment = off
}
TIME_INTEGRATION {
  time_integration_order = first
}
```

A transient problem may be set up with:

```
ANALYSIS {
  title = "Test transient solution"
  sub_title = "8-node brick"
  type = dynamic
}
TIME_SEQUENCE {
  convergence_tolerance = 1.e-3
  min_stagger_iterations = 2
  max_stagger_iterations = 3
  ...
}
TIME_INCREMENT {
  initial_time_increment = 1.
  min_time_increment = 0.01
  max_time_increment = 10.
  auto_time_increment = on
  local_time_increment = off
  ...
}
TIME_INTEGRATION {
  time_integration_order = second
  high_frequency_damping_factor = 1.0
}
```

Possible future command incompatibilities between releases can be detected if *input_version* is specified, for example,

```
ANALYSIS {
 title = "Test steady-state solution"
 sub_title = "8-node brick"
 type = static
 input_version = "1.7"
}
```

where *input_version* is the version of AcuSolve for which the input file was written. If this is earlier than the currently-used version, then AcuSolve will check for any incompatibilities in command usage between these two versions. If any are found, appropriate warning or error messages will be issued. Although optional, it is highly recommended that this parameter be included in all input files. The current version can be obtained from:

```
acuPrep -help | grep Release
```

CONSTRAINT

Specifies the constraints based on responses.

Type

AcuSolve Command

Syntax

CONSTRAINT {parameters}

Qualifier

This command has no qualifier.

Parameters

type (enumerated) [=less_than]

Constraints are specified by relating the response variables with the type. Requires RESPONSE_VARIABLE.

less_than

greater_than

equal_to

response_variable (string) [=""]

User defined name of the response variable that is used to compute the objective and constraint functions.

bound (real) [=0.0]

Bounds are set to control the motion of surface/body in the optimization region. When *type* = equal_to the constraint is satisfied within the given tolerance.

tolerance (real) [=0.0]

Absolute tolerance within which the mass balance constraint of *type* = equal_to is satisfied.

Description

The CONSTRAINT command defines the constraint functions used in optimization. One or more constraints can be created to control the motion of the model during the optimization process. To create a constraint, first a response variable needs to be created and the type and bound needs to be defined. This can be a time history output, surface integrated output, and so on.

In topology optimization with mass balancing, "bound" is the target mass flow rate at the specific exit and "tolerance" is the acceptable absolute deviation from this mass flow rate. The constraint is defined as:

bound - tolerance < mass < bound + tolerance, or |mass - bound| < tolerance.

Here, mass is the mass flow rate which is computed in the *response_variable* corresponding to the specific constraint.

```
CONSTRAINT( "subject to pressure >= bound” ) {
    type                                = larger_than
    response_variable                   = "Inflow Pressure"
    bound                               = 1000
}
 RESPONSE_VARIABLE( "Inflow Pressure” ) {
    type                                = surface_output
    surface_output                      = "Inlet"
    surface_output_variable  = pressure
}
```

```
CONSTRAINT( "flow balance 1” ) {
    type                                        = equal_to
    response_variable                           = "flow_balance_1"
    bound                                       = 0.01
    tolerance                                   = 0.0001
}
RESPONSE_VARIABLE( "flow_balance_1" ) {
    type                                        = topology_variable
    topology_variable                           = surface_mass_flow
    mass_flow_surface        = "Outlet_1"
}
```

DESIGN VARIABLE

Specifies design optimization variables.

Type

AcuSolve Command

Syntax

DESIGN_VARIABLE {parameters}

Qualifier

This command has no qualifier.

Parameters

initial_value *(real) [=0.0]*
: Initial value assigned to the design variable.

lower_bound *(real) [=-1.0]*
: Lower bound of design variable.

upper_bound *(real) [=1.0]*
: Upper bound of design variable.

max_update_factor *(real) [=0.05]*
: Max relative step change allowed for the design variable to change between iterations. The max update factor is relative to the *upper_bound* and the *lower_bound* difference.

supervised_training *(array) [={1.0}]*
: Manual supervised training is used as input data to create a so called response surface. The corresponding response function values are computed by AcuSolve for the given input training data. The response surface is constructed using a Gaussian Process, which essentially is a function approximating the response values as a function of the design variables.

Description

The shape of the deformed geometry is controlled by design variable commands. For each variable a specified shape is defined, for example using mesh displacements defined in a nodal boundary condition command. Design variables may also control parameters by using multiplier functions of type set to design_variable.

```
DESIGN_VARIABLE("DV1") {
 initial_value            = 0.0
 lower_bound              = -1.0
 upper_bound              = 1.0
 max_update_factor        = 0.05
 supervised_training      = {0.1,0.2,0.5,…. }
}
```

DESIGN VARIABLES FIELD

Specifies design optimization variables field for topology optimization.

Type

AcuSolve Command

Syntax

DESIGN_VARIABLES_FIELD {parameters}

Qualifier

This command has no qualifier.

Parameters

default_value *(real) [=1.0]*

Initial constant value assigned to the design variables field.

max_update_factor *(real) [=0.05]*

Max relative step change allowed for the design variables to change between iterations.

design_topology_interpolation *(enumerated) [=simp]*

The interpolation used at the interface of the two different material regions. The two different materials are specified using the ELEMENT_SET command. For topology optimization of flow, the *material_model* parameter is equal to a fluid material without a porosity model, and the parameter *design_material_model* is equal to a fluid material model with a specified porosity model and darcy coefficient. With the material model of type fluid and the design material model of type design, representing the solid, the interface is the region between the fluid and the solid.

min_filter_radius *(real) [=0.0]*

By setting the *min_filter_radius* parameter to a dimensional value, the topology optimizer prevents the creation of narrow topological passages whose width dimension falls below the specified *min_filter_radius* parameter value. The radius must be at least the size of the element mesh size. The minimum resolved design features correspond to two times the given radius. The minimum filter radius used by laplacian topology defined in the EQUATION command.

filter_wall_type *(enumerated) [=free]*

Type of the design topology level for walls.

free	Automatically computes the design value for all wall surfaces using the solution algorithm.
constant	Assign a constant value (either 1 or 0) on the wall, determined by the types specified in the *topology_design_wall* parameter in the SIMPLE_BOUNDARY_CONDITION. When *topology_design_wall* = on, the design value at the wall is set to 1. Conversely, when *topology_design_wall* = off, it is set to 0. This option is the preferred method to solve topology problems

with obstructions. Requires *topology_design_wall* in the SIMPLE_BOUNDARY_CONDITION command.

project_design_topology (enumerated) [=step]
Type of projection used with laplacian_topology filtering defined in the EQUATION command.

step	Stepwise increase of the projection, every 20th case.
continuous	Continuous increase of the projection; every case.

project_design_topology_factor_max (real) [=32]
The maximum value of the projection used with laplacian_topology defined in the EQUATION command. A higher value gives a sharper interface between the fluid and solid regions.

> **Note:** A too high value may give convergence problems.

Description

The shape of the geometry is controlled by the design variables field. For topology flow optimization, the field value of zero represents the fluid and the field value of one represents the solid. The default initial value is equal to one, that is, the optimization starts from solid domain everywhere by default. To start from fluid domain *default_value* should be set to zero. Modifying *max_update_factor* will act as a relaxation factor to the convergence of the optimization cycles. There is a single option for the design topology interpolation, and a single design variables field is supported in this version.

```
DESIGN_VARIABLES_FIELD("design topology") {
default_value = 1.0
max_update_factor = 0.05
design_topology_interpolation = simp
}
```

If the laplacian topology filtering is used, the command can be given by:

```
DESIGN_VARIABLES_FIELD( "topology" ) {
    default_value = 0.0
    project_design_topology = step
    filter_radius = 0.02
    project_design_topology_factor_max = 32
    filter_wall_type = constant
}
```

Here, *design_topology_filter* = laplacian_topology must be set in the EQUATION command. Further, to specify a "solid" wall condition for the design topology, when *filter_wall_type* = constant, use:

```
SIMPLE_BOUNDARY_CONDITION(  "Wall with solid design topology boundary condition" ) {
    ...
    topology_design_wall = on
}
```

EQUATION

Specifies equation systems present in the problem.

Type

AcuSolve Command

Syntax

EQUATION {parameters}

Qualifier

This command has no qualifier.

Parameters

flow (enumerated) [=navier_stokes]

Flow equation type.

navier_stokes	Navier-Stokes equations.
stokes	Time dependent Stokes equations.
compressible_navier_stokes (cns)	Compressible Navier-Stokes equations.
compressible_euler	Compressible Euler equations.

absolute_pressure_offset (real) >=0 [=0]

Offset to convert to absolute pressure units. Used with *ideal_gas* and *isentropic* types in DENSITY_MODEL.

temperature or *temp (enumerated) [=none]*

Energy equation type.

none	No energy equation solved.
advective_diffusive	Energy equation solved.

absolute_temperature_offset (real) >=0 [=0]

Offset to convert to absolute temperature units. When *temperature* = advective_diffusive is used with any of the following conditions: *ideal_gas* type in DENSITY_MODEL, any radiation model, or *radiation_heat_flux* variable in ELEMENT_BOUNDARY_CONDITION.

radiation or *rad (enumerated) [=none]*

Radiation type.

none	No radiation solved.
enclosure	The enclosure radiation model.

p1_model The p1 radiation model. Used with *temperature* = advective_diffusive.

> **Note:** Operating temperature should be absolute temperature.

discrete_ordinate The Discrete Ordinates or DO model. Used with *temperature* = advective_diffusive.

> **Note:** The operating temperature should be absolute temperature.

radiation_quadrature [=S4]
Specifies the order of the Sn angular quadrature set used with the discrete ordinates method. Essentially these represent the directions, or ordinates, along which the radiative transfer equation is solved. The available quadrature orders are: S2, S4, S6, S8 and S10, representing a discrete number of directions: 8, 24, 48, 80 and 120 directions, respectively. The default value is S4, which gives a good balance between computational efficiency and accuracy.

radiation_solve_frequency (int) [=1]
Defines the frequency of the radiation solve relative to other transport equations. For example, the default value of one means the radiative transport is solved every timestep. *radiation_solve_frequency* is only valid if *radiation* = discrete_ordinate.

species_transport or *spec (enumerated) [=none]*
Multi-species transport equation type.

none No species field.

advective_diffusive or ad Advective diffusive equation for multiple species transport. Requires *num_species*.

humid_air_model (boolean) [=off]
Flag specifying whether to use the *humid_air_model*. When set to on, the field is assigned to the humid air. To activate the humid air model, *temperature* = advective_diffusive should be set first. Requires *multi_field*. When the humid air model is switched on, *humidity_film_thickness* calculation will be performed in transient simulation only. For steady-state simulation, it will be ignored.

edem_coupling (boolean) [=off]
Flag specifying whether to conduct AcuSolve-EDEM one-way or two-way coupling. When set to on, *multi_field* should be set to either algebraic_eulerian for one-way coupling or should be set to eulerian_eulerian for two-way coupling.

multi_field (enumerated) [=none]
Type of multi-field model to use.

none or off No multi-field model is used.

levelset	Use the level set method to capture the interface between immiscible fluids. Requires *fields*.
levelset_bfecc	Use the level set method with the Back and Forth Error Compensation and Correction (BFECC) advection scheme. Requires *fields*.
algebraic_eulerian	Use the Algebraic Eulerian method to model multiple dispersed fields with one carrier field. Requires *fields*. This option is also required for AcuSolve-EDEM one-way transient coupling.
eulerian _eulerian	Use the Eulerian Eulerian method to model multiple dispersed fields with one carrier field. Requires *fields*. This option is also required for AcuSolve-EDEM two-way transient coupling.
advective_diffusive	Use the advective diffusive method to model the transport of gas mixture. Requires *fields*.

fields (list) [no default]
List of fields to solve in the case of multi-field analyses. When using *multi_field*=levelset, solutions are limited to two immiscible fields. An unlimited number of dispersed fields can be modeled with one carrier field using *multi_field*=algebraic_eulerian. When using *humid_air_model* and *multi_field*=advective_diffusive, together, or EDEM coupling (one-way and two-way), the number of fields are limited to two.

num_species (integer) >0 <=9 [=1]
Number of species present in the problem. Used with advective_diffusive species transport type.

species_names (array) [no default]
Array of user-defined species names to associate a name string with each species via a *species_names*. Used with *num_species*.

turbulence or turb (enumerated) [=none]
Turbulence model.

none	Laminar flow.
spalart_allmaras or spalart	Spalart-Allmaras one-equation RANS model.
shear_stress_transport or sst	Shear Stress Transport 2-equation RANS model.
k_epsilon or keps	K-epsilon 2-equation RANS model.
realizable_k_epsilon or realkeps	Realizable k-epsilon 2-equation RANS model.
rng_k_epsilon or rngkeps	RNG k-epsilon 2-equation RANS model.
k_omega	K-omega two-equation RANS model.
bsl_k_omega	BSL k-omega two-equation RANS model.

detached_eddy_simulation or des Spalart-Allmaras based detached eddy simulation model.

shear_stress_transport_detached_eddy_simulation or sst_des Shear stress transport based detached eddy simulation model.

large_eddy_simulation or les Classical (Smagorinsky) LES model.

dynamic_model or dynamic Dynamic subgrid LES model.

turbulence_transition_model or *trans* *(enumerated) [=none]*
Turbulence transition model. Valid only when *turbulence*=spalart_allmaras, shear_stress_transport, detached_eddy_simulation, or shear_stress_transport_detached_eddy_simulation.

none Do not include the impact of transition physics into turbulent simulations.

intermittency_momentum_thickness_reynolds_number or gamma_re_theta γ-$Re_{\theta t}$ two-equation transition model.

intermittency or gamma γ one-equation transition model.

viscoelastic or *vest* *(enumerated) [=none]*
Viscoelastic material model type.

none No viscoelastic material model equation.

upper_convected_maxwell or ucm Upper convected Maxwell model used to incorporate viscoelastic material effects.

upper_convected_maxwell_log_model or ucm_log Log formulation of the upper convected Maxwell model used to incorporate viscoelastic material effects.

mesh
Mesh type.

eulerian or fixed Fixed mesh.

arbitrary_lagrangian_eulerian or ale Arbitrary mesh movement.

ale_ogden Enhanced arbitrary mesh movement useful for extreme mesh deformation cases.

specified or enforced Specified mesh movement.

optimization (boolean) [=off]
Flag to turn on optimization. This requires an OPTIMIZATION command and either one or more DESIGN_VARIABLE commands, or a single DESIGN_VARIABLES_FIELD command for topology optimization. If the design variables control change in geometry, mesh has to be set to specified or arbitrary_lagrangian_eulerian.

topology_flow or *topoflow* *(enumerated) [= none]*
Topology flow equation type.

none	No adjoint linearized Navier-Stokes equations are solved.
adjoint_topology_flow	Adjoint linearized Navier-Stokes equations.

design_topology_filter or *topofilt* *(enumerated) [= none]*
Design topology filtering method. Valid only when *toplogy_flow* = adjoint_topology_flow.

none	No design topology filtering.
laplacian_topology	Design topology filtering. Requires *min_filter_radius* in the DESIGN_VARIABLE_FIELD command.

external_code *(boolean) [=off]*
Flag specifying whether an external solid/structural code is coupled with AcuSolve to solve a Direct Coupling Fluid Structure Interaction (DC-FSI) problem.

particle_trace *[=off]*
Flag specifying whether to permit coupling with AcuTrace.

running_average *[=off]*
Flag specifying whether to create running average solution fields. If this option is set to on, the effectiveness of the non-reflecting boundary condition will be improved when *flow* = navier_stokes and *non_reflecting_bc_running_average_field* (nrbcraf) = true.

running_average_steps *(integer) >=1 [=100]*
Maximum number of time steps to average solution fields. Used with *running_average*=on.

gravitational_acceleration *(array) [={0,0,0}]*
The constant value of the gravity force. This array must have exactly three components in the global xyz coordinate system.

> **Note:** *gravitational_acceleration* and *gravity* under GRAVITY, as a part of BODY_FORCE, are cumulative.

porous_media_velocity_type *(enumerated) [=superficial]*
Porous media velocity type. Requires the porosity model. Type of the porosity model is set to constant.

superficial	Superficial velocity porous approach.
physical	Physical velocity porous approach.

When *porous_media_velocity_type*=superficial, velocity is calculated based on volumetric flow rate and the cross section. When *porous_media_velocity_type*=physical, a more accurate representation of velocity inside the porous media can be obtained by solving the continuity and the momentum equation using the intrinsic/physical averaging method.

electric_potential *(enumerated) [=none]*
Electric potential equation type.

none	No charge field.
Joule_heating	Solution of charge conservation equation coupled to the energy equation.
battery_msmd	Solution of charge conservation equation coupled to the energy with batteries modeled using a multi-scale multi-dimensional approach.
battery_joule_heating	Solution of charge conservation equation coupled to the energy with batteries modeled using an equivalent circuit model.

Enables the solution of the charge conservation equation on the assumption of electrostatics. If *electric_potential* = joule_heating the charge conservation equation is coupled to the energy equation via a source term based on Joule's first law.

Battery modeling is enabled using battery_msmd or battery_joule_heating. The former uses a multi-scale approach where there is a cell scale and a sub-domain electrochemical scale. In this model two potential fields are resolved at the cell scale, which represent the potentials of the negative and positive current collectors. The sub-domain model, currently only supporting ECM models, represents the voltage-current response of the battery derived from the solution of an ODE system. Inter-domain coupling between the sub-domain and cell scale is achieved via averaging source terms, to eliminate any spatial dependence, in the dual electric potential and energy equations. Cell scale to sub-domain coupling is achieved using spatially resolved variables directly in the sub-domain equations. The battery_joule_heating model also uses ECM models but applies them as source terms in the energy equation and boundary conditions based on current and terminal voltage for the electric potential in the connected components of the batteries, for example, terminals/tabs and busbars. Both battery approaches use joule heating in the terminals/tabs and busbars.

Description

This command specifies the equations present in the problem. For example, to specify a flow problem with an energy equation, you need:

```
EQUATION {
    flow = navier_stokes
    temperature = advective_diffusive
}
```

The basic continuity, momentum, energy, and species equations are:

$$\begin{aligned} \frac{\partial \rho}{\partial t} + \nabla \bullet \rho u &= 0 \\ \rho \frac{\partial u}{\partial t} + \rho u \bullet \nabla u + \nabla p &= \nabla \bullet \tau + \rho b \\ \rho \frac{\partial h}{\partial t} + \rho u \bullet \nabla h &= \nabla \bullet q + \rho s \\ \rho \frac{\partial \varphi_i}{\partial t} + \rho u \bullet \nabla \varphi_i &= \nabla \bullet \psi_i + \rho \sigma_i \end{aligned} \qquad (1)$$

where ρ is the density, given by the DENSITY_MODEL command; u is the velocity vector, p is the pressure, $\tau = [\tau_{ij}]$ is the viscous stress tensor, given by VISCOSITY_MODEL; b is the momentum source vector, given by GRAVITY and other commands; h is the enthalpy, given by SPECIFIC_HEAT_MODEL; q is the heat flux vector, given by CONDUCTIVITY_MODEL; s is the heat source per unit mass, given by MASS_HEAT_SOURCE; $\varphi_i, i = 1, \ldots,$ num_species, are the scalar species; Ψ_i is the diffusion flux vector for species φ_i, given by DIFFUSIVITY_MODEL; and σ_i is the species source per unit mass for species φ_i, given by MASS_SPECIES_SOURCE. Other terms are added by several commands; the full equations are given in the descriptions of these commands.

The only change for stokes flow is the removal of the $\rho u \cdot \nabla$ (convective) terms. These equations may be also modified by various body forces, references frames, material models, and so on. See the corresponding commands for the specific modifications.

To add enclosure radiation to the energy equation, where the temperature is in degrees Celsius, the above command is modified as follows:

```
EQUATION {
 flow = navier_stokes
 temperature = advective_diffusive
 radiation = enclosure
 absolute_temperature_offset = 273.14
}
```

To add two scalar transport equations to the system:

```
EQUATION {
 flow = navier_stokes
 temperature = advective_diffusive
 radiation = enclosure
 absolute_temperature_offset = 273.14
 species_transport = advective_diffusive
 num_species = 2
 species_names = {"honey", "dust"}
}
```

To add the Spalart-Allmaras turbulence model:

```
EQUATION {
 flow = navier_stokes
 temperature = advective_diffusive
 radiation = enclosure
 absolute_temperature_offset = 273.14
 species_transport = advective_diffusive
 num_species = 2
 species_names = {"honey", "dust"}
 turbulence = spalart_allmaras
}
```

To add the ability to move the mesh arbitrarily and simulate a viscoelastic material:

```
EQUATION {
 flow = navier_stokes
 temperature = advective_diffusive
 radiation = enclosure
 absolute_temperature_offset = 273.14
 species_transport = advective_diffusive
```

```
 num_species = 2
 species_names = {“honey”, “dust”}
 turbulence = spalart_allmaras
 mesh = arbitrary_lagrangian_eulerian
 viscoelastic = upper_convected_maxwell
}
```

Finally, to add a solid/structural equation that will be solved using an external code as part of a Direct Coupling Fluid Structure Interaction (DC-FSI) problem:

```
EQUATION {
    flow = navier_stokes
    temperature = advective_diffusive
    radiation = enclosure
    absolute_temperature_offset = 273.14
    species_transport = advective_diffusive
    num_species = 2
    species_names = {“honey”, “dust”}
    turbulence = spalart_allmaras
    mesh = arbitrary_lagrangian_eulerian
    viscoelastic = upper_convected_maxwell
    external_code = on
}
```

An arbitrary_lagrangian_eulerian mesh is typically active when an external code is used. See the EXTERNAL_CODE command for more details on setting up DC-FSI problems.

The EQUATION command specifies the existence and types of the equation systems and solution fields present in the problem. The TIME_SEQUENCE and STAGGER commands must then be used to request the solution for the specified equations. For example, to solve for a turbulent flow problem using the Spalart-Allmaras turbulence mode, you must have:

```
EQUATION {
    flow = navier_stokes
    turbulence = spalart_allmaras
}
TIME_SEQUENCE {
    staggers = { "flow", "turb" }
    ...
}
    STAGGER( "flow" ) {
    equation = flow
    ...
}
    STAGGER( "turb" ) {
    equation = turbulence
    ...
}
```

As another example, to solve a coupled particle-flow problem with AcuTrace, you must have:

```
EQUATION {
    flow = navier_stokes
    particle = on
}
TIME_SEQUENCE {
    staggers = { "flow", "particle" }
    ...
}
```

```
STAGGER( "flow" ) {
    equation = flow
    ...
}
STAGGER( "particle" ) {
    equation = particle
    ...
}
```

To solve for a flow field that has multiple immiscible fluid phases in contact with each other, it is necessary to employ a multi-field modeling approach. When setting *multifield* = levelset, AcuSolve tracks the interface between the two immiscible fluids using the requested interface tracking scheme. This type of technology is useful for tracking the interface between fluids for applications such as fuel sloshing in a fuel tank, waves on the surface of the ocean, or a liquid being poured into a container. Two options are available for immiscible multiphase simulations, levelset and levelset_bfecc. Both level set approaches rely on two additional staggers to track the interface. In the standard levelset method, the first stagger governs the transport of the interface, and the second stagger controls the sharpness of the interface. The levelset_bfecc method includes extra stagger iterations to reduce the amount of diffusion in the solution field. The levelset_bfecc method captures multiphase interfaces more sharply than the standard levelset method. Because more calculations are performed, more computing time is needed when compared to the standard levelset method.

In typical applications, the flow field is iterated in conjunction with the advection of the interface, and the sharpening is done as a final step in the time sequence within each time step. For example, the following settings represent a multi-field simulation containing air and water with the level set method used to track the interface:

```
EQUATION {
    flow = navier_stokes
    multi_field = levelset
    fields = {"water","air";}
}
TIME_SEQUENCE {
    staggers = { "flow_levelset", "redistancing" }
    ...
}
STAGGER( "flow" ) {
    equation = flow
    ...
}
STAGGER( "levelset" ) {
    equation = levelset
    ...
}
STAGGER( "redistancing" ) {
    equation = levelset_redistancing
    ...
}
STAGGER( "flow_levelset" ) {
    equation = none
    ...
    staggers = { "flow","levelset"}
}
```

To simulate multiple-dispersed fields with one carrier field, it is necessary to set *multifield*=algebraic_eulerian. AcuSolve tracks the dispersed fields (particles, bubbles, droplets) as

Eulerian fields while calculating the momentum exchange between the dispersed fields and the carrier using an algebraic slip model. The carrier field of the problem should be the last field in fields under `EQUATION`. In general, it is recommended that the heavier fluid is specified as the carrier. For the case where the dispersed field and carrier field are separated, the Algebraic Eulerian method tracks the interface as a smooth interface between fluids. Unlike the level set approach, an additional stagger is not needed to track the interface. AcuSolve solves the mixture continuity, mixture momentum equations and field volume fraction equations:

```
EQUATION {
    flow = navier_stokes
    multi_field = algebraic_eulerian
    fields = {"Air", "Water"}
}
TIME_SEQUENCE {
    staggers = {"flow_fields"}
}
STAGGER("flow_fields" ) {
    staggers = {"flow", "field: Air", "field: Water"}
}
STAGGER( "flow" ) {
    equation = flow
}
STAGGER( "field: Air" ) {
    equation = field
    field = "Air"
}
STAGGER( "field: Water" ) {
    equation = field
    field = "Water"
}
```

When using the *`humid_air`* model, AcuSolve tracks the spread of “Water vapour” field by solving the field's mass fractions equation in the form of species transport equation in addition to solving the mixture continuity, momentum equations and energy equations:

```
EQUATION {
    flow = navier_stokes
    temperature = advective_diffusive
    humid_air_model = on
    multi_field = advective_diffusive
    fields = { "Humid_Air", "Air" }
}
TIME_SEQUENCE {
    staggers = { "flow_temp_fields" }
}
STAGGER( "flow_temp_fields" ) {
    staggers = { "flow", “temperature”, "field: Humid_Air", "field: Air",  }
}
STAGGER( "flow" ) {
    equation = flow
}
STAGGER( "temperature" ) {
    equation = flow
}
STAGGER( "field: Humid_Air" ) {
    equation = field
    field = "Air"
}
STAGGER( "field: Air" ) {
```

```
    equation = field
    field = "Air"
}
```

To model incident radiation using the p1 radiation model in a grey, optically thick participating medium, the following command is defined: *radiation* = p1_model. In order to solve the energy equation coupled with radiative transport set *temperature* = advective_diffusive. This provides the influence of thermal radiation, through heat sources and sinks, on the temperature field.

```
EQUATION {
    flow = navier_stokes
    radiation = p1_model
    temperature = advective_diffusive
}
RADIATION {
    view_factor_type                    = hemicube
    num_hemicube_bins                   = 200
    max_surface_subdivision             = 1
    smoothing_type                      = least_squares
    stefan_boltzman_constant            = 5.67e-008      # W/m2-K4
    num_symmetry_planes                 = 0
    symmetry_center = { 0, 0, 0; } ;
    symmetry_direction_1 = { 1, 0, 0; } ;
    symmetry_direction_2 = { 0, 1, 0; } ;
    symmetry_direction_3 = { 0, 0, 1; } ;
}
```

To model radiative transport in a grey, participating medium. The method handles the full spectrum of optical thicknesses. In order to solve the energy equation coupled with radiative transport set *temperature* = advective_diffusive. This provides the influence of thermal radiation, through heat sources and sinks, on the temperature field.

```
EQUATION {
    radiation = discrete_ordinate
    radiation_quadrature = S4
    temperature = advective_diffusive
}
Alternatively,
EQUATION {
    radiation = DO
    radiation_quadrature = S4
    temperature = advective_diffusive
}
```

When using the k_omega, sst or k_epsilon based turbulence models, two staggers are needed for the turbulence closure; one corresponding to the kinetic energy and one corresponding to the auxiliary equation, eddy frequency for the k-omega and SST based models and dissipation rate for the k-epsilon based models. These equations are solved segregated from each other, necessitating individual staggers for each one. The eddy frequency stagger represents a special case in which the transport quantity used by the stagger deviates from the standard equation for eddy frequency. Instead of solving directly for ω, AcuSolve constructs the transport equation in terms of 1/sqrt(ω). This change of variables does not change the solution produced by the SST and k-omega turbulence models. However, the change of variables does lead to significant improvements in stability. The actual transport quantity (1/sqrt(ω)) is written as *sqrt_eddy_period* in the nodal field outputs. The *eddy_frequency* variable that appears in the nodal outputs is computed based on the solution for *sqrt_eddy_period*.

There are a few special cases when a differential equation is not solved for each setting that is specified in the `EQUATION` command. Therefore, some of the turbulence models and mesh equations require no staggers. For the turbulence models large_eddy_simulation and dynamic_model, the turbulence eddy viscosity is automatically computed at the beginning of every stagger based on an algebraic relation. The eulerian mesh equation does not require a stagger since there is no mesh motion at all. There is mesh motion for the specified mesh equation, but since the motion is fully specified at the beginning of each time step no equation needs to be solved and therefore no stagger needs to be given.

The detached_eddy_simulation turbulence equation is a hybrid model combining the accuracy and economy of the Spalart-Allmaras (SA) model for attached boundary layers with the accuracy of the Smagorinsky LES model for separated eddies. Boundary and initial conditions are the same as for the SA model, but a time accurate transient strategy is used as for the LES models. Typically, a static solution using the Spalart-Allmaras model is used as an initial condition. The *shear_stress_transport_detached_eddy_simulation* model provides another detached eddy simulation option. This model relies on the Shear Stress Transport model (SST) in RANS regions as opposed to Spalart-Allmaras.

The physics of the boundary layer transition process can be included into turbulent simulations by activating one of the two turbulent transition models. These models work in conjunction with the underlying turbulence model (spalart_allmaras, shear_stress_transport, detached_eddy_simulation, and shear_stress_transport_detached_eddy_simulation) to evaluate whether the local boundary layer should be modeled as laminar, transitional, or fully turbulent. Both of the available transition models (intermittency_momentum_thickness_reynolds_number and intermittency) are tuned for external aerodynamic applications. Either model may be applied to internal flows, however, recalibration of the models may be required to establish accuracy for these applications.

When setting *transition*=intermittency_momentum_thickness_reynolds_number, two additional scalar equations are added to the simulation. One equation corresponds to the local intermittency of the flow field, the percentage of time that the flow is turbulent at a given point, and the second equation corresponds to the critical momentum thickness Reynolds number, the local Reynolds Number at which the flow transitions to turbulent. For example, the following options will produce a simulation in which two scalar transport equations are solved for the turbulence variables and an additional two scalar equations to govern the transition of the flow from laminar to turbulent:

```
EQUATION {
    flow = navier_stokes
    turbulence = shear_stress_transport
    transition = intermittency_momentum_thickness_reynolds_number
}
```

The combination of the *shear_stress_transport* and the intermittency_momentum_thickness_reynolds_number options model the most complete set of turbulent transition physics. This combination is capable of predicting natural transition of the boundary layer as well as by-pass transition.

If natural transition is the only mechanism of interest, it is possible to reduce the set of underlying equations from four to two by using the spalart_allmaras turbulence model and the intermittency transition model, as shown below.

```
EQUATION {
    flow = navier_stokes
    turbulence = spalart_allmaras
```

```
    transition = intermittency
}
```

This combination of settings provides accurate results for natural transition of external flows, for example, flow over airfoils and wings, using a reduced set of equations. This approach produces faster run times and reduced memory use in comparison to the use of the two-equation turbulence and transition models.

All of the turbulence and transition models in AcuSolve are solved segregated from the flow. Under these circumstances, there is a stagger created for each equation. However, for some equations, there is not necessarily a one-to-one correspondence between equations and staggers. Some equations can be combined into one stagger. For example, a laminar thermal flow problem with radiation may be specified by:

```
EQUATION {
    flow = navier_stokes
    temperature = advective_diffusive
    radiation = enclosure
}
TIME_SEQUENCE {
    staggers = { "tflow", "radiation" }
    ...
}
STAGGER( "tflow" ) {
    equation = temperature_flow
    ...
}
STAGGER( "radiation" ) {
    equation = radiation
    ...
}
```

Solving the temperature and flow equations simultaneously in one stagger can greatly increase the robustness and performance of some simulations, especially ones dominated by free convection. However, this robustness comes at the cost of storing a significantly larger left-hand side matrix. See the `STAGGER` command for details. Similarly, when simulating viscoelastic materials, all six components of the viscoelastic stress tensor are solved using a single stagger.

The *`radiation`* parameter refers to complex radiation models, such as enclosure radiation, P1 radiation or the DO radiation, which require view factors or other non-local data, and a separate stagger to calculate the coupled radiative heat fluxes. Simple radiation heat flux boundary conditions can be specified through `ELEMENT_BOUNDARY_CONDITION`, with radiation of type none. If possible, this is the preferred method since enclosure radiation can be very expensive. In either case, *`absolute_temperature_offset`* is necessary if temperature is not measured in absolute units. For Celsius, this parameter should be 273.14, for Fahrenheit it should be 459.67. This parameter along with *`absolute_pressure_offset`* may also be needed when using certain density models designed for compressible flow. See the `DENSITY_MODEL` command for details. When using the *`absolute temperature_offset`* and *`absolute_pressure_offset`* parameters, AcuSolve internally adds these constants to all user inputs of temperature and pressure, respectively. This applies primarily to initial conditions and boundary condition values. For the temperature equation, the *`absolute_temperature_offset`* is used to convert the inputs from relative scales such as Celcius or Fahrehheit into absolute scales such as Kelvin or Rankine. For pressure values, the *`absolute_pressure_offset`* is used to convert relative pressures (gauge pressure) into absolute

pressures. For example, when using the ideal gas density model, AcuSolve needs to use the absolute pressure and temperature to calculate the density. In the following example, absolute pressures and temperatures are defined for the initial conditions and boundary conditions:

```
EQUATION {
    flow = navier_stokes
    temperature = advective_diffusive
    absolute_temperature_offset = 0.0
    absolute_pressure_offset = 0.0
 }
DENSITY_MODEL("ideal gas density"){
    type = ideal_gas
}

NODAL_INITIAL_CONDITION("absolute pressure"){
    variable = pressure
    type = constant
    constant_value = 101325.0
}
NODAL_INITIAL_CONDITION("absolute temperature"){
    variable = temperature
    type = constant
    constant_value = 293.15
}
SIMPLE_BOUNDARY_CONDITION("constant temperature inflow"){
    ...
    type = inflow
    inflow_type = velocity
    x_velocity   = 32.0
    y_velocity   = 0.0
    z_velocity   = 0.0
    temperature = 293.15
}
SIMPLE_BOUNDARY_CONDITION("average pressure outflow"){
    ...
    type        = outflow
    pressure    = 101325.0
}
```

To run the same problem with the use of gauge pressures and a relative temperature scale, the following sequence of input commands would be used:

```
EQUATION {
    flow = navier_stokes
    temperature = advective_diffusive
    absolute_pressure_offset = 101325.0
    absolute_temperature_offset = 273.15
}
DENSITY_MODEL("ideal gas density"){
    type = ideal_gas
}

NODAL_INITIAL_CONDITION("gauge pressure"){
    variable = pressure
    type = constant
    constant_value = 0.0
}
NODAL_INITIAL_CONDITION("relative temperature"){
    variable = temperature
    type = constant
    constant_value = 20.0
```

```
}
SIMPLE_BOUNDARY_CONDITION("constant temperature inflow"){
    ...
    type          = inflow
    inflow_type   = velocity
    x_velocity    = 32.0
    y_velocity    = 0.0
    z_velocity    = 0.0
    temperature   = 20.0
}
SIMPLE_BOUNDARY_CONDITION("gauge pressure outflow"){
    ...
    type         = outflow
    pressure     = 0.0
}
```

In the above example, all results will be written in terms of gauge pressure and the relative temperature scale. However, internally, AcuSolve will use the *absolute_pressure_offset* and *absolute_temperature_offset* values to convert to absolute scales when calculating the density based on the ideal gas law. When modeling compressible flow the use of *absolute_pressure_offset* and *absolute_temperature_offset* is not recommended. Instead, you should define boundary and initial conditions in terms of absolute values.

If an equation is defined by the EQUATION command, but no active stagger references that equation, the solution field of that equation is unaltered throughout the analysis. Even boundary conditions are not imposed on that solution field. This feature may be used to provide a solution field for the computation of other fields. For example, you can solve an isothermal steady state flow problem by first specifying a flow-only problem:

```
EQUATION {
    flow = navier_stokes
}
TIME_SEQUENCE {
    staggers = { "flow" }
    ...
}
STAGGER( "flow" ) {
    equation = flow
    ...
}
```

Then restart the problem by adding a temperature equation and solving only for the temperature field:

```
RESTART {
}
EQUATION {
    flow = navier_stokes
    temperature = advective_diffusive
}
TIME_SEQUENCE {
    staggers = { "temp" }
    ...
}
STAGGER( "temp" ) {
    equation = temperature
    ...
}
```

The restarted problem contains both the flow and temperature fields. However, the restarted problem does not solve for the flow field. It is simply carried forward from time step to time step. This flow field is used for the solution of the temperature field.

In the original problem, the temperature equation type could have been set to advective_diffusive. If this had been the case, the initial conditions of the temperature field would have simply been carried forward from time step to time step.

Flow, species, turbulence, and radiation equations apply only to fluid or multifield element sets, while the temperature and mesh equations apply to fluid, multifield, solid and shell element sets. See the medium parameter of the `ELEMENT_SET` command for more details. The solid or structural equations solved by an external code do not apply to any element sets within AcuSolve. The external code uses a completely separate mesh.

In order to run topology optimization you need to set-up `EQUATION` as:

```
EQUATION {
flow = navier_stokes
topology_flow = adjoint_topology_flow
optimization = on
}
```

Turbulence model can also be solved to account for topology optimization of cases with larger Reynolds numbers:

```
EQUATION {
flow = navier_stokes
topology_flow = adjoint_topology_flow
optimization = on
turbulence = spalart_allmaras
}
```

If *running_average* is set to on, then a running average field will be created for every equation field in the problem. A running average field is defined by

$$\overline{f_n} = \frac{1}{N} f_n + \left(1 - \frac{1}{N}\right) \overline{f_{n-1}} \qquad (2)$$

where f_n is a solution field, such as velocity and pressure, at step n; $\overline{f_n}$ is its running average; $N=\min(\text{step}, N_{max})$ is the number of steps used for averaging; and N_{max} is the maximum number of steps used for averaging, given by *running_average_steps*. The creation of running average fields does not itself have any effect on the solution. This command only makes such fields available to the following commands:

- `NODAL_INITIAL_CONDITION`
- `SIMPLE_BOUNDARY_CONDITION`
- `TURBULENCE_WALL`
- `RUNNING_AVERAGE_OUTPUT`

Multi-Scale Battery Modeling Approach

For pouch batteries the thermal-electric response can be modeled using a multi-scale approach which bridges the scale between the cell domain and the electrode scale. To achieve this, a multi-scale multi-domain approach (*electric_potential* = battery_msmd) is utilized. In this approach, the electric

potential fields are modeled in a cell composite, or more specifically the current collector, together with the thermal field. This is then coupled using a multi-scale approach down to the subscale, where the electrochemical phenomena, that is, the inner workings of the battery, are modeled.

Cell scale

At the cell scale the temperature and a pair of electric potentials are solved. More specifically, the 3D battery geometry is treated as a homogenized medium with orthotropic electrical and thermal conductivities. Positive and negative electric potential is computed in the battery from charge conservation at the current collectors of the cell composite. For the negative current collector the governing equation is:

$$\nabla\left(\sigma_{-}\nabla\phi_{-}\right) - j = 0 \tag{3}$$

Equivalently at the positive current collector the governing equation is:

$$\nabla\left(\sigma_{+}\nabla\phi_{+}\right) + j = 0 \tag{4}$$

where j is the transfer current density source term and is modeled through an electrochemical sub-model. The above comes with associated boundary conditions such as current, for example, constant current discharge, or voltage.

The energy equation in AcuSolve is modified by an additional source term that couples both sources of ohmic (or joule) heating from the current collectors and electrochemical reaction heat. The source term is given by:

$$q = \sigma_{-}\left|\nabla\phi_{-}\right|^2 + \sigma_{+}\left|\nabla\phi_{+}\right|^2 + q_{\mathrm{EC}} \tag{5}$$

where the first two terms are ohmic heating and the last is electrochemical heating (from the electrochemical sub-model). Currently, AcuSolve supports a single type of electrochemical sub-model: an equivalent circuit type model. This model supports first and second order ECM models.

Battery ECM with Joule Heating Modeling Approach

In this approach the same ECM model as the MSMD model is used, however the electric potential fields are not modeled explicitly in the cell; they are modeled in only in the electrical connected components, for example, tabs/terminals and busbars. The ECM model calculates the voltage drop and current through the homogenized part of the battery cell and applies the associated boundary/interface conditions automatically. This approach is enabled using the following command *`electric potential`* = battery_joule_heating in the `EQUATION` section.

EXTERNAL_CODE

Specifies communication parameters for coupling with an external solid/structural code.

Type

AcuSolve Command

Syntax

EXTERNAL_CODE {parameters}

Qualifier

This command has no qualifier.

Parameters

communication (enumerated) [=socket]

Method of communication between AcuSolve and the external code.

pipe	Pipe communication. External code reads AcuSolve messages from standard in and writes to standard out. Requires *launch_command*.
socket	Socket connection on a given port. Requires *socket_initiate* and *socket_port*.
fifo	Use a fifo special file. Requires *fifo_send_file* and *fifo_receive_file*.

communication (string) [no default]

The command, with parameters, that executes the external code. Used with pipe communication.

socket_initiate (boolean) [=on]

Flag specifying AcuSolve to initiate the socket connection. Otherwise the external code will post the socket. Used with socket communication.

socket_host or host (string) [no default]

User-given name of the host that runs the external code. Used with socket communication and *socket_initiate* = off.

socket_port or port (integer) >=1 (=10000)

Socket port for communication with external code. Used with socket communication.

fifo_send_file (string) [no default]

User-given name of the fifo special file that the external code opens for reading. Used with fifo communication.

fifo_receive_file (string) [no default]

User-given name of the fifo special file that the external code opens for writing. Used with fifo communication.

multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the computed forces, moments, and heat fluxes from AcuSolve before sending them to the external code. If none, no scaling is performed.

filter (enumerated) [=none]
Method of filtering the computed forces, moments and heat fluxes from AcuSolve before sending them to the external code.

none	No filter.
iir	Infinite impulse response digital filter. Requires *iir_input_coefficients* and *iir_output_coefficients*.

iir_input_coefficients (array) [={1}]
Infinite impulse response digital filter coefficients applied to computed values. Used with iir filter.

iir_output_coefficients (array) [={}]
Infinite impulse response digital filter coefficients applied to previously filtered values. Used with iir filter.

Description

This command provides support for solving Direct Coupling Fluid Structure Interaction (DC-FSI) problems by allowing AcuSolve and an external solid/structural code to run in a coupled manner. Forces, moments, and heat fluxes are communicated to the solid/structural code, and displacements are returned to AcuSolve. The code coupling interface (CCI) library handles the required communication and interpolations. Once a solid/structural code is linked to this library, that code can use CCI functions to run in conjunction with AcuSolve to solve a DC-FSI problem. Both explicit and implict couplings with the external code are possible. See also the EQUATION, TIME_SEQUENCE, and EXTERNAL_CODE_SURFACE commands. Complete documentation for CCI is in the file doc/code_coupling_interface.txt, which can be found in the AcuSolve distribution.

The *communication* parameter determines which of three protocols is used. For example,

```
EXTERNAL_CODE {
   communication       = pipe
   launch_command      = "program args"
}
```

Here AcuSolve launches (only from processor 1) the external code (and arguments) as given by the *launch_command* parameter. This program must read AcuSolve messages from standard in and write to standard out.

An example of the socket protocol is given by:

```
EXTERNAL_CODE {
   communication       = socket
   socket_initiate     = off
   socket_host         = "hostname"
   socket_port         = 10000
}
```

If *socket_initiate* is on, AcuSolve will initiate the socket connection on port *socket_port*.
It is then the responsibility of the external code to connect to that socket. Following this, all

communications will be done through sockets. Alternatively, the external code may post the socket when *socket_initiate*=off. In this case, AcuSolve (from processor 1) connects to the posted socket on the machine given by *socket_host* and port given by *socket_port*.

The third protocol is fifo:

```
EXTERNAL_CODE {
     communication              = fifo
     fifo_send_file             = "acuSolve2externalCode.txt"
     fifo_receive_file          = "externalCode2acuSolve.txt"
}
```

This is a quick way of implementing communication between two codes. The external code must open *fifo_send_file* for reading and *fifo_receive_file* for writing. Processor one of AcuSolve and the external code need to be on the same Linux or Unix machine. Windows is not supported. The fifo files are created by using the mkfifo Unix utility.

The accuracy and stability of the solution of the DC-FSI system depend on how the external code equations are coupled to the fluid equations. Two strategies are supported:

- Conventional Sequential Staggered (CSS): This uses a single-pass of displacements/rotations/temperatures and forces/moments/heat-fluxes at each time step. The result is essentially an explicit scheme since the data comes from the previous time step. CSS is used when there is exactly one nonlinear stagger per time step (*max_stagger_iterations*=1 in the STAGGER command).
- Multi-Iterative Coupling (MIC): This strategy corrects the interfacial forces via a multi-pass transfer of displacements/rotations/temperatures and fluid forces/moments/heat-fluxes. From numerical experiments, robustness and efficiency of the scheme are obtained for a mass-density and specific heat ratios of O(1). The scheme requires at least two nonlinear stagger iterations per time step. The interfacial force/moment/heat flux and displacement/rotation/temperature residuals can be obtained by printing the *.Log file, generated by AcuRun, with a verbose level of two.

There are two ways to modify the fluid forces, moments, and heat fluxes before they are sent to the solid/structural code. Such a modification is often necessary to get a robust solution, especially at start up. The first is by specifying a multiplier function:

```
EXTERNAL_CODE {
     communication                  = pipe
     launch_command                 = "program args"
     multiplier_function            = "ramp up"
}
MULTIPLIER_FUNCTION( "ramp up" ) {
   type                             = piecewise_linear
   curve_fit_values                 = {  0., 0. ;
                                        10., 1. ;
                                       100., 1. ; }

curve_fit_variable                  = time
}
```

This example has the effect of decoupling the two codes initially and then linearly ramping up the forces over the first 10 seconds until the full fluid forces are applied within the external code. A multiplier function of mic filter type can also be used. This is most effective when there at least two nonlinear stagger iterations per time step.

The second method uses a digital filter to smooth the forces in time. An iir filter gives an infinite impulse response filter, which is defined by:

$$f_n = \sum_{i=1}^{N_a} a_i \tilde{f}_{n+1-i} + \sum_{i=1}^{N_b} b_i f_{n-i} \tag{6}$$

with the constraint that

$$\sum_{i=1}^{N_a} a_i + \sum_{i=1}^{N_b} b_i = 1 \tag{7}$$

where a_i, $i = 1, \ldots, N_a$ is given by *iir_input_coefficients*; b_i, $i = 1, \ldots, N_b$ is given by *iir_output_coefficients*; f_n is a component of the fluid force applied to the structure at time step n; and $\tilde{f}_n$ is the value of this component before filtering at time step n. N_a0 and N_b are determined by the number of elements of the corresponding arrays. If the constraint is not satisfied, AcuSolve automatically scales the coefficients appropriately. For example,

```
EXTERNAL_CODE {
   ...
   filter                       = iir
   iir_input_coefficients       = { 0.8, 0.8 }
   iir_output_coefficients      = { -0.6 }
}
```

FIELD_BOUNDARY_CONDITION

Specifies field quantity for boundary conditions.

Type

AcuSolve Command

Syntax

FIELD_BOUNDARY_CONDITION("name") {parameters...}

Qualifier

User-given name.

Parameters

type (enumerated) [=value]
Mesh motion type.

value	Uniform value.

field (string) [=none]
User-given name of a field. The field entry must be listed in the EQUATION::FIELDS entry.

multiplier_function (string) [=none]
User-given name of the multiplier function used to scale the field value.

value (real) [=0]
Uniform value for the specified field.

Description

This command is used to define the boundary condition for a given field. The *field_boundary_condition entry* is referenced by *simple_boundary_condition*.

```
FIELD_BOUNDARY_CONDITION( "inlet water" ) {
   field          = "water"
   type           = value
   value          = 1
}
```

FLEXIBLE_BODY

Specifies a solid body for fluid-structure interaction.

Type

AcuSolve Command

Syntax

`FLEXIBLE_BODY("name") {parameters...}`

Qualifier

User-given name.

Parameters

`equation` *(enumerated) [=mesh_displacement}*
Fluid-structure interaction modeling strategy.

mesh_displacement or mesh	Fluid mesh displacement boundary conditions.
velocity or vel	Fluid velocity boundary conditions.

`num_modes` *(integer) >0 [=1]*
Number of modes to model solid body dynamics.

type (enumerated) [=trapezoidal]
Flexible body type.

trapezoidal	Trapezoidal rule used for time integration.
user_function or user	Solid body deformation described by a user function.

`num_sub_steps` *(integer) >=1 (=1)*
Number of sub-steps in time integration. Used with trapezoidal type.

`mass` *(array) [={1}]*
Mass array of dimension *`num_modes`*. Used with trapezoidal type.

`stiffness` *(array) [={1}]*
Stiffness array of dimension *`num_modes`*. Used with trapezoidal type.

`stiffness_multiplier_function` *(string) [=none]*
User-given name of the multiplier function for scaling the stiffness. If none, no scaling is performed.

`damping` *(array) [={0}]*
Damping array of dimension *`num_modes`*. Used with trapezoidal type.

`damping_multiplier_function` *(string)[=none]*
User-given name of the multiplier function for scaling the damping. If none, no scaling is performed.

contact_constraints (array) [={}]
Array of constraints that limit the motion of the solid body. Used with trapezoidal type.

external_force (array) [={0}]
External force array of dimension *num_modes*.

external_force_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the external force. If none, no scaling is performed.

initial_displacement (array) [={0}]
Initial displacement array of dimension *num_modes*.

initial_velocity (array) [={0}]
Initial velocity array of dimension *num_modes*.

initial_force (array) [={0}]
Initial force array of dimension *num_modes*.

internal_force_multiplier_function (string)[=none]
User-given name of the multiplier function for scaling the internal force. If none, no scaling is performed.

user_function or user (string) [no default]
Name of the user-defined function. Used with user_function type.

user_values (array)[={0}]
Array of values to be passed to the user-defined function. Used with user_function type.

surface_outputs (list) [={0}]
Array of SURFACE_OUTPUT command qualifiers used to calculate the force and moment on the solid body.

evaluation (enumerated) [=once_per_time_step]
Frequency of evaluating the external forces.

once_per_solution_update or often	Every solution update.
once_per_time_step or step	Every time step.

filter (enumerated) [=none]
Type of filter to smooth the external forces.

none	No filter.
iir	Infinite impulse response digital filter. Requires *iir_input_coefficients* and *iir_output_coefficients*.

iir_input_coefficients (array) [={1}]
Infinite impulse response digital filter coefficients applied to computed values. Used with iir filter.

iir_output_coefficients (array) [={}]
Infinite impulse response digital filter coefficients applied to previously filtered values. Used with iir filter.

Description

This command supports the modeling of simple fluid-structure interaction (FSI) problems. A few examples of such problems are:

- Flow induced vibration of a solid wall and coupling to computational aero-acoustics (CAA).
- VIV (vortex induced vibration) modeling of oil platform risers.
- Deformation of vein walls due to pulsing blood flow.

FSI is defined as simulating the bidirectional interaction (coupling) between a fluid flow and a deforming solid/structural model. This definition is very specific. It does not include other types of couplings such as fluid with thermal solid/structures or fluid with rigid body dynamics, both of which are already available through other commands in AcuSolve. The coupling may be very complex depending on the degree of deformation of the solid/structure, especially for large deformations leading to nonlinear solid/structure stress/strain behavior. Here, only linear solid/structural deformation is considered that can be represented by the linear finite element dynamic systems of equations:

$$Ma + Cv + Kd = F \tag{8}$$

where a , v, and d are the nodal acceleration, velocity, and displacement fields respectively; M, C, and K are the matrices of mass, damping, and stiffness; and F is the external force. The linearity assumption means the matrices M, C, K, and F are independent of the solution d (and its first and second time derivatives, v and a). Given this assumption, the above equation of motion may be decomposed into its eigen modes:

$$\left(K - \lambda_i M\right) S_i = 0, i = 1, \ldots \text{num_modes} \tag{9}$$

where λ_i is the ith eigenvalue of the above system with eigenvector $\boldsymbol{S}i$. The number of modes in the finite element model of the solid/structure is given by num_modes. Projection of the equation of motion into the eigenspace yields:

$$m\ddot{y} + c\dot{y} + ky = f \tag{10}$$

where y is the vector of amplitudes, henceforth referred to as the "displacements" of the eigen modes, a dot represents the time derivative, and

$$\begin{aligned} m &= S^T M S = I \\ c &= S^T C S \\ k &= S^T K S = \text{diag}(\lambda_i) \\ f &= S^T F \end{aligned} \tag{11}$$

The damping matrix C is assumed to be in the above eigen space. This is actually a minor assumption, since for most problems C is either zero or proportional to K. This also means that the matrix c is diagonal. If this is not quite right, only the diagonal part of c is used.

The above equation for y is a set of num_modes independent ordinary differential equations (ODEs). To solve it, F needs to be computed first and projected to f. Each component of y and its time derivatives can then be solved for. From the solution, the original dependent variables can be reconstructed:

$$\begin{aligned} d &= Sy \\ v &= S\dot{y} \\ a &= S\ddot{y} \end{aligned} \tag{12}$$

A further simplifying assumption is made that only a limited number of eigen modes is needed from the above system to produce a good representation of the solid deformation to be coupled with the fluid flow.

A simple FSI model is implemented in three parts.

- Solid model: A solid/structural finite element model must first be created. The first num_modes eigen modes of this model is computed. This step is performed with an existing linear solid/structural code, such as OptiStruct, Abaqus, ANSYS, Nastran, or other means. Analytic solutions may of course be used if available. This gives a set of eigenvalues and eigenvectors on the solid mesh.
- Projection: Next, an independent fluid mesh is created, and the fluid surfaces that are in contact with the solid are identified. The eigenvectors from the surfaces nodes of the solid mesh are then projected to the surface nodes of the fluid mesh. The AcuPev program is a tool to facilitate this process; see the *AcuSolve Programs Reference Manual* for details. An analytic solution may be possible for simple problems.
- AcuSolve: The projected eigen system is entered into AcuSolve, which will then solve the transient coupled linearized deformation and the fluid flow. The process is somewhat similar to the current AcuSolve solution of coupled rigid body dynamics (see `MESH_MOTION` with *type*=rigid_body_dynamic).

This command provides most of the data to implement this strategy. For example,

```
FLEXIBLE_BODY( "riser" ) {
   equation                                = mesh_displacement
   num_modes                               = 5
   type                                    = trapezoidal
   mass                                    = { 1, 1, 1, 1, 1 }
   damping                                 = { 0, 0, 0, 0, 0 }
   damping_multiplier_function             = none
   stiffness                               = { 1, 1.5, 3.2, 6.8, 12.6 }
   stiffness_multiplier_function           = none
   external_force                          = { 0, 0, 0, 0, 0 }
   external_force_multiplier_function      = none
   initial_displacement                    = { 0, 0, 0, 0, 0 }
   initial_velocity                        = { 0, 0, 0, 0, 0 }
   initial_force                           = { 0, 0, 0, 0, 0 }
   internal_force_multiplier_function      = none
   surface_outputs                         = { "side wall", "bottom wall" }
   evaluation                              = once_per_time_step
   filter                                  = none
}
```

The type of modeling is given by *equation* which may be either mesh_displacement or velocity; see below. The number of eigen modes is given by num_modes, which is also the dimension of all arrays in this command. The time integration strategy is given by *type*; time integration is performed internally for *type*=trapezoidal. See below for *type*=user_function. The diagonal of the mass matrix m is given by mass, which is typically a vector of ones. The diagonal of the stiffness matrix k is given by *stiffness*, which is typically the vector of eigenvalues. The diagonal of the damping matrix c is given by damping; c in turn is given by the projection of C onto the eigenvectors matrix. Forces not induced by the fluid, such as back pressure, may be given by *external_force*, which may be scaled with *external_force_multiplier_function*. The initial conditions are given by *initial_displacement* and *initial_velocity*. The initial fluid force is given by *initial_force*. These last three parameters

will be reset at restart. The surfaces used to compute the fluid forces, that is, F, applied to the solid are given by *surface_outputs*, which references one or more SURFACE_OUTPUT commands. The fluid forces may be scaled with *internal_force_multiplier_function*. The usual use for this is to ease into a fluid/structure simulation by implementing a multiplier function that starts at zero and ramps up to one.

The rest of the data to implement the above strategy is given by one or more NODAL_BOUNDARY_CONDITION commands. For example:

```
NODAL_BOUNDARY_CONDITION( "mesh_x_displacement on side wall" ) {
   nodes                                = Read( "riser.side-wall.nbc" )
   variable                             = mesh_x_displacement
   type                                 = flexible_body
   flexible_body                        = "riser"
   nodal_modes                          = Read( "riser.side-wall.xeig" )
}
```

The first parameter is the usual list of nodes. *variable* may be any of the four mesh displacement variables or any of the four velocity variables, depending on the type of modeling; see below. The direction variables also require the usual direction parameters, including *direction_type*. A *flexible_body* type indicates that this boundary condition is used as part of a FSI coupling. The corresponding FLEXIBLE_BODY command is provided by *flexible_body*, which in particular provides num_modes. The eigenvectors **S**i are given by *nodal_modes*, which contains num_modes+1 columns corresponding to the node number and the num_modes eigenvectors. The set of nodes in *nodal_modes* must be the same as in nodes. The eigenvectors are used for two purposes: to project **F** to form the modal force **f**; and to form **d** and **v**, which are needed for fluid boundary conditions.

The accuracy and stability of the solution of the flexible body equations depend on how those equations are coupled to the fluid equations. Two time integration strategies are supported:

- Conventional Sequential Staggered (CSS): This uses a single-pass of displacements and forces at each time step. The result is essentially an explicit scheme since the data comes from the previous time step. CSS is used when there is exactly one nonlinear stagger per time step (*max_stagger_iterations*=1 in the STAGGER command).
- Multi-Iterative Coupling (MIC): This strategy corrects the interfacial forces via a multi-pass transfer of mesh displacements and fluid forces. From numerical experiments, robustness and efficiency of the scheme is obtained for a mass-density ratio of O(1). The scheme requires at least two nonlinear stagger iterations per time step. The interfacial force and displacement residuals can be obtained by printing the *.Log file (generated by AcuRun) with a verbose level of two.

The parameter *contact_constraints* may be used to specify a set of contact conditions to be imposed on the time evolution of the flexible body. Only simple contact between a set of points attached to the flexible body with a set of fixed rigid planes is accommodated. Contact constraints are supported only with trapezoidal type. Contact introduces a strong nonlinearity on the linear model given above, and thus must be modeled with care.

Consider the contact of a point **x**0 on the flexible body with a rigid plane. The rigid plane is specified by a point on its surface, denoted by **x**p, and its normal direction into the allowable half space, denoted by **n**p:

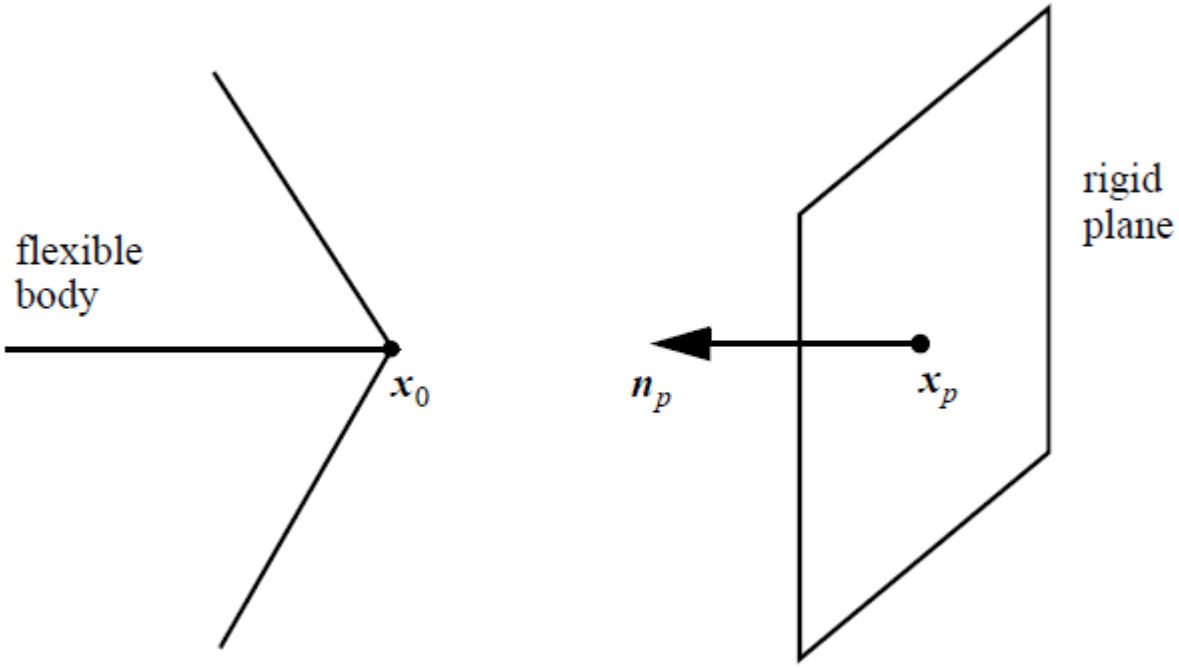


Figure 1: Flexible Body Contact with a Rigid Plane

The contact condition may then be written as

$$(x_0 + d_0) \cdot n_p \geq x_p \cdot n_p \tag{13}$$

where $d_0 = d_0(t)$ is the displacement of the flexible body as a function of time at point x_0. Recall that the displacement is the sum of contributions from each mode of the flexible body; in other words:

$$d_0 = \sum_i y_i S_i \tag{14}$$

where y_i is the displacement of the i-th mode, and **S**i is the i-th eigenvector of the flexible body at point **x**0 . Combining the above two equations yields:

$$\left(\sum_i y_i S_i \right) \bullet n_p \geq (x_p - x_0) \bullet n_p \tag{15}$$

which may be simplified to:

$$\sum_i c_i y_i \geq b \tag{16}$$

where:

$$c_i = \boldsymbol{S}_i \bullet \boldsymbol{n}_p$$
$$b = (\boldsymbol{x}_p - \boldsymbol{x}_0) \bullet \boldsymbol{n}_p \qquad (17)$$

Note: y_i, c_i, and b are all scalar values.

Formally, the solution for the displacement of the flexible body with contact is the transient solution for **y** subject to the inequality constraints above. This system may be solved using the "interior point method" of "quadratic programming".

In addition to the above constraint on the modal displacement field, the "restitution coefficient", denoted by *a*, is used to impose the following condition on the modal velocity field when the contact condition occurs:

$$\boldsymbol{c} \bullet \dot{\boldsymbol{y}}_{n+1} = -\alpha \boldsymbol{c} \bullet \dot{\boldsymbol{y}}_n \qquad (18)$$

where $\dot{\boldsymbol{y}}_n$ and $\dot{\boldsymbol{y}}_{n+1}$ are the modal velocities at time step n and n+1, respectively, and the contraction is over all of the modes. The above equation approximately sets the **c** component of modal velocity after contact occurs to the negative of the corresponding modal velocity prior to contact scaled by the restitution coefficient.

Note: The restitution coefficient is between zero and one; *a* = 0 models an inelastic collision, and *a* = 1 models a perfectly elastic collision. The formulation above is for one contact constraint; AcuSolve supports any number of constraints.

The parameter *contact_constraints* is a two dimensional array with num_modes+2 columns, corresponding to the vector **c** (num_modes columns), the b value, and the restitution coefficient, *a*. Each row of the array corresponds to one contact constraint. This array is usually constructed using the *contact_constraints_file* option of AcuPev; see the *AcuSolve Programs Reference Manual*.

For example, assume that a point on the flexible body is confined between two parallel rigid planes, one at a distance of 0.1 in front of this point and the other 0.15 behind. Assume this body is modeled with two modes, and the projection of the eigenvectors of this point in the normal direction to the first plane are {0.2, 0.1}. The corresponding values for the second plane are the negative of these values. Moreover, assume that the restitution coefficient of both conditions is 0.6. That is, in the absence of any fluid or other forces, the flexible body bounces off the rigid plane at 60 percent of its original speed at the time of contact. These constraints may then be specified as

```
FLEXIBLE_BODY( "body with simple contact" ) {
   equation                          = mesh_displacement
   num_modes                         = 2
   ...
   num_sub_steps                     = 4
   contact_constraints               = { +0.2, +0.1, -0.1, 0.6 ;
                                         -0.2, -0.1, -0.15, 0.6 ; }
}
```

The b values for both conditions above are negative. This is expected. In general, if a flexible body at its reference condition, that is, with no deformation, is not in contact, then all b values will be negative. If

the flexible body is in contact at its reference position, then the b value(s) corresponding to the contact point(s) will be zero.

You must be very careful in modeling a FSI problem with the above *contact_constraints* parameter. There are three main sources of error.

By far the most important source of error is the validity of the flexible body shape as modeled by a limited number of modes when contact occurs. You can typically use fairly small number of modes to accurately model the interaction between a fluid and a flexible body under normal conditions. But contact may require a different set of modes in order to accurately model the shape of the flexible body. When contact occurs, the rigid plane effectively imposes a force on the flexible body. This force is imposed at the point of contact (point **x**0 above). The shape of the flexible body under the influence of contact is as valid as to the degree in which the contact force is represented by the eigenvectors. Let the contact force be given by $F_i=F_i(x)$. The error in the representation of this force is given by:

$$\text{relative error} = \frac{\left\| F_i - S(S^T S)^{-1} S^T F_i \right\|}{\left\| F_i \right\|} \qquad (19)$$

The second source of error is in modeling of the disruptive effect of contact on the fluid forces. In other words, when contact occurs, the flexible body rapidly stops or even reverses direction. This typically causes a rapid change in fluid forces. To minimize the inaccuracy caused by this effect, smaller time increments in the fluid solution may be required.

The final source of error is in the time evolution of the flexible body itself, that is, the contact may occur in the middle of a time step. Thus the flexible body motion within the time step is approximated as is the fluid motion. A partial correction to the body motion but not the fluid motion can be added. The parameter *num_sub_steps* is introduced for this, which is only available for trapezoidal type. The flexible body time evolution is solved with *num_sub_steps* substeps within each fluid time step. Higher values of this parameter provides more accurate modeling of the discontinuities introduced by contact.

A flexible body type of user_function may be used to model more complex behaviors including other kinds of time integration; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user defined functions. Unlike the trapezoidal type described above, a user_function type does not read or use the mass, stiffness, and damping parameters. Instead, this data is supplied by you directly to the user function, or possibly they are not used at all. For example, a flexible baffle inside of a channel may be described by:

```
FLEXIBLE_BODY( "baffle" ) {
   equation                              = mesh_displacement
   num_modes                             = 1
   type                                  = user_function
   surface_outputs                       = { "baffle" }
   user_function                         = "usrClientServer"
   user_strings                          = { "usrFSI.pl" }
}
```

where the user function is implemented in the perl script `usrFSI.pl` which is controlled through the client/server mechanism. See the *AcuSolve User-Defined Functions Manual*.

"usrClientServer" is implemented as:

```
#include "acusim.h"
```

```
#include "udf.h"
UDF_PROTOTYPE(usrClientServer) ;
Void usrClientServer(        UdfHd     udfHd,
                             Real*
                                       outVec,
                             Integer   nItems,
                             Integer   vecDim )

{

char          buffer[1024] ;           /* a string buffer                          */
char*         name ;                   /* name of the UDF                          */
Integer       i ;                      /* a running index                          */
Real*         disp ;                   /* displacement (curr or prev)              */
Real*         dispC ;                  /* next displacement                        */
Real*         extForce ;               /* external force                           */
Real*         intForce ;               /* internal force                           */
Real*         timeInc ;                /* time increment                           */
Real*         usrVals ;                /* user supplied values                     */
Real*         vel ;                    /* velocity (curr or prev)                  */
Real*         velC ;                   /* next velocity                            */
String        command ;                /* execution command                        */
String*       usrStrs ;                /* user supplied strings                    */
udfCheckNumUsrVals(                    udfHd,                      0              ) ;
udfCheckNumUsrStrs(                    udfHd,                      1              ) ;
usrVals       = udfGetUsrVals(         udfHd                                      ) ;
usrStrs       = udfGetUsrStrs(         udfHd                                      ) ;
command       = usrStrs[0] ;
name          = udfGetName(            udfHd                                      ) ;
timeInc       = udfGetTimeInc(         udfHd                                      ) ;
dispC         = outVec ;
velC          = outVec + nItems ;

/* Start the client */
   if ( udfFirstCall(udfHd) ) {
   udfOpenPipePrim(                    udfHd,                       command       ) ;
   }

/* If the first time step */
   if ( udfFirstStep(udfHd) ) {
   disp       = udfGetFbdData( udfHd, name, UDF_FBD_CURR_DISPLACEMENT             ) ;
   vel        = udfGetFbdData( udfHd, name, UDF_FBD_CURR_VELOCITY                 ) ;
   for ( i = 0 ; i < nItems ; i++ ) {
       dispC[i]    = disp[i] ;
       velC[i]     = vel[i] ;
   }
       return ;
}

/* Otherwise, get the data from the external program */
   intForce    = udfGetFbdData( udfHd, name, UDF_FBD_INTERNAL_FORCE               ) ;
   extForce    = udfGetFbdData( udfHd, name, UDF_FBD_EXTERNAL_FORCE               ) ;
   disp        = udfGetFbdData( udfHd, name, UDF_FBD_PREV_DISPLACEMENT            ) ;
   vel         = udfGetFbdData( udfHd, name, UDF_FBD_PREV_VELOCITY                ) ;
   for ( i = 0 ; i < nItems ; i++ ) {
       udfWritePipe(   udfHd,        "%d %.16g %.16g %.16g %.16g %.16g           ",
                       i,            timeInc,        disp[i],
                       vel[i],       intForce[i],    extForce[i]                  ) ;
}
for ( i = 0 ; i < nItems ; i++ ) {
       udfReadPipe(    udfHd,        buffer,         1024                         ) ;
       sscanf(         buffer,       "%le %le",      &dispC[i],
                       &velC[i]                                                   ) ;
```

```
}

} /* end of usrClientServer() */
```

The perl script `usrFSI.pl` may be implemented as:

```
#!/usr/bin/env perl
$| = 1 ;
if ( scalar(@ARGV) != 0 ) {
    die "Usage: $0" ;
}
# Hard code matrices
@mass    =    (    200.      );
@damp    =    (    0.        );
@stiff   =    (    50.       );
$doLog = 0 ;
if ( $doLog ) {
    open( LG, ">fsi.log" ) || die "unable to open fsi.log" ;
}

# Loop through the steps/modes and advance
while( <> ) {
    chomp ;
    ( $indx, $timeInc, $dispP, $velP, $intF, $extF ) = split( /\s/, $_, 6 ) ;
    print LG "< $indx $timeInc $dispP $velP $intF $extF\n" if $doLog ;
    $timeInc != 0 || die "timeInc == 0" ;
    $fct1          = $mass[$indx]          / $timeInc ;
    $fct2          = $damp[$indx]          * 0.5 ;
    $fct3          = $stiff[$indx]         * $timeInc /4. ;
    $fct4          = $stiff[$indx] ;
    $rhs           = $intF + $extF
                   + $fct1 * $velP
                   - $fct2 * $velP
                   - $fct3 * $velP
                   - $fct4 * $dispP ;
    $lhs           = $fct1 + $fct2 +$fct3 ;
   $lhs != 0 || die "lhs == 0" ;
   $velC           = $rhs / $lhs ;
   $dispC          = $dispP + $timeInc * ( $velP + $velC ) / 2 ;
   printf "%.16g %.16g\n", $dispC, $velC ;
   print LG "> $dispC $velC\n" if $doLog ;
}
```

Note the use of the function `udfGetFbdData()`. This is similar to `udfGetOsiData()`, except it provides the solution data for a given `FLEXIBLE_BODY` command. See the *AcuSolve User-Defined Functions Manual* for a list of valid data names. *`outVec`* is used to return both the displacement and velocity at step n+1.

The *`evaluation`* parameter controls how often the external force vector on the solid body is updated. Setting this to once_per_time_step freezes the external force for the rest of the time step.

once_per_solution_update updates the external force every time the fluid solution is updated.

The computed projected forces of AcuSolve may be smoothed in time by applying a digital filter before using the forces for computing the modal response. An iir filter gives an infinite impulse response filter, which is defined by:

$$f_n = \sum_{i=1}^{N_a} a_i \tilde{f}_{n+1-i} + \sum_{i=1}^{N_b} b_i f_{n-i} \quad (20)$$

with the constraint that:

$$\sum_{i=1}^{N_a} a_i + \sum_{i=1}^{N_b} b_i = 1 \quad (21)$$

where $a_i, i = 1, \ldots, N_a$ is given by *iir_input_coefficients*; is given by *iir_output_coefficients*; $b_i, i = 1, \ldots, N_b$ f_n is a component of the fluid force applied to the flexible body at time step n ; and f_n is the value of this component before filtering at time step n. N_a and N_b are determined by the number of elements of the corresponding arrays. If the constraint is not satisfied, AcuSolve automatically scales the coefficients appropriately. For example,

```
FLEXIBLE_BODY( "FSI with digital filter" ) {
   ...
   filter                    = iir
   iir_input_coefficients    = { 0.8, 0.8 }
   iir_output_coefficients   = { -0.6 }
}
```

Two modeling strategies are supported:

- equation=mesh_displacement. ALE or specified mesh must be used. The NODAL_BOUNDARY_CONDITION commands containing *type*=flexible_body must impose boundary conditions on components of the mesh displacement. The fluid velocity is then matched to the mesh motion via other NODAL_BOUNDARY_CONDITION commands.
- equation=velocity. Modeling of the mesh deformation is assumed to be not important and thus its effect may be approximated through velocity boundary conditions on the wall. The NODAL_BOUNDARY_CONDITION commands containing *type*=flexible_body must impose boundary conditions on components of the velocity.

The solution strategy to advance an FSI solution using the CSS method is implemented in two steps:

- The displacement or velocity values of the eigen modes are used to compute new boundary condition values for the mesh displacement or fluid velocity variables at each node of the flexible body NODAL_BOUNDARY_CONDITION commands.
- At the end of each time step, the forces on the surfaces listed in *surface_outputs* are computed. The nodal values of the forces are multiplied by the eigenvectors to compute the internal forces on the flexible body. In other words, the force field applied by the fluid is projected into the eigen space of each mode. This force is added to the external force and the result is used to advance the displacement or velocity of each mode.

The MIC strategy consists of the same two steps executed every nonlinear stagger iteration with the forces and displacements also updated every stagger instead of every time step.

There are many potential difficulties associated with performing FSI simulations. Some of the issues that need to be considered are:

- All three components of mesh displacement (or velocity) may be specified, or just two components, or even just the normal component.
- Mathematically, however, if a direction boundary condition is used, either only one boundary condition per node should be used, or care must be taken to ensure orthogonal components. Otherwise, error in the accumulation of the contribution to the force **f** may occur.
- You must be careful when using direction boundary condition with mesh displacement, since the eigenvectors may only be valid for a particular direction, and the mesh may shift as a result of the boundary condition.
- The collection of nodes in the `NODAL_BOUNDARY_CONDITION` commands corresponding to a particular `FLEXIBLE_BODY` command may be a subset or a superset of the nodes contained in the *`surface_outputs`* list. If a boundary condition node does not have a corresponding surface output node, then that node does not contribute to **f**, but the flexible body does control its motion. This may be used, for example, to define the motion of all nodes in the problem, while only a subset contribute to the solid deformation. A less useful scenario is a node in the *`surface_outputs`* list not appearing in any nodal boundary condition. Probably the only valid case is when this node has other boundary conditions that take precedence. In either case, a warning message will be issued.
- Currently, a zero intersection between these two sets of nodes, flexible body nodal boundary conditions and the surface output list, is disallowed, even though a case can be envisioned where a flexible body independent of any fluid force is used just as a simple mechanism to move the nodes.
- The precedence parameter in a `NODAL_BOUNDARY_CONDITION` command may be used to suppress a FSI boundary condition on a node.

IMMERSED_BOUNDARY_SURFACE

This command has an option of including floating surfaces from which to write output data.

Type

AcuSolve Command

NAME

`IMMERSED_BOUNDARY_SURFACE`- Model specific physics on/in the interior of floating surface mesh.

Syntax

`IMMERSED_BOUNDARY_SURFACE("name") {parameters...}`

Qualifier

User-defined name.

Parameters

`surface_sets` *(list) [={}]*

List of partial surface set names (strings) to use in this immersed boundary surface. When using this option, the connectivity, shape and coordinates of the surfaces are provided by the surface set container.

`active_type` *(enumerated) [=wall]*

Type of boundary surface.

none	Immersed boundary will be ignored. This option enables floating surface monitoring.

MESH_MOTION *(list) [={}]*

Mesh motion command.

`create_surface_output` *(boolean) [=on]*

Flag specifying whether to create a surface output on the given surface elements.

`surface_integrated_output_frequency` *(integer) >=0 [=1]*

Used when *`create_surface_output`* = on to set frequency of integrated surface output. If zero, this option is ignored.

`surface_integrated_output_time_interval` *(integer) >=0 [=0]*

Used when *`create_surface_output`* = on to set time interval of integrated surface output. If zero, this option is ignored.

`surface_statistics_output_frequency` *(integer) >=0 [=1]*

Used when *`create_surface_output`* = on to set frequency of surface statistics output. If zero, this option is ignored.

`surface_statistics_output_time_interval` *(integer) >=0 [=0]*

Used when *`create_surface_output`* = on to set time interval of surface statistics output. If zero, this option is ignored.

surface_nodal_output_frequency (integer) >=0 [=0]
: Used when *create_surface_output* = on to set frequency of nodal output. If zero, this option is ignored.

surface_nodal_output_time_interval (real) >=0 [=0]
: Used when *create_surface_output* = on to set time interval of nodal output. If zero, this option is ignored.

surface_num_saved_states (integer) >=0 [=0]
: Used when *create_surface_output* = on to set number of saves values of nodal output. If zero, all are kept.

MESH_MOTION

Specifies mesh motion of nodes.

Type

AcuSolve Command

Syntax

```
MESH_MOTION("name") {parameters...}
```

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Mesh motion type.

none	No boundary conditions imposed.
zero	Nodes fixed in space.
translation	Simple linear translation.
rotation	Simple rotation.
piecewise_linear or linear	Motion computed from piecewise linear curve fit of two points.
cubic_spline or spline	Motion computed from cubic spline curve fit of two points.
user_function or user	Motion described by a user function.
rigid_body_dynamic or rigid	Nodes move dynamically as a rigid body.
interpolated_motion	Nodes move based on the motion of interpolated motion surfaces. Requires *interpolated_motion_surfaces*.

translation_velocity or vel (array) [={0,0,0}]

Translation velocity. Used with translation type.

translation_variable (enumerated) [=time]

Scaling of translation velocity to obtain translation. Used with translation type.

time	Time.
multiplier_function	The multiplier function for the temporal function of the translation. Requires *translation_variable_multiplier_function*.
multiplier_function_on_time	The multiplier function for the temporal function of the translation velocity for computing the translation

by time integration of translation velocity. Requires *translation_variable_multiplier_function*.

translation_variable_multiplier_function *(string) [=none]*
User-given name of the multiplier function for computing the translation. If none, the translation is zero. Used with multiplier_function translation variable.

rotation_center *(array) [={0,0,0}]*
Center of rotation, specified in global xyz coordinate system. Used with rotation type.

angular_velocity *or* *ang_vel* *(array) [={0,0,0}]*
Angular velocity, in radians per unit of time. Used with rotation type.

rotation_variable *(enumerated) [=time]*
Scaling of angular velocity to obtain the rotation angle. Used with rotation type.

time	Time.
multiplier_function	The multiplier function for the temporal function of the rotation angle. Requires *rotation_variable_multiplier_function*.
multiplier_function_on_time	The multiplier function for the temporal function of the angular velocity for computing the rotation angle by time integration of angular velocity. Requires *rotation_variable_multiplier_function*.

rotation_variable_multiplier_function *(string) [=none]*
User-given name of the multiplier function for computing the rotation angle. If none, the rotation is zero. Used with multiplier_function rotation variable.

curve_fit_values *or* *curve_values* *(array) [={0,0,0,0,1,0,0}]*
Data for the motion of two points as a function of an independent variable. Used with piecewise_linear or cubic_spline type. Requires *curve_fit_variable*.

curve_fit_variable *or* *curve_var* *(enumerated) [=time]*
Independent variable for the mesh motion data. Used with piecewise_linear or cubic_spline type.

time	Time.
multiplier_function	Multiplier function. Requires *curve_fit_variable_multiplier_function*.

curve_fit_variable_multiplier_function *or* *curve_name* *(string) [=none]*
User-given name of the multiplier function that represents the independent variable of the curve fit for computing the mesh motion. If none, the motion is zero. Used with *multiplier_function* curve fit variable.

user_function *or* *user* *(string) [no default]*
Name of the user-defined function. Used with user_function type.

user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function type.

user_strings (list) [={}]
Array of strings to be passed to the user-defined function. Used with user_function type.

interpolated_motion_surfaces (list) [={}]
List of surfaces whose motions are used to scale the displacements of mesh nodes in between. Each surface is defined by one instance of the INTERPOLATED_MOTION_SURFACE command.

rigid_body_x_displacement (enumerated) [=active]
Constraint on x displacement of rigid body. Used with rigid_body type.

active	No constraint.
Zero	Zero x displacement.

rigid_body_y_displacement (enumerated) [=active]
Constraint on y displacement of rigid body. Used with rigid_body type.

active	No constraint.
Zero	Zero y displacement.

rigid_body_z_displacement (enumerated) [=active]
Constraint on z displacement of rigid body. Used with rigid_body type.

active	No constraint.
Zero	Zero z displacement.

rigid_body_x_rotation (enumerated) [=active]
Constraint on rotation around x axis of rigid body. Used with rigid_body type.

active	No constraint.
Zero	Zero x axis rotation.

rigid_body_y_rotation (enumerated) [=active]
Constraint on rotation around y axis of rigid body. Used with rigid_body type.

active	No constraint.
Zero	Zero y axis rotation.

rigid_body_z_rotation (enumerated) [=active]
Constraint on rotation around z axis of rigid body. Used with rigid_body type.

active	No constraint.
Zero	Zero z axis rotation.

rigid_body_rotation_only (boolean) [=off]
Activated when the rigid body motion is fixed-point rotation and type is set to rigid_body_dynamic. Used with *rigid_body_center*, *rigid_body_external_force* and *rigid_body_moment_arm*.

rigid_body_center (array) [={0,0,0}]
Used with type = rigid_body_dynamic and specified in the global xyz coordinate system, *rigid_body_center* denotes the origin of the local xyz coordinate system. When the rigid body is doing general three-dimensional motion with both translation of center of mass and rotation around center of mass, *rigid_body_center* should be the coordinates of center of mass at time=0. When *rigid_body_rotation_only* = on (the rigid body is doing fixed-point rotation) *rigid_body_center* is regarded as the fixed rotation center, not necessarily center of mass. Notice that the effect of gravity or other external force, whenever needed, can be taken into account by using *rigid_body_external force* and *rigid_body_moment_arm*.

rigid_body_moment_arm (array) [={0,0,0}]
Used when *rigid_body_rotation_only* = on. The coordinate, specified in the local xyz coordinate system, of the location of the concentrated external force, specified by *rigid_body_external_force*. In the case of gravity, it is the center of mass specified in the local coordinate system.

rigid_body_direction (array) [={1,0,0;0,1,0;0,0,1}]
Direction of local xyz coordinate system, specified with respect to the global xyz coordinate system. First row is the local x direction. Used with rigid_body type.

rigid_body_mass (real) >0 [=1]
Mass of the rigid body. Used with rigid_body type.

rigid_body_dyadic (array) [={1,1,1,0,0,0}]
Symmetric dyadic (moment of inertia] matrix for the rotational rigid body equation. Used with rigid_body type.

rigid_body_stiffness (array) [={0,0,0,0,0,0}]
Symmetric stiffness matrix for the translational rigid body equation. Used with rigid_body type.

rigid_body_damping (array) [={0,0,0,0,0,0}]
Symmetric damping matrix for the translational rigid body equation. Used with rigid_body type.

rigid_body_rotational_stiffness (array) [={0,0,0,0,0,0}]
Symmetric rotational stiffness matrix for the rotational rigid body equation. If this parameter is non-zero, rotations are limited to π radians. Used with rigid_body type.

rigid_body_rotational_damping (array) [={0,0,0,0,0,0}]
Symmetric rotational damping matrix for the rotational rigid body equation. Used with rigid_body_dynamic type.

rigid_body_external_force_type (enumerated) [constant]
Type of rigid body external force. Used with rigid_body_dynamic type.

constant	External force vector components are constant values.
piecewise_linear or linear	External force vector components are linear functions of time.
cubic_spline or spline	External force vector components are cubic functions of time.
user_function or user	External force vector components are provided by user function.

rigid_body_external_force (array) [={0,0,0}]
External force vector on rigid body, specified in the global coordinate system. Used with *rigid_body_external_force_type* = constant.

rigid_body_external_force_curve_fit_values (array) [={0,0,0,0}]
A four column array of independent-variable/force data values. Used with *rigid_body_external_force_type* = piecewise_linear or cubic_spline.

rigid_body_external_force_curve_fit_variable (enumerated) [=time]
Independent-variable of the curve fit. Used with *rigid_body_external_force_type* = piecewise_linear or cubic_spline.

time	Physical time.

rigid_body_external_force_user_function (string) [no default]
Name of the user-defined function. Used with *rigid_body_external_force_type* = user_function.

rigid_body_external_force_user_values (array)[={}]
Array of values to be passed to the user-defined function. *rigid_body_external_force_type* = user_function.

rigid_body_external_force_user_strings (list)[=none]
Array of strings to be passed to the user-defined function. *rigid_body_external_force_type* = user_function.

rigid_body_external_force_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the external force. If none, no scaling is performed. Used with *rigid_body_dynamic* type.

rigid_body_external_moment_type (enumerated) [constant]
Type of rigid body external moment. Used with *rigid_body_dynamic* type.

constant	External moment vector components are constant values.
piecewise_linear or linear	External moment vector components are linear functions of time.
cubic_spline or spline	External moment vector components are cubic functions of time.
user_function or user	External moment vector components are provided by user function.

rigid_body_external_moment (array) [={0,0,0}]
External moment vector on rigid body. Used with *rigid_body_external_moment_type* = constant.

rigid_body_external_moment_curve_fit_values (array) [={0,0,0,0}]
A four column (array) of independent-variable/moment data values. Used with *rigid_body_external_moment_type* = piecewise_linear or cubic_spline.

rigid_body_external_moment_curve_fit_variable *(enumerated) [=time]*
Independent-variable of the curve fit. Used with *rigid_body_external_moment_type* = piecewise_linear or cubic_spline.

time Physical time.

rigid_body_external_moment_user_function *(string) [no default]*
Name of the user-defined function. Used with *rigid_body_external_moment_type* = user_function.

rigid_body_external_moment_user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with *rigid_body_external_moment_type* = user_function.

rigid_body_external_moment_user_strings *(list) [=none]*
Array of strings to be passed to the user-defined function. Used with *rigid_body_external_moment_type* = user_function.

rigid_body_external_moment_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the external moment. If none, no scaling is performed. Used with rigid_body_dynamic type.

rigid_body_initial_displacement *(array) [={0,0,0}]*
Initial displacement vector of the rigid body. Used with rigid_body_dynamic type.

rigid_body_initial_velocity *(array) [={0,0,0}]*
Initial velocity vector of the rigid body. Used with rigid_body_dynamic type.

rigid_body_initial_rotation *(array) [={0,0,0}]*
Initial rotation vector of the rigid body. Used with rigid_body_dynamic type.

rigid_body_initial_angular_velocity *(array) [={0,0,0}]*
Initial angular velocity vector of the rigid body. Used with rigid_body_dynamic type.

rigid_body_initial_force *(array) [={0,0,0}]*
Initial internal force vector on the rigid body. Used with rigid_body_dynamic type.

rigid_body_internal_force_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the internal force. If none, no scaling is performed. Used with rigid_body_dynamic type.

rigid_body_initial_moment *(array) [={0,0,0}]*
Initial internal moment vector on the rigid body. Used with rigid_body_dynamic type.

rigid_body_internal_moment_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the internal moment. If none, no scaling is performed. Used with rigid_body_dynamic type.

rigid_body_filter *(enumerated) [=none]*
Type of filter to smooth the internal forces and moments. Used with rigid_body_dynamic type.

none No filter.

iir	Infinite impulse response digital filter. Requires *rigid_body_iir_input_coefficients* and *rigid_body_iir_output_coefficients*.

rigid_body_iir_input_coefficients (array) [={1}]
Infinite impulse response digital filter coefficients applied to computed values. Used with iir rigid body filter and rigid_body_dynamic type.

rigid_body_iir_output_coefficients (array) [={}]
Infinite impulse response digital filter coefficients applied to previously filtered values. Used with iir rigid body filter and rigid_body_dynamic type.

rigid_body_surface_outputs (list) [={}]
Array of SURFACE_OUTPUT command qualifiers used to calculate the force and moment on the rigid body. Used with rigid_body_dynamic type.

Description

This command is used to simplify the specification of boundary conditions on mesh displacement, and in some cases to eliminate the solution of a hyperelastic equation normally needed to move the mesh nodes. In addition it can be used to simulate the dynamic motion of a rigid body.

When MESH_MOTION is referenced in the ELEMENT_SET, NODAL_BOUNDARY_CONDITION, or SIMPLE_BOUNDARY_CONDITION commands, all the corresponding nodes are constrained to move as specified in the MESH_MOTION command. In the case where the motion of all the nodes is known a priori, then the solution of a hyperelastic equation for the mesh motion can be skipped by specifying "mesh=specified" in the EQUATION command. In this case the motion of the nodes is determined at the beginning of each time step. A MESH_MOTION command has no effect unless it is referenced by one of the above commands. MESH_MOTION is similar in its usage and behavior to REFERENCE_FRAME, with the main difference that it modifies the mesh displacement (motion) instead of the velocity field. For example,

```
MESH_MOTION( "rotating fan" ) {
   type                                 = rotation
   rotation_variable                    = time
   rotation_center                      = { 0, 0, 0 }
   angular_velocity                     = { 0, 3, 0 }
}
ELEMENT_SET( "rotating fan" ) {
   mesh_motion                          = "rotating fan"
   ...
}
EQUATION {
   mesh                                 = specified
   ...
}
```

In this example, the frame of reference does not rotate, as opposed to the first example in REFERENCE_FRAME, but instead all the mesh nodes in the element set rotate around the y axis at the rate of three radians per unit of time with the axis of rotation centered at the point (zero, zero, zero) in the global xyz coordinate system. The right hand rule is used to define the direction of rotation of the nodes. For this example, the location of each node in the element set is given by

$$x = x_0 \cos 3t - z_0 \sin 3t$$
$$y = y_0 \qquad (22)$$
$$z = z_0 \cos 3t + x_0 \sin 3t$$

where (X_0, Y_0, Z_0) is the initial location of the node and t is given by *rotation_variable*, in this case time. The above will be useful in conjunction with an additional fixed element set and a sliding mesh between the two element sets. Instead of a constant angular velocity, one that approximates the motion of a washing machine agitator can be specified through the use of a multiplier function as follows:

```
MESH_MOTION( "Washing Machine" ) {
    type                                        = rotation
    rotation_variable                           = multiplier_function
    rotation_variable_multiplier_function       = "Washing Machine"
    rotation_center                             = { 0, 0, 0 }
    angular_velocity                            = { 0, 1, 0 }
}
MULTIPLIER_FUNCTION( "Washing Machine" ) {
    type                                        = piecewise_linear
    curve_fit_values                            = {    0, 0 ;
                                                     0.5, -10*PI/180 ;
                                                     1.0, 0 ;
                                                     1.5, +10*PI/180 ;
                                                     2.0, 0 ;
                                                     }
   curve_fit_variable                           = cyclic_time
   curve_fit_variable_cyclic_period             = 2
}
```

where the variable t in the above equations is replaced with the multiplier function, with the result that the agitator swings through an angle of ±30 degrees (10 degrees from the multiplier function itself times three from the second component of *angular_velocity*) with a period of two. Equivalently, if the unit vector for *angular_velocity* is used ({0,1,0}) in MESH_MOTION, the rotation angle is given directly by the multiplier function:

```
MULTIPLIER_FUNCTION( "Washing Machine" ) {
    type                                = piecewise_linear
    curve_fit_values                    = {  0,   0 ;
                                           0.5, -30*PI/180 ;
                                           1.0, 0 ;
                                           1.5, +30*PI/180 ;
                                           2.0, 0 ;
                                           }
curve_fit_variable                      = cyclic_time
curve_fit_variable_cyclic_period        = 2
}
```

Another simple use of this command is to specify a linear translation of a set of mesh nodes. For example, the motion of an element set surrounding a train may be given by

```
MESH_MOTION( "moving train" ) {
   type                                         = translation
   translation_variable                         = time
   translation_variable_multiplier_function     = "none"
   translation_velocity                         = { 1, 0, 0 }
}
```

```
ELEMENT_SET( "train" ) {
   mesh_motion                                  = "moving train"
   ...
}
```

Here all the nodes of the element set move at a constant velocity of one in the positive x direction. For this case, the location of each node in the element set is given by

$$x = x_0 + tv \tag{23}$$

where f_n is the initial location of the node, v is given by *translation_velocity*, and t is time. Analogous to the rotation case, a multiplier function can be defined with *translation_variable*=multiplier_function to modify the scaling of velocity. Then the location of each node is given by

$$x = x_0 + fv \tag{24}$$

where f is given by *translation_variable_multiplier_function*.

A mesh motion type of none has the same effect as not referencing the MESH_MOTION command at all. This parameter is useful, for example, when you want to define a mesh motion but not activate it until a restart. A mesh motion type of zero is equivalent to specifying a NODAL_BOUNDARY_CONDITION command with type zero for each of the three components of mesh displacement.

More complicated mesh motions can be defined with types piecewise_linear and cubic_spline. Both fit a curve to the data given by *curve_fit_values*, which is a seven column array corresponding to time and the location of two points: t, x1, y1, z1, x2, y2, z2, which are defined in the global xyz coordinate system. These two points are attached to the rigid body but don't necessarily have to be on or in the body. The reference points are defined at t=0. At all other times both points are moved relative to the reference points. Since the body is rigid the distance between the two points should remain a constant. However, in order to simplify usage, AcuSolve will internally rescale the distance. For example,

```
MESH_MOTION( "general motion" ) {
   type                                         = piecewise_linear
   curve_fit_values                             = { 0, 0, 0, 0, 1, 0, 0 ; 1, 1, 0,
 0, 0, 1, 0 ; }
   curve_fit_variable                           = time
   curve_fit_variable_multiplier_function       = "none"
}
```

This command will create a combined translation and rotation motion whereby the point originally at the origin is moved in the x direction and a point originally one unit along the x axis is moved along the y axis. As with the translation and rotation types, the time variable can be replaced with a multiplier function by specifying multiplier_function for *curve_fit_variable* and the name of a multiplier function for *curve_fit_variable_multiplier_function*.

A mesh motion type of user_function may be used to model more complex behaviors; see the *AcuSolve User- Defined Functions Manual* for a detailed description of user-defined functions. Unlike the curve fit types above, the user-defined function returns a 3x4 matrix that describes the combined translation and rotation motions. This matrix is defined by:

$$\begin{bmatrix} x \\ y \\ z \end{bmatrix} = \begin{bmatrix} r_{11} & r_{12} & r_{13} & t_1 \\ r_{21} & r_{22} & r_{23} & t_2 \\ r_{31} & r_{32} & r_{33} & t_3 \end{bmatrix} \begin{bmatrix} x_0 \\ y_0 \\ z_0 \\ 1 \end{bmatrix} \tag{25}$$

where (X_0, Y_0, Z_0), is the initial position of a node, (x, y, z) is the mesh displacement of that node, and $R = \{r_{ij}\}$ must be a valid rotation matrix.

For example, consider the case where an element set is simultaneously rotated at a constant rate around the z-axis and translated sinusoidally up and down the same axis. The input commands may be given by:

```
MESH_MOTION( "rotation/translation mesh motion" ) {
   type                      = user_function
   user_function             = "usrRotTransMmo"
   user_values               = { 0,            # x-center of rotation
                                 0,            # y-center of rotation
                                 0,            # z-center of rotation
                                 2,            # rotation frequency
                                 1,            # translation frequency
                                 4}            # translation amplitude
}
ELEMENT_SET( "fluid" ) {
   ...
   mesh_motion               = "rotation/translation mesh motion"
}
```

where the user-defined function "usrRotTransMmo" may be implemented as:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE(usrRotTransMmo) ;          /* function prototype
 */
Void usrRotTransMmo(
   UdfHd udfHd,                          /* Opaque handle for accessing data
 */
   Real* outVec,                         /* Output vector
 */
   Integer nItems,                       /* = 1
 */
   Integer vecDim)                        /* = 12 (for twelve components)
 */
{
   Integer i ;                           /* a running index
 */
   Integer j ;                           /* a running index
 */
   Integer k ;                           /* a running index
 */
   Real angle ;                          /* rot/trans angle
 */
   Real ca ;                             /* cosine of angle
 */
   Real rotMtx[4][4] ;                   /* pure rotation matrix
 */
   Real rotOmega ;                       /* rotation frequency
 */
   Real sa ;                             /* sine of angle
 */
```

```
   Real time ;                          /* current time
 */
   Real trnMtx[4][4] ;                  /* pure translation matrix
 */
   Real trnOmega ;                      /* translation frequency
 */
   Real trnAmp ;                        /* translation amplitude
 */
   Real org[3] ;                        /* origin of rotation
 */
   Real* usrVals ;                      /* user supplied values
 */
   udfCheckNumUsrVals( udfHd, 6 ) ;     /* check for error
 */
   usrVals = udfGetUsrVals( udfHd ) ;   /* get the user vals
 */
   org[0]   = usrVals[0] ;              /* get x-origin of rotation
 */
   org[1]   = usrVals[1] ;              /* get y-origin of rotation
 */
   org[2]   = usrVals[2] ;              /* get z-origin of rotation
 */
   rotOmega = usrVals[3] ;              /* get rotation frequency
 */
   trnOmega = usrVals[4] ;              /* get translation frequency
 */
   trnAmp   = usrVals[5] ;              /* get translation amplitude
 */
   time     = udfGetTime( udfHd ) ;     /* get time
 */

/* Initialize the 4x4 transformation matrices to the identity matrix */
   for ( i = 0 ; i < 4 ; i++ ) {
      for ( j = 0 ; j < 4 ; j++ ) {
         rotMtx[i][j] = 0 ;
         trnMtx[i][j] = 0 ;
      }
      rotMtx[i][i] = 1 ;
      trnMtx[i][i] = 1 ;
   }

/* Construct the rotation transformation */
   angle                                = time * rotOmega * 2 *3.1415926535897931E
+00 ;
   ca                                   = cos( angle ) ;
   sa                                   = sin( angle ) ;
   rotMtx[0][0]                         = +ca ;
   rotMtx[0][1]                         = -sa ;
   rotMtx[1][0]                         = +sa ;
   rotMtx[1][1]                         = +ca ;
   for ( i = 0 ; i < 3 ; i++ ) {
      rotMtx[i][3]                      = org[i]
                                        - rotMtx[i][0] * org[0]
                                        - rotMtx[i][1] * org[1]
                                        - rotMtx[i][2] * org[2] ;
}

/* Construct the translation transformation */
   angle                                = time * trnOmega * 2 * 3.1415926535897931E
+00 ;
   sa                                   = sin( angle ) ;
   trnMtx[2][3]                         = trnAmp * sa ;
```

```
/* Build the output vector (the fourth row is not needed) */
   k                                    = 0 ;
   for ( j = 0 ; j < 4 ; j++ ) {
      for ( i = 0 ; i < 3 ; i++ ) {
         outVec[k]                      = rotMtx[i][0] * trnMtx[0][j]
                                        + rotMtx[i][1] * trnMtx[1][j]
                                        + rotMtx[i][2] * trnMtx[2][j]
                                        + rotMtx[i][3] * trnMtx[3][j] ;
         k++ ;
      }
   }
} /* end of usrRotTransMmo() */
```

The dimension of the returned transformation matrix, outVec, is 12.

Mesh motion based on the dynamics of a rigid body may be defined by the rigid_body_dynamic type.

Since the body is rigid, a detailed geometrical description of it is not necessary. Instead, only the modeled properties appearing as the coefficients in the dynamic translation and rotation equations need to be specified. The rigid body is driven by the sum of two kinds of forces and moments. "Internal" refers to the integrated fluid traction and moment, and "external" to the force and moment specified by you in this command. A local coordinate system is used to simplify the definition of the rigid body model and the solution of the equations of motion. The translational and rotational equations of motion are:

$$\begin{aligned} ma + Cv + Kx &= F_I + F_E \\ M\alpha + D\omega + L\Omega &= R_I + R_E \end{aligned} \qquad (26)$$

where x, v, and a are the translational displacement, velocity, and acceleration of the body; and Ω, ω and α are the rotational equivalents. All quantities in these equations are defined in the local coordinate system. The mass of the body is m, given by *rigid_body_mass*; C is the translational damping matrix, given by rigid_body_damping; K is the translational stiffness matrix, given by rigid_body_stiffness; M is the dyadic (moment of inertia) matrix, given by rigid_body_dyadic; D is the rotational damping matrix, given by *rigid_body_rotational_damping*; and L is the rotational stiffness matrix, given by *rigid_body_rotational_stiffness*. Each of the five coefficient matrices must be symmetric positive semidefinite. The six components are given in the order xx, yy, zz, xy, yz, zx. The external force and moment are **F**E and **R**E , given by *rigid_body_external_force* and *rigid_body_external_moment*, respectively in the local coordinate system. The internal force and moment are **F**I and **R**I, which are defined by the integrated force and moment on the surfaces given by *rigid_body_surface_outputs*. These last two quantities are transformed from the global to local coordinate system as follows:

$$\begin{aligned} \boldsymbol{F}_I &= \boldsymbol{T}\boldsymbol{F}_{I,g} \\ \boldsymbol{R}_I &= \boldsymbol{T}(\boldsymbol{R}_{I,g} - (\boldsymbol{x}_c + \boldsymbol{T}^t\boldsymbol{x}) \times \boldsymbol{F}_{I,g}) \end{aligned} \qquad (27)$$

where **T** is the transformation matrix from the global to local coordinate system, given by *rigid_body_direction*; $F_{I,\,g}$ and $R_{I,\,g}$ are the actual surface output force and moment in global coordinates; x_c is both the center of mass of the rigid body and the origin of the local xyz coordinate system, given by *rigid_body_center*; and a superscript t is the transpose. AcuSolve internally transforms other vector quantities in an analogous way. However, the transformation of the coordinate systems themselves is extremely complex in general and is therefore not given here.

The external force and moment and the internal force and moment may be scaled by *rigid_body_external_force_multiplier_function*, *rigid_body_external_moment_multiplier_function*, *rigid_body_internal_force_multiplier_function* and *rigid_body_internal_moment_multiplier_function*, respectively. The usual use for these is to ease into a dynamic rigid body simulation by implementing a multiplier function that starts at zero and ramps up to one.

The parameters *rigid_body_x_displacement*, *rigid_body_y_displacement*, *rigid_body_z_displacement*, *rigid_body_x_rotation*, *rigid_body_y_rotation*, and *rigid_body_z_rotation* can each be used to constrain the corresponding rigid body degree of freedom by setting it to zero. Setting it to active causes its equation of motion to be solved without constraint. All these parameters are defined in the local coordinate system.

Since each of the equations of motion is a second-order differential equation, each requires two initial conditions. These are given by *rigid_body_initial_displacement*, *rigid_body_initial_velocity*, *rigid_body_initial_rotation*, and *rigid_body_initial_angular_velocity*. All are three component vectors corresponding to the x, y, and z components of the initial conditions in the global coordinate system. The particular numerical method used to solve the equations uses the surface output data from the previous time step. This data is not available during the first time step so initial conditions on the internal force and moment are also required. These are given by *rigid_body_initial_force* and *rigid_body_initial_moment*. Both are also defined in the global coordinate system. All initial condition data may be reset at restart.

The accuracy and stability of the solution of the rigid-body equations depend on how those equations are coupled to the fluid equations. Two strategies are supported:

- Conventional Sequential Staggered (CSS): This uses a single-pass of displacements/rotations and forces/moments at each time step. The result is essentially an explicit scheme since the data comes from the previous time step. CSS is used when there is exactly one nonlinear stagger per time step (*max_stagger_iterations*=1 in the `STAGGER` command).
- Multi-Iterative Coupling (MIC): This strategy corrects the interfacial forces via a multi-pass transfer of mesh displacements/rotations and fluid forces/moments. From numerical experiments, robustness and efficiency of the scheme are obtained for a wide range of physical parameters. The scheme requires at least two nonlinear stagger iterations per time step. The interfacial force/moment and displacement/rotation residuals can be obtained by printing the `*.Log` file (generated by AcuRun) with a verbose level of two.

The computed internal force and moment may be smoothed in time by applying a digital filter before applying them to the rigid body. An iir rigid body filter gives an infinite impulse response filter, which is defined by:

(28)

$$f_n = \sum_{i=1}^{N_a} a_i \tilde{f}_{n+1-i} + \sum_{i=1}^{N_b} b_i f_{n-i}$$

with the constraint that

$$\sum_{i=1}^{N_a} a_i + \sum_{i=1}^{N_b} b_i = 1 \qquad (29)$$

where $a_i, i = 1, \ldots, N_a$ is given by *rigid_body_iir_input_coefficients*; $b_i, i = 1, \ldots, N_b$ is given by *rigid_body_iir_output_coefficients*; f_n is a component of the fluid force or moment applied to the rigid body at time step n ; and f_n is the value of this component before filtering at time step n. N_a and N_b are determined by the number of elements of the corresponding arrays. If the constraint is not satisfied, AcuSolve automatically scales the coefficients appropriately. For example,

```
MESH_MOTION( "dynamic rigid body with digital filter" ) {
   type                                = rigid_body_dynamic
   ...
   rigid_body_filter                   = iir
   rigid_body_iir_input_coefficients   = { 0.8, 0.8 }
   rigid_body_iir_output_coefficients  = { -0.6 }
}
```

The qualifiers of the SURFACE_OUPUT commands used to calculate the internal force and moment on the rigid body are listed in *rigid_body_surface_outputs*. Since for the CSS solution strategy the internal force and moment are updated only when the corresponding SURFACE_OUPUT command is evaluated, it is highly recommended that *integrated_output_frequency* be set to one in each of these commands for best accuracy. For the MIC strategy this frequency parameter is ignored; the internal forces and moments are updated as needed.

For example, an oil platform floating in the ocean subjected to ocean waves and wind may be modeled by:

```
MESH_MOTION( "rigid platform" ) {
   type                                          = rigid_body_dynamic
   rigid_body_x_displacement                     = active
   rigid_body_y_displacement                     = active
   rigid_body_z_displacement                     = zero
   rigid_body_x_rotation                         = zero
   rigid_body_y_rotation                         = zero
   rigid_body_z_rotation                         = zero
   rigid_body_center                             = { 0, 0, 0 }
   rigid_body_direction                          = { 1, 0, 0 ;
                                                     0, 1, 0 ;
                                                     0, 0, 1 ; }
   rigid_body_mass                               = 1.2E+08
   rigid_body_stiffness                          = {6.4E+05,6.4E+05,0,0,0,0}
   rigid_body_damping                            = { 0, 0, 0, 0, 0, 0 }
   rigid_body_dyadic                             = { 0, 0, 0, 0, 0, 0 }
   rigid_body_rotational_stiffness               = { 0, 0, 0, 0, 0, 0 }
   rigid_body_rotational_damping                 = { 0, 0, 0, 0, 0, 0 }
   rigid_body_external_force                     = { 0, 3.E+07, 0 }
   rigid_body_external_force_multiplier_function = none
   rigid_body_external_moment                    = { 0, 0, 0 }
   rigid_body_external_moment_multiplier_function = none
   rigid_body_initial_displacement               = { 0, 0, 0 }
   rigid_body_initial_velocity                   = { 0, 0, 0 }
   rigid_body_initial_rotation                   = { 0, 0, 0 }
   rigid_body_initial_angular_velocity           = { 0, 0, 0 }
```

```
   rigid_body_initial_force                          = { 0, 0, 0 }
   rigid_body_internal_force_multiplier_function     = none
   rigid_body_initial_moment                         = { 0, 0, 0 }
   rigid_body_internal_moment_multiplier_function = none
   rigid_body_filter                                 = none
   rigid_body_surface_outputs                        = { "platform" }
}
NODAL_BOUNDARY_CONDITION( "all x-disp" ) {
   nodes                                             = Read( "all.nbc" )
   variable                                          = mesh_x_displacement
   type                                              = piecewise_linear
   curve_fit_variable                                = x_reference_coordinate
   curve_fit_values                                  = { -175, 0. ;    # side1
                                                         -75, 1. ;    # side1 of
 platform
                                                         +75, 1. ;    # side2 of
 platform
                                                         +175, 0. ; } # side2
   mesh_motion                                       = "rigid platform"
 }
 NODAL_BOUNDARY_CONDITION( "all y-disp" ) {
   nodes                                             = Read( "all.nbc" )
   variable                                          = mesh_y_displacement
   type                                              = piecewise_linear
   curve_fit_variable                                = y_reference_coordinate
   curve_fit_values                                  = { -175, 0. ;    # inflow
                                                         -75, 1. ;    # head of
 platform
                                                         +75, 1. ;    # tail of
 platform
                                                        +600, 0. ; }  # outflow
   mesh_motion                                       = "rigid platform"
}
EQUATION {
   mesh                                              = ale
   ...
}
   SURFACE_OUTPUT( "platform" ) {
   surfaces                                          = Read( "platform.srf" )
   shape                                             = four_node_quad
   element_set                                       = "sea"
   integrated_output_frequency                       = 1
}
```

The platform is allowed to move only horizontally and without any rotation. The equations and parameters corresponding to the other degrees of freedom are ignored. The wires preventing the platform from floating away are modeled by springs; the spring constants appear on the diagonal of the stiffness matrix. The atmosphere is not modeled directly but its effect on the platform is roughly modeled by an external force in the y direction. Since waves are modeled in the "sea" element set, commands not shown here, the z mesh displacement is nonzero. It is controlled by the free surface, other boundary conditions, and an ALE equation. The x and y components of the mesh displacement are controlled by the `NODAL_BOUNDARY_CONDITION` commands above. The effect of the curve fits is to smoothly vary the horizontal mesh displacements from zero at the outer boundaries to the `MESH_MOTION` values near the platform without having to solve an ALE equation for them.

In the above example, if the z mesh displacement could also be constrained, that is, there was no free surface, then there would be no mesh displacement degrees of freedom left to solve for in the ALE equation. In general, if all three components of the mesh displacement on all nodes are specified

through the appropriate commands, then *mesh*=specified can be used in the EQUATION command and the solution of an ALE equation is eliminated (no stagger is needed for it). The appropriate commands are NODAL_BOUNDARY_CONDITION and SIMPLE_BOUNDARY_CONDITION, either with or without *mesh_motion*; and ELEMENT_SET using *mesh_motion*. In the latter case, all nodes in the element set are moved according to the specified MESH_MOTION command.

A mesh motion of *type*=interpolated is used to scale the displacement of each node in one particular set, such as an element set, based on several pre-defined interpolated motion surfaces. Using this approach, the mesh can be deformed without solving the ALE equations even for non-trivial motions. The interpolated motion can be applied to the entire domain or used only in a portion of the domain while. In the following example, there is only one element set named "Fluid", which refers to a mesh motion named "interpolated displacement" with *type*=interpolated_motion. The parameter *interpolated_motion_surfaces* in "interpolated displacement" is a list referring to two surface sets defined by the INTERPOLATED_MOTION_SURFACE command.

```
EQUATION {
...
    mesh                                = specified
...
}
ELEMENT_SET( "Fluid" ) {
…
    mesh_motion                         = "interpolated displacement"
…
}
MESH_MOTION( "interpolated displacement" ) {
    type                                = interpolated_motion
    interpolated_motion_surfaces        = {"top","bottom"}
}
INTERPOLATED_MOTION_SURFACE( "top" ) {
   surfaces                             = Read( "MESH.DIR/top.ebc" )
   shape                                = three_node_triangle
   element_set                          = "Fluid"
   type                                 = faceted
}
INTERPOLATED_MOTION_SURFACE( "bottom" ) {
    surfaces                            = Read( "MESH.DIR/bottom.ebc" )
    shape                               = three_node_triangle
    element_set                         = "Fluid"
    type                                = faceted
}
```

For each node in the "Fluid" element set, AcuSolve computes the distance to the *interpolated_motion_surfaces* and scales its displacement based on the motions of the surfaces. This type of mesh motion is used in place of the ALE approach, or as a complement to it, to achieve mesh deformation in applications such as Fluid Structure Interaction. This is a rather simple method for deforming the mesh that has significant run time savings over using the ALE approach. Because of its simplicity, however, the method is limited in the extent of deformation that it can handle. More importantly, the motion of the mesh on every *interpolated_motion_surfaces* must be fully specified in all three directions. Consequently, free surfaces and guide surfaces cannot be used as interpolated motion surfaces.

In the following example, the solid sphere is moving in one direction within a static fluid domain, and you would like to adopt interpolated mesh motion in that domain to control the mesh deformation. First you need to choose the right *interpolated_motion_surfaces* or say "driving surfaces". Not every

boundary has to serve as a driving surface. Instead, here you choose the sphere, the left, and the right surfaces, excluding the top and bottom surfaces. On these driving surfaces, for every node, you must explicitly specify the mesh displacements of all three directions. In other words, AcuSolve assumes that you want full control of the motion of *interpolated_motion_surfaces*.

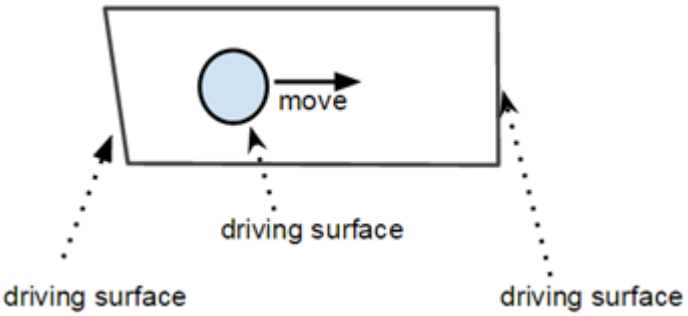


Figure 2:

In this example, nodes on the top or bottom surface are dominated by the interpolated mesh motion. If you want to constrain them within the planar surface where they are located initially, you have to put a slip-type simple boundary condition for mesh displacement or a zero nodal boundary condition for the component that is normal to the planar surface.

Another scenario is that both top and bottom surfaces are included in the driving surfaces as well. In that case you need to specify the motion of these two surfaces. If any component of the mesh displacement is missing, AcuSolve will just keep that component as its initial value. For example, you may set slip for the mesh displacement on the top surface, but you will not see any sliding motion of the nodes because interpolated mesh motion does nothing on nodes on *interpolated_motion_surfaces* or say driving surfaces. For the same reason, if you do not specify any mesh boundary condition for the top or bottom surface nodes there will just stay where they are initially throughout the simulation.

Below is an example of a pendulum of 30 kg. Notice that *rigid_body_center* here means the pivot, located at (2.0, 3.0, 4.0). The location of center of mass is constant when evaluated in the local coordinate system (glued to the pendulum), and it is (0, 0, -1.42).

```
MESH_MOTION
    type                              = rigid_body_dynamic
    rigid_body_x_rotation             = active
    rigid_body_y_rotation             = zero
    rigid_body_z_rotation             = zero
    rigid_body_rotation_only          = on
    rigid_body_moment_arm             = {0, 0, -1.42}          #
local coordinates of center of mass
    rigid_body_external_force         = { 0.0, 0.0, -30*9.8; } #
gravity force in global coordinate system
    rigid_body_x_displacement         = zero
    rigid_body_y_displacement         = zero
    rigid_body_z_displacement         = zero
    rigid_body_center                 = { 2.0, 3.0, 4.0; }     #
rotation center
```

MULTIPLIER_FUNCTION

Specifies a scalar multiplier function to scale values of other commands and parameters at run time.

Type

AcuSolve Command

Syntax

MULTIPLIER_FUNCTION ("name") {parameters...}

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of the multiplier function.

constant or const	Fixed value. Requires *constant_value*.
sine_series or sine	Summation of a series of sine functions. Requires *sine_coefficients*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
piecewise_log_linear or log_linear	Piecewise log/linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
design_variable	User-defined name of the design optimization variable.
response_variable	The responses to be used for optimization.
user_function or user	User-defined function. Requires *user_function*, *user_values* and *user_strings*.
user_global_data or ugd	User-defined global variable. Requires *user_global_data*.
modifiable	Used like constant type, except value may be modified within a user-defined function of another multiplier function.

constant_value or *value (real) (=1)*

The constant value of the multiplier function. Used with constant and modifiable types.

sine_coefficients (array) [={1,0,0}]

A three-column array of amplitude/frequency/phase data values. Used with sine_series type.

curve_fit_values **or** *curve_values* *(array) [={0,1}]*
A two-column array of independent-variable/multiplier-function data values. Used with *piecewise_linear*, *cubic_spline*, and *piecewise_log_linear* types.

curve_fit_variable **or** *curve_var* *(enumerated) [=time]*
Independent variable of the curve fit. Used with *piecewise_linear*, *cubic_spline*, and *piecewise_log_linear* types.

time	Current run time.
cyclic_time	Current run time, except cycled with period *curve_fit_variable_cyclic_period*.
time_step or step	Current time step.
time_increment or dt	Current time increment.
	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
multiplier_function	Multiplier function. Requires *curve_fit_variable_multiplier_function*.
user_global_data	User-defined global variable. Requires *curve_fit_variable_user_global_data*.

curve_fit_variable_multiplier_function *(string) [=none]*
Name of the multiplier function. Used with multiplier_function curve fit variable.

curve_fit_variable_user_global_data *(string) [=none]*
Name of the user global data. Used with user_global_data curve fit variable.

curve_fit_variable_cyclic_period *(real) >=0 [=1]*
The period of cyclic time. Used with cyclic_time curve fit variable.

design_variable *(list) [={}]*
Design_variable list to be used with OPTIMIZATION.

response_variable *(string) [=""]*
User-defined name of the response variable that is computed from the simulation.

user_function **or** *user* *(string) [no default]*
Name of the user-defined function. Used with user_function type.

user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function type.

user_strings *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with user_function type.

user_history_variables *(list) [={}]*
Array of history values passed to the user-defined function. Used with user_function type. For each entry in this array, AcuSolve allocates storage such that you can store and access values across time steps within a UDF.

user_global_data or *ugd* *(string) [no default]*
Name of the user global data variable. Used with user_function type or user_global_data curve fit variable.

evaluation *(enumerated) [=once_per_solution_update]*
Frequency of evaluating the multiplier function.

once_per_solution_update or often	Every solution update.
once_per_time_step or step	Every time step.

filter *(enumerated) [=none]*
Type of filter to smooth the multiplier function. May not be used with modifiable type.

none	No filter.
relaxation or damp	Smooth the value of the multiplier function. Requires *filter_relaxation_factor*.
iir	Infinite impulse response digital filter. Requires *iir_input_coefficients* and *iir_output_coefficients*.
mic	Multi-Iterative Coupling.

filter_relaxation_factor *(real) >=0 <=1 [=0.5]*
The value of the filter relaxation factor. Used with relaxation filter.

iir_input_coefficients *(array) [={1}]*
Infinite impulse response digital filter coefficients applied to computed values. Used with iir filter.

iir_output_coefficients *(array) [={}]*
Infinite impulse response digital filter coefficients applied to previously filtered values. Used with iir filter.

dependencies *(list) [={}]*
List of other multiplier functions this one depends on. Required when this multiplier function depends on another indirectly through a user global data.

Description

This command specifies a scalar multiplier function to be used to dynamically scale various problem parameters at run time.

The commands that accommodate multiplier functions are:

- MULTIPLIER_FUNCTION
- SOLAR_RADIATION
- EXTERNAL_CODE
- REFERENCE_FRAME
- MESH_MOTION
- FLEXIBLE_BODY
- TIME_INCREMENT

- GRAVITY
- MASS_HEAT_SOURCE
- VOLUME_HEAT_SOURCE
- MASS_SPECIES_SOURCE
- VOLUME_SPECIES_SOURCE
- VISCOSITY_MODEL
- CONDUCTIVITY_MODEL
- DIFFUSIVITY_MODEL
- POROSITY_MODEL
- SURFACE_TENSION_MODEL
- CONTACT_ANGLE_MODEL
- EMISSIVITY_MODEL
- NODAL_BOUNDARY_CONDITION
- PERIODIC_BOUNDARY_CONDITION
- LINE_SOURCE
- FAN_COMPONENT
- HEAT_EXCHANGER_COMPONENT
- ELEMENT_BOUNDARY_CONDITION
- SIMPLE_BOUNDARY_CONDITION
- SURFACE_INTEGRATED_CONDITION
- RADIATION_SURFACE

For example, to set a gravity vector to zero for a run time up to 10, linearly change the gravity to -9.81 in the y direction by a run time of 20 and subsequently keep it at that level, you may issue the following commands:

```
GRAVITY( "pulse gravity" ) {
 type                           = constant
 gravity                        = { 0, -9.81, 0 }
 multiplier_function            = "pulse function"
   }
MULTIPLIER_FUNCTION( "pulse function" ) {
 type                           = piecewise_linear
 curve_fit_values               = {  0., 0 ;
                                    10., 0 ;
                                    20., 1 ;
                                   100., 1 ; }
 curve_fit_variable             = time
}
```

The physical gravity values are stored in the *gravity* parameter. The MULTIPLIER_FUNCTION command simply scales these values to produce the values used in AcuSolve.

The scaling process described above may be applied to gravity of all types. For example, in

```
GRAVITY( "pulse gravity" ) {
type                          = user_function
user_function                 = "usrSomeGravity"
```

```
multiplier_function}          = "pulse function"
```

the resulting values from the user-defined function "usrSomeGravity" are scaled by the multiplier function in a similar fashion as above.

A *constant* type provides a fixed scaling factor for all times.

A *sine_series* type defines a time-dependent scaling factor of the form:

$$M = \sum_{i=1}^{N} a_i \sin\left(\Omega_i t + \varphi_i\right) \tag{30}$$

where a_i, Ω_i, and φ_i are the amplitude, frequency and phase of the i -th mode of a sine series given by the i -throw of *sine_coefficients*; N is the number of rows in *sine_coefficients*; t is time; and M is the value of the multiplier function. The phase is in radians and the frequency is in radians per time unit, for example, rads/sec. For example, a two-term sine series multiplier function may be given by:

```
MULTIPLIER_FUNCTION( "sine series" ) {
     type                              = sine_series
     sine_coefficients                 = { 1., 10., 1.57 ;2., 20., 3.14 ; }
}
```

A multiplier function of type *piecewise_linear*, *cubic_spline*, or *piecewise_log_linear* computes an interpolated value from the curve fit data of *curve_fit_values* as a function of *curve_fit_variable*. *curve_fit_values* is a two-column array of real values. The first column specifies the independent variable values while the second column specifies the corresponding multiplier function values. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, say `pulse_function.fit`:

```
0.     0
10.    0
20.    1
100.   1
```

and read as:

```
MULTIPLIER_FUNCTION( "pulse function" ) {
     type = piecewise_linear
     curve_fit_values = Read( "pulse_function.fit" )
     curve_fit_variable = time
}
```

For the *piecewise_log_linear* curve fit, the interpolation is performed on the log of the dependent values, which must be strictly positive. This interpolation provides a geometric interpolation. For example,

```
MULTIPLIER_FUNCTION( "growth" ) {
     type = piecewise_log_linear
     curve_fit_values = { 10, 1 ; 13, 1000 }
     curve_fit_variable = time_step
}
```

would yield the following values as a function of time step:

```
1       1
...
10      1
11      10
12      100
13      1000
14      1000
...
```

If *curve_fit_variable* is set to cyclic_time then the current time is looped in the interval, 0, curve_fit_variable_cyclic_time, and values of the independent variable in *curve_fit_values* should fall in this range. Setting *curve_fit_variable_cyclic_time* to zero is the same as an infinite period, that is, regular time is used. If *curve_fit_variable* is set to multiplier_function then another multiplier function, given by *curve_fit_variable_multiplier_function*, is evaluated to produce the independent variable. Multiplier functions may be chained to any depth in this way, but a non-terminating recursive loop is flagged as an error. Similarly, if *curve_fit_variable* is set to user_global_data, then the variable is set to the current value of the user global data given by curve_fit_variable_user_global_data. The following example returns a value of 50 for the multiplier function:

```
MULTIPLIER_FUNCTION( "curve fit UGD" ) {
     type                                = piecewise_linear
     curve_fit_values                    = { 0, 0 ;100, 200 }
     curve_fit_variable                  = user_global_data
     curve_fit_variable_user_global_data = "var_1"
}
USER_GLOBAL_DATA( "var_1" ) {
     value                               = 25
}
```

A multiplier function of type user_function executes a user-defined function to obtain its scaling factor; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

As an example, consider a transient problem with inflow x-velocity oscillating at 10 Hz. Assume that the file `inflow.nbc` contains the list of inflow nodes and the file `inflow.xvel` contains the x-velocity values; see the `NODAL_BOUNDARY_CONDITION` command for details. Then the input commands may be given as:

```
NODAL_BOUNDARY_CONDITION( "inflow x_velocity" ) {
     variable                = x_velocity
     type                    = nodal
     nodes                   = Read( "inflow.nbc" )
     nodal_values            = Read( "inflow.xvel" )
     multiplier_function     = "sine wave at 10 Hz"
   }
MULTIPLIER_FUNCTION( "sine wave at 10 Hz" ) {
     type                    = user_function
     user_function           = "usrSinWave"
     user_values             = { 10. } # frequency
}
```

where the user-defined function "usrSinWave" may be implemented as:

```
#include "acusim.h"
```

```
#include "udf.h"
UDF_PROTOTYPE( usrSinWave ) ;                    /* function prototype
 */
Void usrSinWave (
     UdfHd udfHd,                                /* Opaque handle for accessing data
 */
     Real* outVec,                               /* Output vector
 */
     Integer nItems,                             /* Number of items in outVec
 */
     Integer vecDim                              /* Vector dimension of outVec
 */
   ) {
     Real freq ;                                 /* frequency
 */
     Real time ;                                 /* run time
 */
     Real* usrVals ;                             /* user values
 */
     udfCheckNumUsrVals( udfHd, 1 ) ;            /* check for error
 */
     usrVals = udfGetUsrVals( udfHd ) ;          /* get the user vals
 */
     freq = usrVals[0] ;                         /* get the frequency
 */
     time = udfGetTime( udfHd ) ;                /* get the run time
 */
     *outVec = sin( time * freq * 6.2832 ) ;     /* compute mult.func.*/
} /* end of usrSinWave() */
```

The next example of a user-defined function illustrates the client/server functionality of AcuSolve. There are four associated UDF functions; these are described in the *AcuSolve User-Defined Functions Manual*. Here it is desired that the average pressure over the "inflow" surface be equal to -4, but it is necessary to impose a nodal boundary condition on the x-velocity for stability and control over the velocity profile. This is possible by adjusting the nodal boundary condition multiplier function until the measured average pressure converges to the target pressure. The perl script `usrPres.pl` running on the remote machine "pelican" waits for an input line containing the measured pressure and then returns the multiplier function based on the measured pressure, target pressure, and the previous value of the multiplier function. An initial value is supplied by the `MULTIPLIER_FUNCTION` command, in this case, one. "inflow" refers to a `SURFACE_OUTPUT` command that computes the average pressure over the appropriate surface. The input commands may be given as:

```
NODAL_BOUNDARY_CONDITION( "inflow x-vel" ) {
     variable                                    = x_velocity
     ...
     multiplier_function                         = "inflow pressure control"
}
MULTIPLIER_FUNCTION( "inflow pressure control" ) {
     type                                        = user_function
     user_function                               = "usrClientServer"
     user_values                                 = { -4, 1, 0 }
     user_strings                                = { "rsh pelican
 usrPres.pl","inflow" }
}
```

where the user-defined function *"usrClientServer"* may be implemented as:

```
#include "acusim.h"
```

```
#include "udf.h"
UDF_PROTOTYPE(usrClientServer) ;              /* function prototype */
Void usrClientServer(
     UdfHd udfHd,                             /* Opaque handle for accessing data */
     Real* outVec,                            /* Output vector */
     Integer nItems,                          /* Number of items in outVec */
     Integer vecDim                           /* Vector dimension of outVec */
) {
     Real mfValue ;                           /* MF return value */
     Real targPres ;                          /* target pressure */
     Real verbose ;                           /* verbose level */
     Real* trac ;                             /* pointer to surface traction */
     Real* usrVals ;                          /* user supplied values */
     String command ;                         /* execution command */
     String inflowOsi ;                       /* inflow surface output */
     String* usrStrs ;                        /* user supplied strings */
     char buffer[1024] ;                      /* a string buffer */
     char* name ;                             /* name of the UDF */
   udfCheckNumUsrVals( udfHd, 3 ) ;
   udfCheckNumUsrStrs( udfHd, 2 ) ;
   usrVals = udfGetUsrVals( udfHd ) ;
   usrStrs = udfGetUsrStrs( udfHd ) ;
   targPres = usrVals[0] ;
   mfValue = usrVals[1] ;
   verbose = usrVals[2] ;
   command = usrStrs[0] ;
   inflowOsi = usrStrs[1] ;
   /* Start the client */
   if ( udfFirstCall(udfHd) ) {
      sprintf( buffer, "%s %g %g %g", command,
               targPres, mfValue, verbose ) ;
      udfOpenPipePrim( udfHd, buffer ) ;
   }
   if  ( !udfFirstStep(udfHd) ) {
       /* Get the average pressure at the inflow surface */
       trac = udfGetOsiData( udfHd,inflowOsi, UDF_OSI_TRACTION ) ;
       /* Send the data to the client program */
       udfWritePipe( udfHd, "pressure= %.16g", -trac[0] ) ;
       /* Receive the data from the client program */
       udfReadPipe( udfHd, buffer, 1024 ) ;
       sscanf( buffer, "%*s %le", &mfValue ) ;
       if ( fabs(-trac[0]-targPres) > 1.e-4*fabs(targPres) ) {
            udfSetSig( udfHd, UDF_SIG_NO_CONV, 0 ) ;
  }
}
outVec[0] = mfValue ;
name = udfGetName( udfHd ) ;
udfCheckUgd( udfHd, name ) ;
udfSetUgdData( udfHd, name, mfValue ) ;
} /* end of usrClientServer() */
```

The perl script `usrPres.pl` may be implemented as:

```
#!/usr/bin/env perl
$| = 1 ;
if ( scalar(@ARGV) != 3 ) {
   die "Usage: $0 target_pressure initValue verbose" ;
}
$targPres = shift @ARGV ;
$mf       = shift @ARGV ;
$verbose  = shift @ARGV ;
$atFct    = 0.8 ;
```

```
while( <> ) {
  chomp ;
  $line = $_ ;
  if ( $line !~ /\Apressure= (.*)\Z/ ) {
   die "Invalid line: $_\n" ;
}
$mfPrev = $mf ;
$p1     = $1 ;
$p2     = $p1 * $p1 ;
$lhs11  = $atFct * $lhs11 + $p1 * $p1 ;
$lhs12  = $atFct * $lhs12 + $p1 * $p2 ;
$lhs22  = $atFct * $lhs22 + $p2 * $p2 ;
$res1   = $atFct * $res1 + $p1 * $mf ;
$res2   = $atFct * $res2 + $p2 * $mf ;
$d      = $lhs11 * $lhs22 - $lhs12 * $lhs12 ;
if ( $d > 1.e-8 * ($lhs11 * $lhs22) ) {
$a1     = (+$lhs22 * $res1 - $lhs12 * $res2) / $d ;
$a2     = (-$lhs12 * $res1 + $lhs11 * $res2) / $d ;
$mf     = ($a2 * $targPres + $a1) * $targPres ;
} elsif ( $lhs11 > 0 ) {
$a1     = $res1 / $lhs11 ;
$mf     = $a1 * $targPres ;
}
$mf     = 2.0 * $mfPrev if $mf > 2.0 * $mfPrev ;
$mf     = 0.5 * $mfPrev if $mf < 0.5 * $mfPrev ;
printf "multiplier_function= %.16g\n", $mf ;
print STDERR "pressure = $p1 mf = $mf\n" if $verbose ;
}
```

A user_global_data type uses the current value of the global data variable given by the user_global_data parameter as the value for the multiplier function.

A modifiable type acts exactly the same as a constant type, except its value may be changed within a user-defined function of another multiplier function as follows:

```
MULTIPLIER_FUNCTION( "Mod MF" ) {
   type                 = modifiable
   constant_value       = 2 #in case not modified
   dependencies         = { "main" }
}
MULTIPLIER_FUNCTION( "main" ) {
   type                 = user_function
   user_function        = "usrSomeMf"
}
```

Note: *dependencies* must be specified since the dependence on the second mulitiplier function cannot be detected automatically. See below.

Within the UDF "usrSomeMf" you can now have:

```
udfSetMfData( udfHd, "Mod MF", 12. );
```

See the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

The result of MULTIPLIER_FUNCTION may be modified in two ways. The *evaluation* parameter controls when the multiplier function is updated. For example, setting this to once_per_time_step freezes the value for the rest of the time step.

The *filter* parameter controls how much smoothing is applied to the multiplier function. If relaxation is specified then the multiplier function is given by the following:

$$M_n = a\tilde{M}_n + (1-a)M_{n-1} \tag{31}$$

where a is given by *filter_relaxation_factor*, M_n is the final value of the multiplier function at time step n, and $\tilde{M}_n$ is the value of the multiplier function without filtering at time step n.

For an iir filter, an infinite impulse response (IIR) digital filter is applied. This is a generalization of the relaxation filter and is given by:

$$M_n = \sum_{i=1}^{N_a} a_i \tilde{M}_{n+1-i} + \sum_{i=1}^{N_b} b_i M_{n-i} \tag{32}$$

with the constraint that

$$\sum_{i=1}^{N_a} a_i + \sum_{i=1}^{N_b} b_i = 1 \tag{33}$$

where $a_i, i = 1, \ldots, N_a$ is given by *iir_input_coefficients* and $b_i, i = 1, \ldots, N_b$ is given by *iir_output_coefficients*. N_a and N_b are determined by the number of elements of the corresponding arrays. If the constraint is not satisfied, AcuSolve automatically scales the coefficients appropriately. For example,

```
MULTIPLIER_FUNCTION( "MF with digital filter" ) {
   ...
   filter                      = iir
   iir_input_coefficients      = { 0.8, 0.8 }
   iir_output_coefficients     = { -0.6 }
}
```

The mic filter is designed to control error amplifications when two fields are coupled together through a time multiplier function. This is generally used for DC-FSI problems. See EXTERNAL_CODE. For example,

```
MULTIPLIER_FUNCTION( "MIC_TMF" ) {
   ...
   evaluation       = often
   filter           = mic
}
```

The *dependencies* parameter does not modify the value of the multiplier function but ensures that it is evaluated correctly. If the current multiplier function is a modifiable type or depends on a user global data variable which in turn is set within a user-defined function of a second multiplier function, then it is necessary for the second multiplier function to be evaluated before the current one. However, AcuSolve has no way to detect this indirect dependency, hence you must list all such dependencies explicitly. Dependencies may be chained indefinitely, but any non-terminating recursive loop is flagged as an error.

Note: Direct dependencies, such as those that occur with the *multiplier_function* type, are automatically detected and do not need to be listed in dependencies.

NODAL SHAPES

Specifies the morph shapes and associated design optimization variables.

Type

AcuSolve Command

Syntax

NODAL_SHAPES ("name") {parameters}

Qualifier

This command has no qualifier.

Parameters

mesh_motion_precedence *(integer) >=0 <=3 [=1]*
: Mesh motion precedence.

design_variables *(list) [={}]*
: User given name of the design optimization variable.

nodal_morphs *(file) [=Read("unknown")]*
: User provided nodal morphs shapes. The nodal morph shapes are always read a file, where the nodal displacement data is given for the coordinate directions x, y and z.

Description

The shape mesh displacement may be defined using the `NODAL_SHAPES` command where the nodal shape file contains the displacements of all three coordinate directions for each design optimization variable. The shape file, such as `shapes.xyzsp`, has a simple file structure where the first column is the node identification number, and the remaining columns are the displacements corresponding to this node for all design optimization variables. For example, if there are 10 design optimization variables, the file `shapes.xyzsp` has 11 columns, one node column and an x-displacement for each of the 10 design optimization variables. There is one row for each node in the shape; assume there are N rows. The y-displacements and z-displacements are also listed concatenated to the bottom of the file. For example:

n_1	δx_1^1	δx_1^2	δx_1^3	δx_1^4	δx_1^5	δx_1^6	δx_1^7	δx_1^8	δx_1^9	δx_1^{10}
n_2	δx_2^1	δx_2^2	δx_2^3	δx_2^4	δx_2^5	δx_2^6	δx_2^7	δx_2^8	δx_2^9	δx_2^{10}
.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.
n_N	δx_N^1	δx_N^2	δx_N^3	δx_N^4	δx_N^5	δx_N^6	δx_N^7	δx_N^8	δx_N^9	δx_N^{10}
n_1	δy_1^1	δy_1^2	δy_1^3	δy_1^4	δy_1^5	δy_1^6	δy_1^7	δy_1^8	δy_1^9	δy_1^{10}
n_2	δy_2^1	δy_2^2	δy_2^3	δy_2^4	δy_2^5	δy_2^6	δy_2^7	δy_2^8	δy_2^9	δy_2^{10}
.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.
n_N	δy_N^1	δy_N^2	δy_N^3	δy_N^4	δy_N^5	δy_N^6	δy_N^7	δy_N^8	δy_N^9	δy_N^{10}
n_1	δz_1^1	δz_1^2	δz_1^3	δz_1^4	δz_1^5	δz_1^6	δz_1^7	δz_1^8	δz_1^9	δz_1^{10}
n_2	δz_2^1	δz_2^2	δz_2^3	δz_2^4	δz_2^5	δz_2^6	δz_2^7	δz_2^8	δz_2^9	δz_2^{10}
.	.	.	.	.	.	.	.	.	.	.
.	.	.	.	.	.	.	.	.	.	.
n_N	δz_N^1	δz_N^2	δz_N^3	δz_N^4	δz_N^5	δz_N^6	δz_N^7	δz_N^8	δz_N^9	δz_N^{10}

Figure 3:

Where the subscripts for each $\delta x_i j$, $\delta y_i j$ and $\delta z_i j$ are associated with the nodal identification number n_i and the superscript is associated with the shape, or more precisely the design optimization variable.

```
NODAL_SHAPES( "NODAL_SHAPES.1" ) {
    mesh_motion_precedence = 1
    design_variables = { "DV1", "DV2", "DV3", "DV4", "DV5", "DV6", "DV7" }
    nodal_morphs = Read( "shapes.NODAL_SHAPES.1.xyzsp" )
}
```

OBJECTIVE

Specifies the objective for an optimization problem.

Type

AcuSolve Command

Syntax

OBJECTIVE {parameters}

Qualifier

This command has no qualifier.

Parameters

type (enumerated) [=minimize]

Specifies if the objective is to minimize or maximize the response.

minimize

maximize

response_variable (string) [=""]

User defined name of response variable that is computed from the simulation.

Description

This command specifies the objective of the optimization. The objective can be either to maximize or minimize the defined response variable. A typical objective defined by the OBJECTIVE command is a function of response variables, defined using the RESPONSE_VARIABLE commands.

```
OBJECTIVE( "Minimize_Pressure_Drop" ) {
    type = minimize
    response_variable = "Inlet_Pressure"
}
```

For topology optimization, minimizing the mechanical energy losses, the referenced response variable should be of type *topology_variable* specified using the RESPONSE_VARIABLE command.

```
OBJECTIVE( "Minimize_Mechanical_Energy" ) {
type = minimize
response_variable = "mechanical_energy"
}
```

OPTIMIZATION

Specifies optimization is activated.

Type

AcuSolve Command

Syntax

OPTIMIZATION {parameters}

Qualifier

This command has no qualifier.

Parameters

type (string) [="shape_optimization"]

The type of optimization to be performed. Shape optimization requires DESIGN_VARIABLE commands and topology optimization requires one DESIGN_VARIABLES_FIELD command.

shape_optimization	Geometrical shape modification using morph shapes.
parametric_optimization	Optimization of model parameters (constants).
topology_optimization	Optimization of topological domain.

objectives(string) [=""]

The objectives parameter lists the objective functions used in optimization. If more than one objective is specified, Pareto optimization is assumed. The objectives are defined using the objective command.

constraints(string) [=""]

The constraints parameter lists the constraint functions used in optimization. The constraints are defined using the constraint command.

optimizer_convergence_tolerance (real) >=0 [=1.e-4]

The optimizer convergence tolerance is the tolerance for the optimization algorithm.

optimizer_oscillation_control_factor (real) >=0 [=0.1]

Optimizer oscillation control factor limits how the design optimization variables may oscillate. If a particular design variable increases a certain amount x during the previous optimization case, then this design variable is limited to decrease x multiplied by the *optimizer_oscillation_control_factor* for the current case. It will not limit the amount the design variable may increase. Vice versa, if a design variable decreases the previous case then there is no limit on how much it may continue to decrease, but there is a limit on how much it may increase.

topology_optimizer (enumerated) [=method_of_moving_asymptotes]

When *type* = topology_optimization, the method of moving asymptotes method is used as the optimization method to minimize the objective function.

topology_defeaturing_length_scale (real) >=0 [=0.0]
Length scale at which defeaturing/smoothing of the topology design is performed. When set to a positive value, a defeaturing algorithm is applied to the topology design and the resulting defeatured/smoothed design is stored in an additional variable named levelset providing a signed distance function to the defeatured/smoothed geometry. The defeaturing algorithm removes from the design any feature of size smaller than the provided length scale and smooths out sharp angle features accordingly to the defeaturing length scale, for example, rounding up sharp corners. If set to zero, or negative, no defeaturing is performed and the levelset variable is not created. Only used for optimization with *type* = topology_optimization.

objective_min_flow_volume (enumerated) [="auto"]
Minimization of fluid volume fraction simultaneously with the objective function minimization/ maximization.

auto	Active if there is no design volume fraction constraint, inactive if there is a design volume fraction constraint.
on	Always active.
off	Always inactive.

objective_min_flow_volume_mult (real) [=1.0]
Multiplier of the fluid volume fraction objective.

topology_auto_darcy (boolean) [="on"]
When *topology_auto_darcy* is on the darcy coefficient is automatically computed based on the amount of flow in the solid design region. The value defined in the input file is used as the initial value for the automatic computation.

on	Darcy coefficient is computed automatically.
off	Darcy coefficient has a constant value.

Description

The essential concept is to solve an optimization problem for user defined objective functions and constraints, given the fluid dynamics solution. Here the geometrical shape of the domain may change due to the variation of the design optimization shape control variables. The form of the optimization problem may for example take the form

$$\begin{aligned} &\text{Maximize } f(flow, shape) \\ &\quad \text{Subject to } g_1(flow, shape) \le 0, \\ &\qquad g_n(flow, shape) \le 0, \\ &\qquad \vdots \\ &\qquad G_{NS}(shape) = 0, \end{aligned}$$

Figure 4:

where f is a user defined objective function and g_i are n user defined constraints depending on the flow solution and the shape determined by the design optimization shape control variables. Further, $G_{NS}(shape) = 0$ denotes the equation system for the flow, including the Navier-Stokes equation for instance. This constraint is taken for granted below, and is not written out in the formulations.

A few commands related to the specification of the optimization problem itself are added to the command structure. In this section, objectives and constraints are defined under the optimization command umbrella, referring to solution related values that are specified in an objective and constraint commands for each such function. The optimization command also specifies the optimization solution method and strategy.

There is always one optimization command for an optimization AcuSolve run, and most parameters besides objectives and constraints have default values.

```
OPTIMIZATION {
    objectives                          = { "Minimize_Pressure_Drop" }
    constraints                         = { "Top_constraint" }
    optimizer_convergence_tolerance     = 1.0e-4
    optimizer_oscillation_control_factor= 0.1
}
```

For topology optimization, minimizing the mechanical energy losses, the optimization command is given by:

```
OPTIMIZATION {
    type = topology
    objectives = "Minimize_Mechanical_Energy"
}
```

PARTICLE_TRACE

Specifies communication parameters for coupling with AcuTrace.

Type

AcuSolve Command

Syntax

PARTICLE_TRACE {parameters}

Qualifier

This command has no qualifier.

Parameters

socket_host or *host* *(string) [=""]*

User-given name of the host that runs AcuTrace.

socket_port or *port integer* *>=1 [=20000]*

Socket port for communication with external code. Used with socket communication.

Description

This command provides support for solving coupled particle-flow problems by allowing AcuSolve and AcuTrace to run in a coupled manner. Flow quantities (such as velocity, pressure, temperature, and species) are communicated to AcuTrace and particle states and source terms are returned to AcuSolve. See also the EQUATION, TIME_SEQUENCE, and STAGGER commands as well as the *AcuTrace Command Reference Manual*. Communication between the two codes uses the socket protocol exclusively.

The host on which AcuTrace runs is determined automatically except in highly unusual circumstances. Leave *socket_trace*="" unless one of these situations arises.

RADIATION

Specifies global radiation algorithms and parameters.

Type

AcuSolve Command

Syntax

RADIATION {parameters}

Qualifier

This command has no qualifier.

Parameters

view_factor_type (enumerated) [=hemicube]
Calculation method for the view factor.

hemicube	Hemicube. Requires *num_hemicube_bins* and *max_surface_subdivision*.

num_hemicube_bins or *bins (integer) >0 [=200]*
Number of bins for the view factor calculation. Used with hemicube view factor type.

max_surface_subdivision (integer) >=1 [=1]
Maximum number of segments to split a surface facet when proximity condition is violated. Used with hemicube view factor type.

smoothing_type (enumerated) [=least_squares]
Type of view factor smoothing algorithm.

least_squares	Least squares smoothing.

stefan_boltzmann_constant (real) >0 [=5.67e-8]
Stefan-Boltzmann constant. Default is in MKS units.

num_symmetry_planes (integer) >=0 <=3 [=0]
Number of orthogonal symmetry planes in geometrical model.

symmetry_center or *center (array) [={0,0,0}]*
Any point that lies at the intersection of all symmetry planes. Used with *num_symmetry_planes*>0.

symmetry_direction_1 (array) [={1,0,0}]
Normal direction to first symmetry plane. Used with *num_symmetry_planes*>0.

symmetry_direction_2 (array) [={0,1,0}]
Normal direction to second symmetry plane. Used with *num_symmetry_planes*>1.

symmetry_direction_3 (array) [={0,0,1}]
Normal direction to third symmetry plane. Used with *num_symmetry_planes*=3.

Description

This command specifies the global parameters for enclosure radiation heat transfer as defined by the RADIATION_SURFACE and EMISSIVITY_MODEL commands. Radiation commands with default values are automatically created for the P1 radiation method to model radiative heat transfer in a grey, optically thick participating medium. These parameters do not apply to radiation boundary conditions defined by ELEMENT_BOUNDARY_CONDITION commands or to heat fluxes defined by SOLAR_RADIATION and related commands.

For example,

```
EQUATION {
   ...
   radiation                        = enclosure
   absolute_temperature_offset      = 273.14
}
RADIATION {
   view_factor_type                 = hemicube
   num_hemicube_bins                = 200
   max_surface_subdivision          = 4
   smoothing_type                   = least_squares
   stefan_boltzmann_constant        = 5.67e-08
   num_symmetry_planes              = 1
   symmetry_center                  = { 0, 0, 0 }
   symmetry_direction_1             = { 1, 0, 0 }
}
```

specifies that the enclosure radiation equation is to be solved, the hemicube algorithm with 200 bins per side is used to calculate the view factor, each radiation facet may be split into a maximum of 4 sub-surfaces to reduce proximity and visibility errors, the least squares method is used to smooth the view factor, the Stefan-Boltzmann constant is 5.67x10-, and one symmetry plane is modeled.

REFERENCE_FRAME

Specifies a reference frame for element sets and boundary conditions.

Type

AcuSolve Command

Syntax

REFERENCE_FRAME("name") {parameters...}

Qualifier

User-given name.

Parameters

centrifugal (boolean) [=on]
: Flag specifying whether to apply centrifugal force.

coriolis (boolean) [=on]
: Flag specifying whether to apply coriolis force.

angular_acceleration (boolean) [=on]
: Flag specifying whether to apply angular acceleration force.

rotation_center or center (array) [={0,0,0}]
: Center of rotation, specified in global xyz coordinate system.

angular_velocity or ang_vel (array) [={0,0,0}]
: Angular velocity vector of the rotating frame, in radians per unit of time.

multiplier_function (string) [=none]
: User-given name of the multiplier function for scaling the angular velocity. If none, no scaling is performed.

Description

This command, instead of ROTATION_FORCE, is the preferred way to specify rotational body forces. Using both commands at the same time will normally produce incorrect results.

This command specifies a rotating frame of reference. This frame of reference has two uses. First, when it is referenced in an ELEMENT_SET command, the elements are assumed to be solved in the given rotating reference frame and rotational body forces are added for that element set. Second, when referenced in a NODAL_BOUNDARY_CONDITION command for a velocity component, the velocity vector is transformed appropriately before the boundary condition is applied.

The rotational body forces are known as the centrifugal, coriolis, and angular acceleration forces. They appear as the last three terms in the momentum equation:

$$\rho\frac{\partial u}{\partial t}+\rho u\cdot\nabla u+\nabla p=\nabla\cdot\tau-\rho\Omega\times\Omega\times\left(x-x_0\right)-2\rho\Omega\times u-\rho\dot{\Omega}\times\left(x-x_0\right) \quad (38)$$

where ρ is the density; u is the velocity vector; p is the pressure; $t=\left[\tau_{ij}\right]$ is the viscous stress tensor; x is the current coordinate vector; x_0 is the center of rotation, given by *rotation_center*; Ω is the angular velocity vector, given by *angular_velocity* (and *multiplier_function*, if specified); and Ω is the angular acceleration vector. For these forces to be active, the REFERENCE_FRAME command must be referenced by an ELEMENT_SET command, for example:

```
REFERENCE_FRAME( "rotating frame" ) {
   centrifugal                          = on
   coriolis                             = on
   rotation_center                      = { 0, 0, 0 }
   angular_velocity                     = { 0, 1, 0 }
}
ELEMENT_SET( "fluid in rotating frame of reference" ) {
   reference_frame                      = "rotating frame"
   ...
}
```

In this example, the frame of reference rotates around the y axis at the rate of one radian per unit of time, with the axis of rotation centered at the point (0,0,0) in the global xyz coordinate system. The right hand rule is used to define the direction of rotation of the reference frame. Since the angular velocity is constant, the angular acceleration is zero and thus the *angular_acceleration* flag has no effect.

In NODAL_BOUNDARY_CONDITION commands the default reference frame is none. This means that no transformation of the velocity field from the fixed (sometimes called "laboratory") reference frame is performed before the boundary condition is applied. If a reference frame is specified then the following transformation is performed:

$$u_{trans}=u_{fix}-\Omega\times\left(x-x_0\right) \tag{39}$$

where u_{fix} is the velocity in the fixed frame. The given boundary condition is then applied to the appropriate component of u_{trans}. For example, consider a fan blade rotating according to the reference frame given above. The velocity at any point on the blade is $\Omega\times\left(x-x_0\right)$ and the transformed velocity is zero. The boundary condition command for the x -component of velocity becomes:

```
NODAL_BOUNDARY_CONDITION( "blade x_velocity" ) {
   variable           = x_velocity
   type               = zero
   nodes              = Read( "blade.nbc" )
   reference_frame    = "rotating frame"
}
```

The commands for the other two components of velocity are similar. The *reference_frame* parameter has no effect on any other variable. Boundary conditions specified in this manner are independent of the reference frame(s) of the elements connected to the boundary condition nodes. In particular, any of these nodes may be connected to two or more elements with different reference frames.

The *multiplier_function* parameter may be used to scale the angular velocity. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file; see the MULTIPLIER_FUNCTION command for an example. The angular acceleration vector is calculated from a simple numerical difference of the multiplier function, as follows:

$$\dot{\Omega} = \Omega\left(f_n - f_{n-1}\right)/\Delta t \qquad (40)$$

where f_n and f_{n-1} are the multiplier function values at the current and previous time steps, Ω is given by *angular_velocity*, and Δt is the time increment. The angular acceleration is set equal to zero during the first step since the previous multiplier function value is not available then. The angular acceleration is always zero if the multiplier function is constant or not specified.

Element sets with different reference frames may be used in the same problem. No special user input is required at the interface of two such element sets, this is handled automatically. An example where this is useful is a mixing tank with rotating blades in the middle and fixed blades attached to the outside walls. The inner rotating element set encloses the rotating blades and is defined with a rotating reference frame while the outer stationary element set contains the fixed blades. The interface between the rotating and stationary element sets must be a surface of revolution, such as a cylinder, cone, or sphere, with the same axis as the reference frame. The rotating blades have nodal boundary conditions with the same reference frame as the rotating element set. The resulting steady solution is certainly not as accurate as a transient solution using a sliding interface and a rotating mesh for the rotating element set, but it is likely to be one or two orders of magnitude cheaper to compute. This strategy works well if the objective is to get a gross effect of the rotating body into the flow, such as the thrust and swirl of a marine impeller. However, it does not yield any information about the effect of rotors passing by stators. The commands to implement this model may be as simple as:

```
ASSIGN {
   variable                              = RPM
   value                                 = (2* PI) / 60
}
REFERENCE_FRAME( "impeller 1" ) {
   rotation_center                       = { 0.0, 0.0, 0.0 }
   angular_velocity                      = { 0, 250. * RPM, 0 }
}
ELEMENT_SET( "impeller 1 region" ) {
   shape                                 = four_node_tet
   material_model                        = glycerin
   elements                              = Read( "mixing.impeller1.cnn" )
   reference_frame                       = "impeller 1"
SIMPLE_BOUNDARY_CONDITION( "impeller 1 wall" ) {
   surfaces                              = Read( "mixing.impeller1.wall.ebc" )
   element_set                           = "impeller 1 region"
   shape                                 = three_node_triangle
   type                                  = wall
   reference_frame                       = "impeller 1"
}
ELEMENT_SET( "stationary region" ) {
   shape                                 = four_node_tet
   material_model                        = glycerin
   elements                              = Read( "mixing.stationary.cnn" )
}
SIMPLE_BOUNDARY_CONDITION( "stationary wall" ) {
   surfaces                              = Read( "mixing.stationary.wall.ebc" )
   element_set                           = "stationary region"
   shape                                 = three_node_triangle
   type                                  = wall
}
```

RESPONSE VARIABLE

Specifies the responses to be used for optimization.

Type

AcuSolve Command

Syntax

RESPONSE_VARIABLE {parameters}

Qualifier

This command has no qualifier.

Parameters

type (enumerated) [=surface_output]
Responsible variable types.

surface_output

element_output

time_history_output

time_history_output_distance or dist

topology_variable

design_variable

constant or const

identity or id

square or sq

cube

square_root or sqrt

exponential_function or exp

natural_logarithm or ln

sine or sin

cosine or cos

tangent or tan

cotangent or cot

arcsine or asin

arccosine or acos

arctangent or atan

arccotangent or acot

negation or neg

reciprocal or one_over

absolute or abs

ratio or over

addition or plus

subtraction or minus

multiplication or times

power or pow

average or mean

average_absolute_value

minimum or min

minimum_absolute_value

absolute_minimum_value

maximum or max

maximum_absolute_value

absolute_maxmimum_value

summation or sum

sum_absolute_value

sum_squares

sqrt_sum_squares or norm

weighted_sum

product

`surface_output` *(string) [=""]*
User defined name of the surface from which the output will be extracted.

`surface_output_variable` *(enumerated) [=pressure]*
The surface output variables are used in the `RESPONSE_VARIABLE` command.

x_traction

y_traction

z_traction

mass_flux

heat_flux

x_velocity or xvel

y_velocity or yvel

z_velocity or zvel

temperature or temp

pressure or pres

area

partial_volume

`element_output` *(string) [=""]*
User defined name of the element set from which the output will be extracted.

`element_output_variable` *(enumerated) [=volume]*
Element output quantities are used in optimization.

volume

x_velocity or xvel

y_velocity or yvel

z_velocity or zvel

pressure or pres

temperature or temp

total_pressure

mass

time_history_output (string) ["unknown"]
User defined name of time history output.

time_history_output_variable (enumerated) [=pressure]
Time history output quantities are used in optimization.

x_undeformed or x0

y_undeformed or y0

z_undeformed or z0

x_deformed or x

y_deformed or y

z_deformed or z

x_velocity or xvel

y_velocity or yvel

z_velocity or zvel

pressure or pres

temperature or temp

time_history_output_1 (string) [" "]
User defined name of time history output point. Used with *time_history_output_distance*.

time_history_output_2 (string) [""]
User defined name of time history output point. Used with *time_history_output_distance*.

time_history_output_coordinates_1 (enumerated) [=deformed]
Specifies the method of tracking *time_history_output_1* location.

undeformed_coordinates or undeformed

deformed_coordinates or deformed

time_history_output_coordinates_2 (enumerated) [=deformed]
Specifies the method of tracking *time_history_output_2* location.

undeformed_coordinates or undeformed

deformed_coordinates or deformed

time_history_index_1 (integer) [=1]
Time history indices must be referenced by the order of the coordinates in the time history output command.

time_history_index_2 (integer) [=1]
Time history indices must be referenced by the order of the coordinates in the time history output command.

design_variable (string) [=""]
User-defined name of the design variable.

topology_variable (enumerated) [=mechanical_energy]

mechanical_energy	Mechanical energy.
design_volume_fraction	The integral of the design variables field variable divided by the volume of the design space. This will give a value of zero for an empty design space, and a value of one for a design space where the design variables field variable is constant one.
surface_mass_flow	The mass flow rate at a flow boundary surface whose name is given in *mass_flow_surface*.

mass_flow_surface (string) [=""]
The name of the flow boundary surface for which a mass flow rate optimization constraint can be imposed when *topology_variable* = surface_mass_flow. Most often this will reference a SIMPLE_BOUNDARY_CONDITION of *type* = outlet.

constant_response (real) [=0.0]
Specifies a constant value.

response_variables (list) [={}]
Specifies the response variables.

weights (array) [={}]
Specifies the factors multiplying the response variables in a weighted sum. If you use M response variables in the weighted sum, you need to specify M weights.

Description

The objectives and constraints are defined in terms of response variables that may reference surface or volume outputs. Response variables may also control parameters by using multiplier functions of type set to response_variable. The values of the objective and constraint functions are computed using response variable commands. In turn, the values of the response variable functions are computed from either (i) integrated surface quantities, (ii) integrated volume quantities, (iii) design variables, (iv) given constants, or (v) functions of one or more other response variables.

Below are some examples of responses.

Response variable generated using surface output:

```
RESPONSE_VARIABLE("inflow pressure") {
 type                    = surface_output
 surface_output          = "inflow"
 surface_output_variable = pressure
}
SURFACE_OUTPUT("inflow") {
 surfaces                     = Read( "inflow.srf" )
 shape                        = three_node_triangle
 element_set                  = "fluid"
 integrated_output_frequency = 1
}
```

Response variable generated using element output:

```
RESPONSE_VARIABLE("mass of object") {
 type                    = element_output
 element_output          = "object"
 element_output_variable = mass
}
ELEMENT_OUTPUT("object") {
 elements                     = Read( "object.cnn" )
 shape                        = four_node_tet
 element_set                  = "object"
 integrated_output_frequency = 1
}
```

Response variable generated using time history output:

```
RESPONSE_VARIABLE("temperature in a node") {
 type                          = time_history_output
 time_history_output           = "oth_node"
 time_history_output_variable = temperature
}
TIME_HISTORY_OUTPUT("oth_node") {
 type             = nodal
 nodes            = { 54 }
 output_frequency = 1
}
RESPONSE_VARIABLE("x-position of a deformed point") {
 type                          = time_history_output
 time_history_output           = "oth_point"
 time_history_output_variable = x_deformed
}
TIME_HISTORY_OUTPUT("oth_point") {
 type             = coordinates
 coordinates      = { 1, 1,0, 0.0, 2.0 }
 output_frequency = 1
}
```

Response variable generated based on time history output distance:

```
RESPONSE_VARIABLE("distance between two deformed points") {
 type                                = time_history_output
 time_history_output_1               = "points"
 time_history_output_2               = "points"
 time_history_output_coordinates_1 = deformed
 time_history_output_coordinates_2 = deformed
 time_history_index_1                = "1"
 time_history_index_2                = "2"
```

```
}
TIME_HISTORY_OUTPUT("oth_point") {
 type              = coordinates
 coordinates       = {
    1, 0.0, 0.0, 0.0
    2, 2.0, 3.0, 0.0
       }
 output_frequency = 1
}
```

The indices in time history output are numbered from one to two and are referenced in the response variable command. In general, the response variable command time history indices must be referenced by the order of the coordinates in the time history output command. The first column is not the index number, but the point identifier.

Response variable for topology optimization:

```
RESPONSE_VARIABLE("mechanical energy") {
 type = topology_variable
topology_variable = mechanical_energy
}
```

SOLAR_RADIATION

Specifies global solar radiation algorithms and parameters.

Type

AcuSolve Command

Syntax

SOLAR_RADIATION {parameters}

Qualifier

This command has no qualifier.

Parameters

type (enumerated) [=piecewise_linear]

Type of solar heat flux.

piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.

curve_fit_values or curve_values (array) [={0,0,0,-1352}]

A four-column array of independent-variable/solar-flux-vector data values. Used with piecewise_linear type.

curve_fit_variable or curve_var (enumerated) [=time]

Independent variable of the curve fit. Used with piecewise_linear type.

time	Current run time.

multiplier_function (string) [=none]

User-given name of the multiplier function for scaling the solar flux vector. If none, no scaling is performed.

computation_type (enumerated) [=ray_trace]

Type of algorithm used to compute the solar heat flux

ray_trace	Ray trace algorithm. Requires *num_rays*, *num_diffuse_sub_rays max_ray_reflections*, *min_ray_energy* and terminated_ray_redistribution_factor

num_rays (integer) >=1 [=1000000]

Number of rays for the solar heat flux calculation. Used with ray_trace computational type.

num_diffuse_sub_rays (integer) >=1 [=4]

Number of sub-rays a ray is split into for diffuse reflections and transmissions. Used with ray_trace computational type.

max_ray_reflections (integer) >=0 [=10]

Maximum number of times a single ray is allowed to reflect before it is terminated. Used with ray_trace computational type.

min_ray_energy (real) >=0 [=1.e-3]
: Minimum relative energy a ray is allowed to have before it is terminated. Used with ray_trace computational type.

terminated_ray_redistribution_factor (real) >=0 <=1.0 [=1.0]
: Factor for distributing the remaining energy of a terminated ray. Used with ray_trace computational type.

smoothing or smooth (boolean) [=on]
: Flag specifying whether to spatially smooth the solar heat flux.

Description

This command specifies the global parameters for solar radiation heat flux as defined by the SOLAR_RADIATION_SURFACE and SOLAR_RADIATION_MODEL commands. These parameters do not apply to radiation boundary conditions defined by ELEMENT_BOUNDARY_CONDITION commands or to RADIATION or related commands.

For example,

```
SOLAR_RADIATION {
   type                                 = piecewise_linear
   curve_fit_values                     = { 0, 0, 0, -1352 }
   curve_fit_variable                   = time
   computation_type                     = ray_trace
   num_rays                             = 1000000
   num_diffuse_sub_rays                 = 4
   max_ray_reflections                  = 10
   min_ray_energy                       = 1.e-3
   terminated_ray_redistribution_factor = 1
   smoothing                            = on
}
```

specifies that the solar radiation heat fluxes are to be added to the thermal energy equation based on a constant solar flux of 1352 in the negative z direction. The fluxes are to be computed using a ray trace algorithm with 1000000 initial rays, diffuse reflections and transmissions split into 4 sub-rays, a maximum of 10 reflections and a minimum relative energy of 10^{-3} before a ray is terminated, and the remaining energy of a terminated ray distributed over the previous absorptions of this ray. The computed solar heat flux is then spatially smoothed.

Local Surface Coordinate System

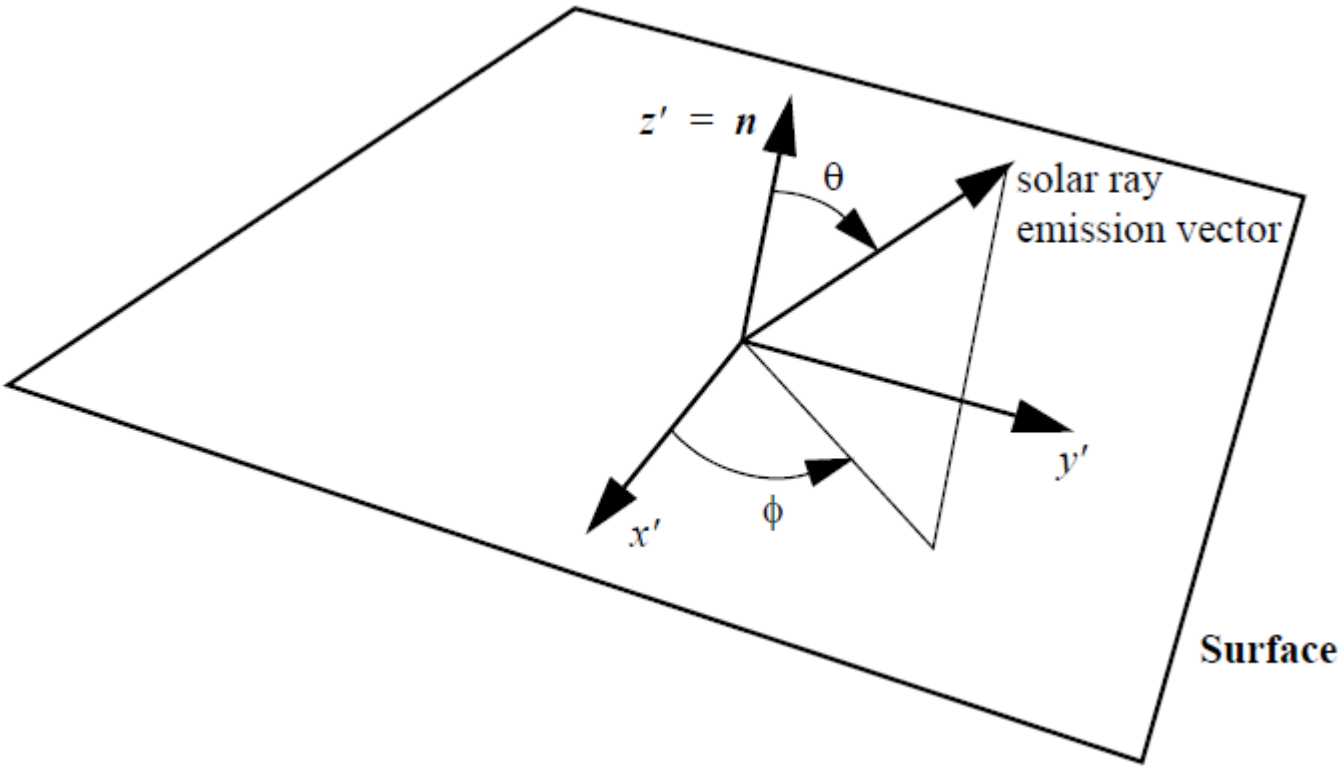


Figure 6:

The interaction of a solar ray photon with a surface is described in terms of a local spherical coordinate system. The polar angle θ is also the angle of incidence or emission. Surface properties dependent on the azimuthal angle Φ are not currently supported. Five different types of interactions may occur. A specular transmission occurs when a photon passes straight through a surface with no change of direction. In a diffuse transmission the photon penetrates the surface, but its outgoing energy is uniformly distributed in solid angle over the hemisphere, weighted by projected surface area. For a specular reflection the angle of reflection is equal to the angle of incidence. Diffuse reflections are similar to diffuse reflections, except the hemisphere over which the outgoing energy is distributed is on the same side of the surface as the incident photon. Finally, the photon may be absorbed by the surface. These five interactions are associated with five surface properties that together must obey the following constraint:

$$\tau_s(\theta)+\tau_d(\theta)+\rho_s(\theta)+\rho_d(\theta)+\alpha(\theta)=1 \qquad (41)$$

where τ_s is the specular transmissivity, τ_d is the diffuse transmissivity, ρ_s is the specular reflectivity, ρ_d is the diffuse reflectivity and α is the absorptivity, The transmissivity and reflectivity properties are defined by the `SOLAR_RADIATION_MODEL` command, and the absorptivity by the above constraint. The energy distribution of a diffuse reflection or transmission must be proportional to $\cos(\theta)$ in order to obtain a uniform distribution of outgoing energy over the projected surface area.

The *`type`*, *`curve_fit_values`*, and *`curve_fit_variable`* parameters define the global incoming solar flux vector. A linear_curve_fit type interpolates the data in *`curve_fit_values`* linearly. The first column in this array is curve_fit_variable, and the second, third, and fourth are the three components of

the global solar heat flux vector. The default of (0,0,-1352) is in W/m^2, and is a typical value. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits. Thus for a constant solar heat flux, just one line in *curve_fit_values* is needed, as in the example above.

Photon-Surface Interactions

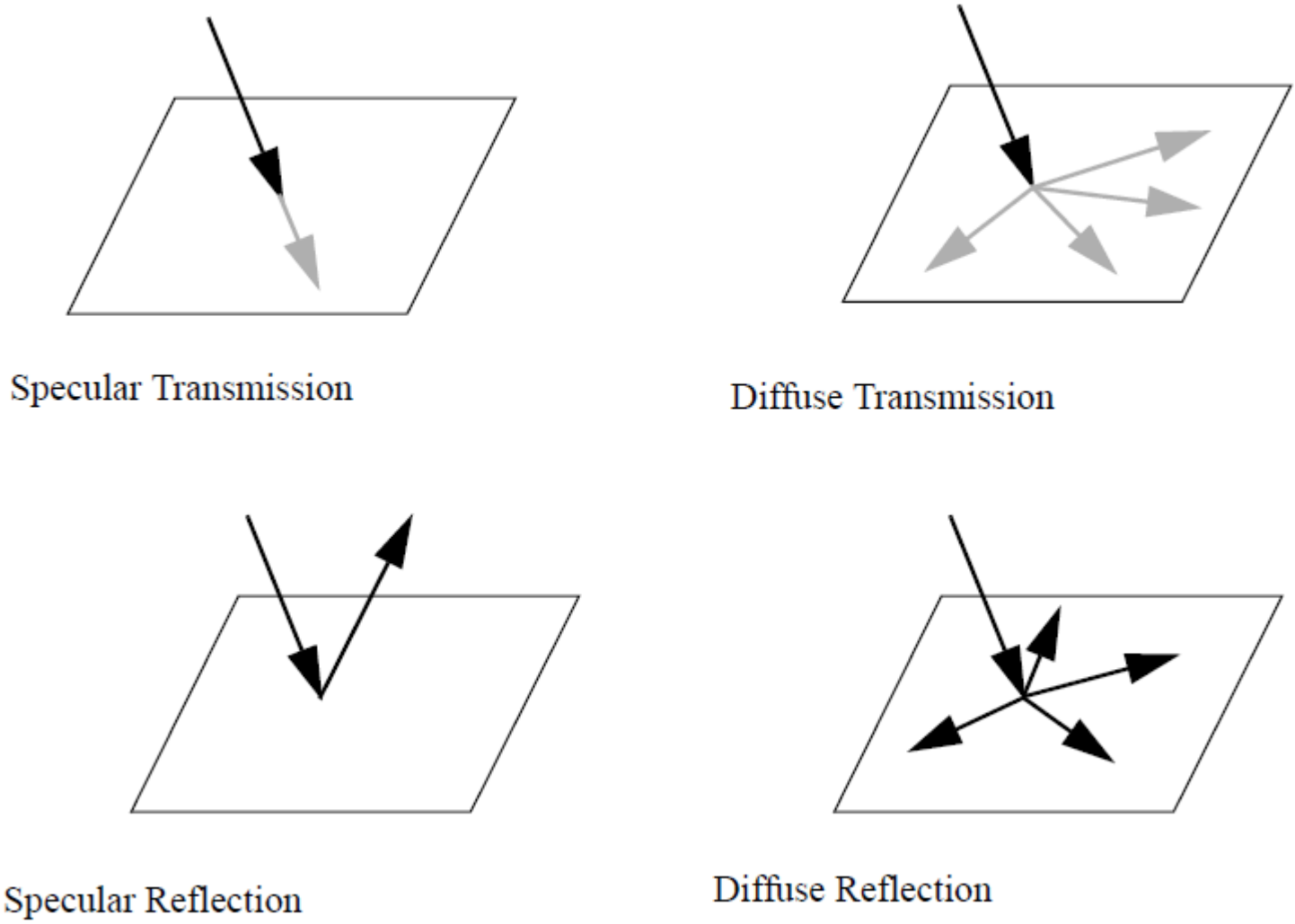


Figure 7:

The solar heat flux values may be read from a file. For example, time-dependent heat flux data may be placed in a file, say "solar_heat_flux.dat":

```
0        100         200           -1000
1000      80         160           -800
2000      60         120           -600
```

and read by:

```
SOLAR_RADIATION {
   type                 = piecewise_linear
   curve_fit_values     = Read( "solar_heat_flux.dat" )
```

```
    curve_fit_variable    = time
    ...
}
```

The *multiplier_function* parameter may be used to scale the solar heat flux vector. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file. For example, the following is equivalent to the above time-dependent solar heat flux:

```
SOLAR_RADIATION {
     type                       = piecewise_linear
     curve_fit_values           = { 0, 100, 200, -1000 }
     curve_fit_variable         = time
     multiplier_function        = "time varying"
     ...
}
MULTIPLIER_FUNCTION( "time varying" ) {
     type                       = piecewise_linear
     curve_fit_values           = {     0, 1.0 ;
                                     1000, 0.8 ;
                                     2000, 0.6;   }
     curve_fit_variable         = time
}
```

A ray_trace computational type uses the Monte Carlo method to compute exchange factors and the solar heat flux on every surface specified by the SOLAR_RADIATION_SURFACE commands. This is done as a pre-processing step for every line in *curve_fit_values*. The heat fluxes are then interpolated during the simulation according to type and *curve_fit_variable*. If only the magnitude of the solar heat flux vector changes and not the direction, it is less expensive to use the *multiplier_function* parameter as above.

The first step in the ray_trace computational type is to spread *num_rays* rays over a rectangle normal to the solar heat flux vector that encompasses all the solar radiation surfaces if they are projected along this vector onto the plane of the rectangle. The cost of the algorithm is proportional to *num_rays*, so you should be careful with using more rays than the default.

For diffuse transmissions and reflections the uniform distribution of the outgoing energy is approximated by *num_diffuse_sub_rays* discrete photons. These photons have random energies with the constraint of adding up to the given total energy. Their directions are also random (within the appropriate hemisphere) with a cos(θ) probability distribution. The cost of the algorithm can be proportional to as much as the square of this parameter.

Each ray and sub-ray is followed until one of three things happen. First, the ray may not hit any solar radiation surface. In this case its energy does not contribute to the heat flux of any surface. Second, it undergoes *max_ray_reflections* reflections and is terminated. Third, its relative energy is reduced through absorptions until it reaches *min_ray_energy* and is terminated. In the last two cases where the ray is terminated the disposal of the remaining energy is determined by *terminated_ray_redistribution_factor*. A value of zero means that this energy is thrown away and does not contribute to the heat flux of any surface. A value of one means that this energy is to be distributed over the previous surfaces that this ray encountered according to the absorptivity of each surfaces. Values between zero and one interpolate between these two limits.

For some problems a spatially noisy distribution of the solar heat flux results from not using a sufficient number of rays. If increasing the number of rays is too expensive then turning on the *smoothing* parameter will smooth the solar heat flux while conserving the integrated solar heat flux.

THERMAL_SHELL

Creates shell elements from surface elements for temperature field only.

Type

AcuSolve Command

Syntax

THERMAL_SHELL ("name") {parameters}

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]
Shape of the surfaces in this set.

three_node_triangle or tri3
Three-node triangle.

four_node_quad or quad4
Four-node quadrilateral.

six_node_triangle or tri6
Six-node triangle.

element_set or *elem_set (string) [no default]*
User-given name of the parent element set.

surfaces (array) [no default]
List of element surfaces.

surface_sets (list) [={}]
List of surface set names (strings) used in the THERMAL_SHELL command. When using this option, the connectivity, shape and parent element of the surfaces are provided by the surface set container. It is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to the THERMAL_SHELL command. This option is used in place of directly specifying these parameters. In the event that both of the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements will be read, and a warning message will be issued. The surface_sets option is a preferred method to specify the surface elements. This option provides a support for mixed element topologies and simplifies pre-processing and post-processing.

num_shell_layers or *layers (integer) >0 [=1]*
Number of layers in the shell.

material_models (list) [={}]
List of material model names (strings) for the shell layers. Number of entries in the list must equal *num_shell_layers*.

shell_thickness_type (enumerated) [=constant]
Type of shell thickness.

`constant` *or* `const`
Constant shell thickness.

`shell_thicknesses` *or* `thick` *(array) [={}]*
Thicknesses of the shell layers. Number of entries in the array must equal `num_shell_layers`. Used with constant shell thickness type and for shell medium only.

`body_force` *(string) (string)*
User-given name of the volumetric body force to be applied to this element set. If none, no body force is applied.

> **Note:** Only heat source will be considered.

Description

This command creates zero physical thickness volumetric shell elements from surface elements. These elements are used for heat conduction analysis as solid medium only. Shell elements can have multiple layers, each with different thicknesses and material models. The `THERMAL_SHELL` command can also be provided together with two dynamic `surface_sets` which is used to set thermal boundary conditions on two sides of shells. Surface_set adjacent to the one given by the `surface_set` parameter is called `shell_inner`, while `surface_set` on the other side of the `THERMAL_SHELL` will be `shell_outer`.

```
THERMAL_SHELL( "pipes" ) {
surface_sets = {“pipe wall”}
num_shell_layers = 2
material_models = {"Aluminum",”Iron”}
shell_thicknesses = {0.001,0.002}
body_force = none
}
```

TURBULENCE_MODEL_PARAMETERS

Specifies constants and parameters for turbulence models.

Type

AcuSolve Command

Syntax

`TURBULENCE_MODEL_PARAMETERS {parameters...}`

Qualifier

This command has no qualifier.

Parameters

`spalart_allmaras_cb1` *(real) >=0 [=0.1355]*
Spalart-Allmaras model constant.

`spalart_allmaras_cb2` *(real) >=0 [=0.622]*
Spalart-Allmaras model constant.

`spalart_allmaras_sigma` *(real) >=0 [=0.66666666667]*
Spalart-Allmaras model constant.

`spalart_allmaras_cw1` *(real) >=0 [=3.23906781678]*
Spalart-Allmaras model constant.

`spalart_allmaras_cw2` *(real) >=0 [=0.3]*
Spalart-Allmaras model constant.

`spalart_allmaras_cw3` *(real) >=0 [=2]*
Spalart-Allmaras model constant.

`spalart_allmaras_cv1` *(real) >=0 [=7.1]*
Spalart-Allmaras model constant.

`spalart_allmaras_rotation_curvature` *(boolean) [=on]*
Turn on/off rotation-curvature correction (SA-RC).

`spalart_allmaras_cr1` *(real) >=0 [=1]*
Spalart-Allmaras rotation-curvature correction (SA-RC) constant.

`spalart_allmaras_cr2` *(real) >=0 [=12]*
Spalart-Allmaras rotation-curvature correction (SA-RC) constant.

`spalart_allmaras_cr3` *(real) >=0 [=1]*
Spalart-Allmaras rotation-curvature correction (SA-RC) constant.

`spalart_allmaras_qcr` *(boolean) [=off]*
Turn on/off Spalart-Allmaras Quadratic Constitutive Relation (SA-RC) model.

`spalart_allmaras_qcr_cr1` *(real) >=0 [=0.3]*
Spalart-Allmaras Quadratic Constitutive Relation (SA-RC) constant.

spalart_allmaras_qcr_cr2 (real) >=0 [=2.5]
Spalart-Allmaras Quadratic Constitutive Relation (SA-QCR) constant.

spalart_allmaras_des_type (enumerated) [=ddes]
Spalart-Allmaras Detached Eddy Simulation (SA-DES) model type.

des97	Spalart's Detached Eddy Simulation model.
ddes	Delayed DES model.
iddes	Improved Delayed DES (IDDES) model.

spalart_allmaras_des_constant (real) >=0 [=0.25]
SA-DES model constant applicable to des97 and ddes model types.

spalart_allmaras_iddes_c1 >=0 [=3.55] (real)
SA-IDDES model constant.

spalart_allmaras_iddes_ct >=0 [=1.63] (real)
SA-IDDES model constant.

spalart_allmaras_iddes_cw (real) >=0 [=0.15]
SA-IDDES model constant.

sst_a1 (real) >=0 [=0.31]
Shear-Stress Transport (SST) model constant.

sst_betastar (real) >=0 [=0.09]
SST model constant.

sst_sigmak1 (real) >=0 [=0.85]
SST model constant.

sst_sigmak2 (real) >=0 [=1]
SST model constant.

sst_sigmao1 (real) >=0 [=0.5]
SST model constant.

sst_sigmao2 (real) >=0 [=0.856]
SST model constant.

sst_beta1 (real) >=0 [=0.075]
SST model constant.

sst_beta2 (real) >=0 [=0.0828]
SST model constant.

sst_gamma1 (real) >=0 [=0.55555555556]
SST model constant.

sst_gamma2 (real) >=0 [=0.44]
SST model constant.

sst_des_constant_1 (real) >=0 [=0.78]
SST-DES model constant.

sst_des_constant_2 (real) >=0 [=0.61]
SST-DES model constant.

sst_production_type (enumerated) [=sst_2003]
SST Production term type.

sst_2003	SST model based on Menter et al., 2003.

sst_rotation_curvature (boolean) [=off]
Turn on/off rotation curvature correction for SST model.

sst_des_type (enumerated) [=ddes]
SST DES model type.

zonal	Zonal DES model.
ddes	Delayed DES model.
iddes	Improved Delayed DES (IDDES) model.

sst_wall_function_type (enumerated) [=all_yplus]
Wall function type.

all_yplus	Wall y+ independent wall function.

k_epsilon_cmu (real) >=0 [=0.09]
Standard K-Epsilon model constant.

k_epsilon_ce1 (real) >=0 [=1.44]
Standard K-Epsilon model constant.

k_epsilon_ce2 (real) >=0 [=1.92]
Standard K-Epsilon model constant.

k_epsilon_sigmak (real) >=0 [=1]
Standard K-Epsilon model constant.

k_epsilon_sigmae (real) >=0 [=1.3]
Standard K-Epsilon model constant.

rng_k_epsilon_cmu (real) >=0 [=0.0845]
ReNormalization Group (RNG) K-Epsilon model constant.

rng_k_epsilon_ce1 (real) >=0 [=1.42]
RNG K-Epsilon model constant.

rng_k_epsilon_ce2 (real) >=0 [=1.92]
RNG K-Epsilon model constant.

rng_k_epsilon_sigmak (real) >=0 [=0.7194]
RNG K-Epsilon model constant.

rng_k_epsilon_sigmae (real) >=0 [=0.7194]
RNG K-Epsilon model constant.

rng_k_epsilon_eta0 (real) >=0 [=4.38]
RNG K-Epsilon model constant.

rng_k_epsilon_beta (real) >=0 [=0.012]
RNG K-Epsilon model constant.

Realizable_k_epsilon_ce1 (real) >=0 [=0.43]
Realizable K-Epsilon model constant.

Realizable_k_epsilon_ce2 (real) >=0 [=1.9]
Realizable K-Epsilon model constant.

Realizable_k_epsilon_sigmak (real) >=0 [=1]
Realizable K-Epsilon model constant.

Realizable_k_epsilon_sigmae (real) >=0 [=1.2]
Realizable K-Epsilon model constant.

Realizable_k_epsilon_eta0 (real) >=0 [=5]
Realizable K-Epsilon model constant.

Realizable_k_epsilon_a0 (real) >=0 [=4.04]
Realizable K-Epsilon model constant.

k_epsilon_twolayer_restar (real) >=0 [=200]
K-Epsilon two-layer wall-function model constant controlling blending layer location.

k_epsilon_twolayer_redelta (real) >=0 [=20]
K-Epsilon two-layer wall-function model constant controlling blending thickness.

k_epsilon_wall_function_type (enumerated) (=two_layer)
Wall function type.

standard	Standard wall-function with first-layer y+ in the log-layer only.
two_layer	Wolfstein two-layer wall-function with all y+ first-layer height.

k_epsilon_realizability_constraint boolean [=on]
Turn on/off Durbin's realizability constraint. Corrects the stagnation point anomaly and improves robustness.

k_omega_alpha (real) >=0 [=0.52]
Wilcox k-omega model constant.

k_omega_betao (real) >=0 [=0.0708]
Wilcox k-omega model constant.

k_omega_clim (real) >=0 [=0.875]
Wilcox k-omega model constant.

k_omega_sigmak (real) >=0 [=0.6]
Wilcox k-omega model constant.

k_omega_sigmao (real) >=0 [=0.5]
Wilcox k-omega model constant.

k_omega_sigmado (real) >=0 [=0.125]
Wilcox k-omega model constant.

k_omega_rotation_curvature (boolean) [=off]
Turn on/off rotation-curvature correction for Wilcox k-omega model.

k_omega_des_constant (real) >=0 [=0.61]
K-omega DES model constants.

bsl_betastar (real) >=0 [=0.09]
Menter's baseline (BSL) K-omega model constant.

bsl_sigmak1 (real) >=0 [=0.5]
BSL K-omega model constant.

bsl_sigmak2 (real) >=0 [=1]
BSL K-omega model constant.

bsl_sigmao1 (real) >=0 [=0.5]
BSL K-omega model constant.

bsl_sigmao2 (real) >=0 [=0.856]
BSL K-omega model constant.

bsl_beta1 (real) >=0 [=0.075]
BSL K-omega model constant.

bsl_beta2 (real) >=0 [=0.0828]
BSL K-omega model constant.

bsl_gamma1 (real) >=0 [=0.55555555556]
BSL K-omega model constant.

bsl_gamma2 (real) >=0 [=0.44]
BSL K-omega model constant.

smagorinsky_constant (real) >=0 [=0.04]
Smagorinsky constant used in Large Eddy Simulation (LES) model.

dynamic_subgrid_model_type (enumerated) [=mij]
Dynamic Subgrid Model type.

kolmogorov	Kolmogorov model.
mij	Mij model.

synthetic_turbulence_input_type or synles (enumerated) (=none)
Large Eddy Simulation (LES) synthetic turbulence inflow type.

none	Deactivate synthetic turbulence flow.
k_omega	Provide k profiles and omega profiles at synthetic inflow.
k_epsilon	Provide k profiles and epsilon profiles at synthetic inflow.

htc_method or htc (enumerated) (=turbulence_wall)
This parameter defines the method used to calculate convective heat transfer coefficient (*htc*) or *surface_film_coefficient*.

user_ref_temperature	Calculates the heat transfer coefficient using a user-specified constant reference temperature. The reference temperature is provided by *user_reference_temperature*. This option requires *user_reference_temperature* and *htc_limit_reference_temp*.
direct	Calculates the heat transfer coefficient by evaluating the nodal temperature at the surface and at a specified distance from the surface. This option utilizes *film_coefficient_yplus* to determine the location within the domain for evaluating the reference temperature. This option requires *film_coefficient_yplus* and *htc_limit_reference_temp*.
turbulence_wall	Utilizes the thermal wall function to determine the surface film coefficient. This option requires *film_coefficient_yplus* and *htc_limit_reference_temp*.

user_reference_temperature or *user_ref_temp* *(real) > 0 [=300.0]*
: User-provided reference temperature. Used with *htc_method* = user_ref_temperature.

htc_limit_reference_temp or *htc_ref_limit* *(boolean) [=on]*
: Flag specifying whether the reference temperature should be limited to the minimum or maximum values present in the nodal solution. Used with *htc_method* = user_ref_temperature or *htc_method* = direct or *htc_method* = turbulence_wall.

film_coefficient_yplus *(real) > 0 [=100]*
: Value of y+ used to evaluate non-dimensional temperature in *surface_film_coefficient*. Used with *htc_method* = direct or *htc_method* = turbulence_wall.

von_karman_constant *(real) >=0 [=0.41]*
: Von Kármán constant used in logarithmic law describing a turbulent boundary layer's velocity profile.

intermittency_ctu1 *(real) [=100]*
: Intermittency equation constant for Gamma transition model.

intermittency_ctu2 *(real) [=1000]*
: Intermittency equation constant for Gamma transition model.

intermittency_ctu3 *(real) [=1]*
: Intermittency equation constant for Gamma transition model.

intermittency_cpg1 *(real) [=14.68]*
: Intermittency equation constant for Gamma transition model.

intermittency_cpg2 *(real) [=-7.34]*
: Intermittency equation constant for Gamma transition model.

intermittency_cpg3 *(real) [=0]*
: Intermittency equation constant for Gamma transition model.

retheta_critical_type *(enumerated) [=default]*
: Critical Re-Theta correlation type.

default	Default value.
user_function	User-defined function.

retheta_critical_user_function (string) [=]
Name of the user-defined function. Used when *retheta_critical_type* = user_function.

retheta_critical_user_values (array) [={ }]
Array of values to be passed to the user-defined function. Used with *retheta_critical_user_function type*.

retheta_critical_user_strings (array) [={ }]
Array of strings to be passed to the user-defined function. Used with *retheta_critical_user_function type*.

flength_type (enumerated) [=default]
F-Length correlation type.

default	Default value.
user_function	User-defined function.

flength_user_function (string) [=]
Name of the user-defined function. Used when *flength_type* = user_function.

flength_user_values (array) [={ }]
Array of values to be passed to the user-defined function. Used with *flength_user_function* type.

flength_user_strings (array) [={ }]
Array of strings to be passed to the user-defined function. Used with *flength_user_function* type.

Description

This command modifies the turbulence model constants and parameters to the user desired quantity or option from the default selection. For example,

```
TURBULENCE_MODEL_PARAMETERS {
    k_epsilon_cmu = 0.1
}
```

modifies the standard k-epsilon turbulence model constant cμ from its default value of 0.09 to 0.1.

Similarly, for Boolean options, example:

```
TURBULENCE_MODEL_PARAMETERS {
    sst_rotation_curvature = on
}
```

turns on rotation-curvature correction for the SST model.

The synthetic turbulence feature accepts two types of inflow profiles: K-Omega and K-Epsilon. For example,

```
TURBULENCE_MODEL_PARAMETERS {
```

```
    Synthetic_turbulence_input_type = k_omega
}
```

This command is only active when *large_eddy_simulation* or *dynamic_model* is chosen at the turbulence command and inflow type is set to velocity or atmospheric. If the k_omega option is set, then corresponding turbulent kinetic energy and turbulent eddy frequency profiles (or values) need to be specified at inflow to generate synthetic turbulence. If k-epsilon is chosen, then turbulent kinetic energy and turbulent dissipation rate profiles (or values) are required to be set at inflow.

htc_method provides three options to compute the Heat Transfer Coefficient (HTC).

```
TURBULENCE_MODEL_PARAMETERS {
    htc_method = user_ref_temperature
    htc_limit_reference_temp = on
    user_reference_temperature = 300
}
```

When *htc_method* is set to user_ref_temperature, AcuSolve calculates the heat transfer coefficient using a user-specified constant reference temperature, denoted as *user_reference_temperature*. Additionally, *htc_limit_reference_temp* determines whether the reference temperature should be limited to the minimum or maximum values present in the nodal, ensuring that the heat transfer coefficient (*surface_filim_coefficient*) remains positive.

$$\mathrm{htc}_1 = q/(T_{ref} - T_{wall}) \tag{42}$$

where q is the local surface heat flux, T_{wall} is the local wall temperature, T_{ref} is the user-specified reference temperature (must be smaller than the minimum of local wall temperature (T_{wall}).

```
TURBULENCE_MODEL_PARAMETERS {
    htc_method = direct
    htc_limit_reference_temp = on
    film_coefficient_yplus = 100
}
```

When *htc_method* is set to direct, AcuSolve calculates the heat transfer coefficient by evaluating the nodal temperature at the surface and at a specified distance away from the surface. This method utilizes *film_coefficient_yplus* to determine the location where the reference temperature is evaluated. Additionally, the *htc_limit_reference_temp* option specifies whether the reference temperature should be constrained to the minimum or maximum values present in the nodal solution, ensuring that the heat transfer coefficient (*surface_filim_coefficient*) remains positive.

$$\mathrm{htc}_2 = q/(T_{\mathrm{ref_local}} - T_{wall}) \tag{43}$$

where $T_{\mathrm{ref_local}}$ is the local reference temperature at user-specified yplus *film_coefficient_yplus*.

```
TURBULENCE_MODEL_PARAMETERS {
    htc_method = turbulence_wall
    htc_limit_reference_temp = on
    film_coefficient_yplus = 100
}
```

When *htc_method* is set to turbulence_wall, AcuSolve uses *film_coefficient_yplus* to calculate the surface film coefficient (heat transfer coefficient) by applying the thermal wall function. Additionally, the *htc_limit_reference_temp* option specifies whether the reference temperature should be constrained

to the minimum or maximum values present in the nodal solution, ensuring that the heat transfer coefficient (*surface_filim_coefficient*) remains positive.

$$\mathrm{htc}_3 = (\rho\, C_p\, U_T) \,/\, T^+ \tag{44}$$

where ρ is the local density, C_p is the local specific heat, $U_T = \sqrt{\frac{T_{\mathrm{wall}}}{\rho}}$ is the local friction velocity (wall shear stress divided by density). T^+ is the non-dimensional temperature.

AcuSolve thermal wall function is

$$T^+ = \mathrm{Pr}\, y^+ e^{-L} + (2.12 \ln(1+y^+) + \beta) e^{-1/L} \tag{45}$$

where $L = \frac{0.01\,(\mathrm{Pr}\, y^+)^4}{1+5\,\mathrm{Pr}^3 y^+}$

$$\beta = \left(3.85\mathrm{Pr}^{1/3} - 1.3\right)^2 + 2.12 \ln(\mathrm{Pr}) \tag{46}$$

$\mathrm{Pr} = \frac{\mu\, C_p}{k}$ the local Prandtl number (Pr) is a function of viscosity μ, specific heat capacity C_p, and conductivity k.

$T^+ = \frac{T_w - T}{T^*}$ non-dimensional temperature.

T_w is the wall temperature, $T^* = q \,/\, (\rho\, C_p\, u^*)$ is the friction temperature.

> **Note:** When using the htc methods direct or user_ref_temperature, AcuTherm will print the results corresponding to direct or user_ref_temperature, rather than turb_wall. It is essential to ensure that turb_wall is selected for *htc_method* to ensure that AcuTherm prints the correct values.

USER_GLOBAL_DATA

Defines and initializes globally-accessible data.

Type

AcuSolve Command

Syntax

USER_GLOBAL_DATA ("name") {parameters...}

Qualifier

User-given name.

Parameters

value (real) [=0]

The value of the user global data.

Description

This command defines and initializes a variable that is accessible and changeable from a number of places. For example,

```
USER_GLOBAL_DATA( "var_1" ) {
value = 10
}
```

defines a variable called "var_1" and initializes it to the value 10. This variable may then be accessed or changed by a user-defined function, and the changed value accessed by another user-defined function. It may be used by a multiplier function in two ways: either as the return value of the multiplier function itself, or as the value of the curve fit variable for one of the curve fit types. See the MULTIPLIER_FUNCTION command for details. The variable may also be changed on the command line during a run through the acuSig utility, for example,

```
acuSig -data "var_1 = 332"
```

WAVE_BOUNDARY_CONDITION

Specifies a wave generator boundary condition.

Type

AcuSolve Command

Syntax

WAVE_BOUNDARY_CONDITION("name") {parameters...}

Qualifier

User-given name.

Parameters

type (enumerated) (=airy)

Type of wave to be generated.

none	No wave is generated.
airy	Linear Airy wave model.

height (real) (=0)

Wave height or amplitude.

length (real) (=0)

Wave length.

depth (real) (=0)

Depth of the wave column assumed where the wave is generated.

direction (array) (={1,0,0})

Direction of propagation of the generated wave.

Description

This command is used to define a wave boundary condition that generates a surface wave in a multiphase flow problem. This boundary condition overrides the velocity and field boundary condition. The *wave_boundary_condition* entry is referenced by *simple_boundary_condition*. An example is provided below:

```
WAVE_BOUNDARY_CONDITION( "wave_1" ) {
 type       = airy
 height      = 0.1
 length      = 4.0
 depth      = 10.0
 direction     = {1.0,0.0,0.0}
}
```

The corresponding reference in a *simple_boundary_condition* along with wave related settings would look like the following:

```
SIMPLE_BOUNDARY_CONDITION( "Inlet" ) {
...
 waves                              = {"wave_1"}
 heavy_fluid_velocity  = {1.0,0.0,0.0}
 light_fluid_velocity  = {0.0,0.0,0.0}
 hydrostatic_pressure  = on
 hydrostatic_pressure_origin = {0.0,0.0,0.0}
...
}
```

WAVE_DAMPING_SOURCE

Specifies a volumetric momentum source term to damp surface waves.

Type

AcuSolve Command

Syntax

WAVE_DAMPING_SOURCE {parameters...}

Qualifier

This command has no qualifier.

Parameters

wave_damping_type (enumerated) (=none)

Type of wave damping to be applied. If none, no damping is applied.

none	No damping.
linear	Linear damping model is applied.

wave_damping_distance or *dist real (=0)*

Distance from the outlet surface where the damping source term is active.

wave_damping_coefficient or *coef real (=10)*

Coefficient multiplying the damping source term.

Description

This command is used to damp a surface wave in a multiphase flow problem by adding momentum source per unit volume to the momentum equation along the direction of gravity. The momentum source per unit volume term takes the following form:

$$S(x) = -C\rho|U_Z|\left(\frac{x-x_O}{x_D}\right)^2 \quad (47)$$

where C is the wave damping coefficient, ρ is the density, U_Z is the velocity component along gravity, x is the local coordinate where the source term is being applied, x_D and x_0 are the wave damping distance and the coordinate of the outlet surfaces, respectively.

Wave damping with a linear model can be specified as follows:

```
WAVE_DAMPING_SOURCE {
    wave_damping_type          = linear
    wave_damping_distance  = 5
    wave_damping_coefficient     = 10
}
```

Optionally, the outlet surface can be excluded from the damping zone calculation by setting the boolean *wave_damping* to "off" within the SIMPLE_BOUNDARY_CONDITION command.

Solution Strategy Commands

9

The solution strategy of a problem is specified by the commands in this section.

This chapter covers the following:

- ALGEBRAIC_MULTIGRID_PARAMETERS (p. 147)
- AUTO_SOLUTION_STRATEGY (p. 170)
- CONVERGENCE_CHECK_PARAMETERS (p. 177)
- LINEAR_SOLVER_PARAMETERS (p. 191)
- STAGGER (p. 198)
- TIME_ INCREMENT (p. 210)
- TIME_INTEGRATION (p. 215)
- TIME_SEQUENCE (p. 219)

Either use the `AUTO_SOLUTION_STRATEGY` command which issues all the other commands in this section or use those commands individually. If the `AUTO_SOLUTION_STRATEGY` command is not used then the `TIME_SEQUENCE` and `STAGGER` commands must be explicitly given in an input file. All others are optional; if they are not given, their default values are used. Explanations of the relationships between these commands and the overall time stepping and nonlinear iterative strategies are given in the `TIME_SEQUENCE` command.

ALGEBRAIC_MULTIGRID_PARAMETERS

Specifies parameters for the algebraic multigrid algorithm.

Type

AcuSolve Command

Syntax

ALGEBRAIC_MULTIGRID_PARAMETERS {parameters}

Qualifier

This command has no qualifier.

Parameters

pressure_negative_coupling_tolerance *(real) >=0 <=1 [=0.6]*

Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly negatively coupled variables for the pressure matrix are considered. Tolerance is the decimal fraction of the maximum value. The default value of 0.6 sets the tolerance to use values of 60 percent or greater of the most negative value. With a value of 1.0, negative coupling is ignored. Negative coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly negatively coupled if the negated value of the matrix entry exceeds the maximum negated value of the matrix multiplied by the negative coupling tolerance. The value of *pressure_negative_coupling_tolerance* can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

pressure_positive_coupling_tolerance *(real) >=0 <=1 [=1.0]*

Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly positively coupled variables for the pressure matrix are considered. Tolerance is the decimal fraction of the maximum value. With the default value of 1.0, positive coupling is ignored. Positive coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly positively coupled if the value of the matrix entry exceeds the maximum value of the matrix multiplied by the positive coupling tolerance. The value of *pressure_positive_coupling_tolerance* can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

max_pressure_final_matrix *(integer) >=1 [=100]*

Maximum number of entries present in the coarsest matrix of the AMG algorithm. With the default value of 100, coarsening will stop when the lowest level matrix is 10 by 10 or smaller. Increasing the size of this parameter may increase the time necessary to obtain a solution on the lowest level matrix.

pressure_standard_interpolation (boolean) [=on]

Interpolation method used to build the coarse multigrid matrices from fine matrices or to transfer the solution from a fine level to a coarse multigrid level, and vice versa. With this option set to off, direct interpolation will be used. Each equation in the fine level matrix that is considered to be important in the coarsening process gets interpolated to build the coarse level matrix. In direct interpolation, just neighboring points within the one fine level equation are used to calculate the interpolation weights. In standard interpolation, in addition to the neighboring points, other coupled equations are also considered in the weight calculations. More information is incorporated in the standard interpolation, leading to better convergence. In most cases, standard interpolation is recommended.

pressure_positive_negative_separate (boolean) [=off]

Determines if the positive and negative weights are separated during either standard or direct interpolation. In either direct or standard interpolation, weights are functions of off diagonal components of the fine level matrix and can be positive or negative. Interpolation can be conducted either separately for positive and negative weights, or these weights can be added together to have one interpolation. With this option set to on, summation of positive and negative weights will be calculated and used for interpolation. For pressure matrices, separation of weights is not recommended.

pressure_truncated_interpolation (boolean) [=on]

Determines if the small interpolation weights will be truncated when interpolating the fine level multigrid matrix to the coarse level matrix or when transferring the solution from the coarse matrix to the finer AMG matrix levels. This flag should be used together with the *pressure_truncation_tolerance* to set the tolerance to determine the small weights. Truncation can be used with both direct and standard interpolation. A minor increase in the computational time to reach convergence can be observed if this flag is set to off.

pressure_truncation_tolerance (real) >=0 <=1 [=0.1]

Sets the portion of the smallest interpolation weights to be truncated during interpolation from a fine level to a coarse level. Used when *pressure_truncated_interpolation*=on. With a value of zero, no interpolation weights will be truncated. With a value of one, all weights will be truncated.

pressure_givens_scaling (boolean) [=on]

Determines if two by two Givens matrix scaling is used for the multigrid smoothing process. With this option turned off, diagonal scaling is used.

pressure_smoothing_type (enumerated) [=chebyshev]

Smoothing method used at each grid level after interpolation when values are from a fine matrix to a coarse matrix, or from a coarse matrix to a fine matrix.

Chebyshev	Chebyshev polynomial smoothing. Chebychev smoothing provides better performance, accurate convergence with less computational time, and is recommended for most matrices.
Jacobi	Jacobi smoothing. Jacobi smoothing is recommended for the fully coupled flow stagger.

pressure_eigenvalue_tolerance (real) >0 [=1.e-2]

Tolerance to stop the eigenvalue calculation iterations. Used with *pressure_smoothing_type*=chebyshev. Chebyshev polynomial smoothing requires the ratio of

the largest to the smallest eigenvalues of the matrix. The Iterative Lanczos algorithm is used to calculate the maximum eigenvalue of the pressure matrix.

max_pressure_eigenvalue_iterations (integer) >=0 [=20]
Used with *pressure_smoothing_type*=chebyshev. Maximum number of iterations of the Lanczos algorithm for computing the largest eigenvalues at each multigrid level. The *pressure_eigenvalue_tolerance* and *max_pressure_eigenvalue_iterations* determine when iterations stop. Whichever constraint is met first stops the algorithm. In general, eigenvalues converge within 20 iterations. For cases where the eigenvalues do not converge, increasing the total number of iterations can lead to better estimates of the eigenvalues at the expense of increased total computation time.

> **Note:** Poor estimates of eigenvalues also increase solution time by slowing the convergence of the AMG algorithm.

pressure_smoothing_order (integer) >=1 [=2]
When *pressure_smoothing_type*=chebyshev, this parameter sets the polynomial order for the smoother. When pressure_smoothing_type=jacobi this parameter sets the number of smoothing passes during the downward and upward multigrid cycles. For both Chebyshev and Jacobi methods, increasing the smoothing order will reduce the total number of linear solver iterations required to converge the system. However, the time per iteration will also increase. There should be a balance between the number of iterations and the time per iteration. For pressure matrices, a value of two is found to give the best performance.

pressure_chebyshev_max_min_ratio (real) >1 <=100 [=10]
When *pressure_smoothing_type*=chebyshev, this parameter provides an estimate of the smallest eigenvalue by dividing the largest eigenvalue calculated by the Lanczos algorithm by this parameter. Chebyshev polynomial smoothing requires the ratio of the largest to the smallest eigenvalues of the matrix to calculate the polynomial coefficients. The smallest value may not be calculated. This parameter can be used to get an estimate. The default of 10 means that the smallest eigenvalue is one tenth of the largest.

pressure_jacobi_relaxation_factor (real) >0 [=0.25]
Used when *pressure_smoothing_type*=jacobi. Jacobi smoothing may not be stable and it should be relaxed. Conventional under relaxation is used to stabilize it. The default of 0.25 means updated values will be calculated by adding 25 percent of new values to 75 percent of the old values.

pressure_setup_tolerance (real) >0 [=0]
The nonlinear convergence tolerance used to control when the AMG setup process is performed. If the residual is higher than this value, the AMG setup process is performed. If the residual is lower than this tolerance, the AMG algorithm uses the existing setup information that is stored in RAM. With the default value of zero, the AMG setup process will occur every time the equations are solved, regardless of the residual. As the simulation converges, the change in solution between time steps decreases, minimizing the need to execute the AMG setup process. If AMG is used for more than one equation, such as pressure and turbulence, and this parameter is greater than zero, the memory requirements will be increased.

> **Note:** For each equation, a different setup tolerance can be used. This value should not be changed unless a convergence tolerance is set for the problem that is less than the default value.

num_pressure_global_basis (integer) >=0 [=0]
This parameter was retired in the 13.0 release.

pressure_global_basis_tolerance (real) >0 [=1.e-6]
This parameter was retired in the 13.0 release.

max_pressure_global_basis_iterations (integer) >0 [=1000]
This parameter was retired in the 13.0 release.

num_pressure_initial_givens_rotations (integer) >=0 [=0]
This parameter was retired in the 13.0 release.

velocity_negative_coupling_tolerance (real) >=0 <=1 [=0.5]
Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly negatively coupled variables for the velocity matrix are considered. Tolerance is the decimal fraction of the maximum value. The default value of 0.5 sets the tolerance to use values of 50 percent or greater of the most negative value. With a value of 1.0, negative coupling is ignored. Negative coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly negatively coupled if the negated value of the matrix entry exceeds the maximum negated value of the matrix multiplied by the negative coupling tolerance. The value of *velocity_negative_coupling_tolerance* can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

velocity_positive_coupling_tolerance (real) >=0 <=1 [=1.0]
Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly positively coupled variables for the velocity matrix are considered. Tolerance is the decimal fraction of the maximum value. With the default value of 1.0, positive coupling is ignored. Positive coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly positively coupled if the value of the matrix entry exceeds the maximum value of the matrix multiplied by the positive coupling tolerance. The value of *velocity_positive_coupling_tolerance* can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

max_velocity_final_matrix (integer) >=1 [=100]
Maximum number of entries present in the coarsest matrix of the AMG algorithm. With the default value of 100, coarsening will stop when the lowest level matrix is 10 by 10 or smaller. Increasing the size of this parameter may lead to longer duration of the solution calculation of the lowest level matrix.

velocity_standard_interpolation (boolean) [=on]
Interpolation method used to build the coarse multigrid matrices from fine matrices or to transfer the solution from a fine level to a coarse multigrid level, and vice versa. With this option set to

off, direct interpolation will be used. Each equation in the fine level matrix that is considered to be important in the coarsening process gets interpolated to build the coarse level matrix. In direct interpolation, just neighboring points within the one fine level equation is used to calculate the interpolation weights. In standard interpolation, in addition to the neighboring points, other coupled equations are also considered in the weight calculations. More information is incorporated in the standard interpolation, leading to better convergence. In most cases, standard interpolation is recommended.

velocity_positive_negative_separate (boolean) [=off]
Determines if the positive and negative weights are separated during either standard or direct interpolation. In either direct or standard interpolation, weights are functions of off diagonal components of the fine level matrix and can be positive or negative. Interpolation can be conducted either separately for positive and negative weights, or these weights can be added together to have one interpolation. With this option set to on, summation of positive and negative weights will be calculated and used for interpolation. For velocity matrices, separation of weights is not recommended.

velocity_truncated_interpolation (boolean) [=on]
Determines if the small interpolation weights will be truncated when interpolating the fine level multigrid matrix to the coarse level matrix or when transferring the solution from the coarse matrix to the finer AMG matrix levels. This flag should be used together with the *velocity_truncation_tolerance* to set the tolerance to determine the small weights. Truncation can be used with both direct and standard interpolation. A minor increase in the computational time to reach convergence can be observed if this flag is set to off.

velocity_truncation_tolerance (real) >=0 <=1 [=0.1]
Sets the portion of the smallest interpolation weights to be truncated during interpolation from a fine level to a coarse level. Used when *velocity_truncated_interpolation*=on. With a value of zero, no interpolation weights will be truncated. With a value of one, all weights will be truncated.

velocity_givens_scaling (boolean) [=off]
Determines if two by two Givens matrix scaling is used for the multigrid smoothing process. With this option turned off, diagonal scaling is used.

velocity_smoothing_type (enumerated) [=chebyshev]
Smoothing method used at each grid level after interpolation when values are from a fine matrix to a coarse matrix, or from a coarse matrix to a fine matrix.

Chebyshev	Chebyshev polynomial smoothing. Chebychev smoothing provides better performance, accurate convergence with less computational time, and is recommended for most matrices.
Jacobi	Jacobi smoothing. Jacobi smoothing is recommended for the fully coupled flow stagger.

velocity_num_krylov_vectors (integer) >=0 [=30]
Used with *velocity_smoothing_type*=chebyshev. Number of Krylov vectors in the Arnoldi algorithm for computing the largest eigenvalues at each multigrid level. In general, eigenvalues converge within 30 Krylov vectors. For cases where the eigenvalues do not converge, increasing the number of Krylov vectors can lead to better estimates of the eigenvalues at the expense of increased total computation time.

> **Note:** Poor estimates of eigenvalues also increase solution time by slowing the convergence of the AMG algorithm.

velocity_smoothing_order (integer) >=1 [=2]

When *velocity_smoothing_type*=chebyshev, this parameter sets the polynomial order for the smoother. When *velocity_smoothing_type*=jacobi this parameter sets the number of smoothing passes during the downward and upward multigrid cycles. For both Chebyshev and Jacobi methods, increasing the smoothing order will reduce the total number of linear solver iterations required to converge the system. However, the time per iteration will also increase. There should be a balance between the number of iterations and the time per iteration. For velocity matrices, a value of two is found to give the best performance.

velocity_chebyshev_max_min_ratio (real) >1 <=100 [=10]

When *velocity_smoothing_type*=chebyshev, this parameter provides an estimate of the smallest eigenvalue by dividing the largest eigenvalue calculated by the Lanczos algorithm by this parameter. Chebyshev polynomial smoothing requires the ratio of the largest to the smallest eigenvalues of the matrix to calculate the polynomial coefficients. The smallest value may not be calculated. This parameter can be used to get an estimate. The default of 10 means that the smallest eigenvalue is one tenth of the largest.

velocity_jacobi_relaxation_factor (real) >0 [=0.25]

Used when *velocity_smoothing_type*=jacobi. Jacobi smoothing may not be stable and it should be relaxed. Conventional under relaxation is used to stabilize it. The default of 0.25 means updated values will be calculated by adding 25 percent of new values to 75 percent of the old values.

velocity_setup_tolerance (real) >0 [=0]

The nonlinear convergence tolerance used to control when the AMG setup process is performed. If the residual is higher than this value, the AMG setup process is performed. If the residual is lower than this tolerance, the AMG algorithm uses the existing setup information that is stored in RAM. With the default value of zero, the AMG setup process will occur every time the equations are solved, regardless of the residual. As the simulation converges, the change in solution between time steps decreases, minimizing the need to execute the AMG setup process. If AMG is used for more than one equation, such as velocity and turbulence, and this parameter is greater than zero, the memory requirements will be increased.

> **Note:** For each equation, a different setup tolerance can be used. This value should not be changed unless a convergence tolerance is set for the problem that is less than the default value.

flow_negative_coupling_tolerance (real) >=0 <=1 [=0.5]

Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly negatively coupled variables for the flow matrix are considered. Tolerance is the decimal fraction of the maximum value. The default value of 0.5 sets the tolerance to use values of 50 percent or greater of the most negative value. With a value of 1.0, negative coupling is ignored. Negative coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are

considered to be highly negatively coupled if the negated value of the matrix entry exceeds the maximum negated value of the matrix multiplied by the negative coupling tolerance. The value of *flow_negative_coupling_tolerance* can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

flow_positive_coupling_tolerance (real) >=0 <=1 [=1.0]
Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly positively coupled variables for the flow matrix are considered. Tolerance is the decimal fraction of the maximum value. With the default value of 1.0, positive coupling is ignored. Positive coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly positively coupled if the value of the matrix entry exceeds the maximum value of the matrix multiplied by the positive coupling tolerance. The value of *flow_positive_coupling_tolerance* can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

max_flow_final_matrix (integer) >=1 [=100]
Maximum number of entries present in the coarsest matrix of the AMG algorithm. With the default value of 100, coarsening will stop when the lowest level matrix is 10 by 10 or smaller. Increasing the size of this parameter may lead to longer duration of the solution calculation of the lowest level matrix.

flow_standard_interpolation (boolean) [=off]
Interpolation method used to build the coarse multigrid matrices from fine matrices or to transfer the solution from a fine level to a coarse multigrid level, and vice versa. With this option set to off, direct interpolation will be used. Each equation in the fine level matrix that is considered to be important in the coarsening process gets interpolated to build the coarse level matrix. In direct interpolation, just neighboring points within the one fine level equation are used to calculate the interpolation weights. In standard interpolation, in addition to the neighboring points, other coupled equations are also considered in the weight calculations. More information is incorporated in the standard interpolation, leading to better convergence. In most cases, standard interpolation is recommended. However, direct is recommended for the fully coupled flow stagger.

flow_positive_negative_separate (boolean) [=off]
Determines if the positive and negative weights are separated during either standard or direct interpolation. In either direct or standard interpolation, weights are functions of off diagonal components of the fine level matrix and can be positive or negative. Interpolation can be conducted either separately for positive and negative weights, or these weights can be added together to have one interpolation. With this option set to on, summation of positive and negative weights will be calculated and used for interpolation. For flow matrices, separation of weights is not recommended.

flow_truncated_interpolation (boolean) [=on]
Determines if the small interpolation weights will be truncated when interpolating the fine level multigrid matrix to the coarse level matrix or when transferring the solution from the coarse matrix to the finer AMG matrix levels. This flag should be used together with the *flow_truncation_tolerance* to set the tolerance to determine the small weights. Truncation can be used with both direct and standard interpolation. A minor increase in the computational time to reach convergence can be observed if this flag is set to off.

flow_truncation_tolerance (real) >=0 <=1 [=0.1]
Sets the portion of the smallest interpolation weights to be truncated during interpolation from a fine level to a coarse level. Used when *flow_truncated_interpolation*=on. With a value of zero, no interpolation weights will be truncated. With a value of one, all weights will be truncated.

flow_givens_scaling (boolean) [=off]
Determines if two by two Givens matrix scaling is used for the multigrid smoothing process. With this option turned off, diagonal scaling is used.

flow_smoothing_type (enumerated) [=jacobi]
Smoothing method used at each grid level after interpolation when values are from a fine matrix to a coarse matrix, or from a coarse matrix to a fine matrix.

Chebyshev	Chebyshev polynomial smoothing. Chebychev smoothing provides better performance, accurate convergence with less computational time, and is recommended for most matrices.
Jacobi	Jacobi smoothing. Jacobi smoothing is recommended for the fully coupled flow stagger.

flow_num_krylov_vectors (integer) >=0 [=30]
Used with *flow_smoothing_type*=chebyshev. Number of Krylov vectors in the Arnoldi algorithm for computing the largest eigenvalues at each multigrid level. In general, eigenvalues converge within 30 Krylov vectors. For cases where the eigenvalues do not converge, increasing the number of Krylov vectors can lead to better estimates of the eigenvalues at the expense of increased total computation time.

> **Note:** Poor estimates of eigenvalues also increase solution time by slowing the convergence of the AMG algorithm.

flow_smoothing_order (integer) >=1 [=2]
When *flow_smoothing_type*=chebyshev, this parameter sets the polynomial order for the smoother. When *flow_smoothing_type*=jacobi this parameter sets the number of smoothing passes during the downward and upward multigrid cycles. For both Chebyshev and Jacobi methods, increasing the smoothing order will reduce the total number of linear solver iterations required to converge the system. However, the time per iteration will also increase. There should be a balance between the number of iterations and the time per iteration. For flow matrices, a value of two is found to give the best performance.

flow_chebyshev_max_min_ratio (real) >1 <=100 [=10]
When *flow_smoothing_type*=chebyshev, this parameter provides an estimate of the smallest eigenvalue by dividing the largest eigenvalue calculated by the Lanczos algorithm by this parameter. Chebyshev polynomial smoothing requires the ratio of the largest to the smallest eigenvalues of the matrix to calculate the polynomial coefficients. The smallest value may not be calculated. This parameter can be used to get an estimate. The default of 10 means that the smallest eigenvalue is one tenth of the largest.

flow_jacobi_relaxation_factor (real) >0 [=0.2]
Used when *flow_smoothing_type*=jacobi. Jacobi smoothing may not be stable and it should be relaxed. Conventional under relaxation is used to stabilize it. The default of 0.2 means updated values will be calculated by adding 20 percent of new values to 80 percent of the old values.

flow_setup_tolerance (real) >0 [=0]
The nonlinear convergence tolerance used to control when the AMG setup process is performed. If the residual is higher than this value, the AMG setup process is performed. If the residual is lower than this tolerance, the AMG algorithm uses the existing setup information that is stored in RAM. With the default value of zero, the AMG setup process will occur every time the equations are solved, regardless of the residual. As the simulation converges, the change in solution between time steps decreases, minimizing the need to execute the AMG setup process. If AMG is used for more than one equation, such as flow and turbulence, and this parameter is greater than zero, the memory requirements will be increased.

> **Note:** For each equation a different setup tolerance can be used. This value should not be changed unless a convergence tolerance is set for the problem that is less than the default value.

temperature_negative_coupling_tolerance (real) >=0 <=1 [=0.5]
Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly negatively coupled variables for the temperature matrix are considered. Tolerance is the decimal fraction of the maximum value. The default value of 0.5 sets the tolerance to use values of 50 percent or greater of the most negative value. With a value of 1.0, negative coupling is ignored. Negative coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly negatively coupled if the negated value of the matrix entry exceeds the maximum negated value of the matrix multiplied by the negative coupling tolerance. The value of *temperature_negative_coupling_tolerance* can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

temperature_positive_coupling_tolerance (real) >=0 <=1 [=1.0]
Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly positively coupled variables for the temperature matrix are considered. Tolerance is the decimal fraction of the maximum value. With the default value of 1.0, positive coupling is ignored. Positive coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly positively coupled if the value of the matrix entry exceeds the maximum value of the matrix multiplied by the positive coupling tolerance. The value of *temperature_positive_coupling_tolerance* can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

max_temperature_final_matrix (integer) >=1 [=100]
Maximum number of entries present in the coarsest matrix of the AMG algorithm. With the default value of 100, coarsening will stop when the lowest level matrix is 10 by 10 or smaller. Increasing the size of this parameter may lead to longer duration of the solution calculation of the lowest level matrix.

temperature_standard_interpolation (boolean) [=on]
Interpolation method used to build the coarse multigrid matrices from fine matrices or to transfer the solution from a fine level to a coarse multigrid level, and vice versa. With this option set to off, direct interpolation will be used. Each equation in the fine level matrix that is considered to be important in the coarsening process gets interpolated to build the coarse level matrix. In direct interpolation, just neighboring points within the one fine level equation are used to calculate the interpolation weights. In standard interpolation, in addition to the neighboring points, other coupled equations are also considered in the weight calculations. More information is incorporated in the standard interpolation, leading to better convergence. In most cases, standard interpolation is recommended.

temperature_positive_negative_separate (boolean) [=off]
Determines if the positive and negative weights are separated during either standard or direct interpolation. In either direct or standard interpolation, weights are functions of off diagonal components of the fine level matrix and can be positive or negative. Interpolation can be conducted either separately for positive and negative weights, or these weights can be added together to have one interpolation. With this option set to on, summation of positive and negative weights will be calculated and used for interpolation. For temperature matrices, separation of weights is not recommended.

temperature_truncated_interpolation (boolean) [=on]
Determines if the small interpolation weights will be truncated when interpolating the fine level multigrid matrix to the coarse level matrix or when transferring the solution from the coarse matrix to the finer AMG matrix levels. This flag should be used together with the *temperature_truncation_tolerance* to set the tolerance to determine the small weights. Truncation can be used with both direct and standard interpolation. A minor increase in the computational time to reach convergence can be observed if this flag is set to off.

temperature_truncation_tolerance (real) >=0 <=1 [=0.1]
Sets the portion of the smallest interpolation weights to be truncated during interpolation from a fine level to a coarse level. Used when *temperature_truncated_interpolation*=on. With a value of zero, no interpolation weights will be truncated. With a value of one, all weights will be truncated.

temperature_givens_scaling (boolean) [=off]
Determines if two by two Givens matrix scaling is used for the multigrid smoothing process. With this option turned off, diagonal scaling is used.

temperature_smoothing_type (enumerated) [=chebyshev]
Smoothing method used at each grid level after interpolation when values are from a fine matrix to a coarse matrix, or from a coarse matrix to a fine matrix.

Chebyshev	Chebyshev polynomial smoothing. Chebychev smoothing provides better performance, accurate convergence with less computational time, and is recommended for most matrices.
Jacobi	Jacobi smoothing. Jacobi smoothing is recommended for the fully coupled flow stagger.

temperature_num_krylov_vectors (integer) >=0 [=30]

Used with *temperature_smoothing_type*=chebyshev. Number of Krylov vectors in the Arnoldi algorithm for computing the largest eigenvalues at each multigrid level. In general, eigenvalues converge within 30 Krylov vectors. For cases where the eigenvalues do not converge, increasing the number of Krylov vectors can lead to better estimates of the eigenvalues at the expense of increased total computation time.

Note: Poor estimates of eigenvalues also increase solution time by slowing the convergence of the AMG algorithm.

temperature_smoothing_order (integer) >=1 [=2]

When *temperature_smoothing_type*=chebyshev, this parameter sets the polynomial order for the smoother. When *temperature_smoothing_type*=jacobi this parameter sets the number of smoothing passes during the downward and upward multigrid cycles. For both Chebyshev and Jacobi methods, increasing the smoothing order will reduce the total number of linear solver iterations required to converge the system. However, the time per iteration will also increase. There should be a balance between the number of iterations and the time per iteration. For temperature matrices, a value of two is found to give the best performance.

temperature_chebyshev_max_min_ratio (real) >1 <=100 [=10]

When *temperature_smoothing_type*=chebyshev, this parameter provides an estimate of the smallest eigenvalue by dividing the largest eigenvalue calculated by the Lanczos algorithm by this parameter. Chebyshev polynomial smoothing requires the ratio of the largest to the smallest eigenvalues of the matrix to calculate the polynomial coefficients. The smallest value may not be calculated. This parameter can be used to get an estimate. The default of 10 means that the smallest eigenvalue is one tenth of the largest.

temperature_jacobi_relaxation_factor (real) >0 [=0.25]

Used when *temperature_smoothing_type*=jacobi. Jacobi smoothing may not be stable and it should be relaxed. Conventional under relaxation is used to stabilize it. The default of 0.25 means updated values will be calculated by adding 25 percent of new values to 75 percent of the old values.

temperature_setup_tolerance (real) >0 [=0]

The nonlinear convergence tolerance used to control when the AMG setup process is performed. If the residual is higher than this value, the AMG setup process is performed. If the residual is lower than this tolerance, the AMG algorithm uses the existing setup information that is stored in RAM. With the default value of zero, the AMG setup process will occur every time the equations are solved, regardless of the residual. As the simulation converges, the change in solution between time steps decreases, minimizing the need to execute the AMG setup process. If AMG is used for more than one equation, such as temperature and turbulence, and this parameter greater than zero, the memory requirements will be increased.

Note: For each equation, a different setup tolerance can be used. This value should not be changed unless a convergence tolerance is set for the problem that is less than the default value.

species_negative_coupling_tolerance (real) >=0 <=1 [=0.5]

Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly negatively coupled variables for the species matrix are considered. Tolerance is the decimal fraction of the maximum value. The default value of 0.5 sets the tolerance to use values of 50 percent or greater of the most negative value. With a value of 1.0, negative coupling is ignored. Negative coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly negatively coupled if the negated value of the matrix entry exceeds the maximum negated value of the matrix multiplied by the negative coupling tolerance. The value of *species_negative_coupling_tolerance* can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

species_positive_coupling_tolerance (real) >=0 <=1 [=1.0]

Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly positively coupled variables for the species matrix are considered. Tolerance is the decimal fraction of the maximum value. With the default value of 1.0, positive coupling is ignored. Positive coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly positively coupled if the value of the matrix entry exceeds the maximum value of the matrix multiplied by the positive coupling tolerance. The value of *species_positive_coupling_tolerance* can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

max_species_final_matrix (integer) >=1 [=100]

Maximum number of entries present in the coarsest matrix of the AMG algorithm. With the default value of 100, coarsening will stop when the lowest level matrix is 10 by 10 or smaller. Increasing the size of this parameter may lead to longer duration of the solution calculation of the lowest level matrix.

species_standard_interpolation (boolean) [=on]

Interpolation method used to build the coarse multigrid matrices from fine matrices or to transfer the solution from a fine level to a coarse multigrid level, and vice versa. With this option set to off, direct interpolation will be used. Each equation in the fine level matrix that is considered to be important in the coarsening process gets interpolated to build the coarse level matrix. In direct interpolation, just neighboring points within the one fine level equation are used to calculate the interpolation weights. In standard interpolation, in addition to the neighboring points, other coupled equations are also considered in the weight calculations. More information is incorporated in the standard interpolation, leading to better convergence. In most cases, standard interpolation is recommended.

species_positive_negative_separate (boolean) [=off]

Determines if the positive and negative weights are separated during either standard or direct interpolation. In either direct or standard interpolation, weights are functions of off diagonal components of the fine level matrix and can be positive or negative. Interpolation can be conducted either separately for positive and negative weights, or these weights can be added together to have one interpolation. With this option set to on, summation of positive and negative

weights will be calculated and used for interpolation. For species matrices, separation of weights is not recommended.

species_truncated_interpolation (boolean) [=on]
Determines if the small interpolation weights will be truncated when interpolating the fine level multigrid matrix to the coarse level matrix or when transferring the solution from the coarse matrix to the finer AMG matrix levels. This flag should be used together with the *species_truncation_tolerance* to set the tolerance to determine the small weights. Truncation can be used with both direct and standard interpolation. A minor increase in the computational time to reach convergence can be observed if this flag is set to off.

species_truncation_tolerance (real) >=0 <=1 [=0.1]
Sets the portion of the smallest interpolation weights to be truncated during interpolation from a fine level to a coarse level. Used when *species_truncated_interpolation*=on. With a value of zero, no interpolation weights will be truncated. With a value of one, all weights will be truncated.

species_givens_scaling (boolean) [=off]
Determines if two by two Givens matrix scaling is used for the multigrid smoothing process. With this option turned *off*, diagonal scaling is used.

species_smoothing_type (enumerated) [=chebyshev]
Smoothing method used at each grid level after interpolation when values are from a fine matrix to a coarse matrix, or from a coarse matrix to a fine matrix.

Chebyshev	Chebyshev polynomial smoothing. Chebychev smoothing provides better performance, accurate convergence with less computational time, and is recommended for most matrices.
Jacobi	Jacobi smoothing. Jacobi smoothing is recommended for the fully coupled flow stagger.

species_num_krylov_vectors (integer) >=0 [=30]
Used with *species_smoothing_type*=chebyshev. Number of Krylov vectors in the Arnoldi algorithm for computing the largest eigenvalues at each multigrid level. In general, eigenvalues converge within 30 Krylov vectors. For cases where the eigenvalues do not converge, increasing the number of Krylov vectors can lead to better estimates of the eigenvalues at the expense of increased total computation time.

Note: Poor estimates of eigenvalues also increase solution time by slowing the convergence of the AMG algorithm.

species_smoothing_order (integer) >=1 [=2]
When *species_smoothing_type*=chebyshev, this parameter sets the polynomial order for the smoother. When *species_smoothing_type*=jacobi this parameter sets the number of smoothing passes during the downward and upward multigrid cycles. For both Chebyshev and Jacobi methods, increasing the smoothing order will reduce the total number of linear solver iterations required to converge the system. However, the time per iteration will also increase. There should be a balance between the number of iterations and the time per iteration. For species matrices, a value of two is found to give the best performance.

species_chebyshev_max_min_ratio (real) >1 <=100 [=10]
When *species_smoothing_type*=chebyshev, this parameter provides an estimate of the smallest eigenvalue by dividing the largest eigenvalue calculated by the Lanczos algorithm by this parameter. Chebyshev polynomial smoothing requires the ratio of the largest to the smallest eigenvalues of the matrix to calculate the polynomial coefficients. The smallest value may not be calculated. This parameter can be used to get an estimate. The default of 10 means that the smallest eigenvalue is one tenth of the largest.

species_jacobi_relaxation_factor (real) >0 [=0.25]
Used when *species_smoothing_type*=jacobi. Jacobi smoothing may not be stable and it should be relaxed. Conventional under relaxation is used to stabilize it. The default of 0.25 means updated values will be calculated by adding 25 percent of new values to 75 percent of the old values.

species_setup_tolerance (real) >0 [=0]
The nonlinear convergence tolerance used to control when the AMG setup process is performed. If the residual is higher than this value, the AMG setup process is performed. If the residual is lower than this tolerance, the AMG algorithm uses the existing setup information that is stored in RAM. With the default value of zero, the AMG setup process will occur every time the equations are solved, regardless of the residual. As the simulation converges, the change in solution between time steps decreases, minimizing the need to execute the AMG setup process. If AMG is used for more than one equation, such as species and turbulence, and this parameter is greater than zero, the memory requirements will be increased.

> **Note:** For each equation, a different setup tolerance can be used. This value should not be changed unless a convergence tolerance is set for the problem that is less than the default value.

turbulence_negative_coupling_tolerance (real) >=0 <=1 [=0.5]
Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly negatively coupled variables for the turbulence matrix are considered. Tolerance is the decimal fraction of the maximum value. The default value of 0.5 sets the tolerance to use values of 50 percent or greater of the most negative value. With a value of 1.0, negative coupling is ignored. Negative coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly negatively coupled if the negated value of the matrix entry exceeds the maximum negated value of the matrix multiplied by the negative coupling tolerance. The value of *turbulence_negative_coupling_tolerance* can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

turbulence_positive_coupling_tolerance (real) >=0 <=1 [=1.0]
Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly positively coupled variables for the turbulence matrix are considered. Tolerance is the decimal fraction of the maximum value. With the default value of 1.0, positive coupling is ignored. Positive coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly positively coupled if the value of the matrix entry exceeds

the maximum value of the matrix multiplied by the positive coupling tolerance. The value of *turbulence_positive_coupling_tolerance* can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

max_turbulence_final_matrix (integer) >=1 [=100]
Maximum number of entries present in the coarsest matrix of the AMG algorithm. With the default value of 100, coarsening will stop when the lowest level matrix is 10 by 10 or smaller. Increasing the size of this parameter may lead to longer duration of the solution calculation of the lowest level matrix.

turbulence_standard_interpolation (boolean) [=on]
Interpolation method used to build the coarse multigrid matrices from fine matrices or to transfer the solution from a fine level to a coarse multigrid level, and vice versa. With this option set to off, direct interpolation will be used. Each equation in the fine level matrix that is considered to be important in the coarsening process gets interpolated to build the coarse level matrix. In direct interpolation, just neighboring points within the one fine level equation are used to calculate the interpolation weights. In standard interpolation, in addition to the neighboring points, other coupled equations are also considered in the weight calculations. More information is incorporated in the standard interpolation, leading to better convergence. In most cases, standard interpolation is recommended.

turbulence_positive_negative_separate (boolean) [=off]
Determines if the positive and negative weights are separated during either standard or direct interpolation. In either direct or standard interpolation, weights are functions of off diagonal components of the fine level matrix and can be positive or negative. Interpolation can be conducted either separately for positive and negative weights, or these weights can be added together to have one interpolation. With this option set to on, summation of positive and negative weights will be calculated and used for interpolation. For turbulence matrices, separation of weights is not recommended.

turbulence_truncated_interpolation (boolean) [=on]
Determines if the small interpolation weights will be truncated when interpolating the fine level multigrid matrix to the coarse level matrix or when transferring the solution from the coarse matrix to the finer AMG matrix levels. This flag should be used together with the *turbulence_truncation_tolerance* to set the tolerance to determine the small weights. Truncation can be used with both direct and standard interpolation. A minor increase in the computational time to reach convergence can be observed if this flag is set to off.

turbulence_truncation_tolerance (real) >=0 <=1 [=0.1]
Sets the portion of the smallest interpolation weights to be truncated during interpolation from a fine level to a coarse level. Used when *turbulence_truncated_interpolation*=on. With a value of zero, no interpolation weights will be truncated. With a value of one, all weights will be truncated.

turbulence_givens_scaling (boolean) [=off]
Determines if two by two Givens matrix scaling is used for the multigrid smoothing process. With this option turned off, diagonal scaling is used.

turbulence_smoothing_type (enumerated) [=chebyshev]
Smoothing method used at each grid level after interpolation when values are from a fine matrix to a coarse matrix, or from a coarse matrix to a fine matrix.

Chebyshev	Chebyshev polynomial smoothing. Chebychev smoothing provides better performance, accurate convergence with less computational time, and is recommended for most matrices.
Jacobi	Jacobi smoothing. Jacobi smoothing is recommended for the fully coupled flow stagger.

turbulence_num_krylov_vectors (integer) >=0 [=30]
Used with *turbulence_smoothing_type*=chebyshev. Number of Krylov vectors in the Arnoldi algorithm for computing the largest eigenvalues at each multigrid level. In general, eigenvalues converge within 30 Krylov vectors. For cases where the eigenvalues do not converge, increasing the number of Krylov vectors can lead to better estimates of the eigenvalues at the expense of increased total computation time.

> **Note:** Poor estimates of eigenvalues also increase solution time by slowing the convergence of the AMG algorithm.

turbulence_smoothing_order (integer) >=1 [=3]
When *turbulence_smoothing_type*=chebyshev, this parameter sets the polynomial order for the smoother. When *turbulence_smoothing_type*=jacobi this parameter sets the number of smoothing passes during the downward and upward multigrid cycles. For both Chebyshev and Jacobi methods, increasing the smoothing order will reduce the total number of linear solver iterations required to converge the system. However, the time per iteration will also increase. There should be a balance between the number of iterations and the time per iteration. For turbulence matrices, a value of two is found to give the best performance.

turbulence_chebyshev_max_min_ratio (real) >1 <=100 [=10]
When *turbulence_smoothing_type*=chebyshev, this parameter provides an estimate of the smallest eigenvalue by dividing the largest eigenvalue calculated by the Lanczos algorithm by this parameter. Chebyshev polynomial smoothing requires the ratio of the largest to the smallest eigenvalues of the matrix to calculate the polynomial coefficients. The smallest value may not be calculated. This parameter can be used to get an estimate. The default of 10 means that the smallest eigenvalue is one tenth of the largest.

turbulence_jacobi_relaxation_factor (real) >0 [=0.25]
Used when *turbulence_smoothing_type*=jacobi. Jacobi smoothing may not be stable and it should be relaxed. Conventional under relaxation is used to stabilize it. The default of 0.25 means updated values will be calculated by adding 25 percent of new values to 75 percent of the old values.

turbulence_setup_tolerance (real) >0 [=0]
The nonlinear convergence tolerance used to control when the AMG setup process is performed. If the residual is higher than this value, the AMG setup process is performed. If the residual is lower than this tolerance, the AMG algorithm uses the existing setup information that is stored in RAM. With the default value of zero, the AMG setup process will occur every time the equations are solved, regardless of the residual. As the simulation converges, the change in solution between time steps decreases, minimizing the need to execute the AMG setup process. If AMG is used for more than one equation, such as temperature and turbulence, and this parameter is greater than zero, the memory requirements will be increased.

> **Note:** For each equation, a different setup tolerance can be used. This value should not be changed unless a convergence tolerance is set for the problem that is less than the default value.

`mesh_negative_coupling_tolerance` *(real) >=0 <=1 [=0.5]*

Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly negatively coupled variables for the mesh matrix are considered. Tolerance is the decimal fraction of the maximum value. The default value of 0.5 sets the tolerance to use values of 50 percent or greater of the most negative value. With a value of 1.0, negative coupling is ignored. Negative coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly negatively coupled if the negated value of the matrix entry exceeds the maximum negated value of the matrix multiplied by the negative coupling tolerance. The value of `mesh_negative_coupling_tolerance` can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

`mesh_positive_coupling_tolerance` *(real) >=0 <=1 [=1.0]*

Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly positively coupled variables for the mesh matrix are considered. Tolerance is the decimal fraction of the maximum value. With the default value of 1.0, positive coupling is ignored. Positive coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly positively coupled if the value of the matrix entry exceeds the maximum value of the matrix multiplied by the positive coupling tolerance. The value of `mesh_positive_coupling_tolerance` can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

`max_mesh_final_matrix` *(integer) >=1 [=100]*

Maximum number of entries present in the coarsest matrix of the AMG algorithm. With the default value of 100, coarsening will stop when the lowest level matrix is 10 by 10 or smaller. Increasing the size of this parameter may lead to longer duration of the solution calculation of the lowest level matrix.

`mesh_standard_interpolation` *(boolean) [=on]*

Interpolation method used to build the coarse multigrid matrices from fine matrices or to transfer the solution from a fine level to a coarse multigrid level, and vice versa. With this option set to off, direct interpolation will be used. Each equation in the fine level matrix that is considered to be important in the coarsening process gets interpolated to build the coarse level matrix. In direct interpolation, just neighboring points within the one fine level equation are used to calculate the interpolation weights. In standard interpolation, in addition to the neighboring points, other coupled equations are also considered in the weight calculations. More information is incorporated in the standard interpolation, leading to better convergence. In most cases, standard interpolation is recommended.

mesh_positive_negative_separate (boolean) [=on]
Determines if the positive and negative weights are separated during either standard or direct interpolation. In either direct or standard interpolation, weights are functions of off diagonal components of the fine level matrix and can be positive or negative. Interpolation can be conducted either separately for positive and negative weights, or these weights can be added together to have one interpolation. With this option set to on, summation of positive and negative weights will be calculated and used for interpolation. For mesh matrices, separation of weights is recommended.

mesh_truncated_interpolation (boolean) [=on]
Determines if the small interpolation weights will be truncated when interpolating the fine level multigrid matrix to the coarse level matrix or when transferring the solution from the coarse matrix to the finer AMG matrix levels. This flag should be used together with the *mesh_truncation_tolerance* to set the tolerance to determine the small weights. Truncation can be used with both direct and standard interpolation. A minor increase in the computational time to reach convergence can be observed if this flag is set to off.

mesh_truncation_tolerance (real) >=0 <=1 [=0.1]
Sets the portion of the smallest interpolation weights to be truncated during interpolation from a fine level to a coarse level. Used when *mesh_truncated_interpolation*=on. With a value of zero, no interpolation weights will be truncated. With a value of one, all weights will be truncated.

mesh_givens_scaling (boolean) [=off]
Determines if two by two Givens matrix scaling is used for the multigrid smoothing process. With this option turned off, diagonal scaling is used.

mesh_smoothing_type (enumerated) [=chebyshev]
Smoothing method used at each grid level after interpolation when values are from a fine matrix to a coarse matrix, or from a coarse matrix to a fine matrix.

Chebyshev	Chebyshev polynomial smoothing. Chebychev smoothing provides better performance, accurate convergence with less computational time, and is recommended for most matrices.
Jacobi	Jacobi smoothing. Jacobi smoothing is recommended for the fully coupled flow stagger.

mesh_eigenvalue_tolerance (real) >0 [=1.e-2]
Tolerance to stop the eigenvalue calculation iterations. Used with *mesh_smoothing_type*=chebyshev. Chebyshev polynomial smoothing requires the ratio of the largest to the smallest eigenvalues of the matrix. The Iterative Lanczos algorithm is used to calculate the maximum eigenvalue of the mesh matrix.

max_mesh_eigenvalue_iterations (integer) >=0 [=20]
Used with *mesh_smoothing_type*=chebyshev. Maximum number of iterations of the Lanczos algorithm for computing the largest eigenvalues at each multigrid level. The *mesh_eigenvalue_tolerance* and *max_mesh_eigenvalue_iterations* determine when iterations stop. Whichever constraint is met first stops the algorithm. In general, eigenvalues converge within 20 iterations. For cases where the eigenvalues do not converge, increasing the total number of iterations can lead to better estimates of the eigenvalues at the expense of increased total computation time.

> **Note:** Poor estimates of eigenvalues also increase solution time by slowing the convergence of the AMG algorithm.

mesh_smoothing_order (integer) >=1 [=2]
When *mesh_smoothing_type*=chebyshev, this parameter sets the polynomial order for the smoother. When *mesh_smoothing_type*=jacobi this parameter sets the number of smoothing passes during the downward and upward multigrid cycles. For both Chebyshev and Jacobi methods, increasing the smoothing order will reduce the total number of linear solver iterations required to converge the system. However, the time per iteration will also increase. There should be a balance between the number of iterations and the time per iteration. For mesh matrices, a value of two is found to give the best performance.

mesh_chebyshev_max_min_ratio (real) >1 <=100 [=10]
When *mesh_smoothing_type*=chebyshev, this parameter provides an estimate of the smallest eigenvalue by dividing the largest eigenvalue calculated by the Lanczos algorithm by this parameter. Chebyshev polynomial smoothing requires the ratio of the largest to the smallest eigenvalues of the matrix to calculate the polynomial coefficients. The smallest value may not be calculated. This parameter can be used to get an estimate. The default of 10 means that the smallest eigenvalue is one tenth of the largest.

mesh_jacobi_relaxation_factor (real) >0 [=0.25]
Used when *mesh_smoothing_type*=jacobi. Jacobi smoothing may not be stable and it should be relaxed. Conventional under relaxation is used to stabilize it. The default of 0.25 means updated values will be calculated by adding 25 percent of new values to 75 percent of the old values.

mesh_setup_tolerance (real) >0 [=0]
The nonlinear convergence tolerance used to control when the AMG setup process is performed. If the residual is higher than this value, the AMG setup process is performed. If the residual is lower than this tolerance, the AMG algorithm uses the existing setup information that is stored in RAM. With the default value of zero, the AMG setup process will occur every time the equations are solved, regardless of the residual. As the simulation converges, the change in solution between time steps decreases, minimizing the need to execute the AMG setup process. If AMG is used for more than one equation, such as mesh and turbulence, and this parameter is greater than zero, the memory requirements will be increased.

> **Note:** For each equation, a different setup tolerance can be used. This value should not be changed unless a convergence tolerance is set for the problem that is less than the default value.

viscoelastic_negative_coupling_tolerance (real) >=0 <=1 [=0.5]
Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly negatively coupled variables for the viscoelastic matrix are considered. Tolerance is the decimal fraction of the maximum value. The default value of 0.5 sets the tolerance to use values of 50 percent or greater of the most negative value. With a value of 1.0, negative coupling is ignored. Negative coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly negatively coupled if the negated value of the matrix entry exceeds the

maximum negated value of the matrix multiplied by the negative coupling tolerance. The value of *viscoelastic_negative_coupling_tolerance* can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

viscoelastic_positive_coupling_tolerance (real) >=0 <=1 [=0.3]
Tolerance within the algebraic multigrid (AMG) coarsening algorithm to determine when strongly positively coupled variables for the viscoelastic matrix are considered. Tolerance is the decimal fraction of the maximum value. The default value of 0.3 sets the tolerance to use values of 30 percent or greater of the most positive value. With a value of 1.0, positive coupling is ignored. Positive coupling tolerance is used when determining which entries from the fine matrix levels are retained at the coarser matrix levels. To select the coarse level grids for a matrix, only the values with a high contribution to the unknown equations are considered. Terms are considered to be highly positively coupled if the value of the matrix entry exceeds the maximum value of the matrix multiplied by the positive coupling tolerance. The value of *viscoelastic_positive_coupling_tolerance* can have significant influence on the structure of coarse level grids, influencing the convergence rate of the linear solution.

max_viscoelastic_final_matrix (integer) >=1 [=100]
Maximum number of entries present in the coarsest matrix of the AMG algorithm. With the default value of 100, coarsening will stop when the lowest level matrix is 10 by 10 or smaller. Increasing the size of this parameter may lead to longer duration of the solution calculation of the lowest level matrix.

viscoelastic_standard_interpolation (boolean) [=on]
Interpolation method used to build the coarse multigrid matrices from fine matrices or to transfer the solution from a fine level to a coarse multigrid level, and vice versa. With this option set to off, direct interpolation will be used. Each equation in the fine level matrix that is considered to be important in the coarsening process gets interpolated to build the coarse level matrix. In direct interpolation, just neighboring points within the one fine level equation are used to calculate the interpolation weights. In standard interpolation, in addition to the neighboring points, other coupled equations are also considered in the weight calculations. More information is incorporated in the standard interpolation, leading to better convergence. In most cases, standard interpolation is recommended.

viscoelastic_positive_negative_separate (boolean) [=off]
Determines if the positive and negative weights are separated during either standard or direct interpolation. In either direct or standard interpolation, weights are functions of off diagonal components of the fine level matrix and can be positive or negative. Interpolation can be conducted either separately for positive and negative weights, or these weights can be added together to have one interpolation. With this option set to on, summation of positive and negative weights will be calculated and used for interpolation. For viscoelastic matrices, separation of weights is not recommended.

viscoelastic_truncated_interpolation (boolean) [=on]
Determines if the small interpolation weights will be truncated when interpolating the fine level multigrid matrix to the coarse level matrix or when transferring the solution from the coarse matrix to the finer AMG matrix levels. This flag should be used together with the *viscoelastic_truncation_tolerance* to set the tolerance to determine the small weights. Truncation can be used with both direct and standard interpolation. A minor increase in the computational time to reach convergence can be observed if this flag is set to off.

viscoelastic_truncation_tolerance (real) >=0 <=1 [=0.1]
Sets the portion of the smallest interpolation weights to be truncated during interpolation from a fine level to a coarse level. Used when *viscoelastic_truncated_interpolation*=on. With a value of zero, no interpolation weights will be truncated. With a value of one, all weights will be truncated.

viscoelastic_givens_scaling (boolean) [=off]
Determines if two by two Givens matrix scaling is used for the multigrid smoothing process. With this option turned off, diagonal scaling is used.

viscoelastic_smoothing_type (enumerated) [=chebyshev]
Smoothing method used at each grid level after interpolation when values are from a fine matrix to a coarse matrix, or from a coarse matrix to a fine matrix.

Chebyshev	Chebyshev polynomial smoothing. Chebychev smoothing provides better performance, accurate convergence with less computational time, and is recommended for most matrices.
Jacobi	Jacobi smoothing. Jacobi smoothing is recommended for the fully coupled flow stagger.

viscoelastic_num_krylov_vectors (integer) >=0 [=30]
Used with *viscoelastic_smoothing_type*=chebyshev. Number of Krylov vectors in the Arnoldi algorithm for computing the largest eigenvalues at each multigrid level. In general, eigenvalues converge within 30 Krylov vectors. For cases where the eigenvalues do not converge, increasing the number of Krylov vectors can lead to better estimates of the eigenvalues at the expense of increased total computation time.

> **Note:** Poor estimates of eigenvalues also increase solution time by slowing the convergence of the AMG algorithm.

viscoelastic_smoothing_order (integer) >=1 [=2]
When *viscoelastic_smoothing_type*=chebyshev, this parameter sets the polynomial order for the smoother. When *viscoelastic_smoothing_type*=jacobi this parameter sets the number of smoothing passes during the downward and upward multigrid cycles. For both Chebyshev and Jacobi methods, increasing the smoothing order will reduce the total number of linear solver iterations required to converge the system. However, the time per iteration will also increase. There should be a balance between the number of iterations and the time per iteration. For viscoelastic matrices, a value of two is found to give the best performance.

viscoelastic_chebyshev_max_min_ratio (real) >1 <=100 [=10]
When *viscoelastic_smoothing_type*=chebyshev, this parameter provides an estimate of the smallest eigenvalue by dividing the largest eigenvalue calculated by the Lanczos algorithm by this parameter. Chebyshev polynomial smoothing requires the ratio of the largest to the smallest eigenvalues of the matrix to calculate the polynomial coefficients. The smallest value may not be calculated. This parameter can be used to get an estimate. The default of 10 means that the smallest eigenvalue is one tenth of the largest.

viscoelastic_jacobi_relaxation_factor (real) >0 [=0.25]
Used when *viscoelastic_smoothing_type*=jacobi. Jacobi smoothing may not be stable and it should be relaxed. Conventional under relaxation is used to stabilize it. The default of 0.25 means updated values will be calculated by adding 25 percent of new values to 75 percent of the old values.

viscoelastic_setup_tolerance (real) >0 [=0]
The nonlinear convergence tolerance used to control when the AMG setup process is performed. If the residual is higher than this value, the AMG setup process is performed. If the residual is lower than this tolerance, the AMG algorithm uses the existing setup information that is stored in RAM. With the default value of zero, the AMG setup process will occur every time the equations are solved, regardless of the residual. As the simulation converges, the change in solution between time steps decreases, minimizing the need to execute the AMG setup process. If AMG is used for more than one equation, such as viscoelastic and turbulence, and this parameter is greater than zero, the memory requirements will be increased.

Note: For each equation, a different setup tolerance can be used. This value should not be changed unless a convergence tolerance is set for the problem that is less than the default value.

Description

This command specifies the parameters of the algebraic multigrid algorithm. This option can be used to speed up the linear solution on problems in which the solver takes a large number of iterations at each step. The AMG algorithm works by interpolating the full matrix equation down to smaller sized matrices that can be solved directly with very little effort. The solution on the coarse matrix is then propagated back up through the successively finer matrices until it reaches the full matrix. At this point, the AMG solution is used to precondition the standard solver used for each stagger. This operation carries a significant amount of additional compute expense to perform, but can provide significant acceleration of the solution in some cases. This method also carries additional memory overhead. For most applications, turning this feature on will lead to approximately 50 percent greater memory consumption than using the standard preconditioning.

For example, to specify the use of AMG on the pressure projection equation when performing the fully coupled flow solve, the following commands can be used:

```
STAGGER("flow"){
   ...
   pressure_projection                    = on
   pressure_algebraic_multigrid           = on
   ...
}
ALGEBRAIC_MULTIGRID_PARAMETERS {
   pressure_standard_interpolation        = on
   pressure_truncated_interpolation       = on
   pressure_negative_coupling_tolerance   = 0.6
   pressure_positive_coupling_tolerance   = 1
   pressure_truncation_tolerance          = 0.1
   max_pressure_final_matrix              = 100
   pressure_eigenvalue_tolerance          = 0.01
   max_pressure_eigenvalue_iterations     = 20
   pressure_smoothing_order               = 2
   pressure_chebyshev_max_min_ratio       = 10
```

```
    pressure_jacobi_relaxation_factor        = 0.25
    pressure_smoothing_type                  = chebyshev
    pressure_positive_negative_separate      = off
    pressure_givens_scaling                  = on
    pressure_setup_tolerance                 = 0
}
```

AUTO_SOLUTION_STRATEGY

Automatically creates the solution strategy by issuing all the other commands in this chapter.

Type

AcuSolve Command

Syntax

AUTO_SOLUTION_STRATEGY {parameters}

Qualifier

This command has no qualifier.

Parameters

min_cases >=0 [=1]
: Minimum number of optimization cases allowed for the run.

max_cases >=0 [=50]
: Maximum number of optimization cases allowed for the run.

optimization_convergence_tolerance >=0 [=1.e-4]
: A necessary condition for the termination of an optimization run is that all convergence measures must be less than this tolerance.

design_optimization (boolean) [=off]
: Flag to turn on *design_optimization*. This requires an OPTIMIZATION command.

min_time_steps or min_steps (integer) >=0 [=1]
: Minimum number of time steps allowed for a run or a case. If zero, this option is ignored.

max_time_steps or max_steps (integer) >=0 [=100]
: Maximum number of time steps allowed for a run or a case. The run may terminate earlier due to other criteria. If more than one case is used for a run, a case may stop earlier due to other criteria, and the run will move to the next case. If zero, this option is ignored.

initial_case_time_steps (integer) >=0 [=0]
: Minimum number of time steps used for the initial case. This option will also override the *max_time_steps* parameter for the initial case. If zero, this option is ignored.

initial_time (real) >=0 [=0]
: Specifies the starting time for the simulation. If set to a value greater than zero, the simulation begins at a specified time. When set to zero, this option is ignored. Applicable only for transient problems.

final_time (real) >=0 [=0]
: Final time of the run. The run will terminate when the analysis time reaches the final time. The run may terminate earlier due to other criteria. If zero, this option is ignored. Used only with transient problems.

initial_time_increment or *init_dt* *(real) >=0 [=1.e+10]*
: Initial value of the time increment for auto time increment control, or the constant value of the time increment for a fixed time increment. If zero, it is set according to a CFL=1 condition.

auto_time_increment or *auto_dt* *(boolean) [=off]*
: Flag specifying whether or not to automatically control the time increment. Used only with transient problems.

min_time_increment or *min_dt* *(real) >=0 [=0]*
: Minimum time increment when using the auto time increment option. Used with transient analyses.

max_time_increment or *max_dt* *(real) >=0 [=0]*
: Maximum time increment when using the auto time increment option. Used with transient analyses.

min_stagger_iterations or *min_stg_iters* *(integer) >=0 [=0]*
: Minimum number of stagger iterations before advancing to the next time step. If zero, this option is determined internally. If set to zero, AcuSolve internally sets the *min_stagger_iterations* to two when *auto_time_increment* is off. When *auto_time_increment* is on, a value of one is used. This option is only used when solving transient analyses.

max_stagger_iterations or *max_stg_iters* *(integer) >=0 [=0]*
: Maximum number of stagger iterations before advancing to the next time step. If zero, this option is determined internally. If set to zero, AcuSolve internally sets the *max_stagger_iterations* to two when *auto_time_increment* is off. When *auto_time_increment* is on, a value of four is used. This option is only used when solving transient analysis.

convergence_tolerance or *conv_tol* *(real) >=0 [=1.e-3]*
: A necessary condition for the termination of a run is that all convergence measures must be less than this tolerance.

num_krylov_vectors or *kvecs* *(integer) >=0 [=10]*
: Number of Krylov vectors used by GMRES and BiCGStab.

flow *(boolean) [=on]*
: Flag specifying whether or not to solve the flow equations.

temperature or *temp* *(boolean) [=on]*
: Flag specifying whether or not to solve the temperature equation.

temperature_flow or *temp_flow* *(boolean) [=off]*
: Flag specifying whether or not to solve thermal-flow problems in a partially coupled manner. Requires *flow* and *temperature* to be both on.

radiation or *rad* *(boolean) [=on]*
: Flag specifying whether or not to solve the radiation equation. Also, p1 incident radiation equation or discrete ordinates method for solving the radiative transfer equation.

species_1 or *spec1* *(boolean) [=on]*
: Flag specifying whether or not to solve the corresponding species equation.

species_2 or *spec2* *(boolean) [=on]*
: Flag specifying whether or not to solve the corresponding species equation.

species_3 or *spec3* *(boolean) [=on]*
Flag specifying whether or not to solve the corresponding species equation.

species_4 or *spec4* *(boolean) [=on]*
Flag specifying whether or not to solve the corresponding species equation.

species_5 or *spec5* *(boolean) [=on]*
Flag specifying whether or not to solve the corresponding species equation.

species_6 or *spec6* *(boolean) [=on]*
Flag specifying whether or not to solve the corresponding species equation.

species_7 or *spec7* *(boolean) [=on]*
Flag specifying whether or not to solve the corresponding species equation.

species_8 or *spec8* *(boolean) [=on]*
Flag specifying whether or not to solve the corresponding species equation.

species_9 or *spec9* *(boolean) [=on]*
Flag specifying whether or not to solve the corresponding species equation.

field *(boolean) [=on]*
Flag specifying whether or not to solve the field equations for multi field simulations.

turbulence or *turb* *(boolean) [=on]*
Flag specifying whether or not to solve the turbulence equation.

turbulence_transition or *ttm* *(boolean) [=on]*
Flag specifying whether or not to solve the turbulence transition equations.

mesh *(boolean) [=on]*
Flag specifying whether or not to solve the mesh displacement equation.

viscoelastic or *vest* *(boolean) [=on]*
Specifying whether or not to solve the viscoelasticity equations.

external_code *(boolean) [=on]*
Flag specifying whether or not to solve the solid/structural equations with an external code.

particle_trace *[=on]*
Flag specifying whether or not to solve the particle equations in a coupled particle-flow using AcuTrace.

relaxation_factor *(real) >=0 <=1 [=0]*
Parameter used to slow the solution convergence for hard problems.

topology_stagger_reordering *(boolean) [=on]*
Flag specifying whether or not the flow and adjoint equations in topology optimization will be solved sequentially or simultaneously. When *topology_stagger_reordering* = on, the flow and adjoint equation will be solved sequentially.

electric_potential or *elec_p* *(boolean) [=on]*
Flag specifying whether to solve the charge conservation equation.

Description

The goal of the AUTO_SOLUTION_STRATEGY command is to completely automate the specification of all the solution strategy commands based on the physical specification of the problem. This is not entirely possible currently, but for most problems only the few parameters from this command are needed. AUTO_SOLUTION_STRATEGY is a functional command, hence it is processed immediately upon being read from the input file. As such its position in the input file may be very important. When AUTO_SOLUTION_STRATEGY is issued, the parameters of the previously-given ANALYSIS and EQUATION commands are used to determine whether the problem is steady or transient and what equations are being solved. Then the appropriate TIME_SEQUENCE, STAGGER, TIME_INCREMENT, TIME_INTEGRATION, and LINEAR_SOLVER_PARAMETERS commands are issued. These commands are also saved in a file called problem.ss.inc, where problem is given by the corresponding option of the AcuRun or AcuPrep programs. See the *AcuSolve Programs Reference Manual*. The parameters from these commands may be overwritten afterward by manually issuing the commands. On the next run problem.ss.inc may be included, modified or not, instead of AUTO_SOLUTION_STRATEGY.

For example, the simplest form of this command for a steady-state problem is:

```
ANALYSIS {
    title = "A steady problem"
    type = steady
}
EQUATION {
    flow = navier_stokes
    turbulence = spalart_allmaras
   temperature = advective_diffusive
}
AUTO_SOLUTION_STRATEGY {
}
```

And for a transient problem:

```
ANALYSIS {
    title = "A transient problem"
    type = transient
}
EQUATION {
    ...
}
AUTO_SOLUTION_STRATEGY {
    initial_time_increment = 0.01
}
```

For a steady-state problem AUTO_SOLUTION_STRATEGY accepts the following parameters:

```
ANALYSIS {
   type                        = steady
}
AUTO_SOLUTION_STRATEGY {
   max_time_steps              = 100
   initial_time_increment      = 1.e+10
   convergence_tolerance       = 1.e-3
   num_krylov_vectors          = 10
   flow                        = on
   temperature                 = on
   temperature_flow            = off
   radiation                   = on
   species_1                   = on
```

```
    species_2                   = on
    species_3                   = on
    species_4                   = on
    species_5                   = on
    species_6                   = on
    species_7                   = on
    species_8                   = on
    species_9                   = on
    field                       = on
    turbulence                  = on
    transition                  = on
    mesh                        = on
    viscoelastic                = on
    external_code               = on
    relaxation_factor           = 0
  }
```

The given values are the defaults. Three of these parameters are determined by the level of difficulty of the problem. The number of krylov vectors for flow and tflow staggers is given by *num_krylov_vectors*; the number of vectors for other staggers is determined by this base value. To solve thermal-flow problems with a partially-coupled strategy *temperature_flow* is set to on, otherwise these problems are solved with a staggered strategy. In the partially coupled strategy *temperature_flow* approach, the continuity, momentum, and energy equations are considered coupled; however, the energy equation is projected out of the coupled system and solved separately in a temperature stagger to obtain temperature preconditioning vectors. After that, these vectors are used to precondition the coupled continuity and momentum equations in the flow stagger. This method will be helpful only if there is a temperature-dependent source term in the momentum equation, such as in the Bousinesq approximation. The choice generally depends on the degree of coupling, that is, buoyancy, between the momentum and thermal equations; forced convection problems may be solved staggered, while free convection problems usually benefit from the more expensive coupled approach. The solution convergence behavior of hard problems may be smoothed by increasing *relaxation_factor*; this parameter will have the most impact on the *velocity_update_factor* parameter of the LINEAR_SOLVER_PARAMETERS command. The other parameters from flow to *external_code* are used only if the corresponding equation in the EQUATION command is defined. These are normally left on, except when an equation exists in a problem but a solution for it is not desired. The other parameters are used the same way as for steady problems.

For a transient problem AUTO_SOLUTION_STRATEGY accepts the following parameters:

```
ANALYSIS {
    type                        = steady
}
AUTO_SOLUTION_STRATEGY {
    max_time_steps              = 100
    initial_time_increment      = 1.e+10
    convergence_tolerance       = 1.e-3
    num_krylov_vectors          = 10
    flow                        = on
    temperature                 = on
    temperature_flow            = off
    radiation                   = on
    species_1                   = on
    species_2                   = on
    species_3                   = on
    species_4                   = on
    species_5                   = on
```

```
    species_6                       = on
    species_7                       = on
    species_8                       = on
    species_9                       = on
    field                           = on
    turbulence                      = on
    transition                      = on
    mesh                            = on
    viscoelastic                    = on
    external_code                   = on
    relaxation_factor               = 0
}
```

Note: The default for *initial_time_increment* is different from the steady case. You are encouraged to set a reasonable value for *initial_time_increment*.

If this parameter is not specified, or set to zero, the time increment which satisfies a maximum CFL condition of one is used. The nodal boundary conditions are imposed over the initial conditions before the CFL numbers are computed. If *auto_time_increment* is specified, then you may also supply *min_time_increment* and *max_time_increment*. If *min_time_increment* is zero, it is reset at the beginning of each time step such that the maximum CFL number is one. If *max_time_increment* is zero, it is ignored. If *min_stagger_iterations* or *max_stagger_iterations* is non-zero, the given value is used to control the nonlinear iterations instead of the default value.

As a more complex example, consider the case where a transient thermal problem is to be solved based on a steady flow field. This may be accomplished by first specifying:

```
ANALYSIS {
     title = "A thermal/flow problem"
     type = steady
}
EQUATION {
    flow = navier_stokes
}
AUTO_SOLUTION_STRATEGY {
}
```

And then restarting with:

```
RESTART {
}
ANALYSIS {
    title = "A thermal/flow problem"
    type = transient
}
EQUATION {
    flow = navier_stokes
    temperature = advective_diffusive
}
AUTO_SOLUTION_STRATEGY {
    initial_time_increment = 0.01
```

You can discover the stagger names used in the *staggers* parameter of the TIME_SEQUENCE command by examining the problem.ss.inc file.

The optimization solution strategy adds the concept of cases. A case normally holds a single design in order to make the solutions output for shape optimization more manageable. In practical terms, a case is a grouping of a sequence of time steps. You specify the minimum and maximum of cases for a single simulation run, and you also specify the minimum and maximum time steps for each case. The ordinary convergence tolerance relates to the time step sequence as usual, and the optimization convergence tolerance relates to the overall convergence of the shape and optimization problem.

The other solution strategy parameters are similar to what is used running regular fluid dynamic runs. It may also be a good idea to use some relaxation, for instance specified by the relaxation factor in the auto solution command.

For an optimization problem `AUTO_SOLUTION_STRATEGY` accepts the following parameters:

```
AUTO_SOLUTION_STRATEGY {
    min_cases = 5
    max_cases = 50
    optimization_convergence_tolerance = 1.e-3
    min_time_steps = 10
    max_time_steps = 20
    initial_case_time_steps = 100
    convergence_tolerance = 1.e-3
```

CONVERGENCE_CHECK_PARAMETERS

Specifies the parameters for fine tuning convergence checking.

Type

AcuSolve Command

Syntax

CONVERGENCE_CHECK_PARAMETERS {parameters}

Qualifier

This command has no qualifier.

Parameters

pressure_residual_check or *pres_res* *(enumerated) [=standard]*

Modifies the convergence check of the residual ratio of the pressure equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

pressure_solution_increment_check or *pres_sol* *(enumerated) [=looser_by_10]*

Modifies the convergence check of the solution increment ratio of the pressure equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.

tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

velocity_residual_check or vel_res (enumerated) [=standard]

Modifies the convergence check of the residual ratio of the velocity equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

velocity_solution_increment_check or vel_sol (enumerated) [=looser_by_10]

Modifies the convergence check of the solution increment ratio of the velocity equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

temperature_residual_check or temp_res (enumerated) [=standard]

Modifies the convergence check of the residual ratio of the temperature equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.

looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`temperature_solution_increment_check` or `temp_sol` (enumerated) [=looser_by_10]

Modifies the convergence check of the solution increment ratio of the temperature equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`radiation_residual_check` or `rad_res` (enumerated) [=none]

Modifies the convergence check of the residual ratio of the radiation equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.

tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

radiation_solution_increment_check or *rad_sol* *(enumerated) [=looser_by_10]*

Modifies the convergence check of the solution increment ratio of the radiation equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

species_residual_check or *species_res* *(enumerated) [=standard]*

Modifies the convergence check of the residual ratio of the species equations.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

species_solution_increment_check or *species_sol* *(enumerated) [=looser_by_10]*

Modifies the convergence check of the solution increment ratio of the species equations.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.

looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`field_residual_check` or `field_res` (enumerated) [=standard]

Modifies the convergence check of the residual ratio of the phase-field transport equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`field_solution_increment_check` or `field_sol` (enumerated) [=looser_by_10]

Modifies the convergence check of the solution increment ratio of the phase-field transport equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.

tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`levelset_residual_check` or `ls_res` *(enumerated) [=looser_by_10]*

Modifies the convergence check of the residual ratio of the level set transport equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`levelset_solution_increment_check` or `ls_sol` *(enumerated) [=looser_by_100]*

Modifies the convergence check of the solution increment ratio of the level set transport equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`redistancing_residual_check` or `rdist_res` *(enumerated) [=looser_by_100]*

Modifies the convergence check of the residual ratio of the level set sharpening equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.

looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`redistancing_solution_increment_check` or *`rdist_sol`* *(enumerated) [=looser_by_100]*

Modifies the convergence check of the solution increment ratio of the level set sharpening equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`turbulence_residual_check` or *`turb_res`* *(enumerated) [=looser_by_10]*

Modifies the convergence check of the residual ratio of the turbulence equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.

tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`turbulence_solution_increment_check` or *`turb_sol`* *(enumerated) [=looser_by_100]*

Modifies the convergence check of the solution increment ratio of the turbulence equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`transition_residual_check` or *`ttm_res`* *(enumerated) [=looser_by_100]*

Modifies the convergence check of the residual ratio of the transition model equations.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`transition_solution_increment_check` or *`ttm_sol`* *(enumerated) [=looser_by_1000]*

Modifies the convergence check of the solution increment ratio of the transition model equations.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.

looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

viscoelastic_stress_residual_check or *vest_res* *(enumerated) [=standard]*

Modifies the convergence check of the residual ratio of the viscoelastic stress equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

viscoelastic_stress_solution_increment_check or *vest sol* *(enumerated) [=looser by 10]*

Modifies the convergence check of the solution increment ratio of the viscoelastic stress equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

topology_pressure_residual_check or *topo_pres_res* *(enumerated) [=standard]*
Modifies the convergence check of the residual ratio of the topology pressure equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

topology_pressure_solution_increment_check or *topo_pres_sol* *(enumerated) [=looser_by_10]*
Modifies the convergence check of the solution increment ratio of the topology pressure equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

topology_velocity_residual_check or *topo_vel_res* *(enumerated) [=standard]*
Modifies the convergence check of the residual ratio of the topology velocity equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.

looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`topology_velocity_solution_increment_check` or `topo_vel_sol` *(enumerated) [=looser_by_10]*

Modifies the convergence check of the solution increment ratio of the topology velocity equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`design_topology_filter_residual_check` or `topo_filt_res` *(enumerated) [=standard]*

Modifies the convergence check of the residual ratio of the design topology filter equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

design_topology_filter_solution_increment_check or *topo_filt_sol* *(enumerated) [=looser_by_10]*

Modifies the convergence check of the solution increment ratio of the velocity equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

mesh_displacement_residual_check or *mesh_res* *(enumerated) [=none]*

Modifies the convergence check of the residual ratio of the mesh displacement equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

mesh_displacement_solution_increment_check or *mesh_sol* *(enumerated) [=looser_by_100]*

Modifies the convergence check of the solution increment ratio of the mesh displacement equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.

looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`electric_potential_residual_check` or `elec_p_res` *(enumerated) [=looser_by_10]*

Modifies the convergence check of the residual ratio of the charge conservation equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

`electric_potential_solution_increment_check` or `elec_p_sol` *(enumerated) [=looser_by_10]*

Modifies the convergence check of the solution increment ratio of the charge conservation equation.

none	Performs no convergence check.
standard or full	Performs convergence check as is.
looser_by_10 or loose_10	Loosens the convergence check by a factor of 10.
looser_by_100 or loose_100	Loosens the convergence check by a factor of 100.
looser_by_1000 or loose_1000	Loosens the convergence check by a factor of 1000.
tighter_by_10 or tight_10	Tightens the convergence check by a factor of 10.
tighter_by_100 or tight_100	Tightens the convergence check by a factor of 100.
tighter_by_1000 or tight_1000	Tightens the convergence check by a factor of 1000.

Description

This command specifies a set of factors useful for fine control over convergence checking of various residual and solution increment ratios.

The time stepping, stagger iteration and nonlinear iteration loops check for the convergence of the solution; see `TIME_SEQUENCE` command for a description. The convergence check is performed on the residual and solution increment ratios of each equation in the problem. For the residual, the ratio of the residual norm over the norm of the forces of the systems is used. For the solution increment, the ratio of the solution increment norm over the solution norm is used. These ratios are computed separately for each solution field, such as pressure, velocity and temperature.

Each iteration loop defines a single convergence tolerance for both residual and solution increment ratios of all equations. The `CONVERGENCE_CHECK_PARAMETERS` command is then used to globally change the relative importance of one measure over another. As an example, consider a steady laminar flow problem with the following setup:

```
TIME_SEQUENCE {
   convergence_tolerance                = 1.e-3
}
CONVERGENCE_CHECK_PARAMETERS {
   pressure_residual_check              = standard
   pressure_solution_increment_check    = none
   velocity_residual_check              = standard
   velocity_solution_increment_check    = looser_by_10
}
```

Steady state is declared when the pressure and velocity residual ratios are below 10^{-3} and the velocity solution increment ratio is below 10^{-2} . The pressure solution increment ratio is ignored.

The default parameters of the `CONVERGENCE_CHECK_PARAMETERS` command are chosen based on experience. You rarely need to change them.

LINEAR_SOLVER_PARAMETERS

Specifies the parameters for fine tuning the linear equation solver.

Type

AcuSolve Command

Syntax

LINEAR_SOLVER_PARAMETERS {parameters}

Qualifier

This command has no qualifier.

Parameters

lhs_storage or *storage* *(enumerated) [=reduced_memory_sparse]*

Type of storage for the left-hand-side (LHS] matrices.

sparse	Compact sparse format.
reduced_memory_sparse or fast_sparse	Single-precision compact sparse format.

min_num_iteration_ratio *(real) >=0 <=1 [=0.5]*

Number of iterations must be at least this ratio times the previous number of iterations (of the same stagger).

pressure_precedence_factor *(real) > = 0 [=1]*

Relaxation factor to relax the pressure equation left-hand side (LHS) matrix relative to the continuity equation. This is an advanced option and is rarely changed from the default value.

num_pressure_projection_vectors or *pres_projs*

Number of available pressure projection vectors for pressure, flow, and *temperature_flow* staggers.

num_velocity_projection_vectors or *vel_projs* *(integer) >=0 [=5]*

Number of available velocity projection vectors for velocity, flow, and *temperature_flow* staggers.

num_flow_projection_vectors or *flow_projs* *(integer) >=0 [=5]*

Number of available flow projection vectors for flow and *temperature_flow* staggers.

num_temperature_projection_vectors or *temp_projs* *(integer) >=0 [=10]*

Number of available temperature projection vectors for temperature and *temperature_flow* staggers.

num_radiation_projection_vectors or *rad_projs* *(integer) >=0 [=10]*

Number of available radiation projection vectors for temperature and *temperature_flow* staggers.

num_temperature_flow_projection_vectors or *temp_flow_projs* *(integer) >=0 [=4]*

Number of available temperature-flow projection vectors for the *temperature_flow* staggers.

num_species_projection_vectors or *species_projs* *(integer) >=0 [=10]*
Number of available species projection vectors for species staggers.

num_turbulence_projection_vectors or *turb_projs* *(integer) >=0 [=10]*
Number of available turbulence projection vectors for turbulence staggers.

num_mesh_displacement_projection_vectors or *mesh_projs* *(integer) >=0 [=10]*
Number of available mesh displacement projection vectors for mesh displacement staggers.

num_viscoelastic_stress_projection_vectors or *vest_projs* *(integer) >=0 [=5]*
Number of available viscoelastic stress projection vectors for *viscoelastic_stress* staggers.

num_field_projection_vectors or *field_projs* *(integer) >=0 [=10]*
Number of available field projection vectors for field staggers.

num_levelset_projection_vectors or *ls_projs* *(integer) >=0 [=10]*
Number of available levelset projection vectors for levelset staggers.

num_redistancing_projection_vectors or *rdist_projs* *(integer) >=0 [=10]*
Number of available redistancing projection vectors for redistancing staggers.

num_transition_projection_vectors or *ttm_projs* *(integer) >=0 [=10]*
Number of available turbulence transition projection vectors for transition staggers.

num_elec_potential_projection_vectors or *elecp_projs* *(integer) >=0 [=10]*
Number of available electric potential projection vectors for charge conservation staggers.

pressure_lhs_inverse_order *(integer) >=0 [=5]*
Polynomial order used to approximate the inverse of the pressure equation LHS for pressure projection.

velocity_lhs_inverse_order *(integer) >=0 [=5]*
Polynomial order used to approximate the inverse of the velocity equation LHS for velocity projection.

flow_lhs_inverse_order *(integer) >=0 [=5]*
Polynomial order used to approximate the inverse of the flow equation LHS for flow projection.

temperature_lhs_inverse_order *(integer) >=0 [=5]*
Polynomial order used to approximate the inverse of the temperature equation LHS for temperature projection.

topology_pressure_lhs_inverse_order *(integer) >=0 [=5]*
Polynomial order used to approximate the inverse of the topology pressure equation LHS for topology pressure projection.

topology_velocity_lhs_inverse_order *(integer) >=0 [=5]*
Polynomial order used to approximate the inverse of the topology velocity equation LHS for topology velocity projection.

topology_flow_lhs_inverse_order *(integer) >=0 [=5]*
Polynomial order used to approximate the inverse of the topology flow equation LHS for topology flow projection.

design_topology_filter_lhs_inverse_order (integer) >=0 [=5]
Polynomial order used to approximate the inverse of the topology design filter equation LHS for design_topology filter projection.

pressure_regularization_factor or pres_reg_fct (real) >=0 [=1.0]
Regularization factor used within the discrete pressure Poisson equation.

velocity_regularization_factor or vel_reg_fct (real) >=0 [=0.5]
Regularization factor used for the solution of the velocity equation.

flow_regularization_factor or flow_reg_fct (real) >=0 [=0]
Regularization factor used for the solution of the flow equation.

temperature_regularization_factor or temp_reg_fct (real) >=0 [=0]
Regularization factor used for the solution of the temperature equation.

temperature_flow_regularization_factor or temp_flow_reg_fct (real) >=0 [=0]
Regularization factor used for the solution of the coupled temperature-flow equation.

species_regularization_factor or species_reg_fct (real) >=0 [=0]
Regularization factor used for the solution of the species equation.

turbulence_regularization_factor or turb_reg_fct (real) >=0 [=0]
Regularization factor used for the solution of the turbulence equation.

num_topology_pressure_projection_vectors or topo_pres_projs (integer) >=0 [=10]
Number of available pressure projection vectors for pressure, flow, and *temperature_flow* staggers.

num_topology_velocity_projection_vectors or topo_vel_projs (integer) >=0 [=5]
Number of available velocity projection vectors for velocity, flow, and *temperature_flow* staggers.

num_topology_flow_projection_vectors or topo_flow_projs (integer) >=0 [=5]
Number of available flow projection vectors for flow and *temperature_flow* staggers.

num_design_topology_filter_projection_vectors or topo_filter_projs (integer) >=0 [=10]
Number of available flow projection vectors for design topology filter staggers.

mesh_displacement_regularization_factor or mesh_reg_fct (real) >=0 [=0]
Regularization factor used for the solution of the mesh displacement equation.

viscoelastic_stress_regulation_factor or vest_reg_fct (real) >=0 [=0]
Regularization factor used for the solution of the viscoelastic stress equation.

field_regularization_factor or field_reg_fct (real) >=0 [=0]
Regularization factor used for the solution of the field equation.

levelset_regularization_factor or ls_reg_fct (real) >=0 [=0]
Regularization factor used for the solution of the levelset equation.

redistancing_regularization_factor or rdist_reg_fct (real) >=0 [=0]
Regularization factor used for the solution of the redistancing equation.

transition_regularization_factor or ttm_reg_fct (real) >=0 [=0]
Regularization factor used for the solution of the turbulence transition equation.

topology_pres_regularization_factor or *topo_pres_reg_fct* *(real) >=0 [=1]*
Regularization factor used for the solution of the topology pressure equation.

topology_velocity_regularization_factor or *topo_vel_reg_fct* *(real) >=0 [=0.5]*
Regularization factor used for the solution of the topology velocity equation.

topology_flow_regularization_factor or *topo_flow_reg_fct* *(real) >=0 [=0]*
Regularization factor used for the solution of the topology flow equation.

design_topology_filter_regularization_factor or *topo_filt_reg_fct* *(real) >=0 [=0]*
Regularization factor used for the solution of the topology design filter equation.

elec_potential_regularization_factor or *elecp_reg_fct* *(real) >=0 [=0]*
Regularization factor used for the solution of the charge conservation equation.

pressure_update_factor or *pres_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of pressure.

velocity_update_factor or *vel_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of velocity.

temperature_update_factor or *temp_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of temperature.

species_update_factor or *species_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of species.

turbulence_update_factor or *turb_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of turbulence.

topology_pressure_update_factor or *topo_pres_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of topology pressure.

topology_velocity_update_factor or *topo_vel_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of topology velocity.

design_topology_filter_update_factor or *topo_filt_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of topology design filter.

mesh_displacement_update_factor or *mesh_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of mesh displacement.

viscoelastic_stress_update_factor or *vest_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of the viscoelastic stress.

radiation_update_factor or *rad_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of radiation heat flux for enclosure radiation. For participating media radiation, this is the update factor for the solution increment of the incident radiation and the radiative intensity for the *p1_model* and *discrete_ordinate* radiation models, respectively.

field_update_factor or *field_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of field.

levelset_update_factor or *ls_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of levelset.

redistancing_update_factor or *rdist_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of redistancing.

transition_update_factor or *tts_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of turbulence transition.

elec_potential_update_factor or *elecp_upd_fct* *(real) >0 <=1 [=1.0]*
Update factor used for the solution increment of charge conservation equation.

max_pressure_update or *max_pres_upd* *(real) >=0 [=0]*
Maximum amount each pressure nodal value may be updated at each solution update of a stagger. If zero there is no limit.

max_velocity_update or *max_vel_upd* *(real) >=0 [=0]*
Maximum amount each velocity nodal value may be updated at each solution update of a stagger. If zero there is no limit.

max_temperature_update or *max_temp_upd* *(real) >=0 [=0]*
Maximum amount each temperature nodal value may be updated at each solution update of a stagger. If zero there is no limit.

max_species_update or *max_species_upd* *(real) >=0 [=0]*
Maximum amount each pressure nodal value may be updated at each solution update of a stagger. If zero there is no limit.

max_turbulence_update or *max_turb_upd* *(real) >=0 [=0]*
Maximum amount each turbulence nodal value may be updated at each solution update of a stagger. If zero there is no limit.

max_topology_pressure_update or *max_topo_pres_upd* *(real) >=0 [=0]*
Maximum amount each topology pressure nodal value may be updated at each solution update of a stagger. If zero there is no limit.

max_topology_velocity_update or *max_topo_vel_upd* *(real) >=0 [=0]*
Maximum amount each topology velocity nodal value may be updated at each solution update of a stagger. If zero there is no limit.

max_design_topology_filter_update or *max_topo_filt_upd* *(real) >=0 [=0]*
Maximum amount each design topology filter nodal value may be updated at each solution update of a stagger. If zero there is no limit.

max_mesh_displacement_update or *max_mesh_upd* *(real) >=0 [=0]*
Maximum amount each pressure nodal value may be updated at each solution update of a stagger. If zero there is no limit.

max_viscoelastic_stress_update or *max_vest_upd* *(real) >=0 [=0]*
Maximum amount each viscoelastic stress nodal value may be updated at each solution update of a stagger. If zero there is no limit.

max_radiation_update or *max_rad_upd* *(real) >=0 [=0]*
Maximum amount each radiation heat flux nodal value may be updated at each solution update of a stagger of an enclosure radiation stagger. For radiation with a participating medium, this is the maximum amount each nodal values of incident radiation or radiation intensity (for *p1_model* and

discrete_ordinate radiation models, respectively) may be updated at each solution update of a *p1_model* or *discrete_ordinate* radiation stagger. If zero there is no limit.

max_reverse_update_factor (real) >=0 [=0]
Maximum negative amount each nodal solution increment may change as a fraction of the previous solution increment. If zero there is no limit.

max_field_update or max_field_upd (real) >=0 [=0]
Maximum amount each field nodal value may be updated at each solution update of a stagger. If zero there is no limit.

max_levelset_update or max_ls_upd (real) >=0 [=0]
Maximum amount each levelset nodal value may be updated at each solution update of a stagger. If zero there is no limit.

max_redistancing_update or max_rdist_upd (real) >=0 [=0]
Maximum amount each redistancing nodal value may be updated at each solution update of a stagger. If zero there is no limit.

max_transition_update or max_ttm_upd (real) >=0 [=0]
Maximum amount each turbulence transition nodal value may be updated at each solution update of a stagger. If zero there is no limit.

max_elec_potential_update or max_elecp_upd (real) >=0 [=0]
Maximum amount each electric potential nodal value may be updated at each solution update of a stagger. If zero there is no limit.

Description

This command specifies the linear equation solver parameters that are common to all staggers.

All left-hand-side (LHS) matrices are stored in the Harwell-Boeing "compressed column storage" format. LHS matrices are formed when creating the residuals. Matrix-vector products within the linear iterations are performed using the stored matrices. On all machines currently supported, the sparse LHS storage uses 64-bit representation "double precision" for the LHS matrix elements, while reduced_memory_sparse used 32- bit representation "single precision". The latter reduces overall memory usage by about one-third and reduces CPU usage by a lesser amount. For some difficult problems, the use of the reduced_memory_sparse storage option may increase the number of nonlinear iterations due to the poorer representation of the LHS.

Sometimes the number of iterations used by a particular stagger wildly oscillates from one linear solve to the next. The *min_num_iteration_ratio* is used to dampen these oscillations by requiring that the number of iterations used in the current solve is at least this ratio times the number of iterations used by the previous solve.

The numbers of projection (deflation) vectors available to the staggers are specified by the *num_pressure_projection_vectors* through *num_mesh_displacement_projection_vectors* parameters. Projection vectors are allocated and used only when projection is requested by an active stagger; see the STAGGER command for more information. If two or more staggers with the same equation request projection, the same vectors are used for both staggers. This includes the pressure projection vectors used by a pressure stagger and pressure projection in a flow stagger.

Larger numbers of projection vectors provide more powerful solution processes, but at the expense of higher CPU time per iteration and more memory to store the projection vectors. Typically only a few projection vectors are used.

The *pressure_lhs_inverse_order* through *temperature_lhs_inverse_order* parameters define the order of the polynomial used to approximate the inverse of the corresponding LHS for the projection operations. These parameters rarely need to be changed.

The regularization parameters specify the amount of relaxation added to the LHS matrix of each equation. More precisely, a fraction of the diagonal of a LHS matrix is added back to the LHS matrix. This fraction is given by the regularization parameters. The regularization makes the LHS matrix more diagonally dominant. This in turn improves the conditioning of the LHS matrix, requiring fewer linear iterations. More importantly the solution of the resulting linear system provides smaller changes to the stagger's solution field, making the stagger approach more stable. On the other hand, regularization deteriorates nonlinear convergence and typically requires more work overall.

The pressure regularization is used by both the pressure stagger, as a part of the segregated solution, and the flow stagger, as a part of the pressure projection. Both cases require a certain amount of regularization, as given by the defaults. The regularization of pressure projection does not translate to regularization of the flow equations. The velocity regularization is used by the velocity stagger as a part of the segregated solver. The segregated solver is rarely used. All other staggers are typically solved with no regularization in order to get the fastest nonlinear convergence. However, if there are several staggers, for example, flow, turbulence and temperature, then the nonlinear iterations may sometimes behave like a segregated solver, in which case regularization on some or all staggers would be recommended.

Similarly, the update factor parameters may be used to nonlinearly stabilize a solution variable. These factors have no effect on the linear solutions, but are used to scale solution increments before solutions are updated.

The parameters *max_pressure_update* through *max_radiation_update* may be used to stabilize a nonlinear solution strategy by limiting the absolute size of a nodal solution increment. Each of these parameters has the same dimensions as the corresponding variable. A value of zero indicates that no limit is placed on the solution increment, other than the limits imposed by the other parameters in this command.

Oscillations in an iterative strategy may be controlled through nonzero values of *max_reverse_update_factor*. For each variable at each node, the following constraint is imposed:

$$\Delta f_{i+1} \geq -\lambda \Delta f_i \qquad (48)$$

where Δf_{i+1} is the new solution increment, Δf_i is the previous solution increment, and λ is given by *max_reverse_update_factor*. For example, if *max_reverse_update_factor*=0.5 and the temperature of a node changed from 100 to 200 in one nonlinear iteration, the minimum allowable temperature on the next iteration is 150.

In summary, you should rarely change the regularization and update parameters given by the defaults.

STAGGER

Specifies a stagger for the solution of an equation.

Type

AcuSolve Command

Syntax

```
STAGGER("name") {parameters...}
```

Qualifier

User-given name.

Parameters

equation (enumerated) [=none]

Equation to be solved.

none	No equation solved.
pressure or pres	Discrete pressure Poisson equation.
velocity or vel	Momentum equations.
flow	Flow equations - momentum and continuity equations.
compressible_flow	Compressible equations.
temperature or temp	Energy equation.
radiation or rad	Radiation heat flux equation. Also, p1 incident radiation equation or discrete ordinates method for solving the radiative transfer equation.
temperature_flow or temp_flow	Coupled energy and flow equations.
species_1 or spec1	Species 1 transport equation.
species_2 or spec2	Species 2 transport equation.
species_3 or spec3	Species 3 transport equation.
species_4 or spec4	Species 4 transport equation.
species_5 or spec5	Species 5 transport equation.
species_6 or spec6	Species 6 transport equation.
species_7 or spec7	Species 7 transport equation.

species_8 or spec8	Species 8 transport equation.
species_9 or spec9	Species 9 transport equation.
field	Field transport equation. Used with phase field multi field transport model.
levelset	Level set transport equation. Used with level set multi field transport model.
levelset_redistancing	Level set sharpening equation. Used with level set multi field transport model.
turbulence or turb	Turbulence equation. Valid only with Spalart-Allmaras and Detached Eddy Simulation turbulence models.
kinetic_energy or tke	Turbulent kinetic energy equation. Valid only with SST, k-omega, and SST-DES turbulence models.
eddy_frequency or tomega	Eddy frequency equation. Valid only with SST, k-omega, and SST-DES turbulence models.
dissipation_rate or teps	Dissipation rate equation. Valid only with k-epsilon, realizable k-epsilon and RNG k-epsilon turbulence models.
intermittency or tintc	Turbulence intermittency equation. Valid only with the Gamma-Re_theta and Gamma turbulent transition models.
transition_re_theta or treth	Critical momentum thickness Reynolds Number equation. Valid only with the Gamma-Re_theta turbulent transition models.
viscoelastic or vest	Viscoelastic stress equations.
topology_pressure or topopres	Adjoint (topology) pressure equation.
topology_velocity or topovel	Adjoint (topology) momentum equations.
topology_flow	Adjoint (topology) flow equations – adjoint momentum and adjoint pressure equations.
mesh_displacement or mesh	Mesh displacement equation.
external_code	Solution for solid/structural equations provided by an external code.
particle	Solution for particle equations by AcuTrace.

explicit_type (enumerated) [=none]

Type of explicit strategy for this stagger.

none	Standard implicit strategy used.

advection or convection	Advection term treated explicitly.

min_stagger_iterations or *min_stg_iters* *(integer) >=0 [=1]*
Minimum number of nonlinear iterations for this stagger.

max_stagger_iterations or *max_stg_iters* *(integer) >=0 [=1]*
Maximum number of nonlinear iterations of this stagger. If zero, this option is ignored.

convergence_tolerance or *conv_tol* *(real) >=0 [=1.e-4]*
Convergence tolerance to end nonlinear iterations of this stagger.

lhs_update_frequency or *lhs_freq* *(integer) >=0 [=1]*
The nonlinear iteration frequency at which the left-hand-side (LHS) matrix of this stagger is discarded. If zero, this option is ignored.

linear_solver or *les* *(enumerated) [=gmres]*
Linear equation solver of this stagger.

gmres	GMRES iterative solver. May be used for all but the pressure and mesh equations.
bicgstab	BiCGStab iterative solver. May be used for all but the pressure and mesh equations.
conjugate_gradient or cg	Conjugate-Gradient iterative solver. May be used only for the pressure and mesh equations.

min_linear_solver_iterations or *min_les_iters* *(integer) >=0 [=1]*
Minimum number of iterations of the linear solution.

max_linear_solver_iterations or *max_les_iters* *(integer) >0 [=1000]*
Maximum number of iterations of the linear solution.

num_krylov_vectors or *kvecs* *(integer) >0 [=10]*
Number of Krylov vectors used by GMRES and BiCGStab.

linear_solver_tolerance or *les_tol* *(real) >=0 [=0.1]*
Linear solver convergence tolerance.

pressure_projection_tolerance or *pres_tol* *(real) >=0 [=0.1]*
Convergence tolerance of the pressure projection algorithm used for the flow or temperature_flow equation. Used if pressure_projection is turned on.

velocity_projection_tolerance or *vel_tol* *(real) >=0 [=0.1]*
Convergence tolerance of the velocity projection algorithm used for the flow or temperature_flow equation. Used if *velocity_projection* is turned on.

temperature_projection_tolerance or *temp_tol* *(real) >=0 [=0.1]*
Convergence tolerance of the temperature projection algorithm used for the temperature_flow equation. Used if *temperature_projection* is turned on.

projection or *proj* *(boolean) [=on]*
Flag specifying whether or not to use projection (deflation) within the linear solution.

pressure_projection or *pres_proj* *(boolean) [=on]*
: Flag specifying whether or not to use pressure projection within the linear solver for the solution of the flow or temperature_flow equation.

velocity_projection or *vel_proj* *(boolean) [=off]*
: Flag specifying whether or not to use velocity projection within the linear solver for the solution of the flow or temperature_flow equation.

temperature_projection or *temp_proj* *(boolean) [=off]*
: Flag specifying whether or not to use temperature projection within the linear solver for the solution of the temperature_flow equation.

pressure_algebraic_multigrid or *pres_amg* *(boolean) [=off]*
: Flag specifying whether or not to use pressure algebraic multigrid preconditioning for the preconditioned conjugated gradient solver. Used with *pressure*, *flow*, and *temperature_flow* equations.

pressure_algebraic_multigrid_update_frequency *(integer) >0 [=0]*
: Frequency to update the pressure algebraic multigrid parameters.

algebraic_multigrid or *amg* *(boolean) [=off]*
: Flag specifying whether or not to use algebraic multigrid preconditioning. Used with *flow*, *turbulence*, *temperature*, *species*, *mesh_displacement*, and *viscoelastic* equations.

algebraic_multigrid_update_frequency *(integer) >0 [=0]*
: Frequency to update the algebraic multigrid parameters. Used with *flow*, *turbulence*, *temperature*, *species*, *mesh_displacement*, and *viscoelastic* equations when *algebraic_multigrid*=on.

velocity_algebraic_multigrid or *vel_amg* *(boolean) [=off]*
: Flag specifying whether or not to use algebraic multigrid preconditioning for the GMRES solver when solving the velocity equations. Used with *flow* and *temperature_flow* equations when the *velocity_projection*=on.

velocity_algebraic_multigrid_update_frequency *(integer) >0 [=0]*
: Frequency to update the velocity algebraic multigrid parameters. Used with *flow* and *temperature_flow* equations when *velocity_projection*=on and when *velocity_algebraic_multigrid*=on.

temperature_algebraic_multigrid or *temp_amg* *(boolean) [=off]*
: Flag specifying whether or not to use algebraic multigrid preconditioning for the GMRES solver when solving the temperature equation. Used with *temperature_flow* equations when *temperature_projection*=on.

temperature_algebraic_multigrid_update_frequency *(integer) >0 [=0]*
: Frequency to update the temperature algebraic multigrid parameters. Used with *temperature_flow* equations when *temperature_projection*=on and *temperature_algebraic_multigrid*=on.

staggers or *stgs* *(list) (no default)*
: List of sub-staggers to be executed. The sub-staggers, if any, are executed after the main equation of the stagger is solved.

Description

This command specifies the nonlinear iteration and linear solver parameters for the solution of an equation. This command also accommodates execution of other staggers. For a detailed description of time stepping and nonlinear solution strategy, see the TIME_SEQUENCE command.

In order for a stagger to be executed, it must be referenced directly or indirectly by the TIME_SEQUENCE command. Direct reference is accomplished by adding the user-given name of the STAGGER command to the list of staggers in the *staggers* parameter of the TIME_SEQUENCE command. For example, in the following:

```
TIME_SEQUENCE {
    staggers                 = { "flow" }
}
STAGGER( "flow" ) {
   equation                  = flow
}
STAGGER( "temp" ) {
   equation                  = temperature
}
```

The "flow" stagger is directly referenced, therefore its equation is solved. The "temp" stagger is not referenced, therefore its equation is not solved.

A stagger may also be indirectly referenced through another (referenced) stagger. For example,

```
EQUATION {
   flow                              = navier_stokes
   multi_field                       = levelset
   fields                            = {"water","air";}
}
TIME_SEQUENCE {
   staggers = { "flow_levelset", "redistancing" }
   ...
}
STAGGER( "flow_levelset" ) {
   equation                           = none
   min_stagger_iterations             = 2
   max_stagger_iterations             = 6
   ...
   staggers                           = { "flow","levelset"}
}
STAGGER( "flow" ) {
   equation                           = flow
   ...
}
STAGGER( "levelset" ) {
   equation                           = levelset
   ...
}
STAGGER( "redistancing" ) {
   equation                           = levelset_redistancing
   ...
}
```

This example demonstrates the use of this feature for an immiscible multi-field simulation where it is desirable to iterate the flow and scalar fields to convergence before performing any sharpening of the interface.

The turbulence equation types *dynamic_model* and *large_eddy_simulation* are not solved through staggers, but rather recomputed at the beginning of each nonlinear iteration. The turbulence equations for the SST, k-omega, BSL k-omega, SST-DES, k-epsilon, realizable k-epsilon and RNG k-epsilon turbulence models are solved in a segregated fashion, requiring a separate stagger for each equation. In the case of the k-omega based models, the staggers can be specified as follows:

```
STAGGER( "eddy_frequency" ) {
   equation                          = eddy_frequency
   lhs_update_frequency              = 1
   linear_solver                     = gmres
   linear_solver_tolerance           = .001
   projection                        = off
}
STAGGER( "kinetic_energy" ) {
   equation                         = kinetic_energy
   lhs_update_frequency             = 1
   linear_solver                    = gmres
   linear_solver_tolerance          = .01
   projection                       = off
}
STAGGER( "turbulence" ) {
   equation                         = none
   min_stagger_iterations           = 1
   max_stagger_iterations           = 3
   staggers                         = {"eddy_frequency","kinetic_energy"}
}
```

The eddy frequency-equation does not have a strong dependence on the kinetic energy equation. However, the kinetic energy source term is strongly impacted by changes in eddy frequency. Therefore, the kinetic energy is typically solved after the eddy frequency in the stagger sequence.

Additionally, the kinetic energy and eddy frequency are solved multiple times for each solve of the flow equations. Experience has shown that this leads to the most optimal convergence rate:

```
TIME_SEQUENCE{
 staggers = {   "flow",
                "turbulence" }
}
```

The eddy frequency stagger represents a special case in which the transport quantity used by the stagger deviates from the standard equation for eddy frequency. Instead of solving directly for ω, AcuSolve constructs the transport equation in terms of 1/sqrt(ω). This change of variables does not change the solution produced by the SST and k-omega turbulence models. However, the change of variables does lead to significant improvements in stability. The actual transport quantity (1/sqrt(ω)) is written as *sqrt_eddy_period* in the nodal field outputs. The *eddy_frequency* variable that appears in the nodal outputs is computed based on the solution for *sqrt_eddy_period*.

For k-epsilon based models, a similar solution strategy is used. The dissipation rate is solved before the kinetic energy and the turbulence equations are solved multiple times for each flow solve:

```
STAGGER( "dissipation_rate" ) {
   equation                          = dissipation_rate
   lhs_update_frequency              = 1
   linear_solver                     = gmres
   linear_solver_tolerance           = .001
   projection                        = off
}
```

```
STAGGER( "kinetic_energy" ) {
   equation                          = kinetic_energy
   lhs_update_frequency              = 1
   linear_solver                     = gmres
   linear_solver_tolerance           = .01
   projection                        = off
}
STAGGER( "turbulence" ) {
   equation                          = none
   min_stagger_iterations            = 1
   max_stagger_iterations            = 3
   staggers = {"dissipation_rate","kinetic_energy"}
}
TIME_SEQUENCE{
   staggers                          = {   "flow",
                                           turbulence"}
}
```

The viscoelastic stress equations are solved as a coupled set, and therefore only require a single stagger. For example:

```
STAGGER( "viscoelastic" ) {
   equation                       = viscoelastic
   lhs_update_frequency           = 1
   linear_solver                  = gmres
   linear_solver_tolerance        = .01
   projection                     = off
}
```

The equation of a referenced stagger must be set via the EQUATION command. On the other hand, an equation set by the EQUATION command does not need to be referenced by any stagger. In this case, the solution field(s) of such equations simply retain their initial values throughout the analysis. The initial values come from either the NODAL_INITIAL_CONDITION command or the restart file.

Each stagger loops over a number of nonlinear iterations, within which the residual and optionally the LHS matrix of the stagger are formed, the resulting linear equation system is solved, the corresponding solution field is updated and its sub-staggers are executed. A minimum of *min_stagger_iterations* and a maximum of *max_stagger_iterations* nonlinear iterations are performed. If the convergence measures are less than the *convergence_tolerance* and *min_stagger_iterations* iterations performed, the stagger loop is done. See the TIME_SEQUENCE command for definitions.

Setting the *explicit_type* to advection removes the advection term from the LHS. It has no effect on the residual but it imposes a CFL limit of about one for stability. For transient problems where small time increments are used anyway for accuracy, this strategy can greatly reduce the CPU required. It is most effective when used together with a strategy that reforms the LHS less often and has just one stagger iteration per time step. However, explicit strategies are far less robust than the standard implicit strategies so you should approach this parameter with caution.

The iterative solvers gmres, bicgstab and conjugate_gradient are used to solve the linear equation systems resulting from the linearization of the discrete finite element formulation. The first two solvers are used for non-symmetric equation systems, while conjugate_gradient is used for the symmetric discrete pressure Poisson and mesh equations. In all cases, a minimum of *min_linear_solver_iterations* and a maximum of *max_linear_solver_iterations* linear iterations are performed. The linear solver may terminate earlier if the residual norm, normalized

with respect to the initial residual norm, reaches the *linear_solver_tolerance* and the *min_linear_solver_iterations* criteria is satisfied.

Both gmres and bicgstab algorithms require a set of Krylov vectors, whose sizes are defined by *num_krylov_vectors*. Larger Krylov spaces typically increase the robustness and the rate of convergence of the algorithm, but require more memory. For simple problems, the Krylov space of 10 for the flow equation is sufficient. For harder problems, a Krylov space of 40 or larger may be required.

The memory for the Krylov vectors of the staggers is overlapped. This memory is proportional to the product of the *num_krylov_vectors*, the number of degrees of freedom (DOF) per node and the number of nodes in the system. The flow stagger has four dofs per node, the *velocity* stagger has three DOFs per node, the *viscoelastic* stagger has six DOFs per node, and all others have one DOF per node. Optimum memory usage is achieved when the *num_krylov_vectors* of the staggers keeps the same proportions. For example,

```
STAGGER( "flow" ) {
   equation                          = flow
   num_krylov_vectors                = 15
}
STAGGER( "turb" ) {
   equation                          = turbulence
   num_krylov_vectors                = 60
}
```

Each of the three linear solvers are equipped with a projection, also known as deflation, algorithm which accelerates the convergence rate of the linear solver at the expense of memory and computational cost per iteration. Projection algorithms work by building information about the linear equation system in the course of solving each linear system. This information is then used to accelerate convergence. The projection is turned on and off through the *projection* parameter. The dimension of the projection vectors is given by the LINEAR_SOLVER_PARAMETERS command.

Experience has shown that projection helps difficult problems while it is more expensive for simple ones. For problems where the LHS matrix does not change significantly from one nonlinear iteration to the next, projection is very effective. Use of projection with a tight convergence tolerance is recommended for the solution of the *temperature* equation when solving a conjugate heat transfer problem. For example,

```
STAGGER( "temp" ) {
   equation                          = temperature
   min_stagger_iterations            = 1
   max_stagger_iterations            = 1
   convergence_tolerance             = 1.e-3
   lhs_update_frequency              = 1
   linear_solver                     = gmres
   min_linear_solver_iterations      = 10
   max_linear_solver_iterations      = 400
   num_krylov_vectors                = 40
   linear_solver_tolerance           = 1.e-3
   projection                        = on
}
LINEAR_SOLVER_PARAMETERS {
   num_temperature_projection_vectors  = 10
}
```

A special and proprietary variant of the GMRES and BiCGStab algorithms is used for solving the *flow* equation. This algorithm performs pressure and velocity projections to greatly enhance the robustness and convergence of the linear solver. These projections are controlled by the *pressure_projection* and *velocity_projection* parameters and their associated tolerance parameters. The tighter the tolerance of the pressure projection, the better the continuity equation can be satisfied. Typically, *pressure_projection_tolerance* is set an order of magnitude tighter than *linear_solver_tolerance*, while *velocity_projection_tolerance* is set equal to *linear_solver_tolerance*. The number of pressure and velocity projection vectors is set in the LINEAR_SOLVER_PARAMETERS command. The following is a typical *flow* stagger:

```
STAGGER( "flow" ) {
   equation                              = flow
   min_stagger_iterations                = 1
   max_stagger_iterations                = 1
   convergence_tolerance                 = 1.e-3
   lhs_update_frequency                  = 1
   linear_solver                         = gmres
   min_linear_solver_iterations          = 10
   max_linear_solver_iterations          = 400
   num_krylov_vectors                    = 10
   linear_solver_tolerance               = 0.1
   pressure_projection_tolerance         = 0.01
   pressure_projection                   = on
   velocity_projection_tolerance         = 0.1
   velocity_projection                   = on
   projection                            = off
}
LINEAR_SOLVER_PARAMETERS {
   num_pressure_projection_vectors       = 10
   num_velocity_projection_vectors       = 4
   num_flow_projection_vectors           = 4
}
```

For hard problems the *num_krylov_vectors* can be increased and the *projection* flag turned on. Experience has shown that velocity projection is rarely helpful, with one exception being for flows with highly nonlinear non-Newtonian viscosity models.

Pressure algebraic multigrid preconditioning can be added to the preconditioned conjugated gradient solver for the *pressure*, *flow*, and *temperature_flow* equations. The frequency of updating the algebraic multigrid state variables is controlled with *pressure_algebraic_multigrid_update_frequency*. See *ALGEBRAIC_MULTIGRID_PARAMETERS* to adjust the other parameters of the method. For example,

```
STAGGER( "flow" ) {
   equation                                      = flow
   ...
   pressure_algebraic_multigrid                  = on
   pressure_projection                           = on
   pressure_algebraic_multigrid_update_frequency = 0
}
```

The *pressure* and *velocity* stagger equations are used to solve a flow problem via a segregated solver strategy. The *pressure* equation corresponds to a discrete pressure Poisson equation, which updates both the pressure and velocity equations to satisfy the continuity equation. The *velocity* equation corresponds to a regularized momentum solver for the velocity field. These equations, collectively

known as the segregated solver, are rarely used. Instead the use of a *flow* stagger is recommended. The *flow* stagger is significantly more stable and has a much faster nonlinear convergence rate.

For flows where the temperature, velocity, and pressure are all tightly coupled together, such as free convection flows at high Grashof numbers, a solution strategy that contains separate *flow* and *temperature* staggers can behave much like a segregated solver. That is, because key coupling terms are neglected in the linearization of the system, the nonlinear convergence may be a lot slower than you would expect. In the *temperature_flow* equation, these coupling terms are included. This results in nonlinear convergence comparable to that for isothermal flows. However, the memory usage will be about 50 percent greater than for the two-stagger strategy and the cost for each linear solve will typically be much greater due to the complexity and stiffness of the matrix system. In addition to velocity and pressure projections, the algorithm includes temperature projection. This projection is controlled by the *temperature_projection* and *temperature_projection_tolerance* parameters, and is recommended for most cases. The following is a typical *temperature_flow* stagger:

```
STAGGER( "temperature_flow" ) {
   equation                                = temperature_flow
   min_stagger_iterations                  = 1
   max_stagger_iterations                  = 1
   convergence_tolerance                   = 1.e-3
   lhs_update_frequency                    = 1
   linear_solver                           = gmres
   min_linear_solver_iterations            = 10
   max_linear_solver_iterations            = 400
   num_krylov_vectors                      = 10
   linear_solver_tolerance                 = 0.1
   pressure_projection_tolerance           = 0.01
   pressure_projection                     = on
   velocity_projection_tolerance           = 0.1
   velocity_projection                     = on
   temperature_projection_tolerance        = 0.001
   temperature_projection                  = on
   projection                              = off
}
LINEAR_SOLVER_PARAMETERS {
   num_pressure_projection_vectors         = 10
   num_velocity_projection_vectors         = 4
   num_temperature_flow_projection_vectors = 4
   num_temperature_projection_vectors      = 4
}
```

Since the cost, per linear iteration, of solving the *turbulence* and *species* linear equation system is significantly less than that of the *flow* equation, it is recommended to have tighter tolerances for these equations. Also, as mentioned above, the *num_krylov_vectors* may be four times larger than the one for the flow equation without increasing the memory usage. Hence, the stagger for the *turbulence* equation may be set up as follows:

```
STAGGER( "turb" ) {
   equation                          = turbulence
   min_stagger_iterations            = 1
   max_stagger_iterations            = 1
   convergence_tolerance             = 1.e-3
   lhs_update_frequency              = 1
   linear_solver                     = gmres
   min_linear_solver_iterations      = 10
   max_linear_solver_iterations      = 400
   num_krylov_vectors                = 40
```

```
   linear_solver_tolerance             = 0.01
   projection                          = off
}
LINEAR_SOLVER_PARAMETERS {
   num_turbulence_projection_vectors   = 5
}
```

The stagger for each species may be set up in a similar fashion.

If the radiation equation is of type enclosure, the heat fluxes from the radiation stagger are automatically incorporated into the temperature or temperature_flow stagger. If the radiation equation is of type *p1_model* or *discrete_ordinate*, radiative heat sources and sinks are automatically incorporated as a source term in the energy equation when *temperature* = advective_diffusive. Thus the radiation stagger looks very similar to the other scalar equation staggers. For example,

```
STAGGER( "radiation" ) {
    equation = radiation
    min_stagger_iterations = 1
    max_stagger_iterations = 1
    convergence_tolerance = 1.e-3
    lhs_update_frequency = 1
    linear_solver = gmres
    min_linear_solver_iterations = 10
    max_linear_solver_iterations = 400
    num_krylov_vectors = 40
    linear_solver_tolerance = 0.01
    projection = on
}
LINEAR_SOLVER_PARAMETERS {
    num_radiation_projection_vectors = 5
}
```

For the discrete ordinates model the above would include a list of sub-staggers, which represent the staggers for the individual ordinate directions. For example, if the *radiation_quadrature* = S2 then the radiation stagger would be:

```
STAGGER( "radiation" ) {
    equation = radiation
    min_stagger_iterations = 1
    max_stagger_iterations = 1
    convergence_tolerance = 1.e-3
    lhs_update_frequency = 1
    linear_solver = gmres
    min_linear_solver_iterations = 10
    max_linear_solver_iterations = 400
    num_krylov_vectors = 40
    linear_solver_tolerance = 0.01
    projection = on
    staggers = {"discrete_ordinate_1",
                "discrete_ordinate_2",
                "discrete_ordinate_3",
                "discrete_ordinate_4",
                "discrete_ordinate_5",
                "discrete_ordinate_6",
                "discrete_ordinate_7",
                "discrete_ordinate_8" } ;
}
```

The sub-stagger discrete_ordinate_N, where N is the ordinate number, for example, 1-8 for S2, would then be defined for each ordinate direction. For example, the first ordinate direction:

```
STAGGER( "discrete_ordinate_1" ) {
      equation = radiation ;
      discrete_ordinate_direction = 1 ;
      explicit_type = none ;
      min_stagger_iterations = 1 ;
      max_stagger_iterations = 1 ;
      convergence_tolerance = 0.1 ;
      lhs_update_frequency = 1 ;
      linear_solver = gmres ;
      min_linear_solver_iterations = 10 ;
      max_linear_solver_iterations = 1000 ;
      num_krylov_vectors = 40 ;
      linear_solver_tolerance = 0.1 ;
      pressure_projection_tolerance = 0.1 ;
      velocity_projection_tolerance = 0.1 ;
      temperature_projection_tolerance = 0.1 ;
      projection = on ;
}
```

For solving the solid/structural part of a Direct Coupling Fluid Structural Interaction (DC-FSI) problem, set *equation* to external_code. This transfers control to an external code for this part of the solution. For example,

```
STAGGER( "abaqus" ) {
 equation                              = external_code
 min_stagger_iterations                = 1
 max_stagger_iterations                = 1
}
```

The above command specifies the Conventional Sequential Staggered (CSS) explicit coupling strategy. If *max_stagger_iterations* is larger than one then the Multi-Iterative Coupling (MIC) implicit strategy is used. See the EXTERNAL_CODE command for more information on solving DC-FSI problems.

For solving the particle part of a coupled particle-flow problem, set *equation* to particle. This transfers control to AcuTrace for this part of the solution. For example,

```
STAGGER( "particle" ) {
 equation                              = particle
}
```

TIME_ INCREMENT

Specifies the time increment values and control parameters.

Type

AcuSolve Command

Syntax

TIME_INCREMENT {parameters}

Qualifier

This command has no qualifier.

Parameters

initial_time_increment or *init_dt* *(real) >=0 [=0]*
: Initial value of the time increment for auto time increment control, or the constant value of the time increment for a fixed time increment. If zero, it is set to the largest time increment that satisfies the *initial_cfl_number* condition.

previous_time_increment *(real) >=0 [=0]*
: Time increment of the previous time step. If zero, it is automatically set for restart runs.

flow_time_increment *(real) >=0 [=0]*
: Time increment associated with the flow field. Needed for frozen flow simulations.

auto_time_increment or *auto_dt* *(boolean) [=on]*
: Flag specifying whether or not to automatically control the time increment.

local_time_increment or *local_dt* *(boolean) [=off]*
: Flag specifying whether or not to use the local time stepping algorithm. If on, a different time increment is used at each quadrature point. This option should only be used for steady-state solutions.

min_time_increment or *min_dt* *(real) >=0 [=0]*
: Minimum time increment when using the auto time increment option.

max_time_increment or *max_dt* *(real) >=0 [=0]*
: Maximum time increment when using the auto time increment option.

cfl_control *(boolean) [=on]*
: Flag specifying whether or not to control the time increment based on the CFL number. This option is used only with the auto time increment option.

cfl_number or *cfl* *(real) >0 [=100]*
: CFL number used with the CFL control and local time increment options.

min_cfl_number or *min_cfl* *(real) >=0 [=0]*
: Minimum value of the CFL number when using the auto time increment option with CFL control.

initial_cfl_number or *init_cfl* *(real) >=0 [=1]*
: Initial value of the CFL number. This option is used to determine an initial time increment when *initial_time_increment* is zero.

time_increment_decrease_factor or *dt_dec_fct* *(real) >0 <=1 [=0.25]*
: Factor used to reduce the time increment when a time step needs to be re-done due to lack of convergence. This option is used only with the auto time increment option.

time_increment_increase_factor or *dt_inc_fct* *(real) >= 1 [=1.25]*
: Factor used to increase the time increment when the previous time step is converged. This option is used only with the auto time increment option.

time_increment_increase_delay or *dt_inc_delay* *(integer) >=0 [=4]*
: Number of time steps to wait, after a time increment reduction and re-do of a time step, before allowing the time increment to increase. This option is used only with the auto time increment option.

min_time_increment_ratio or *min_dt_ratio* *(real) >=0 <=1 [=0.1])*
: The minimum ratio of two consecutive time increments allowed when the previous time step is repeated with a smaller time increment. This option is used only with the auto time increment option.

multiplier_function *(string) [=none]*
: User-given name of the multiplier function for scaling the time increment or CFL number, as appropriate. If none, no scaling is performed.

Description

This command specifies the time increment options for the time marching solution of a problem. Two time increment strategies are available: fixed and auto time increment. In addition, a local time stepping method may be used with either strategy. A fixed time increment strategy with a global time stepping method may be set up as follows:

```
TIME_INCREMENT {
   initial_time_increment      = 1.0
   auto_time_increment         = off
   local_time_increment        = off
}
```

Here a fixed time increment of *initial_time_increment* is used for all time steps. If the time increment is set to zero, such as in

```
TIME_INCREMENT {
   initial_time_increment      = 0
   auto_time_increment         = off
   local_time_increment        = off
   initial_cfl_number          = 10
}
```

then the initial conditions are made to satisfy the boundary conditions of the first time step and the maximum time increment that satisfies the *initial_cfl_number* is computed. This time increment is used for all time steps. If this time increment happens to be zero, then a time increment of one is used.

The CFL number is defined based on the velocity, u, time increment, Δt and a measure of element size, h, as follows:

$$\mathrm{CFL} = \frac{u\Delta t}{h}$$

When local time stepping with fixed time increment strategy is used, such as in:

```
TIME_INCREMENT {
   initial_time_increment      = 1
   auto_time_increment         = off
   local_time_increment        = on
   cfl_number                  = 100.
}
```

a different time increment is used at each quadrature point of each element. This time increment is such that the CFL number at the quadrature is equal to *cfl_number*. In this case, the *initial_time_increment* has little meaning.

Note: The local time stepping method may only be used for steady-state solutions.

An auto time increment strategy with a global time stepping method may be set up as follows:

```
TIME_INCREMENT {
   initial_time_increment              = 1.0
   auto_time_increment                 = on
   local_time_increment                = off
   min_time_increment                  = 0.001
   max_time_increment                  = 10.0
   cfl_control                         = on
   cfl_number                          = 100.
   min_cfl_number                      = 1.
   time_increment_decrease_factor      = 0.25
   time_increment_increase_factor      = 1.25
   time_increment_increase_delay       = 4
   min_time_increment_ratio            = 0.1
}
```

Here an initial time increment of *initial_time_increment* is used. The time increment is allowed to increase up to *max_time_increment* and decrease to *min_time_increment*. Every time step may increase the time increment by no more than a factor of *time_increment_increase_factor*. This growth restriction is also imposed on the CFL number of the time steps. If the cfl_control option is used, the time increment value is restricted so that the CFL number may not exceed *cfl_number* nor fall below *min_cfl_number* at all quadrature points in the problem.

A time step may be redone with a new time increment reduced by a factor of *time_increment_decrease_factor* if any of the following conditions occur:

- A transient problem is being solved (see parameter mode of ANALYSIS command) and the last set of convergence measures of the stagger loop are greater than the *stagger_convergence_tolerance* parameter of the TIME_SEQUENCE command. This condition is ignored if the time increment is already at *min_time_increment*.
- The computed time increment to be used for the next time step is less than *min_time_increment_ratio* times the one used for the current time step. This condition typically occurs when the velocity field increases very rapidly, indicating some kind of numerical instability.

- Any of the algorithms in the code detects errors that may cause severe inaccuracies in the problem. This condition is ignored if the time increment is already at *min_time_increment*.

Once a time step is repeated with a reduced time increment, a minimum of *time_increment_increase_delay* time steps are solved before the time increment is allowed to increase again.

An auto time increment strategy with local time stepping method may be set up similarly to the above case, with *local_time_increment* turned on:

```
TIME_INCREMENT {
   initial_time_increment              = 1.0
   auto_time_increment                 = on
   local_time_increment                = on
   min_time_increment                  = 0.001
   max_time_increment                  = 10.0
   cfl_control                         = on
   cfl_number                          = 100.
   min_cfl_number                      = 1.
   time_increment_decrease_factor      = 0.25
   time_increment_increase_factor      = 1.25
   time_increment_increase_delay       = 4
   min_time_increment_ratio            = 0.1
}
```

In contrast to the fixed time increment case, a different time increment is used at each quadrature point of each element. This time increment is such that the CFL number at the quadrature is equal to a running CFL number. This CFL number is controlled along with the time increment, starting with the *initial_cfl_number*. Again, this option should only be used for steady-state problems.

The *flow_time_increment* parameter is needed when a previously computed flow field is used for the simulation of scalar fields. Typically the flow field is steady and the scalar simulation is transient, but any combination of steady/transient simulations is allowed. The flow field must be computed with a global time stepping strategy and the value supplied for *flow_time_increment* must be equal to the value of the time increment when the flow field was saved. If a different value for *flow_time_increment* is used then the conservation properties of the scalar field will be lost. For example, a steady flow field may be solved with

```
ANALYSIS {
   title                               = "Turbulent steady-state solution"
   type                                = steady
}
EQUATION {
   flow                                = navier_stokes
   turbulence                          = spalart_allmaras
}
TIME_INCREMENT {
   initial_time_increment              = 10.0
   auto_time_increment                 = off
   local_time_increment                = off
}
...
```

If AcuTrans is used to create nodal solution files, then a transient thermal simulation can then be performed using

```
ANALYSIS {
```

```
   title                                = "Transient thermal solution"
   type                                 = transient
}
EQUATION {
   temperature                          = advective_diffusive
}
   NODAL_INITIAL_CONDITION( velocity ) {
   nodal_values                         = Read( "velocity.nic" )
}
NODAL_INITIAL_CONDITION( pressure ) {
   nodal_values                         = Read( "pressure.nic" )
}
NODAL_INITIAL_CONDITION( eddy_viscosity ) {
   nodal_values                         = Read( "eddy_viscosity.nic"
}
TIME_INCREMENT {
   initial_time_increment               = 0.1
   flow_time_increment                  = 10.0
   auto_time_increment                  = on
   local_time_increment                 = off
   cfl_number                           = 100.
}
...
```

If the RESTART command was used instead of the NODAL_INITIAL_CONDITION commands, then *flow_time_increment* does not need to be specified since the appropriate value is saved in the database and used by default. The same applies to *previous_time_increment*.

The *multiplier_function* parameter may be used to scale the time increment. This scaling is done directly if *cfl_control*=off, but indirectly by scaling the CFL number if *cfl_control*=on. In both cases, the final result is still limited by the other parameters. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file; see the *MULTIPLIER_FUNCTION* command for an example.

TIME_INTEGRATION

Specifies the predictor multi-corrector time integration parameters.

Type

AcuSolve Command

Syntax

TIME_INTEGRATION {parameters}

Qualifier

This command has no qualifier.

Parameters

predictor (enumerated) [=same_v]

Predictor strategy.

same_v	Same velocity predictor.
zero_a	Zero acceleration predictor.

time_integration_order or *order (enumerated) [=first]*

Time-accurate order of the time stepping algorithm.

first	First-order time accurate (Implicit Euler).
second	Second-order time accurate.

high_frequency_damping_factor or *damp_fct (real) >=0 <=1 [=1]*

High frequency damping factor of the second-order time marching algorithm.

pressure_damping_type (enumerated) [=max]

Controls high frequency pressure damping in the pressure (continuity) equation. Used with *time_integration_order*=second. For transient compressible flow simulations, this value should be set to match.

max	Sets *high_frequency_damping_factor=1*.
match	Same damping as in other equations (that is uses the value specified by *high_frequency_damping_factor)*.

lumped_mass_factor or *lump_fct (real) >=0 <=1 [=1]*

Fraction of the mass matrix that consists of the lumped mass matrix. The remaining fraction consists of the consistent mass matrix.

initialize_acceleration or *init_accel (boolean) [=off]*

Flag specifying whether or not to initialize the acceleration before the first time step. Used with *time_integration_order*=second.

`initialize_stokes` or `init_stokes` (boolean) [=off]
: Flag to improve initialization of the Navier-Stokes equations by solving the Stokes equation before proceeding to the first time step.

`initialize_turbulence` or `init_turb` (boolean) [=off]
: Flag to generate initial conditions for the turbulence equations before proceeding to the first time step.

`initialize_levelset` or `init_lvset` (boolean) [=on]
: Flag to initialize the level set field by solving the level set redistancing equation on the user prescribed initial conditions. Used only when performing a multi field simulation that utilizes the level set transport model.

Description

This command specifies the predictor multi-corrector parameters of the time integration algorithm.

AcuSolve uses a semi-discrete formulation for the solution of flow problems. The Galerkin/Least-Squares (GLS) finite element method is used to discretize the differential equations in space and the generalized-alpha method is used to advance the solution in time.

The generalized-alpha method is a linear multi-step algorithm which uses "velocity" and "acceleration" like terms at time steps n and n+1, when solving first-order differential equations. Predictor multi-corrector algorithms are then applied on top of the generalized-alpha method to handle solutions of the nonlinear problem.

Two types of predictors are available. The same_v predictor sets the initial velocity (and equivalently the initial temperature, species, and so on) of time step n+1 to the final values of time step n. The initial acceleration (and equivalently the initial time derivatives of temperature, species, and so on) of time step n+1 is computed such that the generalized- alpha difference equation is satisfied. This predictor is very stable, hence typically used.

The zero_a predictor, on the other hand, sets the initial acceleration of time step n+1 to zero and computes the initial velocity of time step n+1 to satisfy the generalized-alpha difference equation. This predictor may yield faster nonlinear convergence within each time step; however, it is less stable and therefore rarely used.

The generalized-alpha method has a number of parameters that control the stability and accuracy of the time integration. These parameters are internally set given the values of *`time_integration_order`* and *`high_frequency_damping_factor`*.

For steady-state problems, the first-order algorithm is typically used:

```
TIME_INTEGRATION {
   predictor                 = same_v
   time_integration_order    = first
   lumped_mass_factor        = 1
}
```

This algorithm is equivalent to the implicit Euler time marching method. This scheme is first-order accurate and unconditionally stable for linear problems.

For transient problems, the second-order algorithm is typically used. With this algorithm you can control the amount of high frequency damping through *high_frequency_damping_factor*. The highest amount of damping is given by:

```
TIME_INTEGRATION {
   predictor                        = same_v
   time_integration_order           = second
   high_frequency_damping_factor    = 1
   lumped_mass_factor               = 1
}
```

This algorithm is equivalent to Gear's method, which is second-order time-accurate, yet unconditionally stable for linear problems. This algorithm has a high degree of stability and still maintains second-order time accuracy.

Alternatively, the command for the mid-point rule (Trapezoidal for linear problems) may be given by:

```
TIME_INTEGRATION {
   predictor                        = same_v
   time_integration_order           = second
   high_frequency_damping_factor    = 0
   lumped_mass_factor               = 1
}
```

This method is second-order time-accurate, but neutrally stable for linear problems. For nonlinear problems, which are typically solved by AcuSolve, this algorithm may lead to instabilities. To retain nonlinear stability, values greater than 0.2 are recommended for *high_frequency_damping_factor*.

For the special case of non-zero isothermal compressibility (see the DENSITY_MODEL command) combined with second-order time integration, the pressure (continuity) equation often requires more high-frequency damping than the other equations. Specifying max for *pressure_damping_type* sets the *high_frequency_damping_factor* to the highest value allowed (one) for the pressure equation only.

The *lumped_mass_factor* parameter specifies the fraction of the mass matrix that is made up of the lumped mass matrix. The remaining is made up of the consistent mass matrix. This same fraction is also used in the residual of the inertia terms of all equations. Lumped mass matrices are typically more stable and have faster linear and nonlinear convergence behavior. However, consistent mass matrices are typically more accurate in time. If tight time accuracy is desired, a *lumped_mass_factor* of zero or nearly zero may be used.

When solving transient problems, where the accuracy of the initial conditions is crucial, the acceleration (and equivalent fields) must satisfy the generalized-alpha equation at the first time step. If *initialize_acceleration* is turned on, a pseudo time step is solved before the first time step, in order to initialize the acceleration fields. This step is performed only for problems with second-order time integration and only when starting from time zero. Restart problems are assumed to have consistent acceleration.

Initialization of the flow variables for all kinds of Navier-Stokes problems can also be improved by turning on *initialize_stokes*. When this option is turned on, AcuSolve will solve the Stokes equations to generate the initial conditions for the full Navier-Stokes solve. Additionally, the initial conditions for the turbulence equations can be improved by turning the *initialize_turbulence* option on. When this option is activated, algebraic relationships based on simplifications to the turbulence model are used to provide initial conditions for the turbulence model equations. This option is only valid when used with

the Spalart-Allmaras, SST, k-omega, BSL k-omega, k-epsilon, realizable k-epsilon, RNG k-epsilon, DES and SST-DES turbulence models. The most drastic impact of this option is typically seen when using the k-omega and k-epsilon based models for steady state simulations. It is often seen to reduce the number of time steps required to achieve a converged solution. This option also initializes the transition variables in the case of a simulation that includes a transition model.

To improve the behavior of multi field simulations using the level set transport model, the `initialize_levelset` option can be activated. This option solves the level set redistancing equation on the user specified level set field to produce better initialization of the variable.

TIME_SEQUENCE

Specifies the time stepping and staggering strategy.

Type

AcuSolve Command

Syntax

`TIME_SEQUENCE {parameters}`

Qualifier

This command has no qualifier.

Parameters

`min_cases` *>=0 [=1]*
: Minimum number of optimization cases allowed for the run.

`max_cases` *>=0 [=50]*
: Maximum number of optimization cases allowed for the run.

`optimization_convergence_tolerance` *>=0 [=1.e-4]*
: A necessary condition for the termination of an optimization run is that all convergence measures must be less than this tolerance.

`final_time` *(real) >=0 [=0]*
: Final time of the run. The run will terminate when the analysis time reaches the final time. The run may terminate earlier due to other criteria. If zero, this option is ignored.

`min_time_steps` *or* *`min_steps`* *(integer) >=0 [=1]*
: Minimum number of time steps before terminating a run due to convergence to steady state, or ending a case if multiple cases are used in a run.

`max_time_steps` *or* *`max_steps`* *(integer) >=0 [=1]*
: Maximum number of time steps allowed for a run or a case. The run may terminate, or the case may end, earlier due to other criteria. If zero, this option is ignored.

`initial_case_time_steps` *(integer) >=0 [=0]*
: Minimum number of time steps used for the initial case. This option will also override the *`max_time_steps`* parameter for the initial case. If zero, this option is ignored.

`convergence_tolerance` *or* *`conv_tol`* *(real) >=0 [=1.e-4]*
: Steady-state convergence tolerance. The run will terminate if all convergence measures within a time step are less than this tolerance. This criteria requires that at least *`min_time_steps`* time steps are solved and that *`termination_delay`* consecutive time steps have passed while satisfying the convergence tolerance.

`termination_delay` *or* *`exit_delay`* *(integer) >=0 [=0]*
: Minimum number of consecutive time steps in which the steady-state convergence criteria must be satisfied before terminating the run. A value of zero results in the same behavior as one.

lhs_update_initial_times or *lhs_init_steps* *(integer) >=0 [=1]*
: The number of initial time steps in which the left-hand-side (LHS) matrices of all staggers are discarded at the start of every time step.

lhs_update_frequency or *lhs_freq* *(integer) >=0 [=0]*
: The time step frequency at which the left-hand-side (LHS) matrices of all staggers are discarded at the start of such time steps. If zero, this option is ignored.

min_stagger_iterations or *min_stg_iters* *(integer) >=0 [=1]*
: Minimum number of stagger iterations before advancing to the next time step. If zero, this option is ignored.

max_stagger_iterations or *max_stg_iters* *(integer) >=0 [=1]*
: Maximum number of stagger iterations before advancing to the next time step.

stagger_convergence_tolerance or *stg_conv_tol* *(real) >=0 [=1.e-4]*
: Time step convergence tolerance. The stagger iteration is terminated when all convergence measures within the stagger iteration are less than this convergence tolerance and at least *min_stagger_iterations* has been solved.

stagger_lhs_update_frequency or *stg_lhs_freq* *(integer) >=0 [=0]*
: The stagger iteration frequency at which the left-hand-side (LHS) matrices of all staggers are discarded at the start of such staggers. If zero, this option is ignored.

staggers or *stgs* *(list) [no default]*
: List of staggers to be executed. Staggers are solved in the specified sequence.

allow_stagger_reordering or *stg_reorder* *(boolean) [=off]*
: Allow AcuSolve to change the stagger order to optimize solution stability.

Description

This command specifies the time stepping and stagger iteration strategy and parameters.

A time marching method is used to obtain solutions for both steady-state and transient problems. Little distinction is made between the two. For steady state problems, you may consider the time stepping algorithm as a fancy nonlinear iteration loop. A predictor multi-corrector scheme is used within each time step. The predictor is performed at the beginning of each time step and correctors are performed within the staggers. Staggers are used to solve for a subset of equations present in the problem. Within each stagger, the residual and the left-hand-side (LHS) matrix of the specified equation system is formed, the resulting linear equation system is solved and the solution is updated (corrected). These steps define a set of nested loops which are used to advance the solution. A pseudo code of the time stepping strategy is shown below:

```
Loop over time steps
   Predict all active solution fields
   Loop over staggers
      Stagger 1:
         Loop over nonlinear iterations
            Form stagger residual and if needed LHS matrix
            Solve linear equation system
            Update stagger solution field(s)
            Check nonlinear convergence
         End nonlinear loop
      ...
      Stagger N:
```

```
         Loop over nonlinear iterations
            Form stagger residual and if needed LHS matrix
            Solve linear equation system
            Update stagger solution field(s)
            Check nonlinear convergence
         End nonlinear loop
      Check stagger convergence
   End stagger loop
   Check time step convergence
   Optionally compute and output results
   Determine time increment of the next time step
End time step loop
```

The loops over the time steps and staggers and the sequence of staggers are controlled by the TIME_SEQUENCE command. The loop over each stagger's nonlinear iterations, formations, and solution of stagger equations is controlled by the STAGGER command. The selection of the time increments is controlled by the TIME_INCREMENT command. The predictor multi-corrector parameters are controlled by the TIME_INTEGRATION command. Convergence checking parameters that are common to all staggers are set in the CONVERGENCE_CHECK_PARAMETERS command. Linear solver parameters that are common to all staggers are set in the LINEAR_SOLVER_PARAMETERS command.

The loop over the time steps terminates when one of the following occurs:

- Analysis time exceeds the *final_time*.
- *max_time_steps* time steps are solved.
- All convergence measures of the last *termination_delay+1* time steps are below the *convergence_tolerance* and a minimum of *min_time_steps* time steps is solved.
- User signals termination.
- A fatal error occurs.

The loop over the staggers terminates when one of the following occurs:

- *max_stagger_iterations* stagger iterations are performed.
- *min_stagger_iterations* stagger iterations are performed and the last set of convergence measures fall below *stagger_convergence_tolerance*.

The *staggers* parameter defines the list of staggers to be solved. For example, to solve a turbulent thermal flow problem, you may specify:

```
TIME_SEQUENCE {
   staggers                 = { "flow", "turb", "temp" }
   ...
}
STAGGER( "flow" ) {
   equation                 = flow
   ...
}
STAGGER( "temp" ) {
   equation                 = temperature
   ...
}
STAGGER( "turb" ) {
   equation                 = turbulence
   ...
}
```

Here the *"flow"* stagger is solved first for the flow equations, then the *"turb"* stagger is solved for the turbulence equation and then the *"temp"* stagger is solved for the energy equation. A stagger may be repeated multiple times. For example,

```
TIME_SEQUENCE {
    staggers                = { "flow", "turb", "flow","temp" }
    ...
}
```

solves the *"flow"* stagger before and after the *"turb"* stagger.

The LHS matrix of a stagger is computed only if it does not exist. Therefore, to force the formation of a LHS matrix, the solver simply discards the current one. The LHS matrices of the staggers are discarded at the beginning of the initial *lhs_update_initial_times* time steps. In addition, the LHS matrices of the staggers are discarded at the beginning of every *lhs_update_frequency* time steps and *stagger_lhs_update_frequency* stagger iterations. For example, given:

```
TIME_SEQUENCE {
    lhs_update_initial_times          = 10
    lhs_update_frequency              = 6
    stagger_lhs_update_frequency      = 3
    min_stagger_iterations            = 1
    max_stagger_iterations            = 3
}
```

the LHS matrices are discarded at the beginning of the first 10 time steps, plus every 6 time steps starting from time step six. In addition, a minimum of one and a maximum of three stagger iterations are performed per time step. If iteration three is required, then the LHS matrices are discarded prior to this iteration.

Convergence checks are performed on the residual and solution increments. For the residual, the norm of the residual is normalized with respect to the norm of the forces making up the residual. This ratio is the relevant measure, because it measures the ratio of the out-of-balance forces to the value of the forces. This ratio is computed separately for each solution field. For the solution increment, the norm of the solution increment is normalized with respect to the norm of the solution field prior to updating the solution. This ratio is also computed separately for each solution field. These ratios are used to check for convergence. Convergence occurs when all residual and solution increment ratios fall below the specified tolerance. The CONVERGENCE_CHECK_PARAMETERS command may be used to control the relative importance of each ratio.

For a steady-state problem, you typically use the time stepping algorithm as a nonlinear iteration strategy. Hence, a single pass through the staggers is typically used. To accelerate convergence the LHS matrix is reformed at every nonlinear iteration. Below is a typical setup:

```
TIME_SEQUENCE {
    final_time                        = 0
    min_time_steps                    = 0
    max_time_steps                    = 100
    convergence_tolerance             = 1.e-3
    termination_delay                 = 0
    lhs_update_initial_times          = 0
    lhs_update_frequency              = 1
    min_stagger_iterations            = 1
    max_stagger_iterations            = 1
    stagger_convergence_tolerance     = 1.0
```

```
    stagger_lhs_update_frequency       = 0
    staggers                           = { "flow", "turb", "temp", "spec1" }
}
STAGGER( "flow" ) {
    equation                           = flow
    min_stagger_iterations             =1
    max_stagger_iterations             =1
    ...
}
...
```

Transient problems are solved with either fixed or variable number of stagger iterations. The CPU cost of reforming the LHS matrices is typically small. Since frequent reformation of the LHS matrices enhances nonlinear convergence, you usually benefit by doing so. If very small time increments are used, one may wish to reform the LHS matrix less often. A fixed iteration count transient problem may be set up as follows:

```
TIME_SEQUENCE {
    final_time                         = 1800
    min_time_steps                     = 0
    max_time_steps                     = 100
    convergence_tolerance              = 0.0
    termination_delay                  = 0
    lhs_update_initial_times           = 0
    lhs_update_frequency               = 1
    min_stagger_iterations             = 2
    max_stagger_iterations             = 2
    stagger_convergence_tolerance      = 1.0
    stagger_lhs_update_frequency       = 1
    staggers                           = { "flow", "turb", "temp", "spec1" }
}
STAGGER( "flow" ) {
    equation                           = flow
    min_stagger_iterations             = 1
    max_stagger_iterations             = 1
    ...
}
...
```

A transient problem with convergence tolerance control may be set up as follows:

```
TIME_SEQUENCE {
   final_time                          = 1800
   min_time_steps                      = 0
   max_time_steps                      = 100
   convergence_tolerance               = 0.0
   termination_delay                   = 0
   lhs_update_initial_times            = 0
   lhs_update_frequency                = 1
   min_stagger_iterations              = 1
   max_stagger_iterations              = 4
   stagger_convergence_tolerance       = 1.e-3
   stagger_lhs_update_frequency        = 1
   staggers                            = { "flow", "turb", "temp","spec1" }
}
STAGGER( "flow" ) {
   equation                            = flow
   min_stagger_iterations              = 1
   max_stagger_iterations              = 2
   convergence_tolerance               = 1.e-3
```

```
   ...
}
...
```

In a Direct Coupling Fluid Structural Interaction (DC-FSI) problem, a stagger is added for the solution of the solid/structural part by an external code:

```
TIME_SEQUENCE {
   ...
   staggers                              = { "abaqus", "flow", "turb", "temp",
 "spec1" }
}
STAGGER( "abaqus" ) {
   equation                              = external_code
   ...
}
```

See the EXTERNAL_CODE command for details on DC-FSI problems.

In a coupled particle-flow problem, a stagger is added for the solution of the particle equations by AcuTrace:

```
TIME_SEQUENCE {
   ...
   staggers                              = { "flow", "particle" }
}
STAGGER( "particle" ) {
   equation                              = particle
}
```

For some applications, the order in which the staggers are solved can have a significant impact on the stability of the simulation. For example, applications involving fluid-structure interaction benefit from solving the mesh stagger first. However, for applications involving free surfaces, the flow equations should be solved first. The allow_stagger_reordering option lets AcuSolve override the stagger order that you specified to create a stagger order with optimal stability for the given application. This parameter only has an impact on the behavior of the solver when a moving mesh simulation is performed.

Consider a practical fluid-structure interaction simulation where you have given the following stagger order:

```
TIME_SEQUENCE {
   staggers                        = { "flow", "turb", "mesh" }
   allow_stagger_reordering        = on
   ...
}
```

If *allow_stagger_reordering*=on, AcuSolve will first check to see if the mesh equation is active. If it is, it then checks to see if there are any free surface boundary conditions present in the simulation. If free surfaces exist, the stagger sequence is ordered such that the flow stagger is solved first. If no free surfaces are present, then the stagger sequence is ordered such that the mesh stagger is solved first.

Note: The default solution strategy created by the AUTO_SOLUTION_STRATEGY command will set this option to off unless *mesh*=arbitrary_lagrangian_eulerian.

Body Force Commands

10

The Body Force commands are described in this chapter. These commands add volumetric source terms to the governing conservation equations.

This chapter covers the following:

- BODY_FORCE (p. 226)
- GRAVITY (p. 229)
- MASS_HEAT_SOURCE (p. 236)
- MASS_SPECIES_SOURCE (p. 242)
- MOMENTUM_SOURCE (p. 248)
- ROTATION_FORCE (p. 254)
- VOLUME_HEAT_SOURCE (p. 256)
- VOLUME_SPECIES_SOURCE (p. 263)

The individual source terms are defined by the first six commands and subsequently collected by the `BODY_FORCE` command. This latter command may be assigned to any number of element sets.

BODY_FORCE

Specifies a collection of volumetric body force terms.

Type

AcuSolve Command

Syntax

`BODY_FORCE("name") {parameters...}`

Qualifier

User-given name.

Parameters

`gravity` or *`grav`* *(string)[=none]*
: User-given name of the gravity command. If none, no gravity is included.

`rotation_force` or *`rot`* *(string) [=none]*
: User-given name of the rotational force command. If none, no rotational force is included.

`mass_heat_source` *(string) [=none]*
: User-given name of the heat source per unit mass command. If none, no mass heat source is included.

`momentum_source` *(string) [=none]*
: User-given name of the momentum source command. If none, no momentum source is included.

`volume_heat_source` *(string) [=none]*
: User-given name of the heat source per unit volume command. If none, no volume heat source is included.

`mass_species_1_source` *(string) [=none]*
: User-given name of the species source per unit mass command for the species 1 equation. If none, no mass species source for the species 1 equation is included.

`mass_species_2_source` *(string) [=none]*
: User-given name of the species source per unit mass command for the species 2 equation. If none, no mass species source for the species 2 equation is included.

`mass_species_3_source` *(string) [=none]*
: User-given name of the species source per unit mass command for the species 3 equation. If none, no mass species source for the species 3 equation is included.

`mass_species_4_source` *(string) [=none]*
: User-given name of the species source per unit mass command for the species 4 equation. If none, no mass species source for the species 4 equation is included.

`mass_species_5_source` *(string) [=none]*
: User-given name of the species source per unit mass command for the species 5 equation. If none, no mass species source for the species 5 equation is included.

mass_species_6_source (string) [=none]
User-given name of the species source per unit mass command for the species 6 equation. If none, no mass species source for the species 6 equation is included.

mass_species_7_source (string) [=none]
User-given name of the species source per unit mass command for the species 7 equation. If none, no mass species source for the species 7 equation is included.

mass_species_8_source (string) [=none]
User-given name of the species source per unit mass command for the species 8 equation. If none, no mass species source for the species 8 equation is included.

mass_species_9_source (string) [=none]
User-given name of the species source per unit mass command for the species 9 equation. If none, no mass species source for the species 9 equation is included.

volume_species_1_source (string) [=none]
User-given name of the species source per unit volume command for the species 1 equation. If none, no volume species source for the species 1 equation is included.

volume_species_2_source (string) [=none]
User-given name of the species source per unit volume command for the species 2 equation. If none, no volume species source for the species 2 equation is included.

volume_species_3_source (string) [=none]
User-given name of the species source per unit volume command for the species 3 equation. If none, no volume species source for the species 3 equation is included.

volume_species_4_source (string) [=none]
User-given name of the species source per unit volume command for the species 4 equation. If none, no volume species source for the species 4 equation is included.

volume_species_5_source (string) [=none]
User-given name of the species source per unit volume command for the species 5 equation. If none, no volume species source for the species 5 equation is included.

volume_species_6_source (string) [=none]
User-given name of the species source per unit volume command for the species 6 equation. If none, no volume species source for the species 6 equation is included.

volume_species_7_source (string) [=none]
User-given name of the species source per unit volume command for the species 7 equation. If none, no volume species source for the species 7 equation is included.

volume_species_8_sourcenone (string) [=none]
User-given name of the species source per unit volume command for the species 8 equation. If none, no volume species source for the species 8 equation is included.

volume_species_9_source (string) [=none]
User-given name of the species source per unit volume command for the species 9 equation. If none, no volume species source for the species 9 equation is included.

Description

This command specifies a collection of volumetric body forces.

> **Note:** ROTATION_FORCE is an obsolete command and will be removed eventually. Instead, REFERENCE_FRAME should be used for rotational forces.

BODY_FORCE commands collect

- GRAVITY
- ROTATION_FORCE
- MASS_HEAT_SOURCE
- MOMENTUM_SOURCE
- VOLUME_HEAT_SOURCE
- MASS_SPECIES_SOURCE
- VOLUME_SPECIES_SOURCE

commands by referencing their user-given name. The BODY_FORCE commands are in turn referenced by ELEMENT_SET commands. For example,

```
GRAVITY( "my gravity" ) { ... }
ROTATION_FORCE( "my rotation force" ) { ... }
MOMENTUM_SOURCE( "my momentum source" ) { ... }
MASS_HEAT_SOURCE( "my heat source" ) { ... }
MASS_SPECIES_SOURCE( "my species source 1" ) { ... }
MASS_SPECIES_SOURCE( "my species source 2" ) { ... }
BODY_FORCE( "my body force" ) {
 gravity = "my gravity"
 rotation_force = "my rotation source"
 momentum_source = "my momentum source"
 mass_heat_source = "my heat source"
 mass_species_1_source = "my species source 1"
 mass_species_2_source = "my species source 2"
 mass_species_3_source = "my species source 1"
}
```

collects all source terms which in turn may be referenced by one or more element sets:

```
ELEMENT_SET( "fluid with body forces" ) {
 body_force = "my body force"
...
}
```

For backward compatibility with AcuSolve version 1.6 and earlier, *heat_source* may be used instead of *mass_heat_source* and *species_1_source* may be used instead of *mass_species_1_source*, and so on. However, these aliases are obsolete and will be deprecated in the future.

Both *mass_heat_source* and *volume_heat_source* may be specified in a single BODY_FORCE command. Similarly, both mass and volume versions of the species source terms may be specified simultaneously. The appropriate terms are simply added together. However, this usage is not recommended.

GRAVITY

Specifies the gravitational body force.

Type

AcuSolve Command

Syntax

`GRAVITY("name") {parameters...}`

Qualifier

User-given name.

Parameters

`type` *(enumerated) [=none]*

Type of the gravitational body force.

none

No gravity.

constant or const

Constant gravity. Requires *`gravity`*.

piecewise_linear or linear

Piecewise linear curve fit. Requires *`curve_fit_values`* and *`curve_fit_variable`*.

cubic_spline or spline

Cubic spline curve fit. Requires *`curve_fit_values`* and *`curve_fit_variable`*.

user_function or user

User-defined function. Requires *`user_function`*, *`user_values`* and *`user_strings`*.

`gravity` *(array) [={0,0,0}]*

The constant value of the gravity force. This array must have exactly three components in the global xyz coordinate system. Used with constant type.

> **Note:** *`gravity`* and *`gravitational_acceleration`* under `EQUATION` are cumulative.

`curve_fit_values` or *`curve_values`* *(array) [={0,0,0,0}]*

A four-column array of independent-variable/gravity data values. Used with piecewise_linear and cubic_spline types.

`curve_fit_variable` or *`curve_var`* *(enumerated) [no default]*

Independent variable of the curve fit. Used with piecewise_linear and cubic_spline types.

time	Current run time.
x_coordinate or xcrd	X-component of coordinates.

y_coordinate or ycrd	Y-component of coordinates.
z_coordinate or zcrd	Z-component of coordinates.
x_reference_coordinate or xrefcrd	X-component of reference coordinates.
y_reference_coordinate or yrefcrd	Y-component of reference coordinates.
z_reference_coordinate or zrefcrd	Z-component of reference coordinates.
x_velocity or xvel	X-component of velocity.
y_velocity or yvel	Y-component of velocity.
z_velocity or zvel	Z-component of velocity.
velocity_magnitude or vel_mag	Velocity magnitude.
pressure or pres	Pressure
temperature or temp	Temperature
eddy_viscosity or eddy	Turbulence eddy viscosity.
kinetic_energy or tke	Turbulence kinetic energy.
velocity_scale or tvel	Transition velocity scale.
dissipation_rate or teps	Turbulence dissipation rate.
eddy_frequency or tomega	Turbulence frequency.
intermittency or tintc	Transition intermittency.
transition_re_theta or treth	Transition Re-Theta.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.
species_7 or spec7	Species 7.

species_8 or spec8	Species 8.
species_9 or spec9	Species 9.
mesh_x_displacement or mesh_xdisp	X-component of mesh displacement.
mesh_y_displacement or mesh_ydisp	Y-component of mesh displacement.
mesh_z_displacement or mesh_zdisp	Z-component of mesh displacement.
mesh_displacement_magnitude or mesh_disp_mag	Mesh displacement magnitude.
mesh_x_velocity or mesh_xvel	X-component of mesh velocity.
mesh_y_velocity or mesh_yvel	Y-component of mesh velocity.
mesh_z_velocity or mesh_zvel	Z-component of mesh velocity.
mesh_velocity_magnitude or mesh_vel_mag	Mesh velocity magnitude.

`user_function` or `user` *(string) [no default]*
Name of the user-defined function. Used with user_function type.

`user_values` *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function type.

`user_strings` *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with user_function type.

`multiplier_function` *(string) [=none]*
User-given name of the multiplier function for scaling the gravity. If none, no scaling is performed.

Description

This command specifies the gravitational body force or like terms applied to the momentum equation:

$$\rho\frac{\partial \boldsymbol{u}}{\partial t} + \rho\boldsymbol{u} \bullet \nabla\boldsymbol{u} + \nabla p = \nabla \bullet \tau + \rho\boldsymbol{g} \qquad (49)$$

where ρ is the density, **u** is the velocity vector, p is the pressure, $\tau = [\tau_{ij}]$ is the viscous stress tensor and **g** is the gravity vector defined here. `GRAVITY` commands are referenced by `BODY_FORCE` commands, which in turn are referenced by `ELEMENT_SET` commands:

```
GRAVITY( "my gravity" ) {
    type                                = constant
    gravity                             = { 0, -9.81, 0 }
}
BODY_FORCE( "my body force" ) {
    gravity                             = "my gravity"
    ...
}
ELEMENT_SET( "fluid with gravity" ) {
    body_force                          = "my body force"
    ...
}
```

This example defines a constant gravitational acceleration in the y direction.

A constant gravity applies a spatially uniform gravity vector to an element set, as in the above example.

Gravity models of types piecewise_linear and cubic_spline may be used to define gravity as a function of a single independent variable. For example,

```
GRAVITY( "curve fit gravity" ) {
    type                                = piecewise_linear
    curve_fit_values                    = { 0., 0., 0., 0. ;
                                            10., 0., 0., -1. ;
                                            20., 0., 0., 0. ; }
    curve_fit_variable                  = x_coordinate
}
```

defines a gravity force in the z direction as a function of the x coordinate. The *curve_fit_values* parameter is a four-column array corresponding to the independent variable and the x, y, and z components of gravity in the global xyz coordinate system. The independent variable values must be given in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, say `gravity.fit`:

```
0.       0.        0.        0.
10.       0.        0.       -1.
20.       0.        0.        0.
```

and read by:

```
GRAVITY( "curve fit gravity" ) {
    type                                = piecewise_linear
    curve_fit_values                    = Read( "gravity.fit" )
    curve_fit_variable                  = x coordinate
}
```

A gravity of type user_function may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider the case where gravity is a linear function of species 1 and species 2. Then the input command may be given by:

```
GRAVITY( "UDF gravity" ) {
    type                                = user_function
    user_function                       = "usrGravityExample"
    user_values                         = { 1., 1.5 } # proportionality constants
}
```

where the user-defined function "usrGravityExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"

UDF_PROTOTYPE( usrGravityExample ) ;                                    /* function
 prototype                */

Void usrGravityExample (
   UdfHd                    udfHd,                                      /* Opaque handle
 for accessing data */
   Real*                    outVec,                                     /* Output vector
                 */
   Integer                  nItems,                                     /* Number of
 elements                 */
   Integer                  vecDim                                      /* = 3 (for three
 components)       */
) {
   Integer                  elem ;                                      /* an element
 counter                 */
   Real                     coef1 ;                                     /* scaling factor 1
                 */
   Real                     coef2 ;                                     /* scaling factor 2
                 */
   Real*                    spec ;                                      /* species vector
                 */
   Real*                    spec1 ;                                     /* species 1 vector
                 */
   Real*                    spec2 ;                                     /* species 2 vector
                 */
   Real*                    usrVals ;                                   /* user values
                 */
   Real*                    xGrav ;                                     /* x-component of
 gravity           */
   Real*                    yGrav ;                                     /* y-component of
 gravity           */
   Real*                    zGrav ;                                     /* z-component of
 gravity           */

   udfCheckNumUsrVals( udfHd, 2 ) ;                                     /* check for error
                 */
   usrVals                  = udfGetUsrVals( udfHd ) ;                  /* get the user
 vals                */
   coef1                    = usrVals[0] ;                              /* get coef. 1
                 */
   coef2                    = usrVals[1] ;                              /* get coef. 2
                 */
   spec                     = udfGetElmData( udfHd, UDF_ELM_SPECIES ) /* get the user
 vals                */
   spec1                    = &spec[0*nItems] ;                         /* localize
 species1                 */
```

```
   spec2                 = &spec[1*nItems] ;                        /* localize
 species2                 */
   xGrav                 = &outVec[0*nItems] ;                      /* localized xGrav
                */
   yGrav                 = &outVec[1*nItems] ;                      /* localized yGrav
                */
   zGrav                 = &outVec[2*nItems] ;                      /* localized zGrav
                */
   for ( elem = 0 ; elem < nItems ; elem++ ) {
         xGrav[elem] = coef1 * spec1[elem] + coef2 * spec2[elem] ;
         yGrav[elem] = 0 ;
         zGrav[elem] = 0 ;
   }
} /* end of usrGravityExample() */
```

The dimension of the returned gravity vector, outVec, is the number of elements times three.

The *multiplier_function* parameter may be used to uniformly scale all gravity values. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file; see the MULTIPLIER_FUNCTION command for an example.

Note: For thermal problems with the Boussinesq approximation the gravity is scaled by the product of the expansivity and the temperature minus reference temperature; see the DENSITY_MODEL command for more details.

A gravity of type user_function can be used to import an acceleration profile as a body force by stacking accelerations in different directions. Below is an example of a UDF that loads and interpolates user data once per time step, formatted as time, x-acceleration, y-acceleration, and z-acceleration. In this UDF, the udfFirstCall is used to load the data table at the start of the simulation. The interpolation results are stored in local static variables for reuse. The gravity field is also computed using a UDF, which simply applies the local x, y, z values to the input vector. For a more convenient solution to this, the piecewise_linear in the GRAVITY can be used with a curve fit variable of time.

```
#include "acusim.h"
#include "udf.h"
#include <stdio.h>
#include <string.h>

float* data = 0;
Integer nPts =0;
Integer curr =0;

float gravx = 0;
float gravy = 0;
float gravz = 0;

UDF_PROTOTYPE( Setup ) ; /* function prototype*/
Void Setup (
      UdfHd           udfHd,                      /* Opaque handle for accessing
 data  */
      Real*           outVec,                     /* Output vector
   */
      Integer         nItems,                     /* Number of BC nodes
   */
      Integer         vecDim                      /* = 1
   */
      ) {
  Real time;
```

```
  int i;
  float alpha;

  if( udfFirstCall( udfHd ) ) {
    FILE* file;
    float t,x,y,z;

    udfPrintMessPrim( udfHd, " ");
    udfPrintMessPrim( udfHd, "   --- UDF reading acceleration file:");
    udfPrintMessPrim( udfHd, " ");
    file = fopen("Acc_data.txt","r");
    nPts = 0;
    while(fscanf(file,"%e %e %e %e",&t,&x,&y,&z) != EOF)
      nPts++;
    rewind(file);
    udfPrintMessPrim( udfHd, "  --- Reading %d points",nPts);
    data = (float*) (udfGetGlobalVector(udfHd,"data",nPts*4*sizeof(float)));
    curr = 0;
    for (i=0 ; i<nPts ;i++) {
      fscanf(file,"%e %e %e %e",data+4*i,data+4*i+1,data+4*i+2,data+4*i+3);
    }
    fclose(file);
  }

  time = udfGetTime( udfHd );
  if (data[curr*4] > time )
    return;
  while ( data [curr*4+4]< time && curr+1<nPts)
    curr++;

  if ( curr == nPts-1) {
    gravx = data[curr*4+1];
    gravy = data[curr*4+2];
    gravz = data[curr*4+3];
  } else {
    alpha = (time-data[curr*4])/(data[curr*4+4]-data[curr*4]);
    gravx = data[curr*4+1]*(1-alpha)+data[curr*4+5]*alpha;
    gravy = data[curr*4+2]*(1-alpha)+data[curr*4+6]*alpha;
    gravz = data[curr*4+3]*(1-alpha)+data[curr*4+7]*alpha;
  }

  udfPrintMessPrim( udfHd, " --- UDF time: %e gravity: %e %e %e
 ---",time,gravx,gravy,gravz);
}

UDF_PROTOTYPE(usrGravity) ; /* Compute Particle Velocity  */

Void usrGravity (
        UdfHd    udfHd,          /* Opaque handle      */
        Real*    outVec,         /* Output vector      */
        Integer nItems,          /* Number of nodes    */
        Integer vecDim           /* = 3                */
)   {
  Integer node;            /* node counter        */
  Integer     i ;          /* a running index     */

  for ( i = 0 ; i < nItems ; i++ ) {
    outVec[i+0*nItems] = gravx;
    outVec[i+1*nItems] = gravy;
    outVec[i+2*nItems] = gravz;
  }
} /* end of usrGravity () */
```

MASS_HEAT_SOURCE

Specifies heat source per unit mass for the energy equation.

Type

AcuSolve Command

Syntax

```
MASS_HEAT_SOURCE("name") {parameters...}
```

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of heat source per unit mass.

none	No mass heat source.
constant or const	Constant mass heat source. Requires *mass_heat_source*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
user_function or user	User-defined function. Requires *user_function*, *user_values* and *user_strings*.

mass_heat_source (real) [=0]

The constant value of the heat source per unit mass. Used with constant type.

curve_fit_values or curve_values (array) [={0,0}]

A two-column array of independent-variable/heat-source data values. Used with piecewise_linear and cubic_spline types.

curve_fit_variable or curve_var (enumerated) [=temperature]

Independent variable of the curve fit. Used with piecewise_linear and cubic_spline types.

x_coordinate or xcrd	X-component of coordinates.
y_coordinate or ycrd	Y-component of coordinates.
z_coordinate or zcrd	Z-component of coordinates.
x_reference_coordinate or xrefcrd	X-component of reference coordinates.

y_reference_coordinate or yrefcrd	Y-component of reference coordinates.
z_reference_coordinate or zrefcrd	Z-component of reference coordinates.
x_velocity or xvel	X-component of velocity.
y_velocity or yvel	Y-component of velocity.
z_velocity or zvel	Z-component of velocity.
velocity_magnitude or vel_mag	Velocity magnitude.
pressure or pres	Pressure.
temperature or temp	Temperature.
eddy_viscosity or eddy	Turbulence eddy viscosity.
kinetic_energy or tke	Turbulence kinetic energy.
velocity_scale or tvel	Transition velocity scale.
dissipation_rate or teps	Turbulence dissipation rate.
eddy_frequency or tomeg	Turbulence frequency.
intermittency or tintc	Transition intermittency.
transition_re_theta or treth	Transition Re-Theta.
eddy_frequency or tomega	Turbulence frequency.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.
species_7 or spec7	Species 7.
species_8 or spec8	Species 8.
species_9 or spec9	Species 9.

mesh_x_displacement or mesh_xdisp	X-component of mesh displacement.
mesh_y_displacement or mesh_ydisp	Y-component of mesh displacement.
mesh_z_displacement or mesh_zdisp	Z-component of mesh displacement.
mesh_displacement_magnitude or mesh_disp_mag	Mesh displacement magnitude.
mesh_x_velocity or mesh_xvel	X-component of mesh velocity.
mesh_y_velocity or mesh_yvel	Y-component of mesh velocity.
mesh_z_velocity or mesh_zvel	Z-component of mesh velocity.
mesh_velocity_magnitude or mesh_vel_mag	Mesh velocity magnitude.

user_function or *user* *(string) [no default]*
Name of the user-defined function. Used with user_function type.

user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function type.

user_strings *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with user_function type.

multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the mass heat source. If none, no scaling is performed.

Description

This command specifies the heat source per unit mass applied to the energy equation:

$$\rho\frac{\partial h}{\partial t} + \rho \boldsymbol{u} \bullet \nabla h = \nabla \bullet \boldsymbol{q} + \rho s \qquad (50)$$

where ρ is the density, h is the enthalpy, **u** is the velocity vector, **q** is the diffusive heat flux vector and s is the heat source per unit mass defined here. This command is equivalent to the HEAT_SOURCE command of AcuSolve version 1.6 and earlier, and the parameter *mass_heat_source* corresponds to the earlier *heat_source*. Both of these aliases may still be used, but they will be obsoleted eventually. MASS_HEAT_SOURCE commands are referenced by BODY_FORCE commands, which in turn are referenced by ELEMENT_SET commands:

```
MASS_HEAT_SOURCE( "my heat source" ) {
```

```
   type                                = constant
   mass_heat_source                    = 10
}
BODY_FORCE( "my body force" ) {
   mass_heat_source                    = "my heat source"
   ...
}
ELEMENT_SET( "fluid with heat source" ) {
   body_force                          = "my body force"
   ...
}
```

This example defines a constant heat source per unit mass. Positive values add heat to the system, while negative values remove heat from the system.

A constant mass heat source applies a spatially uniform volumetric heat source per unit mass, as in the above example.

Mass heat source models of types piecewise_linear and cubic_spline may be used to define the heat source as a function of a single independent variable. For example,

```
MASS_HEAT_SOURCE( "curve fit heat source" ) {
    type                               = piecewise_linear
    curve_fit_values                   = { 0., 0. ;
                                           1., 1. ;
                                           2., 4. ; }
    curve_fit_variable                 = species_1
}
```

defines a mass heat source as a function of the first species. In this case, the problem must contain species transport equations; see the `EQUATION` command. The *`curve_fit_values`* parameter is a two-column array corresponding to the independent variable and the heat source values. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *`curve_fit_variable`* falls outside of the curve fit limits.

The *`curve_fit_values`* data may be read from a file. For the above example, the curve fit values may be placed in a file, say `heat_source.fit`:

```
0.      0.
1.      1.
2.      3.
```

and read by:

```
MASS_HEAT_SOURCE( "curve fit heat source" ) {
   type                                = piecewise_linear
   curve_fit_values                    = Read( "heat_source.fit" )
   curve_fit_variable                  = species_1
}
```

A mass heat source of type user_function may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider the case where heat source is proportional to species one, but is applied slowly as a function of time. Then the input command may be given by:

```
MASS_HEAT_SOURCE( "UDF heat source" ) {
```

```
   type                                = user_function
   user_function                       = "usrHeatSourceExample"
   user_function                       = "usrHeatSourceExample"
   user_values                         = { 2.5, # species scaling
                                           10. } # time to apply in full
}
```

where the user-defined function "usrHeatSourceExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrHeatSourceExample ) ;                                    /*
 function prototype */
Void usrHeatSourceExample (
    UdfHd                    udfHd,                                        /*
 Opaque handle for accessing data */
    Real*                    outVec,                                       /*
 Output vector                    */
    Integer                  nItems,                                       /*
 Number of elements               */
    Integer                  vecDim                                        /* = 1
                            */
) {
    Integer                  elem ;                                        /* an
 element counter                  */
    Real                     fTime ;                                       /* final
 time                            */
    Real                     scale ;                                       /*
 scaling factor                   */
    Real                     time ;                                        /* run
 time                             */
    Real*                    spec ;                                        /*
 species vector                   */
    Real*                    usrVals ;                                     /* user
 values                           */
udfCheckNumUsrVals( udfHd, 2 ) ;                                           /* check
 for error                      */
    usrVals                  = udfGetUsrVals( udfHd ) ;                    /* get
 the user vals                    */
    scale                    = usrVals[0] ;                                /*get
 the factor                       */
    fTime                    = usrVals[1] ;                                /* get
 the final time                   */
    time                     = udfGetTime( udfHd ) ;                       /* get
 the run time                     */
    spec                      = udfGetElmData( udfHd, UDF_ELM_SPECIES ) ;  /* get
 the species                      */
    if ( time < fTime ) {
         scale = scale * time / fTime ;                                    /*
 reduce scaling                      */
    }
    for ( elem = 0 ; elem < nItems ; elem++ ) {
          outVec[elem] = scale * spec[elem] ;
    }
} /* end of usrHeatSourceExample() */
```

The dimension of the returned mass heat source vector, *outVec*, is the number of elements.

The *multiplier_function* parameter may be used to uniformly scale the mass heat source values. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input

file. For example, assuming that species one remains between zero and 10, the above heat source using the user-defined function may be obtained by:

```
MASS_HEAT_SOURCE( "equivalent to UDF heat source" ) {
    type                                = piecewise_linear
    curve_fit_values                    = { 0, 0 ;
                                            10, 25 ; }
    curve_fit_variable                  = species_1
    multiplier_function                 = "ramped"
}
MULTIPLIER_FUNCTION( "ramped" ) {
    type                                = piecewise_linear
    curve_fit_values                    = { 0, 0 ; 10, 1 }
    curve_fit_variable                  = time
}
```

MASS_SPECIES_SOURCE

Specifies species source per unit mass for a species transport equation.

Type

AcuSolve Command

Syntax

MASS_SPECIES_SOURCE("name") {parameters...}

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of mass species source.

none	No mass species source.
constant or const	Constant mass species source. Requires *mass_species_source*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
user_function or user	User-defined function. Requires *user_function*, *user_values* and *user_strings*.

mass_species_source (real) [=0]

The constant value of the mass-species source per unit mass. Used with constant type.

curve_fit_values or *curve_values (array) [={0,0}]*

A two-column array of independent-variable/species-source data values. Used with piecewise_linear and cubic_spline types.

curve_fit_variable or *curve_var (enumerated) [=temperature]*

Independent variable of the curve fit. Used with piecewise_linear and cubic_spline types.

x_coordinate or xcrd	X-component of coordinates.
y_coordinate or ycrd	Y-component of coordinates.
z_coordinate or zcrd	Z-component of coordinates.
x_reference_coordinate or xrefcrd	X-component of reference coordinates.

y_reference_coordinate or yrefcrd	Y-component of reference coordinates.
z_reference_coordinate or zrefcrd	Z-component of reference coordinates.
x_velocity or xvel	X-component of velocity.
y_velocity or yvel	Y-component of velocity.
z_velocity or zvel	Z-component of velocity.
velocity_magnitude or vel_mag	Velocity magnitude.
pressure or pres	Pressure.
temperature or temp	Temperature.
eddy_viscosity or eddy	Turbulence eddy viscosity.
kinetic_energy or tke	Turbulence kinetic energy.
velocity_scale or tvel	Transition velocity scale.
dissipation_rate or teps	Turbulence dissipation rate.
eddy_frequency or tomeg	Turbulence frequency.
intermittency or tintc	Transition intermittency.
transition_re_theta or treth	Transition Re-Theta.
eddy_frequency or tomega	Turbulence frequency.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.
species_7 or spec7	Species 7.
species_8 or spec8	Species 8.
species_9 or spec9	Species 9.

mesh_x_displacement or mesh_xdisp	X-component of mesh displacement.
mesh_y_displacement or mesh_ydisp	Y-component of mesh displacement.
mesh_z_displacement or mesh_zdisp	Z-component of mesh displacement.
mesh_displacement_magnitude or mesh_disp_mag	Mesh displacement magnitude.
mesh_x_velocity or mesh_xvel	X-component of mesh velocity.
mesh_y_velocity or mesh_yvel	Y-component of mesh velocity.
mesh_z_velocity or mesh_zvel	Z-component of mesh velocity.
mesh_velocity_magnitude or mesh_vel_mag	Mesh velocity magnitude.

user_function or *user* *(string) [no default]*
Name of the user-defined function. Used with user_function type.

user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function type.

user_strings *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with user_function type.

multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the mass species source. If none, no scaling is performed.

Description

This command specifies the volumetric source per unit mass term applied to the transport equation for a species φ:

$$\rho\frac{\partial\varphi}{\partial t}+\rho\boldsymbol{u}\bullet\nabla\varphi\;=\;\nabla\bullet\Psi+\rho\sigma \qquad (51)$$

where ρ is the density, **u** is the velocity vector, Ψ is the diffusive flux and σ is the species source per unit mass term defined here. This command is equivalent to the SPECIES_SOURCE command of AcuSolve version 1.6 and earlier, and the parameter *mass_species_source* corresponds to the earlier *species_source*. Both of these aliases may still be used, but they will be obsoleted eventually.

MASS_SPECIES_SOURCE commands are referenced by BODY_FORCE commands, which in turn are referenced by ELEMENT_SET commands:

```
MASS_SPECIES_SOURCE( "my species source" ) {
   type                                = constant
   mass_species_source                 = 10
}
BODY_FORCE( "my body force" ) {
   mass_species_1_source               = "my species source"
   ...
}
ELEMENT_SET( "fluid with species source" ) {
   body_force                          = "my body force"
   ...
}
```

This example defines a constant volumetric species source per unit mass. Positive values add species to the system, while negative values remove species from the system.

A constant mass species source applies a spatially uniform volumetric species source per unit mass, as in the above example.

Mass species source models of types piecewise_linear and cubic_spline may be used to define the species source as a function of a single independent variable. For example,

```
MASS_SPECIES_SOURCE( "curve fit species source" ) {
   type                                = piecewise_linear
   curve_fit_values                    = { 0., 0. ;
                                           1., 1. ;
                                           2., 4. ; }
   curve_fit_variable                  = species_1
}
```

defines a species source as a function of the first species. The *curve_fit_values* parameter is a two-column array corresponding to the independent variable and the species source values. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, say species_source.fit:

```
0.      0.
1.      1.
2.      3.
```

and read by:

```
MASS_SPECIES_SOURCE( "curve fit species source" ) {
   type                                = piecewise_linear
   curve_fit_values                    = Read( "species_source.fit" )
   curve_fit_variable                  = species_1
}
```

A mass species source of type user_function may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider the case where species source is a second order polynomial of species 1. Then the input command may be given by:

```
MASS_SPECIES_SOURCE( "UDF species source" ) {
   type                                = user_function
   user_function                       = "usrSpeciesSourceExample"
   user_values                         = { 1.0,  # constant coef.
                                           0.5,  # linear coef.
                                           0.1 } # quadratic coef

 }
```

where the user-defined function "usrSpeciesSourceExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrSpeciesSourceExample ) ;                              /* function
 prototype                */
Void usrSpeciesSourceExample (
    UdfHd                   udfHd,                                      /* Opaque handle
 for accessing data */
    Real*                   outVec,                                     /* Output vector
                  */
    Integer                 nItems,                                     /* Number of
 elements                */
    Integer                 vecDim                                      /* = 1
                  */
) {
    Integer                 elem ;                                      /* an element
 counter                */
    Real                    coef0 ;                                     /* constant
 coefficient              */
    Real                    coef1 ;                                     /* linear
 coefficient               */
    Real                    coef2 ;                                     /* quadratic
 coefficient             */
    Real*                   spec ;                                      /* species
 vector                    */
    Real*                   usrVals ;                                   /* user values
                  */
udfCheckNumUsrVals( udfHd, 3 ) ;                                        /* check for
 error                    */
    usrVals                 = udfGetUsrVals( udfHd ) ;                  /* get the user
 vals                 */
    coef0                   = usrVals[0] ;                              /* get coef
                  */
    coef1                   = usrVals[1] ;                              /* get coef 2
                  */
    coef2                   = usrVals[2] ;                              /* get coef 3
                  */
    spec                    = udfGetElmData( udfHd,UDF_ELM_SPECIES ) ; /* get the
 species                   */
    for ( elem = 0 ; elem < nItems ; elem++ ) {
        outVec[elem] = coef0
                     + coef1 * spec[elem]
                     + coef2 * spec[elem] * spec[elem] ;
    }
} /* end of usrSpeciesSourceExample() */
```

The dimension of the returned mass species source vector, *outVec*, is the number of elements.

The *multiplier_function* parameter may be used to uniformly scale the mass species source values. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file. For example, to slowly increase a source term from zero to the full amount by time step 10, you can specify:

```
MASS_SPECIES_SOURCE( "ramped species source" ) {
    type                                = constant
    mass_species_source                 = 10
    multiplier_function                 = "ramped"
}
MULTIPLIER_FUNCTION( "ramped" ) {
    type                                = piecewise_linear
    curve_fit_values                    = { 0, 0 ; 10, 1 }
    curve_fit_variable                  = time_step
}
```

MOMENTUM_SOURCE

Specifies the momentum source.

Type

AcuSolve Command

Syntax

MOMENTUM_SOURCE ("name") {parameters...}

Qualifier

User-given name.

Parameters

type (enumerated) [=constant]

Type of the *momentum_source*.

none	No *momentum_source*.
constant or const	Constant *momentum_source*. Requires momentum_source.
piecewise_linear or linear	Piecewise linear curve fit. Requires curve_fit_values and curve_fit_variable.
cubic_spline or spline	Cubic spline curve fit. Requires curve_fit_values and curve_fit_variable.
user_function or user	User-defined function. Requires user_function, user_values and user_strings.

momentum_source (array) [={0,0,0}]

The constant value of the momentum source. This array must have exactly three components in the global xyz coordinate system. Used with constant type.

> **Note:** momentum_source and gravitational_acceleration under EQUATION are cumulative.

curve_fit_values or curve_values (array) [={0,0,0,0}]

A four-column array of independent-variable/momentum source data values. Used with piecewise_linear and cubic_spline types.

curve_fit_variable or curve_var (enumerated) [no default]

Independent variable of the curve fit. Used with piecewise_linear and cubic_spline types.

x_coordinate or xcrd	X-component of coordinates.
y_coordinate or ycrd	Y-component of coordinates.

z_coordinate or zcrd	Z-component of coordinates.
x_reference_coordinate or xrefcrd	X-component of reference coordinates.
y_reference_coordinate or yrefcrd	Y-component of reference coordinates.
z_reference_coordinate or zrefcrd	Z-component of reference coordinates.
x_velocity or xvel	X-component of velocity.
y_velocity or yvel	Y-component of velocity.
z_velocity or zvel	Z-component of velocity.
velocity_magnitude or vel_mag	Velocity magnitude.
pressure or pres	Pressure.
temperature or temp	Temperature.
eddy_viscosity or eddy	Turbulence eddy viscosity.
kinetic_energy or tke	Turbulence kinetic energy.
velocity_scale or tvel	Transition velocity scale.
dissipation_rate or teps	Turbulence dissipation rate.
eddy_frequency or tomega	Turbulence frequency.
eddy_time or ttau	Turbulence eddy time.
intermittency or tintc	Transition intermittency.
transition_re_theta or treth	Transition Re-Theta.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.
species_7 or spec7	Species 7.

species_8 or spec8	Species 8.
species_9 or spec9	Species 9.
mesh_x_displacement or mesh_xdisp	X-component of mesh displacement.
mesh_y_displacement or mesh_ydisp	Y-component of mesh displacement.
mesh_z_displacement or mesh_zdisp	Z-component of mesh displacement.
mesh_displacement_magnitude or mesh_disp_mag	Mesh displacement magnitude.
mesh_x_velocity or mesh_xvel	X-component of mesh velocity
mesh_y_velocity or mesh_yvel	Y-component of mesh velocity.
mesh_z_velocity or mesh_zvel	Z-component of mesh velocity.
mesh_velocity_magnitude or mesh_vel_mag	Mesh velocity magnitude.

`user_function` or `user` *(string) [no default]*

Name of the user-defined function. Used with user_function type.

`user_values` *(array) [={}]*

Array of values to be passed to the user-defined function. Used with user_function type.

`user_strings` *(list) [={}]*

Array of strings to be passed to the user-defined function. Used with user_function type.

`multiplier_function` *(string) [=none]*

User-given name of the multiplier function for scaling the momentum source values. If none, no scaling is performed.

Description

This command specifies the momentum source or like terms applied to the momentum equation:

$$\rho\frac{\partial \boldsymbol{u}}{\partial t} + \rho\boldsymbol{u} \bullet \nabla\boldsymbol{u} + \nabla p = \nabla \bullet \tau + \rho\boldsymbol{g} \qquad (52)$$

where ρ is the density, **u** is the velocity vector, p is the pressure, $\tau = [\tau_{ij}]$ is the viscous stress tensor and **g** is the momentum source vector defined here. `MOMENTUM_SOURCE` commands are referenced by `BODY_FORCE` commands, which in turn are referenced by `ELEMENT_SET` commands:

```
MOMENTUM_SOURCE( "my gravity" ) {
type = constant
momentum_source = { 0, -9.81, 0 }
}
BODY_FORCE( "my body force" ) {
momentum_source = "my gravity"
...
}
ELEMENT_SET( "fluid with gravity" ) {
body_force = "my body force"
...
}
```

This example defines a constant momentum source (acceleration) in the y direction.

A constant *momentum_source* applies a spatially uniform gravity vector to an element set, as shown in the example above.

Gravity models of types piecewise_linear and cubic_spline may be used to define momentum source (acceleration) as a function of a single independent variable. For example,

```
GRAVITY( "curve fit momentum source" ) {
type = piecewise_linear
curve_fit_values = { 0., 0., 0., 0. ;
10., 0., 0., -1. ;
20., 0., 0., 0. ; }
curve_fit_variable = x_coordinate
}
```

defines a body force in the z direction as a function of the x coordinate. The *curve_fit_values* parameter is a four-column array corresponding to the independent variable and the x, y and z components of body force in the global xyz coordinate system. The independent variable values must be given in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, say `momentum_source.fit`:

```
0. 0. 0. 0.
  10. 0. 0. -1.
  20. 0. 0. 0.
```

and read by:

```
MOMENTUM_SOURCE( "curve fit gravity" ) {
type = piecewise_linear
curve_fit_values =Read( "momentum_source.fit" )
curve_fit_variable = x_coordinate
}
```

A body force of type user_function may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider the case where body force is a linear function of species 1 and species 2. Then the input command may be given by:

```
GRAVITY( "UDF body force" ) {
type = user_function
user_function = "usrBodyForceExample"
user_values = { 1., 1.5 } # proportionality constants
}
```

where the user-defined function "usrBodyForceExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrBodyForceExample ) ; /* function prototype */
Void usrBodyForceExample (
UdfHd udfHd, /* Opaque handle for accessing data */
Real*     outVec, /* Output vector */
Integer   nItems, /* Number of elements */
Integer vecDim /* = 3 (for three components) */
) {
Integer  elem ; /* an element counter */
Real  coef1 ; /* scaling factor 1 */
Real  coef2 ; /* scaling factor 2 */
Real*  spec ; /* species vector */
Real* spec1 ; /* species 1 vector */
Real* spec2 ; /* species 2 vector */
Real* usrVals ; /* user values */
Real* xBodyF ; /* x-component of body force */
Real* yBodyF ; /* y-component of body force */
Real* zBodyF ; /* z-component of body force */
udfCheckNumUsrVals( udfHd, 2 ) ; /* check for error */
usrVals  = udfGetUsrVals( udfHd ) ; /* get the user vals */
coef1  = usrVals[0] ; /* get coef. 1 */
coef2  = usrVals[1] ; /* get coef. 2 */
        spec          = udfGetElmData( udfHd, UDF_ELM_SPECIES ) ;
  /* get the user vals */
spec1 = &spec[0*nItems] ; /* localize species1 */
spec2 = &spec[1*nItems] ; /* localize species2 */
xBodyF = &outVec[0*nItems] ; /* localized xBodyF */
yBodyF = &outVec[1*nItems] ; /* localized yBodyF */
zBodyF = &outVec[2*nItems] ; /* localized zBodyF */
for ( elem = 0 ; elem < nItems ; elem++ ) {
xBodyF[elem] = coef1 * spec1[elem] + coef2 * spec2[elem] ;
yBodyF[elem] = 0 ;
zBodyF[elem] = 0 ;
}
} /* end of usrBodyForceExample() */
```

The dimension of the returned momentum source vector, *outVec*, is the number of elements times three.

The *multiplier_function* parameter may be used to uniformly scale all momentum source values. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file; see the MULTIPLIER_FUNCTION command for an example.

Note: For thermal problems with the Boussinesq approximation the momentum source is scaled by the product of the expansivity and the temperature minus reference temperature; see the `DENSITY_MODEL` command for more details.

ROTATION_FORCE

Specifies rotational force resulting from use of rotating frame of reference (obsolete command).

Type

AcuSolve Command

Syntax

`ROTATION_FORCE("name") {parameters...}`

Qualifier

User-given name.

Parameters

`centrifugal` *(boolean) [=off]*
Flag specifying whether to apply centrifugal force.

`coriolis` *(boolean) [=off]*
Flag specifying whether to apply coriolis force.

`rotation_center` *or* *`center`* *(array) [={0,0,0}]*
Center of rotation, specified in global xyz coordinate system. Used with centrifugal force.

`angular_velocity` *or* *`ang_vel`* *(array) [={0,0,0}]*
Vector of angular velocity of the rotating frame, in radians per unit of time.

Description

This command is obsolete and will be removed eventually. The `REFERENCE_FRAME` command should be used instead.

> **Note:** Using both commands at the same time will normally produce incorrect results.

This command specifies the body forces resulting from solving a problem in a rotating frame of reference. Only a constant *`rotation_center`* and a constant *`angular_velocity`* are supported. The centrifugal and coriolis terms appear as the last two terms in the momentum equation:

$$\rho\frac{\partial \boldsymbol{u}}{\partial t} + \rho\boldsymbol{u} \bullet \nabla\boldsymbol{u} + \nabla p = \nabla \bullet \tau - \rho\Omega \times \Omega \times (\boldsymbol{x} - \boldsymbol{x}_0) - 2\rho\Omega \times \boldsymbol{u} \qquad (53)$$

where ρ is the density; **u** is the velocity vector; p is the pressure; $\tau = [\tau_{ij}]$ is the viscous stress tensor; **x** is the current coordinate vector; x_0 is the center of rotation, given by *`rotation_center`*; and Ω is the angular rotation rate vector, given by *`angular_velocity`*. `ROTATION_FORCE` commands are referenced by `BODY_FORCE` commands, which in turn are referenced by `ELEMENT_SET` commands:

```
ROTATION_FORCE ( "rotating frame" ) {
   centrifugal                          = on
   coriolis                             = on
```

```
   center                                = { 0, 0, 0 }
   angular_velocity                      = { 0, 1, 0 }
}
BODY_FORCE( "my body force" ) {
   rotation_force                        = "rotating frame"
   ...
}
ELEMENT_SET( "fluid in rotating frame of reference" ) {
   body_force                            = "my body force"
   ...
}
```

In this example, the frame of reference rotates around the y axis at the rate of one radian per unit of time, with the axis of rotation centered at the point (0,0,0) in the global xyz coordinate system. The right hand rule is used to define the direction of rotation of the reference frame.

VOLUME_HEAT_SOURCE

Specifies heat source per unit volume for the energy equation.

Type

AcuSolve Command

Syntax

```
VOLUME_HEAT_SOURCE("name") {parameters...}
```

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of volume heat source.

none	No volume heat source.
constant or const	Constant volume heat source. Requires *volume_heat_source*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
user_function or user	User-defined function. Requires *user_function*, *user_values* and *user_strings*.
element_list	Parameter to scale the distributed heat source. Used with *element_volume_heat_sources* in ELEMENT_SET. Requires MULTIPLIER_FUNCTION and BODY_FORCE.

volume_heat_source (real) [=0]

The constant value of the volume heat source. Used with constant type.

curve_fit_values or *curve_values (array) [={0,0}]*

A two-column array of independent-variable/heat-source data values. Used with piecewise_linear and cubic_spline types.

curve_fit_variable or *curve_var (enumerated) [=temperature]*

Independent variable of the curve fit. Used with piecewise_linear and cubic_spline types.

x_coordinate or xcrd	X-component of coordinates.
y_coordinate or ycrd	Y-component of coordinates.
z_coordinate or zcrd	Z-component of coordinates.

x_reference_coordinate or xrefcrd	X-component of reference coordinates.
y_reference_coordinate or yrefcrd	Y-component of reference coordinates.
z_reference_coordinate or zrefcrd	Z-component of reference coordinates.
x_velocity or xvel	X-component of velocity.
y_velocity or yvel	Y-component of velocity.
z_velocity or zvel	Z-component of velocity.
velocity_magnitude or vel_mag	Velocity magnitude.
pressure or pres	Pressure.
temperature or temp	Temperature.
eddy_viscosity or eddy	Turbulence eddy viscosity.
kinetic_energy or tke	Turbulence kinetic energy.
velocity_scale or tvel	Transition velocity scale.
dissipation_rate or teps	Turbulence dissipation rate.
eddy_frequency or tomeg	Turbulence frequency.
intermittency or tintc	Transition intermittency.
transition_re_theta or treth	Transition Re-Theta.
eddy_frequency or tomega	Turbulence frequency.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.
species_7 or spec7	Species 7.
species_8 or spec8	Species 8.

species_9 or spec9	Species 9.
mesh_x_displacement or mesh_xdisp	X-component of mesh displacement.
mesh_y_displacement or mesh_ydisp	Y-component of mesh displacement.
mesh_z_displacement or mesh_zdisp	Z-component of mesh displacement.
mesh_displacement_magnitude or mesh_disp_mag	Mesh displacement magnitude.
mesh_x_velocity or mesh_xvel	X-component of mesh velocity.
mesh_y_velocity or mesh_yvel	Y-component of mesh velocity.
mesh_z_velocity or mesh_zvel	Z-component of mesh velocity.
mesh_velocity_magnitude or mesh_vel_mag	Mesh velocity magnitude.

user_function or *user* *(string) [no default]*
Name of the user-defined function. Used with user_function type.

user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function type.

user_strings *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with user_function type.

multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the volume heat source. If none, no scaling is performed.

feedback_condition *(string) [=no default]*
User-given name of the FEEDBACK_CONDITION. This condition enables control of a VOLUME_HEAT_SOURCE based on a control variable at a point defined in the FEEDBACK_CONDITION.

Description

This command specifies the heat source per unit volume applied to the energy equation:

$$\rho\frac{\partial h}{\partial t}+\rho\boldsymbol{u}\bullet\nabla h\;=\;\nabla\bullet\boldsymbol{q}+s \qquad (54)$$

where ρ is the density, h is the enthalpy, **u** is the velocity vector, **q** is the diffusive heat flux vector and s is the heat source per unit volume defined here. `VOLUME_HEAT_SOURCE` commands are referenced by `BODY_FORCE` commands, which in turn are referenced by `ELEMENT_SET` commands:

```
VOLUME_HEAT_SOURCE( "my heat source" ) {
   type                                = constant
   volume_heat_source                  = 10
}
BODY_FORCE( "my body force" ) {
   volume_heat_source                  = "my heat source"
   ...
}
ELEMENT_SET( "fluid with heat source" ) {
   body_force                          = "my body force"
   ...
}
```

This example defines a constant heat source per unit volume. Positive values add heat to the system, while negative values remove heat from the system.

A constant volume heat source applies a spatially uniform heat source per unit volume, as in the above example.

Volume heat source models of types piecewise_linear and cubic_spline may be used to define the heat source as a function of a single independent variable. For example,

```
VOLUME_HEAT_SOURCE( "curve fit heat source" ) {
   type                                = piecewise_linear
   curve_fit_values                    = { 0., 0. ;
                                           1., 1. ;
                                           2., 4. ; }
   curve_fit_variable                  = species_1
}
```

defines a heat source as a function of the first species. In this case, the problem must contain species transport equations; see the `EQUATION` command. The *`curve_fit_values`* parameter is a two-column array corresponding to the independent variable and the heat source values. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *`curve_fit_variable`* falls outside of the curve fit limits.

The *`curve_fit_values`* data may be read from a file. For the above example, the curve fit values may be placed in a file, say `heat_source.fit`:

```
0.      0.
1.      1.
2.      3.
```

and read by:

```
VOLUME_HEAT_SOURCE( "curve fit heat source" ) {
   type                                = piecewise_linear
   curve_fit_values                    = Read( "heat_source.fit" )
   curve_fit_variable                  = species_1
}
```

A volume heat source of type element_list is used to scale the distributed heat source based on the element number, which may be read from a file, named `heatSource.vhs` in this example. This is

necessary since the VOLUME_HEAT_SOURCE command cannot create a spatially varying heat source, unless implemented manually via a user-defined function, and the *element_volume_heat_source* entry in the ELEMENT_SET does not support MULTIPLIER_FUNCTION. The element_list type removes this limitation by applying both entries together. Below is an example of heat source varying with elements.

```
98046     2.1E12
98047     2,14E12
98052     2,17E12
…
154253  2.2E12
154257  2.21E12
```

The file (heatSource.vhs) is assigned to the solid "SolidVol":

```
ELEMENT_SET (“solidVol”) {
   volume_sets = {“tet4_solidVol”}
   medium = solid
   material_model = “copper”
   body_force = “heat”
   element_volume_heat_sources = Read (“heatSource.vhs”)
}
```

The type of volume heat source element_list can be utilized to scale *element_volume_heat_sources* using MULTIPLIER_FUNCTION and BODY_FORCE.

```
MULTIPLIER_FUNCTION ("heatSourceMF") {
    type = piecewise_linear
    curve_fit_values = {0,1;
                        10,2;
                        20,0;
                        30,0.5;
                        100,1}
    curve_fit_variable = time_step
}

VOLUME_HEAT_SOURCE ("heat") {
    type = element_list
    multiplier_function = "heatSourceMF"
}

BODY_FORCE ("heat") {
   volume_heat_source = "heat"
}
```

A volume heat source of type user_function may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider the case where heat source is proportional to species one, but is applied slowly as a function of time. Then the input command may be given by:

```
VOLUME_HEAT_SOURCE( "UDF heat source" ) {
   type                                = user_function
   user_function                       = "usrHeatSourceExample"
   user_values                         = { 2.5,   # species scaling
                                           10.}   # time to apply in full
}
```

where the user-defined function "usrHeatSourceExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrHeatSourceExample ) ;                                  /* function
 prototype                */
Void usrHeatSourceExample (
    UdfHd                  udfHd,                                        /* Opaque handle
 for accessing data */
    Real*                  outVec,                                       /* Output vector
                  */
    Integer                nItems,                                       /* Number of
 elements               */
    Integer                vecDim                                        /* = 1
                  */
) {
    Integer                elem ;                                        /* an element
 counter                */
    Real                   fTime ;                                       /* final time
                  */
    Real                   scale ;                                       /* scaling
 factor                    */
    Real                   time ;                                       /* run time
                  */
    Real*                  spec ;                                        /* species
 vector                    */
    Real*                  usrVals ;                                     /* user values
                  */
udfCheckNumUsrVals( udfHd, 2 ) ;                                         /* check for
 error                  */
    usrVals                = udfGetUsrVals( udfHd ) ;                    /* get the user
 vals                */
    scale                  = usrVals[0] ;                                /* get the
 factor                    */
    ftime                  = usrVals[1] ;                                /* get the final
 time                  */
    time                   = usrVals[2] ;                                /* get the run
 time                */
    spec                   = udfGetElmData( udfHd,UDF_ELM_SPECIES ) ; /* get the
 species                   */
    if ( time < fTime ) {
         scale = scale * time / fTime ;                                  /* reduce
 scaling                  */
    }
    for ( elem = 0 ; elem < nItems ; elem++ ) {
          outVec[elem] = scale * spec[elem] ;
    }
} /* end of usrHeatSourceExample() */
```

The dimension of the returned volume heat source vector, *outVec*, is the number of elements.

The *multiplier_function* parameter may be used to uniformly scale the volume heat source values. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file. For example, assuming that species one remains between zero and 10, the above heat source using the user-defined function may be obtained by:

```
VOLUME_HEAT_SOURCE( "equivalent to UDF heat source" ) {
    type                              = piecewise_linear
    curve_fit_values                  = { 0, 0 ;
                                          10, 25 ; }
    curve_fit_variable                = species_1
```

```
    multiplier_function                 = "ramped"
}
MULTIPLIER_FUNCTION( "ramped" ) {
    type                                = piecewise_linear
    curve_fit_values                    = { 0, 0 ; 10, 1 }
    curve_fit_variable                  = time
}
```

VOLUME_SPECIES_SOURCE

Specifies species source per unit volume for a species transport equation.

Type

AcuSolve Command

Syntax

```
VOLUME_SPECIES_SOURCE("name") {parameters...}
```

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of volume species source.

none	No volume species source.
constant or const	Constant volume species source. Requires *volume_species_source*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
user_function or user	User-defined function. Requires *user_function*, *user_values* and *user_strings*.

volume_species_source (real) [=0]

The constant value of the volume species source. Used with constant type.

curve_fit_values or curve_values (array) [={0,0}]

A two-column array of independent-variable/species-source data values. Used with piecewise_linear and cubic_spline types.

curve_fit_variable or curve_var (enumerated) [=temperature]

Independent variable of the curve fit. Used with piecewise_linear and cubic_spline types.

x_coordinate or xcrd	X-component of coordinates.
y_coordinate or ycrd	Y-component of coordinates.
z_coordinate or zcrd	Z-component of coordinates.
x_reference_coordinate or xrefcrd	X-component of reference coordinates.

y_reference_coordinate or yrefcrd	Y-component of reference coordinates.
z_reference_coordinate or zrefcrd	Z-component of reference coordinates.
x_velocity or xvel	X-component of velocity.
y_velocity or yvel	Y-component of velocity.
z_velocity or zvel	Z-component of velocity.
velocity_magnitude or vel_mag	Velocity magnitude.
pressure or pres	Pressure.
temperature or temp	Temperature.
eddy_viscosity or eddy	Turbulence eddy viscosity.
kinetic_energy or tke	Turbulence kinetic energy.
velocity_scale or tvel	Transition velocity scale.
dissipation_rate or teps	Turbulence dissipation rate.
eddy_frequency or tomeg	Turbulence frequency.
intermittency or tintc	Transition intermittency.
transition_re_theta or treth	Transition Re-Theta.
eddy_frequency or tomega	Turbulence frequency.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.
species_7 or spec7	Species 7.
species_8 or spec8	Species 8.
species_9 or spec9	Species 9.

mesh_x_displacement or mesh_xdisp	X-component of mesh displacement.
mesh_y_displacement or mesh_ydisp	Y-component of mesh displacement.
mesh_z_displacement or mesh_zdisp	Z-component of mesh displacement.
mesh_displacement_magnitude or mesh_disp_mag	Mesh displacement magnitude.
mesh_x_velocity or mesh_xvel	X-component of mesh velocity.
mesh_y_velocity or mesh_yvel	Y-component of mesh velocity.
mesh_z_velocity or mesh_zvel	Z-component of mesh velocity.
mesh_velocity_magnitude or mesh_vel_mag	Mesh velocity magnitude.

`user_function` *or* `user` *(string) [no default]*
Name of the user-defined function. Used with user_function type.

`user_values` *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function type.

`user_strings` *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with user_function type.

`multiplier_function` *(string) [=none]*
User-given name of the multiplier function for scaling the volume species source. If none, no scaling is performed.

Description

This command specifies the volumetric source per unit volume term applied to the transport equation for a species φ:

$$\rho\frac{\partial\varphi}{\partial t} + \rho\boldsymbol{u}\bullet\nabla\varphi = \nabla\bullet\Psi + \sigma \qquad (55)$$

where ρ is the density, **u** is the velocity vector, Ψ is the diffusive flux and σ is the species source per unit volume term defined here. `VOLUME_SPECIES_SOURCE` commands are referenced by `BODY_FORCE` commands, which in turn are referenced by `ELEMENT_SET` commands:

```
VOLUME_SPECIES_SOURCE( "my species source" ) {
   type                                = constant
   volume_species_source               = 10
```

```
}
BODY_FORCE( "my body force" ) {
   volume_species_1_source             = "my species source"
   ...
}
ELEMENT_SET( "fluid with species source" ) {
   body_force                          = "my body force"
   ...
}
```

This example defines a constant volumetric species source per unit volume. Positive values add species to the system, while negative values remove species from the system.

A constant volume species source applies a spatially uniform volumetric species source per unit volume, as in the above example.

Volume species source models of types piecewise_linear and cubic_spline may be used to define the species source as a function of a single independent variable. For example,

```
VOLUME_SPECIES_SOURCE( "curve fit species source" ) {
   type                                = piecewise_linear
   curve_fit_values                    = { 0., 0. ;
                                           1., 1. ;
                                           2., 4. ; }
   curve_fit_variable                  = species_1
}
```

defines a species source as a function of the first species. The *curve_fit_values* parameter is a two-column array corresponding to the independent variable and the species source values. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, say `species_source.fit`:

```
0.      0.
1.      1.
2.      3.
```

and read by:

```
VOLUME_SPECIES_SOURCE( "curve fit species source" ) {
   type                                = piecewise_linear
   curve_fit_values                    = Read( "species_source.fit" )
   curve_fit_variable                  = species_1
}
```

A volume species source of type user_function may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider the case where species source is a second-order polynomial of species one. Then the input command may be given by:

```
VOLUME_SPECIES_SOURCE( "UDF species source" ) {
   type                                = user_function
   user_function                       = "usrSpeciesSourceExample"
   user_values                         = { 1.0,  # constant coef.
                                           0.5,  # linear coef.
```

```
                                                0.1 } # quadratic coef
 }
```

where the user-defined function "usrSpeciesSourceExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrSpeciesSourceExample ) ;                              /* function
 prototype               */
Void usrSpeciesSourceExample (
    UdfHd                   udfHd,                                      /* Opaque handle
 for accessing data */
    Real*                   outVec,                                     /* Output vector
                   */
    Integer                 nItems,                                     /* Number of
 elements                */
    Integer                 vecDim                                      /* = 1
                   */
) {
    Integer                 elem ;                                      /* an element
 counter                */
    Real                    coef0 ;                                     /* constant
 coefficient             */
    Real                    coef1 ;                                     /* linear
 coefficient              */
    Real                    coef2 ;                                     /* quadratic
 coefficient             */
    Real*                   spec ;                                      /* species
 vector                  */
    Real*                   usrVals ;                                   /* user values
                   */
udfCheckNumUsrVals( udfHd, 3 ) ;                                        /* check for
 error                   */
    usrVals                 = udfGetUsrVals( udfHd ) ;                  /* get the user
 vals                */
    coef0                   = usrVals[0] ;                              /* get coef
                   */
    coef1                   = usrVals[1] ;                              /* get coef 2
                   */
    coef2                   = usrVals[2] ;                              /* get coef 3
                   */
    spec                    = udfGetElmData( udfHd,UDF_ELM_SPECIES ) ; /* get the
 species                 */
    for ( elem = 0 ; elem < nItems ; elem++ ) {
        outVec[elem] = coef0
                     + coef1 * spec[elem]
                     + coef2 * spec[elem] * spec[elem] ;
    }
} /* end of usrSpeciesSourceExample() */
```

The dimension of the returned volume species source vector, *outVec*, is the number of elements.

The *multiplier_function* parameter may be used to uniformly scale the volume species source values. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file. For example, to slowly increase a source term from zero to the full amount by time step 10, you can specify:

```
VOLUME_SPECIES_SOURCE( "ramped species source" ) {
    type                                = constant
    volume_species_source               = 10
    multiplier_function                 = "ramped"
```

```
}
MULTIPLIER_FUNCTION( "ramped" ) {
    type                                = piecewise_linear
    curve_fit_values                    = { 0, 0 ; 10, 1 }
    curve_fit_variable                  = time_step
}
```

Material Model Commands

11

The Material Model commands are described in this chapter.

This chapter covers the following:

- BATTERY_MODEL (p. 270)
- CONDUCTIVITY_MODEL (p. 280)
- CONTACT_ANGLE_MODEL (p. 291)
- DENSITY_MODEL (p. 293)
- DIFFUSIVITY_MODEL (p. 300)
- ELECTRICAL_RESISTIVITY_MODEL (p. 305)
- EMISSIVITY_MODEL (p. 308)
- GAS_KINETIC_MODEL (p. 312)
- FIELD (p. 314)
- FIELD_INTERACTION_MODEL (p. 315)
- MATERIAL_MODEL (p. 329)
- MATERIAL_RADIATION_MODEL (p. 334)
- MULTI_FIELD_MODEL (p. 336)
- POROSITY_MODEL (p. 339)
- SOLAR_RADIATION_MODEL (p. 347)
- SPECIFIC_HEAT_MODEL (p. 351)
- SURFACE_TENSION_MODEL (p. 358)
- VISCOELASTIC_MODEL (p. 361)
- VISCOSITY_MODEL (p. 365)

All the individual models except the models related to surface tension and radiation are collected by the `MATERIAL_MODEL` command. Every element set of a problem must be assigned a material model (every layer for a shell element set) and the material model must contain all the individual models corresponding to the equations of the problem.

BATTERY_MODEL

Specifies a battery model and its parameters.

Type

AcuSolve Command

Syntax

```
BATTERY_MODEL("name") {parameters...}
```

Parameters

battery_model_type (enumerated) [=first_order]

Type of battery model.

first_order_ecm	Equivalent circuit model with a single ohmic resistance in series with one parallel RC (Resistor-Capacitor) pair to represent the dynamic voltage transients (or diffusion voltages).
second_order_ecm	Equivalent circuit model with a single ohmic resistance in series with two parallel RC pairs to represent the dynamic voltage transients (or diffusion voltages). The 2nd order model is considered the optimal choice, striking a balance between accuracy and ease of parameter determination.
third_order_ecm	Equivalent circuit model with a single ohmic resistance in series with three parallel RC pairs to represent the dynamic voltage transients (or diffusion voltages). This model has an advantage over lower-order versions by more accurately capturing the battery's dynamic voltage response.

ohmic_resistance_type (enumerated) [=constant]

Ohmic resistance in first or second order ECM model.

constant or const	Constant ohmic resistance. Requires ohmic_resistance.
piecewise_linear or linear	Piecewise linear curve fit ohmic resistance profile. Requires curve_fit_values and curve_fit_variable.
piecewise_bilinear or bilinear	Piecewise bilinear curve fit ohmic resistance profile. Requires curve_fit_values and curve_fit_variable.
cubic_spline or spline	Cubic spline curve fit electrical current profile. Requires curve_fit_values and curve_fit_variable.

ohmic_resistance (real) [=0.0]

Constant value of ohmic resistance. Used when *ohmic_resistance_type* = constant.

ohmic_resistance_curve_fit_values (array) [={0,0}]
A two-column array of independent-variable/isotropic ohmic resistance data values. The independent variable is either state of charge or time. Used with *ohmic_resistance_type* = piecewise_linear or *ohmic_resistance_type* = cubic_spline.

ohmic_resistance_curve_fit_variable (enumerated) [=soc]
Independent variable of the curve fit for ohmic resistance. Used with *ohmic_resistance_type* = piecewise_linear or *ohmic_resistance_type* = cubic_spline.

soc	state_of_charge
time	time

ohmic_resistance_soc_temperature_values (2D array) [={0,0;0,1}]
Independent variable of the curve fit for ohmic resistance. Used with *ohmic_resistance_type* = piecewise_bilinear.

ohmic_resistance_bilinear_curve_fit_row_variable [=soc]
Independent row variable of the curve fit for ohmic resistance. Used with *ohmic_resistance_type* = piecewise_bilinear.

soc	state_of_charge

ohmic_resistance_bilinear_curve_fit_column_variable [=temperature]
Independent column variable of the curve fit for ohmic resistance. Used with *ohmic_resistance_type* = piecewise_bilinear.

temp	temperature

polarization_resistance1_type (enumerated) [=constant]
Polarization resistance for the first RC pair in first or second order ECM model.

constant or const	Constant polarization resistance. Requires polarization_resistance1.
piecewise_linear or linear	Piecewise linear curve fit polarization resistance 1 profile. Requires curve_fit_values and curve_fit_variable.
piecewise_bilinear or bilinear	Piecewise bilinear curve fit polarization resistance 1 profile. Requires curve_fit_values and curve_fit_variable.
cubic_spline or spline	Cubic spline curve fit polarization resistance 1 profile. Requires curve_fit_values and curve_fit_variable.

polarization_resistance1 (real) [=0.0]
Constant value of polarization resistance 1. When *polarization_resistance1_type* = constant.

polarization_resistance1_curve_fit_values (array) [={0,0}]
A two-column array of independent-variable/isotropic polarization resistance 1 data values. The independent variable is either state of charge or time. Used with *polarization_resistance1_type* = piecewise_linear or *polarization_resistance1_type* = cubic_spline.

polarization_resistance1_curve_fit_variable *(enumerated) [=soc]*
Independent variable of the curve fit for polarization resistance 1. Used with *polarization_resistance1_type* = piecewise_linear or *polarization_resistance1_type* = cubic_spline.

soc	state_of_charge
time	time

polarization_resistance1_soc_temperature_values *(2D array) [={0,0;0,0}]*
Independent variable of the curve fit for polarization resistance 1. Used with *polarization_resistance1_type* = piecewise_bilinear.

polarization_resistance1_bilinear_curve_fit_row_variable *[=soc]*
Independent row variable of the curve fit for polarization resistance 1. Used with *polarization_resistance1_type* = piecewise_bilinear.

soc	state_of_charge

polarization_resistance1_bilinear_curve_fit_column_variable *[=temperature]*
Independent column variable of the curve fit for polarization resistance 1. Used with *polarization_resistance1_type* = piecewise_bilinear.

temp	temperature

polarization_resistance2_type *(enumerated) [=constant]*
Polarization resistance for the second RC in second order ECM model.

constant or const	Constant polarization resistance. Requires polarization_resistance2.
piecewise_linear or linear	Piecewise linear curve fit polarization resistance 2 profile. Requires curve_fit_values and curve_fit_variable.
piecewise_bilinear or bilinear	Piecewise bilinear curve fit polarization resistance 2 profile. Requires curve_fit_values and curve_fit_variable.
cubic_spline or spline	Cubic spline curve fit polarization resistance 2 profile. Requires curve_fit_values and curve_fit_variable.

polarization_resistance2 *(real) [=0.0]*
Constant value of polarization resistance 2. When *polarization_resistance2_type* = constant.

polarization_resistance2_curve_fit_values *(array) [={0,0}]*
A two-column array of independent-variable/isotropic polarization resistance 2 data values. The independent variable is either state of charge or time. Used with *polarization_resistance2_type* = piecewise_linear or *polarization_resistance2_type* = cubic_spline.

polarization_resistance2_curve_fit_variable *(enumerated) [=soc]*
Independent variable of the curve fit for polarization resistance 2. Used with *polarization_resistance2_type* = piecewise_linear or *polarization_resistance2_type* = cubic_spline.

soc	state_of_charge
time	time

polarization_resistance2_soc_temperature_values (2D array) [={0,0;0,1}]
Independent variable of the curve fit for polarization resistance 2. Used with *polarization_resistance2_type* = piecewise_bilinear.

polarization_resistance2_bilinear_curve_fit_row_variable [=soc]
Independent row variable of the curve fit for polarization resistance 2. Used with *polarization_resistance2_type* = piecewise_bilinear.

soc	state_of_charge

polarization_resistance2_bilinear_curve_fit_column_variable [=temperature]
Independent column variable of the curve fit for polarization resistance 2. Used with *polarization_resistance2_type* = piecewise_bilinear.

temp	temperature

polarization_capacitance1_type (enumerated) [=constant]
Polarization capacitance for the first RC pair in first or second order ECM model.

constant or const	Constant polarization capacitance. Requires polarization_capacitance1.
piecewise_linear or linear	Piecewise linear curve fit polarization capacitance 1 profile. Requires curve_fit_values and curve_fit_variable.
piecewise_bilinear or bilinear	Piecewise bilinear curve fit polarization capacitance 1 profile. Requires curve_fit_values and curve_fit_variable.
cubic_spline or spline	Cubic spline curve fit polarization capacitance 1 profile. Requires curve_fit_values and curve_fit_variable.

polarization_capacitance1 (real) [=0.0]
Constant value of polarization capacitance 1. When *polarization_capacitance1_type* = constant.

polarization_capacitance1_curve_fit_values (array) [={0,0}]
A two-column array of independent-variable/isotropic polarization capacitance 1 data values. The independent variable is either state of charge or time. Used with *polarization_capacitance1_type* = piecewise_linear or *polarization_capacitance1_type* = cubic_spline.

polarization_capacitance1_curve_fit_variable (enumerated) [=soc]
Independent variable of the curve fit for polarization capacitance 1. Used with *polarization_capacitance1_type* = piecewise_linear or *polarization_capacitance1_type* = cubic_spline.

soc	state_of_charge
time	time

polarization_capacitance1_soc_temperature_values (2D array) [={0,0;0,1}]
Independent variable of the curve fit for polarization capacitance 1. Used with *polarization_capacitance1_type* = piecewise_bilinear.

polarization_capacitance1_bilinear_curve_fit_row_variable [=soc]
Independent row variable of the curve fit for polarization capacitance 1. Used with *polarization_capacitance1_type* = piecewise_bilinear.

soc	state_of_charge

polarization_capacitance1_bilinear_curve_fit_column_variable [=temperature]
Independent column variable of the curve fit for polarization capacitance 1. Used with *polarization_capacitance1_type* = piecewise_bilinear.

temp	temperature

polarization_capacitance2_type (enumerated) [=constant]
Polarization capacitance for the second RC pair in second order ECM model.

constant or const	Constant polarization capacitance. Requires polarization_capacitance2.
piecewise_linear or linear	Piecewise linear curve fit polarization capacitance 2 profile. Requires curve_fit_values and curve_fit_variable.
piecewise_bilinear or bilinear	Piecewise bilinear curve fit polarization capacitance 2 profile. Requires curve_fit_values and curve_fit_variable.
cubic_spline or spline	Cubic spline curve fit polarization capacitance 2 profile. Requires curve_fit_values and curve_fit_variable.

polarization_capacitance2 (real) [=0.0]
Constant value of polarization capacitance 2. When *polarization_capacitance2_type* = constant.

polarization_capacitance2_curve_fit_values (array) [={0,0}]
A two-column array of independent-variable/isotropic polarization capacitance 2 data values. The independent variable is either state of charge or time. Used with *polarization_capacitance2_type* = piecewise_linear or *polarization_capacitance2_type* = cubic_spline.

polarization_capacitance2_curve_fit_variable (enumerated) [=soc]
Independent variable of the curve fit for polarization capacitance 2. Used with *polarization_capacitance2_type* = piecewise_linear or *polarization_capacitance2_type* = cubic_spline.

soc	state_of_charge
time	time

polarization_capacitance2_soc_temperature_values (2D array) [={0,0;0,1}]
Independent variable of the curve fit for polarization capacitance 2. Used with *polarization_capacitance2_type* = piecewise_bilinear.

battery_capacity *(real) [=5.0]*
The capacity of a battery cell measured in Ah (Amp-hr).

polarization_capacitance2_bilinear_curve_fit_row_variable *[=soc]*
Independent row variable of the curve fit for polarization capacitance 2. Used with *polarization_capacitance2_type* = piecewise_bilinear.

soc	state_of_charge

polarization_capacitance2_bilinear_curve_fit_column_variable *[=temperature]*
Independent column variable of the curve fit for polarization capacitance 2. Used with *polarization_capacitance2_type* = piecewise_bilinear.

temp	temperature

open_circuit_voltage_type *(enumerated) [=constant]*
Open circuit potential is the potential established between positive and negative electrode terminals when current is zero.

constant or const	Constant open circuit potential. Requires open_circuit_voltage.
piecewise_linear or linear	Piecewise linear curve fit open circuit potential profile. Requires curve_fit_values and curve_fit_variable.
piecewise_bilinear or bilinear	Piecewise bilinear curve fit open circuit potential profile. Requires curve_fit_values and curve_fit_variable.
cubic_spline or spline	Cubic spline curve fit open circuit potential profile. Requires curve_fit_values and curve_fit_variable.

open_circuit_voltage *(real) [=0.0]*
Constant value of open circuit potential. When *open_circuit_voltage_type* = constant.

open_circuit_voltage_curve_fit_values *(array) [={0,0}]*
A two-column array of independent-variable open circuit potential data values. The independent variable is either state of charge or time. Used with *open_circuit_voltage* = piecewise_linear or *open_circuit_voltage_type* = cubic_spline.

open_circuit_voltage_curve_fit_variable *(enumerated) [=soc]*
Independent variable of the curve fit for open circuit potential. Used with *open_circuit_voltage* = piecewise_linear or *open_circuit_voltage_type* = cubic_spline.

soc	state_of_charge
time	time

open_circuit_voltage_soc_temperature_values *(2D array) [={0,0;0,1}]*
Independent variable of the curve fit for open circuit potential. Used with *open_circuit_voltage_type* = piecewise_bilinear.

open_circuit_voltage_bilinear_curve_fit_row_variable *[=soc]*
Independent row variable of the curve fit for open circuit potential. Used with *open_circuit_voltage_type* = piecewise_bilinear.

soc	state_of_charge

open_circuit_voltage_bilinear_curve_fit_column_variable [=temperature]
Independent column variable of the curve fit for open circuit potential. Used with *open_circuit_voltage_type* = piecewise_bilinear.

temp	temperature

entropic_heat_coefficient_type (enumerated) [=constant]
Entropic heat coefficient is the derivative of open circuit potential with respect to temperature. It represents reversible heat generation in the battery cell.

constant or const	Constant entropic heat coefficient. Requires entropic_heat_coefficient.
piecewise_linear or linear	Piecewise linear curve fit entropic heat coefficient profile. Requires curve_fit_values and curve_fit_variable.
piecewise_bilinear or bilinear	Piecewise bilinear curve fit entropic heat coefficient profile. Requires curve_fit_values and curve_fit_variable.
cubic_spline or spline	Cubic spline curve fit entropic heat coefficient profile. Requires curve_fit_values and curve_fit_variable.

entropic_heat_coefficient (real) [=0.0]
Constant value of entropic heat coefficient. When *entropic_heat_coefficient_type* = constant.

entropic_heat_coefficient_curve_fit_values (array) [={0,0}]
A two-column array of independent-variable entropic heat coefficient data values. The independent variable is either state of charge or time. Used with *entropic_heat_coefficient_type* = piecewise_linear or *entropic_heat_coefficient_type* = cubic_spline.

entropic_heat_curve_fit_variable (enumerated) [=soc]
Independent variable of the curve fit for entropic heat coefficient. Used with *entropic_heat_coefficient_type* = piecewise_linear or *entropic_heat_coefficient_type* = cubic_spline.

soc	state_of_charge
time	time

entropic_heat_soc_temperature_values (2D array) [={0,0;0,1}]
Independent variable of the curve fit for entropic heat coefficient. Used with *entropic_heat_coefficient_type* = piecewise_bilinear.

entropic_heat_bilinear_curve_fit_row_variable [=soc]
Independent row variable of the curve fit for entropic heat coefficient. Used with *entropic_heat_coefficient_type* = piecewise_bilinear.

soc	state_of_charge

entropic_heat_bilinear_curve_fit_column_variable [=temperature]
Independent column variable of the curve fit for entropic heat coefficient. Used with *entropic_heat_coefficient_type* = piecewise_bilinear.

temp temperature

Description

The BATTERY_MODEL is included as part of a material model of a battery cell when using *equation* = battery_joule_heating or *equation* = battery_msmd. The BATTERY_MODEL specifies the parameters for a first or second order equivalent circuit model (ECM), the entropic heat source and the battery capacity.

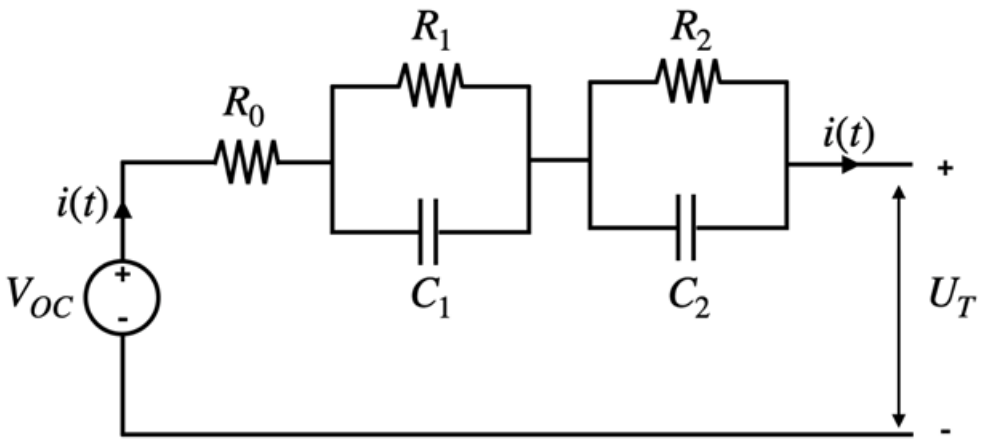


Figure 8: Equivalent Circuit Battery Model

In an ECM model the electric behavior, for example, the voltage-current response, is modeled using a phenomenological electric circuit approach. For example, in a second order ECM (see above figure), that is, two RC pairs, the governing set of ODEs are given by:

$$U_T = V_{oc}(\text{soc}, T) + V_1 + V_2 - R_o(\text{soc}, T) i(t) \tag{56}$$

$$\frac{dV_1}{dt} = \frac{1}{R_1(\text{soc}, T) C_1(\text{soc}, T)} V_1 - \frac{1}{C_1(\text{soc}, T)} i(t) \tag{57}$$

$$\frac{dV_2}{dt} = \frac{1}{R_2(\text{soc}, T) C_2(\text{soc}, T)} V_2 - \frac{1}{C_2(\text{soc}, T)} i(t) \tag{58}$$

where U_T is the terminal voltage, V_{oc} is the open circuit voltage, R_o is the ohmic resistance, R_1 is the polarization resistance of the first RC pair, R_2 is the polarization resistance of the second RC pair, C_1 is the polarization capacitance of the first RC pair, C_2 is the polarization capacitance of the second RC pair, and $i(t)$ is the input current. The output terminal voltage is therefore dependent on the input current and the ECM parameters (R_o, R_1, R_2, C_1) which can be both state of charge and temperature dependent.

The battery state of charge (soc) is updated using coulomb counting:

$$\frac{d\text{soc}}{dt} = \frac{1}{3600 Q_{\text{Ah}}} \tag{59}$$

The equivalent circuit models are enabled using the BATTERY_MODEL command with *battery_model_type* = first_order_ecm or *battery_model_type* = second_order_ecm.

An example BATTERY_MODEL with a second order ECM model and an *open_circuit_voltage_type* = linear and values read from a file is shown below:

```
BATTERY_MODEL( "ecm" ) {
    battery_model_type = second_order_ecm
    ohmic_resistance_type = constant
    ohmic_resistance = 0.011
    polarization_resistance1_type = constant
    polarization_resistance1 = 0.0063
    polarization_resistance2_type = constant
    polarization_resistance2 = 0.0043
    polarization_capacitance1_type = constant
    polarization_capacitance1 = 657.42
    polarization_capacitance2_type = constant
    polarization_capacitance2 = 6574.23
    open_circuit_voltage_type = piecewise_linear
    open_circuit_voltage_curve_fit_values = Read( "open_circuit_voltage.fit" )
    entropic_heat_coefficient_type = constant
    entropic_heat_coefficient = 0.0
    battery_capacity = 18000
}
```

"open_circuit_voltage.fit" defines the file with the ocp versus soc relationship, for example:

```
1.0  4.2828
0.99 4.2681
0.98 4.2538
…
```

For ohmic resistance, polarization resistance 1 and 2, polarization capacitance 1 and 2, open circuit voltage, and entropic heat coefficient, you can also have them as a variable for both soc and temperature. Two example inputs are given below:

1. Ohmic resistance specified as a function of both soc and temperature:

```
BATTERY_MODEL( "ecm" ) {
 ...
 ohmic_resistance_type = piecewise_bilinear
 ohmic_resistance_soc_temperature_values = {
            0,     0.2,       0.4,      0.6,       0.8,     1.0 ;
            298,   0.011,     0.0111,   0.0112,    0.0113,  0.0114 ;
            333,   0.0115,    0.0116,   0.0117,    0.0118,  0.0119 ; }
 ...
}
```

2. Open circuit potential as a bilinear fit for both soc and temperature:

```
BATTERY_MODEL( "ecm" ) {
 …
 open_circuit_voltage_type = piecewise_bilinear
 open_circuit_voltage_soc_temperature_values = {
            0,   0.2,   0.4,   0.6,   0.8,   1.0 ;
            298, 3.783, 3.896, 3.968, 4.039, 4.147 ;
            333, 3.782, 3.903, 3.976, 4.056, 4.150 ;
 …
}
```

Bernardi's heat generation equation

In this approach, a source term that represents the irreversible and reversible heat generation is included in the energy equation, given by:

$$q = \frac{i(t)}{V}\left[\left(V_{oc} - U_T\right) - T\frac{dV_{oc}}{dT}\right] + \sigma_+\nabla^2\sigma_+ + \sigma_-\nabla^2\sigma_- \qquad (60)$$

The first term of the RHS represents the cell overpotential and includes irreversibilities, such as ohmic losses and mass transport limitations, while the second term represents entropic heat generation. The last two terms represent ohmic heat generation and are only enabled when *electric_potential* = battery_msmd. Bernardi's heat generation equation is automatically included in the energy equation as a source term for all battery models. The entropic heat is included in the BATTERY_MODEL and can be both temperature and state_of_charge dependent.

CONDUCTIVITY_MODEL

Specifies a thermal conductivity model for the temperature equation.

Type

AcuSolve Command

Syntax

```
CONDUCTIVITY_MODEL("name") {parameters...}
```

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of the thermal conductivity.

constant or const	Constant isotropic conductivity. Requires *conductivity*.
constant_prandtl_number or const_pr	Constant Prandtl number isotropic conductivity. Requires *prandtl_number*.
ramped or ramp	Ramp the isotropic conductivity to a constant value. Requires *conductivity*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit isotropic conductivity. Requires *curve_fit_values* and *curve_fit_variable*.
user_function or user	User-defined function. Requires *user_function*, *user_values* and *user_strings*.
constant_anisotropic or const_aniso	Constant anisotropic conductivity. Requires *anisotropic_conductivity*.
piecewise_linear_anisotropic	Piecewise linear curve fit anisotropic conductivity. Requires *anisotropic_curve_fit_values* and *anisotropic_curve_fit_variable*.
cubic_spline_anisotropic	Cubic spline curve fit anisotropic conductivity. Requires *anisotropic_curve_fit_values* and *anisotropic_curve_fit_variable*.
user_function_anisotropic	User-defined function anisotropic conductivity. Requires *anisotropic_user_function*, *anisotropic_user_values* and *anisotropic_user_strings*.

cylindrical_anisotropic	Anisotropic conductivity in a cylindrical system. Requires *cylindrical_center*, *cylindrical_axis*, *anisotropic_conductivity*.
sutherland	Sutherland temperature dependent conductivity. Requires *sutherland_reference_conductivity*, *sutherland_reference_temperature*, *sutherland_constant*.
gas_kinetic	Gas kinetic conductivity model for gases only. Requires GAS_KINETIC_MODEL.

conductivity or *cond* *(real) >=0 [=0]*
Constant value of the isotropic thermal conductivity. Used with constant and ramped types.

prandtl_number or *pr* *(real) >=0 [=0.71]*
Prandtl number value. Used with constant_prandtl_number type.

curve_fit_values or *curve_values* *(array) [={0,0}]*
A two-column array of independent-variable/isotropic conductivity data values. Used with piecewise_linear and cubic_spline types.

curve_fit_variable or *curve_var* *(enumerated) [=temperature]*
Independent variable of the curve fit for isotropic conductivity. Used with piecewise_linear and cubic_spline types.

x_coordinate or xcrd	X-component of coordinates.
y_coordinate or ycrd	Y-component of coordinates.
z_coordinate or zcrd	Z-component of coordinates.
x_reference_coordinate or xrefcrd	X-component of reference coordinates.
y_reference_coordinate or yrefcrd	Y-component of reference coordinates.
z_reference_coordinate or zrefcrd	Z-component of reference coordinates.
temperature or temp	Temperature.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.

species_7 or spec7 Species 7.

species_8 or spec8 Species 8.

species_9 or spec9 Species 9.

`sutherland_reference_conductivity` *(real) >= 0 [=0.0241]*
Sutherland model reference conductivity at `sutherland_reference_temperature`. Used with sutherland type.

`sutherland_reference_temperature` *(real) > 0 [=273]*
Sutherland model reference temperature corresponding to `sutherland_reference_conductivity`. Used with sutherland type.

`sutherland_constant` *(real) > 0 [=194]*
Sutherland model constant. Used with sutherland type.

`user_function` *or* `user` *(string) [no default]*
Name of the user-defined function. Used with user_function type.

`user_values` *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function type.

`user_strings` *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with user_function type.

`anisotropic_conductivity` *or* `aniso_cond` *(array) [={0,0,0,0,0,0}]*
A six-element array of the form {k11, k22, k33, k12, k23, k31} for the anisotropic conductivity. Used with constant_anisotropic type.

`anisotropic_curve_fit_values` *or* `aniso_curve_values` *(array) [={0,0,0,0,0,0,0}]*
A seven-column array of independent-variable/anisotropic conductivity data values, in the order variable, k11, k22, k33, k12, k23, k31. Used with piecewise_linear_anisotropic and cubic_spline_anisotropic types.

`anisotropic_curve_fit_variable` *or* `aniso_curve_var` *(enumerated) [=temperature]*
Independent variable of the curve fit for anisotropic conductivity. Used with piecewise_linear_anisotropic and cubic_spline_anisotropic types.

x_coordinate or xcrd X-component of coordinates.

y_coordinate or ycrd Y-component of coordinates.

z_coordinate or zcrd Z-component of coordinates.

x_reference_coordinate or xrefcrd X-component of reference coordinates.

y_reference_coordinate or yrefcrd Y-component of reference coordinates.

z_reference_coordinate or zrefcrd Z-component of reference coordinates.

temperature or temp	Temperature.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.
species_7 or spec7	Species 7.
species_8 or spec8	Species 8.
species_9 or spec9	Species 9.

anisotropic_user_function or *aniso_user* *(string) [no default]*
Name of the user-defined function for anisotropic conductivity. Used with user_function_anisotropic type.

anisotropic_user_values *(array) [={}]*
Array of values to be passed to the user-defined function for anisotropic conductivity. Used with user_function_anisotropic type.

anisotropic_user_strings *(list) [={}]*
Array of strings to be passed to the user-defined function for anisotropic conductivity. Used with user_function_anisotropic type.

cylindrical_center *(array) [={0,0,0}]*
Array of the origin of a local cylindrical coordinate system. Used with cylindrical_anisotropic type and cylindrical_axis.

cylindrical_axis *(array) [={0,0,1}]*
Array of the longitudinal axis of a local cylindrical coordinate system. Used with cylindrical_anisotropic type and cylindrical_center.

multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the conductivity. If none, no scaling is performed.

turbulent_prandtl_number_type *(enumerated) [=constant]*
Used with turbulence models.

constant	User-specified turbulent Prandtl number. Requires *turbulent_prandtl_number*.
auto	Turbulent Prandtl number is computed based on local flow and material properties.

turbulent_prandtl_number or *turb_pr* *(real) >0 [=0.91]*
Value of the turbulent Prandtl number. Used when *turbulent_prandtl_number_type* = constant.

Description

This command specifies a thermal conductivity model for the temperature (energy) equation. This model is applicable to fluid, solid, and shell element sets.

CONDUCTIVITY_MODEL commands are referenced by MATERIAL_MODEL commands, which in turn are referenced by ELEMENT_SET commands:

```
CONDUCTIVITY_MODEL( "my conductivity model" ) {
   type                                = constant
   conductivity                        = 2.521e-2
 }
 MATERIAL_MODEL( "my material model" ) {
   conductivity_model                  = "my conductivity model"
   ...
 }
 ELEMENT_SET( "fluid elements" ) {
   material_model                      = "my material model"
   ...
}
```

The heat flux vector **q** is given by either:

$$q = \kappa \nabla T \tag{61}$$

where κ is an isotropic thermal conductivity, or by:

$$q = K \cdot \nabla T \tag{62}$$

where **K** is an anisotropic thermal conductivity.

A constant conductivity model defines a spatially constant isotropic thermal conductivity, as in the above example. A constant_anisotropic conductivity model is also spatially constant but varies in each direction. It requires that the six elements of the conductivity tensor be specified:

```
CONDUCTIVITY_MODEL( "my anisotropic conductivity model" ) {
    type                       = constant_anisotropic
    anisotropic_conductivity   = { 0.02, 0.03, 0.04, 0, 0, 0 }
}
```

The elements of the anisotropic conductivity tensor are always specified in the order k11, k22, k33, k12, k23, k31.

For cylindrical objects, for example, the battery packs, it is preferable to specify anisotropic conductivity in the local cylindrical system. When a cylinder is at an angle of 45 degrees (0.785 rad) to the x and z axis and the cylinder center is located at the origin of the coordinate, the elements of the anisotropic conductivity tensor are specified in the order krr, ktt, kzz, krt, ktz, krz, krr is for the radial conductivity 1.0 W/(m K); ktt is the azimuthal conductivity 50 W/(m K); kzz is the axial conductivity 50 W/(m K).

```
CONDUCTIVITY_MODEL( "MyCylindricalConductivity" ) {
    type                       = cylindrical_anisotropic
    cylindrical_center         = { 0.0,   0.0,   0.0; }
    cylindrical_axis           = { 0.785,   0.0,   0.785; }
    anisotropic_conductivity   = { 1.0, 50.0, 50.0, 0.0, 0.0, 0.0; }
```

```
}
```

A constant_prandtl_number conductivity model computes the isotropic thermal conductivity from the specific heat at constant pressure and the viscosity:

$$\kappa = \frac{C_p \mu}{P_r} \tag{63}$$

where C_p is the specific heat at constant pressure, specified by the `SPECIFIC_HEAT_MODEL` command; μ is the viscosity, specified by the `VISCOSITY_MODEL` command; and P_r is the Prandtl number, given by constant_prandtl_number. For example,

```
SPECIFIC_HEAT_MODEL( "air" ) {
    type                    = constant
    specific_heat           = 1005
}
VISCOSITY_MODEL( "air" ) {
    type                    = constant
    viscosity               = 1.781e-5
}
CONDUCTIVITY_MODEL( "air" ) {
    type = constant_prandtl_number
    prandtl_number          = 0.71
}
MATERIAL_MODEL( "air" ) {
    specific_heat_model     = "air"
    viscosity_model         = "air"
    conductivity_model      = "air"
}
```

yields a thermal conductivity of 2.521e-2.

A ramped conductivity model is the same as the constant one, except that the isotropic conductivity is ramped down from 1000 times the *conductivity* value at time step 1 to the *conductivity* value at time step 10. For example,

```
CONDUCTIVITY_MODEL( "ramped conductivity model" ) {
    type            = ramped
    conductivity    = 2.521e-2
}
```

starts at conductivity of 2.521e+1 at time step one and reduces the conductivity until time step 10, at which time a conductivity of 2.521e-2 is used.

Conductivity models of types piecewise_linear and cubic_spline may be used to define the isotropic conductivity as a function of a single independent variable. For example,

```
CONDUCTIVITY_MODEL( "curve fit conductivity model" ) {
    type                = piecewise_linear
    curve_fit_values    = { 273, 2.42e-2 ;
                            323, 2.76e-2 ;
                            373, 3.10e-2 ;
                            423, 3.44e-2 ; }
curve_fit_variable      = temperature
}
```

defines the conductivity as a function of temperature. The *curve_fit_values* parameter is a two-column array corresponding to the independent variable and the conductivity values. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, such as `conductivity.fit`:

```
273         2.42e-2
323         2.76e-2
373         3.10e-2
423         3.44e-2
```

and read by:

```
CONDUCTIVITY_MODEL( "curve fit conductivity model" ) {
    type                = piecewise_linear
    curve_fit_values    = Read( "conductivity.fit" )
    curve_fit_variable  = temperature
}
```

Conductivity models of types piecewise_linear_anisotropic and cubic_spline_anisotropic may be used to define the anisotropic conductivity as a function of a single independent variable. For example,

```
CONDUCTIVITY_MODEL( "curve fit anisotropic conductivity model" ) {
    type                              = piecewise_linear_anisotropic
    anisotropic_curve_fit_values      = { 273, 2.42e-2, 3.42e-2, 4.42e-2, 0, 0, 0 ;
                                          323, 2.76e-2, 3.76e-2, 4.76e-2, 0, 0, 0 ;
                                          373, 3.10e-2, 4.10e-2, 5.10e-2, 0, 0, 0 ;
                                          423, 3.44e-2, 4.44e-2, 5.44e-2, 0, 0, 0 ; }
    anisotropic_curve_fit_variable    = temperature
}
```

defines the anisotropic conductivity as a function of temperature. The *anisotropic_curve_fit_values* parameter is a seven-column array corresponding to the independent variable and the component values of the conductivity tensor. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *anisotropic_curve_fit_variable* falls outside of the curve fit limits.

The *anisotropic_curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, such as `anisotropic_conductivity.fit`:

```
273    2.42e-2    3.42e-2    4.42e-2    0    0    0
323    2.76e-2    3.76e-2    4.76e-2    0    0    0
373    3.10e-2    4.10e-2    5.10e-2    0    0    0
423    3.44e-2    4.44e-2    5.44e-2    0    0    0
```

and read by:

```
CONDUCTIVITY_MODEL( "curve fit anisotropic conductivity model" ) {
    type                              = piecewise_linear_anisotropic
    anisotropic_curve_fit_values      = Read( "anisotropic_conductivity.fit" )
    anisotropic_curve_fit_variable    = temperature
}
```

The sutherland type defines a temperature dependent conductivity model according to the following equation:

$$k = k_0 \left(\frac{T}{T_0} \right)^{3/2} \frac{T_0 + S}{T + S} \tag{64}$$

where k_0 is the *sutherland_reference_conductivity* at the specified *sutherland_reference_temperature* (T_0), T is the model temperature and S is the *sutherland_constant*.

For example,

```
VISCOSITY_MODEL( "Sutherland Model" ) {
type = sutherland
sutherland_reference_conductivity = 2.41e-2
sutherland_reference_temperature = 273.0
sutherland_constant = 194
}
```

Conductivity types user_function and user_function_anisotropic may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user defined functions.

For example, consider an isotropic conductivity model which is a linear function of temperature and species 1. Then the input command may be given by:

```
CONDUCTIVITY_MODEL( "UDF conductivity model" ) {
    type          = user_function
    user_function = "usrConductivityExample"
    user_values   = { 0.02,   # constant value
                      1.e-3,  # temperature scale
                      1.e-3 } # species scale
}
```

where the user-defined function "usrConductivityExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrConductivityExample ) ;                    /* function prototype
               */
Void usrConductivityExample (
    UdfHd    udfHd,                                          /* Opaque handle for
 accessing data  */
 Real*    outVec,                                          /* Output vector
           */
 Integer  nItems,                                          /* Number of elements
           */
 Integer  vecDim                                           /* = 1
           */
) {
    Integer  elem ;                                        /* an element counter
              */
    Real     cond0 ;                                       /* reference
 conductivity             */
    Real     sScale ;                                      /* species scale
              */
    Real     tScale ;                                      /* temperature scale
              */
```

```
    Real*    spec ;                                              /* species field
               */
    Real*    temp ;                                              /* temperature field
               */
    Real*    usrVals ;                                           /* user values
               */
udfCheckNumUsrVals( udfHd, 3 ) ;                                 /* check for error
               */
    usrVals  = udfGetUsrVals( udfHd ) ;                          /* get the user vals
               */
    cond0    = usrVals[0] ;                                      /* reference cond.
               */
    tScale   = usrVals[1] ;                                      /* temp. scale
               */
    sScale   = usrVals[2] ;                                      /* spec. scale
               */
    temp     = udfGetElmData( udfHd, UDF_ELM_TEMPERATURE ) ;     /* get temperature
               */
    spec     = udfGetElmData( udfHd, UDF_ELM_SPECIES ) ;         /* get species
               */
    for ( elem = 0 ; elem < nItems ; elem++ ) {
        outVec[elem] = cond0
                     + tScale * temp[elem]
                     + sScale * spec[elem] ;
    }
} /* end of usrConductivityExample()
               */
```

The dimension of the returned conductivity vector, *outVec*, is the number of elements. The same conductivity model but in anisotropic form can be defined by:

```
CONDUCTIVITY_MODEL( "UDF anisotropic conductivity model" ) {
   type                          = user_function_anisotropic
   anisotropic_user_function     = "usrAnisoConductivityExample"
   anisotropic_user_values       = { 0.02,   # constant value
                                     1.e-3,  # temperature scale
                                     1.e-3 } # species scale
}
```

where the user-defined function "usrAnisoConductivityExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrAnisoConductivityExample ) ;                                  /*
 function prototype                */
Void usrAnisoConductivityExample (
    UdfHd                        udfHd,                                         /*
 Opaque handle for accessing data */
 Real*                        outVec,                                       /*
 Output vector                     */
 Integer                      nItems,                                       /*
 Number of elements                */
 Integer                      vecDim                                        /* = 6
                              */
) {
    Integer                      elem ;                                         /*
 an element counter                */
    Real                         cond ;                                         /*
 conductivity                      */
    Real                         cond0 ;                                        /*
 reference conductivity            */
```

```
   Real                         sScale ;                                    /*
 species scale                    */
   Real*                        tScale ;                                    /*
 temperature scale                */
   Real*                        spec ;                                      /*
 species field                    */
   Real*                        temp ;                                      /*
 temperature field                */
   Real*                        usrVals ;                                   /*
 user values                      */
   udfCheckNumUsrVals( udfHd, 3 ) ;                                         /*
 check for error                  */
   usrVals                      udfGetUsrVals( udfHd ) ;                    /*
 get the user vals                */
 cond0                      = usrVals[0] ;                                 /*
 reference cond.                  */
   tScale                       = usrVals[1] ;                              /*
 temp. scale                      */
   sScale                       = usrVals[2] ;                              /*
 spec. scale                      */
   temp                         = udfGetElmData( udfHd, UDF_ELM_TEMPERATURE ) ;  /*
 get temperature                  */
   spec                         = udfGetElmData( udfHd, UDF_ELM_SPECIES ) ;      /*
 get species                      */
   for ( elem = 0 ; elem < nItems ; elem++ ) {
      cond = cond0 + tScale * temp[elem] + sScale * spec[elem] ;
      outVec[0*nItems+elem]    = cond ;
      outVec[1*nItems+elem]    = cond ;
      outVec[2*nItems+elem]    = cond ;
      outVec[3*nItems+elem]    = 0 ;
      outVec[4*nItems+elem]    = 0 ;
      outVec[5*nItems+elem]    = 0 ;
   }
}                                                                           /*
 end of usrAnisoConductivityExample() */
```

The dimension of the returned conductivity vector, *outVec*, is the number of elements times six.

The *multiplier_function* parameter may be used to uniformly scale both isotropic and anisotropic conductivity values. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file. For example, a ramped isotropic conductivity may be specified by:

```
CONDUCTIVITY_MODEL( "similar to ramped conductivity model" ) {
   type                     = constant
   conductivity             = 2.521e-2
   multiplier_function      = "ramped"
}
   MULTIPLIER_FUNCTION( "ramped" ) {
   type                     = piecewise_log_linear
   curve_fit_values         = { 1, 1000 ; 10, 1 }
   curve_fit_variable       = time_step
}
```

In areas of turbulent flow the molecular conductivity κ or $\mathbf{K}$ is replaced by the total conductivity $\kappa + \kappa_t$ or $\mathbf{K} + \kappa_t\mathbf{I}$(where $\mathbf{I}$ is the identity matrix) in the definition of the heat flux vector for the isotropic or anisotropic cases respectively. The turbulent conductivity κ_t is defined by

$$\kappa_t = \frac{C_p \mu_t}{Pr_t} \tag{65}$$

where μ_t is the turbulent viscosity, given by the turbulence model in use; and Pr_t is the turbulent Prandtl number, given by *turbulent_prandtl_number*.

CONTACT_ANGLE_MODEL

Specifies a contact angle model for free surfaces with surface tension.

Type

AcuSolve Command

Syntax

CONTACT_ANGLE_MODEL("name") {parameters...}

Qualifier

User-given name.

Parameters

type (enumerated) [=free]

Type of the contact angle.

free	Automatically computed. Used at inflow and outflow boundaries.
constant or const	Constant contact angle. Requires *contact_angle*.

contact_angle (real) >0 <180 [=90]

Constant value of the contact angle. Used with constant type.

multiplier_function (string) [=none]

User-given name of the multiplier function for scaling the contact angle. If none, no scaling is performed. Not used with free type.

Description

This command specifies a contact angle model for free surfaces with surface tension. See the SURFACE_TENSION_MODEL command. The contact angle is defined as the angle between the outward normal to the free surface and the outward normal to the side boundary. It may be any angle between zero and 180 degrees; it is small for a hydrophilic surface and large for a hydrophobic surface. The contact angle is a function of both fluids and the wall material, and must be determined experimentally. For example, a constant contact angle may be given by:

```
CONTACT_ANGLE_MODEL( "water/air @ smooth aluminum" ) {
   type                                = constant
   contact_angle                       = 90
}
```

If the side wall is not a solid wall but an inflow or outflow boundary, two strategies are available. If the boundary is far away from the region of interest, a constant contact angle of 90 degrees can be used. Otherwise, *type*=free is recommended:

```
CONTACT_ANGLE_MODEL( "water/air @ inflow" ) {
   type       = free
```

```
}
```

Here, the contact angle is not given but instead extracted from the solution in such a way as to make the slope of the free surface smooth near the boundary.

`CONTACT_ANGLE_MODEL` commands are referenced by `SIMPLE_BOUNDARY_CONDITION` and `FREE_SURFACE` commands, see the `SURFACE_TENSION_MODEL` command for examples.

Like flux boundary conditions, the contact angle is weakly enforced. This means that the solution may not obviously reflect the specified contact angle, especially if the meniscus is not well resolved.

The *`multiplier_function`* parameter may be used to uniformly scale the contact angle values. The value of this parameter refers to the user-given name of a `MULTIPLIER_FUNCTION` command in the input file. For example, a contact angle that ramps from 90 degrees to 120 degrees over the first 10 time steps may be specified by:

```
CONTACT_ANGLE_MODEL( "ramped contact angle model" ) {
   type                = constant
   contact_angle       = 90
   multiplier_function = "ramped"
   }
   MULTIPLIER_FUNCTION( "ramped" ) {
   type                = piecewise_linear
   curve_fit_values    = { 1, 90 ; 10, 120 }
   curve_fit_variable  = time_step
}
```

The contact angle is clipped so that it always remains strictly between zero and 180 degrees.

DENSITY_MODEL

Specifies a density model.

Type

AcuSolve Command

Syntax

```
DENSITY_MODEL("name") {parameters...}
```

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of the density model.

constant or const	Constant density. Requires *density*.
boussinesq	Boussinesq approximation. Requires *density*, *expansivity_type* and *reference_temperature*.
isentropic	Isentropic gas. Requires *density*, *specific_heat_ratio* and *reference_pressure*, and also *absolute_pressure_offset* from the EQUATION command.
ideal_gas	Ideal gas law. Requires *gas_constant*, and also *absolute_pressure_offset* and *absolute_temperature_offset* from the EQUATION command.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
user_function or user	User-defined function. Requires *user_function*, *user_values* and *user_strings*.

density or dens (real) >0 [=1]

Constant value of the density when used with constant and boussinesq types, and value of the reference density when used with isentropic type.

expansivity_type or expans_type (enumerated) [=constant]

Type of the expansivity function used for the Boussinesq approximation. Used with boussinesq type.

constant or const	Constant expansivity. Requires *expansivity*.

expansivity or *expans* *(real) >=0 [=1]*
Constant value of the expansivity. Used with constant expansivity type.

reference_temperature or *ref_temp* *(real) [=273.15]*
Reference temperature of the Boussinesq approximation. Used with boussinesq type.

reference_pressure or *ref_pres* *(real) [=0]*
Value of the reference pressure. Used with isentropic type.

specific_heat_ratio *(real) >=1 [=1.4]*
Value of the specific heat ratio. Used with isentropic type.

gas_constant *(real) [=287.058]*
Value of the gas constant. Used with *ideal_gas* type.

isothermal_compressibility *(real) >=0 [=0]*
Value of the isothermal compressibility.

curve_fit_values or *curve_values* *(array) [={0,0}]*
A two-column array of independent-variable/density data values. Used with piecewise_linear and cubic_spline types.

curve_fit_variable or curve_var (enumerated) [=temperature]
Independent variable of the curve fit. Used with piecewise_linear and cubic_spline types.

x_coordinate or xcrd	X-component of coordinates.
y_coordinate or ycrd	Y-component of coordinates.
z_coordinate or zcrd	Z-component of coordinates.
x_reference_coordinate or xrefcrd	X-component of reference coordinates.
y_reference_coordinate or yrefcrd	Y-component of reference coordinates.
z_reference_coordinate or zrefcrd	Z-component of reference coordinates.
pressure or pres	Pressure.
temperature or temp	Temperature.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.

species_7 or spec7 Species 7.

species_8 or spec8 Species 8.

species_9 or spec9 Species 9.

user_function or *user* *(string) [no default]*
Name of the user-defined function. Used with user_function type.

user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function type.

user_strings *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with user_function type.

Description

This command specifies a density model for element sets. All types of element sets, fluid, solid, and shell, require a density model.

DENSITY_MODEL commands are referenced by MATERIAL_MODEL commands, which in turn are referenced by ELEMENT_SET commands:

```
DENSITY_MODEL( "my density model" ) {
   type                                = constant
   density                             = 1.225
   isothermal_compressibility          = 0
 }
 MATERIAL_MODEL( "my material model" ) {
   density_model                       = "my density model"
   ...
 }
 ELEMENT_SET( "fluid elements" ) {
   material_model                      = "my material model"
   ...
}
```

A constant density model uses a constant density for the entire element set, as in the above example.

A density model of type boussinesq employs the Boussinesq approximation. The variation in density is assumed to be a linear function of temperature and to be relatively small. Hence, as a first level approximation, the variation of density is ignored for all terms except the body force term. The effect of the density variation on the body force is modeled by scaling the body force resulting from the GRAVITY, ROTATION_FORCE, and REFERENCE_FRAME commands by:

$$1 - \alpha_P (T - T_{\text{ref}}) \qquad (66)$$

where α_P is the thermal expansivity, given by *expansivity*; T is the temperature; and T_{ref} is the temperature at which the density equals the given constant *density* value, given by *reference_temperature*. For example,

```
DENSITY_MODEL( "air" ) {
   type                        = boussinesq
   density                     = 1.225
   expansivity_type            = constant
```

```
   expansivity                = 0.0034722
   reference_temperature      = 288
}
```

The Boussinesq model assumes that the problem contains a temperature field; see the EQUATION command. If the problem does not contain any temperature equation, then a boussinesq density model is internally converted to a constant density model.

An isentropic (constant entropy) gas may be modeled with an isentropic type. This is a full variable-density model where the variation in density is taken into account in all terms. For example,

```
DENSITY_MODEL( "isentropic air" ) {
   type                       = isentropic
   density                    = 1.225
   specific_heat_ratio        = 1.4
   reference_pressure         = 0
}
```

The density is given by:

$$\rho = \rho_0 \left(\frac{p + p_a}{p_0 + p_a} \right)^{1/\gamma} \tag{67}$$

where ρ is the density, given by this command; ρ_0 is the reference density, given by density; p is the pressure; p_0 is the reference pressure, given by *reference_pressure*; γ is the specific heat ratio, given by *specific_heat_ratio*; and p_a is the absolute pressure offset, given by *absolute_pressure_offset* from the EQUATION command.

The specific heat ratio for the ideal gas law is computed internally through Mayer's relation by

$$\gamma = \frac{C_p(T)}{(C_p(T) - R)} \tag{68}$$

where C_p is the specific heat capacity; T is the temperature; R is the gas constant.

A density model based on the ideal gas law is provided by the ideal_gas type. This is a full variable-density model where the variation in density is taken into account in all terms. For example,

```
DENSITY_MODEL( "ideal gas air" ) {
   type               = ideal_gas
   gas_constant       = 287.058
}
```

The density is given by:

$$\rho = \frac{p + p_a}{R(T + T_a)} \tag{69}$$

where ρ is the density, given by this command; R is the gas constant, given by *gas_constant*; p is the pressure; T is the temperature; and p_a and T_a are the absolute pressure and temperature offsets, given by *absolute_pressure_offset* and *absolute_temperature_offset* from the EQUATION command. This model requires the availability of the temperature field; an error will be

issued otherwise. When modeling compressible flow the use of *absolute_pressure_offset* and *absolute_temperature_offset* is not recommended. Instead, you should define boundary and initial conditions in terms of absolute values.

Pseudo compressibility may be modeled by adding the isothermal compressibility term to the continuity equation:

$$\beta_T \rho \frac{\partial p}{\partial t} + \frac{\partial \rho}{\partial t} + \nabla \bullet \rho \boldsymbol{u} = 0 \qquad (70)$$

where β_T is the isothermal compressibility, given by *isothermal_compressibility*; ρ is density; p is pressure; t is time; and **u** is the velocity vector. Isothermal compressibility is defined by:

$$\beta_T = \frac{1}{\rho}\frac{\partial \rho}{\partial p} \qquad (71)$$

Pseudo compressibility is often used along with constant type to model wind-generated noise. Pseudo compressibility should not be activated for *flow* = compressible_navier_stokes.

Full variable-density models are also supported by the piecewise_linear, cubic_spline, and user_function types. As opposed to the Boussinesq type, the density for these types is variable in all terms.

Density models of types piecewise_linear and cubic_spline may be used to define density as a function of a single independent variable. For example,

```
DENSITY_MODEL( "air" ) {
   type                      = piecewise_linear
   curve_fit_values          = { 60, 1.05943 ;
                                 160, 0.814836 ;
                                 260, 0.662024 ;
                                 360, 0.557587 ; }
   curve_fit_variable      = temperature
}
```

gives a density model by defining density as a function of temperature. Use caution when defining density as a function of pressure; this can easily lead to instability.

The *curve_fit_values* is a two-column array corresponding to the independent variable, temperature, in this case, and density. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, say `density.mks`:

```
60      1.05943
160     0.814836
260     0.662024
360     0.557587
```

and read by:

```
DENSITY_MODEL( "air" ) {
   type                    = piecewise_linear
   curve_fit_values        = Read ( "density.mks" )
   curve_fit_variable      = temperature
```

```
}
```

A density of type user_function may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider a version of the Boussinesq density model discussed above. Here, the density variation appears in all terms but you "clip" the density to that corresponding to 10 degrees above and below the reference temperature. The derivative of density with respect to temperature is also returned; this Jacobian is needed to form the appropriate contribution to the left-hand side matrix. If it is not returned, no error is issued but the Jacobian is assumed to be zero and nonlinear convergence may suffer.

Note: Only Jacobians with respect to temperature are supported currently.

The density may also be a function of pressure and species, but extreme caution must be taken since Jacobians with respect to these variables are not supported. The input command may be given by:

```
DENSITY_MODEL( "thermal air" ) {
   type                = user_function
   user_function       = "usrDensity"
   user_values         = { 1.225,          # reference density
                           288,            # reference temperature
                         0.0034722 }       # expansivity
}
```

where the user-defined function "usrDensity" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrDensity ) ;                                          /* function
 prototype                */
Void usrDensity (
    UdfHd                  udfHd,                                      /* Opaque handle
 for accessing data */
    Real*                  outVec,                                     /* Output vector
                  */
    Integer                nItems,                                     /* Number of
 elements                */
    Integer                vecDim                                      /* = 1
                  */
) {
    Real                   expn ;                                      /* expansivity
                  */
    Real                   rho0 ;                                      /* reference
 density                 */
    Real                   rhoMin ;                                    /* minimum
 density                   */
    Real                   rhoMax ;                                    /* maximum
 density                   */
    Real                   temp0 ;                                     /* reference
 temperature             */
    Integer                i ;                                         /* a running
 index                   */
    Real*                  densJacTemp ;                               /* partial
 dens / partial temp       */
    Real*                  temp ;                                      /* temperature
                  */
```

```
    Real*                  usrVals ;                                    /* user supplied
 values             */

udfCheckNumUsrVals( udfHd, 3 ) ;                                        /* check for
 error                  */
    usrVals               = udfGetUsrVals( udfHd ) ;
    rho0                  = usrVals[0] ;
    temp0                 = usrVals[1] ;
    expn                  = usrVals[2] ;
    rhoMin                = rho0 * (1 - 10 * expn ) ;
    rhoMax                = rho0 * (1 + 10 * expn ) ;
    temp                  = udfGetElmData( udfHd, UDF_ELM_TEMPERATURE ) ;
    densJacTemp           = udfGetElmJac( udfHd, UDF_ELM_JAC_TEMPERATURE )
    for ( i = 0 ; i < nItems ; i++ ) {
       densJacTemp[i]     = -rho0 * expn ;
       outVec[i]          = rho0 * ( 1 + expn * (temp0 - temp[i]) ) ;
       if ( outVec[i] < rhoMin ) {
            densJacTemp[i] = 0 ;
            outVec[i]      = rhoMin ;
       if ( outVec[i] > rhoMax ) {
            densJacTemp[i] = 0 ;
            outVec[i]      = rhoMax ;
      }
}                                                                       /* end of
 usrDensity()             */
```

The dimensions of the returned density vector, *outVec*, and the Jacobian vector, *densJacTemp*, are the number of elements.

DIFFUSIVITY_MODEL

Specifies a diffusivity model for the species transport equations.

Type

AcuSolve Command

Syntax

```
CONDUCTIVITY_MODEL("name") {parameters...}
```

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of diffusivity.

constant or const	Constant diffusivity. Requires *diffusivity*.
ramped or ramp	Ramp the diffusivity to a constant value. Requires *diffusivity*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
user_function or user	User-defined function. Requires *user_function*, *user_values* and *user_strings*.
gas_kinetic	Gas kinetic diffusivity model. Requires GAS_KINETIC_MODEL.

diffusivity or *diff (real) >=0 [=0]*

Constant value of diffusivity. Used with constant and ramped types.

curve_fit_values or *curve_values (array) [={0,0}]*

A two-column array of independent-variable/diffusivity data values. Used with piecewise_linear and cubic_spline types.

curve_fit_variable or *curve_var (enumerated) [=temperature]*

Independent variable of the curve fit. Used with piecewise_linear and cubic_spline types.

x_coordinate or xcrd	X-component of coordinates.
y_coordinate or ycrd	Y-component of coordinates.
z_coordinate or zcrd	Z-component of coordinates.

x_reference_coordinate or xrefcrd	X-component of reference coordinates.
y_reference_coordinate or yrefcrd	Y-component of reference coordinates.
z_reference_coordinate or zrefcrd	Z-component of reference coordinates.
temperature or temp	Temperature.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.
species_7 or spec7	Species 7.
species_8 or spec8	Species 8.
species_9 or spec9	Species 9.

`user_function` or `user` *(string) [no default]*
: Name of the user-defined function. Used with user_function type.

`user_values` *(array) [={}]*
: Array of values to be passed to the user-defined function. Used with user_function type.

`user_strings` *(list) [={}]*
: Array of strings to be passed to the user-defined function. Used with user_function type.

`multiplier_function` *(string) [=none]*
: User-given name of the multiplier function for scaling the viscosity. If none, no scaling is performed.

`turbulent_schmidt_number` or `turb_sc` *(real) >0 [=0.91]*
: Value of the turbulent Schmidt number. Used with turbulence models.

Description

This command specifies a diffusivity model for the species transport equations. This model is only applicable to fluid element sets.

`DIFFUSIVITY_MODEL` commands are referenced by `MATERIAL_MODEL` commands, which in turn are referenced by `ELEMENT_SET` commands:

```
DIFFUSIVITY_MODEL( "my diffusivity model" ) {
   type                                = constant
```

```
   diffusivity                          = 1
 }
 MATERIAL_MODEL( "my material model" ) {
   diffusivity_1_model                  = "my diffusivity model"
   ...
 }
 ELEMENT_SET( "fluid elements" ) {
   material_model                       = "my material model"
   ...
}
```

The diffusion flux vector Ψ is given by:

$$\Psi = d\nabla\varphi \tag{72}$$

where d is the diffusivity for species φ

A constant diffusivity model applies a spatially constant diffusivity, as in the above example.

A ramped diffusivity model is the same as the constant one, except that the diffusivity is ramped down from 1000 times the *diffusivity* value at time step one to the *diffusivity* value at time step ten. For example,

```
DIFFUSIVITY_MODEL( "ramped diffusivity model" ) {
   type              = ramped
   diffusivity       = 1
}
```

starts at diffusivity of 1000 at time step one and reduces the diffusivity until time step 10, at which time a diffusivity of one is used.

Diffusivity models of types piecewise_linear and cubic_spline may be used to define diffusivity as a function of a single independent variable. For example,

```
DIFFUSIVITY_MODEL( "curve fit diffusivity model" ) {
   type                 = piecewise_linear
   curve_fit_values     = { 273, 1.0 ;
                            323, 1.1 ;
                            373, 1.3 ;
                            423, 1.6 ;
   curve_fit_variable   = temperature
}
```

defines diffusivity as a function of temperature. In this case, the problem must contain a temperature equation; see the EQUATION command. The *curve_fit_values* parameter is a two-column array corresponding to the independent variable and the diffusivity values. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, such as diffusivity.fit:

```
273      1.0
323      1.1
373      1.3
423      1.6
```

and read by:

```
DIFFUSIVITY_MODEL( "curve fit diffusivity model" ) {
   type                    = piecewise_linear
   curve_fit_values        = Read( "diffusivity.fit"
   curve_fit_variable      = temperature
}
```

A diffusivity of type user_function may be used to model more complex behaviors; see the *AcuSolve User- Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider a diffusivity model which is a linear function of temperature and species one. Then the input command may be given by:

```
DIFFUSIVITY_MODEL( "UDF diffusivity model" ) {
   type                 = user_function
   user_function        = "usrDiffusivityExample"
   user_values          = { 1.0,  # constant value
                            0.1,  # temperature scale
                            0.2 } # species scale
}
```

where the user-defined function "usrDiffusivityExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrDiffusivityExample ) ;                    /* function prototype
             */
Void usrDiffusivityExample (
    UdfHd     udfHd,                                        /* Opaque handle for
 accessing data */
 Real*     outVec,                                         /* Output vector
          */
 Integer   nItems,                                         /* Number of elements
          */
 Integer   vecDim                                          /* = 1
          */
) {
    Integer   elem ;                                       /* an element counter
            */
    Real      diff0 ;                                      /* reference
 diffusivity            */
    Real      sScale ;                                     /* species scale
            */
    Real      tScale ;                                     /* temperature scale
            */
    Real*     spec ;                                       /* species field
            */
    Real*     temp ;                                       /* temperature field
            */
    Real*     usrVals ;                                    /* user values
            */
udfCheckNumUsrVals( udfHd, 3 ) ;                           /* check for error
            */
    usrVals   = udfGetUsrVals( udfHd ) ;                   /* get the user vals
            */
    diff0     = usrVals[0] ;                               /* reference diff.
            */
    tScale    = usrVals[1] ;                               /* temp. scale
            */
```

```
    sScale    = usrVals[2] ;                                  /* spec. scale
              */
    temp      = udfGetElmData( udfHd, UDF_ELM_TEMPERATURE ) ; /* get temperature
              */
    spec      = udfGetElmData( udfHd, UDF_ELM_SPECIES ) ;     /* get species
              */
    for ( elem = 0 ; elem < nItems ; elem++ ) {
       outVec[elem] = diff0
                    + tScale * temp[elem]
                    + sScale * spec[elem] ;
    }
} /* end of usrDiffusivityExample() */
```

The dimension of the returned diffusivity vector, *outVec*, is the number of elements.

The *multiplier_function* parameter may be used to uniformly scale the diffusivity values. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file. For example, a ramped diffusivity may be specified by:

```
DIFFUSIVITY_MODEL( "similar to ramped diffusivity model" ) {
   type                = constant
   diffusivity         = 1
   multiplier_function = "ramped"
   }
   MULTIPLIER_FUNCTION( "ramped" ) {
   type                = piecewise_log_linear
   curve_fit_values    = { 1, 1000 ; 10, 1 }
   curve_fit_variable  = time_step
}
```

In areas of turbulent flow the "molecular" diffusivity d is replaced by the total diffusivity $d+d_t$ in the diffusion flux vector. The turbulent diffusivity d_t is defined by

$$d_t = \frac{\mu_t}{\rho Sc_t} \tag{73}$$

where μ_t is the turbulent viscosity, given by the turbulence model in use; ρ is the density; and SC_t is the turbulent Schmidt number, given by *turbulent_schmidt_number*.

ELECTRICAL_RESISTIVITY_MODEL

Specifies an electrical resistivity model for the charge conservation equation. The electrical resistivity represents the ability of a material to resist the flow of current. Electrical resistivity model applies to solid materials only.

Type

AcuSolve Command

Syntax

ELECTRICAL_RESISTVITY_MODEL("name"){parameters...}

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of the electrical resistivity model.

constant or const	Constant electrical resistivity. Requires *electrical_resistivity*.
linear_temperature	Linear temperature dependent resistivity.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
user_function or user	User-defined function. Requires *user_function*, *user_values* and *user_strings*.
contact_resistance	Contact resistance between two connected materials. Requires *electrical_contact_resistance*.

electrical_resistivity (real) [=1.72e-8]

Constant value of electrical resistivity. When *type* is constant.

linear_temperature_reference_temperature (real) [=293.15]

Reference temperature for linear_temperature resistivity type.

linear_temperature_reference_temperature_resistivity (real) [=1.72e-8]

Resistivity of the material at the reference temperature for linear_temperature resistivity type.

linear_temperature_temperature_coefficient (real) [=0.00393]

Relative change of resistivity for a given change in temperature for linear_temperature resistivity type.

curve_fit_values or *curve_values* *(array) [={0,0}]*
: A two-column array of independent-variable/isotropic electrical resistivity data values. Used with piecewise_linear and cubic_spline types.

curve_fit_variable or *curve_var* *(enumerated) [=temperature]*
: Independent variable of the curve fit for isotropic electrical resistivity. Used with piecewise_linear and cubic_spline types.

user_function or *user* *(string) [no default]*
: Name of the user-defined function. Used with user_function type.

user_values *(array) [={}]*
: Array of values to be passed to the user-defined function. Used with user_function type.

user_strings *(list) [={}]*
: Array of strings to be passed to the user-defined function. Used with user_function type.

electrical_contact_resistance *(real) [=1.0e-6]*
: electrical_contact resistance between two connected materials. Used with contact_resistance type. Currently supported only for *electric_potential* = battery_joule_heating.

Description

The resistivity model is used for both the charge conservation equation and for the coupling of charge conservation to the energy equation. A brief description of both models in the context of electrical resistivity is given below.

Conservation of charge is a fundamental physical principle and is given by:

$$\nabla \cdot \mathbf{j} = 0 \tag{74}$$

Where $\mathbf{j}$ is the current density vector (units: A/m^2). Typically, it is assumed that the current density is proportional to the electric field, where the proportionality constant is the electrical conductivity (σ) of the material. The current density can then be expressed as:

$$\mathbf{j} = -\sigma \nabla \Phi \tag{75}$$

where Φ (units: V) is the electric potential field. The inverse of conductivity σ is the electrical resistivity ρ_E of the material.

To couple the electrical model into the thermal simulation in AcuSolve, a source term (S) is introduced into the energy equation. The source term is based on Joule's first law which says that the heat generated per unit volume is equal to the product of current density and the electric field ($\mathbf{E} = -\nabla\phi$). For example:

$$S = \mathbf{j} \cdot \mathbf{E} = \sigma |\nabla \phi|^2 \tag{76}$$

The available resistivity model is set in the ELECTRICAL_RESISTIVITY_MODEL command and is referenced by MATERIAL_MODEL commands. For example:

```
ELECTRICAL_RESISTIVITY_MODEL( "my resistivity model" ) {
    type = constant
    electrical_resistivity = 2.9e-8
}
MATERIAL_MODEL( "my material model" ) {
```

```
    electrical_resistivity_model = "my resistivity model"
}
```

The simplest case is *type* = constant. This applies a constant resistivity in both the conservation of charge equation and the source term in the energy equation. An example of a constant electrical resistivity value is shown above.

The second type is a linear temperature dependent resistivity (*type*=linear_temperature). For this model the resistivity is a linear function of temperature given by:

$$\rho_E(T) = \rho_{E0}\left(1 + \alpha(T - T_{REF}\right) \tag{77}$$

Where ρ_{E0} is the reference temperature resistivity, α the temperature coefficient, T the local temperature and T_{REF} the reference temperature. The temperature coefficient describes the relative change of resistivity for a given change in temperature.

In the input file a linear temperature dependent resistivity would be defined as follows:

```
ELECTRICAL_RESISTIVITY_MODEL( "my resistivity model" ) {
    type = linear_temperature
    linear_temperature_reference_temperature = 293.15
    linear_temperature_reference_temperature_resistivity = 1.754e-8
    linear_temperature_temperature_coefficient = 0.0039
}
```

Electrical resistivity models of types piecewise_linear and cubic_spline may be used to define electrical resistivity as a function of a single independent variable. These types of electrical resistivity models are consistent with other material models. Currently the only *curve_fit_variable* supported is temperature.

An electrical resistivity model of type user_function may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

Electrical Contact Resistance

Electrical contact resistance occurs at the interface between connected electrical components, for example, welded joints. Contact resistance creates ohmic losses that lead to thermal hotspots and other undesirable thermal properties.

Contact resistance is enabled through the material model by setting the *type* = contact_resistance in the ELECTRICAL_RESISTIVITY_MODEL and linking this material to a THERMAL_SHELL. The electrical resistance is then given in $\Omega \cdot m^2$. Currently, this is only supported by *electric_potential* = battery_joule_heating.

An example contact resistance input:

```
MATERIAL_MODEL("weld") {
 type = solid
 ...
 electrical_resistivity_model = "weld"
}
ELECTRICAL_RESISTIVITY_MODEL("weld") {
 type = contact_resistance
 electrical_contact_resistance = 1e2
}
```

EMISSIVITY_MODEL

Specifies an emissivity model for the radiation equation.

Type

AcuSolve Command

Syntax

EMISSIVITY_MODEL("name") {parameters...}

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of the emissivity.

constant or const	Constant emissivity. Requires *emissivity*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
user_function or user	User-defined function. Requires *user_function*, *user_values* and *user_strings*.

emissivity or *emis (real) >0 <=1 [=1]*

Constant value of the emissivity. Used with constant type.

diffused_fraction (real) [= 1.0]

To enable specular reflection a RADIATION_SURFACE must be defined and the diffused fraction, a, and emissivity, ε, must both be less than one.

curve_fit_values or *curve_values (array) [={0,0}]*

A two-column array of independent-variable/emissivity data values. Used with piecewise_linear and cubic_spline types.

curve_fit_variable or *curve_var (enumerated) [=temperature]*

Independent variable of the curve fit. Used with piecewise_linear and cubic_spline types.

temperature or temp	Temperature.

user_function or *user (string) [no default]*

Name of the user-defined function. Used with user_function type.

user_values (array) [={}]

Array of values to be passed to the user-defined function. Used with user_function type.

user_strings (list) [={}]
Array of strings to be passed to the user-defined function. Used with user_function type.

multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the viscosity. If none, no scaling is performed.

Description

This command specifies an ideal grey-surface emissivity model for the radiation equation. This model is only applicable to radiation surfaces, and is not used for the *radiation_heat_flux* variable in the ELEMENT_BOUNDARY_CONDITION command.

EMISSIVITY_MODEL commands are referenced by RADIATION_SURFACE commands:

```
EMISSIVITY_MODEL( "my emissivity model" ) {
   type                                = constant
   emissivity                          = 0.5
}
RADIATION_SURFACE( "hot wall" ) {
 emmissivity_mdoel                 = "my emmissivity model"
 ...
}
```

The emissivity is the factor ε in the Stefan-Boltzmann law for the total emissive power of an ideal grey surface:

$$W = \varepsilon\sigma(T + T_{off})^4 \qquad (78)$$

where σ is the Stefan-Boltzmann constant, given by the *Stefan_boltzmann_constant* parameter of the RADIATION command; T is the temperature; and T_{off} is the offset to convert to an absolute temperature, given by the *absolute_temperature_offset* parameter of the EQUATION command.

A constant emissivity model applies a spatially constant emissivity, as in the above example.

Emissivity models of types piecewise_linear and cubic_spline may be used to define emissivity as a function of a single independent variable. For example,

```
EMISSIVITY_MODEL( "curve fit emissivity model" ) {
   type                = piecewise_linear
   curve_fit_values    = { 273, 0.2 ; 323, 0.2 ;    373, 0.3 ;    423, 0.4 ; }
   curve_fit_variable = temperature
}
```

defines emissivity as a function of temperature. In general, the problem must contain the variable defined by *curve_fit_variable*; this is not an issue here since radiation problems always contain temperature. The *curve_fit_values* parameter is a two-column array corresponding to the independent variable and the emissivity values. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, such as emissivity.fit:

```
273      0.2
```

```
323      0.2
373      0.3
423      0.3
```

and read by:

```
EMISSIVITY_MODEL( "curve fit emissivity model" ) {
   type               = piecewise_linear
   curve_fit_values   = Read( "emissivity.fit" )
   curve_fit_variable = temperature
}
```

An emissivity of type *user_function* may be used to model more complex behaviors; see the *AcuSolve User- Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider an emissivity that is equal to 0.5 between temperatures of 300 and 400, and 0.3 otherwise. The input command may be given by:

```
EMISSIVITY_MODEL( "UDF emissivity model" ) {
   type          = user_function
   user_function = "usrEmissivityExample"
   user_values   = { 300,    # lower temp. limit of band
                     400,    # upper temp. limit of band
                       0.3,  # default emissivity
                       0.5 }   # emissivity in temperature band
}
```

where the user-defined function "usrEmissivityExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrEmissivityExample ) ;  /* function prototype */
Void usrEmissivityExample (
    UdfHd    udfHd,                      /* Opaque handle for accessing data */
 Real*    outVec,                     /* Output vector */
 Integer  nItems,                     /* Number of elements */
 Integer  vecDim                      /* = 1 */
) {
    Integer  elem ;                      /* an element counter */
    Real     temp0 ;                     /* lower temp. limit of band */
    Real     temp1 ;                     /* upper temp. limit of band */
    Real     eDef ;                      /* default emissivity */
    Real     eBand ;                     /* emissivity in temperature band */
    Real*    temp ;                      /* temperature */
    Real*    usrVals ;                   /* user values */
udfCheckNumUsrVals( udfHd, 4 ) ;         /* check for error */
    usrVals  = udfGetUsrVals( udfHd ) ;  /* get the user vals */
    temp0    = usrVals[0] ;              /* lower temp  */
    temp1    = usrVals[1] ;              /* upper temp */
    eDef     = usrVals[2] ;              /* default emissivity */
    eBand    = usrVals[3] ;              /* band emissivity */
temp            = udfGetRsfData( udfHd, UDF_RSF_TEMPERATURE ) ;  /* get the temp. */
for ( elem = 0 ; elem < nItems ; elem++ ) {
    if ( temp[elem] >= temp0 && temp[elem] <= temp1 ) {
       outVec[elem] = eBand ;
    } else {
       outVec[elem] = eDef ;
    }
}
} /* end of usrEmissivityExample() */
```

The dimension of the returned emissivity vector, *outVec*, is the number of elements.

The *multiplier_function* parameter may be used to uniformly scale the emissivity values. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file. For example, a ramped emissivity may be specified by:

```
EMISSIVITY_MODEL( "ramped emissivity model" ) {
   type                = constant
   emissivity          = 0.4
   multiplier_function = "ramped"
   }
   MULTIPLIER_FUNCTION( "ramped" ) {
   type               = piecewise_linear
   curve_fit_values   = { 1, 0.1 ; 10, 1 }
   curve_fit_variable = time_step
}
```

GAS_KINETIC_MODEL

Specifies a gas kinetic model for the advective_diffusive multifield type.

Type

AcuSolve Command

Syntax

GAS_KINETIC_MODEL("name") {parameters...}

Qualifier

User-given name.

Parameters

geometrical_configuration_index (integer) >=0 <=2 [=1]
: An integer value indicating the type of molecular configuration. If the value is 0, the molecule is a single atom. If it is 1 the molecule is linear. Finally, if it is 2, the molecule is nonlinear.

Lennard_Jones_potential_well_depth (real) [=97.530]
: The Lennard-Jones potential well depth of the gas in Kelvin.

Lennard_Jones_collision_diameter (real) [=3.621]
: The Lennard-Jones collision diameter of the gas in Angstroms.

dipole_moment (real) [=0.0]
: The dipole moment of the gas in Debye.

Polarizability (real) [=1.760]
: The polarizability in cubic Angstroms.

rotational_relaxation_collision_number (real) [=4.0]
: The rotational relaxation collision number at 298 Kelvin.

molecular_weight (real) [=0.028013]
: Molecular weight of the gas.

Description

This command sets kinetic model molecular parameters for each gas. Gas kinetic model is used to calculate viscosity, conductivity, and mass diffusion coefficients for each model. This model should be referred to in the MATERIAL_MODEL of the gas. Below is an example to set parameters for Nitrogen:

```
GAS_KINETIC_MODEL( "N2" ) {

    geometrical_configuration_index     = 1
    molecular_weight                    = 0.028013
    Lennard_Jones_potential_well_depth  = 97.530
    Lennard_Jones_collision_diameter    = 3.621
    dipole_moment                       = 0.0
    polarizability                      = 1.760
    rotational_relaxation_collision_number = 4.000
}
```

```
VISCOSITY_MODEL( "N2" ) {
    type                                = gas_kinetic
}

CONDUCTIVITY_MODEL( "N2" ) {
    type                                = gas_kinetic
    turbulent_prandtl_number            = 0.91
}
```

If the species_transport equation is getting solved, gas kinetic diffusivity can be set as:

```
DIFFUSIVITY_MODEL( "N2" ) {
    type                                = gas_kinetic
}
```

If the multi_field equation type advective-diffusive is getting solved, gas kinetic diffusivity should be set using `FIELD_INTERACTION_MODEL`:

```
FIELD_INTERACTION_MODEL( "Mixture" ) {
    type                                = advective_diffusive
    diffusion_matrix_type               = gas_kinetic
}
```

FIELD

Associates a material model to a field.

Type

AcuSolve Command

Syntax

`FIELD ("name") {parameters...}`

Qualifier

User-given name.

Parameters

`material_model` *(string) [none]*

Name of a *`material_model`* entry.

Description

This command specifies the material model associated with the given field. A `FIELD` entry is required for all fields used in the simulation, that is, those listed in the `EQUATION::FIELDS` entry.

```
FIELD ( "water" ) {
    material_model        = "water model"
}
```

FIELD_INTERACTION_MODEL

Specifies an interaction model between fields.

Type

AcuSolve Command

Syntax

```
FIELD_INTERACTION_MODEL("name") {parameters...}
```

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of the field interaction model.

none	No field interaction model.
levelset	Use levelset technology to ensure immiscibility between the fields listed in *fields_1* and *fields_2* entries. Requires levelset in the *multi_field* entry of the EQUATION command.
levelset_bfecc	Use levelset technology with the BFECC advection scheme. Requires levelset_bfecc in the *multi_field* entry of the EQUATION command.
algebraic_eulerian	Use the Algebraic Eulerian model to simulate momentum exchange between a carrier field and a dispersed field or conduct one-way transient AcuSolve-EDEM coupling. Requires algebraic_eulerian in the *multi_field* entry of the EQUATION command. For AcuSolve-EDEM coupling, the *edem_coupling* flag in the EQUATION command should be set to on.
eulerian _eulerian	Use the Eulerian Eulerian model to simulate momentum exchange between a carrier field and a dispersed field or conduct two-way AcuSolve-EDEM coupling. Requires eulerian _eulerian in the *multi_field* entry of the EQUATION command. For AcuSolve-EDEM coupling, the *edem_coupling* flag in the EQUATION command should be set to on.
advective_diffusive	Use the advective_diffusive method to model momentum exchange between *Humid_Air* and *Air*. Requires advective_diffusive in the *multi_field* entry of the EQUATION command.

fields_1 (string) [=none]
List of names of fields. Currently only accepts one field name. The name must refer to a defined field entry. Only used for field interaction with *type* = levelset or levelset_bfecc or advective_diffusive.

fields_2 (string) [=none]
List of names of fields. Currently only accepts one field name. The name must refer to a defined field entry. Only used for field interaction with *type* = levelset or levelset_bfecc or advective_diffusive.

levelset_redistancing (enumerated) [=implicit]
Method used to perform levelset redistancing at the end of each time step. Only used for field interaction with *type* = levelset or levelset_bfecc.

implicit	An implicit integration method is used to solve the levelset redistancing equation. This approach is more robust but also usually more CPU time expensive.
explicit	An explicit integration method is used to solve the levelset redistancing equation. This approach is usually faster than the implicit method but can be ill-suited for some cases.

fields (list) [={}]
Name of the fields for *advective_diffusive*. Used by *diffusion_matrix* to get array indices. For the humid air model two field names are accepted. The name must refer to a defined field entry.

algebraic_slip_model (boolean) [=on]
Flag specifying whether to calculate momentum exchange between fields by employing the algebraic slip model. If off, carrier and dispersed fields are treated as homogeneous fields. Only used for field interaction with *type* = algebraic_eulerian.

phase_change_type (enumerated) [=none]
Type of the phase change for algebraic_eulerian.

none	No phase change modeling.
wall_boiling	In this model, nucleate boiling by the heated wall is modeled with RPI wall boiling model. In the RPI model, it is assumed that heat flux on the wall gets partitioned to two parts corresponding to sub-cooled liquid and saturated evaporating vapor. The sub-cooled liquid is also partitioned to convective and quenching heat flux: $$Q_{wall} = Q_{convectivr} + Q_{quenching} + Q_{evaporative} \quad (84)$$ where convective heat is calculated using standard turbulence wall function. Separate models are used for quenching and evaporative parts.

nucleating_site_density_type (enumerated)[=Lemmert_Chawla]
Type of nucleating site density used for calculating evaporative heat flux $Q_{evaporative}$.

Lemmert_Chawla In this model, the following model is used to calculate nucleation site density:

$$N = \left(C_m\left(T_{wall} - T_{saturation}\right)\right)^n \quad (85)$$

C_m and C_n are user changeable values and can be set in *nucleating_site_density_coefficient* and *nucleating_site_density_power_index* parameters. Vapor phase *latent_heat_temperature* is used to set $T_{saturation}$.

user_function User-defined function. Requires *nucleating_site_density_user_function*, *nucleating_site_density_user_values* and *nucleating_site_density_user_strings*.

nucleating_site_density_coefficient *(real) [=210]*
Coefficient of nucleating site density C_m. Use with *nucleating_site_density_type* = Lemmert_Chawla.

nucleating_site_density_power_index *(real) [=1.805]*
Power index of nucleating site density C_n. Use with *nucleating_site_density_type* = Lemmert_Chawla.

nucleating_site_density_user_function *(string) [no default]*
Name of the user-defined function. Used with *nucleating_site_density_type* = user_function.

nucleating_site_density_user_values *(array)[={}]*
Array of values to be passed to the user-defined function. Used with *nucleating_site_density_type* = user_function.

nucleating_site_density_user_strings *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with *nucleating_site_density_type* = user_function.

quenching_heat_transfter_coefficient_type *(enumerated) [=Del_Valle_Kenning]*
Type of quenching heat transfer coefficient used for calculating quenching heat flux $Q_{quenching}$.

Del_Valle_Kenning In this model, the following model is used to calculate nucleation site density:

$$2\kappa_l f_v \sqrt{\frac{t_w \rho_l C p_l}{\pi \kappa_l}} \quad (86)$$

κ_l, ρ_l and Cp_l are liquid phase thermal conductivity, density and specific heat, f_v is vapor production frequency and t_w is bubble wait time on nucleating site.

user_function User-defined function. Requires *quenching_heat_transfter_coefficient _user_function*, *quenching_heat_transfter_coefficient _user_values* and *quenching_heat_transfter_coefficient_user_strings*.

quenching_heat_transfter_coefficient _user_function (string) [no default]
Name of the user-defined function. Used with *quenching_heat_transfter_coefficient_type* = user_function.

quenching_heat_transfter_coefficient _user_values (array) [={}]
Array of values to be passed to the user-defined function. Used with *quenching_heat_transfter_coefficient_type* = user_function.

quenching_heat_transfter_coefficient _user_strings (list) [={}]
Array of strings to be passed to the user-defined function. Used with *quenching_heat_transfter_coefficient_type* = user_function.

bubble_waiting_time_fraction (real) [=0.1]
Bubble fraction C_w used to model bubble wait time as $t_w = \frac{C_w}{f_v}$. Use with *quenching_heat_transfter_coefficient _type* = Del_Valle_Kenning.

bubble_departure_diameter_type (enumerated) [=Tolubinski_Kostanchuk]
Type of bubble departing diameter type used for calculating evaporating heat flux $Q_{evaporating}$.

Tolubinski_Kostanchuk — In this model, the following model is used to calculate departing bubble diameter:

$$\min\left(D_{ref}\left(-\frac{T_{saturation}-T_{bulk}}{\Delta T_{scal}}\right), D_{max}\right) \quad (87)$$

D_{ref}, D_{max} and T_{scal} are reference diameter, maximum diameter and scaling temperature. You can specify these parameters. Vapor phase *latent_heat_temperature* is used to set $T_{saturation}$, *liquid_subcooling_temperature_type* is used to calculate T_{bulk} and *bubble_departure_scaling_temperature* is used to set T_{scal}.

user_function — User-defined function. Requires *bubble_departure_diameter _user_function*, *bubble_departure_diameter _user_values* and *bubble_departure_diameter_user_strings*.

bubble_departure_reference_diameter (real) [=0.0006]
Departing bubble reference diameter, D_{ref}. Use with *bubble_departure_diameter_type* = Tolubinski_Kostanchuk.

bubble_departure_maximum_diameter (real) [=0.0014]
Departing bubble maximum diameter, D_{max}. Use with *bubble_departure_diameter_type* = Tolubinski_Kostanchuk.

bubble_departure_scaling_temperature (real) [=45]
Departing bubble scaling temperature, T_{scal}. Use with *bubble_departure_diameter_type* = Tolubinski_Kostanchuk.

bubble_departure_diameter _user_function *(string) [no default]*
Name of the user-defined function. Used with *bubble_departure_diameter _type* = user_function.

bubble_departure_diameter_user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with *bubble_departure_diameter_type* = user_function.

bubble_departure_diameter_user_strings *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with *bubble_departure_diameter_type* = user_function.

liquid_subcooling_temperature_type *(enumerated) [=fixed_yplus]*
Type of sub-cooling or bulk temperature calculation for *bubble_departure_diameter*.

fixed_yplus	If selected, a fixed user-given y^+ location will be considered for bulk temperature calculation. The value can be set using *liquid_subcooling_fixed_yplus*.
fixed_temperature	If selected, a fixed user-given temperature will be considered for bulk temperature calculation. The value can be set using *liquid_subcooling_fixed_temperature*.

liquid_subcooling_fixed_yplus *(real) [=250]*
Fixed y^+ that will be used to calculate bulk temperature for *liquid_subcooling_temperature_type* = fixed_yplus.

liquid_subcooling_fixed_temperature *(real) [=273.15]*
Fixed temperature that will be used to calculate bulk temperature for *liquid_subcooling_temperature_type* = fixed_temperature.

condensation_factor *(real) [=1.0]*
Volumetric condensation correction factor used with *phase_change_type* = wall_boiling.

evaporation_factor *(real) [=1.0]*
Volumetric evaporation correction factor used with *phase_change_type* = wall_boiling.

carrier_field *(string) [=none]*
Name of a carrier field. Only one carrier field is accepted. The name must refer to a defined field entry. Only used for field interaction with *type* = algebraic_eulerian or eulerian _eulerian. The heavier fluid should be assigned as *carrier_field* when *type* = algebraic_eulerian or eulerian _eulerian. For *phase_change_type* = wall_boiling, carrier should be the liquid phase.

dispersed_field *(string) [=none]*
Name of a dispersed field. Only one dispersed field per *field_interacion_model* is accepted. The name must refer to a defined field entry. Only used for field interaction with *type* = algebraic_eulerian or eulerian _eulerian. For *phase_change_type* = wall_boiling, dispersed should be the vapor phase.

dispersed_field_diameter_type *(enumerated) [=constant]*
Type of the dispersed field, such as bubbles, droplets and particles. For non-boiling applications, currently constant is only supported. Only used for field interaction with *type* = algebraic_eulerian or eulerian _eulerian.

constant A constant value is used for dispersed field. Should be set using *dispersed_field_diameter*.

boiling_Kurul_linear If used for *phase_change_type* = wall_boiling, dispersed field (vapor) diameter will be calculated using piecewise linear model proper By Kurul. In this model, diameter will be changed from 0 to 2 mm.

boiling_Unal If used for *phase_change_type* = wall_boiling, dispersed field (vapor) diameter will be calculated using piecewise linear model proper By Unal. In this model, diameter will be changed from 0 to 1.5 mm.

boiling_Kurul_exponential If used for *phase_change_type* = wall_boiling, dispersed field (vapor) diameter will be calculated using an exponentially smoothed piecewise linear model proper By Kurul. In this model, diameter will be changed from 0 to 2 mm.

dispersed_field_diameter *(real) [=0.001]*
Diameter of the dispersed field, such as bubbles, droplets and particles, when *dispersed_field_diameter_type* = constant. Only used for field interaction with *type* = algebraic_eulerian.

drag_model *(enumerated) [=schiller_nauman]*
Type of the drag model used in algebraic slip model to calculate momentum exchange between fields. Only used for field interaction with *type* = algebraic_eulerian or eulerian _eulerian.

schiller_nauman Used for small sparsely distributed spherical bubbles, droplets and particles. Bubble and droplet deformations are ignored.

Ishii_Zuber Used for sparsely distributed fluid particles and bubbles. Bubble and droplet deformations can be considered if a surface tension model is specified.

Grace Used for sparsely distributed fluid particles and bubbles. Bubble and droplet deformations can be considered if a surface tension model is specified. This model is suitable for large bubbles.

Wen_Yu Used for densely distributed solid particles. For AcuSolve-EDEM coupling, it is the only value for low solid fraction regimes.

Gidaspow Also called Ergun-Wen-Yu. Used for AcuSolve-EDEM coupling for all solid fraction ranges. Can be used for spherical and non-spherical particles. The model is also used for granular flows for all solid fraction ranges and it can be used for spherical particles only.

DiFelice	Used for AcuSolve-EDEM coupling for all solid fraction ranges. Can be used for spherical and non-spherical particles.
Syamlal_Obrien	Used for AcuSolve-EDEM coupling for all solid fraction ranges. Can be used for spherical particles only.
Beetstra	Used for AcuSolve-EDEM coupling for all solid fraction ranges. Can be used for spherical particles only.
Rong	Used for AcuSolve-EDEM coupling for all solid fraction ranges, primarily used for non-spherical particles.
EMMS	Used for AcuSolve-EDEM coupling for all solid fraction ranges, and it can be used for spherical and non-spherical particles. The model is also used for granular flows for all solid fraction ranges and it can be used for spherical particles only.
user_function	User-defined function for AcuSolve-EDEM coupling drag model. Requires *drag_model_user_function*, *drag_model_user_values*, and *drag_model_user_strings*.

drag_model_user_function *(string) [no default]*
Name of the drag model user-defined function. Used with *drag_model* = user_function.

drag_model_user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with *drag_model*= user_function.

drag_model_user_strings *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with *drag_model*= user_function.

Ergun_drag_coefficient_A *(real) [=150]*
Constant factor of viscos term in Ergun drag model. Used with *drag_model*= Gidaspow or EMMS.

Ergun_drag_coefficient_B *(real) [=1.75]*
Constant factor of inviscid term in Ergun drag model. Used with *drag_model*= Gidaspow or EMMS.

nonspherical_drag_coef_model *(enumerated) [=none]*
Type of the drag coefficient model used for non-spherical particles in AcuSolve-EDEM coupling. Only used with *drag_model*= Gidaspo, DiFelice and Rong.

none	If used, spherical drag models will be considered.
Haider_Levenspiel	Used to calculate drag coefficient for isometric shape particles where orientation of particles can be ignored.
Holzer_Sommerfeld	Used to calculate drag coefficient for non-isometric shape particles.
Ganser	Used to calculate drag coefficient for non-isometric shape particles.

edem_coarse_grain_factor (real) [=1.0]
The *edem_coarse_grain_factor* that is used to scale the coupling forces in a coarse-grain system. In this approach, a certain number of particles is represented by a single bigger particle known as grain which reduces the total number of particles that need to be simulated. The coupling forces must be scaled accordingly so that the dynamics of the grains in the system represents the dynamics of particles in the original system. The coarse grain factor (k) is the ratio of the grain radius to the original particle. Each grain will virtually contain k^3 number of particles. In the coupled simulation of a coarse-grain system, the forces are calculated based on the original particles and then get scaled by k^3 value.

edem_particles_names (string) [=none]
List of particle names used to assign dedicated drag, lift and torque models based on particle shape and size.

edem_mass_transfer (boolean) [=off]
Flag for conducting mass transfer for the liquid phase in AcuSolve-EDEM coupling.

edem_contact_angle (real) [=70]
Contact angle (degree) between particle and gas/liquid interface, varying from 0 degrees to 180 degrees and used for the three-phase AcuSolve-EDEM coupling. The *surface_tension_model* for all three phases is required.

max_packing_volume_fraction (real) [=1.0]
Maximum packing of dispersed field volume fraction when *drag_model* = ishii_zuber. The default value of 1.0 is used for fluid particles. For solid a value of 0.62 is recommended.

lift_model (enumerated) [=legendre_magnaudet]
Type of the lift model to calculate momentum exchange between fields. Used for field interaction with *type* = algebraic_eulerian or eulerian_eluerian.

none	Lift force will be ignored in the algebraic slip model.
legendre_magnaudet	Used for small bubbles, droplets and particles. Bubble and droplet deformations are ignored.
Saffman_Mei	Used for small sparse bubbles, droplets and particles. Bubble and droplet deformations are ignored.
Tomiyama	Used for gas bubbles in liquids. A surface tension model is required.
Moraga	Used for small sparse bubbles, droplets and particles. Bubble and droplet deformations are ignored.
nonspherical_lift	Used only with AcuSolve-EDEM coupling for solid particles. Lift force caused by the particle's incident angle.
Saffman_Magnus	Used only with AcuSolve-EDEM coupling for solid particles. Lift force due to the rotation of a particle (Magnus) and due to the flow velocity gradient (Saffman).

Saffman_Magnus_nonspherical_lift	Use only with AcuSolve-EDEM coupling for solid particles. Lift force accounting both Saffman_Magnus and nonspherical_lift.
user_function	User-defined function for AcuSolve-EDEM coupling lift model. Requires *lift_model_user_function*, *lift_model_user_values*, and *lift_model_user_strings*.

lift_model_user_function *(string) [no default]*
Name of the lift model user-defined function. Used with *lift_model* = user_function.

lift_model_user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with *lift_model*= user_function.

lift_model_user_strings *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with *lift_model*= user_function.

Saffman_lift_coefficient *(real) [=1.615]*
Constant factor of Saffman lift force. Used with *lift_model*= Saffman_Magnus or Saffman_Magnus_nonspherical_lift.

Magnus_lift_coefficient *(real) [=0.125]*
Constant factor of Magnus lift force. Used with *lift_model*= Saffman_Magnus or Saffman_Magnus_nonspherical_lift.

torque_model *(enumerated) [=none]*
Type of the torque model used in the AcuSolve-EDEM coupling to calculate the torque acting on particles. Only used for field interaction with *type*= algebraic_eulerian or eulerian_eulerian.

none	Torque will be ignored.
pitching_torque	Torque due to the offset of the center of pressure from geometric center.
rotational_torque	Torque caused by the relative rotation of a particle with respect to rotation of fluid.
pitching_rotational_torque	Torque accounting both pitching torque and rotational torque.
user_function	User-defined function for AcuSolve-EDEM coupling torque model. Requires *torque_model_user_function*, *torque_model_user_values*, and *torque_model_user_strings*.

torque_model_user_function *(string) [no default]*
Name of the torque model user-defined function. Used with *torque_model* = user_function.

torque_model_user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with *torque_model* = user_function.

torque_model_user_strings (list) [={}]
Array of strings to be passed to the user-defined function. Used with *torque_model* = user_function.

wall_lubrication_model (enumerated) [=none]
The wall lubrication model is used to calculate the repulsive force between dispersed phase and the wall. Only used for field interaction with *type* = {algebraic_eulerian, eulerian_eulerian}.

none	Wall lubrication force will be ignored.
Antal	Used for computing the repulsive force between bubbles (dispersed phase) and wall.

virtual_mass_model (boolean) [=off]
Flag specifying whether to calculate virtual mass force. This force is due to the relative acceleration between phases. Only used for field interaction with *type* = {algebraic_eulerian, eulerian_eulerian}.

diffusion_matrix_type (enumerated) [=constant]
Type of diffusion coefficient used to calculate individual binary diffusion of the mixture.

constant
Constant values of self (diagonal) and binary (off diagonal) diffusion.

Coefficients should be provided through *diffusion_matrix*. Using the values in the matrix, binary/mixture diffusion will be used to calculate diffusion for individual field.

gas_kinetic
Gas kinetic diffusivity model for gases only. Requires GAS_KINETIC_MODEL. Using the values from gas kinetic model, binary/mixture diffusion will be used to calculate diffusion for individual field.

diffusion_matrix (array) [={}]
Symmetric array of diffusion coefficients for the field mixture diffusion. The first three numbers define the diagonal values (self-diffusion coefficients) and the last three numbers define the binary coefficients. For the three-field mixture, it will be {D11,D22,D33,D12,D13,D23}. Use with *advective_diffusive*. Order of indices will be extracted from fields.

granular_flow (boolean) [=off]
Flag specifying whether to calculate solid properties (solid pressure and solid viscosity) for granular flows. If off, solid properties will be ignored. Only used for field interaction with *type*= eulerian_eulerian.

particle_restitution (real) [=0.9]
The coefficient of particle restitution is the ratio of the final to initial relative velocity between two particles after they collide. It normally ranges from 0 to 1, where 1 would be a perfectly elastic collision. It is used when *granular_flow*= on. The default value of 0.9 is used.

angle_internal_friction (real) [=30.0]
The angle of internal friction is a measure of the ability of a unit of rock or soil to withstand a shear stress. It is the angle on the graph (Mohr's Circle) of shear stress and normal effective stresses at which shear failure occurs. It is used when *granular_flow*= on. The default value of 30.0 is used. The relation between the angle of internal friction and standard penetration number

is shown below (Peck, B. P., Hanson, E. W., & Thornburn, H. T. (1974). Foundation Engineering. 2nd. Edition. New York: John Willy & Sons, 7-8.)

SPT penetration, N-Value (blows/ foot)	**Density of Sand**	**Angle of internal friction (degrees)**
<4	Very loose	<29
4 - 10	Loose	29 - 30
10 - 30	Medium	30 - 36
30 - 50	Dense	36 - 41
>50	Very dense	>41

granular_temperature_model (enumerated) [=algebraic]
Type of the granular temperature model used to calculate granular temperature for granular flows. Only used for field interaction with *type*= eulerian_eulerian.

none	The granular temperature will be ignored.
constant	Constant granular temperature will be used.
algebraic	The granular temperature will be computed from the algebraic granular temperature equation.

constant_granular_temperature (real) [=0.0001]
The value of constant granular temperature. It is used when *granular_temperature_model*= constant. The default value of 0.0001 is used.

solid_pressure_model (enumerated) [=kinetic_collision]
Type of solid pressure model used for granular flows. Only used for field interaction with *type*= eulerian_eulerian.

none	The solid pressure will be ignored.
collision	Only the collisional component of solid pressure will be considered.
kinetic_collision	Both the kinetic and collisional components of the solid pressure will be considered.

solid_viscosity_model (enumerated) [=Gidaspow]
Type of solid viscosity model used for granular flows. Only used for field interaction with *type*= eulerian_eulerian.

constant	Constant value of solid viscosity will be used.

Gidaspow	The Gidaspow model for solid viscosity will be used.

constant_solid_viscosity (real) [=0.01]
The value of constant solid viscosity. It is used when *solid_viscosity_model* = constant. The default value of 0.01 is used.

surface_tension_model (string) [=none]
Name of a valid surface tension model that will be used for the immiscible or miscible models. For miscible models, a surface tension model is used to model liquid droplet and gas bubble deformation. For *type* = eulerian_eulerian, the *surface_tension_model* is in effect for all drag and lift models. For *type* = algebriac_eulerian, the *surface_tension_model* is used when *drag_model* = ishii_zuber, *drag_model* = grace or *lift_model* = tomiyama.

thickness_type (enumerated) [=automatic]
Type of the interface thickness type for immiscible field interaction (levelset and algebraic_eulerian).

constant or const	Constant value of the interface thickness.
num_elements or nelems	Constant value of the interface thickness calculated based on the average element size.
automatic or auto	Equivalent to the num_elements option with the value of *thickness_num_elements* = 3.

thickness (real) >=0 [=0]
Constant value of the interface thickness. Used with *thickness_type* = constant option.

> **Note:** A value of zero will be interpreted as *thickness_type* = automatic.

thickness_num_elements (real) >=1 [=1]
Multiplies the average element size to obtain the interface thickness length.

> **Note:** The average element size is calculated once at the beginning of the simulation and is calculated using only the elements intersected by the interface. If no interface is present in the volume, then the average is computed using all elements.

turbulence_damping_levelset (boolean) [=off]
Flag specifying whether to mitigate the turbulence generation at the interface for immiscible field interaction (levelset). Only used with the spalart-allmaras (SA) turbulence model or shear stress transport (SST) turbulence model.

Description

This command specifies the interaction between fields.

FIELD_INTERACTION_MODEL commands are referenced by the MULTI_FIELD command.

In AcuSolve-EDEM coupling, you may need to select different drag models for particles based on their size and shape. To achieve this, each drag model must be defined using the dedicated FIELD_INTERACTION_MODEL. To support this, a new parameter is introduced, *edem_particles_names*,

which retrieves the list of EDEM particles referenced in the `FIELD_INTERACTION_MODEL`. Below is an example involving three particles: wheat, rice, and pea.

```
FIELD_INTERACTION_MODEL( "Particle1-Air" ) {
    carrier_field = "Air"
    dispersed_field = "Particle"
    drag_model = Rong
    nonspherical_drag_coef_model = Holzer_Sommerfeld
    lift_model = Saffman_Magnus_nonspherical_lift
    edem_particles_names = { “Rice", “Wheat" }
}
FIELD_INTERACTION_MODEL( "Particle2-Air" ) {
    carrier_field = "Air"
    dispersed_field = "Particle"
    drag_model = Ergun_Wen_Yu
    nonspherical_drag_coef_model = none
    lift_model = none
    edem_particles_names = { “Pea" }
                    }
# +-------------------------------------------------------------------------+
# | Multi Field Model                                 |
# +-------------------------------------------------------------------------+
 MULTI_FIELD_MODEL( "Particle-Air" ) {
    field_interaction_models = { "Particle1-Air“, "Particle2-Air" }
    density_type = linear
    specific_heat_type = linear
    viscosity_type = linear
    conductivity_type = linear
    immiscible_interface_type = smooth
}
```

The following is an example of the levelset field interaction model:

```
FIELD_INTERACTION_MODEL( "water-air combo" ) {
   type           = levelset
   fields_1       = {"air"}
   fields_2       = {"water"}
   thickness_type = auto
}
```

The following is an example of the algebric_eulerian field-interaction model setup, which assigns the heavier fluid as *carrier_field*. It is also noted that the *carrier_field* has to be a fluid, while the dispersed field can be any medium, for example gas bubbles, particles and droplets.

```
FIELD_INTERACTION_MODEL( "water-bubble combo" ) {
    type                      = algebraic_eulerian
    dispersed_field           = “bubble”
    carrier_field           = “water”
    algebraic_slip_model          = on
    dispersed_field_diameter      = 0.001
    drag_model                    = schiller_nauman
    lift_model                    = legendre_magnaudet
    thickness_type           = auto
   turbulent_schmidt_number   = 0.85
}
```

In the special case of the humid-air model, AcuSolve incorporates these coefficients internally, hence {0,0,0;} is used in an input file.

```
FIELD_INTERACTION_MODEL( "humidAir" )
    fields = {"humid_air","air"}
    diffusion_matrix = {0,0,0;}
}
```

MATERIAL_MODEL

Specifies a material model by collecting specific models.

Type

AcuSolve Command

Syntax

```
MATERIAL_MODEL("name") { parameters...}
```

Qualifier

User-given name.

Parameters

type (enumerated) [=fluid]

Type of the material model.

fluid	Fluid material model.
solid	Solid material model.

density_model or dens (string) [no default]

User-given name of the density model. This is a required parameter. Density is used in the fluid, solid, and shell element sets.

specific_heat_model or cp (string) [=none]

User-given name of the specific heat at constant pressure model. This parameter is needed for solving the temperature equation in the fluid, solid, and shell element sets. If none, no specific heat model is provided.

viscosity_model or visc (string) [=none]

User-given name of the viscosity model. This parameter is needed for solving the flow equations in the fluid element sets. If none, no viscosity model is provided.

conductivity_model or cond (string) [=none]

User-given name of the thermal conductivity model. This parameter is needed for solving the temperature equation in the fluid, solid, and shell element sets. If none, no conductivity model is provided.

vapor_phase_model (string) [=none]

User-given name of the vapor-phase model. This parameter is used exclusively for solving the single-phase nucleate boiling. This material model represents the vapor state of the current material and whose properties are used in the evaluation of the heat transfer due to boiling. It is important to note that for single-phase nucleate boiling, the material model for the vapor phase requires the latent heat and latent heat temperature entries. Used when *nucleate_boiling* is set to on to the wall type. If none, no vapor phase model is provided.

surface_tension_model (string) [=none]
User-given name of the surface tension model. This parameter is needed for solving the single-phase nucleate boiling. Used when *nucleate_boiling* is set to on to the wall type. If none, no surface tension model is provided.

diffusivity_1_model or *diff_1 (string) [=none]*

diffusivity_2_model or *diff_2 (string) [=none]*

diffusivity_3_model or *diff_3 (string) [=none]*

diffusivity_4_model or *diff_4 (string) [=none]*

diffusivity_5_model or *diff_5 (string) [=none]*

diffusivity_6_model or *diff_6 (string) [=none]*

diffusivity_7_model or *diff_7 (string) [=none]*

diffusivity_8_model or *diff_8 (string) [=none]*

diffusivity_9_model or *diff_9(string) [=none]*
User-given name of the diffusivity model for the corresponding species transport equation. This parameter is needed for solving the corresponding species transport equation in the fluid element sets. If none, no diffusivity model for the corresponding species is provided.

porosity_model or *poros (string) [=none]*
User-given name of the porosity model. This model may be used for solving the flow equations in the fluid element sets. If none, no porosity modeling is provided.

viscoelastic_model (string) [=none]
User-given name of the viscoelastic model. This model may be used for solving the flow equations in the fluid element sets. If none, no viscoelastic modeling is provided.

radiation_model (string) [=none]
User-given name of the material radiation model. This model may be used for solving the P1 and discrete ordinates radiation equations in the fluid or solid element sets that are participating. Used with MATERIAL_RADIATION_MODEL. If none, no MATERIAL_RADIATION_MODEL is provided and the medium does not participate in radiative heat transfer.

Description

This command specifies a material model.

MATERIAL_MODEL commands collect

- DENSITY_MODEL
- SPECIFIC_HEAT_MODEL
- VISCOSITY_MODEL
- CONDUCTIVITY_MODEL
- DIFFUSIVITY_MODEL
- POROSITY_MODEL
- VISCOELASTIC_MODEL
- MATERIAL_RADIATION_MODEL

commands by referencing their user-given names. The MATERIAL_MODEL commands are in turn referenced by ELEMENT_SET commands. For example,

```
DENSITY_MODEL( "my density model" ) { ... }
SPECIFIC_HEAT_MODEL( "my specific heat model" ) { ... }
VISCOSITY_MODEL( "my viscosity model" ) { ... }
CONDUCTIVITY_MODEL( "my conductivity model" ) { ... }
DIFFUSIVITY_MODEL( "my diffusivity model 1" ) { ... }
DIFFUSIVITY_MODEL( "my diffusivity model 2" ) { ... }
POROSITY_MODEL( "my porosity model" ) { ... }
VISCOELASTIC_MODEL( "my viscoelastic model" ) { ... }
MATERIAL_RADIATION_MODEL( "my radiation model" ) { ... }
MATERIAL_MODEL( "my material model" ) {
 density_model       = "my density model"
 specific_heat_model = "my specific heat model"
 viscosity_model     = "my viscosity model"
 conductivity_model  = "my conductivity model"
 diffusivity_1_model = "my diffusivity model 1"
 diffusivity_2_model = "my diffusivity model 1"
 diffusivity_3_model = "my diffusivity model 2"
 porosity_model      = "my porosity model"
 viscoelastic_model  = "my viscoelastic model"
 radiation_model     = "my radiation model"
}
ELEMENT_SET( "fluid elements" ) {
 material_model = "my material model"
...
}
```

Every ELEMENT_SET command of medium fluid or solid must reference a MATERIAL_MODEL command. ELEMENT_SET commands of medium shell must reference an array of MATERIAL_MODEL commands, one for each layer. The same material model may be shared by any number of element sets or shell layers. All material models not referenced are ignored.

A material model must contain all the necessary parts associated with the equations being solved in the problem. These equations are defined by the EQUATION command.

A material model which is referenced by a fluid element set (see the medium parameter of the ELEMENT_SET command) must include the parameters:

Table 1:

Must Include	When Solving For
density_model	any equation
viscosity_model	flow and turbulence
specific_heat_model	temperature
conductivity_model	temperature
diffusivity_1_model	species 1
diffusivity_2_model	species 2

Must Include	When Solving For
diffusivity_3_model	species 3
diffusivity_4_model	species 4
diffusivity_5_model	species 5
diffusivity_6_model	species 6
diffusivity_7_model	species 7
diffusivity_8_model	species 8
diffusivity_9_model	species 9
viscoelastic_model	viscoelasticity
radiation_model	P1 or discrete ordinates radiation

In addition, the *porosity_model* may be added for the flow equations. In general, you should always include the *viscosity_model* even if the flow equations are not being solved. Some models, such as the conductivity model with a constant Prandtl number, may require the viscosity.

Since only the energy and mesh displacement equations are defined in the solid and shell element sets, the corresponding material models need only the *density_model*, *specific_heat_model*, and *conductivity_model* parameters.

In topology optimization the design domain should have a separate element set where a design material model is defined, in addition to the normal material model. For topology flow optimization, the design material model is representing a solid, which is defined as a fluid type with a porosity model with a high Darcy coefficient.

Note: A single element set must be used to define the design domain (where `medium = design`). Any number of element sets of other medium types may still be used in the simulation.

```
ELEMENT_SET( "Design" ) {
    …
    medium = design
    material_model = "fluid_medium"
    design_material_model = "poros_solid_medium"
    design_variables_field = "topology"
    …
}
```

```
MATERIAL_MODEL( "fluid_medium" ) {
    type = fluid
    density_model = "air"
    viscosity_model = "air"
    porosity_model = "none"
}
```

```
DENSITY_MODEL( "air" ) {
…
}
VISCOSITY_MODEL( "air" ) {
…
}
POROSITY_MODEL( "air" ) {
    type = none
}
MATERIAL_MODEL( "poros_solid_medium" ) {
    type = fluid
    density_model = "air"
    viscosity_model = "air"
    porosity_model = "poros"
}
DENSITY_MODEL( "air" ) {
}
VISCOSITY_MODEL( "air" ) {
}
POROSITY_MODEL( "poros" ) {
    type = "constant"
    porosity = 1.0
    forchheimer_coefficient = 0.0
    darcy_coefficient = 1.e5
}
```

MATERIAL_RADIATION_MODEL

Specifies material for the P1 and *discrete_ordinate* radiation model.

Type

AcuSolve Command

Syntax

MATERIAL_RADIATION_MODEL("name") {parameters...}

Qualifier

User-given name.

Parameters

absorption_coefficient *(real) [=1.0 1/meter]*
: Absorption of thermal radiation per unit length by the participating medium. Absorption coefficient is greater or equal to zero for the *discrete_ordinate* model and greater than zero for the *p1_model*. A value of zero means the medium is non-participating.

scattering_coefficient *(real) [= 0.0 1/meter]*
: Scattering of radiation leading to further attenuation (weakening) of the incident radiation.

phase_coefficient *(real) [= 0.0]*
: Linear anisotropic phase function coefficient for linear anisotropic scattering of the incident radiation. *phase_coefficient* range: -1 < *phase_coefficient* < 1. The three types of scattering are isotropic scattering (*phase_coefficient* = 0.0), i.e., scattering is the same in all directions; forward dominant scattering (*phase_coefficient* > 0) of the incident radiation, that is, more scattering in the forward direction than backward direction; and backward dominant scattering (*phase_coefficient* < 0). The closer the coefficient is to 1 or –1 the more dominant the scattering is in the forward or backward direction, respectively.

refractive_index *(real) [= 1.0]*
: Ratio of the velocity of incident radiation propagation in a medium compared to a vacuum.

Description

This command specifies radiation material parameters for the P1 and discrete ordinates radiation model. All media, both fluid or solid, that are participating require a MATERIAL_RADIATION_MODEL.

For a MATERIAL_MODEL to be participating, the *radiation_model* command is used to reference a MATERIAL_RADIATION_MODEL. For example,

```
MATERIAL_MODEL( "my_participating_medium" ) {
radiation_model = "my_radiation_material"
...
}
MATERIAL_RADIATION_MODEL( "my_radiation_material" ){
    absorption_coefficient              = 200.0
    scattering_coefficient              = 0.0
    phase_coefficient                   = 0.0
    refractive_index                    = 1.51
```

```
}
```

If the MATERIAL_MODEL is not participating the *radiation_model* can simply be removed.

For a radiation material that is participating the total attenuation/loss, also termed extinction, of the incident radiative intensity is governed by the combination of absorption and scattering within the medium. The former represents absorption of intensity by the medium, while the latter is scattering of intensity onto other directions.

> **Note:** For the P1 model the directional dependency is integrated out. The absorption and scattering coefficients represent a per unit length intensity loss within a participating medium.

The following represents a material that has an absorption coefficient of 1.0 m^{-1} and a scattering coefficient of 1.0 m^{-1} that is isotropic in all directions. Therefore, the extinction coefficient for this material is 2.0. The refractive index of the material means the velocity of incident radiation is identical to that of a vacuum. Materials, such as glass, have a higher refractive index meaning radiation propagates slower through the medium.

```
MATERIAL_RADIATION_MODEL( "Medium_HM" ) {
    absorption_coefficient              = 1.0
    scattering_coefficient              = 1.0
    phase_coefficient                   = 0.0
    refractive_index                    = 1.0
}
```

MULTI_FIELD_MODEL

Specifies an equivalent material model for multi-field simulation.

Type

AcuSolve Command

Syntax

```
MULTI_FIELD_MODEL("name") {parameters...}
```

Qualifier

User-given name.

Parameters

field_interation_models *(string) [=none]*

List of names of field interaction models that are applied to this material model. The name must refer to a defined *field_interaction_model* entry.

specific_heat_type *(enumerated) [=linear]*

Type of blending of the different material specific heat.

linear	Linear combination of the individual material specific heat weighted by their respective volume fraction.

viscosity_type *(enumerated) [=linear]*

Type of blending of the different material viscosity.

linear	Linear combination of the individual material viscosity weighted by their respective volume fraction.

conductivity_type *(enumerated) [=linear]*

Type of blending of the different material conductivity.

linear	Linear combination of the individual material conductivity weighted by their respective volume fraction.

density_type *(enumerated) [=linear]*

Type of blending of the different material density.

linear	Linear combination of the individual material density weighted by their respective volume fraction.
harmonic	Harmonic combination of the individual material density weighted by their respective volume fraction.

diffusivity_1_type *(enumerated) [=linear]*

Type of blending of the different material scalar diffusivity (scalar 1).

linear Linear combination of the individual material diffusivity for scalar 1 weighted by their respective volume fraction.

diffusivity_2_type *(enumerated) [=linear]*
Type of blending of the different material scalar diffusivity (scalar 2).

linear Linear combination of the individual material diffusivity for scalar 2 weighted by their respective volume fraction.

diffusivity_3_type *(enumerated) [=linear]*
Type of blending of the different material scalar diffusivity (scalar 3).

linear Linear combination of the individual material diffusivity for scalar 3 weighted by their respective volume fraction.

diffusivity_4_type *(enumerated) [=linear]*
Type of blending of the different material scalar diffusivity (scalar 4).

linear Linear combination of the individual material diffusivity for scalar 4 weighted by their respective volume fraction.

diffusivity_5_type *(enumerated) [=linear]*
Type of blending of the different material scalar diffusivity (scalar 5).

linear Linear combination of the individual material diffusivity for scalar 5 weighted by their respective volume fraction.

diffusivity_6_type *(enumerated) [=linear]*
Type of blending of the different material scalar diffusivity (scalar 6).

linear Linear combination of the individual material diffusivity for scalar 6 weighted by their respective volume fraction.

diffusivity_7_type *(enumerated) [=linear]*
Type of blending of the different material scalar diffusivity (scalar 7).

linear Linear combination of the individual material diffusivity for scalar 7 weighted by their respective volume fraction.

diffusivity_8_type *(enumerated) [=linear]*
Type of blending of the different material scalar diffusivity (scalar 8).

linear Linear combination of the individual material diffusivity for scalar 8 weighted by their respective volume fraction.

diffusivity_9_type *(enumerated) [=linear]*
Type of blending of the different material scalar diffusivity (scalar 9).

linear Linear combination of the individual material diffusivity for scalar 9 weighted by their respective volume fraction.

immiscible_interface_type *(enumerated) [=smooth]*
Type of interface modelling for multi-field simulation.

smooth	The interface is diffused over a given distance with material properties transitioning smoothly between the different phases.

Description

This command specifies models to be used to compute the average properties for the fluid in multi-field simulation.

`MULTI_FIELD_MODEL` commands are referenced by `ELEMENT_SET` commands and replace the `MATERIAL_MODEL` used in single phase simulation.

```
MULTI_FIELD_MODEL( "my multi fluid model" ) {
    field_interaction_models      = {"my FIM"}
    density_type                  = linear
    specific_heat_type            = linear
    viscosity_type                = linear
   immiscible_interface_type      = smooth
}
```

POROSITY_MODEL

Specifies a porosity model for the flow equation.

Type

AcuSolve Command

Syntax

```
POROSITY_MODEL("name") {parameters...}
```

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of the porosity model.

none	No porosity.
constant or const	Darcy-Forchheimer porosity model. Requires *permeability*, *darcy_coefficient*, *forchheimer_coefficient* and *permeability_direction*.

porosity (real) >0 <=1 [=1]

Porosity value.

permeability_type (enumerated) [=cartesian]

Permeability type.

cartesian	Use Cartesian coordinate system to specify permeabilities and directions.
cartesian_bidirectional	Use Cartesian coordinate system to specify directions and separate permeabilities for Darcy and Forchheimer terms.
cylindrical	Use cylindrical coordinate system to specify permeabilities and directions.
spherical	Use spherical coordinate system to specify permeabilities and directions.

direction_1_permeability (real) [=1]

direction_2_permeability (real) [=1]

direction_3_permeability (real) [=1]

Permeability of the porous media in each of the three orthogonal principal axes. Used with constant type. These commands are valid only when *permeability_type*=cartesian. Also, these commands are equivalent to the previously used permeability array [={1,1,1}], which has been deprecated. AcuSolve will automatically map the *permeability_array* to the appropriate

direction_1_permeability, *direction_2_permeability*, and *direction_3_permeability* parameters when reading an input file that uses the deprecated syntax. If both parameters are specified, this mapping will cause the permeability to take precedence over the values that are specified for *direction_1_permeability*, *direction_2_permeability* and *direction_3_permeability*.

direction_1_permeability_forch *(real) [=1]*

direction_2_permeability_forch *(real) [=1]*

direction_3_permeability_forch *(real) [=1]*
Permeability for the Forchheimer term of the porous media in each of the three orthogonal principal axes. Used with *cartesian_bidirectional*.

> **Note:** The directions of the axes are specified in the same manner as the cartesian option and is the same as the Darcy term.

direction_1_permeability_darcy *(real) [=1]*

direction_2_permeability_darcy *(real) [=1]*

direction_3_permeability_darcy *(real) [=1]*
Permeability for the Darcy term of the porous media in each of the three orthogonal principal axes. Used with *cartesian_bidirectional*.

> **Note:** The directions of the axes are specified in the same manner as the cartesian option and is the same as the Forchheimer term.

cartesian_permeability_directions *(array) [={1,0,0;0,1,0;0,0,1}]*
Orthogonal principal axes of the *permeability* values. The vectors are ortho-normalized prior to use. Used with constant type. This command is equivalent to the previously used permeability_direction [={1,0,0;0.1.0;0,0,1}]. If both are specified in the input file, the one that appears last will be used for the computation.

direction_1_permeability_multiplier_function *(string) [=none]*

direction_2_permeability_multiplier_function *(string) [=none]*

direction_3_permeability_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the permeabilities. If none, no scaling is performed. These commands are valid only when *permeability_type* = cartesian.

radial_permeability *(real) [=1]*
Radial permeability of the porous media in cylindrical and spherical coordinate systems. This command is valid only when *permeability_type* = cylindrical or spherical.

axial_permeability *(real) [=1]*
Axial permeability of the porous media in cylindrical coordinate system. This command is valid only when *permeability_type* = cylindrical.

tangential_permeability *(real) [=1]*
Tangential permeability of the porous media in cylindrical and spherical coordinate systems. This command is valid only when *permeability_type* = cylindrical or spherical.

cylindrical_permeability_axis (array) [={0,0.0;1,0,0}]
Principal axis of the axial permeability value. Other two principal axes in cylindrical coordinate are computed internally and ortho-normalized prior to use.

radial_permeability_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the permeabilities. If none, no scaling is performed. This command is valid only when *permeability_type* = cylindrical or spherical.

axial_permeability_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the permeabilities. If none, no scaling is performed. This command is valid only when *permeability_type* = cylindrical.

tangential_permeability_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the permeabilities. If none, no scaling is performed. This command is valid only when *permeability_type* = cylindrical or spherical.

spherical_permeability_center (array) [={0,0,0}]
Spherical center of permeability values. This command is valid only when *permeability_type* = spherical.

darcy_coefficient (real) [=0]
Coefficient of Darcy's (linear) term. Used with constant type.

forchheimer_coefficient (real) [=0]
Coefficient of Forchheimer's (quadratic) term. Used with constant type.

darcy_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the Darcy coefficient. If none, no scaling is performed.

forchheimer_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the Forchheimer coefficient. If none, no scaling is performed.

effectiveness_type (enumerated) [=none]
Type of the effectiveness of the porous model.

constant or const	Requires *effectiveness*.
piecewise_linear	Piecewise linear curve fit. Requires *effectiveness_curve_fit_variable* and *effectiveness_curve_fit_values*.
cubic_spline	Cubic spline curve fit. Requires *effectiveness_curve_fit_variable* and *effectiveness_curve_fit_values*.
user_function	User-defined function. Requires *effectiveness_user_function*, *effectiveness_user_values*, and *effectiveness_user_strings*.

effectiveness (real) >0 <=1 [=1]
Effectiveness of the flow resistance exerted by the porous medium.This command is valid only when *effectiveness_type* = constant.

effectiveness_curve_fit_variable (enumerated) [=temperature]
Independent variable of the curve fit for effectiveness. Used with piecewise_linear and cubic_spline types.

x_coordinate	x coordinate
y_coordinate	y coordinate
z_coordinate	z coordinate
x_reference_coordinate	x reference coordinate
y_reference_coordinate	y reference coordinate
z_reference_coordinate	z reference coordinate
temperature	Temperature
species_1	species 1
species_2	species 2
species_3	species 3
species_4	species 4
species_5	species 5
species_6	species 6
species_7	species 7
species_8	species 8
species_9	species 9

effectiveness_curve_fit_values (array) [={0,0}]
A two-column array of independent-variable/effectiveness data values. Used with piecewise_linear and cubic_spline effectiveness types.

effectiveness_user_function (string) [no default]
Name of the effectiveness user-defined function. Used with *effectiveness_type* = user_function.

effectiveness_user_values (array) [={}]
Array of values to be passed to the user-defined function. Used with *effectiveness_type* = user_function.

effectiveness_user_strings (array) [={}]
Array of strings to be passed to the user-defined function. Used with *effectiveness_type* = user_function.

Description

This command specifies a porosity model for the flow (momentum) equations. This model is only applicable to fluid element sets.

`POROSITY_MODEL` commands are referenced by `MATERIAL_MODEL` commands, which in turn are referenced by `ELEMENT_SET` commands:

```
POROSITY_MODEL( "my porous media" ) {
   type                                = constant
   direction_1_permeability            = 1.e-6
   direction_2_permeability            = 1.e-2
   direction_3_permeability            = 1.e-6
   cartesian_permeability_directions   = { 1, 0, 0 ;
                                           0, 1, 0 ;
                                           0, 0, 1 ; }
   darcy_coefficient                   = 0
   forchheimer_coefficient             = 0.5
 }
 MATERIAL_MODEL( "my material model" ) {
   porosity_model                      = "my porous media"
   ...
 }
 ELEMENT_SET( "fluid elements" ) {
   material_model                      = "my material model"
   ...
}
```

In AcuSolve, a porous media problem can be solved using two methods:

- Superficial Velocity
- Physical Velocity

In AcuSolve, the porous media is solved at the macroscopic level by averaging the behavior of the flow. The continuity and momentum equation of the flow, in AcuSolve, can be averaged using the superficial averaging method or intrinsic/physical averaging method. AcuSolve, by default, reports the superficially averaged value in the form of superficial velocity. Superficial velocity is calculated based on volumetric flow rate as follows:

$$V_{superficial} = \frac{Q}{A} \tag{88}$$

where Q is the volumetric flow rate and A is the cross-sectional area of the porous media. The continuity equation in superficial averaged form is divergence free for incompressible flow, which ensures continuity of velocity vector across the porous media boundaries. The superficial averaged method may not be very accurate but is useful when flow and pressure drop needs to be solved at the macro level. In AcuSolve, the following momentum equation is used for the superficial method:

$$\rho\frac{\partial u}{\partial t} + \rho u \cdot \nabla u = -\nabla p + \rho b + \nabla \cdot \tau - Rf \tag{89}$$

where ρ is the density; u is the velocity vector; p is the pressure; b is the specific body force; $\tau = [\tau_{ij}]$ is the viscous stress tensor; f is the porous media contribution, defined in the orthogonal principal axes given below; and R is the rotational tensor which rotates f to the global coordinate axes. The porous media term f for the superficial method is defined as;

$$f = \left[\frac{C_{\text{darcy}}\mu}{k} + \frac{C_{\text{forch}}\rho}{\sqrt{k}} |u| \right] u \tag{90}$$

where C_{darcy} is the Darcy coefficient; C_{forch} is the Forchheimer coefficient, and k is the permeability.

A more accurate representation of velocity inside the porous media can be attained by solving the continuity and momentum equation inside the porous media using the intrinsic/physical averaging method. AcuSolve solves for physical velocity $<v_\beta>^\beta$ inside the porous media using the below form of the flow equation:

$$\frac{\partial \varepsilon_\beta <\rho_\beta>^\beta}{\partial t} + <\rho_\beta v_\beta>^\beta \nabla \varepsilon_\beta + \varepsilon_\beta \nabla \cdot <(\rho_\beta v_\beta)>^\beta = 0 \tag{94}$$

$$\begin{aligned} &\rho_\beta \frac{\partial <v_\beta>^\beta}{\partial t} + \varepsilon_\beta^{-1} \nabla \cdot (\varepsilon_\beta \rho_\beta <v_\beta>^\beta <v_\beta>^\beta) = -\nabla <p_\beta>^\beta + \rho_\beta g \\ &+ \mu_\beta [\nabla^2 <v_\beta>^\beta + \varepsilon_\beta^{-1} \nabla \varepsilon_\beta \cdot \nabla <v_\beta>^\beta + \varepsilon_\beta^{-1} \nabla <v_\beta>^\beta \nabla^2 \varepsilon_\beta] - Rf \end{aligned} \tag{95}$$

where the term f is defined as:

$$f = \varepsilon_\beta \left[\frac{C_{Darcy}\mu_\beta}{k} + \varepsilon \frac{C_{Forch}\rho_\beta}{\sqrt{k}} <v_\beta>^\beta \right] <v_\beta>^\beta \tag{96}$$

The superficial velocity and the physical velocity are related using the equation below:

$$<v_\beta> = \varepsilon_\beta <v_\beta>^\beta \tag{97}$$

The *darcy_multiplier_function* parameter may be used to uniformly scale the Darcy coefficient. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file. For example, a Darcy coefficient ramped from zero to 0.5 over the first 10 time steps may be specified by:

```
POROSITY_MODEL( "ramped Darcy coefficient" ) {
   type                                = constant
   permeability_type                   = cartesian
   direction_1_permeability            = 1.e-6
   direction_2_permeability            = 1.e-2
   direction_3_permeability            = 1.e-6
   cartesian_permeability_directions   = { 1, 0, 0 ;    0, 1, 0 ;    0, 0, 1 ; }
   darcy_coefficient                   = 1
   forchheimer_coefficient             = 0.5
   darcy_multiplier_function           = "ramped"
   }
   MULTIPLIER_FUNCTION( "ramped" ) {
   type       = piecewise_linear
   curve_fit_vales                     = { 1, 0 ; 10, 0.5 }
   curve_fit_variable                  = time_step
}
```

Similarly, *forchheimer_multiplier_function* may be used to uniformly scale the Forchheimer coefficient. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION

command in the input file. For example, using the same `MULTIPLIER_FUNCTION` command as above, a Forchheimer coefficient ramped from zero to 0.5 over the first 10 time steps may be specified by:

```
POROSITY_MODEL( "ramped Forchheimer coefficient" ) {
   type                                = constant
   direction_1_permeability            = 1.e-6
   direction_2_permeability            = 1.e-2
   direction_3_permeability            = 1.e-6
   cartesian_permeability_direction    = { 1, 0, 0 ;    0, 1, 0 ;    0, 0, 1 ; }
   darcy_coefficient                   = 0.5
   forchheimer_coefficient             = 1
   forchheimer_multiplier_function     = "ramped"
}
```

While the default coordinate system is Cartesian, you can also use cylindrical or spherical coordinate systems to specify permeabilities and their directions. The `POROSITY_MODEL` commands below are an example of cylindrical coordinate system:

```
POROSITY_MODEL( "my porous media" ) {
    type                          = constant
    permeability_type             = cylindrical
    radial_permeability           = 1.e-6
    axial_permeability            = 1.e-2
    tangential_permeability       = 1.e-6
    cylindrical_permeability_axis = {0, 0, 0 ;   1, 0, 0 ; }
    darcy_coefficient             = 1
    forchheimer_coefficient       = 0.5
}
```

and for spherical coordinate system:

```
POROSITY_MODEL( "my porous media" ) {
    type                        = constant
    permeability_type           = spherical
    radial_permeability         = 1.e-2
    tangential_permeability     = 1.e-6
    spherical_permeability_axis = {0, 0, 0 ;
    darcy_coefficient           = 1
    forchheimer_coefficient     = 0.5
}
```

The *effectiveness* parameter may be used to define a multiplicative constant applied to the force f in equations (Equation 90) and (Equation 96). This parameter controls the effectiveness of the porous medium at providing a resistance to the flow. By setting a value of 1, the porous medium fully exerts its resistance to the flow, while by setting a value of 0, no resistance to the flow is applied.

The porous medium effectiveness can be made temperature dependent to model freezing and melting. The `POROSITY_MODEL` commands below are an example of a temperature dependent effectiveness going from 1 to 0 at a temperature of 262.15K linearly over 2K:

```
POROSITY_MODEL( "ModelforMelting" ) {
   type                                = constant
   direction_1_permeability            = 1.0
   direction_2_permeability            = 1.0
   direction_3_permeability            = 1.0
   cartesian_permeability_direction    = { 1, 0, 0 ; 0, 1, 0 ; 0, 0, 1}
   darcy_coefficient                   = 5e+8
   forchheimer_coefficient             = 0
```

```
   effectiveness_type                = piecewise_linear
   effectiveness_curve_fit_values    = { 0, 1 ; 261.15, 1; 263.15, 0; 1000, 0 }
   effectiveness_curve_fit_variable  = temperature
```

An effectiveness of type user_function may be used to model more complex temperature dependencies; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions. An example is provided of a temperature dependent effectiveness going from 1 to 0 at a temperature of 262.15K linearly over 2K as in the previous example. The input command may be given by:

```
POROSITY_MODEL( "ModelForMelting" ) {
 type                              = constant
 direction_1_permeability          = 1.0
 direction_2_permeability          = 1.0
 direction_3_permeability          = 1.0
 cartesian_permeability_direction  = { 1, 0, 0 ; 0, 1, 0 ; 0, 0, 1 }
 darcy_coefficient                 = 5e+8
 forchheimer_coefficient           = 0
 effectiveness_type                = user_function
 effectiveness_user_function       = usrEffectiveness
 effectiveness_user_values         = {262.15, 2.0}
}
```

where the user-defined function "usrEffectiveness" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrEffectiveness ) ;    /* function prototype   */
Void usrEffectiveness (
 UdfHd   udfHd,    /* Opaque handle for accessing data */
 Real*   outVec,    /* Output vector   */
 Integer   nItems,    /* Number of elements   */
 Integer   vecDim    /* = 1, porosity is a scalar  */
) {
 Integer   elem ;    /* an element counter   */
 Real*   temp ;    /* Temperatures of quadrature point */
 Real*   usrVals ;   /* user values    */
 Real   tmelt ;    /* melting temperature   */
 Real   dmelt ;    /* delta melting         */
 Real   Tp, Tm ;   /* temperature intervals  */
 Real   phi ;    /* porosity effectiveness  */
 udfCheckNumUsrVals( udfHd, 2 ) ;   /* check for error   */
 usrVals= udfGetUsrVals( udfHd ) ;   /* get the user vals   */
 tmelt = usrVals[0] ;     /* User Value - Melting temperature */
 dmelt = usrVals[1] ;     /* User Value - Melting delta  */
 Tp=tmelt+dmelt*0.5 ;
 Tm=tmelt-dmelt*0.5 ;
 // get temperature at integration points
 temp = udfGetElmData( udfHd, UDF_ELM_TEMPERATURE ) ;
 for ( elem = 0 ; elem < nItems ; elem++ ) {
  // compute porosity effectiveness at integration points
  if ( temp[elem] > Tp) {
   phi=0.0 ;
  } else if ( temp[elem] < Tm) {
   phi=1.0 ;
  } else {
   phi=(Tp-temp[elem])/dmelt ;
  }
  outVec[elem] = phi ;
 }
}        /* end of usrEffectiveness */
```

SOLAR_RADIATION_MODEL

Specifies a solar radiation model.

Type

AcuSolve Command

Syntax

```
SOLAR_RADIATION_MODEL("name") {parameters...}
```

Qualifier

User-given name.

Parameters

specular_transmissivity_type (enumerated) [=constant]
Type of the specular transmissivity.

constant or const	Constant specular transmissivity. Requires *specular_transmissivity*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *specular_transmissivity_curve_fit_values* and *specular_transmissivity_curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *specular_transmissivity_curve_fit_values* and *specular_transmissivity_curve_fit_variable*.

specular_transmissivity (enumerated) >=0 <=1 [=0]
Constant value of the specular transmissivity. Used with constant specular transmissivity type.

specular_transmissivity_curve_fit_values (array) [={0,0;90,0}]
A two-column array of independent-variable/emissivity data values. Used with piecewise_linear and cubic_spline specular transmissivity types.

specular_transmissivity_curve_fit_variable (enumerated) [=incidence_angle]
Independent variable of the curve fit. Used with piecewise_linear and cubic_spline specular transmissivity types.

incidence_angle or incident_angle or angle	Incidence angle with respect to the surface.

diffuse_transmissivity_type (enumerated) [=constant]
Type of the diffuse transmissivity.

constant or const	Constant diffuse transmissivity. Requires *diffuse_transmissivity*.

piecewise_linear or linear Piecewise linear curve fit. Requires *diffuse_transmissivity_curve_fit_values* and *diffuse_transmissivity_curve_fit_variable*.

cubic_spline or spline Cubic spline curve fit. Requires *diffuse_transmissivity_curve_fit_values* and *diffuse_transmissivity_curve_fit_variable*.

diffuse_transmissivity *(enumerated) >=0 <=1 [=0]*
Constant value of the diffuse transmissivity. Used with constant diffuse transmissivity type.

diffuse_transmissivity_curve_fit_values *(array) [={0,0;90,0}]*
A two-column array of independent-variable/emissivity data values. Used with piecewise_linear and cubic_spline diffuse transmissivity types.

diffuse_transmissivity_curve_fit_variable *(enumerated) [=incidence_angle]*
Independent variable of the curve fit. Used with piecewise_linear and cubic_spline diffuse transmissivity types.

incidence_angle or incident_angle or angle Incidence angle with respect to the surface.

specular_reflectivity_type *(enumerated) [=constant]*
Type of the specular reflectivity.

constant or const
Constant specular reflectivity. Requires *specular_reflectivity*.

piecewise_linear or linear
Piecewise linear curve fit. Requires *specular_reflectivity_curve_fit_values* and *specular_reflectivity_curve_fit_variable*.

cubic_spline or spline
Cubic spline curve fit. Requires *specular_reflectivity_curve_fit_values* and *specular_reflectivity_curve_fit_variable*.

specular_reflectivity *(enumerated) >=0 <=1 [=0]*
Constant value of the specular reflectivity. Used with constant specular reflectivity type.

specular_reflectivity_curve_fit_values *(array) [={0,0;90,0}]*
A two-column array of independent-variable/emissivity data values. Used with piecewise_linear and cubic_spline specular reflectivity types.

specular_reflectivity_curve_fit_variable *(enumerated) [=incidence_angle]*
Independent variable of the curve fit. Used with piecewise_linear and cubic_spline specular reflectivity types.

incidence_angle or incident_angle or angle Incidence angle with respect to the surface.

diffuse_reflectivity_type *(enumerated) [=constant]*
Type of the diffuse reflectivity.

constant or const	Constant diffuse reflectivity. Requires *diffuse_reflectivity*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *diffuse_reflectivity_curve_fit_values* and *diffuse_reflectivity_curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *diffuse_reflectivity_curve_fit_values* and *diffuse_reflectivity_curve_fit_variable*.

diffuse_reflectivity (enumerated) >=0 <=1 [=0]
Constant value of the diffuse reflectivity. Used with constant diffuse reflectivity type.

diffuse_reflectivity_curve_fit_values (array) [={0,0;90,0}]
A two-column array of independent-variable/emissivity data values. Used with piecewise_linear and cubic_spline diffuse reflectivity types.

diffuse_reflectivity_curve_fit_variable (enumerated) [=incidence_angle]
Independent variable of the curve fit. Used with piecewise_linear and cubic_spline diffuse reflectivity types.

incidence_angle or incident_angle or angle	Incidence angle with respect to the surface.

Description

This command specifies an ideal grey-surface solar radiation model to calculate the solar heat flux. This model is only applicable to solar radiation surfaces, and is not used for the *radiation_heat_flux* variable in the ELEMENT_BOUNDARY_CONDITION command or for radiation surfaces. Four material properties of the surface are specified here: specular transmissivity, diffuse transmissivity, specular reflectivity, and diffuse reflectivity. The surface absorptivity is defined to be one minus the sum of these four. See the SOLAR_RADIATION command for further details.

SOLAR_RADIATION_MODEL commands are referenced by SOLAR_RADIATION_SURFACE commands:

```
SOLAR_RADIATION_MODEL( "my solar radiation model" ) {
   specular_transmissivity_type        = constant
   specular_transmissivity             = 0.2
   diffuse_transmissivity_type         = constant
   diffuse_transmissivity              = 0.15
   specular_reflectivity_type          = constant
   specular_reflectivity               = 0.1
   diffuse_reflectivity_type           = constant
   diffuse_reflectivity                = 0.05
}
SOLAR_RADIATION_SURFACE( "hot wall" ) {
   solar_radiation_mdoel               = "my solar radiation model"
   ...
}
```

Each of the four surface property parameters may be constant, as in the above example, or may be given as a function of a single independent variable. Currently, the only independent variable supported

is the incidence angle. Two types of functions are available: piecewise_linear and cubic_spline curve fits. For example,

```
SOLAR_RADIATION_MODEL( "curve fit specular transmissivity model" ) {
   specular_transmissivity_type                = piecewise_linear
   specular_transmissivity_curve_fit_values    = { 0, 0 ;30, 0 ;90, 0.2 }
   specular_transmissivity_curve_fit_variable  = incidence_angle
   ...
}
```

reduces the specular transmissivity from 0.2 at normal (90 degree) incidence to zero for incidence angles less than 30 degrees. In general, the problem must contain the variable defined by curve fit variable parameter; this is not an issue here since the incidence angle is always available. Each of the curve fit values parameters is a two-column array corresponding to the independent variable and the surface property values. The independent variable values must be in ascending order. The limit point values of the curve fit are used when the curve fit variable falls outside of the curve fit limits.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, such as `spec_trans.fit`:

```
0      0
30      0
90      0.2
```

and read by:

```
SOLAR_RADIATION_MODEL( "curve fit specular transmissivity model" ) {
   specular_transmissivity_type                = piecewise_linear
   specular_transmissivity_curve_fit_values    = Read( "spec_trans.fit" )
   specular_transmissivity_curve_fit_variable = incidence_angle
   ...
}
```

SPECIFIC_HEAT_MODEL

Specifies a specific heat at constant pressure model.

Type

AcuSolve Command

Syntax

SPECIFIC_HEAT_SOURCE("name") {parameters...}

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of the specific heat model.

constant or const	Constant specific heat. Requires *specific_heat*.
piecewise_linear_enthalpy	Piecewise linear curve fit for enthalpy. Requires *curve_fit_values* and *curve_fit_variable*.
piecewise_polynomial_enthalpy	Piecewise polynomial curve fit for enthalpy. Requires *piecewise_polynomial_values* and *piecewise_polynomial* variables.
piecewise_bilinear_enthalpy	Piecewise bilinear curve fit for enthalpy. Requires *bilinear_curve_fit_values*, *bilinear_curve_fit_row_variable* and *bilinear_curve_fit_column_variable*.
cubic_spline_enthalpy	Cubic spline curve fit for enthalpy. Requires *curve_fit_values* and *curve_fit_variable*.
user_function_enthalpy	User-defined function for enthalpy. Requires *user_function*, *user_values* and *user_strings*.

specific_heat or *cp (real) >0 [=1]*

Constant value of the specific heat. Used with constant type.

curve_fit_values or *curve_values (array) [={0,1}]*

A two-column array of independent-variable/enthalpy data values. Used with piecewise_linear_enthalpy and cubic_spline_enthalpy types.

curve_fit_variable or *curve_var (enumerated) [=temperature]*

Independent variable of the curve fit. Used with piecewise_linear_enthalpy and cubic_spline_enthalpy types.

temperature or temp	Temperature.

piecewise_polynomial_values (array) [={}]
Array of values to specify a piecewise polynomial equation. Used with *piecewise_polynomial_enthalpy* type.

piecewise_polynomial_variable (enumerated) [=temperature]
Independent variable of the piecewise polynomial curve fit. Used with *piecewise_polynomial_enthalpy* type.

bilinear_curve_fit_values (array) [={}]
Array of values to specify a piecewise polynomial equation. Used with *piecewise_bilinear_enthalpy* type.

bilinear_curve_fit_row_variable (enumerated) [=temperature]
Independent variable of the rows of the bilinear curve fit table. Used with *piecewise_bilinear_enthalpy* type.

bilinear_curve_fit_column_variable (enumerated) [no default]
Independent variable of the columns of the bilinear curve fit table. Variable can be either pressure or species. Used with *piecewise_bilinear_enthalpy* type.

user_function or user
User-defined function. Requires *user_function*, *user_values* and *user_strings*.

user_values (array) [={}]
Array of values to be passed to the user-defined function. Used with user_function_enthalpy type.

user_strings (list) [={}]
Array of strings to be passed to the user-defined function. Used with user_function_enthalpy type.

latent_heat_type (enumerated) [=none]
Type of the latent heat.

none	No latent heat present.
constant or const	Constant latent heat. Requires *latent_heat*, *latent_heat_temperature* and *latent_heat_temperature_interval*.

latent_heat (real) [=0]
Constant value of latent heat. Used with constant latent heat type.

latent_heat_temperature (real) [=0]
Temperature at which the latent heat is released. Used with constant latent heat type.

latent_heat_temperature_interval (real) >=0 [=0]
Temperature interval over which the latent heat takes effect. Also referred to as the mushy interval. Used with constant latent heat type.

Description

This command specifies a specific heat model for the energy equation. This model is applicable to fluid, solid, and shell element sets.

SPECIFIC_HEAT_MODEL commands are referenced by MATERIAL_MODEL commands, which in turn are referenced by ELEMENT_SET commands:

```
SPECIFIC_HEAT_MODEL( "my cp model" ) {
   type                                = constant
   specific_heat                       = 1005
 }
 MATERIAL_MODEL( "my material model" ) {
   specific_heat_model                 = "my cp model"
   ...
 }
 ELEMENT_SET( "fluid elements" ) {
   material_model                      = "my material model"
   ...
}
```

A constant specific heat model uses a constant value for the entire element set, as in the above example.

When specific heat is not constant, piecewise linear specific heat models of types *piecewise_linear_enthalpy* or *cubic_spline_enthalpy* can be used to define enthalpy (or specific enthalpy) as a function of temperature. Enthalpy is given by:

$$h_2 = h_1 + \int_{T_1}^{T_2} C_p(T)\, dT \tag{98}$$

where T is temperature, C_p is specific heat capacity and h is specific enthalpy.

When specific heat does not vary with respect to temperature, it is often useful to treat it as constant. If so,

$$h_2 = h_1 + C_p(T_2 - T_1) \tag{99}$$

In cases where the specific heat is a function of temperature, the integral above can be numerically evaluated. One possible integration scheme is the trapezoidal rule:

$$h_2 = h_1 + \frac{(C_{p1} + C_{p2})}{2}(T_2 - T_1) \tag{100}$$

Note: This integration (irrespective of the method use) must be performed prior to the CFD calculation as AcuSolve does not currently offer any capabilities in this regard.

The *curve_fit_values* is a two-column array corresponding to the independent variable (temperature in this case) and enthalpy. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, say enthalpy.fit, and read by:

```
SPECIFIC_HEAT_MODEL( "ice" ) {
   type                  = piecewise_linear_enthalpy
   curve_fit_values      = Read( "enthalpy.fit" )
   curve_fit_variable    = temperature
}
```

The *piecewise_polynomial_enthalpy* type is used to specify the high-order multiple polynomial temperature functions for specific heat capacity in different temperature ranges. The specific heat is calculated by AcuSolve by calculating the derivative of the enthalpy polynomial with respect to temperature. For example, if the temperature dependent enthalpy equation is given by:

$$H^o(T) = \underbrace{a_0 R}_{A0} + \underbrace{a_1 R}_{A1} T + \underbrace{\frac{a_2 R}{2}}_{A2} T^2 + \underbrace{\frac{a_3 R}{3}}_{A3} T^3 + \underbrace{\frac{a_4 R}{4}}_{A4} T^4 + \underbrace{\frac{a_5 R}{5}}_{A5} T^5 \tag{101}$$

for

$$T_{a_min} < T < T_{a_max} \tag{102}$$

and

$$H^o(T) = \underbrace{b_0 R}_{B0} + \underbrace{b_1 R}_{B1} T + \underbrace{\frac{b_2 R}{2}}_{B2} T^2 + \underbrace{\frac{b_3 R}{3}}_{B3} T^3 + \underbrace{\frac{b_4 R}{4}}_{B4} T^4 + \underbrace{\frac{b_5 R}{5}}_{B5} T^5 \tag{103}$$

for

$$T_{b_min} < T < T_{b_max} \tag{104}$$

where R is the universal gas constant.

Note: Most of the references for the thermodynamic properties of gases provide the coefficients $a_0, a_1, a_2, \ldots$ which need to be multiplied with the appropriate values shown above to arrive at the values required by AcuSolve.

```
SPECIFIC_HEAT_MODEL( "H" ) {
     type = piecewise_polynomial_enthalpy
     piecewise_polynomial_variable = temperature
     piecewise_polynomial_values = { Ta_min, Ta_max, A0, A1, A2, A3, A4, A5;
         Tb_min, Tb_max, B0, B1, B2, B3, B4,  B5}
}
```

The first entry of *piecewise_polynomial_values* is the minimum temperature, followed by the maximum temperature to indicate the applicable temperature range. Next, the polynomial coefficients are entered to define the first polynomial equation. The second set of *piecewise_polynomial_values* is separated with the semi-colon. The order of polynomial coefficient in *piecewise_polynomial_values* is from lower to higher, starting from the index 0. Negative indices of temperature are not supported by AcuSolve.

A *piecewise_bilinear_enthalpy* type defines a piecewise bilinear curve fit as a function of two independent variables. *bilinear_curve_fit_row_variable* and *bilinear_curve_fit_column_variable* define the two independent variables.

The format of the interpolation table is as follows:

```
bilinear_curve_fit_data =
{   0,           cVal1,     cVal2,     cVal3,     ... ;
Temperature 1, h(T1,c1), h(T1,c2), h(T1,c3), ... ;
Temperature 2, h(T2,c1), h(T2,c2), h(T2,c3), ... ;
Temperature 3, h(T3,c1), h(T3,c2), h(T3,c3), ... ;
...,           ...,       ...,       ... ,      ... ;
}
```

The first entry in the table must be 0, then followed by the values of the column independent variable. The first entry of each row must be the temperature value followed by the corresponding enthalpy. The row entries are comma separated with the semi-colon separating each of the rows.

A specific heat of type user_function_enthalpy may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider a version of the piecewise linear latent heat model discussed above. The model below improves on this one by adding two "buffer" regions on either side of the latent heat transition. These are used to smooth the transition and make the specific heat continuous with temperature. Unlike for the curve fit types AcuSolve has no way of calculating the specific heat from the enthalpy, so both must be supplied by the user defined function. The specific heat is needed to form the appropriate contribution to the left-hand side matrix. An error is issued if it is not returned. Only Jacobians with respect to temperature are supported currently. The enthalpy may also be a function of pressure and species, but extreme caution must be taken since Jacobians with respect to these variables are not supported. The input command may be given by:

```
SPECIFIC_HEAT_MODEL( "UDF ice" ) {
   type                = user_function_enthalpy
   user_function       = "usrSpecHeat"
   user_values          = { 3000,          # reference density
                            3.33e5,       # reference temperature
                          273,            # expansivity
                            2 }           # temperature interval
}
```

where the user-defined function "usrSpecHeat" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrSpecHeat ) ;                                          /* function
 prototype                */
    Void usrSpecHeat (
    UdfHd                  udfHd,                                      /* Opaque handle
 for accessing data */
    Real*                  outVec,                                     /* Output vector
                  */
    Integer                nItems,                                     /* Number of
 elements                */
    Integer                vecDim                                      /* = 1
                  */
) {
    Integer                elem ;                                      /* element index
                  */
    Real                   cp0 ;                                       /* specific heat
                  */
    Real                   delTemp ;                                   /* temperature
 interval                */
    Real                   dt2 ;                                       /* delTemp / 2
                  */
    Real                   enpy ;                                      /* partial
 enthalpy                */
    Real                   latentHeat ;                                /* latent heat
                  */
    Real                   lh2 ;                                       /* latent heat /
 2                */
    Real                   lh4 ;                                       /* latent heat /
 4                */
```

```
    Real                    lhDt ;                                      /* latent heat /
delTemp              */
    Real                    refTemp ;                                   /* reference
 temperature            */
    Real                    tmp ;                                       /* a temporary
 temperature          */
    Real*                   temp ;                                      /* a temperature
                     */
    Real*                   cp ;                                        /* specific heat
                     */
    Real*                   usrVals ;                                   /* user supplied
 values              */

udfCheckNumUsrVals( udfHd, 4 ) ;                                        /* check for
 error                  */
    usrVals                 = udfGetUsrVals( udfHd ) ;
    cp0                     = usrVals[0] ;
    latentheat              = usrVals[1] ;
    refTemp                 = usrVals[2] ;
    delTemp                 = usrVals[3] ;
    if ( delTemp <= 0 ) {
       udfSetError( udfHd,
           "temperature interval <%g> is not positive", delTemp ) ;
    }
    dt2                     = delTemp / 2 ;
    lh2                     = latentHeat / 2 ;
    lh4                     = latentHeat / 4 ;
    lhDt                    = latentHeat / delTemp ;
    temp                    = udfGetElmData( udfHd, UDF_ELM_TEMPERATURE ) ;

    /* Jacobian of enthalpy with respect to temp (same as specific heat) */
    cp      = udfGetElmJac( udfHd, UDF_ELM_JAC_TEMPERATURE ) ;
    for ( elem = 0 ; elem < nItems ; elem++ ) {
        tmp                 = (temp[elem] - refTemp) / dt2 ;
        if ( tmp < -1.5 ) tmp = -1.5 ;
        if ( tmp > +1.5 ) tmp = +1.5 ;
        if ( tmp < -0.5 ) {
            tmp             = 1.5 + tmp ;
            enpy            = cp0 * temp[elem] + lh4 * tmp * tmp ;
            cp[elem]        = cp0 + lhDt * tmp ;
        } else if ( tmp <= 0.5 ) {
            tmp             = 0.5 + tmp ;
            enpy            = cp0 * temp[elem] + lh2 * tmp + lh4 ;
            cp[elem]        = cp0 + lhDt ;
        } else {
            tmp             = 1.5 - tmp ;
            enpy            = cp0 * temp[elem] + latentHeat - lh4 * tmp * tmp ;
            cp[elem]        = cp0 + lhDt * tmp ;
        }
        outVec[elem]        = enpy ;
    }
} /* end of usrSpecHeat() */
```

The dimensions of the returned enthalpy vector, outVec, and the Jacobian vector, cp, are the number of elements.

A latent heat of formation may be added directly to any specific heat model. For example,

```
SPECIFIC_HEAT_MODEL( "ice" ) {
    type                            = constant
    specific_heat                   = 3000
    latent_heat_type                = constant
```

```
    latent_heat                         = 3.33e5
    latent_heat_temperature             = 273
    latent_heat_temperature_interval    = 2
}
```

Here, a constant specific heat of 3000 is used except at a temperature of 273, where the enthalpy is increased by $3.33x10^5$. The jump in enthalpy is spread over two degrees, centered at 273. This example models the same material as the piecewise_linear_enthalpy type example above. One advantage of using the latent heat parameters is that a more stable transition form is internally implemented for the jump in the enthalpy. This form uses extra buffer regions like in the user-defined function above.

SURFACE_TENSION_MODEL

Specifies a surface tension model for free surfaces.

Type

AcuSolve Command

Syntax

```
SURFACE_TENSTION_MODEL("name") {parameters...}
```

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of the surface tension.

constant or const	Constant surface tension. Requires *surface_tension*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
user_function or user	User-defined function. Requires *user_function*, *user_values* and *user_strings*.

surface_tension (real) >=0 [=0]

Constant value of the surface tension. Used with constant type.

curve_fit_values or curve_values (array) [={0,0}]

A two-column array of independent-variable/surface tension data values. Used with piecewise_linear and cubic_spline types.

curve_fit_variable or curve_var (enumerated) [=temperature]

Independent variable of the curve fit. Used with piecewise_linear and cubic_spline types.

temperature or temp	Temperature.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.

species_6 or spec6 Species 6.

species_7 or spec7 Species 7.

species_8 or spec8 Species 8.

species_9 or spec9 Species 9.

user_function or *user* *(string) [no default]*
Name of the user-defined function. Used with user_function type.

user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function type.

user_strings *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with user_function type.

multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the viscosity. If none, no scaling is performed.

Description

Surface tension adds a traction to a free surface of the form

$$2H\sigma\boldsymbol{n} + \nabla_s\sigma \qquad (105)$$

where H is the surface curvature, **n** is the outward normal, ∇_s is the surface gradient, and σ is the surface tension defined by this command. The formulation used in AcuSolve removes the necessity of computing the surface curvature and the surface gradient of the surface tension through an integration by parts procedure. However, the contact angle then explicitly appears, which is modeled separately. See the CONTACT_ANGLE_MODEL command. For example, a constant surface tension may be defined by:

```
SURFACE_TENSION_MODEL( "air/water" ) {
   type                              = constant
   surface_tension                   = 0.07
}
```

SURFACE_TENSION_MODEL commands are referenced by SIMPLE_BOUNDARY_CONDITION and FREE_SURFACE commands, for example:

```
SIMPLE_BOUNDARY_CONDITION( "free surface" ) {
   surfaces              = Read( "free_surface.ebc" )
   shape                 = three_node_triangle
   element_set           = "water"
   type                  = free_surface
   surface_tension_model = "water/air"
   contact_angle_model   = "water/air @ smooth aluminum"
}
```

and

```
FREE_SURFACE( "free surface" ) {
   surfaces               = Read( "free_surface.ebs" )
   shape                  = three_node_triangle
```

```
   element_set             = "water"
   surface_tension_model   = "water/air"
   contact_angle_model     = "water/air @ smooth aluminum"
}
```

Surface tension models of types piecewise_linear and cubic_spline may be used to define surface tension as a function of a single independent variable. For example,

```
SURFACE_TENSION_MODEL( "curve fit surface tension model" ) {
   type                = piecewise_linear
   curve_fit_values    = { 273, 0.07 ;
                           323, 0.07 ;
                           373, 0.06 ;
                           423, 0.05 ; }
   curve_fit_variable   = temperature
}
```

defines surface tension as a function of temperature. In general, the problem must contain the variable defined by *curve_fit_variable*. The *curve_fit_values* parameter is a two-column array corresponding to the independent variable and the surface tension values. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, such as surface_tension.fit:

```
273      0.07
323      0.07
373      0.06
423      0.05
```

and read by:

```
SURFACE_TENSION_MODEL( "curve fit surface tension model" ) {
   type                = piecewise_linear
   curve_fit_values    = Read( "surface_tension.fit" )
   curve_fit_variable  = temperature
}
```

A surface tension model of type user_function may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

The *multiplier_function* parameter may be used to uniformly scale the surface tension values. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file. For example, a ramped surface tension may be specified by:

```
SURFACE_TENSION_MODEL( "curve fit surface tension model" ) {
   type                = piecewise_linear
   curve_fit_values    = Read( "surface_tension.fit" )
   curve_fit_variable = temperature
}
MULTIPLIER_FUNCTION( "ramped" ) {
   type                = piecewise_linear
   curve_fit_values    = { 1, 0.1 ; 10, 1 }
   curve_fit_variable = time_step
}
```

VISCOELASTIC_MODEL

Specifies a viscoelastic model for the flow equation.

Type

AcuSolve Command

Syntax

VISCOELASTIC_MODEL("name") {parameters...}

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of the viscoelastic model.

none	No viscoelastic model.
oldroyd_b	Oldroyd-B model describing the viscoelastic behavior of fluid. Requires *time_constant* and *viscosity_model*.
giesekus	Giesekus model describing the viscoelastic behavior of fluid. Requires *time_constant*, *giesekus_mobility_factor* and *viscosity_model*.
phan_thien_tanner or ptt	Requires *time_constant*, *phan_thien_tanner_extensibility_factor*, *phan_thien_tanner_differential_ratio*, and *viscosity_model*.
user_function_viscoelastic	User-defined function. Requires *user_function*, *user_values* and *user_strings*.

viscosity_model (string) [none]

Refers to viscosity of the polymer. Valid models are described in the VISCOSITY_MODEL command. Used with oldroyd_b, giesekus, and phan_thien_tanner types.

time_constant (real) >=0 [=0]

Relaxation time. Used with oldroyd_b, giesekus, and phan_thien_tanner types.

time_constant _multiplier_function (string) [none]

Multiplier function used for scaling the time constant. Used with oldroyd_b, giesekus, and phan_thien_tanner types.

giesekus_mobility_factor or giesekus_alpha >=0 [=0]

Mobility factor that controls extensional viscosity and the ratio of second normal stress difference to the first normal stress. Shear thinning behavior is observed when the mobility factor is greater than zero. Used with giesekus model.

phan_thien_tanner_extensibility_factor or *ptt_alpha* *(real) >=0 [=0]*
Controls the level of shear thinning. Used with phan_thien_tanner model.

phan_thien_tanner_differential_ratio or *ptt_xi* *(real) >=0 <=2 [=0]*
Factor that controls extensional viscosity and accounts for the slip between the molecular network and the continuum medium. Used with phan_thien_tanner model.

user_function or *user* *(string) [no default]*
Name of the user-defined function. Used with user_function type.

user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function type.

user_strings *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with user_function type.

multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the viscosity. If none, no scaling is performed.

Description

This command specifies a viscoelastic model for the flow (momentum) equations. This model is only applicable to fluid element sets.

VISCOELASTIC_MODEL commands are referenced by MATERIAL_MODEL commands, which in turn are referenced by ELEMENT_SET commands:

```
VISCOELASTIC_MODEL( "my viscoelastic model" ) {
   type             = oldroyd_b
   time_constant    = 0.1
   viscosity_model  "polymer viscosity"
}
VISCOSITY_MODEL( "polymer viscosity" ) {
   type            = constant
   viscosity       = 0.888888
}
```

In the case of viscoelastic fluids the stress depends on both instantaneous strain rate and strain history. For an Upper Convective Maxwell fluid, the constitutive equation is given by:

$$\begin{aligned} \boldsymbol{\sigma} &= -p\mathbf{I} + \mathbf{T} \\ \mathbf{T} &= \mathbf{T}_1 + \mathbf{T}_2 \\ \mathbf{T}_1 + \lambda \overset{\nabla}{\mathbf{T}}_1 + \mathbf{f}_c\left(\mathbf{T}_1, \varepsilon(\mathbf{u})\right) + \mathbf{f}_d\left(\mathbf{T}_1\right) &= 2\mu_1 \varepsilon(\mathbf{u}) \\ \mathbf{T}_2 &= 2\mu_2 \varepsilon(\mathbf{u}) \\ \mu_1 + \mu_2 &= \mu \end{aligned} \qquad (106)$$

Here, σ denotes the stress tensor, p is the fluid pressure, T represents the viscous stress tensor, λ is the relaxation time, μ_1 is the viscosity of the elastic polymer dissolved in a viscous solvent with viscosity μ_2.

The coefficient $\mathbf{f}_c$ represents the rate at which the stress builds up and $\mathbf{f}_d$ represents the rate at which stress decays. $\overset{\nabla}{\mathbf{T}}$ is the upper-convected derivative and $\dot{\varepsilon}(\mathbf{u})$ is the rate-of-strain tensor:

$$\overset{\nabla}{\mathbf{T}} = \frac{\partial \mathbf{T}}{\partial t} + \mathbf{u} \bullet \nabla \mathbf{T} - \left(\nabla \mathbf{u} \mathbf{T} + \mathbf{T} \left(\nabla \mathbf{u} \right)^T \right)$$

$$\varepsilon(\mathbf{u}) = \frac{1}{2} \left(\nabla \mathbf{u} + \left(\nabla \mathbf{u} \right)^T \right) \qquad (107)$$

The solvent viscosity μ_2 is defined by the parameter *viscosity_model*. For a complete description of viscosity models, refer to VISCOSITY_MODEL.

Oldroyd-B Model

The Oldroyd-B model can be obtained by defining:

$$\mathbf{f}_c \left(\mathbf{T}_1, \varepsilon(\mathbf{u}) \right) = \mathbf{f}_d \left(\mathbf{T}_1 \right) = 0 \qquad (108)$$

Here you define only the relaxation time, given by *time_constant* and polymer viscosity, given by *viscosity*.
For example,

```
VISCOELASTIC_MODEL( "Oldroyd model" ) {
   type            = oldroyd_b
   time_constant   = 1.0
   viscosity_model = "Oldroyd model:viscoelastic"
}
VISCOSITY_MODEL( "Oldroyd model:viscoelastic" ) {
    type           = constant
    viscosity      = 0.8888888888  # kg/m-sec
}
```

Giesekus Model

This model can be obtained by defining

$$\mathbf{f}_c(\mathbf{T}_1, \varepsilon(\mathbf{u})) = \mathbf{0},$$
$$\mathbf{f}_d(\mathbf{T}_1) = \frac{\alpha \lambda}{\mu_1} \mathbf{T}_1 \mathbf{T}_1 \qquad (109)$$

Here you define the relaxation time, given by *time_constant*, *polymer viscosity*, given by *viscosity*, and the mobility factor *a* given by *giesekus_mobility_factor* or *giesekus_alpha*.
For example,

```
VISCOELASTIC_MODEL( "giesekus" ) {
   type            = giesekus
   viscosity_model = "polymer viscosity"
   time_constant   = 0.1
   giesekus_alpha  = 0.1
}
VISCOSITY_MODEL( "polymer viscosity" ) {
   type            = constant
   viscosity       = 0.888889
}
```

Phan-Thien-Tanner Model

This model can be obtained by defining:

$$\mathbf{f}_c(\mathbf{T}_1, \varepsilon(\mathbf{u})) = \lambda\xi\left(\varepsilon(\mathbf{u})\cdot\mathbf{T}_1 + \mathbf{T}_1\cdot\varepsilon(\mathbf{u})\right),$$
$$\mathbf{f}_d(\mathbf{T}_1) = \exp\left(\frac{\alpha\lambda}{\mu_1}\mathrm{tr}(\mathbf{T}_1) - \mathbf{I}\right)\mathbf{T}_1 \qquad (110)$$

Here you define the relaxation time, given by *time_constant*, *polymer viscosity*, given by *viscosity*, the extensibility factor, a, given by *phan_thien_tanner_extensibility_factor* or *ptt_alpha*, and diffential ratio ς given by *phan_thien_tanner_differential_ratio* or *ptt_xi*. For example,

```
VISCOELASTIC_MODEL( "ptt" ) {
   type              = ptt
   viscosity_model   = "polymer viscosity"
   time_constant     = 1.0
   ptt_alpha         = 0.25
   ptt_xi            = 0.1
}
VISCOSITY_MODEL( "polymer viscosity" ) {
   type              = constant
   viscosity         = 0.888889
}
```

VISCOSITY_MODEL

Specifies a viscosity model for the flow equation.

Type

AcuSolve Command

Syntax

```
VISCOSITY_MODEL("name") {parameters...}
```

Qualifier

User-given name.

Parameters

type (enumerated) [=none]

Type of the viscosity.

constant or const	Constant viscosity. Requires *viscosity*.
ramped or ramp	Ramp the viscosity to a constant value. Requires *viscosity*.
power_law	Non-Newtonian power law viscosity model. Requires *power_law_viscosity*, *power_law_time_constant*, *power_law_index* and *power_law_lower_strain_rate*.
bingham	Non-Newtonian Bingham viscosity model. Requires *bingham_viscosity*, *bingham_yield_stress*, *bingham_stress_growth_exponent*, *bingham_time_constant*, *bingham_index* and *bingham_infinite_shear_viscosity*.
carreau	Non-Newtonian Carreau-Yasuda viscosity model. Requires *carreau_zero_shear_viscosity*, *carreau_infinite_shear_viscosity*, *carreau_time_constant*, *carreau_index* and *carreau_transition_index*.
gas_kinetic	Gas kinetic viscosity model for gases only. Requires GAS_KINETIC_MODEL.
sutherland	Sutherland temperature dependent viscosity model. Requires *sutherland_reference_viscosity*, *sutherland_reference_temperature*, *sutherland_constant*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *curve_fit_values* and *curve_fit_variable*.

user_function or user User-defined function. Requires *user_function*, *user_values* and *user_strings*.

viscosity or *visc* *(real) >=0 [=0]*
Constant value of the dynamic viscosity. Used with constant and ramped types.

power_law_viscosity *(real) >=0 [=0]*
Power law model reference viscosity. Used with power_law type.

power_law_time_constant *(real) >=0 [=0]*
Power law model time constant. Used with power_law type.

power_law_index *(real) [=1]*
Power law model power index. Used with power_law type.

power_law_lower_strain_rate *(real) >=0 [=0]*
Power law model lower limit of strain rate. Used with power_law type.

bingham_viscosity *(real) >0 [=1]*
Bingham model reference viscosity. Used with bingham type.

bingham_yield_stress *(real) >=0 [=0]*
Bingham model yield stress. Used with bingham type.

bingham_stress_growth_exponent *(real) >0 [=500]*
Bingham model unyielded stress region growth rate exponent. Used with bingham type.

bingham_time_constant *(real) >0 [=1]*
Bingham model power law time constant. Used with bingham type.

bingham_index *(real) [=1]*
Bingham model power law index. Used with bingham type.

bingham_infinite_shear_viscosity *(real) [=0]*
Bingham model minimum viscosity. Used with bingham type.

carreau_zero_shear_viscosity *(real) >=0 [=0]*
Carreau model viscosity at zero shear rate. Used with carreau type.

carreau_infinite_shear_viscosity *(real) >=0 [=0]*
Carreau model viscosity at infinite shear rate. Used with carreau type.

carreau_time_constant *(real) >=0 [=0]*
Carreau model power law time constant. Used with carreau type.

carreau_index *(real) [=1]*
Carreau model power law index. Used with carreau type.

carreau_transition_index *(real) >0 [=2]*
Carreau model transition index. Used with carreau type.

carreau_temperature_dependence_type *(enumerated) [=none]*
Types to add the temperature dependence terms to the Carreau Yasuda viscosity material to account for shear heating due to viscous dissipation. Used with carreau type when *temperature* = advective_diffusive under the EQUATION command and *viscous_heating* = on in ELEMENT_SET.

Arrhenius	Arrhenius model. Requires *activation energy*.
William_Landel_Ferry	William Landel Ferry (WLF) model. Requires *William_Landel_Ferry_coefficient_C1*, *William_Landel_Ferry_coefficient_C2*, *glass_transition_temperature*.
exponential	Exponential model. Requires *exponential_reference_temperature* and *exponential_beta_coefficient*.

activation_energy *(real) >0 [=16628]*
Arrhenius model constant to define the temperature sensitivity. Used with carreau type and *carreau_temperature_dependence_type* = Arrhenius.

William_Landel_Ferry_coefficient_C1 *(real) > 0 [=17.44]*
William-Landel-Ferry (WLF) model constant. Used with carreau type and *carreau_temperature_dependence_type* = William_Landel_Ferry.

William_Landel_Ferry_coefficient_C2 *(real) > 0 [=51.6]*
William-Landel-Ferry (WLF) model constant. Used with carreau type and *carreau_temperature_dependence_type* = William_Landel_Ferry.

glass_transition_temperature *(real) > 0 [=320]*
Glass transition temperature used in the William-Landel-Ferry (WLF) model. Used with carreau type and *carreau_temperature_dependence_type* = William_Landel_Ferry.

exponential_reference_temperature *(real) > 0 [=350]*
Reference temperature defined in the exponential model. Used with carreau type and *carreau_temperature_dependence_type* = exponential.

exponential_beta_coefficient *(real) > 0 [=0.005]*
Exponential model constant. Used with carreau type and *carreau_temperature_dependence_type* = exponential.

sutherland_reference_viscosity *(real) > = 0 [=0.01716]*
Sutherland model reference viscosity at *sutherland_reference_temperature*. Used with sutherland type.

sutherland_reference_temperature *(real) > 0 [=273]*
Sutherland model reference temperature corresponding to *sutherland_reference_viscosity*. Used with sutherland type.

sutherland_constant *(real) > 0 [=111]*
Sutherland model constant. Used with sutherland type.

curve_fit_values *or* *curve_values* *array [={0,0}]*
A two-column array of independent-variable/viscosity data values. Used with *piecewise_linear* and *cubic_spline* types.

curve_fit_variable *or* *curve_var* *(enumerated) [=temperature]*
Independent variable of the curve fit. Used with *piecewise_linear* and *cubic_spline* types.

x_coordinate or xcrd	X-component of coordinates.
y_coordinate or ycrd	Y-component of coordinates.
z_coordinate or zcrd	Z-component of coordinates.
x_reference_coordinate or xrefcrd	X-component of reference coordinates.
y_reference_coordinate or yrefcrd	Y-component of reference coordinates.
z_reference_coordinate or zrefcrd	Z-component of reference coordinates.
pressure or pres	Pressure.
temperature or temp	Temperature.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.
species_7 or spec7	Species 7.
species_8 or spec8	Species 8.
species_9 or spec9	Species 9.
strain_rate_second_invariant or strain_i2	Second invariant of the strain rate tensor.

`user_function` or `user` (string) [no default]
Name of the user-defined function. Used with user_function type.

`user_values` (array) [={}]
Array of values to be passed to the user-defined function. Used with user_function type.

`user_strings` (list) [={}]
Array of strings to be passed to the user-defined function. Used with user_function type.

`multiplier_function` (string) [=none]
User-given name of the multiplier function for scaling the viscosity. If none, no scaling is performed.

Description

This command specifies a dynamic viscosity model for the flow (momentum) equation. This model is only applicable to fluid element sets.

`VISCOSITY_MODEL` commands are referenced by `MATERIAL_MODEL` commands, which in turn are referenced by `ELEMENT_SET` commands:

```
VISCOSITY_MODEL( "my viscosity model" ) {
   type                                = constant
   viscosity                           = 1.781e-5
 }
 MATERIAL_MODEL( "my material model" ) {
   viscosity_model                     = "my viscosity model"
   ...
 }
 ELEMENT_SET( "fluid elements" ) {
   material_model                      = "my material model"
   medium                              = fluid
   ...
}
```

The viscous stress tensor $\tau = [\tau_{ij}]$ is given by:

$$\tau = 2\mu\nabla_s \boldsymbol{u} \tag{111}$$

where μ is the dynamic viscosity and $\nabla_s u$ is the symmetric part of the strain-rate tensor.

A constant type applies a spatially constant viscosity model, as in the above example. This is the simplest case of a Newtonian model for the stress tensor.

The power_law type defines a non-Newtonian viscosity model in which the dynamic-viscosity is a function of the velocity field:

$$\mu = \mu_0 \left[\eta^2 \max\left(I_2, S_{\min}^2\right)\right]^{(n-1)/2} \tag{112}$$

where μ_0 is the reference viscosity, given by *power_law_viscosity*; η is the time constant, given by *power_law_time_constant*; n is the power index, given by *power_law_index*; S_{min} is the lower limit of the strain rate, given by *power_law_lower_strain_rate*; and I_2 is the second invariant of the strain rate tensor:

$$I_2 = 2\nabla_s u : \nabla_s u \tag{113}$$

One of the commonly used models for characterizing non-Newtonian viscosity fluids is the power law model. This model describes the behavior of shear thinning fluids, where the viscosity decreases as the shear rate increases. The relationship between the viscosity μ and the shear rate $\dot{\gamma}$ can be expressed as follows:

$$\mu = m\,\dot{\gamma}^{n-1} \tag{114}$$

where μ represents the viscosity (Pa sec), $\dot{\gamma}$ is the shear rate (1/sec), n is the power law index; m is the flow consistency.

To determine the power law index (n) and the flow consistency (m), curve fitting is performed using the rheometer measurement.

Note: The power-law equation (Eq. 2) used in AcuSolve is different from the typical power model, requiring a conversion process to obtain the input parameters for the AcuSolve's Power Law viscosity model (Eq. 2).

For example, if the viscometer measurement yields the following curve-fit result

$$\mu = 200\,\dot{\gamma}^{-1.05} \tag{115}$$

The power index ($n = -0.05$) can be obtained from $-1.05 = n - 1$. The power law viscosity can be calculated by substituting 200 to the equation 2 as shown below. It is important to account for a factor of 2 due to definition of the second invariant of the strain tensor:

$$200 = \mu_0 2^{(n-1)/2} \tag{116}$$

Thus, power law viscosity is $\mu_0 = 287.8$.

The power law time constant η is typically set to 1, and the maximum shear rate should be 0.1 or less to ensure that it corresponds to the maximum viscosity.

```
VISCOSITY_MODEL( "Basic Power Law" ) {
  type = power_law
  power_law_viscosity = 287.8
  power_law_time_constant = 1.0
  power_law_index = -0.05
  power_law_lower_strain_rate = 0.1
}
```

The bingham type defines another non-Newtonian viscosity model in which Papanastasiou's unyielded stress region and power law modifications are added to the basic Bingham viscosity model:

$$\mu = \mu_\infty + \left(\mu_0 - \mu_\infty\right)(\eta g)^{n-1} + \frac{\tau_Y}{g}\left(1 - e^{-Mg}\right) \tag{117}$$

where μ_0 is the reference viscosity, given by *bingham_viscosity*; η is the time constant, given by *bingham_time_constant*; n is the power index, given by bingham_index; τ_Y is the yield stress, given by *bingham_yield_stress*; M is the unyielded stress region growth rate exponent, given by *bingham_stress_growth_exponent*; μ_∞ is the viscosity at infinite shear rate (or equivalently the minimum viscosity), given by *bingham_infinite_shear_viscosity*; and g is the square root of the second invariant of the strain rate tensor:

$$g = \sqrt{2\nabla_s u : \nabla_s u} \tag{118}$$

This may be used as a basic Bingham viscosity model, $\mu = \mu_0 + \tau_Y / g$, as follows:

```
VISCOSITY_MODEL( "Basic Bingham Law" ) {
    type                    = bingham
    bingham_viscosity       = 0.001
    bingham_yield_stress    = 10
}
```

Papanastasiou's unyielded stress region growth rate exponent modification limits the viscosity at low shear rates to $\mu = \mu_0 + \tau_Y M$ and provides for a smooth transition from the unyielded to the yielded stress regions. For example:

```
VISCOSITY_MODEL( "Bingham Law with Papanastasiou exponent" ) {
    type                           = bingham
    bingham_viscosity              = 0.001
    bingham_yield_stress           = 10
    bingham_stress_growth_exponent = 1000
}
```

To add a power law model modification to the viscosity model:

```
VISCOSITY_MODEL( "Bingham Law with Papanastasiou exponent and power law" ) {
    type                           = bingham
    bingham_viscosity              = 0.001
    bingham_yield_stress           = 10
    bingham_stress_growth_exponent = 1000
    bingham_time_constant          = 1
    bingham_index                  = 0.75
}
```

The carreau type defines a third non-Newtonian viscosity model:

$$\mu = \mu_\infty + (\mu_0 - \mu_\infty)(1 + (\eta g)^a)^{(n-1)/a} \tag{119}$$

where μ_0 is the viscosity at zero shear rate, given by *carreau_zero_shear_viscosity*; μ_∞ is the viscosity at infinite shear rate, given by *carreau_infinite_shear_viscosity*; η is the time constant, given by *carreau_time_constant*; n is the power index, given by *carreau_index*; a is the transition index, given by *carreau_transition_index*; and g is the square root of the second invariant of the strain rate tensor (see above). In this form this model is known as the Carreau-Yasuda model. If a = 2, then the classic Carreau model is recovered.

For example,

```
VISCOSITY_MODEL( "Carreau Model" ) {
    type                              = carreau
    carreau_zero_shear_viscosity      = 681.8
    carreau_infinite_shear_viscosity  = 0.17
    carreau_time_constant             = 31.7
    carreau_index                     = 0.11
    carreau_transition_index          = 2
}
```

The sutherland type defines a temperature dependent viscosity model according to the following equation:

$$\mu = \mu_0 \left(\frac{T}{T_0}\right)^{3/2} \frac{T_0 + S}{T + S} \tag{120}$$

where μ_0 is the *sutherland_reference_viscosity* at the specified *sutherland_reference_temperature* (T_0), T is the model temperature and S is the *sutherland_constant*.

For example,

```
VISCOSITY_MODEL( "Sutherland Model" ) {
type = sutherland
sutherland_reference_viscosity = 1.716e-5
sutherland_reference_temperature = 273.0
sutherland_constant = 111
}
```

A ramped viscosity model is the same as the constant one, except that the viscosity is ramped down from 1000 times the viscosity value at time step one to the viscosity value at time step ten. For example,

```
VISCOSITY_MODEL( "ramped viscosity model" ) {
    type        = ramped
    viscosity   = 1.781e-5
}
```

starts with a viscosity of 1.781×10^{-2} at time step one and reduces the viscosity until time step 10, at which time a viscosity of 1.781×10^{-5} is used.

Viscosity models of types piecewise_linear and cubic_spline may be used to define viscosity as a function of a single independent variable. For example,

```
VISCOSITY_MODEL( "curve fit viscosity model" ) {
    type                = piecewise_linear
    curve_fit_values    = { 273, 1.72e-5 ;
                            323, 1.95e-5 ;
                            373, 2.17e-5 ;
                            423, 2.38e-5 ; }
    curve_fit_variable  = temperature
}
```

defines viscosity as a function of temperature. In this case, the problem must contain a temperature equation; see the `EQUATION` command. The *`curve_fit_values`* parameter is a two-column array corresponding to the independent variable and the viscosity values. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *`curve_fit_variable`* falls outside of the curve fit limits.

The *`curve_fit_values`* data may be read from a file. For the above example, the curve fit values may be placed in a file, such as `viscosity.fit`:

```
273       1.72e-5
323       1.95e-5
373       2.17e-5
423       2.38e-5
```

and read by:

```
VISCOSITY_MODEL( "curve fit viscosity model" ) {
    type                = piecewise_linear
    curve_fit_values    = Read( "viscosity.fit" )
    curve_fit_variable  = temperature
}
```

A viscosity of type *user_function* may be used to model more complex behaviors; the see *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider a simplified version of the non-Newtonian power law viscosity model discussed above. The input command may be given by:

```
VISCOSITY_MODEL( "UDF viscosity model" ) {
    type           = user_function
    user_function  = "usrViscosityExample"
    user_values    = { 9000,   # reference viscosity
                       12.5,   # time constant
                        0.48 }  # power index
}
```

where the user-defined function "usrViscosityExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrViscosityExample ) ;                                  /* function
 prototype                */
Void usrViscosityExample (
    UdfHd         udfHd,                                                /* Opaque
 handle for accessing data */
 Real*         outVec,                                                /* Output
 vector                     */
 Integer       nItems,                                                /* Number of
 elements                */
 Integer       vecDim                                                 /* = 1
                    */
) {
    Integer       elem ;                                               /* an
 element counter               */
    Real          exp ;                                                /* exponent
 ( (index-1)/2 )         */
    Real          index ;                                              /* power
 index                      */
    Real          invar2 ;                                             /* second
 invariant of strain rate  */
    Real          tConst ;                                             /* time
 constant                     */
    Real          tConst2 ;                                            /* scaling
 factor (tConst ^ 2)      */
    Real          visc0 ;                                              /*
 reference viscosity             */
    Real*         gradV ;                                              /* gradient
 of velocity             */
    Real*         u11 ;                                                /* partial
 u1 / partial x1          */
    Real*         u12 ;                                                /* partial
 u1 / partial x2          */
    Real*         u13 ;                                                /* partial
 u1 / partial x3          */
    Real*         u21 ;                                                /* partial
 u2 / partial x1          */
    Real*         u22 ;                                                /* partial
 u2 / partial x2          */
    Real*         u23 ;                                                /* partial
 u2 / partial x3          */
    Real*         u31 ;                                                /* partial
 u3 / partial x1          */
```

```
    Real*         u32 ;                                                 /* partial
 u3 / partial x2          */
    Real*         u33 ;                                                 /* partial
 u3 / partial x3          */
    Real*         usrVals ;                                             /* user
 values                     */
udfCheckNumUsrVals( udfHd, 3 ) ;                                        /* check
 for error                   */
    usrVals      = udfGetUsrVals( udfHd ) ;                             /* get the
 user vals                  */
    visc0        = usrVals[0] ;                                         /*
 reference visc.                    */
    tConst       = usrVals[1] ;                                         /* time
 constant                   */
    index        = usrVals[2] ;                                         /* index
                   */
    tConst2      = tConst * tConst ;                                    /* for
 efficiency                  */
    exp          = (index - 1) / 2 ;                                    /* for
 efficiency                  */
    gradV        = udfGetElmData( udfHd, UDF_ELM_GRAD_VELOCITY ) ;      /* get the
 grad-vel.                  */
    u11          = &gradV[0*nItems] ;                                   /* localize
 u11                        */
    u12          = &gradV[1*nItems] ;                                   /* localize
 u12                        */
    u13          = &gradV[2*nItems] ;                                   /* localize
 u13                        */
    u21          = &gradV[3*nItems] ;                                   /* localize
 u21                        */
    u22          = &gradV[4*nItems] ;                                   /* localize
 u22                        */
    u23          = &gradV[5*nItems] ;                                   /* localize
 u23                        */
    u31          = &gradV[6*nItems] ;                                   /* localize
 u31                        */
    u32          = &gradV[7*nItems] ;                                   /* localize
 u32                        */
    u33          = &gradV[8*nItems] ;                                   /* localize
 u33                        */
    for ( elem = 0 ; elem < nItems ; elem++ ) {
        invar2       = 2.
                     * ( u11[elem] * u11[elem]
                     + u22[elem] * u22[elem]
                     + u33[elem] * u33[elem]
                     + (u12[elem]+u21[elem]) * (u12[elem]+u21[elem])
                     + (u23[elem]+u32[elem]) * (u23[elem]+u32[elem])
                     + (u31[elem]+u13[elem]) * (u31[elem]+u13[elem]) ;

        outVec[elem] = visc0 ;
        if ( tConst2 * invar2 > ) {
            outVec[elem] = visc0 * pow( tConst2 * invar2, exp ) ;
        }
    }
} /* end of usrViscosityExample() */
```

The dimension of the returned viscosity vector, outVec, is the number of elements.

The *multiplier_function* parameter may be used to uniformly scale the viscosity values. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file. For example, a ramped viscosity may be specified by:

```
VISCOSITY_MODEL( "similar to ramped viscosity model" ) {
    type                    = constant
    viscosity               = 1.781e-5
    multiplier_function      = "ramped"
}
MULTIPLIER_FUNCTION( "ramped" ) {
    type                    = piecewise_log_linear
    curve_fit_values        = { 1, 1000 ; 10, 1 }
    curve_fit_variable      = time_step
}
```

In areas of turbulent flow the molecular viscosity μ is replaced by the total viscosity $\mu + \mu_t$ in the definition of the viscous stress tensor. The turbulent viscosity μ_t is given by the turbulence model in use; see the EQUATION command for details.

Nodal Data Commands

12

The Nodal Data commands are outlined in this chapter.

This chapter covers the following:

- COORDINATE (p. 377)
- DERIVED_QUANTITY_OUTPUT (p. 379)
- ERROR_ESTIMATOR_OUTPUT (p. 380)
- EXTERNAL_OUTPUT (p. 383)
- NODAL_INITIAL_CONDITION (p. 389)
- NODAL_BOUNDARY_CONDITION (p. 402)
- NODAL_RESIDUAL_OUTPUT (p. 427)
- NODAL_OUTPUT (p. 429)
- PERIODIC_BOUNDARY_CONDITION (p. 431)
- RESTART_OUTPUT (p. 446)
- RUNNING_AVERAGE_OUTPUT (p. 448)
- TIME_AVERAGE_OUTPUT (p. 449)
- TIME_HISTORY_OUTPUT (p. 451)

The `COORDINATE` command must be explicitly specified in the input file. All other commands in this chapter are optional. If the `NODAL_OUTPUT` or `TIME_AVERAGE_OUTPUT` commands are not given, their default values are assumed.

COORDINATE

Specifies the nodal coordinates of the problem.

Type

AcuSolve Command

Syntax

COORDINATE {parameters}

Qualifier

This command has no qualifier.

Parameters

coordinates or *coord* *(array) [no default]*

An array of nodal coordinates. Each row contains a unique node number and the x, y, and z coordinates of a nodal point in the global xyz coordinate system.

Description

This command specifies the global nodal coordinates of the problem. This command is mandatory. All the nodes of the problem must be defined with this command. For example,

```
COORDINATE {
   coordinates = {
                    1, 0., 0., 0. ;
                    2, 1., 0., 0. ;
                   11, 0., 1., 0. ;
                   12, 1., 1., 0. ;
                   -1, 0., 0., 1. ;
                   -2, 1., 0., 1. ;
                  -11, 0., 1., 1. ;
                  -12, 1., 1., 1. ;
                                  }
      }
}
```

defines eight nodes on the corners of a cube.

The *coordinates* parameter is a four-column array corresponding to the node number and the x, y, and z coordinates of each nodal point, one row per nodal point. The node number may be any arbitrary but unique integer. Negative numbers are acceptable. The coordinates are in the global xyz coordinate system.

The nodal coordinates may be read from a file. For the above example, the coordinates may be placed in a file, such as `cube.crd`:

```
  1      0.          0.            0.
  2      1.          0.            0.
 11      0.          1.            0.
 12      1.          1.            0.
 -1      0.          0.            1.
```

```
 -2      1.          0.          1.
-11      0.          1.          1.
-12      1.          1.          1.
```

and read by:

```
COORDINATE {
  coordinates =Read( "cube.crd" )
}
```

AcuSolve requires a single set of nodal points, even for mixed-physics simulations.

DERIVED_QUANTITY_OUTPUT

Specifies nodal output parameters for derived quantities.

Type

AcuSolve Command

Syntax

DERIVED_QUANTITY_OUTPUT {parameters}

Qualifier

This command has no qualifier.

Parameters

output_frequency or *out_freq* *(integer) >=0 [0]*
: Time step frequency at which to output nodal derived quantities. If zero, this option is ignored.

output_time_interval or *out_intv* *(real) >=0 [=0]*
: Time frequency at which to output nodal derived quantities. If zero, this option is ignored.

num_saved_states or *save* *(integer) >=0 [=0]*
: Number of nodal derived quantity files to keep. If zero, all are kept.

Description

"Nodal derived quantities" are the nodal projections of material model quantities, body forces, and turbulence data. This command specifies the frequency at which nodal derived quantities are written into a results database on disk. For example,

```
DERIVED_QUANTITY_OUTPUT {
   output_frequency      = 10
   output_time_interval  = 2.5
}
```

specifies that the nodal derived quantities are to be output at every 10 time steps, at every 2.5 units of run time and at the last time step.

Once the nodal derived quantities have been written to disk, they can be translated to other formats using the AcuTrans program and other post-processing modules; see the *AcuSolve Programs Reference Manual* for details.

If either *output_frequency* or *output_time_interval* is non-zero, the nodal derived quantities will be output at the end of the run. If both are zero, no data are written to disk.

Run times may not coincide with *output_time_interval*. In this case, at every time step which passes through a multiple of *output_time_interval*, the nodal derived quantities are output.

The *num_saved_states* parameter indicates the number of nodal derived quantity files to save. Once the (*num_saved_states* + 1)th file is written, the first one is removed. When running in parallel, once all files are written (in parallel), then the old ones are removed (in parallel).

ERROR_ESTIMATOR_OUTPUT

Specifies parameters for outputting the nodal error estimator.

Type

AcuSolve Command

Syntax

ERROR_ESTIMATOR_OUTPUT {parameters}

Qualifier

This command has no qualifier.

Parameters

type (enumerated) [=pde_residual]
Which error estimator to use.

pde_residual or pde_res Error estimate base on equation residual.

output_frequency or *out_freq (integer) >=0 [=0]*
Time step frequency at which to output the error estimator. If zero, this option is ignored.

output_time_interval or *out_intv (real) >=0 [=0]*
Time frequency at which to output the error estimator. If zero, this option is ignored.

num_saved_states or *save (integer) >=0 [=0]*
Number of error estimator files to keep. If zero, all error estimator files are kept.

time_average_output_frequency or *ave_out_freq (integer) >=0 [=0]*
Time step frequency at which to output the time average of the error estimator. If zero, this option is ignored.

time_average_output_time_interval or *ave_out_intv (real) >=0 [=0]*
Time frequency at which to output the time average of the error estimator. If zero, this option is ignored.

num_time_average_saved_states or *ave_save (integer) >=0 [=0]*
Number of time average of the error estimator files to keep. If zero, all time average of the error estimator files are kept.

time_average_reset_frequency or *ave_reset_freq (integer) >=0 [=0]*
Time step frequency at which to reset the averaging process. If zero, never reset.

Description

This command specifies the parameters for writing the nodal error estimator to disk. This is an a posteriori error estimation based on the residuals of the mass, momentum, energy, species, and turbulence equations. Unlike the discrete residual referred to in the NODAL_RESIDUAL_OUTPUT command and elsewhere, these residuals are based on evaluating the strong form of the differential equations

(see the EQUATION command) and the corresponding least-squares metric, I, of GLS stabilization terms. Hence, these residuals are not expected to go to zero with solver convergence. For example,

```
ERROR_ESTIMATOR_OUTPUT {
   type                 = pde_residual
   output_frequency     = 100
   output_time_interval = 0
   num_saved_states     = 2
}
```

writes a nodal error estimator file to disk every 100 time steps as well as the last time step. Only the last two files are kept. Once the third nodal error estimator file is written to disk, the first nodal error estimator file is removed to preserve disk space.

ERROR_ESTIMATOR_OUTPUT is an advanced command that can be used to evaluate the quality of the mesh or problem modeling.

If either *output_frequency* or *output_time_interval* is non-zero, an error estimator file will be written at the end of the run. If both are zero, no error estimator files are created.

Run times may not coincide with *output_time_interval*. In this case, at every time step which passes through a multiple of *output_time_interval*, an error estimator file is written.

The *num_saved_states* parameter indicates the number of nodal error estimator files to save. Once the (*num_saved_states* + 1)th nodal error estimator file is written, the first one is removed. When running in parallel, once all files are written (in parallel), then the old ones are removed (in parallel).

For transient problems, it is usually desirable to average the error estimator over a number of time steps. This may be accomplished by, for example,

```
ERROR_ESTIMATOR_OUTPUT {
   type                                = pde_residual
   time_average_output_frequency       = 100
   time_average_output_time_interval = 0
   num_time_average_saved_states       = 2
   time_average_reset_frequency        = 10
}
```

Here, the error estimator is averaged over 10 steps before it is output.

The pde_residual type gives an error estimator of the form:

$$e_a = \sqrt{\frac{\int_\Omega Na\, R^2 d\Omega}{\int_\Omega Na\, d\Omega}} \qquad (121)$$

where e_a is the error estimator at node a, R is the residual of one of the differential equations (mass, one of the momentum equations, heat, one of the species equations, or turbulence) and the corresponding least-squares metric, I, of GLS stabilization terms, N_a is the shape function for node a, and Ω is volume.

An example of a mesh adaptivity algorithm that uses this error estimator is as follows. A nodal volume and mesh size may be given by:

$$v_a = \int_\Omega Na \, d\Omega$$
$$h_a = \sqrt[3]{v_a} \qquad (122)$$

The total error and total volume are given by volume-weighted sums over all the nodes of the corresponding nodal quantities:

$$E = \sum_a v_a e_a$$
$$V = \sum_a v_a \qquad (123)$$

A target mesh size h_a^t is desired such that the error is distributed uniformly across the problem domain. This uniform target error is:

$$e^t = E/V \qquad (124)$$

Assuming that the error converges as h^a (typically $a = 2$) then the unnormalized target element size is:

$$\tilde{h}_a^t = h_a (e^t / e_a)^{1/\alpha} \qquad (125)$$

To correct for the change in total volume of the new elements, this needs to be scaled by:

$$\beta = \sqrt[3]{\frac{V}{\sum_a (\tilde{h}_a^t)^3}} \qquad (126)$$

The target mesh size is given by:

$$h_a^t = \beta \tilde{h}_a^t \qquad (127)$$

EXTERNAL_OUTPUT

Specifies parameters to enable an external code or user-defined function to collect data from AcuSolve.

Type

AcuSolve Command

Syntax

`EXTERNAL_OUTPUT("name") {parameters...}`

Qualifier

User-given name.

Parameters

`type` (enumerated) [=socket]

Type of the communication.

socket	Socket communication. Requires *`socket_host`*, *`socket_port`* and *`launch`*.
user	User-defined function is called periodically, which may itself communicate with an external code. Requires *`user_function`*.

`output_frequency` or *`out_freq`* *(integer) >=0 [=0]*

Time step frequency at which to allow an external code or user-defined function to collect data from AcuSolve. If zero, this option is ignored.

`output_time_interval` or *`out_intv`* *(real) >=0 [=0]*

Time frequency at which to allow an external code or user-defined function to collect data from AcuSolve. If zero, this option is ignored.

`socket_host` or *`host`* *(string) [no default]*

User-given name of the host that runs the external code. Used with socket type.

`socket_port` or *`port`* *(integer) >0 [=20000]*

Socket port for communication with the external code. Used with socket type.

`user_function` or *`user`* *(string) [no default]*

Name of the user-defined function. Used with user type.

`user_values` *(array) [={}]*

Array of values to be passed to the user-defined function. Used with user type.

`user_strings` *(list) [={}]*

Array of strings to be passed to the user-defined function. Used with user type.

`launch` *(boolean) [=off]*

Flag specifying whether or not AcuSolve should launch the external code. Used with socket type.

launch_command (string) [no default]
Command to launch the external code. Used with *launch*=on and socket type.

Description

This command specifies the parameters to enable an external code or user-defined function to extract data from a running AcuSolve instance. For an external code this extraction is done via the socket type and the External Code Output (ECO) library. The ECO library is documented in the file doc/external_output.txt, which can be found in the AcuSolve distribution. The ECO functions are listed below. Once this library is linked to an external code, the external code is able to collect data from AcuSolve, process it, and store it in any desirable form.

For example, the following command may be used to notify AcuSolve that external output is to be used from an external code:

```
EXTERNAL_OUTPUT( "test" ) {
   type                = socket
   output_frequency    = 1
   output_time_interval = 0
   socket_host         = "hostname"
   socket_port         = 20000
   launch              = off
}
```

By default both output frequencies are zero, which means that this output mechanism is turned off. If either is non-zero, then *socket_host* gives the name of the machine where the external code will run and *socket_port* is the port that the external code will be listening to. Run times may not coincide with *output_time_interval*. In this case, at every time step which passes through a multiple of *output_time_interval*, output to the external code is enabled.

When AcuSolve encounters this command, it will connect to the socket with the given host and port. Every AcuSolve subdomain will establish connection with the external code. The external code must be listening at that port and establish connection to each of the AcuSolve processors. It will then communicate with each processor in order to get the needed information. You can launch the EXTERNAL_OUTPUT command, for example, extOut 20000, directly from the AcuSolve input file. For example,

```
EXTERNAL_OUTPUT( "test" ) {
   ...
   launch         = on
   launch_command = "extOut 20000"
   ...
}
```

A user type is an advanced feature that allows access to AcuSolve data at a fine granularity. One purpose is for a user-defined function to act as an interface to an external visualizer. As such, this function will be able to traverse much of the data contained in AcuSolve using most of the user-defined function support routines. See the *AcuSolve User-Defined Functions Manual* for a detailed description of these routines. For example,

```
EXTERNAL_OUTPUT( "test" ) {
   type              = user
   output_frequency  = 1
   user_function     = "usrParaView"
```

```
   user_values       = { 1., 2., 3. }
   user_strings      = { "visualizer" }
}
```

The following is an example of an external code for *type*=socket written in C:

```
/* Include necessary header files */
#include "acusim.h"
#include "eco.h"
#include "ver.h"
/* Main function */
int main( int, char** ) ;
int
main( int argc,
    char* argv[] )
{
    char*         funcName = "main" ;    /* function name                  */
    EcoHd         ecoHd ;                /* an ECO handle                  */
    Integer       nNodes ;               /* No. nodes                      */
    Integer*      usrIds ;               /* user ids                       */
    Integer*      elemIds ;              /* user ids                       */
    Integer*      cnn ;                  /* cnns                           */
    Real*         crd ;                  /* coordinates                    */
    Real*         vec ;                  /* output vector                  */
    Integer       port ;                 /* port number                    */
    Integer       verbose ;              /* verbose level                  */
    Integer       waitTime ;             /* wait time                      */
    Integer       i ;                    /* temporary integer              */
    Integer       iElm ;                 /* temporary integer              */
    Integer       iVar ;                 /* temporary integer              */
    Integer       tsId ;                 /* time step Id                   */
    Real          time ;                 /* time level                     */
    Real          timeInc ;              /* time increment                 */
    Integer       nElms ;                /* No. element sets               */
    Integer       nElems ;               /* No. elements                   */
    Integer       nElemNodes ;           /* No. nodes per element          */
    Integer       nVars ;                /* No. variables                  */
    Integer       nDims ;                /* Variable dimension             */
    Integer       iTmp ;                 /* temporay integer               */
    Integer       medium ;               /* medium type                    */
    char          buff[SYS_BUFF_SIZE] ;  /* a temporary buffer             */
    IopHd         iopHd ;                /* output handle                  */
/* Set the program name and version */
    sysSetProgName( argv[0] ) ;
    sysSetVersion( VER_VERSION ) ;
    sysSetMessFile( "ECO.txt" ) ;
    sysLineBuffMess( ) ;
/* Get the options */
    if ( argc != 2 ) {
       sysMess( "Usage: %s port", argv[0] ) ;
       sysExit( 1 ) ;
   }
/* Initialize ECO */
    sscanf( argv[1], "%i", &port ) ;
    waitTime      = 3600 ;
    verbose       = 1 ;
    nNodes        = 60 ;
    usrIds        = memNew( Integer, nNodes ) ;
    crd           = memNew( Real, 3*nNodes ) ;
    iopReadFileArrays(      "channel.crd",    nNodes,          2,
                             usrIds,          IOP_ARRAY_INT,   1,
                             crd,             IOP_ARRAY_REAL,  3,          ) ;
```

```
ecoHd             = ecoNew( "oriole",          port,             waitTime,
                            usrIds,            nNodes,           verbose      ) ;
/* Process */
   ecoGetUsrIds(                       ecoHd,  usrIds                         ) ;
   ecoGetCrd(                          ecoHd,  crd                            ) ;
/* Time related output */
   tsId           = ecoGetTimeStep(    ecoHd                                  ) ;
   time           = ecoGetTime(        ecoHd                                  ) ;
   timeInc        = ecoGetTimeInc(     ecoHd                                  ) ;
/* Variables output */
   nVars          = ecoGetNOutVars(    ecoHd                                  ) ;
   for ( iVar = 0 ; iVar < nVars ; iVar++ ) {
       nDims      = ecoGetOutVarDim(   ecoHd,  iVar                           ) ;
       vec        = memNew(            Real,   nDims*nNodes
       ecoGetOutVarName(               ecoHd,  iVar,             buff         ) ;
       sysMess("variable=%s", buff)
       ecoGetOutValues(                ecoHd,  iVar,             vec          ) ;
       for ( i = 0 ; i < nNodes ; i++ ) {
           sysMess("%s @ node %d=%lf", buff, i, vec[i]                        ) ;
   }
}
/* Element output */
   nElms          = ecoGetNElms( ecoHd                                       ) ;
   nElems         = 0 ;
   for ( iElm = 0 ; iElm < nElms ; iElm++ ) {
       iTmp       = ecoGetNElems(     ecoHd,  iElm                           ) ;
       nElems += iTmp ;
       nElemNodes = ecoGetNElemNodes( ecoHd,  iElm                           ) ;
       medium     = ecoGetElmMedium ( ecoHd,  iElm                           ) ;
   }
   cnn            = memNew(          Integer, nElemNodes*nElems              ) ;
   for ( iElm = 0 ; iElm < nElms ; iElm++ ) {
   ecoGetCnn(                          ecoHd,  iElm,             cnn          ) ;
   }
   sprintf( buff, "eco_cnn.txt" ) ;
   sysMess( "%26s = %s", "Writing cnn file", buff ) ;
   iopHd          = iopOpenFile( buff, IOP_WRITE | IOP_ASCII ) ;
   iopWriteInts(                       iopHd,  "cnn",  nElemNodes, nElems,
                                       cnn                                    ) ;
   ecoContinue(                        ecoHd                                  ) ;
/* Cleanup */
   ecoFree(                            ecoHd                                  ) ;
   free(                               usrIds                                 ) ;
   free(                               crd                                    ) ;
   free(                               vec                                    ) ;
   free(                               cnn                                    ) ;
} /* end of main() */
```

This Python example is similar to the above. However, memory management is done by Python.

```
# Import utilities
import acueco
import acudb
import numarray
# Call graphics utility
problem            = "channel"
port               = 20004
codeName           = "test"
waitTime           = 3600
verbose            = 1
matchInputIds      = False                # use condensed node numbers
matchInputIds      = True               # match input node numbers
```

```
if not matchInputIds:
   eco             = acueco.Acueco( port, codeName, waitTime, None, verbose)
else:
   adb             = acudb.Acudb( problem, 'ACUSIM.DIR', 0 )
   adb.openRun( 0 )
   usrIds          = adb.get( 'usrIds' )
   usrIds          = numarray.array( usrIds )
   eco             = acueco.Acueco( port, codeName, waitTime, usrIds, verbose)
# Extract nodal data
nSds               = eco.getNSds()
nNodes             = eco.getNNodes()
usrIds             = eco.getUsrIds()
# Extract element data
# Alternative: the elements may be obtained from acudb
nElms              = eco.getNElms( )
cnns               = []
for i in range(nElms):
   cnn             = eco.getCnn( i )
   cnns.append     ( cnn )
# Get the velocity index
nVars              = eco.getNVars()
velId              = -1
for i in
range(nVars):
   varName         = eco.getVarName( i )
   if varName      == "velocity": velId = i
# Get the velocity field at each time step
while True:
   step            = eco.getTimeStep()
   time            = eco.getTime()
   timeInc         = eco.getTimeInc()
   vel             = eco.getOutValues( velId )
   last            = eco.getLastStepFlag()
   print "Step     =", step
   print "Time     =", time
   print "TimeInc  =", timeInc
   print "Max Vel  =", vel.max()
   eco.continueRun()
   if last: break
```

The functions available in the ECO library are:

ecoNew	Initialize external output.
ecoGetNSds	Return the number of subdomains in the problem.
ecoGetNNodes	Return the number of nodes in the problem.
ecoGetUsrIds	Get the user-prescribed nodal IDs of the problem.
ecoGetCrd	Get the coordinates of the domain.
ecoGetNElms	Get the number of element sets in the domain.
ecoGetNElems	Get the number of elements in the domain.
ecoGetNElemNodes	Get the number of nodes per element.
ecoGetCnn	Output the element connectivity table.

ecoGetElmMedium	Output the medium type (fluid, solid, or shell) of the element set.
ecoGetNOutVars	Output the number of variables in the problem.
ecoGetOutVarName	Output the name of variables in the problem.
ecoGetOutVarDim	Output the dimension of variables.
ecoGetOutValues	Get the values of variables in the domain.
ecoGetTimeStep	Get the time step ID during run time.
ecoGetTime	Get the current time level during simulation.
ecoGetTimeInc	Get the time-step increment.
ecoGetLastStepFlag	Obtain the flag for the last time-step.
ecoContinue	Get the command for the continuation of simulation.
ecoStop	Get the termination command for the run.
ecoFree	Deallocate all the arrays, close all file descriptors and sockets, disconnect from AcuSolve.

NODAL_INITIAL_CONDITION

Specifies the nodal initial conditions for a solution field.

Type

AcuSolve Command

Syntax

```
NODAL_INITIAL_CONDITION("name") {parameters...}
```

Parameters

variables [=auto]

One of the following:

auto	(scalar)
pressure or pres	(scalar)
pressure_time_derivative or pres_rate	(scalar)
velocity or vel	(vector)
x_velocity or xvel	(scalar)
y_velocity or yvel	(scalar)
z_velocity or zvel	(scalar)
acceleration or accel	(vector)
temperature or temp	(scalar)
temperature_time_derivative or temp_rate	(scalar)
species_1 or spec1	(scalar)
species_1_time_derivative or spec1_rate	(scalar)
species_2 or spec2	(scalar)
species_2_time_derivative or spec2_rate	(scalar)
species_3 or spec3	(scalar)
species_3_time_derivative or spec3_rate	(scalar)

species_4 or spec4 (scalar)

species_4_time_derivative or spec4_rate (scalar)

species_5 or spec5 (scalar)

species_5_time_derivative or spec5_rate (scalar)

species_6 or spec6 (scalar)

species_6_time_derivative or spec6_rate (scalar)

species_7 or spec7 (scalar)

species_7_time_derivative or spec7_rate (scalar)

species_8 or spec8 (scalar)

species_8_time_derivative or spec8_rate (scalar)

species_9 or spec9 (scalar)

species_9_time_derivative or spec9_rate (scalar)

particle_source_temperature (scalar)

particle_source_species_1 (scalar)

particle_source_species_2 (scalar)

particle_source_species_3 (scalar)

particle_source_species_4 (scalar)

particle_source_species_5 (scalar)

particle_source_species_6 (scalar)

particle_source_species_7 (scalar)

particle_source_species_8 (scalar)

particle_source_species_9 (scalar)

eddy_viscosity or eddy (scalar)

eddy_viscosity_time_derivative or eddy_rate	(scalar)
kinetic_energy or tke	(scalar)
kinetic_energy_time_derivative or tke_rate	(scalar)
dissipation_rate or teps	(scalar)
dissipation_rate_time_derivative or teps_rate	(scalar)
eddy_frequency or tomega	(scalar)
eddy_frequency_time_derivative or tomega_rate	(scalar)
eddy_time or ttau	(scalar)
eddy_time_time_derivative or ttau_rate	(scalar)
intermittency or tintc	(scalar)
intermittency_time_derivative or tintc_rate	(scalar)
transition_re_theta or treth	(scalar)
transition_re_theta_time_derivative or treth_rate	(scalar)
viscoelastic_stress or vest	(symmetric tensor)
field	(scalar)
relative_humidity	(scalar)
dewpoint_temperature	(scalar)
humidity_film_thickness	(scalar)
incident_radiation	(scalar)
electric_potential	(scalar)
design_topology_filter or topo_filt	(scalar)
design_topology_filter_derivative or topo_filt_rate	(scalar)

mesh_displacement or mesh (scalar)

running_average_pressure (scalar)

running_average_velocity (vector)

running_average_temperature (scalar)

running_average_species_1 (scalar)

running_average_species_2 (scalar)

running_average_species_3 (scalar)

running_average_species_4 (scalar)

running_average_species_5 (scalar)

running_average_species_6 (scalar)

running_average_species_7 (scalar)

running_average_species_8 (scalar)

running_average_species_9 (scalar)

running_average_eddy_viscosity (scalar)

running_average_kinetic_energy (scalar)

running_average_dissipation_rate (scalar)

running_average_eddy_frequency (scalar)

running_average_eddy_time (scalar)

running_average_intermittency (scalar)

running_average_transition_re_theta (scalar)

running_average_relative_humidity (scalar)

running_average_dewpoint_temperature (scalar)

running_average_humidity_film_thickness (scalar)

running_average_mesh_displacement (vector)

selection_type *(enumerated) [=all]*
Method of selecting node set.

all All nodes.

node_list	Node IDs are specified in a user provided list. Requires *nodes* parameter.
element_set	Node list is extracted from the element set. Requires *element_set*.

nodes (array) [no default]
List of node IDs in this set. Used with *selection_type* = node_list.

element_set or *elem_set (string) [no default]*
Name of element set whose nodes are to be included in this initial condition command.

default_value or *value (real) [=0]*
The default value of the initial condition. This applies to scalar fields when *type*=nodal_values. If the nodal values provided in the *nodal_values* array do not contain all of the nodes identified for this NODAL_INITIAL_CONDITION command, the *default_value* is used to assign the value of the initial condition to the nodes that are missing.

default_values or *values (array) [={0,0,0}]*
The default values of the initial conditions. This applies to vector and tensor fields when *type*=nodal_values. If the nodal values provided in the *nodal_values* array do not contain all of the nodes identified for this NODAL_INITIAL_CONDITION command, the *default_value* is used to assign the value of the initial condition to the nodes that are missing.

type (enumerated) [=zero]
Type of the initial condition.

zero	Zero for the set.
constant or const	Constant value. Requires *constant_value* for scalar variables and *constant_values* for vector and tensor variables.
nodal_values* or *nodal	Nodal values. Requires nodal_values.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
user_function or user	User-defined function. Requires *user_function*, *user_values* and *user_strings*.

constant_value (real) [=0]
Constant value of the initial condition. Used with constant type and scalar variables.

constant_values (array) [{0.0,0.0,0.0}]
Constant values of the initial condition. Used with constant type and vector/tensor variables.

field (string) [no default]
Value of the initial condition for field. Used with variable field.

nodal_values or *values* *(array) [no default]*
Array of node IDs (first column) and data values. Used with nodal_values type.

curve_fit_values or *curve_values* *(array) [={0,0}]*
A two-column array of independent-variable/initial condition data values. Used with piecewise_linear and cubic_spline types.

curve_fit_variable or *curve_var* *(enumerated) [=temperature]*
Independent variable of the curve fit. Used with piecewise_linear and cubic_spline types.

x_coordinate or xcrd	X-component of coordinates.
y_coordinate or ycrd	Y-component of coordinates.
z_coordinate or zcrd	Z-component of coordinates.
x_reference_coordinate or xrefcrd	X-component of reference coordinates.
y_reference_coordinate or yrefcrd	Y-component of reference coordinates.
z_reference_coordinate or zrefcrd	Z-component of reference coordinates.
x_velocity or xvel	X-component of velocity.
y_velocity or yvel	U-component of velocity.
z_velocity or zvel	Z-component of velocity.
velocity_magnitude or vel_mag	Velocity magnitude.
pressure or pres	Pressure.
temperature or temp	Temperature.
eddy_viscosity or eddy	Turbulence kinematic eddy viscosity.
kinetic_energy or tke	Turbulence kinetic energy.
eddy_frequency	Turbulence eddy frequency.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.

species_6 or spec6	Species 6.
species_7 or spec7	Species 7.
species_8 or spec8	Species 8.
species_9 or spec9	Species 9.
mesh_x_displacement or mesh_xdisp	X-component of mesh displacement.
mesh_y_displacement or mesh_ydisp	Y-component of mesh displacement.
mesh_z_displacement or mesh_zdisp	Z-component of mesh displacement.
mesh_displacement_magnitude or mesh_disp_mag	Mesh displacement magnitude.
mesh_x_velocity or mesh_xvel	X-component of mesh velocity.
mesh_y_velocity or mesh_yvel	Y-component of mesh velocity.
mesh_z_velocity or mesh_zvel	Z-component of mesh velocity.
mesh_velocity_magnitude or mesh_vel_mag	Mesh velocity magnitude.

`user_function` or `user` *(string) [no default]*
Name of the user-defined function. Used with user_function type.

`user_values` *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function type.

`user_strings` *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with user_function type.

`precedence` *(integer) [=1]*
Precedence of this initial condition set with respect to other sets. The set with the highest value has precedence.

`satisfy_boundary_condition` *(boolean) [=off]*
Flag specifying whether or not to change the initial condition data to satisfy the nodal and periodic boundary conditions.

Description

This command specifies the nodal initial conditions of a solution field.

For example, the nodal initial conditions of the temperature field may be specified by:

```
NODAL_INITIAL_CONDITION( "temperature initial condition"  ) {
   selection_typ      e       = all
   variable                   = temperature
   default_value              = 273
   type                       = nodal_values
   nodal_values               = { 1, 300 ;
                                  2, 310 ; }
   satisfy_boundary_condition = off
}
```

The temperature of all nodal points is set to the default value of 273, except for nodes one and two which have temperature values of 300 and 310, respectively. No modifications are made for boundary conditions.

Starting in AcuSolve 14.0, the target variable of the initial condition is specified through the *variable* entry. Alternatively, the variable can be specified via the name of the NODAL_INITIAL_CONDITION entry by selecting the *variable* = auto option, or omitted altogether since it is the default value. This behavior exists only to preserve backward compatibility. For example, the legacy use of this command was as follows:

```
NODAL_INITIAL_CONDITION( temperature ) {
   default_value              = 273.0
   nodal_values               = { 1, 300.0 ;
                                  2, 310.0 ; }
   satisfy_boundary_condition = off
}
```

Which sets the initial condition on temperature to a default value of 273.0 except at nodes one and two, which are set to 300.0 and 310.0, respectively. The full version of this command, including the default values, is as follows:

```
NODAL_INITIAL_CONDITION( temperature ) {
   selection_type             = all
   variable                   = auto
   default_value              = 273.0
   nodal_values               = { 1, 300.0 ;
                                  2, 310.0 ; }
   satisfy_boundary_condition  = off
}
```

For scalar fields, *default_value* specifies a single scalar value to be used by all nodes contained in the set. The default value may be overwritten for a subset of nodes by *nodal_values*. This parameter expects a two-column array corresponding to the node numbers and initial values of the scalar field. Node numbers must be valid numbers, as given by the COORDINATE command.

For vector fields, such as *velocity* and *acceleration*, *default_values* provides the default field. This parameter is a vector of three components in the global xyz coordinate system. Again, the default values may be overwritten for a subset of the nodes by *nodal_values*. In this case, however, the array has four columns, corresponding to the node number and the x, y, and z components of the vector field. For example,

```
NODAL_INITIAL_CONDITION( "velocity initial condition" ) {
   selection_type = all
   variable       = velocity
```

```
   default_values = { 1, 0, 0 }
   type           = nodal_values
   nodal_values   = { 1, 2.5, 0.0, 1.5 ;
                      2, 2.5, 1.0, 1.5 ; }
}
```

The nodal values may be read from a file. For the above example, the velocity values may be placed in a file, such as, `velocity.nic`:

```
1      2.5          0.0             1.5
2      2.5          1.0             1.5
```

and read by:

```
NODAL_INITIAL_CONDITION( "velocity other initial condition" ) {
   selection_type    = all
   variable          = velocity
   default_values    = { 1, 0, 0 }
   type              = nodal_values
   nodal_values      = Read( "velocity.nic" )
}
```

For symmetric tensor fields, such as *viscoelastic_stress*, *default_values* requires six components. The default values may be overwritten for a subset of the nodes by *nodal_values*. In this case, however, the array has seven columns, corresponding to the node number and the xx, yy,zz,xy,yz,zx components of the stress field. For example,

```
NODAL_INITIAL_CONDITION( "stress initial condition" ) {
   selection_type = all
   variable       = viscoelastic_stress
   type           = nodal_values
   default_values = { 1.0, 0.0, 0.0, 1.0, 0.0, 1.0 }
   nodal_values   = { 1, 1.1, 0.0, 0.0, 1.2, 0.0, 1.5 ;
                      2, 2.1, 0.0, 0.0, 2.2, 0.0, 2.5 ; }
}
```

The nodal values may be read from a file. For the above example, the viscoelastic stress values may be placed in a file, such as `viscoelastic.nic`:

```
1      1.1          0.0             0.0               1.2                  0.0
            1.5
2      2.1          0.0             0.0               2.2                  0.0
            2.5
```

And read by:

```
NODAL_INITIAL_CONDITION( "stress initial condition" ) {
   selection_type  = all
   variable        = viscoelastic_stress
   type            = nodal_values
   default_values  = { 1.0, 0.0, 0.0, 1.0, 0.0, 1.0 }
   nodal_values    = Read( "viscoelastic_stress.nic" )
}
```

The *default_value* and *default_values* parameters are restricted to the set of nodes that the instance of the `NODAL_INITIAL_CONDITION` command applies to. For example, the following command

selects all nodes from the element set named "fluid elements", but then only explicitly defines nodal values for a portion of that set.

```
NODAL_INITIAL_CONDITION( "temperature - element set" ) {
   selection_type             = element_set
   element_set                = "fluid_elements"
   variable                   = temperature
   type                       = nodal_values
   nodal_values               = { 1, 300.0 ;
                                  2, 310.0 ; }
   default_value              = 273.0
   satisfy_boundary_condition = off
   precedence                 = 1
}
```

In this case, all node IDs from the element set are collected into a single array, and their value is initialized to the default_value of 273.0. The node IDs that are listed in the *nodal_values* array are then overwritten with the values listed in the array. The end result is that all nodes in the element set, with the exception of nodes one and two, receive a temperature of 273.0. Nodes one and two are set to 300.0 and 310.0, respectively.

The process of setting initial conditions when *type* = nodal is as follows:

- Collect all node IDs that are contained in the node set defined by the current instance of the NODAL_INITIAL_CONDITION command.
- Compare the node IDs specified by the *nodal_values* array against that of the collected node list.
  - For node IDs in the *nodal_values* array that are contained within the collected set of nodes, their initial condition values are assigned based on the *nodal_values* array.
  - For node IDs that are present in the collected list, but do not have a value explicitly assigned through the *nodal_values* array, the *default_value* is taken as the initial condition.
  - For node IDs that appear in the *nodal_values* array, but are not a part of the collected node list, a warning message is issued and no initial condition value is set.

This behavior is common across all methods of selecting the target nodes for the NODAL_INITIAL_CONDITION command when *type*=nodal_values. The nodes listed in the *nodal_values* array are checked for inclusion in the set of nodes that the NODAL_INITIAL_CONDITION command applies to, and then the logic listed above is invoked to set the values.

The initial condition data may be further modified to satisfy the initial nodal and periodic boundary conditions by setting *satisfy_boundary_condition* to on. This may smooth the solution and help convergence during the first few times steps.

If a running average variable is defined, then its initial condition may be set here. See the EQUATION command for the definition of running average fields and the counter N used below. The following rules are used to initialize each running average field:

- If no initial condition on the running average field is specified, then it is set to zero with N = zero. That is, its first non-zero values will be set at the end of the first time step.
- If it is initialized using this command with *satisfy_boundary_condition* = off, then the specified values are used for the initial values, with N = 1.
- If it is initialized using this command with *satisfy_boundary_condition* = on, then its initial values are set to the initial values of the corresponding solution field and N = 1.

- At restart, its values and counter are obtained from the restart file. The restart values may be overwritten by this command in which case rules two and three apply.

> **Note:** Because of the above flexibility, each variable carries its own running average counter N. However, there is only one counter for all species variables.

The particle source initial conditions are used only when running coupled particle-flow problems with AcuTrace. Except in rare cases, the default value of zero is sufficient.

Nodal initial conditions of types piecewise_linear and cubic_spline may be used to define initial values as a function of a single independent variable. For example,

```
NODAL_INITIAL_CONDITION( "curve fit IC on temperature" ) {
   selection_type      = node_list
   variable            = temperature
   type                = cubic_spline
   nodes               = { 1 ; 3 ; 6 ; }
   curve_fit_values    = { -1, 0.0 ;
                            0, 2.5 ;
                            1, 0.0 ; }
   curve_fit_variable  = y_coordinate
}
```

defines an initial condition on the temperature as a function of the y-coordinate. In this example you get a parabolic profile centered at y = 0, with a width of 2 and a center-line magnitude of 2.5. The *curve_fit_values* parameter is a two-column array corresponding to the independent variable and the initial condition values. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits.

> **Note:** The initial conditions are only applied to nodes one, three, and six in this case.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, such as `temperature.nic.fit`:

```
-1 0.0
 0 2.5
 1 0.0
```

and read by:

```
NODAL_INITIAL_CONDITION( "other curve fit IC on temperature" ) {
   selection_type     = node_list
   variable           = temperature
   type               = cubic_spline
   node_type          = specified
   nodes              = { 1 ; 3 ; 6 ; }
   curve_fit_values   = Read( "velocity.nic.fit" )
   curve_fit_variable = y_coordinate
}
```

The nodal initial conditions for multi field simulations can be defined by using the *nodal_values* command to define the field. This may be specified by:

```
NODAL_INITIAL_CONDITION( field ) {
   variable       = "field"
```

```
   field          = "Gasoline"
   default_value  = 1
   nodal_values   = Read( "field.nic" )
}
```

Where the *nodal_values* file "field.nic" is a two column array containing node ID and field value.

A nodal initial condition of user_function type may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider the case where the velocity is initialized to a parabolic profile inside a circular pipe. Then the input command may be given by:

```
NODAL_INITIAL_CONDITION( "UDF IC on velocity" ) {
   variable        = velocity
   type            = user_function
   selection_type  = node_list
   nodes           = Read( "inflow.nic" )
   user_function   = "usrNodalIcExample"
   user_values     = { 0,  # y-center
                       0,  # z-center
                       1,  # radius
                       1 } # max velocity
}
```

where the user-defined function "usrNodalIcExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrNodalIcExample ) ;          /* function prototype              */
   Void usrNodalIcExample (
   UdfHd      udfHd,                         /* Opaque handle for accessing data */
   Real*      outVec,                        /* Output vector                    */
   Integer    nItems,                        /* Number of BC nodes               */
   Integer    vecDim                         /* = 3 */
   ) {
   Integer    node ;                         /* a node counter                   */
   Real       radius ;                       /* pipe radius                      */
   Real       scale ;                        /* scaling factor                   */
   Real       vel0 ;                         /* velocity at center line          */
   Real       y ;                            /* y-location                       */
   Real       yCenter ;                      /* y-coordinates of flow axis       */
   Real       z ;                            /* z-location                       */
   Real       zCenter ;                      /* y-coordinates of flow axis       */
   Real       crd ;                          /* coordinates                      */
   Real       usrVals ;                      /* user values                      */
   Real       xCrd ;                         /* x-coordinates                    */
   Real       yCrd ;                         /* y-coordinates                    */
   Real       zCrd ;                         /* z-coordinates                    */
udfCheckNumUsrVals( udfHd, 4 ) ;              /* check for error                  */
   usrVals    = udfGetUsrVals( udfHd ) ;     /* get the user vals                */
   yCenter    = usrVals[0] ;                 /* get y-center                     */
   zCenter    = usrVals[1] ;                 /* get z-center                     */
   radius     = usrVals[2] ;                 /* get pipe radius                  */
   vel0       = usrVals[3] ;                 /* get max velocity                 */
   scale      = 1 / ( radius * radius )      /* for convenience                  */
   crd        = udfGetNbcCrd( udfHd ) ;      /* get the coord.                   */
   xCrd       = &crd[0*nItems] ;             /* localize x-coord.                */
   yCrd       = &crd[1*nItems] ;             /* localize y-coord.                */
   zCrd       = &crd[2*nItems] ;             /* localize z-coord.                */
   for ( node = 0 ; node < nItems ; node++ ) {
```

```
      y          = yCrd[node] - yCenter ;
      z          = zCrd[node] - zCenter ;
                                              /* specifying velocity in x,y,z
 directions */
      outVec[node+0*nItems] = vel0 * ( 1 - scale * ( y*y + z*z ) ) ;
      outVec[node+1*nItems] = 0 ;
      outVec[node+2*nItems] = 0 ;
   }
 } /* end of usrNodalIcExample() */
```

The dimension of the returned boundary condition vector, *outVec*, is the number of nodes times the number of dimensions.

Any number of NODAL_INITIAL_CONDITION commands can be present for a given variable. The precedence option determines which instance of the command will override the others. Higher precedence values override lower precedence values. For example, the following set of commands can be used to define initial conditions on the temperature field for a conjugate heat transfer simulation. The temperature in the solid is initialized to a temperature of 300 and the temperature in the fluid is initialized to 325. The nodes on the shared surface between the element sets are present in both commands. The higher precedence assigned to the nodes in the solid ensure that the temperature on these nodes is enforced at 300.

```
NODAL_INITIAL_CONDITION( "solid temperature" ) {
   variable       = temperature
   type           = constant
   constant_value = 300.0
   selection_type = element_set
   element_set    = "solid elements"
   precedence     = 2
}
```

```
NODAL_INITIAL_CONDITION( "fluid temperature" ) {
   variable        = temperature
   type            = constant
   constant_value  = 325.0
   selection_type  = element_set
   element_set     = "fluid elements"
   precedence        = 1
}
```

NODAL_BOUNDARY_CONDITION

Specifies boundary conditions for a solution field on a set of nodes.

Type

AcuSolve Command

Syntax

```
NODAL_BOUNDARY_CONDITION("name") {parameters...}
```

Qualifier

User-given name.

Parameters

`variable` or `var` (enumerated) [no default]

Boundary condition variable.

x_velocity or xvel	X-component of velocity.
y_velocity or yvel	U-component of velocity.
z_velocity or zvel	Z-component of velocity.
direction_velocity or dvel	A velocity component in an arbitrary direction. Requires *`direction_type`*.
pressure or pres	Pressure.
temperature or temp	Temperature.
relative_humidity or relHum	Relative humidity.
dewpoint_temperature or dewpoint	Dewpoint temperature.
electric_potential	Electric potential value. A value of zero represents a ground voltage.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.

species_7 or spec7	Species 7.
species_8 or spec8	Species 8.
species_9 or spec9	Species 9.
eddy_viscosity or eddy	Turbulence kinematic eddy viscosity.
kinetic_energy or tke	Turbulence kinetic energy.
eddy_frequency or tomega	Turbulence eddy frequency.
eddy_time or ttau	Turbulence eddy time.
dissipation_rate or teps	Turbulence dissipation rate.
intermittency or tintc	Transition intermittency.
transition_re_theta or treth	Transition Re-Theta
viscoelastic_xx_stress or xxvest	x-x component of the viscoelastic stress.
viscoelastic_yy_stress or yyvest	y-y component of the viscoelastic stress.
viscoelastic_zz_stress or zzvest	z-z component of the viscoelastic stress.
viscoelastic_xy_stress or xyvest	x-y component of the viscoelastic stress.
viscoelastic_yz_stress or yzvest	y-z component of the viscoelastic stress.
viscoelastic_zx_stress or zxvest	z-x component of the viscoelastic stress.
field	Volume fraction of field in multi field for levelset and algebraic Eulerian problems or mass fraction of field in multi field for the humid air model.
design_topology_filter or topo_filt	Design topology filter.
mesh_x_displacement or mesh_xdisp	X-component of mesh displacement.
mesh_y_displacement or mesh_ydisp	Y-component of mesh displacement.

mesh_z_displacement or mesh_zdisp	Z-component of mesh displacement.
mesh_direction_displacement or mesh_ddisp	A mesh displacement component in an arbitrary direction. Requires *direction_type*.
mesh_x_velocity or mesh_xvel	X-component of mesh velocity.
mesh_y_velocity or mesh_yvel	Y-component of mesh velocity.
mesh_z_velocity or mesh_zvel	Z-component of mesh velocity.
mesh_direction_velocity or mesh_dvel	A mesh velocity component in an arbitrary direction. Requires *direction_type*.

precedence (integer) [=1]
Precedence of this boundary condition set with respect to other sets. The set with the highest value has precedence.

type (enumerated) [=zero]
Type of the boundary condition.

zero	Zero for the set.
constant or const	Constant value. Requires constant_value.
nodal	Nodal values. Requires *nodal_values*.
nodal_time_series	Time series for each node. Requires *nodal_values* and times_series.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
user_function or user	User-defined function. Requires *user_function*, *user_values* and *user_strings*.
scattered_data	Spatially interpolate from data. Requires *scattered_data*, *scattered_data_origin* and *scattered_data_axes*.
scattered_data_time_series	Spatially and temporally interpolate from data. Requires *scattered_data*, *scattered_data_origin*, *scattered_data_axes*, and times_series.

external_file	Read values dynamically from an external file. Requires external_file, *external_file_evaluation* and *external_file_nodal_offset*.
match_mesh_velocity	Set equal to the corresponding component of mesh velocity. Used with x_velocity, y_velocity, z_velocity and direction_velocity variables.
mesh_motion	Mesh displacement given by MESH_MOTION command. Requires *mesh_motion*. Used with mesh_x_displacement, mesh_y_displacement and mesh_z_displacement variables.
flexible_body	Mesh displacement or velocity given by FLEXIBLE_BODY command. Requires flexible_body and *nodal_modes*. Used with mesh displacement and velocity variables.
design_variables	User-given name of the design optimization variable.
guide_surface	Allows mesh to slip on a given surface. Requires *guide_surface*. Used with x_velocity, y_velocity, z_velocity, direction_velocity and mesh_direction_displacement variables.
external_code	Mesh displacement given by an external solid/structural code. Used with mesh_x_displacement, mesh_y_displacement, mesh_z_displacement and mesh_direction_displacement variables.

nodes (array) [no default]
List of nodal points in this set.

constant_value or *value (real) [=0]*
Constant value of the boundary condition. Used with constant type.

field (real) [no default]
Value of the boundary condition for field. Used with *variable* field.

nodal_values or *values (array) [no default]*
Array of nodes (first column) and data values. Used with nodal and nodal_time_series types.

curve_fit_values or *curve_values (array) [={0,0}]*
A two-column array of independent-variable/boundary-condition data values. Used with piecewise_linear and cubic_spline types.

curve_fit_variable or *curve_var (enumerated) [=temperature]*
Independent variable of the curve fit. Used with piecewise_linear and cubic_spline types.

x_coordinate or xcrd	X-component of coordinates.
y_coordinate or ycrd	Y-component of coordinates.
z_coordinate or zcrd	Z-component of coordinates.

x_reference_coordinate or xrefcrd	X-component of reference coordinates.
y_reference_coordinate or yrefcrd	Y-component of reference coordinates.
z_reference_coordinate or zrefcrd	Z-component of reference coordinates.
x_velocity or xvel	X-component of velocity.
y_velocity or yvel	U-component of velocity.
z_velocity or zvel	Z-component of velocity.
velocity_magnitude or vel_mag	Velocity magnitude.
pressure or pres	Pressure.
temperature or temp	Temperature.
relative_humidity	Relative humidity.
dewpoint_temperature	Dewpoint temperature.
eddy_viscosity or eddy	Turbulence kinematic eddy viscosity.
kinetic_energy or tke	Turbulence kinetic energy.
eddy_frequency	Turbulence eddy frequency.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.
species_7 or spec7	Species 7.
species_8 or spec8	Species 8.
species_9 or spec9	Species 9.
mesh_x_displacement or mesh_xdisp	X-component of mesh displacement.

mesh_y_displacement or mesh_ydisp	Y-component of mesh displacement.
mesh_z_displacement or mesh_zdisp	Z-component of mesh displacement.
mesh_displacement_magnitude or mesh_disp_mag	Mesh displacement magnitude.
mesh_x_velocity or mesh_xvel	X-component of mesh velocity.
mesh_y_velocity or mesh_yvel	Y-component of mesh velocity.
mesh_z_velocity or mesh_zvel	Z-component of mesh velocity.
mesh_velocity_magnitude or mesh_vel_mag	Mesh velocity magnitude.

user_function or *user* *(string) [no default]*
Name of the user-defined function. Used with user_function type.

user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function type.

user_strings *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with user_function type.

user_nodal_values *(array) [={}]*
Array of data values (starting with second column) for each node (first column) to be passed to the user-defined function. Used with user_function type.

scattered_data *(array) [no default]*
Array of xyz coordinates (first three columns] and data values. Used with scattered_data and scattered_data_time_series types.

scattered_data_origin *(array) [={0,0,0}]*
Origin for transformation of *scattered_data* coordinates to global coordinates. Used with scattered_data and scattered_data_time_series types.

scattered_data_axes *(array) [={1,0,0;0,1,0;0,0,1}]*
Axes for transformation of *scattered_data* coordinates to global coordinates. Used with scattered_data and scattered_data_time_series types.

external_file *(string) [no default]*
User-given file name referencing the file containing boundary condition data. The file is in local binary format. Used with external_file type.

external_file_evaluation *(enumerated) [=once_per_timestep]*
Frequency of reading the external file. Used with external_file type.

once_per_timestep or step Once per time step.

external_file_nodal_offset (array) [no default]
Two-column array giving the node and the byte offset of the datum of each node for reading the external file. Used with external_file type.

auxiliary_nodes (array) [no default]
Array of auxiliary nodes (starting with second column) for each node (first column) to be passed to the user defined function. Used with user_function type.

reference_frame (string) [=none]
User-given name of the reference frame for transforming a velocity boundary condition. If none, or if *variable* is not a velocity component, no transformation takes place.

multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the boundary condition. If none, no scaling is performed.

mesh_motion (string) [=none]
User-given name of the MESH_MOTION command for specifying (mesh_motion type) or scaling (other types) a mesh displacement boundary condition. If none, or if variable is not a mesh displacement component, no scaling is performed. none is not supported with mesh_motion type.

flexible_body (string) [no default]
User-given name of the FLEXIBLE_BODY command for specifying a mesh displacement or velocity boundary condition. Used with flexible_body type.

guide_surface (string) [no default]
User-given name of the GUIDE_SURFACE command for specifying a velocity or mesh displacement boundary condition. Used with guide_surface type.

nodal_modes (array) [no default]
Array of node number (first column) and eigenvector values (last num_modes columns, where *num_modes* is given by the FLEXIBLE_BODY command referenced by flexible_body). Used with flexible_body type.

time_series (array) [={0}]
Array of time values. Used with nodal_time_series and scattered_data_time_series types.

time_series_variable (enumerated) [=time]
Independent variable for time series. Used with nodal_time_series and scattered_data_time_series types.

time	Time
cyclic_time	Time cycled over a period. Requires *time_series_cyclic_period*.
multiplier_function	User-given name of the multiplier function that replaces time. Requires *time_series_variable_multiplier_function*.

time_series_cyclic_period (real) >=0 [=1]
Value of the time period. Zero denotes an infinite period. Used with cyclic_time time series variable.

time_series_variable_multiplier_function (string) [=none]
User-given name of the multiplier function for the independent variable of the time series. If none, the independent variable is zero. Used with *multiplier_function* time series variable.

coupling_type (enumerated) [=structural]
Manner in which the nodes are coupled to an external code. Used with *type*=external_code.

structural	Nodes are coupled with either a line or surface element set in the external.
line	Nodes are coupled with a line element set in the external code.
surface	Nodes are coupled with a surface element set in the external code.
rigid_body	Nodes are coupled with a rigid body in the external code.

rigid_body_name (array) [no default]
Name of the rigid body the nodal boundary condition is coupled with. Used with mesh displacement and velocity variables, *type*=external_code and *coupling_type*=rigid_body.

external_code_tags (list) [={}]
Array of strings to be passed to the external code for identifying surface associations for the purpose of boundary condition assignment. Used with mesh displacement and velocity variables and when *type*=external_code. Used with *coupling_type* of structural, line, or surface. When this option is specified, the associativity of a given node in the AcuSolve model is restricted based on this mapping and not a simple projection of the external code coordinates onto the AcuSolve surfaces.

direction_type or dir_type (enumerated) [=constant]
Type of the direction vector. Used with direction_velocity variable.

constant or const	Constant direction. Requires *constant_direction*.
nodal	Nodal direction. Requires *nodal_directions*.
normal	Direction is normal to the boundary. Direction is computed automatically.

constant_direction (array) [={1,0,0}]
Constant direction vector specified in the global xyz coordinate system. Direction is normalized before use. Used with constant direction type.

nodal_directions (array) [no default]
Nodal direction vector. Directions are normalized before use. Used with nodal direction type.

active_type (enumerated) [=all]
Type of the active flag. Determines which nodes in this set will have this boundary condition imposed.

all	All nodes in this set are active.
none	No node in this set is active.
inflow or in	Only nodes where the flow is into the fluid domain are active.
user_function or user	Active nodes determined by a user-defined function. Requires *active_user_function*, *active_user_values* and *active_user_strings*.
no_outflow	Only nodes that where there is no outflow are active.
no_interface	Only nodes that are not on an interface surface or do not find a contact surface of an appropriate medium are active.
far_field	Only nodes that have a velocity boundary condition vector that is entering into the volume are active.

active_user_function *(string) [no default]*
Name of the user-defined function for the active flag. Used with user_function active type.

active_user_values *(array) [={}]*
Array of values to be passed to the active flag user-defined function. Used with user_function active type.

active_user_strings *(list) [={}]*
Array of strings to be passed to the active flag user-defined function. Used with user_function active type.

synthetic_turbulence *(boolean) (=off)*
Flag specifying whether to activate the synthetic turbulence feature at inflow, with velocity or atmospheric inflow type. This command is only activated when *synthetic_turbulence_input_type* in *turbulence_model_parameters* is set to k_omega (or k_epsilon).

Description

This command specifies nodal boundary conditions for a solution variable on a set of nodes. For example,

```
NODAL_BOUNDARY_CONDITION( "constant BC on inflow x-velocity" ) {
   variable        = x_velocity
   type            = constant
   nodes           = { 1 ;
                       3 ;
                       6 ; }
   constant_value  = 12.5
}
```

defines nodal boundary conditions on the x-component of velocity for global nodes one, three and six. The boundary condition is a constant value of 12.5 for all three nodes.

The *variable* and *nodes* parameters are mandatory. *variable* specifies the solution field to which the boundary condition applies. If the problem does not solve for such a variable (see the STAGGER

command), the boundary condition is ignored. *nodes* is a single-column array containing a list of node numbers. Node numbers must be valid numbers, as given by the COORDINATE command. Duplicate nodes are not allowed in a single set. This means that to specify two or more boundary conditions on the same variable (such as direction_velocity) and on the same node, multiple NODAL_BOUNDARY_CONDITION commands must be used.

The nodal numbers may be read from a file. For the above example the nodes may be placed in a file, such as inflow.nbc:

```
1
3
6
```

and read by:

```
NODAL_BOUDARY_CONDITION( "constant BC on inflow x-velocity" ) {
   variable        = x_velocity
   type            = constant
   nodes           = Read( "inflow.nbc" )
   constant_value  = 12.5
}
```

The field variable may be specified through the NODAL_BOUNDARY_CONDITION command by assigning the associated FIELD as defined according to the material model selected.

```
NODAL_BOUNDARY_CONDITION( "Inflow field" ) {
   nodes          = Read( "Inflow.quad4.nbc" )
   variable       = "field"
   field          = "Air"
   type           = constant
   precedence     = 2
   constant_value = 1
   active_type    = all
}
```

For the above example the field attribute has been previously defined with the FIELD specifier, as follows:

```
FIELD( "Air" ) {
   material_model  = "Air"
}
```

If more than one nodal boundary condition is specified for a given variable on a given node, then the *precedence* parameter is used to determine which boundary condition to satisfy. The boundary condition set with the highest value of *precedence* takes precedence. If two or more boundary conditions have the same highest value, then the one with value of zero takes precedence. For the purpose of this evaluation, boundary conditions of types piecewise_linear, cubic_spline, user_function, and mesh_motion are assumed nonzero. Finally, if the zero condition is ambiguous, the last boundary condition set specified is used. For vector variables, such as velocity, the precedence is determined collectively on all components of the variable. For example, if the direction of a direction_velocity boundary condition coincides with the x-axis and there is also an x_velocity boundary condition on the same node, the above precedence rule is used to determine which boundary condition to satisfy.

A constant boundary condition type applies the same boundary condition on all nodes in the set, as in the above example.

A zero type is a short-hand form of the constant type, with a constant value of zero. For example,

```
NODAL_BOUNDARY_CONDITION( "zero BC on inflow y-velocity" ) {
   variable  = y_velocity
   type      = zero
   nodes     = Read( "inflow.nbc" )
}
```

A nodal type specifies a different value for each node. For example,

```
NODAL_BOUNDARY_CONDITION( "nodal BC on inflow x-velocity" ) {
   variable      = x_velocity
   type          = nodal
   nodes         = { 1 ;
                     3 ;
                     6 ; }
   nodal_values  = { 1, 0 ;
                     3, 1 ;
                     6, 2 ; }
}
```

The *nodal_values* parameter is a two-column array corresponding to the node number and boundary condition value. The node numbers must match those given by the nodes parameter. No node may be missing.

The nodal values may be read from a file. For the above example, the boundary conditions may be placed in a file, such as `inflow.bcu`:

```
1      0
3      1
6      2
```

and read by:

```
NODAL_BOUNDARY_CONDITION( "nodal BC on inflow x-velocity" ) {
   variable     = x_velocity
   type         = nodal
   nodes        = Read( "inflow.nbc" )
   nodal_values = Read( "inflow.bcu" )
}
```

The preceding can be generalized to a function of time at each node with the nodal_time_series type. The time points are given by *time_series* and must be in ascending order. If the number of points in *time_series* is N, then the number of columns in *nodal_values* must be N+1. Piecewise linear interpolation of the *nodal_values* data is used for times within the time series, and the limit point values are used for times outside the range of the time series. For example,

```
NODAL_BOUNDARY_CONDITION( "time-dependent nodal BC on inflow x-velocity" ) {
   variable     = x_velocity
   type         = nodal_time_series
   nodes        = Read( "inflow.nbc" )
   nodal_values = { 1, 0., 0.1, 0.2, 0.5 ;
                    3, 1., 1.2, 1.5, 1.8 ;
                    6, 2., 2.3, 2.6, 2.9 ; }
```

```
   time_series  = { 0., 0.5, 1.0, 1.5 }
}
```

The data may be read from files. `inflow.bcu` becomes

```
1 0. 0.1 0.2 0.5
3 1. 1.2 1.5 1.8
6 2. 2.3 2.6 2.9
```

and the time series may placed in `inflow.time`:

```
0. 0.5 1.0 1.5
```

These files are then read by:

```
NODAL_BOUNDARY_CONDITION( "time-dependent nodal BC on inflow x-velocity" ) {
   variable     = x_velocity
   type         = nodal_time_series
   nodes        = Read( "inflow.nbc" )
   nodal_values = Read( "inflow.bcu" )
   time_series  = Read( "inflow.time" )
}
```

If the boundary condition is periodic in time *time_series_variable* may be set to cyclic_time with the period given by *time_series_cyclic_period*. In this case time is replaced by mod(time, *time_series_cyclic_period*) before the temporal interpolation is done. Data in *nodal_values* outside of the first period in time are ignored. An infinite period is denoted by *time_series_cyclic_period* = 0, which is equivalent to *time_series_variable* = time. For example,

```
NODAL_BOUNDARY_CONDITION( "time-periodic nodal BC on inflow x-velocity" ) {
   variable                 = x_velocity
   type                     = nodal_time_series
   nodes                    = Read( "inflow.nbc" )
   nodal_values             = Read( "inflow.bcu" )
   time_series              = Read( "inflow.time" )
   time_series_variable     = cyclic_time
   time_series_cyclic_period = 2
}
```

Time in the above temporal interpolation may also be replaced by a multiplier function:

```
NODAL_BOUNDARY_CONDITION( "time-periodic nodal BC on inflow x-velocity" ) {
    variable                                = x_velocity
    type                                    = nodal_time_series
    nodes                                   = Read( "inflow.nbc" )
    nodal_values                            = Read( "inflow.bcu" )
    time_series                             = Read( "inflow.time" )
    time_series_variable                    = multiplier_function
    time_series_variable_multiplier_function  = "ramped time"
}
MULTIPLIER_FUNCTION( "ramped time" ) {
    type               = piecewise_linear
    curve_fit_values   {  0, 0 ;
                          0.1, 0.01 ;
                          0.2, 0.04 ;
                          0.3, 0.09 ;
                          0.4, 0.16 ;
                          0.5, 0.25 ;
```

```
                          10.5, 10.25 ; }
    curve_fit_variable = time
}
```

This multiplier function has the effect of "slowing down" time during the first 0.5 units of time. Such functions are often used to prevent poor physical or numerical behavior caused by the sudden imposition of a boundary condition. The multiplier function specified by *time_series_variable_multiplier_function* is independent of the one specified by *multiplier_function*, which may be given as well.

Nodal boundary conditions of types piecewise_linear and cubic_spline may be used to define boundary condition values as a function of a single independent variable. For example,

```
NODAL_BOUNDARY_CONDITION( "curve fit BC on inflow x-velocity" ) {
   variable           = x_velocity
   type               = cubic_spline
   nodes              = { 1 ; 3 ; 6 ; }
   curve_fit_values   = { -1, 0.0 ;  0, 2.5 ;  1, 0.0 ; }
   curve_fit_variable = y_coordinate
}
```

defines an x-velocity boundary condition as a function of the y-coordinate. In this example you get a parabolic velocity profile centered at y = 0, with a width of 2 and a center-line magnitude of 2.5. The *curve_fit_values* parameter is a two-column array corresponding to the independent variable and the boundary condition values. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, such as `inflow.bcu.fit`:

```
-1 0.0
 0 2.5
 1 0.0
```

and read by:

```
NODAL_BOUNDARY_CONDITION( "curve fit BC on inflow x-velocity" ) {
   variable           = x_velocity
   type               = cubic_spline
   nodes              = { 1 ; 3 ; 6 ; }
   curve_fit_values   = Read( "inflow.bcu.fit" )
   curve_fit_variable = y_coordinate
}
```

A nodal boundary condition of user_function type may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Guide* for a detailed description of user-defined functions.

For example, consider the case where the x-velocity has a parabolic profile at the inflow of a circular pipe. Then the input command may be given by:

```
NODAL_BOUNDARY_CONDITION( "UDF BC on inflow x-velocity" ) {
   variable       = x_velocity
   type           = user_funtion
   nodes          = Read( "inflow.nbc" )
   user_function  = "usrNodalBcExample"
   user_values    = { 0,  # y-center 0,  # z-center 1,  # radius 1 } # max velocity
```

```
}
```

where the user-defined function "usrNodalBcExample" may be implemented as follows.

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrNodalBcExample ) ; /* function prototype*/
   Void usrNodalBcExample (
   UdfHd            udfHd,                      /* Opaque handle for accessing data
 */
   Real*            outVec,                     /* Output vector
 */
   Integer          nItems,                     /* Number of BC nodes
 */
   Integer          vecDim                      /* = 1
 */
   ) {
   Integer          node ;                      /* a node counter
 */
   Real             radius ;                    /* pipe radius
 */
   Real             scale ;                     /* scaling factor
 */
   Real             ve10 ;                      /* velocity at center line
 */
   Real             y ;                         /* y-location
 */
   Real             yCenter ;                   /* y-coordinates of inflow center
 */
   Real             z ;                         /* z-location
 */
   Real             zCenter ;                   /* y-coordinates of inflow center
 */
   Real             crd ;                       /* coordinates
 */
   Real             usrVals ;                   /* user values
 */
   Real             xCrd ;                      /* x-coordinates
 */
   Real             yCrd ;                      /* y-coordinates
 */
   Real             zCrd ;                      /* z-coordinates
 */
udfCheckNumUsrVals( udfHd, 4 ) ;                /* check for error
 */
   usrVals          = udfGetUsrVals( udfHd ) ;  /* get the user vals
 */
   yCenter          = usrVals[0] ;              /* get y-center
 */
   zCenter          = usrVals[1] ;              /* get z-center
 */
   radius           = usrVals[2] ;              /* get pipe radius
 */
   vel0             = usrVals[3] ;              /* get max velocity
 */
   scale            = 1 / ( radius * radius )   /* for convenience
 */
   crd              = udfGetNbcCrd( udfHd ) ;   /* get the coord.
 */
   xCrd             = &crd[0*nItems] ;          /* localize x-coord.
 */
```

```
   yCrd            = &crd[1*nItems] ;          /* localize y-coord.
 */
   zCrd            = &crd[2*nItems] ;          /* localize z-coord.
 */
   for ( node = 0 ; node < nItems ; node++ ) {
       y           = yCrd[node] - yCenter ;
       z           = zCrd[node] - zCenter ;
       outVec[node] = vel0 * ( 1 - scale * ( y*y + z*z ) ) ;
    }
 } /* end of usrNodalBcExample() */
```

The dimension of the returned boundary condition vector, *outVec*, is the number of nodes.

The *auxiliary_nodes* parameter provides a mechanism to access data from other nodes in a user-defined function. The format of the input array is two dimensional where the first column contains the boundary condition nodes and the second and successive columns contain the auxiliary nodes which must be valid node numbers. Then for each node the user-defined function can obtain data from any of its auxiliary nodes through the use of functions such as `udfGetNbcAuxCrd` and `udfGetNbcAuxData`. For example,

```
NODAL_BOUNDARY_CONDITION( "BC based on other nodes" ) {
   ...
   type            = user_funtion
   auxiliary_nodes = { 1, 101, 201, 301 ;... }
   ...
}
```

In the above example, the auxiliary nodes of node one are 101, 201 and 301.

The *user_nodal_values* parameter extends user_values to each node. The format of the input array is the same as for *auxiliary_nodes* except the auxiliary nodes are replaced with user values. For example,

```
NODAL_BOUNDARY_CONDITION( "BC with individual coefs" ) {
   ...
   type              = user_funtion
   user_values       = { 1.0, 2.0, 3.0 }
   user_nodal_values = { 1, 4.0, 5.0 ;... }
}
```

Using both *auxiliary_nodes* and *user_nodal_values* enables the computation of interpolated and projected data inside of user functions.

The *scattered_data* type allows direct use of experimental or other sources of data specified at coordinates that do not coincide with nodes. The data is specified in *scattered_data* where the first three columns give the xyz coordinates in an orthonormal xyz coordinate system defined by you, and the last column contains the boundary condition data. The origin of the user coordinate system in the global system is given by *scattered_data_origin*. The three rows in *scattered_data_axes* gives the direction of the x, y, and z axes in the global system. If the user coordinate system is the same as the global coordinate system then these last two parameters do not need to be specified. For example, experimental temperature data may be used for inflow boundary conditions as follows:

```
NODAL_BOUNDARY_CONDITION( "experimental data for inflow temperature" ) {
   nodes                = Read( "channel.inflow.nbc" )
   variable             = temperature
   type                 = scattered_data
```

```
   scattered_data          = { 0.0, 0.0,
                               0.0, 100.0 ;
                               2.0, 0.0, 0.0, 102.0 ;
                               0.0, 1.0, 0.0, 105.0 ;
                               1.0, 1.5, 0.0, 107.0 ;
                               2.0, 1.0, 0.0, 110.0 ; }
   scattered_data_origin   = { 0, -10, 0 }
   scattered_data_axes     = { 0, 1, 0 ;
                               0, 0, 1 ;
                               1, 0, 0 ; }
}
```

This command defines a temperature boundary condition from a set of scattered data. Here, the data was measured in the xy-plane, while the inflow plane of the model is parallel to the yz-plane and shifted by -10 in the y axis.

The *scattered_data* data may be read from a file. For the above example, the experimental data may be placed in a file, such as `measured.temp.dat`:

```
0.0 0.0 0.0 100.0
2.0 0.0 0.0 102.0
0.0 1.0 0.0 105.0
1.0 1.5 0.0 107.0
2.0 1.0 0.0 110.0
```

and read by:

```
NODAL_BOUNDARY_CONDITION( "experimental data for inflow temperature" ) {
   nodes                 = Read( "channel.inflow.nbc" )
   variable              = temperature
   type                  = scattered_data
   scattered_data        = Read( "measured.temp.dat" )
   scattered_data_origin = { 0, -10, 0 }
   scattered_data_axes   = { 0, 1, 0 ;
                             0, 0, 1 ;
                             1, 0, 0 ; }
}
```

In general, the user coordinates are first transformed to the global coordinate system. Then a plane is fitted to these points and all the points are projected to this plane. In the present example the data is already planar so there is no approximation. The points are triangulated and a 2D mesh is constructed. Each boundary condition node is then projected to this mesh and the data linearly interpolated to produce the boundary condition value. If the projection falls outside of the constructed mesh, the closest edge of the mesh is found and linear interpolation is performed along the edge. This algorithm works well for planar or nearly planar user data. If the data are far from planar, the results may be unpredictable. The algorithm also guarantees all boundary condition values are bounded by the user data. The disadvantage is that the interpolation is faceted and not smooth.

The scattered_data_time_series type is a generalization of the scattered_data type to include a piecewise linear function of time at each spatial data point. This generalization is completely analogous to the generalization from nodal to nodal_time_series type described above. The time points are given by *time_series* and must be in ascending order. If the number of points in *time_series* is N, then the number of columns in *scattered_data* must be N+3, where the first three columns give the xyz user coordinates, and the last N columns contain the time series data. Piecewise linear interpolation of the *scattered_data* data is used for times within the time series, and the limit point values are used

for times outside the range of the time series. This time interpolation is done first for all the rows in *scattered_data* and then the spatial interpolation described in the previous paragraph is performed. For example,

```
NODAL_BOUNDARY_CONDITION( "time series experimental data for inflow temp" ) {
   nodes                  = Read( "channel.inflow.nbc" )
   variable               = temperature
   type                   = scattered_data_time_series
   scattered_data         = { 0.0, 0.0, 0.0, 100.0, 110.0, 120.0 ;
                              2.0, 0.0, 0.0, 102.0, 112.0, 122.0 ;
                              0.0, 1.0, 0.0, 105.0, 115.0, 125.0 ;
                              1.0, 1.5, 0.0, 107.0, 117.0, 127.0 ;
                              2.0, 1.0, 0.0, 110.0, 120.0, 130.0 ; }
   time_series            = { 0.0, 0.5, 1.0 }
   scattered_data_origin  = { 0, -10, 0 }
   scattered_data_axes    = { 0, 1, 0 ;
                              0, 0, 1 ;
                              1, 0, 0 ; }
}
```

The parameters *time_series_variable*, *time_series_cyclic_period*, and *time_series_variable_multiplier_function* may be used with scattered_data_time_series type in exactly the same way as they are used with nodal_time_series type above.

The external_file type allows the boundary conditions to be modified each time step during the run. This is normally used in conjunction with another program that monitors the AcuSolve solution and creates new boundary conditions in response. For example,

```
NODAL_BOUNDARY_CONDITION( "inflow x-vel" ) {
   variable                  = x_velocity
   nodes                     = Read("inflow.nbc")
   type                      = external_file
   external_file             = "inflow.exf"
   external_file_nodal_offset = Read("inflow.loc")
   external_file_evaluation   = once_per_time_step
}
```

At the beginning of each time step, the boundary condition values for nodes in "inflow.nbc" and variable x_velocity are read from "inflow.exf". The file is deleted after it is read by all nodal boundary condition commands referencing this file. This deletion may be used as a signal that the next file may be written. The appearance of the file signals AcuSolve that the data is ready for reading. An error is issued and AcuSolve exits if the file does not appear within 600 seconds. In order to avoid reading incomplete data, it is recommended that a temporary file be used to hold the data which is then renamed appropriately after it is closed. The values are read as native 8-byte binary floating point numbers. The byte offset for each node (from the beginning of the file) is specified in the "inflow.loc" file. This is a 2-column file, where the first column contains the nodes given in *nodes* parameter ("inflow.nbc" in this case) and the second column specifies the byte offset, typically set to 0, 8, 16, 24, and so on.

By default, all boundary conditions are specified with respect to the fixed (or "laboratory") reference frame. If the REFERENCE_FRAME command is used to define the rotational body forces (this is recommended, as opposed to using the ROTATION_FORCE command), then the velocity boundary conditions on a rotating wall are non-trivial. However, the *reference_frame* parameter provides a convenient way of specifying these conditions. Instead of the fixed frame, velocity boundary conditions are applied in the given frame. In many cases, a zero type can then be used. See the REFERENCE_FRAME

command for more details and an example. The *reference_frame* parameter has no effect on variables other than velocity components.

The *multiplier_function* parameter may be used to uniformly scale the nodal boundary condition values. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file. For example, to linearly increase the inflow parabolic x-velocity from zero at time zero to the full amount at time one, the above boundary condition may be changed to:

```
NODAL_BOUNDARY_CONDITION( "UDF BC on inflow x-velocity" ) {
   variable            = x_velocity
   type                = user_function
   nodes               = Read( "inflow.nbc"
   user_function       = "usrNodalBcExample"
   user_values         = { 0,   # y-center
                           0,   # z-center
                           1,   # radius
                           1 }  # max velocity
   multiplier_funtion  = "ramped"
}
   MULTIPLIER_FUNCTION( "ramped" ) {
   type               = piecewise_linear
   curve-fit_values   = { 0, 0 ; 10, 1 }
   curve_fit_variable = time
}
```

A nodal boundary condition of match_mesh_velocity type is used to ensure conservation of mass in moving mesh problems. On a solid wall conservation requires that the normal component of flow velocity equals the normal velocity of the wall itself (that is, the normal velocity of the mesh). If the mesh velocity is specified, then it is a trivial matter to use the same boundary condition for the flow velocity. However, if the mesh displacement is specified (this is usually the best strategy), the mesh velocity is determined internally (it is not the analytic time derivative of the displacement) and a match_mesh_velocity type is needed. This type will cause the given component of flow velocity to match the corresponding component of mesh velocity. You are responsible for ensuring the normal component of flow velocity has the appropriate boundary condition on it. For example, on a wall parallel to the x-z plane moving in the y direction, you may have:

```
NODAL_BOUNDARY_CONDITION( "accelerating wall" ) {
   variable            = y_mesh_displacement
   type                = constant
   constant_value      = 5.
   nodes               = { 1 ; 3 ; 6 ; }
   multiplier_function  = "accelerate"
}
MULTIPLIER_FUNCTION( "accelerate" ) {
   type                = cubic_spine
   curve_fit_values    = { 0, .0 ;
                           1, .1 ;
                           2, .4 ;
                           3, .9 ; }
   curve_fit_variable  = time
}
NODAL_BOUNDARY_CONDITION( "match y velocity" ) {
   variable            = y_velocity
   type                = match_mesh_velocity
   nodes               = { 1 ; 3 ; 6 ; }
}
```

Note: The match_mesh_velocity type may only be used with velocity-component variables and may not be combined with a multiplier function.

Mesh displacement boundary conditions may be scaled with mesh_motion. The value of this parameter refers to the user-given name of a MESH_MOTION command in the input file. This command is evaluated and the appropriate component of the resulting mesh displacement vector is used to scale the boundary condition. The mesh_motion type is shorthand for *type* = constant constant_value = 1, except that it is not compatible with *mesh_motion* = none. It is possible to specify both mesh_motion and *multiplier_function*; they are simply multiplied together and the result used to scale the boundary condition. For example,

```
NODAL_BOUNDARY_CONDITION( "buffer region for mesh displacement" ) {
   variable            = mesh_x_displacement
   type                = piecewise_linear
   nodes               = Read( "buffer.nbc" ) # all nodes in -10 < x <10
   curve_fit_values    = { -10, 0.0 ; -1, 1.0 ;  1, 1.0 ; 10, 0.0 }
   curve_fit_variable  = x_reference_coordinate
   mesh_motion         = "moving body"
}
MESH_MOTION( "moving body" ) {
   type                = rigid_body
   ...
}
```

If the rigid body is originally within -1 < x < 1 then the mesh in this area follows the rigid body without distortion. The distortion takes place in the larger buffer area away from the body and smoothly ramps down to zero at x = ±10.

Note: This strategy also allows the use of *mesh*=specified in the EQUATION command and thus avoids solving an ALE equation.

Fluid-structure interaction (FSI) problems use the flexible_body type in a manner similar to the rigid_body type. The *flexible_body* parameter references a FLEXIBLE_BODY command that is used to drive either a mesh displacement or velocity boundary condition here, depending on the modeling type specified by the *equation* parameter of FLEXIBLE_BODY. The eigenvectors required by a FSI calculation are given by *nodal_modes* here. See the FLEXIBLE_BODY command for a detailed description of FSI modeling and an example.

The guide_surface type is used along with a velocity variable or mesh_direction_displacement variable to allow mesh nodes to slip along a surface specified by the *guide_surface* parameter. For flat surfaces this can be accomplished by specifying a constant direction type. But for curved surfaces the appropriate direction may change every step as the fluid node moves along the surface. The guide_surface type overcomes this difficulty by continuously recomputing which guide surface facet each node is in (or closest to) and then using the normal to that facet. See the GUIDE_SURFACE command for more details. The boundary conditions on velocity are more subtle. Consider a case similar to the moving surface example given in SIMPLE_BOUNDARY_CONDITION, but with a no-slip wall. The NODAL_BOUNDARY_CONDITION command would be used as follows:

```
MESH_MOTION( "boat motion" ) {
   type                        = rigid_body_dynamic
   rigid_body_mass             = 100
```

```
   ...
   rigid_body_surface_outputs  = { "ocean: boat hull" }
}
SURFACE_OUTPUT( "ocean: boat hull" ) {
   surfaces                    = Read( "ocean.hull.ebc" )
   shape                       = three_node_triangle
   element_set                 = "ocean"
}
GUIDE_SURFACE( "moving boat surface" ) {
   ...
   element_set                 = "boat"
   ...
   mesh_motion                 = "boat motion"
}
NODAL_BOUNDARY_CONDITION( "slip condition for mesh" ) {
   variable                    = mesh_direction_displacement
   type                        = guide_surface
   nodes                       = Read( "boat.wall.nbc" )
   guide_surface               = "moving boat surface"
}
NODAL_BOUNDARY_CONDITION( "no-slip condition for x-velocity" ) {
   variable                    = x_velocity
   type                        = guide_surface
   nodes                       = Read( "boat.wall.nbc" )
   guide_surface               = "moving boat surface"
}
NODAL_BOUNDARY_CONDITION( "no-slip condition for y-velocity" ) {
   variable                    = y_velocity
   type                        = guide_surface
   nodes                       = Read( "boat.wall.nbc" )
   guide_surface               = "moving boat surface"
}
NODAL_BOUNDARY_CONDITION( "no-slip condition for z-velocity" ) {
   variable                    = z_velocity
   type                        = guide_surface
   nodes                       = Read( "boat.wall.nbc" )
   guide_surface               = "moving boat surface"
}
NODAL_BOUNDARY_CONDITION( "match normal velocity to mesh" ) {
   variable                    = direction_velocity
   type                        = match_mesh_velocity
   nodes                       = Read( "boat.wall.nbc" )
}
```

Four boundary conditions are given for the velocity. The direction_velocity condition is necessary to conserve mass, but using a match_mesh_velocity type for the velocity vector would be wrong since the mesh slips and the velocity does not. This apparent over-specification is resolved internally by giving the direction_velocity condition a higher precedence. Effectively, the velocity produced by the x, y, and z conditions is projected to the nearest velocity that satisfies conservation of mass. A slip velocity condition is obtained by removing the three commands for the x, y, and z velocity components.

The external_code type is only used in Direct Coupling Fluid Structure Interaction (DC-FSI) problems. It is not necessary even then, since mesh displacement boundary conditions are set in the `EXTERNAL_CODE_SURFACE` command for nodes on the corresponding surface. However, it is sometimes desirable to have nodes not on such surfaces to also follow the solid deformation. For example, this may be accomplished by:

```
NODAL_BOUNDARY_CONDITION( "match normal mesh displacement to solid" ) {
   variable            = mesh_direction_displacement
```

```
   type                = external_code
   nodes               = Read( "external.nbc" )
   external_code_tags  = { "Surface 1"; }
   coupling_type       = surface
}
```

Assuming the specified nodes are not on the external surface, this command constrains the nodes to remain a constant distance away from the deforming solid, but allows them to slide relative to the solid. In this case, the *external_code_tags* option was used to tell the external code to force these nodes to move with a surface tagged as "Surface 1". This parameter is optional and is only necessary when the standard mapping procedure between fluid and solid codes produces an ambiguous surface association for the nodes of the fluid. If all three components of the mesh displacement are specified (in three commands) then the nodes will move with the solid. In both cases the point on the solid surface closest to the given node is used to determine the mesh motion and the normal direction.

The boundary condition variables direction_velocity, mesh_direction_displacement and mesh_direction_velocity require specification of a direction type, except when the type is match_mesh_velocity, guide_surface, or external_code. For example,

```
NODAL_BOUNDARY_CONDITION( "slip velocity -- constant direction" ) {
   variable            = direction_velocity
   type                = zero
   nodes               = Read( "wall.nbc" )
   direction_type      = constant
   constant_direction  = { 1, 1, 0 }
}
```

imposes a zero boundary condition on the (1,1,0) component of velocity of all nodes given in the "wall.nbc" file. The vector *constant_direction* specifies the direction in the global xyz coordinate system. The direction vector is normalized before use.

The direction type of *constant_direction* specifies the same direction vector for all nodes in the boundary condition set, as in the above example.

The direction type of nodal specifies a different direction for each node in the set. For example,

```
NODAL_BOUNDARY_CONDITION( "slip velocity -- nodal direction" ) {
   variable          = direction_velocity
   type              = zero
   nodes             = { 1 ; 3 ; 6 ; }
   direction_type    = nodal
   nodal_directions = { 1, 1, 1, 1 ;
                        3, 1, 1, 0 ;
                        6, 1, 1,-1 ; }
}
```

The *nodal_directions* parameter is a four-column array corresponding to node numbers and the x, y, and z components of the direction vectors. The node numbers must match those given by the nodes parameter. No node may be missing. All direction vectors are normalized before use.

The nodal directions may be read from a file. For the above example, the directions may be placed in a file, such as `slip.dir`:

```
1    1    1   1
3    1    1   0
6    1    1  -1
```

and read by:

```
NODAL_BOUNDARY_CONDITION( "slip velocity -- nodal direction" ) {
   variable          = direction_velocity
   type              = zero
   nodes             = { 1 ; 3 ; 6 ; }
   direction_type    = nodal
   nodal_directions  = Read( "slip.dir" )
}
```

The direction type of normal specifies that the normal direction to the boundary of the mesh is to be used for the direction vector of the boundary condition. The normal direction is computed as the area-weighted average of the neighboring element surface outward normals. The normal direction of each node must be unambiguous; however this algorithm does not guarantee such an outcome. For example,

```
NODAL_BOUNDARY_CONDITION( "slip velocity -- normal direction" ) {
   variable       = direction_velocity
   type           = zero
   nodes          = { 1 ;
                      3 ;
                      6 ; }
   direction_type = normal
}
```

computes the normal direction to nodes one, three, and six, and imposes a zero velocity in that normal direction.

In some circumstances it is necessary to "turn off" or "deactivate" a previously-defined nodal boundary condition. This is accomplished by setting the active flag through *active_type*. A value of all is the default and means that all nodes in the set are active (active flag = 1) and will have the boundary condition imposed on them. A value of none means that no node is active (active flag = 0). For example, assume that one run was made with the following commands to model a no-slip wall:

```
NODAL_BOUNDARY_CONDITION( "wall x-velocity" ) {
   variable       = x_velocity
   nodes          = Read( "wall.nbc" )
}
NODAL_BOUNDARY_CONDITION( "wall y-velocity" ) {
   variable       = y_velocity
   nodes          = Read( "wall.nbc" )
}
NODAL_BOUNDARY_CONDITION( "wall z-velocity" ) {
   variable       = z_velocity
   nodes          = Read( "wall.nbc" )
}
```

Then the entire input file to convert this wall to a slip wall on restart may be given by (assuming the normal to the wall is in the y direction):

```
RESTART {}
NODAL_BOUNDARY_CONDITION( "wall x-velocity" ) {
   active_type       = none
}
NODAL_BOUNDARY_CONDITION( "wall z-velocity" ) {
   active_type       = none
}
RUN {}
```

An inflow active type is typically used to stabilize outflow boundary conditions. For those nodes in the set where the normal velocity is out of the fluid domain (that is, the desired situation for outflow boundaries) then the boundary condition is turned off. However, for nodes with reverse flow (normal velocity into the fluid domain), the boundary condition is activated. This is particularly useful for scalar variables and both components of the tangential velocity, which would otherwise be undefined at reverse flow nodes and be potential sources of instability. A no_outflow active type is the same as inflow except nodes with zero normal velocity are also activated.

A user_function active type may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Guide* for a detailed description of user-defined functions. Values of the active flag between zero and one are allowed; such values are used to linearly interpolate between the previous value of the variable and its given boundary condition, with the result imposed as the actual boundary condition.

For example, consider the case where the velocity boundary conditions are to be turned on in a wave-like manner in order to (crudely) model a wall moving in the positive x direction through the mesh at speed c:

$$\text{active flag} = \begin{cases} 0 & \text{if } x > x_w + \varepsilon \\ 1 & \text{if } x < x_w \\ 1 - (x - x_w)/\varepsilon & \text{if } x_w + \varepsilon \geq x \geq x_w \end{cases} \qquad (128)$$

where $x_w = x_0 - ct$ is the position of the wall at time t, x_0 is the initial position and ε is a small positive parameter (the "mushy interval") used to smooth the transition between no boundary condition and an active one. The input command for the x-velocity may be given by:

```
NODAL_BOUNDARY_CONDITION( "UDF moving wall for x-velocity" ) {
   variable              = x_velocity
   type                  = constant
   constant_value        = 5
   nodes                 = Read( "moving_wall.nbc" )
   active_type           = user_function
   active_user_function  = "usrActiveFlagExample"
   active_user_values    = { -100,    # initial wall location
                             5,    # wall speed
                             0.1 } # mushy interval
}
```

with similar commands for the y- and z-velocities. The user-defined function "usrActiveFlagExample" may be implemented as follows.

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrActiveFlagExample ) ;     /* function prototype              */
   Void usrActiveFlagExample (
   UdfHd        udfHd,                      /* Opaque handle for accessing data */
   Real*        outVec,                     /* Output vector                   */
   Integer      nItems,                     /* Number of BC nodes              */
   Integer      vecDim                      /* = 1                             */
   ) {
   Integer      node ;                      /* a node counter                  */
   Real         x0 ;                        /* initial location of wall        */
   Real         speed ;                     /* speed of wall                   */
```

```
   Real         mushy ;                    /* mushy interval                  */
   Real         xWall ;                    /* location of wall                */
   Real         time ;                     /* time                            */
   Real         delta_x ;                  /* distance from wall              */
   Real*        crd ;                      /* coordinates                     */
   Real*        usrVals ;                  /* user values                     */
   Real*        xCrd ;                     /* x-coordinates                   */
udfCheckNumUsrVals( udfHd, 3 ) ;           /* check for error                 */
   time         = udfGetTime( udfHd ) ;    /* get the time                    */
   usrVals      = udfGetUsrVals( udfHd ) ; /* get the user vals               */
   x0           = usrVals[0] ;             /* get initial loc                 */
   speed        = usrVals[1] ;             /* get speed                       */
   mushy        = usrVals[2] ;             /* get mushy interval              */
   crd          = udfGetNbcCrd( udfHd ) ;  /* get the coord.                  */
   xCrd         = &crd[0*nItems] ;         /* localize x-coord.               */
   xWall        = x0 + speed * time ;      /* location of wall                */
   for ( node = 0 ; node < nItems ; node++ ) {
       delta_x = xCrd[node] - xWall ;
       if ( delta_x >= mushy ) {
          outVec[node] = 0 ;
       } else if ( delta_x <= 0 ) {
          outVec[node] = 1 ;
       } else {
         outVec[node] = 1. - delta_x / mushy ;
       }
    }
} /* end of usrActiveFlagExample() */
```

The dimension of the returned active flag vector, *outVec*, is the number of nodes.

A no_interface active type is used to automatically turn on and off boundary conditions in the presence of interfaces. The boundary condition is applied (active flag = 1) if the node is not part of any surface of an INTERFACE_SURFACE set or if it does not find a contact surface of the appropriate medium. That is, each node will fall into one of five possible cases:

- The node is not in any INTERFACE_SURFACE set; active flag = 1.
- The node is a part of an INTERFACE_SURFACE set, but it does not find a contact surface; active flag = 1.
- The node finds a contact surface of fluid medium; active flag = 0.
- The node finds a contact surface of solid/shell medium, and *variable* is defined on the solid/shell side (currently only temperature); active flag = 0.
- The node finds a contact surface of solid/shell medium, and *variable* is not defined on the solid/shell side (that is, it is not temperature; active flag = 1.

The pressure may be imposed either strongly via the NODAL_BOUNDARY_CONDITION command or weakly via the ELEMENT_BOUNDARY_CONDITION command. The latter command should be used in all cases, except when a boundary condition is imposed on the normal component of velocity on all boundary nodes (this includes defining boundary conditions on all three velocity components). In this case pressure is defined up to a constant, and it is advisable to specify pressure at one and only one nodal point to define that constant.

Below is an example to setup synthetic turbulence for a large eddy simulation inflow condition:

```
NODAL_BOUNDARY_CONDITION( "some x-velocity" ) {
   Synthetic_turbulence  = on
}
```

Note: *synthetic_turbulence* is normally set to on for all three velocity components and two turbulence profiles corresponding to the *synthetic_turbulence_input_type* defined in *turbulence_model_parameters*.

NODAL_RESIDUAL_OUTPUT

Specifies parameters for outputting the nodal residual.

Type

AcuSolve Command

Syntax

NODAL_RESIDUAL_OUTPUT {parameters}

Qualifier

This command has no qualifier.

Parameters

output_frequency or *out_freq* *(integer) >=0[=0]*
: Time step frequency at which to output the nodal residual. If zero, this option is ignored.

output_time_interval or *out_intv* *(real) >=0 [=0]*
: Time frequency at which to output the nodal residual. If zero, this option is ignored.

num_saved_states or *save* *(integer) >=0 [=0]*
: Number of nodal residual files to keep. If zero, all nodal residual files are kept.

Description

This command specifies the parameters for writing the nodal residual to disk. For example,

```
NODAL_RESIDUAL_OUTPUT {
   output_frequency      = 100
   output_time_interval  = 0
   num_saved_states      = 2
}
```

writes a nodal residual file to disk every 100 time steps as well as the last time step. Only the last two files are kept. Once the third nodal residual file is written to disk, the first nodal residual file is removed to preserve disk space.

NODAL_RESIDUAL_OUTPUT is rarely needed or used. It is used mainly for debugging purposes. At steady state, or at the end of each time step for transient problems, the residual for each equation at each node should be zero. In practice, convergence is assumed if the normalized residuals are within the specified tolerances; see the TIME_SEQUENCE command. If there are difficulties in achieving convergence, the output of this command may be useful in determining where the problem lies.

If either *output_frequency* or *output_time_interval*output_time_interval is non-zero, a nodal residual file will be written at the end of the run. If both are zero, no nodal residual files are created.

Run times may not coincide with *output_time_interval*. In this case, at every time step which passes through a multiple of *output_time_interval*, nodal residual files are written.

The *num_saved_states* parameter indicates the number of nodal residual files to save. Once the (*num_saved_states* + 1)th nodal residual file is written, the first one is removed. When running in parallel, once all files are written (in parallel), then the old ones are removed (in parallel).

NODAL_OUTPUT

Specifies nodal output parameters for solution variables.

Type

AcuSolve Command

Syntax

NODAL_OUTPUT {parameters}

Qualifier

This command has no qualifier.

Parameters

output_frequency or *out_freq* *(integer) >=0 [=1000]*
: Time step frequency at which to output nodal solutions. If zero, this option is ignored.

output_time_interval or *out_intv* *(real) >=0 [=0]*
: Time frequency at which to output nodal solutions. If zero, this option is ignored.

output_initial_condition *(boolean) [=off]*
: Flag specifying whether or not to output the initial condition.

continuous_output *(boolean) [=off]*
: Flag specifying whether or not to output nodal solutions every step, saving only the current and previous solution.

num_saved_states or *save* *(integer) >=0 [=0]*
: Number of nodal solution files to keep. If zero, all are kept.

case_output_frequency or *cycle_output_frequency* *>=0 [=10]*
: Frequency at which to output optimization case solutions when optimization under EQUATION is set to on. If zero, this option is ignored.

output_initial_case or *output_initial_cycle* *(boolean) [on]*
: Flag specifying whether or not to output initial case when optimization under EQUATION is set to on.

Description

This command specifies the frequency at which the nodal solutions are written into a results database. For example,

```
NODAL_OUTPUT {
   output_frequency     = 10
   output_time_interval = 2.5
}
```

specifies that the nodal solutions are to be output at every 10 time steps, at every 2.5 units of run time and at the last time step.

Once the solution has been written to disk, it can be translated to other formats using the AcuTrans program and other post-processing modules; see the *AcuSolve Programs Reference Manual* for details.

If either *output_frequency* or *output_time_interval* is non-zero, the solution will be output at the end of the run. If both are zero, no data are written to disk.

Run times may not coincide with *output_time_interval*. In this case, at every time step which passes through a multiple of *output_time_interval*, the solution is output.

The initial conditions may be output as time step 0 by setting *output_initial_condition* to on. This nodal solution will not reflect any boundary conditions.

The *num_saved_states* parameter indicates the number of nodal solution files to save. Once the (*num_saved_states* + 1)th file is written, the first one is removed. When running in parallel, once all files are written (in parallel), then the old ones are removed (in parallel).

A convenient way to output nodal solutions continuously is to set *continuous_output* to on. This will output the solution every step but save the solution files for only the current and previous steps. It is equivalent to setting *output_frequency*=1 and *num_saved_states*=2.

PERIODIC_BOUNDARY_CONDITION

Specifies general periodic boundary conditions on pairs of nodes.

Type

AcuSolve Command

Syntax

```
PERIODIC_BOUNDARY_CONDITION("name") {parameters...}
```

Qualifier

User-given name.

Parameters

`variable` or *`var`* *(enumerated) [=all]*

Periodic boundary condition variable.

all	All variables. May be used only with periodic, axisymmetric and two_reference_frames types.
velocity or vel	Velocity vector.
pressure or pres	Pressure.
temperature or temp	Temperature.
eddy_viscosity or eddy	Turbulence kinematic eddy viscosity.
kinetic_energy or tke	Turbulence kinetic energy.
eddy_frequency or tomega	Turbulence eddy frequency.
dissipation_rate	Turbulence dissipation rate.
intermittency	Transition intermittency.
transition_re_theta	Transition Re-Theta.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.

species_7 or spec7	Species 7.
species_8 or spec8	Species 8.
species_9 or spec9	Species 9.
viscoelastic_stress or vest	Viscoelastic stresses.
field	Volume fraction of field in multi field for levelset and algebraic Eulerian problems or mass fraction of field in multi field for the humid air model.
mesh_displacement or mesh_disp	Mesh displacement vector.

precedence (integer) [=1]
Precedence of this boundary condition set with respect to other sets. The set with the highest value has precedence.

type (enumerated) [=periodic]
Type of the periodic boundary condition.

periodic or zero	Simple periodic condition.
axisymmetric	Axisymmetric condition. Requires *rotation_axis*.
two_reference_frames	Interface condition between two regions with different reference frames. Requires *rotation_force_1* and *rotation_force_2*. Obsolete parameter.
constant_offset	Periodic condition with a constant offset. Requires *constant_value* for scalar variables and *constant_values* for vector variables.
nodal_pair_offset	Periodic condition with offset given on a nodal-pair basis. Requires *nodal_pair_values*.
user_function_offset	Periodic condition with offset given by a user-defined function. Requires user_function, user_values and user_strings.
constant_coefficients	General two-node constraint with constant coefficients. Requires *constant_values*.
nodal_pair_coefficients	General two-node constraint with coefficients given on a nodal-pair basis. Requires *nodal_pair_values*.
user_function_coefficients	General two-node constraint with coefficients given by a user-defined function. Requires user_function, user_values and user_strings.
single_unknown_offset	Periodic condition with a single unknown offset.

single_unknown_ratio	Periodic condition with a single unknown ratio.

nodal_pairs or *pairs* *(array) [no default]*
List of nodal point pairs in this set.

rotation_axis or *axis* *(array) [={0,0,0;0,0,1}]*
Axis of rotation. Used with axisymmetric type.

rotation_force_1 or *rot_1* *(string) [=none]*
User-given name of the rotational force command of the first set of nodes. If none, stationary frame of reference is assumed. Used with two_reference_frames type. Obsolete parameter.

rotation_force_2 or *rot_2* *(string) [=none]*
User-given name of the rotational force command of the second set of nodes. If none, stationary frame of reference is assumed. Used with two_reference_frames type. Obsolete parameter.

constant_value or *value* *(real) [=0]*
Constant value of offset. Used with constant_offset type for scalar variables.

constant_values or *values* *(array) [={0,0,0}]*
Constant values of offset or the constant coefficients. Used with constant_offset type for vector variables and with constant_coefficients type.

field *(string) [no default]*
Value of the periodic boundary condition for field. Used with variable field.

nodal_pair_values *(array) [no default]*
Constraint values for nodal-pairs. Used with nodal_pair_offset and nodal_pair_coefficients types.

user_function or *user* *(string) [no default]*
Name of the user-defined function. Used with user_function_offset and user_function_coefficients types.

user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function_offset and user_function_coefficients types.

user_strings *(list) [={}]*
Array of strings to be passed to the user-defined function. Used with user_function_offset and user_function_coefficients types.

multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the offset values of the periodic boundary condition. If none, no scaling is performed.

active_type *(enumerated) [=all]*
Type of the active flag. Determines which nodal pairs in this set will have this periodic boundary condition imposed.

all	All nodal pairs in this set are active.
none	No nodal pair in this set are active.
user_function or user	User-defined function. Requires *active_user_function*, *active_user_values* and *active_user_strings*.

active_user_function (string) [no default]
Name of the user-defined function for the active flag. Used with user_function active type.

active_user_values (array) [={}]
Array of values to be passed to the active flag user-defined function. Used with user_function active type.

active_user_strings (list) [={}]
Array of strings to be passed to the active flag user-defined function. Used with user_function active type.

Description

This command specifies generalized periodic boundary conditions for a solution variable (or all variables) on a set of nodal pairs. For example,

```
PERIODIC_BOUNDARY_CONDITION( "inflow/outflow periodicity" ) {
   variable     = all
   type         = periodic
   nodal_pairs  = { 1, 101, 201 ;
                    2, 102, 202 ;
                    3, 103, 203 ; }
}
```

defines a simple periodic boundary condition on all solution fields of the problem. This command specifies that the solution at node 101 is equal to that of node 201, and similarly that the solution of node 102 is equal to that of node 202, and so on.

The *variable* parameter specifies the solution field to which the periodic boundary condition applies. If the problem does not solve for such a variable, the periodic boundary condition is ignored. See the STAGGER command. The all variable may be used to specify periodic boundary condition on all variables in the problem. This variable is only allowed with types periodic, axisymmetric, and two_reference_frames. A periodic boundary condition on a scalar variable, such as temperature, defines a single constraint. However, for vector variables, such as velocity, a periodic boundary condition imposes three constraints, typically in the x, y, and z directions.

The *nodal_pairs* parameter is a three-column array corresponding to a unique, within this set, nodal-pair number, and node one and two of the pair, one pair per row. The node numbers must be valid numbers, as given by the COORDINATE command. AcuSolve properly handles duplicate node pairs and nodes periodic with themselves.

The nodal pairs may be read from a file. For the above example, the pairs may be placed in a file, such as inflow-outflow.pbc:

```
1      101          201
2      102          202
2      103          203
```

and read by:

```
PERIODIC_BOUNDARY_CONDITION( "inflow/outflow periodicity" ) {
   variable     = all
   type         = periodic
   nodal_pairs  = Read( "inflow-outflow.pbc" )
}
```

For a given variable, a given node may be included in more than one periodic boundary condition pair. In this case, enough periodic conditions must be supplied to uniquely specify the entire condition set. These conditions may be imposed via a single or multiple `PERIODIC_BOUNDARY_CONDITION` commands. For example, nodal pairs (1,2) and (3,4) may have simple periodic conditions, and nodal pairs (1,3) and (2,4) have axisymmetric conditions, with the same axis of rotation.

> **Note:** This is an over-specification of constraints (but consistent).

Any potential inconsistency in multiply-specified periodic boundary conditions on a given node is resolved through precedence. The constraints with higher values of *precedence* take precedence over constraints with lower values. For constraints with the same *precedence* value, the periodic boundary condition sets specified later take precedence over the ones specified earlier. In all cases, nodal boundary conditions take precedence over periodic boundary conditions.

A periodic type periodic boundary condition specifies that the values of a variable on two nodes are equal, as in the above example. In case of a vector variable, each component of the vector on the two nodes are equal.

An axisymmetric periodic boundary condition specifies that a vector field of two nodes be equal in a rotated coordinate system defined by an axis of rotation. The axis of rotation is given by *rotation_axis*. This parameter is a 3x2 array defining two points, p_1 and p_2, on the axis of rotation. The points are defined in the global xyz coordinate system. This provides a cylindrical coordinate system for each node of a nodal pair, n_1 and n_2 , where **a**, **r** and θ are the axial, radial and tangential directions, respectively; see the image below.

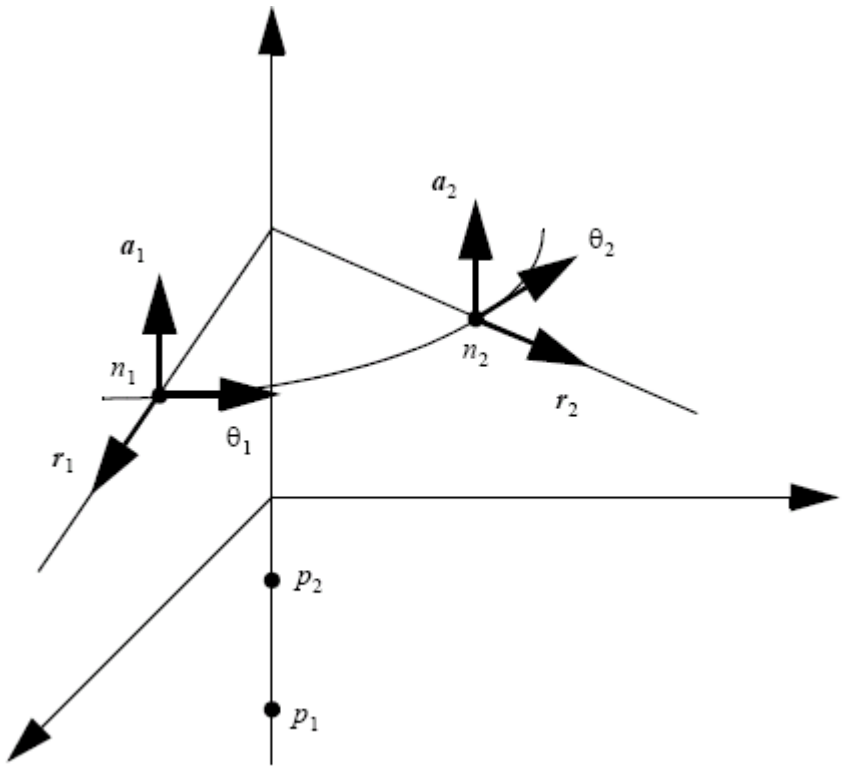


Figure 9: Axisymmetric Condition on a Vector Field

Let u_1 and u_2 be the velocity of points one and two, respectively. Then the axisymmetric conditions on the velocity field may be written as

$$\begin{aligned} \boldsymbol{u}_2 \cdot \boldsymbol{a}_2 &= \boldsymbol{u}_1 \cdot \boldsymbol{a}_1 \\ \boldsymbol{u}_2 \cdot \boldsymbol{r}_2 &= \boldsymbol{u}_1 \cdot \boldsymbol{r}_1 \\ \boldsymbol{u}_2 \cdot \theta_2 &= \boldsymbol{u}_1 \cdot \theta_1 \end{aligned} \qquad (129)$$

For example,

```
PERIODIC_BOUNDARY_CONDITION( "a pie cut axisymmetric constraint" ) {
   variable      = all
   type          = axisymmetric
   nodal_pairs   = { 1, 1, 2 ; 2, 11, 12 ; 3, 21, 22 ; }
   rotation_axis = { 0, 0, 0 ; 0, 0, 1 ; }
}
```

defines an axis of rotation in the global z-direction. The rotated velocity vectors of nodes 1 and 2 are equal. Similar conditions apply to nodal pairs (11,12) and (21,22). For a scalar variable the axisymmetric periodic boundary condition type is equivalent to the periodic type.

A two_reference_frames periodic boundary condition type is used to interface two regions with different rotating frames of reference that are specified through the ROTATION_FORCE command. This type is obsolete and support for it will be removed eventually. See the REFERENCE_FRAME command for the

preferred, and much easier, way of specifying rotating forces and reference frames. At the interface boundaries, there must be two sets of nodes with the same nodal coordinates. Each set belongs to one of the rotating frames of reference. The assignment is implicitly specified by the reference frame of the elements referencing the nodes. For example, consider the case where one element set is rotating in the y-direction, while the other is stationary:

```
COORDINATE {
   coordinates      = { 1, 1.,
                        0., 0. ;
                       11, 1., 0., 0. ; # same as node 1
                       ... }
}
ROTATION_FORCE( "rotating frame" ) {
   centrifugal       = on
   coriolis          = on
   center            = { 0, 0, 0 }
   angular_velocity  = { 0, 1, 0 }
}
BODY_FORCE( "my body force" ) {
   rotation_force    = "rotating frame"
}
ELEMENT_SET( "fluid in rotating frame of reference" ) {
   elements          = { 1, 1, 2, 3, 4 ;
                         ... }
   body_force        = "my body force"
   ...
   }
ELEMENT_SET( "fluid in stationary frame of reference" ) {
   elements          = { 1, 11, 45, 46, 47 ;
                         ... }
   body_force        = none
   ...
}
PERIODIC_BOUNDARY_CONDITION( "interface between two frames of reference" ) {
   variable          = all
   type              = two_reference_frames
   nodal_pairs       = { 1, 1, 11 ;
                         ... }
   rotation_force_1  = "rotating frame"
   rotation_force_2  = none
}
```

Nodes one and 11 have the same nodal coordinates. Node one is referenced by the element set "fluid in rotating frame of reference" which is in a rotating frame of reference given by "rotating frame"; and node 11 is referenced by the element set "fluid in stationary frame of reference" which is stationary. The given two_reference_frames periodic boundary condition specifies that the velocities of nodes 1 and 11 transformed to the global stationary frame of reference are equal.

Let **u**1 and **u**2 , respectively, be the velocities of the first and second nodes. These node must have common coordinates, denoted here by **x**. Also, let these nodes be in frames of reference rotating at ω_1 and ω_2 radians per unit time and centered at 0_1 and 0_2, respectively. Then the velocity periodic boundary condition reduces to:

$$\boldsymbol{u}_2 - \omega_2 \times (\boldsymbol{x} - \boldsymbol{o}_2) = \boldsymbol{u}_1 - \omega_1 \times (\boldsymbol{x} - \boldsymbol{o}_1) \qquad (130)$$

For all other variables, the two_reference_frames periodic boundary condition reduces to the simple equality condition. If n > 2 regions with different frames of reference intersect at a point or an edge,

one set of nodes for each region is needed and at least n - 1 conditions must be specified. Depending on the particular problem, there are typically several geometrical constraints on the relationship between the parameters of the reference frames and the surface separating the element sets associated with each reference frame. For instance, the velocity at **x** due to the reference frame motion itself, the second term on each side in the equation above, may need to be parallel to this surface. Since these constraints are not general, AcuSolve cannot check for them and so the responsibility for ensuring geometric compatibility between reference frames falls to you.

A constant_offset periodic boundary condition specifies that the nodal values of the second set are equal to those of the first set plus a constant value. For example,

```
PERIODIC_BOUNDARY_CONDITION( "inflow/outflow pressure drop" ) {
   variable       = pressure
   type           = constant_offset
   nodal_pairs    = { 1, 101, 201 ; 2, 102, 202 ; 3, 103, 203 ; }
   constant_value = 2
}
```

The pressure of node 201 equals to those of node 101 plus two. That is, let p1 and p2 be the pressure of first and second nodes of a nodal pair, and let c be the constant offset. Then the periodic condition is

$$p_2 = p_1 + c \tag{131}$$

For vector variables, *constant_values* is used to define the offset for each of the three components. For example,

```
PERIODIC_BOUNDARY_CONDITION( "accelerated velocity" ) {
   variable         = velocity
   type             = constant_offset
   nodal_pairs      = { 1, 15, 16 ; 2, 25, 26 ; }
   constant_values  = { 1, 0, 0 }
}
```

The nodal_pair_offset periodic boundary condition specifies a different offset for each nodal pair. For example,

```
PERIODIC_BOUNDARY_CONDITION( "interface between two frames of reference" ) {
   variable           = velocity
   type               = constant_pair_offset
   nodal_pairs        = { 1, 1, 11 ; ... }
   nodal_pairs_values = { 1, 0, 0, 1 ; ... }
}
```

This example implements the two_reference_frames periodic boundary condition example given above. The *nodal_pair_values* parameter is a multi-column array, corresponding to the nodal-pair number and the offset values. For a scalar variable this array has two columns and for a vector variable it has four columns.

The nodal-pair values may be read from a file. For the above example, the values may be placed in a file, such as `"two-frame.pbc.val`:

```
1 0 0 1
...
```

and read by:

```
PERIODIC_BOUNDARY_CONDITION( "interface between two frames of reference" ) {
   variable           = velocity
   type               = constant_pair_offset
   nodal_pairs        = { 1, 1, 11 ; ... }
   nodal_pairs_values = Read( "two-frame.pbc.val" )
}
```

A periodic boundary condition of user_function_offset type may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider the above two reference frame periodic boundary condition. Then the input command may be given by:

```
PERIODIC_BOUNDARY_CONDITION( "UDF BC for two frames of reference" ) {
   variable       = velocity
   type           = user_function_offset
   nodal_pairs    = { 1, 1, 11 ; ... }
   user_function  = "usrPbcOffsetExample"
   user_values    = { 0, 0, 0,  # rotating center
                      1 0, 1, 0,  # angular velocity
                      1 0, 0, 0,  # rotating center
                      2 0, 0, 0 } # angular velocity 2
}
```

where the user-defined function `"usrPbcOffsetExample"` may be implemented as follows.

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrPbcOffsetExample ) ;                  /* function prototype */
Void usrPbcOffsetExample (
    UdfHd       udfHd,                                  /* Opaque handle for accessing
 data   */
 Real*       outVec,                                 /* Output vector  */
 Integer     nItems,                                 /* Number of PBC pairs  */
 Integer     vecDim                                  /* = 3  */
) {
    Integer     pair ;                                  /* a nodal pair counter   */
    Real        xOrg1 ;                                 /* x-center 1  */
    Real        yOrg1 ;                                 /* y-center 1  */
    Real        zOrg1 ;                                 /* z-center 1  */
    Real        xOmega1 ;                               /* x-angular velocity 1  */
    Real        yOmega1 ;                               /* y-angular velocity 1  */
    Real        zOmega1 ;                               /* z-angular velocity 1  */
    Real        xOrg2 ;                                 /* x-center 2  */
    Real        yOrg2 ;                                 /* y-center 2   */
    Real        zOrg2 ;                                 /* z-center 2  */
    Real        xOmega2 ;                               /* x-angular velocity 2  */
    Real        yOmega2 ;                               /* y-angular velocity 2  */
    Real        zOmega2 ;                               /* z-angular velocity 2  */
    Real        xOff ;                                  /* reduced x-offset  */
    Real        yOff ;                                  /* reduced y-offset  */
    Real        zOff ;                                  /* reduced z-offset  */
    Real        xOmega ;                                /* reduced x-angular velocity
 */
    Real        yOmega ;                                /* reduced y-angular velocity
 */
```

```
    Real        zOmega ;                                /* reduced z-angular velocity
 */
    Real *      crd ;                                   /* coordinates  */
    Real *      usrVals ;                               /* user values  */
    Real *      xCrd ;                                  /* x-coordinates of node 1 */
    Real *      yCrd ;                                  /* y-coordinates of node 1  */
    Real *      zCrd ;                                  /* z-coordinates of node 1  */
    Real *      xOffset ;                               /* x-component of offset  */
    Real *      yOffset ;                               /* y-component of offset  */
    Real *      zOffset ;                               /* z-component of offset  */
udfCheckNumUsrVals( udfHd, 12 ) ;                       /* check for error   */
     usrVals    = udfGetUsrVals( udfHd ) ;              /* get the user vals  */
     xOrg1      = usrVals[ 0] ;                         /* get x-center 1  */
     yOrg1      = usrVals[ 1] ;                         /* get y-center 1  */
     zOrg1      = usrVals[ 2] ;                         /* get z-center 1  */
     xOmega1    = usrVals[ 3] ;                         /* get x-ang vel 1  */
     yOmega1    = usrVals[ 4] ;                         /* get y-ang vel 1  */
     zOmega1    = usrVals[ 5] ;                         /* get z-ang vel 1  */
     xOrg2      = usrVals[ 6] ;                         /* get x-center 2  */
     yOrg2      = usrVals[ 7] ;                         /* get y-center 2  */
     zOrg2      = usrVals[ 8] ;                         /* get z-center 2  */
     xOmega2    = usrVals[ 9] ;                         /* get x-ang vel 2  */
     yOmega2    = usrVals[ 10] ;                        /* get y-ang vel 2  */
     zOmega2    = usrVals[ 11] ;                        /* get z-ang vel 2  */
     xOmega     = xOmega2 - xOmega1 ;                   /* get reduced omega  */
     yOmega     = yOmega2 - yOmega1 ;                   /* get reduced omega  */
     zOmega     = zOmega2 - zOmega1 ;                   /* get reduced omega  */
     xOff       = yOmega2 * zOrg2 - zOmega2 * yOrg2
                - yOmega1 * zOrg1 + zOmega1 * yOrg1;    /* get reduced offset  */
     yoff       = zOmega2 * xOrg2 - xOmega2 * zOrg2
                -  zOmega1 * xOrg1 + xOmega1 * zOrg1;   /* get reduced offset  */
     zoff       = xOmega2 * yOrg2 - yOmega2 * xOrg2
                - xOmega1 * yOrg1 + yOmega1 * xOrg1     /* get reduced offset  */
    crd         = udfGetPbcCrd( udfHd ) ;               /* get the coord.  */
    xCrd        = &crd[0*nItems] ;                      /* localize x-coord.  */
    yCrd        = &crd[1*nItems] ;                      /* localize y-coord.  */
    zCrd        = &crd[2*nItems] ;                      /* localize z-coord.  */
    xOffset     = &outVec[0*nItems] ;                   /* localize x-offset  */
    yOffset     = &outVec[1*nItems] ;                   /* localize y-offset   */
    zOffset     = &outVec[2*nItems] ;                   /* localize z-offset  */
    for ( pair = 0 ; pair < nItems ; pair++ ) {
        xOffset[pair] = yOmega * zCrd[pair] - zOmega * yCrd[pair] - xOff ;
        yOffset[pair] = zOmega * xCrd[pair] - xOmega * zCrd[pair] - yOff ;
        zOffset[pair] = xOmega * yCrd[pair] - yOmega * xCrd[pair] - zOff ;
    }
} /* end of usrPbcOffsetExample() */
```

For a vector variable (as above) the dimension of the returned boundary condition vector, *outVec*, is the number of nodal pairs times three. For a scalar variable, it is the number of nodal pairs.

A constant_coefficients periodic boundary condition specifies a general two-node constraint on a variable. For example,

```
PERIODIC_BOUNDARY_CONDITION( "inflow/outflow scaled temperature profile" ) {
   variable        = temperature
   type            = constant_coefficients
   nodal_pairs     = { 1, 101, 201 ;
                       2, 102, 202 ;
                       3, 103, 203 ; }
   constant_values = { 2, -1, 0 }
}
```

The temperature of node 201 is twice that of node 101. That is, let T1 and T2 be the temperature of first and second nodes of a nodal pair, and let {a_1, a_2, c} be the coefficients given by *constant_values*, which is a three-component array in this case. Then the periodic condition is:

$$a_1 T_1 + a_2 T_2 = c \tag{132}$$

For vector variables the *constant_values* parameter has 21 components, corresponding to three constraints and seven coefficients per constraint. The coefficient are for the x, y, and z components of nodes one and two plus the constraint constant. Let u_1, v_1, w_1, u_2, v_2, and w_2 be the velocity components of the first and second nodes of a pair, and let $a_1^1, \ldots, a_6^1, c^1, a_1^2, \ldots a_6^2, c^2, a_1^3, \ldots a_6^3, c^3$ be the 21 coefficients given by *constant_values*. Then the periodic conditions are:

$$\begin{aligned} a_1^1 u_1 + a_2^1 v_1 + a_3^1 w_1 + a_4^1 u_2 + a_5^1 v_2 + a_6^1 w_2 &= c^1 \\ a_1^2 u_1 + a_2^2 v_1 + a_3^2 w_1 + a_4^2 u_2 + a_5^2 v_2 + a_6^2 w_2 &= c^2 \\ a_1^3 u_1 + a_2^3 v_1 + a_3^3 w_1 + a_4^3 u_2 + a_5^3 v_2 + a_6^3 w_2 &= c^3 \end{aligned} \tag{133}$$

The nodal_pair_coefficients periodic boundary condition specifies a different set of coefficients for each pair. For example, let node one and two be at coordinates (1,1,0) and (-1,1,0), then the above axisymmetric periodic boundary condition may be specified by:

```
PERIODIC_BOUNDARY_CONDITION( "a pie cut axisymmetric constraint" ) {
   variable          = velocity
   type              = nodal_pair_coefficients
   nodal_pairs       = { 1, 1, 2 ;
                         ... }
   nodal_pair_values = { 1,  0, 0,  1,  0,  0, -1,   0,  # z-axis
                         1,  1, 0,  1, -1,  0,  0,       # radial
                         1, -1, 0, -1, -1,  0,  0;       # tangential... }
}
```

The *nodal_pair_values* parameter is a multi-column array, corresponding to the nodal-pair number and the coefficient values. For a scalar variable this array has four columns and for a vector variable it has 22 columns. The nodal-pair values may be read from a file, as given in an example above.

A periodic boundary condition of user_function_coefficients type may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider a thermal periodic condition where the inflow temperature scaled by the inflow bulk temperature is equal to the outflow temperature scaled by the outflow bulk temperature. Let "inflow surface" and "outflow surface" be the names of SURFACE_OUTPUT sets for inflow and outflow surfaces. Then the input command may be given by:

```
PERIODIC_BOUNDARY_CONDITION( "UDF BC with fixed wall temperature" ) {
   variable       = temperature
   type           = user_function_coefficients
   nodal_pairs    = Read( "inflow-outflow.pbc" )
   user_function  = "usrPbcCoefExample"
   user_values    = { 0,    # fixed wall temperature
                      10,   # initial inflow bulk temperature
```

```
                      9,    # initial outflow bulk temperature
                   1005 }   # specific heat
   user_strings   = { "inflow surface", "outflow surface" }
}
```

where the user-defined function "usrPbcCoefExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrPbcOffsetExample ) ; /* function prototype*/
Void usrPbcOffsetExample (
    UdfHd            udfHd,                                 /* Opaque handle for
 accessing data */
    Real*            outVec,                                /* Output vector  */
    Integer          nItems,                                /* Number of PBC pairs  */
    Integer          vecDim                                 /* = 3  */
) {
    Integer          pair ;                                 /* a running index  */
    Real             wallTemp ;                             /* wall temperature  */
    Real             inBulkTemp ;                           /* inflow bulk temperature
  */
    Real             outBulkTemp ;                          /* outflow bulk
 temperature  */
    Real             cp ;                                   /* specific heat  */
    Real             inCoef ;                               /* inflow temp coefficient
  */
    Real             outCoef ;                              /* outflow temp
 coefficient  */
    Real             cCoef ;                                /* constant coefficient
 */
    Real *           convFlux ;                             /* convective temperature
 flux  */
    Real *           massFlux ;                             /* mass flux  */
    Real *           usrVals ;                              /* user values  */
    Real *           inCoefs ;                              /* return inflow temp
 coefficient  */
    Real *           outCoefs ;                             /* return outflow temp
 coefficient  */
    Real *           cCoefs ;                               /* return constant
 coefficient  */
    String*          usrStrs ;                              /* user strings  */
    String           inflowOsi ;                            /* inflow surface output
 */
    String           outflowOsi ;                           /* outflow surface output
 */
udfCheckNumUsrVals( udfHd, 4 ) ;                            /* check for error  */
udfCheckNumUsrVals( udfHd, 2 ) ;                            /* check for error  */
    usrVals          = udfGetUsrVals( udfHd ) ;             /* get the user vals  */
    usrStrs          = udfGetUsrStrs( udfHd ) ;             /* get the user strs  */
    wallTemp         = usrVals[0] ;                         /* get wall temp.  */
    inBulkTemp       = usrVals[1] ;                         /* get inflow bulk T  */
    outBulkTemp      = usrVals[2] ;                         /* get output bulk T */
    cp               = usrVals[3] ;                         /* get specific heat  */
    inflowOsi        = usrStrs[0] ;                         /* get inflow surf.  */
    outflowOsi       = usrStrs[1] ;                         /* get outflow surf.  */
    if ( !udfFirstStep(udfHd) ) {                           /* initial step? */
        massFlux     = udfGetOsiData( udfHd, inflowOsi,    */
                       UDF_OSI_MASS_FLUX ) ;                 /* inflow mass flux */
        convFlux     = udfGetOsiData( udfHd, inflowOsi,
                       UDF_OSI_CONVECTIVE_TEMPERATURE) ;    /* inflow conv.temp  */
        inBulkTemp   = *convFlux / (*massFlux * cp) ;       /* inflow bulk temp  */
        massFlux     = udfGetOsiData( udfHd, outflowOsi,
```

```
                          UDF_OSI_MASS_FLUX ) ;                /* outflow mass flux  */
        convFlux      = udfGetOsiData( udfHd, outflowOsi,
                          UDF_OSI_CONVECTIVE_TEMPERATURE) ;    /* outflow conv.temp  */
        outBulkTemp   = *convFlux / (*massFlux * cp) ;         /* outflow bulk temp  */
    }
    if ( inBulkTemp  == wallTemp ) {
        udfSetError( udfHd, "inflow bulk temp. equals wall temp." ) ;
    }
    if ( outBulkTemp == wallTemp ) {
        udfSetError( udfHd, "outflow bulk temp. equals wall temp." ) ;
    }
    inCoef            = 1. / (inBulkTemp - wallTemp) ;
    outCoef           = -1. / (outBulkTemp - wallTemp) ;
    cCoef             = (inCoef + outCoef) * wallTemp ;
    inCoefs           = &outVec[0*nItems] ;                /* localize outVec  */
    outCoefs          = &outVec[1*nItems] ;                /* localize outVec  */
    cCoefs            = &outVec[2*nItems] ;                /* localize outVec  */
    for ( pair = 0 ; pair < nItems ; pair++ ) {
        inCoefs [pair]  = inCoef ;
        outCoefs[pair]  = outCoef ;
        cCoefs [pair]   = cCoef ;
    }
} /* end of usrPbcCoefExample() */
```

The dimension of the returned boundary condition vector, *outVec*, is the number of nodal pairs times three for this scalar variable. For a vector variable, it is the number of nodal pairs times 21.

The *multiplier_function* parameter may be used to uniformly scale the offset values for the offset- and coefficient-type periodic boundary conditions. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file. For example, to linearly increase the pressure drop between the inflow and outflow nodes, you may specify:

```
PERIODIC_BOUNDARY_CONDITION( "inflow/outflow pressure drop" ) {
   variable             = pressure
   type                 = constant_offset
   nodal_pairs          = { 1, 101, 201 ;
                            2, 102, 202 ;
                            3, 103, 203 ; }
   constant_value       = 2
   multiplier_function  = "ramped"
}
MULTIPLIER_FUNCTION( "ramped" ) {
   type                 = piecewise_linear
   curve_fit_values     = { 0, 0 ; 10, 1 }
   curve_fit_variable   = time
}
```

In some circumstances it is necessary to "turn off" or "deactivate" a previously-defined periodic boundary condition. This is accomplished by setting the active flag through *active_type*. A value of all is the default and means that all of the nodal pairs in the set are active (active flag = 1) and will have the periodic boundary condition imposed on them. A value of none means that no pair is active (active flag = 0). For example, assume that one run was made with the following command to model a periodic channel flow:

```
PERIODIC_BOUNDARY_CONDITION( "inflow/outflow periodicity" ) {
   variable    = all
   type        = periodic
   nodal_pairs = Read( "inflow-outflow.pbc" )
```

```
}
```

Then the entire input file to convert this model on restart to a channel with plug flow at the inlet may be given by (assuming the normal to the inlet surface is in the x direction):

```
RESTART {}
PERIODIC_BOUNDARY_CONDITION( "inflow/outflow periodicity" )
   active_type     = none
}
NODAL_BOUNDARY_CONDITION( "inflow x-velocity" ) {
   variable        = x_velocity
   nodes           = Read( "inflow.nbc" )
   type            = constant
   constant_value  = 1
}
NODAL_BOUNDARY_CONDITION( "inflow y-velocity" ) {
   variable        = y_velocity
   nodes           = Read( "inflow.nbc" )
   type            = zero
}
NODAL_BOUNDARY_CONDITION( "inflow z-velocity" ) {
   variable        = z_velocity
   nodes           = Read( "inflow.nbc" )
   type            = zero
   }
RUN {}
```

An active flag of user_function type may be used to model more complex behaviors; see the *AcuSolve User- Defined Functions Manual* for a detailed description of user-defined functions. Values of the active flag between zero and one are allowed; such values are used to linearly interpolate between the previous value of the periodic condition and the given condition, with the result imposed as the actual periodic condition.

For example, consider a model of a window suddenly opening. Two surfaces with coincident nodes define the window. When it is closed, all the nodes have no-slip velocity boundary conditions and no condition on the pressure, which allows the window to support a pressure difference. When the window opens, the no-slip conditions are removed by turning the active flag off in the `NODAL_BOUNDARY_CONDITION` commands and the pairs of coincident nodes are made equivalent by applying a periodic boundary condition on all variables, turning on the active flag. The input commands for implementing this model may be given by:

```
NODAL_BOUNDARY_CONDITION( "UDF opening window for x-velocity" ) {
   variable             = x_velocity
   type                 = zero
   nodes                = Read( "window.nbc" )
   active_type          = user_defined
   active_user_function = "usrActiveFlagNBC"
   active_user_values   = { 100,    # deactivation time
                            0.1 } # mushy interval for time
}
PERIODIC_BOUNDARY_CONDITION( "UDF opening window" ) {
   variable             = all
   type                 = periodic
   nodal_pairs          = Read( "window.pbc" )
   active_type          = user_defined
   active_user_function = "usrActiveFlagNBC"
   active_user_values   = { 100,    # deactivation time
                            0.1 } # mushy interval for time
```

```
}
```

with similar commands for the y- and z-velocities. The user-defined function "usrActiveFlagPBC" may be implemented as follows.

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrActiveFlagPBC ) ;          /* function prototype */
Void usrActiveFlagPBC (
    UdfHd        udfHd,                      /* Opaque handle for accessing data  */
    Real*        outVec,                     /* Output vector  */
    Integer      nItems,                     /* Number of PBC pairs  */
    Integer      vecDim                      /* = 1  */
) {
    Integer      pair ;                      /* nodal pair counter  */
    Real         t0 ;                        /* activation time   */
    Real         mushy ;                     /* mushy interval in time  */
    Real         delta_t ;                   /* time relative to activation time  */
    Real         time ;                      /* time  */
    Real*        usrVals ;                   /* user values  */
udfCheckNumUsrVals( udfHd, 2 ) ;             /* check for error       */
    time         = udfGetTime( udfHd ) ;     /* get the time  */
    usrVals      = udfGetUsrVals( udfHd ) ;  /* get the user vals  */
    t0           = usrVals[0] ;              /* get activation time  */
    mushy        = usrVals[1] ;              /* get mushy interval  */
    for ( pair = 0 ; pair < nItems ; pair++ ) {
        delta_t = time - t0 ;
        if ( delta_t >= mushy ) {
            outVec[pair] = 1 ;
        } else if ( delta_t <= 0 ) {
            outVec[pair] = 0 ;
        } else {
            outVec[pair] = delta_t / mushy ;
        }
    }
} /* end of usrActiveFlagPBC() */
```

The dimension of the returned active flag vector, *outVec*, is the number of nodal pairs. The implementation for the user-defined function "usrActiveFlagNBC" is similar except the activation flag changes from one to zero at the activation time. See NODAL_BOUNDARY_CONDITION for an example.

RESTART_OUTPUT

Specifies parameters for outputting restart files.

Type

AcuSolve Command

Syntax

RESTART_OUTPUT {parameters}

Qualifier

This command has no qualifier.

Parameters

output_frequency or *out_freq* *(integer) >=0 [=1000]*
: Time step frequency at which to output restart data. If zero, this option is ignored.

output_time_interval or *out_intv* *(real) >=0 [=0]*
: Time frequency at which to output restart data. If zero, this option is ignored.

num_saved_states or *save* *(integer) >=0 [=0]*
: Number of restart states to keep. If zero, all restart files are kept.

case_output_frequency or *cycle_output_frequency* *>=0 [=1000]*
: Frequency at which to output optimization case solutions when optimization under EQUATION is set to on. If zero, this option is ignored.

Description

This command specifies the parameters for writing restart files to disk. For example,

```
RESTART_OUTPUT {
   output_frequency     = 100
   output_time_interval = 0
   num_saved_states     = 2
}
```

writes a restart file to disk every 100 time steps as well as the last time step. Only the last two restart steps are kept. Once the third restart file is written to disk, the first restart file is removed to preserve disk space.

The restart files are tagged by their problem name, run ID, and time step. The RESTART command uses these tags to access the files.

If either *output_frequency* or *output_time_interval* is non-zero, a restart file will be written at the end of the run. If both are zero, no restart files are created.

Run times may not coincide with *output_time_interval*. In this case, at every time step which passes through a multiple of *output_time_interval*, restart data are output.

The *num_saved_states* parameter indicates the number of restart files to save. Once the (*num_saved_states* + 1)th restart file is written, the first one is removed. When running in parallel, once all files are written (in parallel), then the old ones are removed (in parallel).

RUNNING_AVERAGE_OUTPUT

Specifies nodal output parameters for running average fields.

Type

AcuSolve Command

Syntax

RUNNING_AVERAGE_OUTPUT {parameters}

Qualifier

This command has no qualifier.

Parameters

output_frequency or *out_freq* *(real) >=0 [=0]*
: Time step frequency at which to output nodal running average fields. If zero, this option is ignored.

output_time_interval or *out_intv* *(real) >=0 [=0]*
: Time frequency at which to output nodal running average fields. If zero, this option is ignored.

num_saved_states or *save* *(integer) >=0 [=0]*
: Number of nodal running average field files to keep. If zero, all are kept.

Description

This command specifies the frequency at which nodal running average fields are written into a results database on disk. Running average fields are output only if they are defined by the EQUATION command. For example,

```
RUNNING_AVERAGE_OUTPUT {
   output_frequency      = 10
   output_time_interval  = 2.5
}
```

specifies that the nodal running average fields are to be output at every 10 time steps, at every 2.5 units of run time and at the last time step.

Once the nodal running average fields have been written to disk, they can be translated to other formats using the AcuTrans program and other post-processing modules; see the *AcuSolve Programs Reference Manual* for details.

If either *output_frequency* or *output_time_interval* is non-zero, the nodal running average fields will be output at the end of the run. If both are zero, no data are written to disk.

Run times may not coincide with *output_time_interval*. In this case, at every time step which passes through a multiple of *output_time_interval*, the nodal running average fields are output.

The *num_saved_states* parameter indicates the number of nodal running average field files to save. Once the (*num_saved_states* + 1)th file is written, the first one is removed. When running in parallel, once all files are written (in parallel), then the old ones are removed (in parallel).

TIME_AVERAGE_OUTPUT

Specifies parameters to output time-averaged nodal fields.

Type

AcuSolve Command

Syntax

TIME_AVERAGE_OUTPUT {parameters}

Qualifier

This command has no qualifier.

Parameters

order (integer) >=1 <=2 [=2]
: Order of output fields. If one, first-order data are output. If two, first and second-order data are output. Currently only two is supported.

output_frequency or *out_freq (integer) >=0 [=0]*
: Time step frequency at which to output nodal solutions. If zero, this option is ignored.

output_time_interval or *out_intv (real) >=0 [=0]*
: Time frequency at which to output nodal solutions. If zero, this option is ignored.

reset_frequency or *reset_freq (integer) >=0 [=0]*
: Time step frequency at which to reset the averaging process. If zero, never reset.

num_saved_states or *save (integer) >=0 (=0)*
: Number of nodal solutions to keep. If zero, all nodal solution files are kept.

Description

This command specifies the frequency at which to output time-averaged nodal values into a results database. For example,

```
TIME_AVERAGE_OUTPUT {
   order                = 2
   output_frequency     = 10
   output_time_interval = 0
   reset_frequency      = 10
}
```

specifies that the time-averaged nodal values up to the second-order terms are to be output at every 10 time steps and at the last time step. The sums for averaging the data will be reset every 10 time steps.

Once the solution has been written to disk, it can then be translated to other formats using the AcuTrans program and other post-processing modules; see the *AcuSolve Programs Reference Manual* for details.

When order is set to one, then the averaged quantities of velocity, pressure, and stress are output. The velocity averaging is performed on the conservative set of solutions. When order is two, the averaged quantities are these plus the square of velocity and the square of pressure. Only *order*=2 is supported currently.

If either *output_frequency* or *output_time_interval* is non-zero, the solution will be output at the end of the run. If both are zero, no time averaging is performed and no data are output.

Run times may not coincide with *output_time_interval*. In this case, at every time step which passes through a multiple of *output_time_interval*, the solution is output.

The *num_saved_states* parameter indicates the number of nodal solution files to save. Once the (*num_saved_states* + 1)th file is written, the first one is removed. When running in parallel, once all files are written (in parallel), then the old ones are removed (in parallel).

TIME_HISTORY_OUTPUT

Specify nodal solution output for a set of prescribed points.

Type

AcuSolve Command

Syntax

TIME_HISTORY_OUTPUT("name") {parameters...}

Qualifier

User-given name.

Parameters

type (enumerated) [=nodal]

How the spatial location of the output points is specified.

nodal	Nodal points. Requires nodes.
coordinates	Coordinate points. Requires coordinates.

nodes (array) [no default]

List of nodes to be output. Used with nodal type.

coordinates or *coord (array) [no default]*

Four-column array consisting of a point identifier (integer) plus the xyz coordinate points to be output. Used with coordinates type.

output_frequency or *out_freq (integer) >=0 [=1]*

Time step frequency at which to output nodal solutions. If zero, this option is ignored.

output_time_interval or *out_intv (integer) >=0 [=0]*

Time frequency at which to output nodal solutions. If zero, this option is ignored.

Description

This command specifies a list of points whose nodal solution fields are to be written into a results database. For example,

```
TIME_HISTORY_OUTPUT( "sample nodes" ) {
   type                = nodal
   nodes               = { 1 ;13 ;26 ; }
   output_frequency    = 10
   output_time_interval = 2.5
}
```

specifies that the nodal solution values of nodes 1, 13, and 26 are to be output every 10 time steps, every 2.5 units of time, and at the last time step.

The *nodes* parameter is a single-column array containing a list of node numbers. Node numbers must be valid numbers, as given by the COORDINATE command. The nodal solution of these nodes is output at the frequency specified.

The nodal numbers may be read from a file. For the above example, the nodes may be placed in a file, such as sample.nod:

```
1
13
26
```

and read by:

```
TIME_HISTORY_OUTPUT( "sample nodes" ) {
   type                 = nodal
   nodes                = Read( "sample.nod" )
   output_frequency     = 10
   output_time_interval = 2.5
}
```

Instead of outputting the nodal solution at particular nodes, the coordinates type may be used to interpolate the nodal solutions to the coordinate points given by coordinates. If there is mesh movement the reference coordinates are used, as opposed to the current coordinates. For example,

```
TIME_HISTORY_OUTPUT( "sample coordinates" ) {
   type                 = coordinates
   coordinates          = { 1, 1.1, 0.5, 7.8 ;
                            2, 1.5, 2.3, 4.6 ;
                            3, -3.2, 1.8, 5.4 ; }
   output_frequency     = 10
   output_time_interval = 2.5
}
```

The coordinates parameter is a four-column array containing an identifying integer plus the xyz coordinates. This identifier is unrelated to the node numbers. For the above example, the coordinates may be placed in a file, such as sample.crd:

```
1 1.1 0.5 7.8
2 1.5 2.3 4.6
3 -3.2 1.8 5.4
```

and ready by:

```
TIME_HISTORY_OUTPUT( "sample coordinates" ) {
   type                 = coordinates
   coordinates          = Read( "sample.crd" )
   output_frequency     = 10
   output_time_interval = 2.5
}
```

Any point in coordinates that falls outside of the mesh is projected to the nearest point in the mesh.

Once the output to disk is complete, it can then be translated to other formats using the AcuTrans program and other post-processing modules; see the *AcuSolve Programs Reference Manual* for details.

If either *output_frequency* or *output_time_interval* is non-zero, the solution points will be output at the end of the run. If both are zero, no data are output.

Run times may not coincide with *output_time_interval*. In this case, the solution points are output every time step which passes through a multiple of *output_time_interval*.

The TIME_HISTORY_OUTPUT command is designed for outputting a select number of points at higher frequency levels than is practical with the NODAL_OUTPUT command. However, if the number of points in TIME_HISTORY_OUTPUT becomes large, the use of TIME_HISTORY_OUTPUT can become less efficient than the use of the NODAL_OUTPUT command.

Element Data Commands

13

The Element Data commands are outlined in this chapter.

This chapter covers the following:

- BATTERY_COMPONENT_MODEL (p. 456)
- BATTERY_MODULE (p. 457)
- BATTERY_PACK (p. 470)
- BATTERY_THERMAL_RUNAWAY_MODEL (p. 473)
- CAA_ELEMENT_OUTPUT (p. 477)
- CAA_OUTPUT (p. 479)
- CAA_SURFACE_OUTPUT (p. 488)
- ELEMENT_BOUNDARY_CONDITION (p. 490)
- ELEMENT_OUTPUT (p. 505)
- ELEMENT_SET (p. 509)
- EXTERNAL_CODE_SURFACE (p. 526)
- FAN_COMPONENT (p. 532)
- FEEDBACK_CONDITION (p. 544)
- FIELD_BOUNDARY_CONDITION (p. 547)
- FREE_SURFACE (p. 550)
- FWH_SURFACE_OUTPUT (p. 554)
- FWH_OUTPUT (p. 556)
- GUIDE_SURFACE (p. 557)
- HEAT_EXCHANGER_COMPONENT (p. 562)
- INTERFACE_SURFACE (p. 576)
- INTERPOLATED_MOTION_SURFACE (p. 581)
- LINE_SOURCE (p. 585)
- MESH_BOUNDARY_CONDITION (p. 590)
- PARTICLE_SURFACE (p. 600)
- RADIATION_SURFACE (p. 604)
- SIMPLE_BOUNDARY_CONDITION (p. 611)
- SOLAR_RADIATION_SURFACE (p. 663)
- SURFACE_INTEGRATED_CONDITION (p. 669)
- SURFACE_OUTPUT (p. 674)
- SURFACE_SET (p. 679)
- TURBULENCE_WALL (p. 682)
- VAO_ELEMENT_OUTPUT (p. 689)

- VAO_OUTPUT (p. 691)
- VAO_SURFACE_OUTPUT (p. 692)
- VOLUME_SET (p. 694)

A problem must contain at least one `ELEMENT_SET` command. All other commands in this chapter are optional.

BATTERY_COMPONENT_MODEL

Specifies a battery component model and its parameters.

Type

AcuSolve Command

Syntax

```
BATTERY_COMPONENT("name") {parameters...}
```

Parameters

component_type (enumerated) [=none]

Type of component in the battery module.

none	No label given to an electrical component. This represents all electric components that are not cells, tab/terminals, or busbars.
battery_cell	A battery cell representing the homogenized region of the battery, for example, electrode pairs and separator.
tab_positive	Positive terminal/tab of a battery cell.
tab_negative	Negative terminal/tab of a battery cell.
busbar	Busbar component in the module.

cell_id (int) [=1]

Cell ID of battery cell in a module.

initial_state_of_charge (real) [=1.0]

The initial state of charge in a cell in a battery module. This parameter can only be variable across different cells when the module current is provided as an input.

battery_module (string) [=no default]

User-given name of the BATTERY_MODULE to which this component belongs.

BATTERY_MODULE

Specifies a battery module and its parameters.

Type

AcuSolve Command

Syntax

```
BATTERY_MODULE("name") {parameters...}
```

Parameters

number_of_parallel_cells *(int) ≥1 [=1]*

Number of cells connected in parallel in a battery module for each serial unit. For example, for a 4s2p battery pack the number of parallel cells is 2.

number_of_serial_cells *(int) ≥1 [=1]*

Number of parallel units connected together in series. For example, for a 4s2p battery pack the number of serial cells is 4.

number_of_parallel_connected_units *(int) ≥1 [=1]*

Number of parallel connected units in the module configuration. For example, in a 4s2p battery module 2 parallel connected units means two sub-units of size 2s2p are connected.

electrical_input_type *(enumerated) [=current]*

Type of global electrical input to battery module.

current	Electric current input. Requires *current_type*.
c_rate	C-rate input. Requires *c_rate_type*.
voltage	Voltage input. Requires *voltage_type* = constant.
power	Power input. Requires *power_type*.
standard_charge_profile	Pre-defined charging profile.

current_type *(enumerated) [=none]*

Type of electrical current input.

constant or const	Constant electrical current. Requires *current*.
piecewise_linear or linear	Piecewise linear curve fit electrical current profile. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit electrical current profile. Requires *curve_fit_values* and *curve_fit_variable*.

current *(real) [=0]*

Constant value of electrical current. When *current_type* is constant. Notice you have positive current for discharging and negative current for charging.

current_curve_fit_values(array) [={0,0}]
A two-column array of independent-variable/isotropic electrical current data values. The independent variable is either state of charge or time. Used with *piecewise_linear* and *cubic_spline* types.

current_curve_fit_variable(enumerated) [=time]
Independent variable of the curve fit for isotropic electrical current. Used with *piecewise_linear* and *cubic_spline* types.

time	time

c_rate_type (enumerated) [=none]
Type of c-rate input.

none	No c-rate type specified.
constant or const	Constant c-rate value. Requires *c_rate*.
piecewise_linear or linear	Piecewise linear curve fit c-rate profile. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit c-rate profile. Requires *curve_fit_values* and *curve_fit_variable*.

c_rate (real) [=0]
Constant value of c-rate. When type is constant.

c_rate_curve_fit_values(array) [={0,0}]
A two-column array of independent-variable/isotropic c-rate data values. The independent variable is time. Used with piecewise_linear and cubic_spline types.

c_rate_curve_fit_variable (enumerated) [=time]
Independent variable of the curve fit for isotropic c-rate. Used with piecewise_linear and cubic_spline types.

time	time

voltage_type (enumerated) [=constant]
Type of voltage input.

constant or const	Constant voltage value. Requires *voltage*.

voltage (real) [=0]
Constant value of module input voltage. When *voltage_type* = constant.

power_type (enumerated) [=constant]
Type of power input.

constant or const	Constant power value. Requires *power*.
piecewise_linear or linear	Piecewise linear curve fit power profile. Requires *curve_fit_values* and *curve_fit_variable*.

cubic_spline or spline Cubic spline curve fit power profile. Requires *curve_fit_values* and *curve_fit_variable*.

power (real) [=0]
Constant value of module input power. When *power_type* = constant. Input can be positive or negative.

power_curve_fit_values (array) [={0,0}]
A two-column array of independent-variable/isotropic power data values. The independent variable is time. Used with *piecewise_linear* and *cubic_spline* types.

power_curve_fit_variable (enumerated) [=time]
Independent variable of the curve fit for power. Used with *piecewise_linear* and *cubic_spline* types.

time time

charging_profile_type (enumerated) [=none]
Type of power input.

none No charging profile used.

constant_current_constant_voltage or cccv Constant current constant voltage charging profile. Requires a module input current or *c_rate* to be specified.

constant_power_constant_voltage or cpcv Constant power constant voltage standard charging profile. Requires module power to be specified.

cut_off_current_percent (real) [=0.01]
Current at which the battery is considered fully charged when using charge profiles.

maximum_cell_voltage (real) [=4.20]
Maximum allowed voltage of an individual battery cell. This avoids overcharging.

minimum_cell_voltage (real) [=3.0]
Minimum allowed voltage of an individual battery cell.

maximum_cell_state_of_charge (real) [=0.95]
State of charge at which a battery cell is considered fully charged.

minimum_cell_state_of_charge (real) [=0.05]
State of charge at which a battery cell is considered fully discharged.

module_id (int) ≥ 0 [=1]
Module ID of battery module in a pack.

battery_pack (string) [=no default]
User-given name of the BATTERY_PACK to which this BATTERY_MODULE belongs.

Description

Battery Modules and Components

The BATTERY_MODULE command specifies parameters related to a battery module (multiple battery cells connected in series and parallel).

BATTERY_MODULE commands are referenced by BATTERY_COMPONENT_MODEL commands, which in turn are referenced by an ELEMENT_SET command.

```
BATTERY_MODULE( "4s2p battery module" ) {
        number_of_parallel_cells = 2
        number_of_serial_cells = 4
        electrical_input_type = current
        current_type = constant
        current = 20
        number_of_parallel_connected_units = 2
}
BATTERY_COMPONENT_MODEL( "battery cell" ) {
   battery_module = "4s2p battery module"
   ...
}

BATTERY_COMPONENT_MODEL( "battery positive terminal" ) {
   battery_module = "4s2p battery module"
   ...
}

ELEMENT_SET( "battery cell element set" ) {
   Battery_component_model = "battery cell"
   ...
}
ELEMENT_SET( "terminal element set" ) {
   Battery_component_model = "battery positive terminal"
   ...
}
```

Cell Electrical Connectivity

The number of parallel and serial cells, *number_of_parallel_cells* and *number_of_serial_cells* respectively, are defined by their electrical connectivity, for example a pouch module made up of 8 cells in a 4s2p configuration would be:

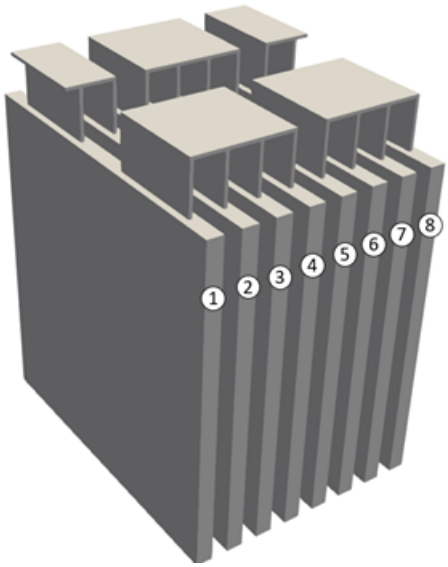


Figure 10:

Where four pairs of parallel connected cells: 1&2, 3&4, 5&6, and 7&8, are connected in series to make up the module configuration 4s2p. Alternatively, for a cylindrical module, with 48 cells in an 8s6p configuration:

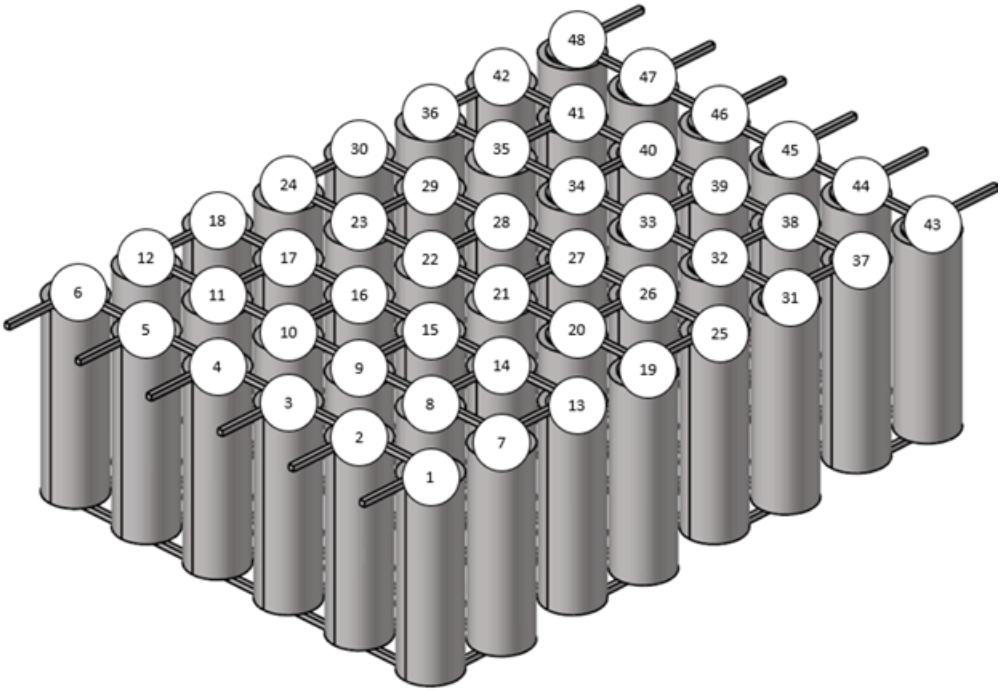


Figure 11:

Each set of 6 cells, for example, 1-6 or 7-12, are connected in parallel. These banks of parallel cells are then connected in series, for example, cells 1-6 are connected in series to cells 7-12. The *number_of_parallel_connected_units* is how many PCUs (parallel connected unit) you have in the configuration. Conceptually a PCU and an SCU, given by PCU being equal to the number of serial connected cells, is demonstrated in the following figure:

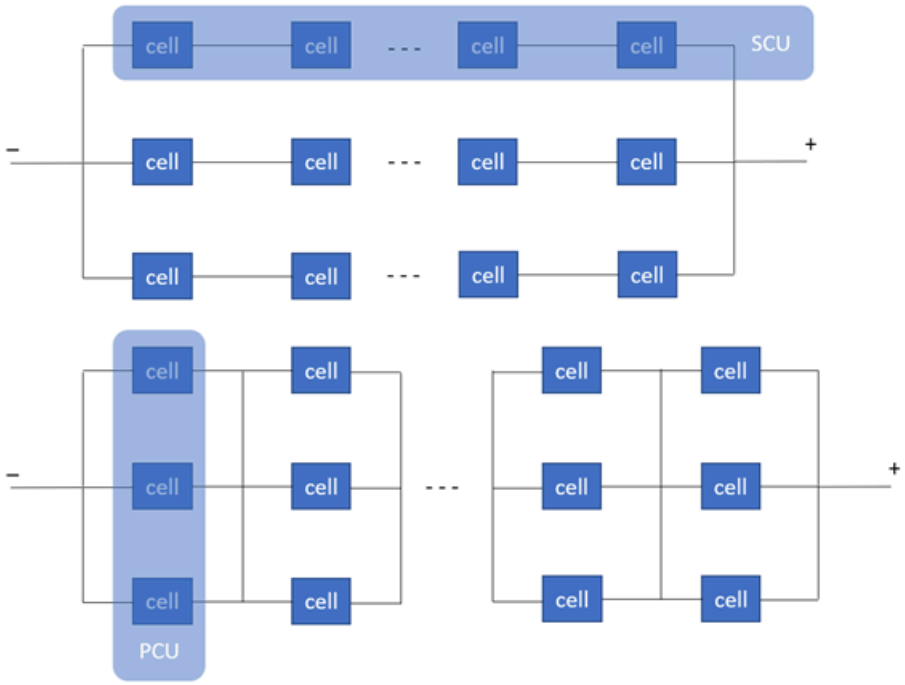


Figure 12: Battery Module Configurations

For example, in a 4s2p case, where the `number_of_parallel_connected_units` = 1, 2 or 4. The corresponding electrical connection of the battery cells would be:

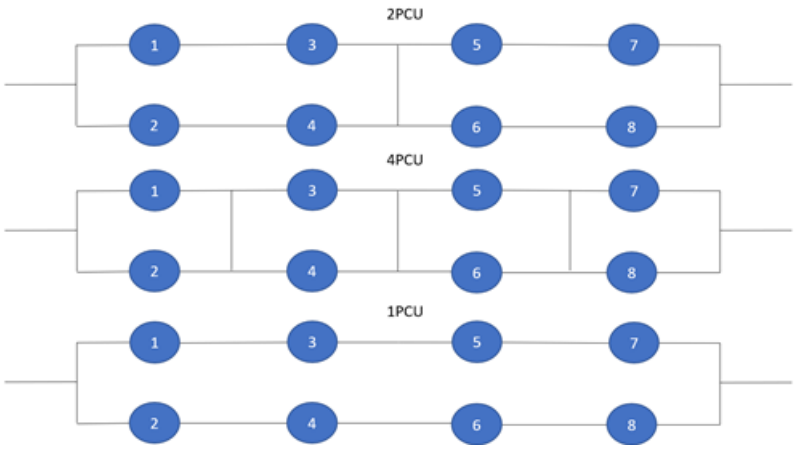


Figure 13: Example PCU Configurations

Multi-Battery Module Treatment

In a battery module, batteries are connected in a nSmP pattern, for example, 8s6p, where m batteries are connected in parallel and then n of these parallel connected units are connected in serial to form a pack. In the MSMD method, two potential equations are solved for the ϕ_1 and ϕ_2 electrical field. Depending on battery connectivity, ϕ_1 and ϕ_2 could be the potential from the positive current collector or the negative one. Therefore, in a battery module simulation, the two potential equations are solved with the transfer current density changing sign dependent on the location in the module. The cells are

alternating as odd and even stages. See figure 4. In tabs and busbars, you only solve for one of the fields. In figure 4, the ϕ_1 and ϕ_2 on the left indicates which field was solved in the tabs/busbars in the line. The governing equations for the cell scale become:

$$\nabla(\sigma\nabla\phi)=j \tag{134}$$

For ϕ_1 in an odd stage:

$$\sigma=\sigma_+ \tag{135}$$

$$j=-I/V \tag{136}$$

For ϕ_1 in an even stage:

$$\sigma=\sigma_- \tag{137}$$

$$j=I/V \tag{138}$$

For ϕ_2 in an odd stage:

$$\sigma=\sigma_- \tag{139}$$

$$j=I/V \tag{140}$$

For ϕ_2 in an even stage:

$$\sigma=\sigma_+ \tag{141}$$

$$j=-I/V \tag{142}$$

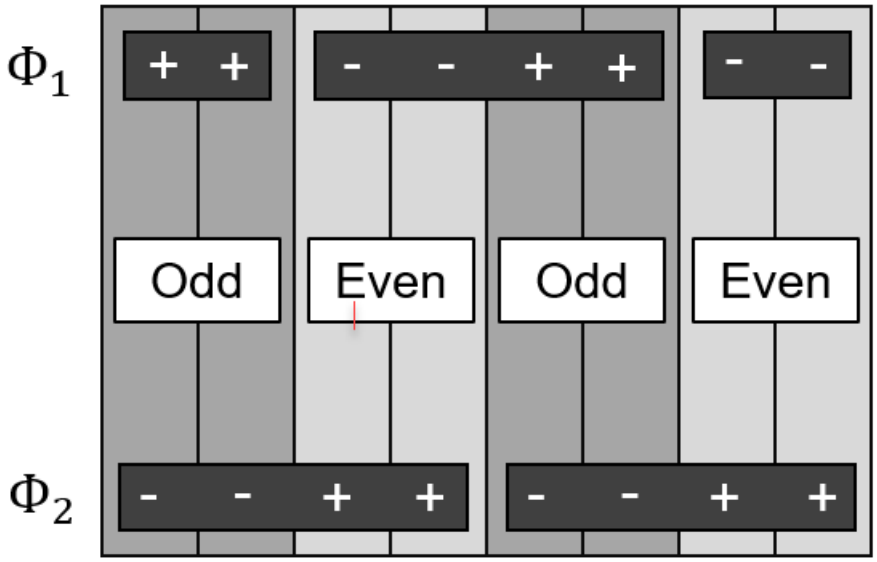


Figure 14: Odd and Even Stages for Cell

Cell Current/Voltage Calculation in Module

To model serial and parallel connected modules, state and model information is needed for each individual cell. The module current I^k is given, but the individual current through each battery (I_j^k) is needed for the electric potential equation, energy equation, ECM model, and the voltage across each PCU as the reference potential. As an example, suppose two PCUs are connected in series. Then the electrical circuit diagram would be as follows:

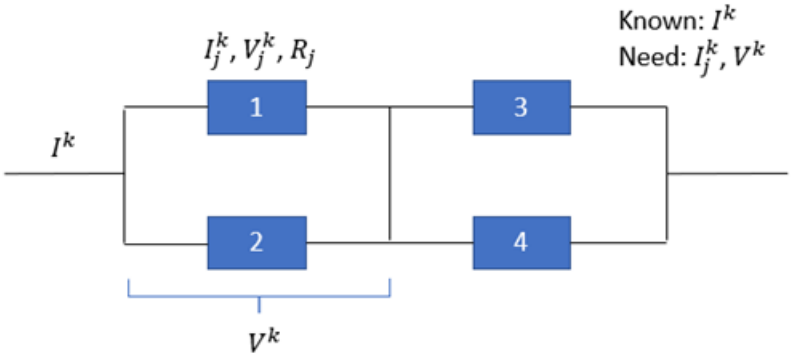


Figure 15: Battery Pack Current and Voltage

To get current and voltage distribution over each cell, the terminal voltage is separated into a "fixed" part (V_j^k) and a "variable" part.

$$V = V_{\text{fixed}} + V_{\text{variable}} \tag{143}$$

$$V_{\text{fixed}} = V_{\text{o.c}} - \sum_i R_i I_R^k = V_j^k \tag{144}$$

$$V_{\text{variable}} = R_0 I_j^k \tag{145}$$

Everything is known for the "fixed" part from the last time step, the reference potential V^k can be calculated across any PCU based on the following equation. I^k is the pack charging/discharging current, which is an input.

$$V^K = \frac{\sum_j \frac{V_j^k}{R_j} - I^k}{\sum_j \frac{1}{R_j}} \tag{146}$$

$$I_j^k = \frac{V_j^k - V_k}{\sum_j R_{s,j}} \tag{147}$$

$I_j^k I_R^k I_j^k$ is then used in the subscale model to calculate I_R^k, R_1, C_1 is the polarization resistance and capacitance from the first pair, R_2, C_2 is the polarization resistance and capacitance from the second pair, $I_{R_1}^k$, $I_{R_2}^k$, $I_{R_1}^{k+1}$, $I_{R_2}^{k+1}$ are diffusion resistor current at time step k and k+1.

$$I_{R_1}^{k+1} = \exp\left(\frac{-\Delta t}{R_1 C_1}\right) I_{R_1}^{k} + \left(1 - \exp\left(\frac{-\Delta t}{R_1 C_1}\right)\right) I_j^k \quad (148)$$

$$I_{R_2}^{k+1} = \exp\left(\frac{-\Delta t}{R_2 C_2}\right) I_{R_2}^{k} + \left(1 - \exp\left(\frac{-\Delta t}{R_2 C_2}\right)\right) I_j^k \quad (149)$$

Electrical Input Type

Different types of electrical input can be included as an input to the battery module using the *electrical_input_type* command. These are summarized in the following sections.

Current

Current supports both generic charging and discharging, these include: constant; piecewise_linear; cubic_spline; and current profiles.

The following gives an example input for a current profile:

```
BATTERY_MODULE( "4S2P battery module" ) {
    number_of_parallel_cells = 2
    number_of_serial_cells = 4
    electrical_input_type = current
    current_type = piecewise_linear
    current_curve_fit_values = Read( "time_current.txt" )
    current_curve_fit_variable = time
    cut_off_current_percent = 0.1
    maximum_cell_voltage = 4.2
    minimum_cell_voltage = 2.7
    minimum_cell_state_of_charge = 0.05
    maximum_cell_state_of_charge = 0.95
}
```

And `time_current.txt` would look like this:

```
0               0.66809696
9.61104062      0.66809696
12.5842409      0.66809696
13.34691228     15.37136268
…
```

Figure 7 shows a typical driving profile with current input. Figure 8 shows the voltage and soc response from this particular current input. The driving profile is taken from [1].

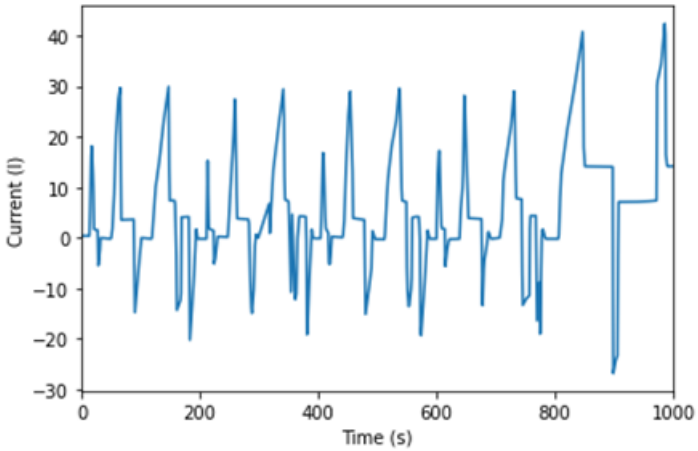


Figure 16: Current Input for a Driving Profile

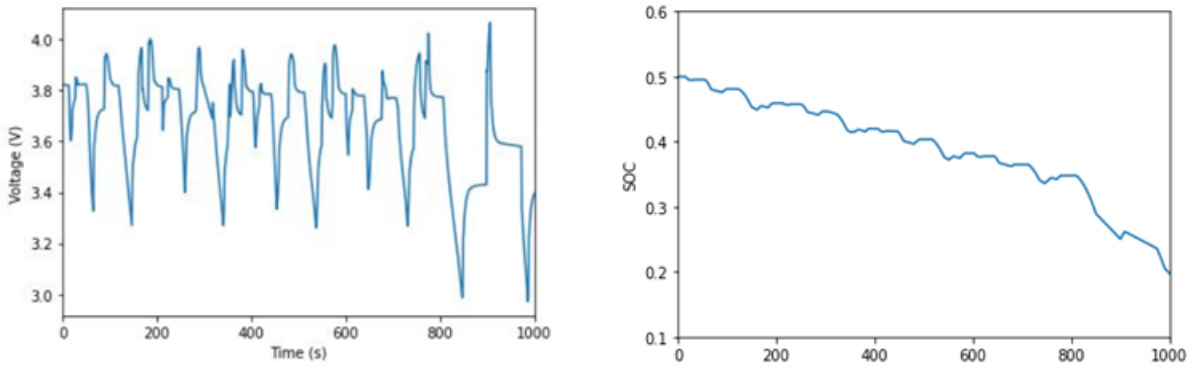


Figure 17: Voltage and soc Output for a Driving Profile

[1] Mahfoudi, Nadjiba, et al. "Thermal Analysis of LMO/Graphite Batteries Using Equivalent Circuit Models." Batteries 7.3 (2021): 58.

C-Rate

C-rate supports both generic charging and discharging, these include: constant; piecewise_linear; cubic_spline; and c-rate profiles.

C-rate is the speed at which a battery is charged/discharged. For example, if you have a battery with a capacity of 3.5Ah, then you can calculate how many amps it can provide for different charging or discharging rates. Below are some examples:

- Battery with capacity of 3.5Ah discharged at a c rate of 0.5C delivers 1.75A for 2 hours.
- Battery with capacity of 3.5Ah discharged at a c rate of 1C delivers 3.5A for 1 hour.
- Battery with capacity of 3.5Ah discharged at a c rate of 2C delivers 7A for 30 mins.

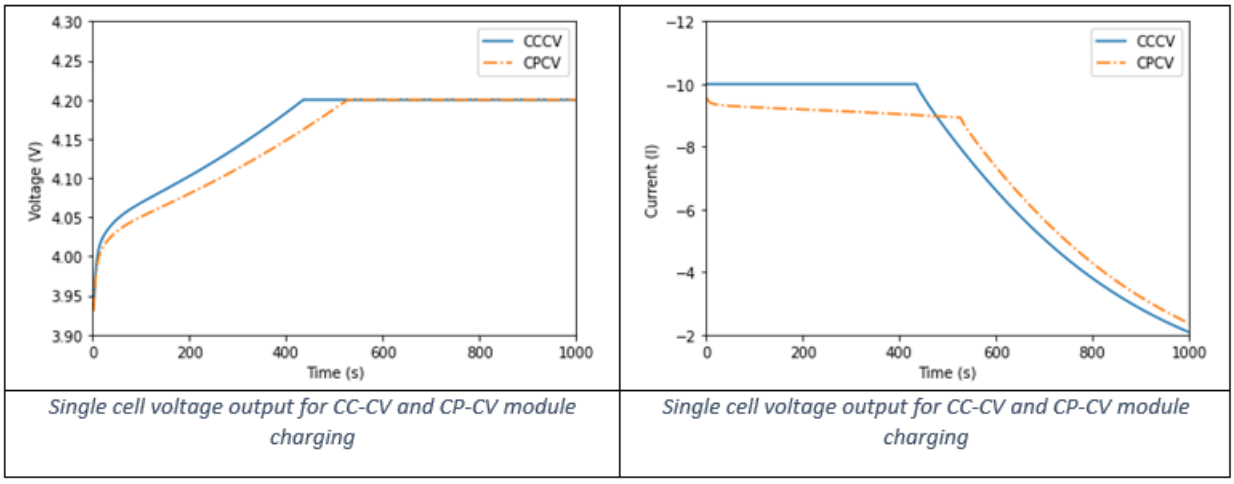
To define a constant c-rate for a 1C discharge the `BATTERY_MODULE` would be:

```
BATTERY_MODULE( "4s2p battery module" ) {
    number_of_parallel_cells = 2
    number_of_serial_cells = 4
    electrical_input_type = c_rate
    c_rate = 1
```

```
    number_of_parallel_connected_units = 2
}
```

Charging Profiles

For charging profiles, you can have CC-CV or CP-CV. For CC-CV the charging starts with a constant current, or c_rate, and then switches to a constant voltage charging method once the voltage limit is reached. The simulation will terminate when the current drops to a pre-defined level, for example, minimum current or maximum soc. For CP-CV, the initial charging is constant power and switches to constant voltage once the voltage limit is reached. The CP-CV charging terminates in the same manner as CC-CV. A typical CC-CV or CP-CV charging current, voltage and soc curve are shown in the following figures.



Single cell voltage output for CC-CV and CP-CV module charging

Single cell voltage output for CC-CV and CP-CV module charging

Figure 18: Voltage and Current Single Cell Responses for CC-CV and CP-CV Charging

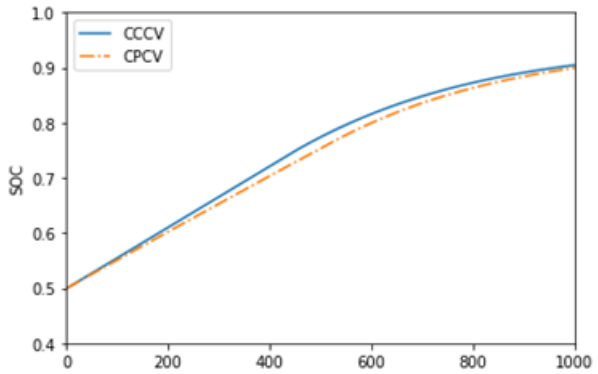


Figure 19: Single Cell soc Response for CC-CV and CP-CV Module Charging

Example CC-CV input: For a CC-CV input, the following parameters need to be defined:

- *electrical_input_type* = standard_charge_profile
- *charging_profile_type* = constant_current_constant_voltage
- *current*/*c_rate*

The following parameters can typically be taken to be default:

- *cut_off_current_percent*
- *maximum_cell_voltage*
- *minimum_cell_voltage*
- *minimum_cell_state_of_charge*
- *maximum_cell_state_of_charge*

```
BATTERY_MODULE( "4S2P battery module" ) {
    number_of_parallel_cells = 2
    number_of_serial_cells = 4
    electrical_input_type = standard_charge_profile
    charging_profile_type = constant_current_constant_voltage
    current_type = constant
    current = -20
    cut_off_current_percent = 0.1
    maximum_cell_voltage = 4.2
    minimum_cell_voltage = 3.0
    minimum_cell_state_of_charge = 0.05
    maximum_cell_state_of_charge = 0.95
}
```

Example input CP-CV: The input for CP-CV the following parameters need to be defined:

- *electrical_input_type* = standard_charge_profile
- *charging_profile_type* = constant_power_constant_voltage
- *power*

The following parameters can typically be taken to be default:

- *cut_off_current_percent*
- *maximum_cell_voltage*
- *minimum_cell_voltage*
- *minimum_cell_state_of_charge*
- *maximum_cell_state_of_charge*

```
BATTERY_MODULE( "4S2P battery module" ) {
    number_of_parallel_cells = 2
    number_of_serial_cells = 4
    electrical_input_type = standard_charge_profile
    charging_profile_type = constant_power_constant_voltage
    power_type = constant
    power = 300
    cut_off_current_percent = 0.1
    maximum_cell_voltage = 4.2
    minimum_cell_voltage = 3.0
    minimum_cell_state_of_charge = 0.05
    maximum_cell_state_of_charge = 0.95
}
```

Constant Power (CP)

CP is charging with constant power, for example, *power_type* = constant.

Example input CP: For a CP input, the following parameters need to be defined:

- *electrical_input_type* = power

- *power*

The following parameters can typically be taken to be default:

- *cut_off_current_percent*
- *maximum_cell_voltage*
- *minimum_cell_voltage*
- *minimum_cell_state_of_charge*
- *maximum_cell_state_of_charge*

```
BATTERY_MODULE( "4S2P battery module" ) {
    number_of_parallel_cells = 2
    number_of_serial_cells = 4
    electrical_input_type = power
    power_type = constant
    power = 300
    cut_off_current_percent = 0.1
    maximum_cell_voltage = 4.2
    minimum_cell_voltage = 2.7
    minimum_cell_state_of_charge = 0.05
    maximum_cell_state_of_charge = 0.95
}
```

Constant Voltage (CV)

CV is charging with constant voltage, for example, *voltage_type* = constant. The input module voltage divided by the number of parallel cells should be between the minimum and maximum cell voltage.

Example input CV: For a CV input, the following parameters need to be defined:

- *electrical_input_type* = voltage
- *voltage*

```
BATTERY_MODULE( "4S2P battery module" ) {
    number_of_parallel_cells = 2
    number_of_serial_cells = 4
    electrical_input_type = voltage
    voltage_type = constant
    voltage = 16
    cut_off_current_percent = 0.1
    maximum_cell_voltage = 4.2
    minimum_cell_voltage = 2.7
    minimum_cell_state_of_charge = 0.05
    maximum_cell_state_of_charge = 0.95
}
```

BATTERY_PACK

Specifies a battery pack and its parameters.

Type

AcuSolve Command

Syntax

`BATTERY_PACK("name"){parameters...}`

Parameters

`number_of_parallel_modules` *(int) ≥1 [=1]*

Number of battery modules connected in parallel.

`number_of_serial_modules` *(int) ≥1 [=1]*

Number of battery modules connected in series.

`electrical_input_type` *(enumerated) [=current]*

Type of global electrical input to battery pack.

current	Electric current input. Requires *`current_type`*.
c_rate	C-rate input. Requires *`c_rate_type`*.
voltage	Voltage input. Requires *`voltage_type`* = constant.
power	Power input. Requires *`power_type`* = constant.
standard_charging_profile or scp	Pre-defined charging profile.

`current_type` *(enumerated) [=constant]*

Type of electrical current input.

constant or const	Constant electrical current. Requires current.
piecewise_linear or linear	Piecewise linear curve fit electrical current profile. Requires *`curve_fit_values`* and *`curve_fit_variable`*.
cubic_spline or spline	Cubic spline curve fit electrical current profile. Requires *`curve_fit_values`* and *`curve_fit_variable`*.

`current` *(real) [=0]*

Constant value of electrical current. When *`current_type`* is constant. Notice you have a positive current for discharging and negative current for charging.

`current_curve_fit_values` *(array) [={0,0}]*

A two-column array of independent-variable/isotropic electrical current data values. The independent variable is time. Used with *`piecewise_linear`* and *`cubic_spline`* types.

current_curve_fit_variable (enumerated) [=time]
Independent variable of the curve fit for isotropic electrical current. Used with *piecewise_linear* and *cubic_spline* types.

time	time

c_rate_type (enumerated) [=none]
Type of c-rate input.

none	No c-rate type specified.
constant or const	Constant c-rate value. Requires c_rate.
piecewise_linear or linear	Piecewise linear curve fit c-rate profile. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit c-rate profile. Requires *curve_fit_values* and *curve_fit_variable*.

c_rate (real) [=0]
Constant value of c-rate. When *c_rate_type* is constant.

c_rate_curve_fit_values (array) [={0,0}]
A two-column array of independent-variable/isotropic c-rate data values. The independent variable is time. Used with *piecewise_linear* and *cubic_spline* types.

c_rate_curve_fit_variable (enumerated) [=time]
Independent variable of the curve fit for isotropic c-rate. Used with *piecewise_linear* and *cubic_spline* types.

time	time

voltage_type (enumerated) [=constant]
Type of voltage input.

constant or const	Constant voltage value. Requires voltage.

voltage (real) [=0]
Constant value of module input voltage. When *voltage_type* = constant.

power_type (enumerated) [=constant]
Type of power input.

constant or const	Constant power value. Requires power.
piecewise_linear or linear	Piecewise linear curve fit power profile. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit power profile. Requires *curve_fit_values* and *curve_fit_variable*.

power (real) [=0]
Constant value of module input power. When *power_type* = constant. Input can be positive or negative.

power_curve_fit_values (array) [={0,0}]
A two-column array of independent-variable/isotropic power data values. The independent variable is time. Used with *piecewise_linear* and *cubic_spline* types.

power_curve_fit_variable (enumerated) [=time]
Independent variable of the curve fit for power. Used with *piecewise_linear* and *cubic_spline* types.

time	time

standard_charging_profile_type (enumerated) [=none]
Type of power input.

none	No charging profile used.
constant_current_constant_voltage or cccv	Constant constant voltage charging profile. Requires a module input current or c_rate to be specified.
constant_power_constant_voltage or cpcv	Constant power constant voltage standard charging profile. Requires module power to be specified.

cut_off_current_percent (real) [=0.01]
Current at which the battery is considered fully charged when using charge profiles.

thermal_runaway_model (string) [=no default]
User-given name of the THERMAL_RUNAWAY_MODEL used with this battery pack.

cell_current_calculation (enumerated) [=none]
How current is calculated in parallel connected battery cells.

none	Current in parallel cells is determined from the current calculated in the busbar connected to the battery cell. The current is determined during the electric potential solve.
constant	Current is calculated assuming the current is uniform across parallel cells.

BATTERY_THERMAL_RUNAWAY_MODEL

Specifies a thermal runaway model and its parameters.

Type

AcuSolve Command

Syntax

```
BATTERY_THERMAL_RUNAWAY_MODEL("name") {parameters...}
```

Parameters

type (enumerated) [NREL_abuse_model]

Type of thermal runaway model.

arc_reaction_model	Fitting of Arrhenius kinetic equations to ARC experimental data.
heat_rate	Direct reading of dT/dt. Can be from ARC or other sources.
NREL_abuse_model	Arrhenius kinetic models for heat release of different cell components based on the NREL thermal runaway model.

equation_type (enumerated) [four_equation]

Type of thermal runaway model.

four_equation	Four equation thermal runaway heat generation model from Hatchard.
five_equation	Hatchard thermal runaway heat generation model (four_equation) with additional heat generation due to electrolyte decomposition.

arc_type (enumerated) [multi_stage_fit]

ARC data fit model.

multi_stage_fit	Parameters for the multi-stage fit of the ARC data.

formulation_type (enumerated) [lumped]

Method used for thermal runaway.

nodal	Thermal runaway heat contribution is calculated at each node in a battery cell.
lumped	Thermal runaway heat contribution is modeled as a lumped parameter for a battery cell.
maximum	Thermal runaway heat contribution is modeled based on the maximum temperature in a battery cell.

ecm_model (enumerated) [yes]
Turn on ecm model for thermal runaway.

yes	Model thermal runaway heat generation with an ECM model included for the batteries operating under normal conditions.
no	Model only thermal runaway heat generation.

internal_short_circuit (enumerated) [off]
Add internal short circuit equation to thermal runaway model.

off	Internal short circuit equation is disabled.
on	Internal short circuit equation is enabled.

internal_short_circuit_model (enumerated) [simple]
Choose different equation to model internal short circuit.

simple	Simpler model does not include consumption of anode and cathode material, from Coman 2017.

internal_short_circuit_temperature (real) [=438.15]
Temperature at which internal short circuit occurs.

sub_iteration_initial_time_step_size (real) [=1000]
Initial time step for sub-stepping the reaction kinetic equation.

sub_iteration_minimum_time_step_size (real) [=1e-4]
Minimum time step for sub-stepping the reaction kinetic equations.

sub_iteration_time_step_size_tolerance (real) [=1e-3]
Sensitivity of the change in sub-step time step size to changes in the solution. For example, a small tolerance value will lead to smaller thermal runaway ODE sub-stepping time step sizes.

sub_iteration_time_step_size_factor (real) [=0.1]
Factor to time with tolerance and minimum time step size for SOC equation.

anode_decomposition_frequency_factor (real) [=2.5e13]
Pre-exponential (frequency) factor for the anode decomposition reaction rate.

sei_decomposition_frequency_factor (real) [=1.67e15]
Pre-exponential (frequency) factor for the sei (solid electrolyte interphase) decomposition reaction rate.

cathode_decomposition_frequency_factor (real) [=6.67e11]
Pre-exponential (frequency) factor for the cathode decomposition reaction rate.

electrochemical_reactions_frequency_factor (real) [=3.37e12]
Pre-exponential (frequency) factor for the electrochemical reaction rate.

electrolyte_decomposition_frequency_factor (real) [=5.14e25]
Pre-exponential (frequency) factor for the electrolyte decomposition reaction rate.

anode_decomposition_activation_energy (real) [=2.24e-19]
Minimum energy needed for the anode decomposition reaction to occur.

sei_decomposition_activation_energy *(real) [=2.24e-19]*
Minimum energy needed for the sei decomposition reaction to occur.

cathode_decomposition_activation_energy *(real) [=2.03e-19]*
Minimum energy needed for the cathode decomposition reaction to occur.

electrochemical_reactions_activation_energy *(real) [=1.58e-19]*
Minimum energy needed for the electrochemical (short circuit) reaction to occur.

electrolyte_decomposition_activation_energy *(real) [=4.544e-19]*
Minimum energy needed for the electrolyte decomposition reaction to occur.

mass_type *(real) [=specific]*
Specify total or specific mass (mass/volume) of the jellyroll.

specific	Specific mass (Kg/m^3).
total	Total mass (Kg).

anode_mass *(real) [=0.006]*
Mass of the anode material in battery core.

cathode_mass *(real) [=0.012]*
Mass of the cathode material in battery core.

sei_mass *(real) [=0.006]*
Mass of the sei in battery core.

electrolyte_mass *(real) [=0.004]*
Mass of the electrolyte in battery core.

anode_decomposition_reaction_enthalpy *(real) [=1.714E6]*
Specific enthalpy for the anode decomposition reaction.

cathode_decomposition_reaction_enthalpy *(real) [=3.14E5]*
Specific enthalpy for the cathode decomposition reaction.

sei_decomposition_reaction_enthalpy *(real) [=2.57E5]*
Specific enthalpy for the sei decomposition reaction.

electrochemical_reaction_enthalpy_type *(enumerated) [=value]*

value	Input enthalpy value directly.
calculate	Calculate enthalpy value based on ECM output voltage or maximum cell voltage (if ECM is off).

electrochemical_reaction_enthalpy *(real) [=4.989E6]*
Enthalpy of the short circuit derived from joule heating.

electrolyte_decomposition_reaction_enthalpy *(real) [=1.55E5]*
Specific enthalpy for the electrolyte decomposition reaction.

initial_anode_lithium_fraction *(real) [=0.75]*
Fractional amount of lithium intercalated in the anode.

initial_sei_lithium_fraction (real) [=0.15]
Fraction of lithium-containing species in the solid electrolyte interphase.

initial_sei_thickness (real) [=0.033]
Thickness of the solid electrolyte interphase.

initial_degree_of_cathode_conversion (real) [=0.04]
Initial degree of conversion of cathode material for cathode decomposition.

initial_electrolyte_fraction (real) [=1.0]
Initial electrolyte fraction.

heat_rate_type (enumerated) [=none]
Type ARC experimental data fitting.

none	No ARC data specified.
piecewise_linear or linear	Piecewise linear curve fit of ARC experimental data.
cubic_spline or spline	Cubic spline curve fit of ARC experimental data.

heat_rate_curve_fit_values (array) [={0,0}]
A two-column array of heat rate (dT/dt) data values. The independent variable must be temperature. Used with piecewise_linear and cubic_spline types.

heat_rate_curve_fit_variable (enumerated) [=temperature]
Independent variable of the curve fit of ARC experimental data. Used with piecewise_linear and cubic_spline types.

temperature	temperature

heat_rate_max_temperature (real) [=700]
Maximum cell temperature recorded during the ARC experiment.

heat_rate_trigger_temperature (real) [=500]
Trigger temperature for thermal runaway recorded during the ARC experiment. At or above this temperature the ARC data is sub-stepped in time for improved accuracy.

heat_rate_onset_temperature (real) [=373.15]
Thermal runaway self-heating onset temperature recorded during the ARC experiment. At or above this temperature the ARC data model is enabled.

parameter_fit_type (arrhenius) [=none]
Type of Arrhenius kinetic parameter fitting approach.

arrhenius	Arrhenius kinetics fitted to each stage and active throughout.
arrhenius_trigger	Arrhenius kinetics fitted to each stage. Final stage heat only active following a trigger temperature.

multi_stage_model_parameters (array) [={}]
An N rows times six-column array of enthalpy (J); activation energy (J); frequency factor (1/s); reaction order n (-); and reaction order m (-); and initial condition (-) values. Each row of the array represents a specific stage (up to five).

CAA_ELEMENT_OUTPUT

Specifies a set of elements for CAA integrated-element output.

Type

AcuSolve Command

Syntax

CAA_ELEMENT_OUTPUT("name") {parameters...}

Qualifier

User-given name.

Parameters

caa_output (string) [no default]
Qualifier of the CAA_OUTPUT command that outputs quantities integrated over the elements defined here and defines the nodes referenced by the element connectivity.

shape (enumerated) [no default]
Topology of the elements in this set.

four_node_tet or tet4	Four-node tetrahedron.
five_node_pyramid or pyramid5	Five-node pyramid.
six_node_wedge or wedge6	Six-node wedge (prism).
eight_node_brick or hex8	Eight-node brick (hexahedron).

elements (array) [no default]
List of element connectivity.

quadrature (enumerated) [=full]
Quadrature rule used for volume integration.

full	Full quadrature.
reduced	Reduced quadrature.

Description

This command specifies a set of elements to be used by a CAA_OUTPUT command (given by *caa_output*) for integrating certain quantities over these elements. Element geometry and integration rules are given here, while the output quantities and related parameters are specified in the referenced CAA_OUTPUT command. For example,

```
CAA_ELEMENT_OUTPUT( "sample elements" ) {
   caa_output    = "sample points"
   shape         = eight_node_brick
```

```
   elements     = { 1, 1, 2, 4, 3, 11, 12, 14, 13 ;
                    2, 3, 4, 6, 5, 13, 14, 16, 15 ; }
   quadrature   = full
}
```

defines two elements that will contribute to any element-integrated quantities specified in the CAA_OUTPUT command qualified by "sample_points". If there are no such quantities in the referenced command, then this command has no effect. The *elements* parameter defines the element connectivity and has the same format as the corresponding parameter in ELEMENT_SET (see that command for format and node ordering), but the elements defined here need not have any relationship to any other elements or element sets. The nodes contained in the element connectivity must all be defined in the referenced CAA_OUTPUT command. Unlike ELEMENT_SET this command does not support higher-order elements (that is, ten-node tetrahedra) or nodal quadrature.

The element connectivity may be read from a file. For the above example, the elements may be placed in a file, such as sample.elm":

```
1 1 2 4 3 11 12 14 13
2 3 4 6 5 13 14 16 15
```

and read by:

```
CAA_ELEMENT_OUTPUT( "sample elements" ) {
   caa_output   = "sample points"
   shape        = eight_node_brick
   elements     = Read( "sample.elm" )
   quadrature   = full
}
```

A separate CAA_ELEMENT_OUTPUT command must be issued for each element shape and each CAA_OUTPUT command that references element integrated quantities.

CAA_OUTPUT

Specifies output parameters for Computational Aero-Acoustics (CAA) post-processing.

Type

AcuSolve Command

Syntax

```
CAA_OUTPUT("name") {parameters...}
```

Qualifier

User-given name.

Parameters

coordinates or *coord* *(array) [no default]*
: An array of sample points. Each row contains a unique identification number and the x, y, and z coordinates of a point in the global xyz coordinate system.

directions or *dir* *(array) [no default]*
: An array of outward normal direction vectors corresponding to the coordinates array. Each row contains an identification number number and the x, y, and z components of a vector in the global xyz coordinate system. Required by some of the output quantities.

gap *(real) >=0 [=0]*
: Tolerance for sample points falling outside of all elements.

output_coordinates *(boolean) [=off]*
: Flag specifying whether to output the coordinates.

output_velocity *(boolean) [=off]*
: Flag specifying whether to output the velocity.

output_pressure *(boolean) [=off]*
: Flag specifying whether to output the pressure.

output_momentum_stress *(boolean) [=off]*
: Flag specifying whether to output the momentum stress.

output_total_stress *(boolean) [=off]*
: Flag specifying whether to output the total stress.

output_lighthill_stress *(boolean) [=off]*
: Flag specifying whether to output the Lighthill stress.

output_reduced_lighthill_stress *(boolean) [=off]*
: Flag specifying whether to output the reduced Lighthill stress.

output_div_momentum_stress *(boolean) [=off]*
: Flag specifying whether to output the divergence of the momentum stress.

output_div_total_stress (boolean) [=off]
Flag specifying whether to output the divergence of the total stress.

output_div_lighthill_stress (boolean) [=off]
Flag specifying whether to output the divergence of the Lighthill stress.

output_div_reduced_lighthill_stress (boolean) [=off]
Flag specifying whether to output the divergence of the reduced Lighthill stress.

output_normal_momentum_flux (boolean) [=off]
Flag specifying whether to output the normal component of the momentum flux.

output_normal_momentum_flux_rate (boolean) [=off]
Flag specifying whether to output the normal component of the momentum flux rate.

output_normal_div_total_stress (boolean) [=off]
Flag specifying whether to output the normal component of the divergence of the total stress.

output_fwh_monopole (boolean) [=off]
Flag specifying whether to output the monopole term of the Ffowcs Williams Hawkings acoustics equations.

output_fwh_dipole (boolean) [=off]
Flag specifying whether to output the dipole term of the Ffowcs Williams Hawkings acoustics equations.

output_ei_na (boolean) [=off]
Flag specifying whether to output the element-integrated volume.

output_ei_na_velocity (boolean) [=off]
Flag specifying whether to output the element-integrated velocity.

output_ei_na_pressure (boolean) [=off]
Flag specifying whether to output the element-integrated pressure.

output_ei_na_momentum_stress (boolean) [=off]
Flag specifying whether to output the element-integrated momentum stress.

output_ei_na_total_stress (boolean) [=off]
Flag specifying whether to output the element-integrated total stress.

output_ei_na_lighthill_stress (boolean) [=off]
Flag specifying whether to output the element-integrated Lighthill stress.

output_ei_na_reduced_lighthill_stress (boolean) [=off]
Flag specifying whether to output the element-integrated reduced Lighthill stress.

output_ei_na_div_momentum_stress (boolean) [=off]
Flag specifying whether to output the element-integrated divergence of the momentum stress.

output_ei_na_div_total_stress (boolean) [=off]
Flag specifying whether to output the element-integrated divergence of the total stress.

output_ei_na_div_lighthill_stress (boolean) [=off]
Flag specifying whether to output the element-integrated divergence of the Lighthill stress.

output_ei_na_div_reduced_lighthill_stress (boolean) [=off]
Flag specifying whether to output the element-integrated divergence of the reduced Lighthill stress.

output_ei_gna_div_momentum_stress (boolean) [=off]
Flag specifying whether to output the element-integrated shape function gradient contracted with the divergence of the momentum stress.

output_ei_gna_div_total_stress (boolean) [=off]
Flag specifying whether to output the element-integrated shape function gradient contracted with the divergence of the total stress.

output_ei_gna_div_lighthill_stress (boolean) [=off]
Flag specifying whether to output the element-integrated shape function gradient contracted with the divergence of the Lighthill stress.

output_ei_gna_div_reduced_lighthill_stress (boolean) [=off]
Flag specifying whether to output the element-integrated shape function gradient contracted with the divergence of the reduced Lighthill stress.

output_si_na (boolean) [=off]
Flag specifying whether to output the surface-integrated area.

output_si_na_normal (boolean) [=off]
Flag specifying whether to output the surface-integrated normal.

output_si_na_pressure (boolean) [=off]
Flag specifying whether to output the surface-integrated pressure.

output_si_na_normal_pressure (boolean) [=off]
Flag specifying whether to output the surface-integrated normal-pressure.

output_si_na_normal_momentum_flux (boolean) [=off]
Flag specifying whether to output the surface-integrated normal component of the momentum flux.

output_si_na_normal_momentum_flux_rate (boolean) [=off]
Flag specifying whether to output the surface-integrated normal component of the momentum flux rate.

output_si_na_normal_div_total_stress (boolean) [=off]
Flag specifying whether to output the surface-integrated normal component of the divergence of the total stress.

output_si_na_fwh_monopole (boolean) [=off]
Flag specifying whether to output the surface-integrated monopole term of the Ffowcs Williams Hawkings acoustics equations.

output_si_na_fwh_dipole (boolean) [=off]
Flag specifying whether to output the surface-integrated dipole term of the Ffowcs Williams Hawkings acoustics equations.

reference_density or *ref_dens (real) [=1.225]*
Reference density. Required by some of the output quantities.

reference_pressure or *ref_pres* *(real) [=0]*
: Reference pressure. Required by some of the output quantities.

reference_sound_speed or *sound* *(real) [=330]*
: Reference sound speed. Required by some of the output quantities.

output_frequency or *out_freq* *(integer) >=0 [=1000]*
: Time step frequency at which to output CAA data. If zero, this option is ignored.

output_time_interval or *out_intv* *(real) >=0 [=0]*
: Time frequency at which to output CAA data. If zero, this option is ignored.

num_saved_states or *save* *(integer) >=0 [=0]*
: Number of CAA data files to keep. If zero, all are kept.

Description

This command specifies output data to support Computational Aero-Acoustics (CAA) post-processing. For example,

```
CAA_OUTPUT( "sample points" ) {
    coordinates                                 = { 1, 0., 0., 0. ;
                                                    2, 1., 0., 0. ;
                                                   11, 0., 1., 0. ;
                                                   12, 1., 1., 0. ; }
    directions                                  = { 1, 0., 0., 1. ;
                                                    2, 0., 0., 1. ;
                                                   11, 0., 0., 1. ;
                                                   12, 0., 0., 1. ; }
    gap                                         = 0
    output_coordinates                          = on
    output_velocity                             = on
    output_pressure                             = off
    output_momentum_stress                      = off
    output_total_stress                         = off
    output_lighthill_stress                     = off
    output_reduced_lighthill_stress             = off
    output_div_momentum_stress                  = off
    output_div_total_stress                     = off
    output_div_lighthill_stress                 = off
    output_div_reduced_lighthill_stress         = off
    output_normal_momentum_flux                 = off
    output_normal_momentum_flux_rate            = off
    output_normal_div_total_stress              = off
    output_fwh_monopole                         = off
    output_fwh_dipole                           = off
    output_ei_na                                = off
    output_ei_na_velocity                       = off
    output_ei_na_pressure                       = off
    output_ei_na_momentum_stress                = off
    output_ei_na_total_stress                   = off
    output_ei_na_lighthill_stress               = off
    output_ei_na_reduced_lighthill_stress       = off
    output_ei_na_div_momentum_stress            = off
    output_ei_na_div_total_stress               = off
    output_ei_na_div_lighthill_stress           = off
    output_ei_na_div_reduced_lighthill_stress   = off
    output_ei_gna_div_momentum_stress           = off
    output_ei_gna_div_total_stress              = off
    output_ei_gna_div_lighthill_stress          = off
```

```
    output_ei_gna_div_reduced_lighthill_stress = off
    output_si_na                               = off
    output_si_na_normal                        = off
    output_si_na_normal_pressure               = off
    output_si_na_normal_momentum_flux          = off
    output_si_na_normal_momentum_flux_rate     = off
    output_si_na_normal_div_total_stress       = off
    output_si_na_fwh_monopole                  = off
    output_si_na_fwh_dipole                    = off
    reference_density                          = 1.225
    reference_pressure                         = 0
    reference_sound_speed                      = 330
    output_frequency                           = 1
    output_time_interval                       = 0
    num_saved_states                           = 0
}
```

A set of sample points is specified by the *coordinates* parameter, which is a four-column array corresponding to an identification number and the x, y, and z coordinates of each sample point, one row per point. The identification number may be any arbitrary but unique (within the set) integer. Negative numbers are acceptable. The coordinates are in the global xyz coordinate system. This is the same format as the *coordinates* parameter of the COORDINATE command, but the sample points do not have to be a subset of the nodal coordinates nor related to them in any other way. The *coordinates* parameter is mandatory. Within AcuPrep, an element is found for each sample point and interpolation shape functions are computed. If a sample point falls outside all fluid elements, then the closest element face is found, up to a distance specified by *gap*. An error is issued if no element is found.

Some of the output fields require a direction vector. If needed, the *directions* parameter must be given as a four column array: identification number, and the x, y, and z components of the direction. The identification numbers must be the same set as given by coordinates. The parameters *reference_density*, *reference_pressure* and *reference_sound_speed* provide the required data for certain fields. Any of these extra parameters that a particular field requires are noted in the table below.

The sample points and normal directions may be read from files. For the above example, the sample points may be placed in a file, such as `channel.caa.crd`:

```
1 0. 0. 0.
2 1. 0. 0.
11 0. 1. 0.
12 1. 1. 0.
```

and the normal directions placed in another file, such as `channel.caa.dir`:

```
1 0. 0. 1.
 2 0. 0. 1.
11 0. 0. 1.
12 0. 0. 1.
```

Both may then by read by:

```
CAA_OUTPUT( "sample points" ) {
   coordinates = Read( "channel.caa.crd" )
   directions = Read( "channel.caa.dir" )
   ...
}
```

Since the volume of data written on disk by `CAA_OUTPUT` could be very large, this command provides the option of selecting which data are to be written on disk via the parameters *output_coordinates* through *output_si_na_fwh_dipole*. The default is to not output any fields. The output quantities and their required data are defined in the following table:

Name	DOFs	Definition	Data
coordinates	3	**x**	
velocity	3	**u**	
pressure	1	p	
momentum_stress	6	$\rho \boldsymbol{u} \otimes \boldsymbol{u}$	
total_stress	6	$\rho \boldsymbol{u} \otimes \boldsymbol{u} + p\boldsymbol{d} - \tau$	
lighthill_stress	6	$\boldsymbol{T} = \rho \boldsymbol{u} \otimes \boldsymbol{u} + [(p - p_0) - c^2(\rho - \rho_0)]\boldsymbol{d} - \tau$	ρ_0, p_0, c
reduced_lighthill_stress	6	$\boldsymbol{R} = \rho \boldsymbol{u} \otimes \boldsymbol{u} - \tau$	
div_momentum_stress	3	$\nabla \bullet (\rho \boldsymbol{u} \otimes \boldsymbol{u})$	
div_total_stress	3	$\nabla \bullet (\rho \boldsymbol{u} \otimes \boldsymbol{u} + p\boldsymbol{d} - \tau)$	
div_lighthill_stress	3	$\overline{\nabla \bullet \boldsymbol{T}}$	ρ_0, p_0, c
div_reduced_lighthill_stress	3	$\nabla \bullet \boldsymbol{R}$	
normal_momentum_flux	1	$\boldsymbol{n} \bullet (\rho \boldsymbol{u})$	$\boldsymbol{n}$
normal_momentum_flux_rate	1	$\boldsymbol{n} \bullet \frac{\partial(\rho \boldsymbol{u})}{\partial t}$	$\boldsymbol{n}$
normal_div_total_stress	1	$\boldsymbol{n} \bullet [\nabla \bullet (\rho \boldsymbol{u} \otimes \boldsymbol{u} + p\boldsymbol{d} - \tau)]$	$\boldsymbol{n}$
fwh_monopole	1	$\boldsymbol{n} \bullet [\rho(\boldsymbol{u} - \boldsymbol{v}) + \rho_0 \boldsymbol{v}]$	ρ_0
fwh_dipole	1	$\boldsymbol{n} \bullet \{\nabla \bullet [\rho \boldsymbol{u} \otimes (\boldsymbol{u} - \boldsymbol{v}) + (p - p_0)\boldsymbol{d}]\}$	ρ_0
ei_na	1	$\int_e Na \, dV$	

Name	DOFs	Definition	Data
ei_na_velocity	3	$\int_e Na\,\boldsymbol{u}\,dV$	
ei_na_pressure	1	$\int_e Na\,p\,dV$	
ei_na_momentum_stress	6	$\int_e Na(\rho\boldsymbol{u}\otimes\boldsymbol{u})dV$	
ei_na_total_stress	6	$\int_e Na(\rho\boldsymbol{u}\otimes\boldsymbol{u}+p\boldsymbol{d}-\tau)dV$	
ei_na_lighthill_stress	6	$\int_e Na\,\boldsymbol{T}\,dV$	ρ_0, p_0, c
ei_na_reduced_lighthill_stres	6	$\int_e Na\,\boldsymbol{R}\,dV$	
ei_na_div_momentum_stress	3	$\int_e Na\,\nabla\bullet(\rho\boldsymbol{u}\otimes\boldsymbol{u})dV$	
ei_na_div_total_stress	3	$\int_e Na\,\nabla\bullet(\rho\boldsymbol{u}\otimes\boldsymbol{u}+p\boldsymbol{d}-\tau)dV$	
ei_na_div_lighthill_stress	3	$\int_e Na\,\nabla\bullet\boldsymbol{T}\,dV$	ρ_0, p_0, c
ei_na_div_reduced_lighthill_s	3	$\int_e Na\,\nabla\bullet\boldsymbol{R}\,dV$	
ei_gna_div_momentum_stres	1	$\int_e(\nabla Na)\bullet(\nabla\bullet(\rho\boldsymbol{u}\otimes\boldsymbol{u}))dV$	
ei_gna_div_total_stress	1	$\int_e(\nabla Na)\bullet[\nabla\bullet(\rho\boldsymbol{u}\otimes\boldsymbol{u}+p\boldsymbol{d}-\tau)]dV$	
ei_gna_div_lighthill_stress	1	$\int_e(\nabla Na)\bullet(\nabla\bullet\boldsymbol{T})dV$	ρ_0, p_0, c

Name	DOFs	Definition	Data
ei_gna_div_reduced_lighthill_	1	$\int_e (\nabla Na) \bullet (\nabla \bullet \boldsymbol{R}) dV$	
si_na	1	$\int_s Na \, dA$	
si_na_normal	3	$\int_s Na \, \boldsymbol{n} \, dA$	
si_na_pressure	1	$\int_s Na \quad p \, dA$	
si_na_normal_pressure	3	$\int_s Na \, \boldsymbol{n} \, p \, dA$	
si_na_normal_momentum_flu	1	$\int_s Na \, \boldsymbol{n} \bullet (\rho \boldsymbol{u}) dA$	
si_na_normal_momentum_flu	1	$\int_s Na \, \boldsymbol{n} \bullet \frac{\partial(\rho \boldsymbol{u})}{\partial t} dA$	$\boldsymbol{n}$
si_na_normal_div_total_stres	1	$\int_s Na \, \boldsymbol{n} \bullet [\nabla \bullet (\rho \boldsymbol{u} \otimes \boldsymbol{u} + p\boldsymbol{d} - \tau)] dA$	$\boldsymbol{n}$
si_na_fwh_monopole	1	$\int_s Na \, \boldsymbol{n} \bullet [\rho(\boldsymbol{u} - \boldsymbol{v}) + \rho_0 \boldsymbol{v}] dA$	ρ_0
si_na_fwh_dipole	1	$\int_s Na \, \boldsymbol{n} \bullet \{\nabla \bullet [\rho \boldsymbol{u} \otimes (\boldsymbol{u} - \boldsymbol{v}) + (p - p_0)\boldsymbol{d}]\}$	ρ_0

where **x** is the coordinate vector; **u** is the velocity; **v** is the mesh velocity; **n** is the outward normal direction, given by *directions*; p is the pressure; p_0 is the reference pressure, given by *reference_pressure*; ρ is the density; ρ_0 is the reference density, given by *reference_density*; c is the reference speed of sound, given by *reference_sound_speed*; **d** is the Kronecker delta (identity matrix); τ is the viscous stress tensor (including turbulence effects); and Na is the element shape function. The stress-type output quantities with 6 DOFs are ordered as xx, yy, zz, xy, yz, zx. Due to the mesh velocity term the *fwh_monopole* variables do not contain the usual time derivative.

The variables starting with "ei_" are element-integrated. The elements that contribute to the integration are specified by the CAA_ELEMENT_OUTPUT commands that reference the qualifier of the CAA_OUTPUT command.

The element connectivity is defined in terms of the sample points in *coordinates*. If any of these points is not referenced by the element connectivity then the element integrals are zero at that point. Similarly, variables starting with "si_" are surface-integrated, and the surfaces that contribute to the integration are specified by the CAA_SURFACE_OUTPUT commands that reference the qualifier of the CAA_OUTPUT command. Unreferenced points again yield zero integrals. In both cases, the integrals appearing in the definitions above include sums over the appropriate shape functions. Because of the inclusion of the shape function each integral is local to the sample points and effectively yields a smoothing of the integrand. For example,

```
CAA_OUTPUT( "sample points" ) {
    ...
    output_ei_na                  = on
    output_ei_na_lighthill_stress = on
    output_si_na                  = on
    output_si_na_fwh_monopole     = on
    ...
}
CAA_ELEMENT_OUTPUT( "sample elements" ) {
    caa_output                    = "sample points"
    shape                         = eight_node_brick
    elements                      = Read( "sample.elm" )
}
CAA_SURFACE_OUTPUT( "sample surfaces" ) {
    caa_output                    = "sample points"
    shape                         = four_node_quad
    partial_surfaces              = Read( "sample.srf" )
}
```

The parameter *output_frequency* determines how often the CAA data will be output, in number of steps. *output_time_interval* does the same, except in time. If either of these parameters is non-zero, the CAA data will be output at the end of the run. If both are zero, no data are written to disk.

Run times may not coincide with *output_time_interval*. In these cases the data are output at every time step which passes through a multiple of *output_time_interval*.

The *num_saved_states* parameter indicates the number of CAA data files to save. Once the (*num_saved_states* + 1)th file is written, the first one is removed. When running in parallel, once all files are written (in parallel), then the old ones are removed (in parallel).

Once the CAA data have been written to disk, they can be translated to other formats using the AcuTrans program; see the *AcuSolve Programs Reference Manual* for details. Access to the CAA results are provided also via the C-callable API library, libadb.a (Acusim Data Base), and its python/perl/shell bindings/access programs. See variables "nOaas" through "oaaValues" (15 variables total) in doc/adb .txt.

CAA_SURFACE_OUTPUT

Specifies a set of surfaces for CAA integrated-surface output.

Type

AcuSolve Command

Syntax

CAA_SURFACE_OUTPUT("name") {parameters...}

Qualifier

User-given name.

Parameters

caa_output (string) [no default]

Qualifier of the CAA_OUTPUT command that outputs quantities integrated over the surfaces defined here.

shape (enumerated) [no default])

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral.

partial_surfaces (array) [no default]

List of surfaces with no references to elements or element sets.

quadrature (enumerated) [=full]

Quadrature rule used for surface integration.

full	Full quadrature.
reduced	Reduced quadrature.

Description

This command specifies a set of surfaces to be used by a CAA_OUTPUT command, given by *caa_output*, for integrating certain quantities over these surfaces. Surface geometry and integration rules are given here, while the output quantities and related parameters are specified in the referenced CAA_OUTPUT command. For example,

```
CAA_SURFACE_OUTPUT( "sample surfaces" ) {
   caa_output       = "sample points"
   shape            = four_node_quad
   partial_surfaces = { 1, 1, 2, 4, 3 ;
                        2, 3, 4, 6, 5 ; }
   quadrature       = full
}
```

defines two surfaces that will contribute to any surface-integrated quantities specified in the CAA_OUTPUT command qualified by "sample_points". If there are no such quantities in the referenced command, then this command has no effect.

The *partial_surfaces* parameter contains the connectivity of the surfaces. This parameter is a multi-column array, but it does not have the same format as the corresponding parameter in other surface commands, for example, SURFACE_OUTPUT. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has four columns, corresponding to a unique (within this set) surface number and the three nodes of the surface. For four_node_quad, *partial_surfaces* has five columns, corresponding to a surface number and the four nodes of the surface. One row per surface must be given. Since there is no parent element definition the three or four nodes of the surface must be ordered around the periphery of the surface. Also unlike other surface commands this command does not support higher-order surfaces, that is, six-node triangles, or nodal quadrature. The nodes contained in the surface connectivity must all be defined in the referenced CAA_OUTPUT command.

The surface connectivity may be read from a file. For the above example, the elements may be placed in a file, such as sample.srf:

```
1 1 2 4 3
2 3 4 6 5
```

and read by:

```
CAA_SURFACE_OUTPUT( "sample surfaces" ) {
   caa_output       = "sample points"
   shape            = four_node_quad
   partial_surfaces = Read( "sample.srf" )
   quadrature       = full
}
```

A separate CAA_SURFACE_OUTPUT command must be issued for each surface shape and each CAA_OUTPUT command that references surface integrated quantities.

ELEMENT_BOUNDARY_CONDITION

Specifies an element boundary condition for a solution field on a set of element faces.

Type

AcuSolve Command

Syntax

ELEMENT_BOUNDARY_CONDITION("name") {parameters...}

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral
six_node_triangle or tri6	Six-node triangle.

element_set or elem_set (string) [no default]

User-given name of the parent element set.

surfaces (array) [no default]

List of element surfaces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this element boundary condition. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to the ELEMENT_BOUNDARY_CONDITION command. This option is used in place of directly specifying these parameters. In the event that both of the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

variable or var (enumerated) [no default]

Boundary condition variable. All are scalars except *tangential_traction* is a three-component vector.

mass_flux or mass	Mass flux (mass flow rate).
pressure or pres	Pressure.

stagnation_pressure or stag_pres	Stagnation pressure.
tangential_traction or trac	Tangential components of traction.
heat_flux or heat	Thermal heat flux.
convective_heat_flux or conv_heat	Convective heat flux.
radiation_heat_flux or rad_heat	Radiation heat flux.
species_1_flux or spec1	Species 1 flux.
convective_species_1_flux or conv_spec1	Convective flux for species 1.
species_2_flux or spec2	Species 2 flux.
convective_species_2_flux or conv_spec2	Convective flux for species 2.
species_3_flux or spec3	Species 3 flux.
convective_species_3_flux or conv_spec3	Convective flux for species 3.
species_4_flux or spec4	Species 4 flux.
convective_species_4_flux or conv_spec4	Convective flux for species 4.
species_5_flux or spec5	Species 5 flux.
convective_species_5_flux or conv_spec5	Convective flux for species 5.
species_6_flux or spec6	Species 6 flux.
convective_species_6_flux or conv_spec6	Convective flux for species 6.
species_7_flux or spec7	Species 7 flux.
convective_species_7_flux or conv_spec7	Convective flux for species 7.
species_8_flux or spec8	Species 8 flux.
convective_species_8_flux or conv_spec8	Convective flux for species 8.

species_9_flux or spec9	Species 9 flux.
convective_species_9_flux or conv_spec9	Convective flux for species 9.
field_flux or field	Multi field flux.
convective_field_flux or conv_field	Multi field convective flux.
turbulence_flux or turb	Turbulence diffusion flux.
kinetic_energy_flux or tke	Turbulence kinetic energy flux.
dissipation_rate_flux or teps	Turbulence dissipation rate flux.
eddy_frequency_flux or tomega	Turbulence eddy frequency flux.
intermittency_flux or tintc	Transition intermittency flux.
transition_re_theta_flux or treth	Transition Re-theta flux.
design_topology_filter_flux or topofiltflux	Design topology filter flux.

type (enumerated) [=zero]
Type of the boundary condition.

zero	Zero for the set.
constant or const	Constant value. Requires *constant_value*.
free	Computes the boundary values from the solution field.
outflow or out	An outflow condition for the *mass_flux* variable.
inflow or in	An inflow condition for the *mass_flux* variable.
per_surface or surf	Surface values. Requires *surface_values*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *curve_fit_values* and *curve_fit_variable*.
user_function or user	User-defined function. Requires *user_function*, *user_values* and *user_strings*.

constant_value or value (real) [=0]
Constant scalar value of the boundary condition. Used with constant type and scalar variables.

constant_values (array) [={0,0,0}]
Constant vector value of the boundary condition. Used with constant type and vector variables (currently only *tangential_traction*).

field (string) [no default]
Value of the boundary condition for field. Used with *variable* field.

surface_values or values (array) [no default]
A two-column (for scalar variables) or four-column (for vector variables, currently only *tangential_traction*) array of surface/boundary-condition data values. Used with per_surface type.

curve_fit_values or curve_values (array) [={0,0}]
A two-column (for scalar variables) or four-column (for vector variables, currently only *tangential_traction*) array of independent-variable/boundary-condition data values. Used with piecewise_linear and cubic_spline types.

curve_fit_variable or curve_var (enumerated) [=temperature]
Independent variable of the curve fit. Used with piecewise_linear and cubic_spline types.

x_coordinate or xcrd	X-component of coordinates.
y_coordinate or ycrd	Y-component of coordinates.
z_coordinate or zcrd	Z-component of coordinates.
x_reference_coordinate or xrefcrd	X-component of reference coordinates.
y_reference_coordinate or yrefcrd	Y-component of reference coordinates.
z_reference_coordinate or zrefcrd	Z-component of reference coordinates.
x_velocity or xvel	X-component of velocity.
y_velocity or yvel	U-component of velocity.
z_velocity or zvel	Z-component of velocity.
velocity_magnitude or vel_mag	Velocity magnitude.
normal_velocity	Normal velocity.
pressure or pres	Pressure.
temperature or temp	Temperature.
relative_humidity	Relative humidity.
dewpoint_temperature	Dewpoint temperature.

eddy_viscosity or eddy	Turbulence kinematic eddy viscosity.
kinetic_energy or tke	Turbulence kinetic energy.
dissipation_rate or teps	Turbulence dissipation rate.
intermittency or tintc	Transition intermittency.
transition_re_theta or treth	Transition Re-theta.
eddy_frequency or tomega	Turbulence eddy frequency.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.
species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.
species_7 or spec7	Species 7.
species_8 or spec8	Species 8.
species_9 or spec9	Species 9.
mesh_x_displacement or mesh_xdisp	X-component of mesh displacement.
mesh_y_displacement or mesh_ydisp	Y-component of mesh displacement.
mesh_z_displacement or mesh_zdisp	Z-component of mesh displacement.
mesh_displacement_magnitude **or** ***mesh_disp_mag***	Mesh displacement magnitude.
mesh_x_velocity **or** ***mesh_xvel***	X-component of mesh velocity.
mesh_y_velocity **or** ***mesh_yvel***	Y-component of mesh velocity.
mesh_z_velocity **or** ***mesh_zvel***	Z-component of mesh velocity.
mesh_velocity_magnitude **or** ***mesh_vel_mag***	Mesh velocity magnitude.

user_function or *user* *(string) [no default]*
Name of the user-defined function. Used with user_function type.

user_values (array) [={}]
Array of values to be passed to the user-defined function. Used with user_function type.

user_strings (list) [={}]
Array of strings to be passed to the user-defined function. Used with user_function type.

multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the boundary condition values. If none, no scaling is performed.

reference_temperature or ref_temp (real) [=273.15]
Reference temperature for the convective and radiation heat flux boundary conditions

reference_temperature_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the reference temperature. If none, no scaling is performed.

reference_species or ref_spec (real) [=0]
Reference species for the convective species boundary condition.

reference_species_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the reference species. If none, no scaling is performed.

non_reflecting_factor (real) >=0 [=0]
Amount of non-reflecting modification. If zero, there is no effect. If one, waves can pass through the boundary without reflection. Used with *pressure* variable at outflow boundaries and *mass_flux* variable at inflows. Turning on both *non_reflecting_bc_running_average_field* and *running_average* specifies the use of running average fields to enhance the performance of non-reflective boundary conditions when *flow* = navier_stokes is used.

> **Note:** The *non_reflecting_bc_running_average_field* is a parameter of AcuSolve. By default, it is set to TRUE. You can enable the running_average option under EQUATION of the Global Command.

When *flow* = compressible_navier_stokes is set, the running_average option does not need to be turned on.

pressure_loss_factor (real) >=0 [=0]
Coefficient of a pressure loss term added/subtracted to outflow/inflow pressure boundary conditions. Used with pressure variable.

pressure_loss_factor_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the pressure loss factor. Used with pressure variable. If none, no scaling is performed.

hydrostatic_pressure (boolean) [=off]
Flag specifying whether to add hydrostatic pressure to pressure and stagnation pressure boundary conditions.

hydrostatic_pressure_origin (array) [={0,0,0}]
Coordinates of any location where the hydrostatic pressure is zero. Used with *hydrostatic_pressure*=on, and pressure and stagnation pressure boundary conditions.

active_type (enumerated) [=all]
Type of the active flag. Determines which surfaces in this set will have element boundary conditions imposed by this command.

all	All surfaces in this set are active.
none	No surfaces in this set are active.
no_interface	Only surfaces that are not in an interface surface set or do not find a contact surface of an appropriate medium are active.

Description

This command specifies element boundary conditions for a solution variable on a set of surfaces (element faces). The surfaces of an element boundary condition are defined with respect to the elements of an element set. For example,

```
ELEMENT_SET( "flow elements" ) {
    shape           = four_node_tet
    elements        = { ...
                        4, 2, 5, 6, 8 ;
                        5, 2, 6, 3, 5 ;
                        ... }
...
}
ELEMENT_BOUNDARY_CONDITION( "constant BC on heated wall" ) {
    shape           = three_node_triangle
    element_set     = "flow elements"
    surfaces        = { 4, 41, 2, 5, 6 ;
                        5, 51, 5, 6, 3 ; }
    variable        = heat_flux
    type            = constant
    constant_value  = 12.
}
```

defines a thermal heat flux element boundary condition, with a constant value of 12 on two surfaces of the element set "flow elements".

There are two main forms of this command. The legacy version (or single topology version) of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad
six_node_wedge	four_node_quad

Element Shape	**Surface Shape**
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number (of the parent element set), a unique (within this set) surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as `heated_wall.ebc`:

```
4 41 2 5 6
5 51 5 6 3
```

and read by:

```
ELEMENT_BOUNDARY_CONDITION( "constant BC on heated wall" ) {
 shape           = three_node_triangle
 element_set     = "flow elements"
 surfaces        = Read( "heated_wall.ebc" )
 variable        = heat_flux
 type            = constant
 constant_value  = 12.
}
```

The mixed topology form of the `ELEMENT_BOUNDARY_CONDITION` command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the `SURFACE_SET` command, and are then referred to by the `ELEMENT_BOUNDARY_CONDITION` command. For example, a collection of triangular and quadrilateral element faces can be defined using the following `SURFACE_SET` commands.

```
SURFACE_SET( "tri faces" ) {
   surfaces    = { 1, 1, 1, 2, 4 ;
                   2, 2, 3, 4, 6 ;
                   3, 3, 5, 6, 8 ; }
   shape       = three_node_triangle
   volume_set  = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
   surfaces    = { 1, 1, 1, 2, 4, 9 ;
                   2, 2, 3, 4, 6, 12 ;
                   3, 3, 5, 6, 8, 15 ; }
   shape       = four_node_quad
   volume_set  = "prisms"
```

Then, a single `ELEMENT_BOUNDARY_CONDITION` command is defined that contains the tri and quad faces as follows:

```
ELEMENT_BOUNDARY_CONDITION ("constant BC on heated wall") {
   surface_sets       = {"tri_faces", "quad_faces"}
   ...
}
```

The list of surface sets can also be placed in a file, such as `surface_sets.srfst`:

```
tri faces
quad faces
```

and read using:

```
ELEMENT_BOUNDARY_CONDITION ("constant BC on heated wall") {
   surface_sets       = Read("surface_sets.srfst")
   ...
}
```

The mixed topology version of the `ELEMENT_BOUNDARY_CONDITION` command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *`surface_sets`* and *`surfaces`* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

The variable to which the boundary condition applies is given by *`variable`*. If the problem does not solve for a field associated with this variable, the boundary condition is simply ignored.

The element boundary conditions are applied at the quadrature points of the surfaces. The quadrature rule is inherited from the parent element set of the surface.

A constant boundary condition applies the same boundary condition for all quadrature points of the surfaces, as in the above example.

A zero boundary condition is a short-hand form of the constant type, with a constant value of zero. For example,

```
ELEMENT_BOUNDARY_CONDITION( "constant BC on outflow pressure" ) {
 surface_sets = {"tri_faces", "quad_faces"}
 variable     = pressure
 type         = zero
}
```

When the free element boundary condition type is specified, the values of the boundary condition are computed from the current solution. For example,

```
ELEMENT_BOUNDARY_CONDITION( "outflow traction" ) {
 surface_sets = {"tri_faces", "quad_faces"}
 variable     = tangential_traction
 type         = free
}
```

On boundary surfaces where no element boundary condition is specified for the *mass_flux* variable, one is added of type free. The same is done for the pressure variable. For all other variables nothing is added (which is the same as adding an element boundary condition of type zero).

> **Note:** Outflow and free surfaces normally require a pressure boundary condition of type zero (or constant).

These must be specified. The free type is not available for convective and radiation type variables.

The outflow and inflow boundary condition types are a special form of the free type which may only be specified for the *mass_flux* variable. These boundary conditions indicate that the boundary is an outflow or an inflow boundary, respectively. The code takes certain measures to insure that the flow goes in the proper direction, that is, to prevent reverse flows. This is a weakly enforced condition; it does not guarantee no reverse flows. However, it does improve stability in situations where reverse flow is likely but weak. For example,

```
ELEMENT_BOUNDARY_CONDITION( "outflow mass flux" ) {
 surface_sets = {"tri_faces", "quad_faces"}
 variable     = mass_flux
 type         = outflow
}
```

A per_surface boundary condition specifies a different value for each surface. For example,

```
ELEMENT_BOUNDARY_CONDITION( "per surface BC on heated wall" ) {
 shape          = three_node_triangle
 element_set    = "flow elements"
 surfaces       = { 4, 41, 2, 5, 6 ;
                    5, 51, 5, 6, 3 ; }
 variable       = heat_flux
 type           = per_surface
 surface_values = { 41, 2 ;
                    51, 4 ; }
}
```

For scalar variables, the *surface_values* parameter is a two-column array corresponding to the surface number and boundary condition values. For vector variables (currently only *tangential_traction*), *surface_values* is a four-column array corresponding to the surface number and the xyz components of the boundary condition. The surface numbers must match those given by *surfaces*. No surface may be missing.

The surface values may be read from a file. For the above example, the boundary conditions may be placed in a file, such as `heated_wall.heat`:

```
41 2
51 4
```

and read by:

```
ELEMENT_BOUNDARY_CONDITION( "per surface BC on heated wall" ) {
 shape          = three_node_triangle
 element_set    = "flow elements"
 surfaces       = Read( "heated_wall.ebc" )
 variable       = heat_flux
 type           = per_surface
```

```
 surface_values = Read( "heated_wall.heat" )
}
```

The per_surface option is currently not supported with mixed topology input.

The element boundary conditions of types piecewise_linear and cubic_spline may be used to define boundary condition values as a function of a single independent variable. For example,

```
ELEMENT_BOUNDARY_CONDITION( "curve fit BC on convective heat wall" ) {
 surface_sets          = {"tri_faces", "quad_faces"}
 variable              = convective_heat_flux
 type                  = piecewise_linear
 curve_fit_values      = { 0, 0.0 ;
                           10, 1.5 ; }
 curve_fit_variable    = x_coordinate
 reference_temperature = 25
}
```

defines a convective heat transfer boundary condition as a function of the x-coordinates. In this example the convective heat coefficient varies linearly from h = 0 at x = 0 to h = 1.5 at x = 10. For scalar variables the *curve_fit_values* parameter is a two-column array corresponding to the independent variable and the boundary condition values. For vector variables, currently only *tangential_traction*, *curve_fit_values* is a four-column array corresponding to the independent variable and the xyz components of the boundary condition. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *curve_fit_variable* falls outside of the curve fit limits. The curve fit variable nornal_velocity is positive when the flow goes out of the parent element.

The *curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, such as `conv_wall.fit`:

```
0 0.0
10 1.5
```

and read by:

```
ELEMENT_BOUNDARY_CONDITION( "curve fit BC on convective heat wall" ) {
 surface_sets          = {"tri_faces", "quad_faces"}
 variable              = convective_heat_flux
 type                  = piecewise_linear
 curve_fit_values      = Read( "conv_wall.fit" )
 curve_fit_variable    = x_coordinate
 reference_temperature = 25
}
```

An element boundary condition of type user_function may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, a stagnation pressure boundary condition equivalent to the *stagnation_pressure* variable may be implemented through an element boundary condition on pressure. The input command may be given by:

```
ELEMENT_BOUNDARY_CONDITION( "UDF BC on inflow pressure" ) {
 surface_sets  = {"tri_faces", "quad_faces"}
 variable      = pressure
 type          = user_function
```

```
 user_function = "usrElementBcExample"
 user_values   = { 100,      # pressure
                   1.225 } # density
}
```

where the user-defined function "usrElementBcExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrElementBcExample ) ; /* function prototype */
Void usrElementBcExample (
    UdfHd      udfHd,                          /* Opaque handle for accessing data */
    Real*      outVec,                         /* Output vector */
    Integer    nItems,                         /* Number of BC surfaces */
    Integer    vecDim                          /* = 1 */
) {
    Integer    surf ;                          /* a surface counter */
    Real       dens ;                          /* density */
    Real       stagPres ;                      /* stagnation pressure */
    Real       velSqr ;                        /* velocity square */
    Real*      usrVals ;                       /* user values */
    Real*      vel ;                           /* velocity */
    Real*      xVel ;                          /* x-component of velocity */
    Real*      yVel ;                          /* y-component of velocity */
    Real*      zVel ;                          /* z-component of velocity */
    udfCheckNumUsrVals( udfHd, 2 ) ;           /* check for error */
    usrVals    = udfGetUsrVals( udfHd ) ;      /* get the user vals */
    stagPres   = usrVals[0] ;                  /* get BC value */
    dens       = usrVals[1] ;                  /* get density */
    vel        = udfGetEbcData( udfHd, UDF_EBC_VELOCITY ) ;  /* get the velocity */
    xVel       = &vel[0*nItems] ;              /* localize x-vel. */
    yVel       = &vel[1*nItems] ;              /* localize y-vel. */
    zVel       = &vel[2*nItems] ;              /* localize z-vel. */
    for ( surf = 0 ; surf < nItems ; surf++ ) {
        velSqr = xVel[surf] * xVel[surf]
               + yVel[surf] * yVel[surf]
               + zVel[surf] * zVel[surf] ;
    outVec[surf] = stagPres - 0.5 * dens * velSqr ;
    }
} /* end of usrElementBcExample() */
```

For scalar variables, the dimension of the returned boundary condition vector, *outVec*, is the number of surfaces.

For vector variables (currently only *tangential_traction*), the dimension is the number of surfaces times three, with the surface dimension being "faster" (as it is for the velocity vector in the example).

For the *tangential_traction* variable, the three components specified are in the global xyz coordinate system. The component normal to the surface is removed before the boundary condition is imposed. Thus, there will be no conflict if a pressure (normal component of traction) element boundary condition is also imposed on the same surfaces.

The *multiplier_function* parameter may be used to uniformly scale the element boundary condition values. The value of this parameter refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file. For *variable*=stagnation_pressure the multiplier function scales only the pressure part of the stagnation pressure. For example, to impose a uniform heat flux as a function of time, the following commands may be issued:

```
ELEMENT_BOUNDARY_CONDITION( "constant BC on heat flux" ) {
```

```
    surface_sets        = {"tri_faces", "quad_faces"}
    variable            = heat_flux
    type                = constant
    constant_value      = 2.5
    multiplier_function = "time varying"
}
MULTIPLIER_FUNCTION( "time varying" ) {
    type                = piecewise_linear
    curve_fit_values    = { 0, 0.0 ;
                            10, 1.0 ;
                            20, 0.5 ;
                            40, 0.7 ;
                            80, 1.2 ; }
    curve_fit_variable = time
}
```

For convective and radiation variables the value of the element boundary condition is the value of a coefficient, not the value of a flux variable. The convective and radiation heat fluxes are given by:

$$q_{conv} = h(T - T_{ref})$$

$$q_{rad} = h[(T + T_{off})^4 - (T_{ref} + T_{off})^4] \quad (150)$$

where h is the value of the element boundary condition; T is the temperature; T_{ref} is the reference temperature, given by *reference_temperature*; and T_{off} is the offset to convert to an absolute temperature, given by the *absolute_temperature_offset* parameter of the EQUATION command.

For a radiation heat flux boundary condition, the coefficient may be calculated from $h = \varepsilon\sigma$, where ε is the grey-body emissivity/ absorptivity and σ is the Stefan-Boltzmann constant. A description of the last two quantities may be found in the RADIATION command. However, the radiation heat flux is specified independently of the radiation equation and the RADIATION, RADIATION_SURFACE, and EMISSIVITY_MODEL commands, as well as the SOLAR_RADIATION and related commands. The convective species fluxes are defined similarly to the convective heat flux, except that *reference_species* is used instead of *reference_temperature*.

The *reference_temperature_multiplier_function* parameter may be used to scale the reference temperature. For example,

```
ELEMENT_BOUNDARY_CONDITION( "time dependent outside temperature" ) {
    surface_sets                              = {"tri_faces", "quad_faces"}
    variable                                  = convective_heat_flux
    type                                      = constant
    constant_value                            = 12.5
    reference_temperature                     = 1
    reference_temperature_multiplier_function = "outside temperature"
}
MULTIPLIER_FUNCTION( "outside temperature" ) {
    type                                      = cubic_spline
    curve_fit_values                          = { 0*3600, 295 ;
                                                  12*3600, 312 ;
                                                  24*3600, 295 ; }
    curve_fit_variable                        = time
}
```

Similarly, *reference_species_multiplier_function* may be used to scale the reference species.

Reflections of waves from inflow and outflow boundaries can be reduced by setting *non_reflecting_factor* to a positive value. This parameter applies to inflow types with *inflow_type* = mass_flux and to outflow types. For inflows only the mass flux element boundary condition is modified and for outflows only the pressure element boundary condition is modified. It is ignored for all other cases. Also, there is no effect for the constant density model unless a positive *isothermal_compressibility* value is specified.

Setting *non_reflecting_factor* = 0 has no effect; the given element boundary condition is imposed. Using a value of one gives the full effect; waves pass through the boundary without reflection but at the cost of the variable drifting away from its specified value. Other values yield a blend between these effects. Values greater than one for *non_reflecting_factor* may lead to instabilities for the mass flux condition. If running average fields are defined, that is, *running_average* = on in the EQUATION command, then an enhanced non-reflecting boundary condition is used, leading to less reflection for the same value of *non_reflecting_factor*. The *non_reflecting_bc_running_average_field* needs to be turned on for this option.

Note: When *flow* = compressible_navier_stokes, the *running_average* option does not need to be turned on; the running average files are automatically used in non- reflecting boundary conditions.

Pressure loss at inflow or outflow boundaries may be modeled through the *pressure_loss_factor* parameter. For *variable*=pressure, the following term is added to the pressure boundary condition:

$$\frac{1}{2}\,\mathrm{sgn}(\boldsymbol{u}\bullet\boldsymbol{n})k\rho(\boldsymbol{u}\bullet\boldsymbol{n})^2 \qquad (151)$$

where k is given by *pressure_loss_factor*; ρ is the density; **u** is the velocity; **n** is the outward-pointing normal to the surface; and sgn(**u**•**n**) is - for inflow boundaries and +1 for outflow boundaries. If *multiplier_function* is given, it is applied to the pressure variable before this term is added. The pressure loss factor may be scaled with *pressure_loss_factor_multiplier_function*. For example,

```
ELEMENT_BOUNDARY_CONDITION( "outflow" ) {
    ...
    variable                                = pressure
    pressure                                = 0
    pressure_loss_factor                    = 100
    pressure_loss_factor_multiplier_function = "pLoss_TMF"
    ...
}
MULTIPLIER_FUNCTION( "pLoss_TMF" ) {
    type                                    = piecewise_linear
    curve_fit_values                        = { 1, 1 ;
                                                10 , 0 }
    curve_fit_variable                      = time_step
}
```

The hydrostatic pressure term may be added to pressure and stagnation pressure element boundary conditions by setting *hydrostatic_pressure*=on. This term is given by

$$\rho(\boldsymbol{x}-\boldsymbol{x}_0)\bullet\boldsymbol{g} \qquad (152)$$

where ρ is the density; **x** is the current coordinate vector; **x**0 is any coordinate vector where the hydrostatic pressure is zero, given by *hydrostatic_pressure_origin*; and **g** is the gravity vector given by *body_force* in the parent ELEMENT_SET command or is the *gravity_vector* given by *gravitational_acceleration* in EQUATION. If *variable* is not *pressure* or *stagnation_pressure*, then *hydrostatic_pressure* has no effect. If *multiplier_function* is given, it is applied before this term is added. If gravitational force is modeled using the GRAVITY command, then *hydrostatic_pressure*=on should be set for most pressure or stagnation pressure boundary conditions in order to properly account for the hydrostatic pressure. This is most commonly needed on outflow boundaries. For free surface pressure conditions *hydrostatic_pressure* is normally set to off in order to ensure that the imposed pressure is constant along the entire surface. If this pressure or the nominal height of the free surface is nonzero, then *hydrostatic_pressure_origin* may be adjusted for any other pressure boundary condition so that its evaluation at the nominal surface height equals the pressure imposed on the free surface. That is, $p_{\mathrm{BC}} + \rho(x_{\mathrm{NOM}} - x_0) \bullet g = p_{\mathrm{FS}}$ where x_{NOM} is a point on the nominal free surface, p_{BC} is the given pressure in an ELEMENT_BOUNDARY_CONDITION command with *hydrostatic_pressure*=on, and p_{FS} is the given pressure on the free surface. An example of this adjustment is given in SIMPLE_BOUNDARY_CONDITION.

In some circumstances it is necessary to "turn off" or "deactivate" a previously-defined element boundary condition. This is accomplished by setting the active flag through *active_type*. A value of all is the default and means that all surfaces in the set are active (active flag = 1) and will have the boundary condition imposed on them. A value of none means that no surface is active (active flag = 0).

A no_interface active type is used to automatically turn on and off element boundary conditions in the presence of interfaces. The boundary condition is applied (active flag = 1) if the surface does not belong to an INTERFACE_SURFACE set or if it does not find a contact surface of the appropriate medium. That is, each surface will fall into one of five possible cases:

- The surface is not in any INTERFACE_SURFACE set; active flag = 1.
- The surface is in an INTERFACE_SURFACE set, but it does not find a contact surface; active flag = 1.
- The surface finds a contact surface of fluid medium; active flag = 0.
- The surface finds a contact surface of solid/shell medium, and *variable* is defined on the solid/shell side, currently only the heat flux variables,; active flag = 0.
- The surface finds a contact surface of solid/shell medium, and *variable* is not defined on the solid/shell side, that is, it is not one of the heat flux variables,; active flag = 1.

ELEMENT_OUTPUT

Specifies output parameters for a set of elements.

Type

AcuSolve Command

Syntax

`ELEMENT_OUTPUT("name") {parameters...}`

Qualifier

User-given name.

Parameters

`element_set` or *`elem_set`* *(string) [no default]*
User-given name of the parent element set. All *`volume_sets`* within the given *`element_set`* will be considered. Used with *`type`* = subset and *`type`* = all.

`volume_sets` or *`vol_sets`* *(list) [{}]*
User-given list of volume sets. Used with *`type`*=subset and *`type`* = all.

`type` *(enumerated) [= subset]*
Type of the user-given elements.

subset	User-given list of elements will be output. Requires *`elements`* parameter.
all	All available elements within user-given volume sets and element sets will be output.

`elements` *(array) [no default]*
List of element numbers.

`integrated_output_frequency` or *`intg_freq`* *(integer) >=0 [=1]*
Time step frequency at which to output averaged element quantities. If zero, this option is ignored.

`integrated_output_time_interval` or *`intg_intv`* *(real) >=0 [=0]*
Time frequency at which to output averaged element quantities. If zero, this option is ignored.

`statistics_output_frequency` or *`stat_freq`* *(integer) >=0 [=1]*
Time step frequency at which to output statistics of element quantities. If zero, this option is ignored.

`statistics_output_time_interval` or *`stat_intv`* *(real) >=0 [=0]*
Time frequency at which to output statistics of element quantities. If zero, this option is ignored.

`num_user_output` *(real) >=0 [=0]*
Number of user-defined quantities to be included with averaged element quantities. If zero, this option is ignored. If greater than zero, requires *`user_function`*.

user_function or *user* *(string) [no default]*
: Name of the user-defined function that returns *num_user_output* element quantities. Used with *num_user_output* > 0.

user_values *(array) [={}]*
: Array of values to be passed to the user-defined function. Used with *num_user_output* > 0.

user_strings *(list) [={}]*
: Array of strings to be passed to the user-defined function. Used with *num_user_output* > 0.

Description

This command specifies a set of elements whose quantities are to be averaged by integrating through the volume of the elements. This command then outputs the averaged quantities to disk.

The ELEMENT_OUTPUT command can refer to a subset of elements within a given element set, or to the entire collection of elements. For example,

```
ELEMENT_SET( "flow" ) {
    shape                           = eight_node_brick
    elements                        = { 1, 1, 2, 4, 3, 11, 12, 14, 13 ;
                                        2, 3, 4, 6, 5, 13, 14, 16, 15 ;
                                        ... }
    ...
}
ELEMENT_OUTPUT( "sample" ) {
    element_set                     = "flow"
    type                            = subset
    elements                        = { 1 ;
                                        2 ; }
    integrated_output_frequency     = 5
    integrated_output_time_interval = 60.0
    num_user_output                 = 0
}
```

specifies that the volume-averaged quantities over elements one and two are to be output every five time steps, every 60 units of run time and at the last time step. The elements are numbered with respect to the element set "flow". When *type* = subset, a check is performed to ensure that the IDs are not duplicated within the *elements* parameter. This condition will cause an error to be issued.

To output all of the elements in the "sample" element set, the following command can be issued:

```
ELEMENT_OUTPUT( "sample" ) {
 element_set                     = "flow"
 type                            = all
 integrated_output_frequency     = 5
 integrated_output_time_interval = 60.0
 num_user_output                 = 0
}
```

In this case, the *elements* parameter is not necessary. The element IDs are taken directly from the element set itself.

There are two main forms of this command. The legacy version, or single topology version, of the command relies on the use of the *element_set* parameter to define the volume elements. When using this form of the command, all elements within a given set must have the same shape. The alternate, or

mixed topology version, of this command relies on the *volume_sets* parameter to define the elements. For example,

```
VOLUME_SET( "tetrahedra" ) {
    elements                          = { 1, 1, 2, 4, 3 ;
                                          2, 3, 4, 6, 5 ;
                                          3, 5, 6, 8, 7 ; }
    shape                             = four_node_tet
}
VOLUME_SET( "prisms" ) {
    elements                          = { 1, 1, 2, 4, 9, 10, 11 ;
                                          2, 3, 4, 6, 12, 13, 14 ;
                                          3, 5, 6, 8, 15, 16, 17 ; }
    shape                             = six_node_wedge
}
ELEMENT_OUTPUT( "fluid elements" ) {
    volume_sets                       = {"tetrahedra","prisms"}
    type                              = all
    integrated_output_frequency       = 5
    ...
}
```

can be used to define a single element output command that contains both tetrahedra and prism elements. All elements defined by the volume sets are output. The *type* = subset option can also be used with the mixed topology form of this command to select only a subset of these elements. Under these circumstances, the element IDs specified by the *elements* parameter must be uniquely defined within the volume sets that are specified or an error will be issued.

The mixed topology version of the ELEMENT_OUTPUT command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *volume_sets* and *element_set* parameters are provided in the same instance of the command, the full collection of volume elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

The element numbers may be read from a file for both the single and mixed topology forms of this command. For the above example, the element numbers may be placed in a file, such as sample.elm:

```
1
2
```

and read by:

```
ELEMENT_OUTPUT( "sample" ) {
   ...
   elements                         = Read( "sample.elm" )
   integrated_output_frequency      = 5
   integrated_output_time_interval  = 60.0
   num_user_output                  = 0
}
```

User-defined quantities may be included for output by specifying a non-zero *num_user_output*. This quantity corresponds to the vector dimension of the user-defined function given by *user_function*. See the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

Once the averaged element quantities have been written to disk, they can be translated to other formats using the AcuTrans program and other post-processing modules; see the *AcuSolve Programs Reference Manual* for details.

If either *integrated_output_frequency* or *integrated_output_time_interval* is non-zero, the averaged element quantities will be output at the end of the run. If both are zero, no data are written to disk.

Run times may not coincide with *integrated_output_time_interval*. In this case, the data are output for every time step which passes through a multiple of *output_time_interval*.

ELEMENT_SET

Specifies a set of elements.

Type

AcuSolve Command

Syntax

ELEMENT_SET ("name") {parameters}

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Topology of the elements in the set. Used only when the elements are specified using the elements option.

four_node_tet* or *tet4	Four-node tetrahedron.
five_node_pyramid* or *pyramid5	Five-node pyramid.
six_node_wedge* or *wedge6	Six-node wedge (prism).
eight_node_brick* or *hex8	Eight-node brick (hexahedron).
ten_node_tet* or *tet10	Ten-node tetrahedron.

elements (array) [no default]

A set of element connectivity.

volume_sets (list) [={}]

List of volume set names (strings) to use in this element set. When using this option, the connectivity and shape of the elements are provided by the volume set container and it is unnecessary to specify the shape and elements options directly to the ELEMENT_SET command. In general, this option is used in place of directly specifying the *shape* and *elements* parameters in the ELEMENT_SET command. In the event that both the *volume_sets* and *elements* parameters are provided, the full collection of elements is read and a warning message is issued. The *volume_sets* option is the preferred method to specify the elements for the element set. This option provides support for mixed element topologies within a given element set and simplifies pre-processing and post-processing.

medium (enumerated) [=fluid]

Type of physics being solved.

none	No medium. Deactivates element set.
fluid	Fluid medium. Requires *material_model*.

multi_field	Multi-fluid medium. Requires *multi_field_model*.
solid	Solid medium. Solve only for temperature field. Requires *material_model*.
shell	Shell medium. Solve only for temperature field. Requires *num_shell_layers*, *shell_material_models*, *shell_thickness_type* and *shell_thicknesses*.
design	Design medium. Needed for topology optimization design field.

quadrature (enumerated) [=full]
Quadrature rule used to integrate the element contribution.

reduced	Reduced quadrature.
full	Full quadrature.
high_1	High-order quadrature.
high_2	High-order quadrature (higher than high_1).
nodal	Nodal quadrature.

material_model or *material (string) [no default]*
User-given name of the material model to be used in this element set. Used for fluid and solid media.

design_material_model (string) [no default]
User-given name of the design material model to be used for topology optimization. Must be defined in a separate MATERIAL_MODEL command.

design_variables_field (string) [no default]
User-given name of the design variables field defined in topology optimization. Must be defined in a unique DESIGN_VARIABLES_FIELD command.

multi_field_model (string) [no default]
User-given name of the multi field model to be used in this element set. Used only for multi_field medial.

body_force (string) (string)
User-given name of the volumetric body force to be applied to this element set. If none, no body force is applied.

element_volume_heat_sources (array) [={}]
Heat source per unit volume given on a per-element basis in a two-column array. This option is not supported with mixed topology element sets.

element_volume_momentum_sources (array) [={}]
Momentum source per unit volume given on a per-element basis in a four-column array. This option is not supported with mixed topology element sets.

reference_frame *(string) [=none]*
User-given name of the reference frame for this element set. If none, the fixed reference frame is used.

mesh_motion *(string) [=none]*
User-given name of the MESH_MOTION command for specifying mesh displacement boundary conditions on all the nodes in this element set. If none, no boundary conditions are imposed.

mesh_motion_precedence *(integer) >=0 [=0]*
Precedence of the given mesh motion. When specifying the mesh motion on an element set level, all nodes within the set, including the boundary nodes, inherit the mesh motion. For applications where the surface nodes of the volume have a conflicting mesh motion constraint that is specified through the SIMPLE_BOUNDARY_CONDITION or NODAL_BOUNDARY_CONDITION command, the *mesh_motion_precedence* is used to resolve the conflict. The constraint with the highest precedence takes priority. The default value of *mesh_motion_precedence* is equal to zero, indicating that SIMPLE_BOUNDARY_CONDITION and NODAL_BOUNDARY_CONDITION constraints will take priority. Their default precedence is 1. In the case of equal precedence values, the SIMPLE_BOUNDARY_CONDITION and NODAL_BOUNDARY_CONDITION constraints will override those set on the ELEMENT_SET level.

num_shell_layers *or* *layers* *(integer)>0 [=1]*
Number of layers in the shell. Used for shell medium only.

shell_material_models *(list) [={}]*
List of material model names (strings) for the shell layers. Number of entries in the list must equal *num_shell_layers*. Used for shell medium only.

shell_thickness_type *(enumerated) [=constant]*
Type of shell thickness. Used for shell medium only.

constant or const	Constant shell thickness.

shell_thicknesses *or* *thick* *(array) [={}]*
Thicknesses of the shell layers. Number of entries in the array must equal *num_shell_layers*. Used with constant shell thickness type and for shell medium only.

viscous_heating *(boolean) [=off]*
Flag specifying whether to add the viscous heating term in the temperature equation.

compression_heating *(boolean) [=off]*
Flag specifying whether to add the compression heating term in the temperature equation.

residual_control *(boolean) [=on]*
Flag specifying whether to use nonlinear residual control to further stabilize the algorithm.

oscillation_control *(boolean) [=on]*
Flag specifying whether to use oscillation control to further stabilize the temperature, species, and turbulence equations.

mesh_distortion_correction_factor *(real) >=0 [=0]*
Amount of correction to be applied to the element Jacobian of highly distorted elements.

mesh_distortion_tolerance *(real) >=0 [=0]*
: Determines if correction needs to be applied to the element Jacobian for highly distorted elements.

auxiliary_elements *(array) [no default]*
: Element connectivity for auxiliary elements. Used with element user-defined functions. This option is not supported with mixed topology element sets.

auxiliary_material_model *(string) [no default]*
: User-given name of the material model to be used for the auxiliary elements. This option is not supported with mixed topology element sets.

auxiliary_reference_frame *(string) [=none]*
: User-given name of the reference frame to be used for the auxiliary elements. This option is not supported with mixed topology element sets.

turbulence_suppression *(boolean) [=off]*
: Flag specifying whether to suppress turbulence production and ignore the contribution of eddy viscosity in a turbulence model.

porous_media_region *(boolean) [=off]*
: Flag specifying whether to activate the porous medium in the carrier field element set. The *porosity_model* must be defined in the MATERIAL_MODEL for carrier field. This flag is available when *edem_coupling* under EQUATION is set to on and *porous_media_velocity_type* under EQUATION is set to physical.

Description

This command specifies a set of elements in the problem. All elements in a set have the same medium, material model, body force, and so on. The elements are grouped in sets to improve efficiency of memory and CPU usage. There are two main forms of this command. The legacy version, or single topology version, of the command relies on the use of the *elements* parameter to define the elements. When using this form of the command, all elements within a given element set must have the same shape, and it is necessary to include both the *elements* and *shape* parameters in the command. For example,

```
ELEMENT_SET( "air flow" ) {
   elements            = { 1001, 1, 2, 4, 3, 11, 12, 14, 13 ;
                           1002, 3, 4, 6, 5, 13, 14, 16, 15 ;
                           1003, 5, 6, 8, 7, 15, 16, 18, 17 ; }
   medium              = fluid
   shape               = eight_node_brick
   quadrature          = full
   material_model      = "air at standard temperature"
   reference_frame     = none
   mesh_motion         = none
   body_force          = none
   residual_control    = on
   oscillation_control = on
}
```

specifies an element set with three eight-node brick fluid elements.

The mixed topology form of the ELEMENT_SET command provides a more powerful and flexible mechanism for defining the elements. Using this form of the command, it is possible to define an

element set that contains different element shapes. This is accomplished through the use of the *volume_sets* parameter. The elements are first read into AcuPrep using the VOLUME_SET command, and are then referred to by the ELEMENT_SET command. For example, a collection of tetrahedra and prism elements can be defined using the following VOLUME_SET commands.

```
VOLUME_SET( "tetrahedra" ) {
   elements  = { 1, 1, 2, 4, 3 ;
                 2, 3, 4, 6, 5 ;
                 3, 5, 6, 8, 7 ; }
   shape     = four_node_tet
}
VOLUME_SET( "prisms" ) {
   elements  = { 1, 1, 2, 4,  9, 10, 11 ;
                 2, 3, 4, 6, 12, 13, 14 ;
                 3, 5, 6, 8, 15, 16, 17 ; }
   shape     = six_node_wedge
}
```

Then, a single element set can be defined that contains the tetrahedra and prism elements as follows:

```
ELEMENT_SET( "fluid elements" ) {
   volume_sets      = {"tetrahedra","prisms"}
   medium           = fluid
   quadrature       = full
   material_model   = "air at standard temperature"
   reference_frame  = none
   mesh_motion      = none
   body_force       = none
   residual_control = on
oscillation_control = on
}
```

The list of volume sets can also be placed in a file, such as volume_sets.vst:

```
tetrahedra
prisms
```

and read using:

```
ELEMENT_SET( "fluid elements" ) {
   volume_sets       = Read("volume_sets.vst")
   medium            = fluid
   quadrature        = full
   material_model    = "air at standard temperature"
   reference_frame   = none
   mesh_motion       = none
   body_force        = none
   residual_control  = on
oscillation_control  = on
}
```

The mixed topology version of the ELEMENT_SET command is preferred. This version provides support for multiple element topologies within a given element set and simplifies pre-processing and post-processing. In the event that both the *volume_sets* and *elements* parameters are provided in the same instance of ELEMENT_SET, the full collection of elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

In the examples above, full quadrature rule is used. The "air at standard temperature" material model is applied. The elements are in the fixed reference frame and not subject to mesh motion. No body force is added. Both the residual and oscillation controls are turned on.

The *medium* parameter specifies whether the element set is of type fluid, solid, shell, or none. A fluid element set solves for all equations in the system. A solid or shell element set solves only the temperature and mesh displacement equations, while all other equations, such as flow, turbulence, and species, are ignored. Therefore, appropriate boundary conditions at the interface of the fluid and solid/shell elements must be imposed. A type of none effectively deactivates the element set and all of its associated surfaces; the elements do not contribute to any equation. The nodes may still be moved through the use of the *mesh_motion* parameter. The element set may be reactivated by changing its type on restart.

For all forms of the command, the *material_model* parameter is mandatory. When defining the elements using the single topology form of the command, the *elements* and *shape* parameters are also mandatory. *shape* specifies the shape (topology) of the elements in the set. *elements* provide the element connectivity. This parameter is a multi column array, with one row per element. The first column corresponds to the element numbers, which are unique numbers within this element set. Negative values are acceptable. The remaining columns are the element nodes. The number of element nodes and their order must match the shape of the element set. The element node numbers must be valid numbers, as given by the COORDINATE command.

A four-node tetrahedron element requires four nodes ordered with the local node numbering order shown below:

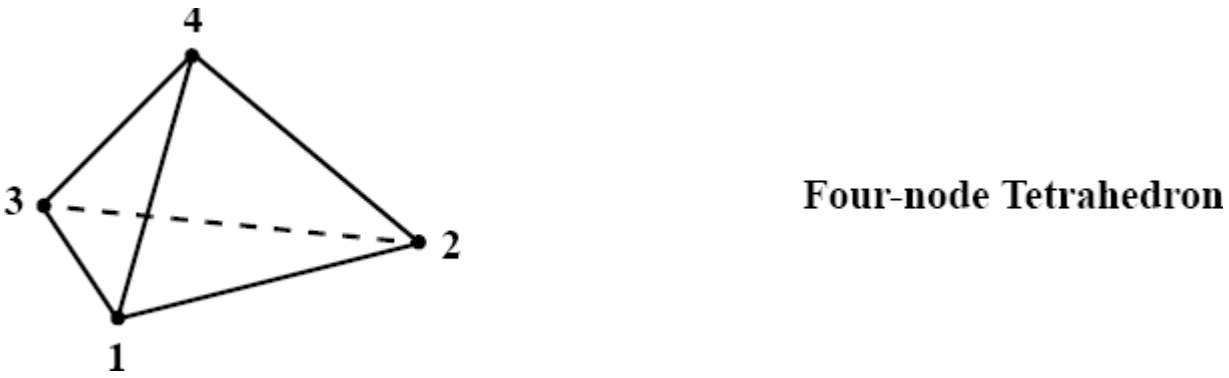


Figure 20: Four-node Tetrahedron Element

A five-node pyramid element requires five nodes ordered with the local node numbering order shown below:

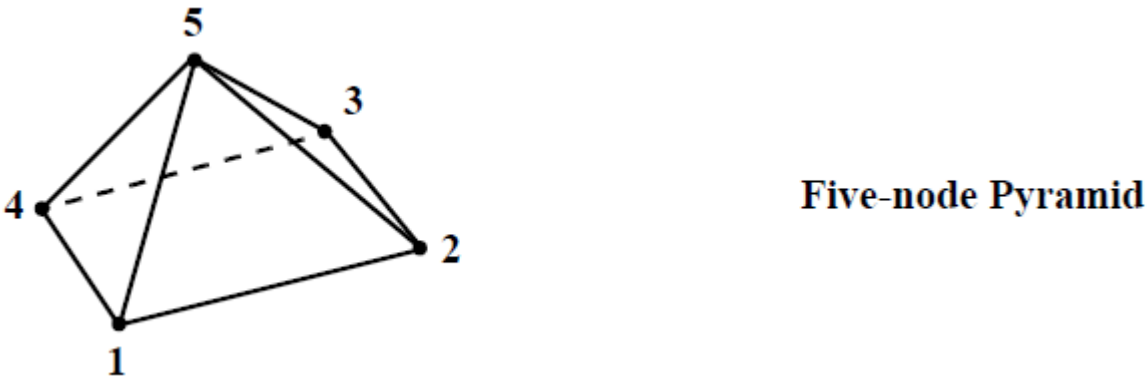


Figure 21: Five-node Pyramid Element

A six-node wedge (or prism) element requires six nodes ordered with the local node numbering order shown below:

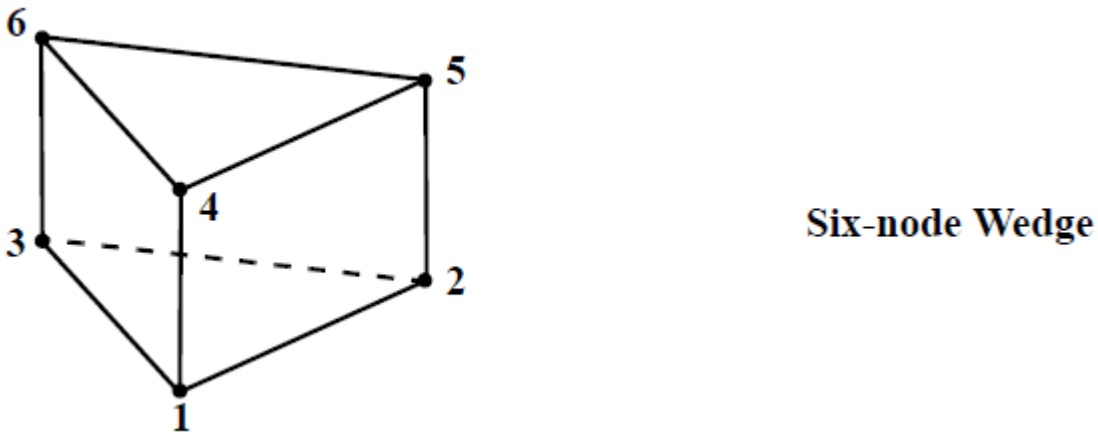
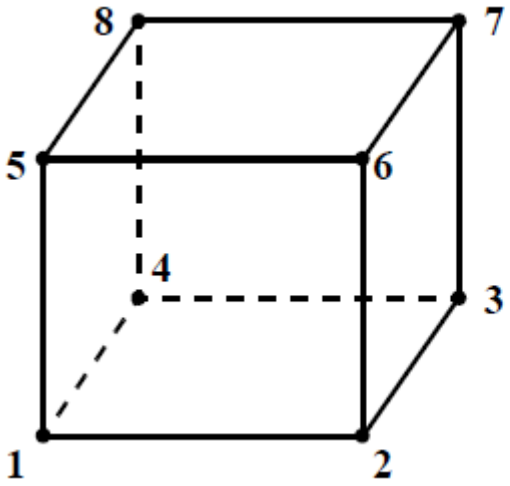


Figure 22: Six-node Wedge Element

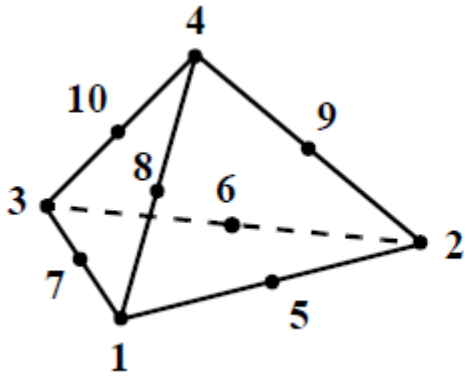
An eight-node brick (or hexahedron) element requires eight nodes ordered with the local node numbering order shown below:



Eight-node Brick

Figure 23: Eight-node Brick Element

A ten-node tetrahedron element requires ten nodes ordered with the local node numbering order shown below:



Ten-node Tetrahedron

Figure 24: Ten-node Tetrahedron Element

The element nodes may be read from a file. For the above example, the element nodes may be placed in a file, such as `air.cnn`:

```
1001      1   2    4   3   11   12    14     13
1002      3   4    6   5   13   14    16     15
1003      5   6    8   7   15   16    18     17
```

and read by:

```
ELEMENT_SET( "air flow" ) {
   elements        = Read( "air.cnn" )
   medium          = fluid
   shape           = eight_node_brick
   material_model  = "air at standard temperature"
}
```

The *quadrature* parameter defines the quadrature rule used to integrate element contributions. The actual quadrature rule used depends on the element shape as given in the table below:

Shape	**Quadrature rule**				
	full	reduced	nodal	high_1	high_2
four_node_tet	4-point	1-point	4-point nodal	14-point	20-point
five_node_pyram	5-point	1-point	5- point nodal	13-point	13-point
six_node_wedge	3x2	2-point	6-point nodal	6x3	7x4
eight_node_brick	2x2x2	2x2x 2	8-point nodal	3x3x3	4x4x 4
ten_node_tet	8-point	4-point	10-point nodal	20-point	35-point

All quadrature rules except for *nodal* satisfy the following properties: are Gauss; are inside the elements; have positive quadrature weights; are node-order independent; and use the minimal number of the points for a given integration order. Appropriate surface and edge quadrature rules corresponding to these interior rules are used automatically for those surfaces and edges associated with a given element set. For example, the first high-order quadrature rule can be defined as follows:

```
ELEMENT_SET( "HIGH QUAD 1" )
   ...
   quadrature      = high_1
   ...
}
```

The *material_model* parameter provides the material information, such as density, dynamic viscosity and conductivity to the element set for fluid and solid media. This parameter accepts the user-given name of a MATERIAL_MODEL command. For example,

```
ELEMENT_SET( "air flow" ) {
   material_model       = "air at standard temperature"
   ...
   }
   MATERIAL_MODEL( "air at standard temperature" ) {
   ...
}
```

By default, all element sets are defined with respect to the fixed, or "laboratory", reference frame. If the REFERENCE_FRAME command is used to define the rotational body forces, this is recommended, as opposed to using the ROTATION_FORCE command, then *reference_frame* is used to define the reference frame for the element set. In this case the *body_force* command referenced by the *body_force* parameter (see below) should not include the *rotation_force* parameter, since rotational forces have already been included. See the REFERENCE_FRAME command for more details and an example.

The *mesh_motion* parameter is provided as a shortcut way to impose nodal boundary conditions on all three components of the mesh displacement. The parameter refers to a MESH_MOTION command where the motion is defined. Specifying *mesh_motion* is equivalent to providing three NODAL_BOUNDARY_CONDITION commands on the components of the mesh displacement, with each command referring to all the nodes in the element set, setting *precedence*=0, and including the

same *mesh_motion* parameter. The low precedence value means that specifying *mesh_motion* in a SIMPLE_BOUNDARY_CONDITION command overrides the boundary conditions set here. If *mesh*=eulerian in the EQUATION command or *mesh_motion*=none then this parameter has no effect. For example, an element set that rotates with a fan may be defined by:

```
ELEMENT_SET( "rotating fan" ) {
   mesh_motion       = "rotating fan"
   ...
}
MESH_MOTION( "rotating fan" ) {
   type              = rotation
   rotation_center   = { 0, 0, 0 }
   angular_velocity  = { 0, 3, 0 }
}
```

Note: The important difference between *reference_frame* and *mesh_motion*: the former introduces an accelerating reference frame and modifies the velocity field while the latter moves the mesh through the reference frame. For complex problems both may be used simultaneously, although great care needs to be exercised.

The *body_force* parameter may be used to add volumetric body forces, such as gravity and heat source, to the element set. This parameter accepts the user-given name of the BODY_FORCE command. For example,

```
ELEMENT_SET( "air flow" ) {
   body_force        = "added heat"
   ...
}
BODY_FORCE( "added heat" ) {
   mass_heat_source  = "my heat source"
}
```

A volumetric heat source may be given on a per-element basis with the *element_volume_heat_sources* parameter. For example,

```
ELEMENT_SET( "air flow" ) {
   ...
   element_volume_heat_sources       = { 1, 1.5 ;
                                         2, 2.5 ;
                                         3, 3.5 }
}
```

where the first column contains the element numbers and the second column contains the volume heat source data.

A shell medium is used to model the energy equation in solid materials where the thinness of the geometry makes it inconvenient to use a solid medium. The shell medium supports only wedges and bricks. The triangular faces of the wedges must be on the outside surfaces of the shell. Geometrically, the shell is infinitely thin, so that the pairs of nodes in an element that are on opposite sides of the shell have the same coordinates. These nodes normally need to have different node numbers in order to support temperature and pressure differences across the shell, but in cases where no such differences exist (for example, an edge of a shell that is surrounded by fluid) they would share the same node number. Such "collapsed" elements are not supported for fluid and solid media. The first three (for

wedges) or four (for bricks) nodes of the connectivity define the first face of the shell and the last three or four nodes define the second face. Between these faces, the element is divided up into a number of layers, given by *num_shell_layers*. The first layer of the shell is adjacent to the first face, the last (*num_shell_layers*) is next to the second face and the others are placed in order in between. Each layer is assigned a material model and a thickness through the *shell_material_models* and *shell_thicknesses* parameters. Both of these must be specified and both are arrays where the layer data must be given in order. *material_model* is ignored for shells.

Two important assumptions are made for shells. First, the thicknesses parameter always applies in the direction perpendicular to the surface of the shell on an element basis. This might be inaccurate for thick shells and tightly curved surfaces. Second, the intermediate temperatures needed at the quadrature points of each layer are calculated based on a simplified analysis. A one-dimensional heat equation through the shell thickness at the element nodes is derived by neglecting thermal inertia and conduction parallel to the surface. This means that the heat flux is the same in all the layers. If each layer material model has a constant conductivity, then this simplified heat equation is solved exactly. If any of the conductivities is a function of temperature, then a two-pass procedure is used to approximately solve the resulting nonlinear system. Once the temperatures are known at the corners of all the layers in each element, then they can be interpolated to the quadrature points in the usual manner. The assumptions used to obtain the intermediate temperatures do not apply to the overall formulation for the shells. That is, thermal inertia, multi-dimensional conduction, and nonlinearities are all modeled accurately in the final solution.

The input commands for a two-layer shell may be given by:

```
MATERIAL_MODEL( "layer 1" ) {
   density_model          = "layer 1"
   specific_heat_model    = "layer 1"
   conductivity_model     = "layer 1"
}
MATERIAL_MODEL( "layer 2" ) {
   density_model          = "layer 2"
   specific_heat_model    = "layer 2"
   conductivity_model     = "layer 2"
}
ELEMENT_SET( "shells" ) {
   elements               = Read( "shell.shl.cnn" )
   shape                  = eight_node_brick
   quadrature             = full
   medium                 = shell
   num_shell_layers       = 2
   shell_thicknesses      = { 5.e-3, 2.e-3 }
   shell_material_models  = { "layer 1", "layer 2" }
}
```

The *viscous_heating* and *compression_heating* parameters control whether the corresponding terms are added to the temperature equation, which is actually solved for the enthalpy h:

$$\rho\frac{\partial h}{\partial t} + \rho\boldsymbol{u}\bullet\nabla h = \nabla\bullet\boldsymbol{q} + \rho s + \boldsymbol{\tau}:\nabla\boldsymbol{u} + \frac{Dp}{Dt} \qquad (153)$$

See the EQUATION command for a description of the individual terms. The last term is the material derivative of pressure and known as the compression heating term, and the adjacent term is the viscous heating term.

The *residual_control* and *oscillation_control* parameters add stabilization terms to the finite element formulation. The residual control option adds the so-called "discontinuity capturing" operator to all fluid equations. This operator controls unresolved internal and boundary discontinuities. The oscillation control option adds a nonlinear maximum principal operator onto the advective term of the scalar (temperature, species, and turbulence) equations, which helps to control any spurious oscillations. These two operators are turned on by the defaults and they should be rarely turned off.

When AcuSolve is performing a moving mesh simulation, a check is performed to detect the presence of bad elements, for example with negative Jacobian. If any element has Jacobian<=0, the code either exits, when *mesh*=specified or *mesh*=eulerian, or adds more non-linear iterations with an incremental update strategy, when *mesh*=ale. To relax this restriction on mesh quality, an element Jacobian correction strategy may be used to handle negative or zero Jacobian situations that often time arise during contact or with squeezing fluid boundaries. The formulation relies on the least-squares minimization of contravariant metrics. Two parameters control the algorithm: *mesh_distortion_tolerance* and *mesh_distortion_correction_factor*. The first parameter determines whether the correction needs to be applied to the element Jacobian and the second quantifies the amount of correction. Mesh distortion is checked within the ALE loop, and then AcuSolve decides how forgiving it should be to highly distorted elements. The parameters can be used as follows:

```
ELEMENT_SET( "PINCHED TUBE" ) {
   ...
   mesh_distortion_correction_factor = 1.0e-4
   mesh_distortion_tolerance         = 1.0e-2
   ...
}
```

Considering the settings shown above, AcuSolve allows element Jacobians to decrease to a level of -1.0e-4 before terminating the simulation (when *mesh*=specified, or *mesh*=eulerian) or adding non-linear iterations to the mesh displacement stagger (when *mesh*=ale).

When the element Jacobian satisfies the following relationship:

0 >= element Jacobian > -*mesh_distortion_tolerance*, and the *mesh_distortion_correction_factor* is non-zero, the element Jacobian is adjusted. Using this approach, a target Jacobian is computed that is proportional to *mesh_distortion_correction_factor*. The actual element Jacobian is then perturbed to match the target Jacobian. The larger the *mesh_distortion_correction_factor*, the further the Jacobain deviates from the true value. This has two implications. The first is that the flow solver will continue running with more highly distorted elements (a higher degree of penetrating faces). The second is that the solution in the impacted elements may deviate further from what it should be considering the actual location of the nodes. Therefore, this option should be used cautiously. It should also be noted that this approach does not actually adjust the position of the nodes. Rather, it enforces a correction on the Jacobian of the element that allows the flow solver to continue without exiting when a negative Jacobian is encountered.

Several commands, such as ELEMENT_BOUNDARY_CONDITION and SURFACE_OUTPUT, require element surfaces. These surfaces are defined with respect to the elements of an ELEMENT_SET command. In addition to their geometrical connection, these commands also inherit the quadrature rule, material models, and other attributes of the element set.

An element set may be turned into a fan or a heat exchanger component by referencing it in a FAN_COMPONENT or a HEAT_EXCHANGER_COMPONENT command. For example,

```
ELEMENT_SET( "radiator element set" ) {
   ...
}
HEAT_EXCHANGER_COMPONENT( "radiator" ) {
   element_set    = "radiator element set"
   ...
}
```

turns the element set "radiator element set" into a radiator. In such cases, the element set must contain all the elements of the component and no other elements. Moreover, the element set may be referenced by only one component command.

The *auxiliary_elements* parameter provides a mechanism for element user-defined functions, such as heat source or body force, to access data from other elements. This is similar to the *auxiliary_nodes* parameter of the NODAL_BOUNDARY_CONDITION command. The format of the array is the same as that for elements, including the number of elements, element numbering, and the element topology. For each original element a user-defined function can obtain data from its auxiliary element by using the functions udfGetElmAuxCrd() and udfGetElmAuxData(); see the *AcuSolve User-Defined Functions Manual* for a detailed description of these functions. The data returned will be for the equivalent quadrature point of the original element. For this reason the auxiliary element connectivity normally is ordered in the same way as for the original element. The *auxiliary_material_model* parameter defines the material model to be used with the auxiliary elements. This parameter accepts the user-given name of a MATERIAL_MODEL command. Similarly, *auxiliary_reference_frame* defines the reference frame to be used for the auxiliary elements. These last two parameters are needed to properly extract the data from the auxiliary element. For example,

```
ELEMENT_SET( "air" ) {
   elements                   = { 1, 1, 2, 3, 4 ;
                                  2, ... ;
                                  3, ... ; }
   shape                      = four_node_tet
   medium                     = fluid
   material_model             = "air"
   body_force                 = "air/fin heat transfer"
   auxiliary_elements         = { 1, 11, 12, 13, 14 ;
                                  2, ... ; 3, ... ; }
   auxiliary_material_model   = "fin"
   auxiliary_reference_frame  = none
}
ELEMENT_SET( "fin" ) {
   elements                   = { 1001, 11, 12, 13, 14 ;
                                  1002, ... ;
                                  1003, ... ; }
   shape                      = four_node_tet
   medium                     = fluid
   material_model             = "fin"
   body_force                 = "fin/air heat transfer"
   auxiliary_elements         = { 1001, 1, 2, 3, 4 ; 1002, ... ; 1003, ... ; }
   auxiliary_material_model   = "air"
   auxiliary_reference_frame  = none
}
BODY_FORCE( "air/fin heat transfer" ) {
   mass_heat_source          = "air/fin heat transfer"
}
```

```
MASS_HEAT_SOURCE( "air/fin heat transfer" ) {
   type                        = user_function
   user_function               = "usrFinAirHeat"
   user_values                 = { 1 , # 1: air side, 2: fin side
                                   1.e6/1.225 } # coefficient
}
BODY_FORCE( "fin/air heat transfer" ) {
   mass_heat_source            = "fin/air heat transfer"
}
MASS_HEAT_SOURCE( "fin/air heat transfer" ) {
   type                        = user_function
   user_function               = "usrFinAirHeat"
   user_values                 = { 2 , # 1: air side, 2: fin side
                                   1.e6/8666 } # coefficient
}
```

Here the pairs of nodes

```
1      11
2      12
3      13
4      14
```

have the same coordinates. This restriction is not necessary, but in most cases the auxiliary element will be simply related to the original element, that is, shifted, scaled and/or rotated. The auxiliary element nodes may be read from a file. For the above example, the auxiliary element nodes for the "air" element set may be placed in a file, such as `aux_fin.cnn`:

```
1      11    12   13   14
2      ...
3      ...
```

and read by:

```
ELEMENT_SET( "air" ) {
   ...
   auxiliary_elements          = Read( "aux_fin.cnn" )
   auxiliary_material_model    = "fin"
   auxiliary_reference_frame   = none
}
```

The user-defined function "usrFinAirHeat" may be implemented as:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrFinAirHeat ) ;                    /* function prototype
          */
   Void usrFinAirHeat (
   UdfHd                udfHd,                      /* Opaque handle for accessing
 data      */
   Real*                outVec,                     /* Output vector
          */
   Integer              nItems,                     /* Number of elements in this
 block      */
   Integer              vecDim                      /* = 1
          */
) {
   Integer              elem ;                      /* an element counter
          */
```

```
   Integer              side ;                          /* which side of heat transfer?
         */
   Real                 fct ;                           /* scaling factor
         */
   Real                 hCoef ;                         /* modified heat transfer
 coefficient    */
   Real                 heatCoef ;                      /* heat transfer coefficient
         */
   Real*                airTemp ;                       /* air temperature
         */
   Real*                airVel ;                        /* air velocity
         */
   Real*                crd ;                           /* coordinates
         */
   Real*                finTemp ;                       /* fin temperature
         */
   Real*                srcJac ;                        /* partial heat source /
 partial temp    */
   Real*                usrVals ;                       /* user values
         */
/* Get the data */
         */
   udfCheckNumUsrVals( udfHd, 2 ) ;                     /* check for error
         */
   usrVals              = udfGetUsrVals( udfHd ) ;      /* get the user vals
         */
   side                 = (Integer) usrVals[0] ;        /* get the side
         */
   heatCoef             = usrVals[1] ;                  /* get the heat coef.
         */
/* Get the material data */
   if ( side == 1 ) { /* air side */
      airTemp           = udfGetElmData(                udfHd,
 UDF_ELM_TEMPERATURE      ) ;
      airVel            = udfGetElmData(                udfHd,          UDF_ELM_VELOCITY
        ) ;
      finTemp           = udfGetElmAuxData(             udfHd
 UDF_ELM_TEMPERATURE      ) ;
      srcJac            = udfGetElmJac(                 udfHd,
 UDF_ELM_JAC_TEMPERATURE  ) ;
      crd               = udfGetElmCrd(                 udfHd
        ) ;
      fct               = 1. ;
   } else {
      airTemp           = udfGetElmAuxData(             udfHd,
 UDF_ELM_TEMPERATURE      ) ;
      airVel            = udfGetElmAuxData(             udfHd,          UDF_ELM_VELOCITY
        ) ;
      finTemp           = udfGetElmData(                udfHd,
 UDF_ELM_TEMPERATURE      ) ;
      srcJac            = udfGetElmJac(                 udfHd,
 UDF_ELM_JAC_TEMPERATURE  ) ;
      crd               = udfGetElmAuxCrd(              udfHd
        ) ;
      fct               = -1. ;
   }
/* Compute the heat flux */
   for ( elem = 0 ; elem < nItems ; elem++ ) {
      hCoef             = heatCoef ;                    /* could be a function of
 airVel          */
      outVec[elem]      = fct * hCoef * ( finTemp[elem] - airTemp[elem]);
      srcJac[elem]      = -hCoef ;
   }
```

```
} /* end of usrFinAirHeat() */
```

The dimension of the returned coefficient vector, *outVec*, is the number of elements.

The design ELEMENT_SET for topology optimization should include the following as well.

```
ELEMENT_SET( "Design" ) {
   …
   medium = design
   material_model = "fluid_medium"
   design_material_model = "poros_solid_medium"
   design_variables_field = "topology"
   …
}
```

Modeling a screen or a filter using the AcuSolve-EDEM coupling approach presents a challenge due to the extremely high mesh count required for AcuSolve. Simplification is necessary to reduce computational time. In this scenario, the pressure drop is modeled in the flow using a porous medium, while particle interaction with the screen is handled in EDEM. The *porous_medium_region* parameter in the ELEMENT_SET can be used to activate the porous medium based on the physical velocity porous approach in one of the carrier field elements. To enable this parameter, AcuSolve-EDEM bidirectional coupling must be turned on by setting *edem_coupling* = on and *porous_media_velocity_type* = physical under EQUATION. Similar to single-phase porous medium applications, a POROSITY_MODEL is required to define the porosity of the flow equations. This is applicable only to the carrier field element set. The POROSITY_MODEL command is referenced by the MATERIAL_MODEL command, which is, in turn, linked to FIELD commands for defining FIELD_INTERACTION_MODELS in the Eulerian_Eulerian model.

Below is an example usage of the *porous_medium_region*:

```
EQUATION {
   porous_media_velocity_type = physical
   multi_field = eulerian_eulerian
   fields = {"EDEM Particle", "Air"}
   edem_coupling = on
}
POROSITY_MODEL( "Porous" ) {
   type = constant
   porosity = 0.36
   permeability_type = cartesian
   permeability_direction = { 1, 0, 0; 0, 1, 0; 0, 0, 1 }
   direction_1_permeability = 2e-8
   direction_2_permeability = 2e-10
   direction_3_permeability = 2e-10
   darcy_coefficient = 40
   forchheimer_coefficient = 18
   effectiveness_type = none
}
MATERIAL_MODEL( "Air" ) {
   type = fluid
   density_model = "Air"
   conductivity_model = "Air"
   specific_heat_model = "Air"
   viscosity_model = "Air"
   porosity_model = "Porous"
}
FIELD( "Air" ) {
   material_model = "Air"
}
```

```
FIELD( "EDEM Particle" ) {
   material_model = "EDEM Particle"
}
FIELD_INTERACTION_MODEL( "Air-EDEM Particle" ) {
   type = eulerian_eulerian
   carrier_field = "Air"
   dispersed_field = "EDEM Particle”
}
MULTI_FIELD_MODEL( "Air-EDEM Particle - 2way" ) {
   field_interaction_models = { "Air-EDEM Particle" }
}
ELEMENT_SET( "Main Domain" ) {
   multi_field_model = "Air-EDEM Particle - 2way"
   medium = multi_field
}
ELEMENT_SET( "Porous media" ) {
   multi_field_model = "Air-EDEM Particle - 2way"
   medium = multi_field
   porous_media_region = on
}
```

EXTERNAL_CODE_SURFACE

Specifies a fluid surface that interacts with a solid/structural surface from an external code.

Type

AcuSolve Command

Syntax

EXTERNAL_CODE_SURFACE("name") {parameters...}

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral
six_node_triangle or tri6	Six-node triangle.

element_set or elem_set (string) [no default]

User-given name of the parent element set.

surfaces (array) [no default]

List of element surfaces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this command. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to this command. This option is used in place of directly specifying these parameters. In the event that both of the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

coupling_type (enumerated) [=structural]

Manner in which the AcuSolve surface is coupled with the external code.

structural	AcuSolve surface is coupled with either a line or surface element set in the external code. This option requires *mesh_displacement_type* = tied when coupling to line elements.

line	AcuSolve surface is coupled with a line element set in the external code. This option requires *mesh_displacement_type* = tied.
surface	AcuSolve surface is coupled with a surface element set in the external code.
rigid_body	AcuSolve surface is coupled with a rigid body in the external code.

coupling_direction (enumerated) [=both]
External code coupling direction for the surface.

both	Exchanges information with the external code object in both directions. AcuSolve sends loading to the external code and the external code sends displacements back to AcuSolve.
to_external_code	AcuSolve sends loading information to the external code, but no displacement information is returned.
from_external_code	AcuSolve does not send loading information to the external code, but displacement information is provided to AcuSolve at each step.

rigid_body_name (string) [no default]
Name of the rigid body the external code surface is coupled with. Used with *coupling_type* = rigid_body.

velocity_type (enumerated) [=wall]
Type of the velocity boundary condition.

wall	Fluid follows the solid (no-slip).
slip	Fluid slips on the solid.

temperature_type (enumerated) [=tied]
Type of the temperature boundary condition.

none	No fluid temperature boundary condition (adiabatic).
tied	Fluid temperature same as solid wall.

mesh_displacement_type (enumerated) [=tied]
Type of the mesh displacement boundary condition.

tied	Fluid mesh is fixed to surface.
slip	Fluid mesh slides on the solid.

turbulence_wall_type (enumerated) [=wall_function]
Type of the turbulence wall modeling. Used with wall velocity type.

none	No wall modeling. When this option is set, the surface does not participate in the wall distance calculation for turbulence and no wall modeling is performed.
low_reynolds_number or low_re	Low Reynolds number damping functions.
wall_function or func	Turbulence wall function. Requires *roughness_height*.
running_average_wall_function	Turbulence wall function based on running average variables. Requires *roughness_height*.

roughness_height (real) >=0 [=0]
Average wall roughness height. Used with low_reynolds_number, wall_function and running_average_wall_function turbulence wall types.

wall_function_heat_flux_factor >=1 [=1]
Constant factor used to scale the turbulent thermal conductivity within the first element off the wall. Used with *turbulence_wall_type* = wall_function and running_average_wall_function.

external_code_tags (list) [={}]
Array of strings used to this fluid surface to solid surfaces. The null list matches to all solid surfaces and beams.

precedence (integer) [=1]
Precedence of the nodal boundary conditions defined here with respect to others defined elsewhere. Highest value has precedence on a per-variable basis.

gap_factor (real) >=0 [=1]
Nondimensional (with respect to the length of an element face) maximum gap allowed for two element faces to be in contact. No maximum if zero.

gap (real) >=0 [=0]
Dimensional maximum gap allowed for two element faces to be in contact. No maximum if zero.

Description

This command specifies a set of surfaces (element faces) to be used to interact with the solid surfaces in an external code in order to solve Direct Coupling Fluid Structure Interaction (DC-FSI) problems. For example,

```
EXTERNAL_CODE_SURFACE( "fluid wetted surface" ){
 shape                   = four_node_quad
 element_set             = "fluid region"
 surfaces                = { 1, 101, 1, 3, 11, 12 ;
                             2, 102, 3, 5, 13, 14 ; }
 velocity_type           = wall
 temperature_type        = tied
 mesh_displacement_type  = slip
 turbulence_wall_type    = wall_function
 roughness_height        = 0
 precedence              = 1
 gap_factor              = 1
 gap                     = 0
}
```

There are two main forms of this command. The legacy version (or single topology version) of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number (of the parent element set), a unique (within this set) surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as dcfsi.wet.ebc:

```
1 101 1 3 11
2 102 3 5 13
```

and read by:

```
EXTERNAL_CODE_SURFACE( "fluid wetted surface" ) {
 shape         = four_node_quad
 element_set   = "fluid region"
 surfaces      = Read( "dcfsi.wet.ebc" )
 ...
}
```

The mixed topology form of the EXTERNAL_CODE_SURFACE command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the

SURFACE_SET command, and are then referred to by the EXTERNAL_CODE_SURFACE command. For example, a collection of triangular and quadrilateral element faces can be defined using the following SURFACE_SET commands.

```
SURFACE_SET( "tri faces" ) {
    surfaces       = { 1, 1, 1, 2, 4 ;
                       2, 2, 3, 4, 6 ;
                       3, 3, 5, 6, 8 ; }
    shape          = three_node_triangle
    volume_set     = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
    surfaces       = { 1, 1, 1, 2, 4, 9 ;
                       2, 2, 3, 4, 6, 12 ;
                       3, 3, 5, 6, 8, 15 ; }
    shape          = four_node_quad
    volume_set     = "prisms"
```

Then, a single EXTERNAL_CODE_SURFACE command is defined that contains the tri and quad faces as follows:

```
EXTERNAL_CODE_SURFACE ( "fluid wetted surface" ) {
   surface_sets       = {"tri_faces", "quad_faces"}
   ...
}
```

The list of surface sets can also be placed in a file, such as surface_sets.srfst:

```
tri faces
quad faces
```

and read using:

```
EXTERNAL_CODE_SURFACE ( "fluid wetted surface" ) {
   surface_sets       = Read("surface_sets.srfst")
   ...
}
```

The mixed topology version of the EXTERNAL_CODE_SURFACE command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *surface_sets* and *surfaces* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

The fluid forces, moments, and heat fluxes are communicated from all the surfaces defined in EXTERNAL_CODE_SURFACE commands to an external solid/structural code defined by the EXTERNAL_CODE command. Mesh displacements and temperatures are returned from the external code to the same surfaces. For each quadrature point of each external code surface element face, a search is made over all appropriate solid model faces in the external code for:

- The distance between the point and the face is less than *gap*; and
- The distance between the point and the face, normalized by the length of the face, is less than *gap_factor*.

Each constraint is ignored if the corresponding parameter is zero. If both parameters are zero, then there are no constraints and each quadrature point is guaranteed to be in contact with another face. Fluid forces are actually applied to the solid model only if the fluid quadrature point is determined to be in contact with a solid face. Similarly, nodal boundary conditions from the solid model are applied only if the solid model node is in contact with the external code surface.

The search over solid surfaces may be constrained through the use of tags. For example,

```
EXTERNAL_CODE_SURFACE( "Drill" ) {
   ...
   external_code_tags = {"0"}
   ...
}
EXTERNAL_CODE_SURFACE( "Riser" ) {
   ...
   external_code_tags = {}
   ...
}
```

The "Drill" surface will be matched only to solid surfaces tagged with "0", while the "Riser" surface will be matched to all solid surfaces.

A *coupling_type* = structural indicates that all surfaces participating in the exchange between AcuSolve and the external code are of a single, consistent element type (that is line/beam elements or surface elements). For example, if a simulation contains five EXTERNAL_CODE_SURFACE commands, they must all be coupled to line elements or surface elements in the structural model. Prior to the release of AcuSolve 13.0, it was only possible to couple with a single type and this option is therefore supplied for backward compatibility. Starting in AcuSolve 13.0, it is possible to couple with surface elements, line elements and rigid body surfaces within a single simulation. This is accomplished by explicitly defining the *coupling_type* parameter for each EXTERNAL_CODE_SURFACE command that is issued.

> **Note:** If a *coupling_type* = structural is used in the same simulation in which a *coupling_type* = line, *coupling_type* = surface or *coupling_type* = rigid_body is specified, an assertion will be issued and the code will exit.

This EXTERNAL_CODE_SURFACE command determines all the boundary conditions on the wetted surface, that is, all the points that are determined to be in contact, if precedence is sufficiently high. The parameter *mesh_displacement_type* may be either slip, allowing the mesh to slide against the solid, such as in a half-filled tank,; or tied, where the fluid mesh displacement is tied to the solid. For slip, the solid surface acts just like a guide surface; see the description of guide_surface type in the NODAL_BOUNDARY_CONDITION command for details about the algorithm used. If *velocity_type* is set to wall, the fluid velocity moves with the solid wall. If viscous effects are not important, it may be set to slip. For the former case, turbulence nodal boundary conditions are set automatically. Wall modeling is also set with *turbulence_wall_type* and *roughness_height*.

Temperature is set equal to the solid temperature if *temperature_type* is tied, otherwise the wetted surface is adiabatic, unless other temperature boundary conditions are specified on the surface.

FAN_COMPONENT

Specifies a fan component for an element set.

Type

AcuSolve Command

Syntax

FAN_COMPONENT("name") {parameters...}

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral.
six_node_triangle or tri6	Six-node triangle.

element_set or elem_set (string) [no default]

User-given name of the parent element set. This element set is turned into a fan component by this command.

surfaces (array) [no default]

List of component inlet element surfaces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this fan component. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to the FAN_COMPONENT command. This option is used in place of directly specifying these parameters. In the event that both the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

type (enumerated) [=axial]

Type of fan.

axial	Axial fan. Fluid enters axially.
radial	Radial fan. Fluid enters radially.

center direction [={0,0,0}]

A point on the center of rotation, specified in the global xyz coordinate system.

direction or *dir* *(array) [={1,0,0}]*
Axial direction of the fan, specified in the global xyz coordinate system.

angular_speed or *omega* *(real) [=1]*
Fan rotational speed, in radians per unit of time.

angular_speed_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the angular speed. If none, no scaling is performed.

tip_radius or *radius* *(real) >0 [=1}*
Radius of the tip of the fan.

thickness *(real) >0 [=1]*
Fan thickness.

axial_coefficient_type *(enumerated) [=constant]*
Type of the axial coefficient.

constant or const	Constant for the entire set. Requires *axial_coefficient*.
piecewise_bilinear or bilinear	Piecewise bilinear curve fit. Requires *axial_curve_fit_values*, *axial_curve_fit_row_variable* and *axial_curve_fit_column_variable*.
user_function or user	User-defined function. Requires *axial_user_function*, *axial_user_values* and *axial_user_strings*.

axial_coefficient *(real) [=1]*
The constant value of the axial coefficient. Used with constant.

axial_curve_fit_values *(array) [={0,0;0,1}]*
Axial coefficient as a two dimensional curve fit table with two independent variables. Used with piecewise_linear axial coefficient type.

axial_curve_fit_row_variable *(enumerated) [=normalized_radius]*
Independent variable of the rows of the axial curve fit table. Used with piecewise_linear axial coefficient type.

normalized_radius or radius	Radius normalized by the tip radius.

axial_curve_fit_column_variable *(enumerated) (=normalized_flow_rate)*
Independent variable of the columns of the axial curve fit table. Used with piecewise_linear axial coefficient type.

normalized_flow_rate or flow_rate	Flow rate normalized with respect to inlet area, density, and tip velocity.

axial_user_function *(string) [no default]*
Name of the user-defined function for the axial coefficient. Used with user_function axial coefficient type.

axial_user_values *(array) [={}]*
Array of values to be passed to the user-defined function. Used with user_function axial coefficient type.

axial_user_strings (list) [={}]
Array of strings to be passed to the user-defined function. Used with user_function axial coefficient type.

axial_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the axial coefficient. If none, no scaling is performed.

radial_coefficient_type (enumerated) [=constant]
Type of the radial coefficient

constant or const	Constant for the entire set. Requires *radial_coefficient*.
piecewise_linear or linear	Piecewise bilinear curve fit. Requires *radial_curve_fit_values*, *radial_curve_fit_row_variable* and *radial_curve_fit_column_variable*.
user_function or user	User-defined function. Requires *radial_user_function*, *radial_user_value* and *radial_user_strings*.

radial_coefficient (real) [=1]
The constant value of the radial coefficient. Used with constant radial coefficient type.

radial_curve_fit_values (array) [={0,0;0,1}]
Radial coefficient as a two dimensional curve fit table with two independent variables. Used with piecewise_linear radial coefficient type.

radial_curve_fit_row_variable (enumerated) [=normalized_radius]
Independent variable of the rows of the radial curve fit table. Used with piecewise_linear radial coefficient type.

normalized_radius or radius	Radius normalized by the tip radius.

radial_curve_fit_column_variable (enumerated) [=normalized_flow_rate]
Independent variable of the columns of the radial curve fit table. Used with piecewise_linear radial coefficient type.

normalized_flow_rate or flow_rate	Flow rate normalized with respect to inlet area, density, and tip velocity.

radial_user_function (string) [no default]
Name of the user-defined function for the radial coefficient. Used with user_function radial coefficient type.

radial_user_values (array) [={}]
Array of values to be passed to the user-defined function. Used with user_function radial coefficient type.

radial_user_strings (list) [={}]
Array of strings to be passed to the user-defined function. Used with user_function radial coefficient type.

radial_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the radial coefficient. If none, no scaling is performed.

tangential_coefficient_type (enumerated) [=constant]
Type of the tangential coefficient.

constant or const	Constant for the entire set. Requires *tangential_coefficient*.
piecewise_linear or linear	Piecewise bilinear curve fit. Requires *tangential_curve_fit_values*, *tangential_curve_fit_row_variable* and *tangential_curve_fit_column_variable*.
user_function or user	User-defined function. Requires *tangential_user_function*, *tangential_user_values* and *tangential_user_strings*.

tangential_coefficient (real) [=1]
The constant value of the tangential coefficient. Used with constant tangential coefficient type.

tangential_curve_fit_values (array) [={0,0;0,1}]
Tangential coefficient as a two dimensional curve fit table with two independent variables. Used with piecewise_linear tangential coefficient type.

tangential_curve_fit_row_variable (enumerated) [=normalized_radius]
Independent variable of the rows of the tangential curve fit table. Used with piecewise_linear tangential coefficient type.

normalized_radius or radius	Radius normalized by the tip radius.

tangential_curve_fit_column_variable (enumerated) [=normalized_flow_rate]
Independent variable of the columns of the tangential curve fit table. Used with piecewise_bilinear tangential coefficient type.

normalized_flow_rate or flow_rate	Flow rate normalized with respect to inlet area, density, and tip velocity.

tangential_user_function (string) [no default]
Name of the user-defined function for the tangential coefficient. Used with user_function tangential coefficient type.

tangential_user_values (array) [={}]
Array of values to be passed to the user-defined function. Used with user_function tangential coefficient type.

tangential_user_strings (list) [={}]
Array of strings to be passed to the user-defined function. Used with user_function tangential coefficient type.

tangential_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the tangential coefficient. If none, no scaling is performed.

pq_curve_type (enumerated) [=none]
Type of the fan pressure-flow rate (P-Q) performance curve to simplify the fan modeling process. You can specify fan performance directly through P-Q curve, by-passing the legacy approach of non-dimensional coefficient calculations.

none	None. Specified when using non-dimensional coefficient as input.
piecewise_linear	Piecewise linear curve fit. Requires *pq_curve_fit_values*.

pq_curve_fit_values (array) [={}]
The *pq_curve_fit_values* parameter is a two-dimensional array which defines the fan performance curve with the first column of flow rate in m^3/s and the second column of pressure increase in Pa.

swirl_factor (real) [=0]
Ratio of tangential body force and the primary body force.

Description

This command specifies the parameters of a fan component. It also turns all the elements of an element set into a single fan.

For example,

```
ELEMENT_SET( "engine fan" ) {
   shape                           = four_node_tet
   elements                        = { 1, 1, 9, 8, 3 ;
                                       2, 3, 8, 9, 5 ;... }
   ...
}
FAN_COMPONENT( "fan" ) {
   shape                           = three_node_triangle
   element_set                     = "engine fan"
   surfaces                        = { 1, 101, 1, 9, 8 ;
                                       2, 102, 8, 9, 5 ; }
   type                            = axial
   center                          = { 0, 0, 0 }
   direction                       = { 1, 0, 0 }
   angular_speed                   = 3600 * (2*PI) / 60 # 3600 RPM
   tip_radius                      = 0.11
   thickness                       = 0.06
   axial_coefficient_type          = piecewise_bilinear
   axial_curve_fit_values          = {
          0 ,   0.0876, 0.0943, 0.1098, 0.1523, 0.1724, 0.2632 ;
          0.43, 0.3783, 0.2685, 0.2385, 0.1783, 0.1493, 0.0185 ;
          0.61, 0.3878, 0.2738, 0.2429, 0.1856, 0.1576, 0.0246 ;
          0.72, 0.3925, 0.2782, 0.2482, 0.1909, 0.1649, 0.0289 ;
          0.95, 0.3943, 0.2819, 0.2517, 0.1962, 0.1692, 0.0342 ;
   axial_curve_fit_row_variable    = normalized_radius
   axial_curve_fit_column_variable = normalized_flow_rate
   radial_coefficient_type         = constant
   radial_coefficient              = 0
   tangential_coefficient_tye      = piecewise_bilinear
   tangential_curve_fit_values     = {
          0 ,   0.0876, 0.0943, 0.1098, 0.1523, 0.1724, 0.2632 ;
          0.43, 0.0205, 0.0351, 0.0774, 0.1102, 0.1299, 0.0608 ;
          0.61, 0.0899, 0.0764, 0.0771, 0.0798, 0.0766, 0.0423 ;
```

```
          0.72, 0.0599, 0.0523, 0.0572, 0.0603, 0.0587, 0.0398 ;
          0.95, 0.0401, 0.0344, 0.0399, 0.0412, 0.0402, 0.0254 ;
   tangential_curve_fit_row_variable      = normalized_radius
   tangential_curve_fit_column_variable   = normalized_flow_rate
}
```

turns the "engine fan" element set into a fan component. The inlet of this axial fan is defined by the two surfaces (element faces) of elements 1 and 2. The axis of the fan passes through the global point (zero, zero, zero) in the global xyz coordinate system. The fan blows in the positive global x-direction. The fan speed is 3600 RPM (377 radians per second) in the clockwise direction. The fan tip radius (at the shroud) is 0.11. The distance from the front to the rear of the fan, fan thickness, is 0.06. The axial and tangential coefficients of the fan use a two-dimensional curve fit, while the radial coefficient is zero.

There are two main forms of this command. The previous version (or single topology version) of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element shape	Surface shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number (of the parent element set), a unique (within this set) surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, surfaces has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The mixed topology form of the FAN_COMPONENT command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the SURFACE_SET command, and are then referred to by the FAN_COMPONENT command. For example, a

collection of triangular and quadrilateral element faces can be defined using the following SURFACE_SET commands.

```
SURFACE_SET( "tri faces" ) {
   surfaces    = { 1, 1, 1, 2, 4 ;
                   2, 2, 3, 4, 6 ;
                   3, 3, 5, 6, 8 ; }
   shape       = three_node_triangle
   volume_set  = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
   surfaces    = { 1, 1, 1, 2, 4, 9 ;
                   2, 2, 3, 4, 6, 12 ;
                   3, 3, 5, 6, 8, 15 ; }
   shape       = four_node_quad
   volume_set  = "prisms"
```

Then, a single FAN_COMPONENT command is defined that contains the tri and quad faces as follows:

```
FAN_COMPONENT( "fan" ) {
   surface_sets       = { "tri_faces", "quad_faces" }
   ...
}
```

The list of surface sets can also be placed in a file, such as surface_sets.srfst:

```
tri faces
quad faces
```

and read using:

```
FAN_COMPONENT( "fan" ) {
   surface_sets       = Read ("surface_sets.srfst")
   ...
}
```

The mixed topology version of the FAN_COMPONENT command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both of the *surface_sets* and *surfaces* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

Two types of fans are supported. In an axial type, the fluid enters the fan in the axial direction, such as in the case of an engine fan. In a radial type, the flow enters radially at the hub of the fan, such as in the case of a blower fan.

A fan component is modeled by adding axial, tangential, and radial forces to the momentum equations. For an axial fan type, these forces increase the pressure across the component by:

$$\Delta P_{\text{axial}} = \frac{1}{2}\alpha_{\text{axial}}\rho u_{\text{tip}}^2 + \frac{1}{2}\rho \bar{u}^2$$
$$\Delta P_{\text{radial}} = \frac{1}{2}\alpha_{\text{radial}}\rho u_{\text{tip}}^2 \quad (154)$$
$$\Delta P_{\text{tang}} = \frac{1}{2}\alpha_{\text{tang}}\rho u_{\text{tip}}^2$$

where ΔP_{axial}, ΔP_{radial}, and ΔP_{tang} are, respectively, the pressure increases in the axial, radial, and tangential directions; α_{axial}, α_{radial}, and α_{tang} are, respectively, the axial, radial, and tangential coefficients of the fan component, specified by *axial_coefficient_type*, *radial_coefficient_type*, *tangential_coefficient_type*, and related parameters; ρ is the density; $u_{\text{tip}} = \omega r_{\text{tip}}$ is the tip velocity; ω is the fan rotational speed, specified by *angular_speed*; r_{tip} is the fan tip radius, specified by *tip_radius*; and $\bar{u}$ is the mass averaged velocity through the inlet surface of the fan, where the inlet surface is specified by *surfaces*.

For a radial fan type, the pressure increases in the radial and tangential directions are the same as for an axial type. However, the pressure increase in the axial direction is given by:

$$\Delta P_{\text{axial}} = \frac{1}{2}\alpha_{\text{axial}}\rho u_{\text{tip}}^2 \quad (155)$$

The orientation of the fan is defined by *radial* and *direction*. These are in the global xyz coordinate system. The fan speed is given by *angular_speed*. Angular speed is in radians per unit time and uses the right hand rule to define the direction of the fan rotations. For positive values, the fan (looking in the direction axis) rotates clockwise, while for negative values the fan rotates in the counter-clockwise direction.

tip_radius defines the radius at the tip of the fan. This parameter must be consistent with the calibration data used to derive the fan coefficients.

thickness specifies the thickness of the fan. For an axial fan, this is the thickness in the axial direction of the fan. For a radial fan, this is the distance from the hub to the shroud.

A constant axial (tangential or radial) coefficient applies a uniform coefficient value to all quadrature points of the element set. For the above example, the radial coefficient is uniformly set to zero.

A piecewise_bilinear axial coefficient type defines a piecewise bilinear curve fit as a function of two independent variables. *axial_curve_fit_row_variable* and *axial_curve_fit_column_variable* define the two independent variables. The *axial_curve_fit_values* parameter is a two dimensional array which defines the coefficients of the interpolation. The (zero, zero) entry of this array must be zero. The first row of the table defines the values of the independent column variable, while the first column of the table defines the values of the independent row variable. The values of both independent variables must be in ascending order. For the above example, the first row (0.0876, ...,

0.2632) defines the *normalized_flow_rate* values and the first column (0.43, ..., 0.95) defines the *normalized_radius* values. The values

```
0.3783 ... 0.0185
...
0.3943 ... 0.0342
```

are the axial coefficients.

The coefficients may be read from a file. For the above example, the axial coefficients may be placed in a file, such as `fan.axial.fit`:

```
0    0.0876 0.0943 0.1098 0.1523 0.1724 0.2632
0.43 0.3783 0.2685 0.2385 0.1783 0.1493 0.0185
0.61 0.3878 0.2738 0.2429 0.1856 0.1576 0.0246
0.72 0.3925 0.2782 0.2482 0.1909 0.1649 0.0289
0.95 0.3943 0.2819 0.2517 0.1962 0.1692 0.0342
```

and read by:

```
FAN_COMPONENT( "fan" ) {
   ...
   axial_curve_fit_values = Read( "fan.axial.fit" )
}
```

A piecewise_bilinear tangential or radial coefficient type is defined as in the axial case.

The user_function type for axial, tangential or radial coefficients allows for a more complex behavior; see the *AcuSolve User-Defined Functions Guide* for a detailed description of user-defined functions.

For example, consider the case where the tangential coefficient is a quadratic function of average inlet velocity and decays linearly with radius.

```
FAN_COMPONENT( "fan with UDF tangential coefficient" ) {
    ...
    tangential_coefficient_type  = user_function
    tangential_user_function     = "usrFanExample"
    tangential_user_values       = { 0,   0,   0,    # fan center
                                     1,   0,   0,    # fan direction
                                     0,   0,   0.2,  # velocity coefs
                                     0.11,            # fan tip radius
                                  3600                # fan speed
                                   }
}
```

where the user-defined function "usrFanExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrFanExample ) ; /* function prototype */
Void usrFanExample (
     UdfHd             udfHd,                                          /*
 Opaque handle for accessing data */
  Real*             outVec,                                          /*
 Output vector  */
  Integer           nItems,                                          /*
 Number of elements  */
  Integer           vecDim                                           /* =
 1  */
```

```
) {
    Integer             elem ;                                          /*
 an element counter  */
    Real                coef0 ;                                         /*
 constant coefficient  */
    Real                coef1 ;                                         /*
 linear coefficient  */
    Real                coef2 ;                                         /*
 quadratic coefficient */
    Real                radRat ;                                        /*
 radius ratio */
    Real                radius ;                                        /*
 fan radius  */
    Real                rpm ;                                           /*
 fan speed  */
    Real                tipVel ;                                        /*
 tip velocity  */
    Real                tmp ;                                           /*
 a temporary value */
    Real                velCoef ;                                       /*
 velocity coefficient  */
    Real                xCenter ;                                       /*
 x-coordinates of cente  */
    Real                xDir ;                                          /*
 x-direction of fan  */
    Real                xPnt ;                                          /*
 x-coord. of the point  */
    Real                yCenter ;                                       /*
 y-coordinates of center  */
    Real                yDir ;                                          /*
 y-direction of fan  */
    Real                yPnt ;                                          /*
 y-coord. of the point  */
    Real                zCenter ;                                       /*
 z-coordinates of center  */
    Real                zDir ;                                          /*
 z-direction of fan  */
    Real                zPnt ;                                          /*
 z-coord. of the point  */
    Real*               aveVel ;                                        /*
 average velocity  */
    Real*               crd ;                                           /*
 coordinates  */
    Real*               usrVals ;                                       /*
 user values  */
    Real*               xCrd ;                                          /*
 x-coordinates  */
    Real*               yCrd ;                                          /*
 y-coordinates  */
    Real*               zCrd ;                                          /*
 z-coordinates  */
    char*               name ;                                          /*
 set name  */
udfCheckNumUsrVals( udfHd, 11 ) ;                                       /*
 check for error  */
    usrVals             = udfGetUsrVals( udfHd ) ;                      /*
 get the user vals */
    xCenter             = usrVals[0] ;                                  /*
 get x-center  */
    yCenter             = usrVals[1] ;                                  /*
 get y-center  */
    zCenter             = usrVals[2] ;                                  /*
 get z-center  */
```

```
    xDir                = usrVals[3] ;                                          /*
 get x-direction   */
    yDir                = usrVals[4] ;                                          /*
 get y-direction    */
    zDir                = usrVals[5] ;                                          /*
 get z-direction   */
    coef0               = usrVals[6] ;                                          /*
 get const coef.   */
    coef1               = usrVals[7] ;                                          /*
 get lin. coef.    */
    coef2               = usrVals[8] ;                                          /*
 get quad. coef.  */
    radius              = usrVals[9] ;                                          /*
 get fan radius   */
    rpm                 = usrVals[10] ;                                         /*
 get fan RPM    */
    name                = udfGetName( udfHd ) ;                                 /*
 get the fan name  */
    aveVel              = udfGetFanData( udfHd, name, UDF_FAN_AVERAGE_VELOCITY ) ;  /*
 get ave. velocity */
    crd                 = udfGetElmCrd( udfHd ) ;                               /*
 get the coord.  */
    xCrd                = &crd[0*nItems] ;                                      /*
 localize x-coord.  */
    yCrd                = &crd[1*nItems] ;                                      /*
 localize y-coord.  */
    zCrd                = &crd[2*nItems] ;                                      /*
 localize z-coord.  */
    tipVel              = radius                                                /*
 tip velocity  */
                        * rpm * 2 * 3.1415926535897931 / 60 ;
    tmp                 = *aveVel / tipVel ;                                    /*
 velocity ratio  */
    velCoef             = coef0 + coef1 * tmp + coef2 * tmp * tmp ;             /*
 quadratic func.  */
    for ( elem          = 0 ; elem < nItems ; elem++ ) {
        xPnt            = xCrd[elem] - xCenter ;
        yPnt            = yCrd[elem] - yCenter ;
        zPnt            = zCrd[elem] - zCenter ;
        tmp             = xDir * xPnt + yDir * yPnt + zDir * zPnt ;
        radRat          = xPnt * xPnt + yPnt * yPnt + zPnt * zPnt
                        - tmp * tmp ;
        if ( radRat < 0 ) radRat = 0 ;
        radRat          = sqrt( radRat ) / radius ;
        outVec[elem]    = velCoef * ( 1. - radRat ) ;
    }
} /* end of usrFanExample() */
```

The dimension of the returned coefficient vector, outVec, is the number of elements.

The *angular_speed_multiplier_function*, *axial_multiplier_function*, *tangential_multiplier_function*, and *radial_multiplier_function* parameters may be used to scale the angular speed, and the axial, tangential, and radial coefficients respectively. The value of each of these parameters refers to the user-given name of a MULTIPLIER_FUNCTION command in the input file. For example, to impose a uniform axial coefficient as a function of time, the following commands may be issued:

```
FAN_COMPONENT( "fan" ) {
   ...
   type                         = axial
   center                       = { 0, 0, 0 }
```

```
   direction                    = { 1, 0, 0 }
   angular_speed                = 3600 * (2*PI) / 60 # 3600 RPM
   tip_radius                   = 0.11
   thickness                    = 0.06
   axial_coefficient_type       = constant
   axial_coefficient            = 1.
   axial_multiplier_function    = "time varying"
   tangential_coefficient_type  = constant
   tangential_coefficient       = 0
   radial_coefficient_type      = constant
   radial_coefficient           = 0
}
MULTIPLIER_FUNCTION( "time varying" ) {
   type                         = piecewise_linear
   curve_fit_values             = { 0, 0.0 ;10, 1.0 ;20, 0.5 ;40, 0.7 ;80, 1.2 ; }
   curve_fit_variable           = time
}
```

AcuSolve 2020 (or a later release) can provide an option of specifying the fan performance curve directly, by-passing the non-dimensional coefficient calculation. As a consequence of this, input variables such as *tip_radius* and *angular_speed* are not required. Below is an example:

```
FAN_COMPONENT( "Fan_Inlet" ) {
    surface_sets                    = {  "Fan_Inlet_Fan_Upstream_Duct" }
    type                            = axial
    center                          = { 0, 0, 0 }
    direction                       = { 1, 0, 0 }
    thickness                       = 0.06
    pq_curve_type                   = piecewise_linear
    pq_curve_fit_values             = {525.35, 494.910;
                                    890.21, 474.63;
                                    1161.63, 424.9;
                                    1272.76, 389.11;
                                    1356.57, 350.42;
                                    1431.84, 308.18;
                                    1494.69, 268.35;
                                    1551.39, 230.89;
                                    }
    swirl_factor                    = 0.0
}
```

The axis of the axial fan passes through the origin of the global coordinate system. The fan blows in the positive x-direction. The distance from the front to the rear of the fan, fan thickness, is 0.06 m. The fan performance curve is defined as the two-dimensional curve fit, while the swirl factor is set to zero. The first column of the table defines the flow rate (m^3/s), while the second column of the table is the pressure increase (Pa). The swirl factor is a ratio of the tangential body force to the primary body force (axial or radial).

Some scalar data are written to disk for each fan component and at every time step. This data can be translated to other formats using the AcuTrans program and other post-processing modules; see the *AcuSolve Programs Reference Manual* for details.

FEEDBACK_CONDITION

Specifies a feedback condition and its parameters.

Type

AcuSolve Command

Syntax

```
FEEDBACK("name") {parameters...}
```

Parameters

type (enumerated) [point]

Type of feedback condition.

point	Feedback condition is associated with a reference point.

point (string) [no default]

User-given name of the TIME_HISTORY_OUTPUT that provides the points on which the feedback condition will be evaluated.

control_type (enumerated) [upper_limit]

Condition that is satisfied.

upper_limit	Feedback condition is enabled if the condition is above a certain value.
target_value	Feedback condition is enabled/disabled if the condition is below/above a certain value.

control_variable_evaluation (enumerated) [average]

How the feedback condition on the reference region is satisfied.

average	*control_type* evaluated based on the average value of the control variable in the reference region. For *type* = points this represents the average value of all points in a set.
maximum	*control_type* evaluated based on the maximum value of the control variable in the reference region. For *type* = points this represents the maximum value of all points in a set.
minimum	*control_type* evaluated based on the minimum value of the control variable in the reference region. For *type* = points this represents the minimum value of all points in a set.

control_variable (enumerated) [temperature]

Variable used to evaluate the control type.

temperature Temperature is used to evaluate the *control_type* in the reference region.

control_value (real) [=0]
Value of the *control_variable* in the reference region used to evaluate the *control_type*.

input_variable (enumerated) [heat]
Input variable that is controlled by the condition specified in the reference region.

heat Heat input is controlled by the *control_type* and *control_variable*.

Description

The FEEDBACK_CONDITION is designed to regulate input heat, for example, a surface flux or volumetric heat source, based on specific criteria within a designated region or point in the domain, for example, a surface. An illustrative example of this condition is an oven during the heating phase. As the oven heats up, the oven heater operates, while the air temperature is continuously monitored to ensure the oven reaches the desired temperature. As long as the oven temperature remains below the desired level, the heater remains active. Once the oven temperature reaches or exceeds the desired level, the heater deactivates and remains off while the temperature remains at or above the desired threshold.

To achieve the above protocol, a FEEDBACK_CONDITION can be created:

```
FEEDBACK_CONDITION("feedback_example") {
    type = point
    point = "reference point"
    control_type = target_value
    control_variable_evaluation = maximum
    control_variable = temperature
    control_value = 453.15
    input_variable = heat
 }
```

In this scenario, the FEEDBACK_CONDITION references a TIME_HISTORY_OUTPUT, that is, the "reference point", which could represent a thermocouple continuously monitoring temperature, for instance:

```
TIME_HISTORY_OUTPUT(  "reference_point" ) {
    type = nodal
    nodes = { 48326 }
    output_frequency = 1.0
    output_time_interval = 0.0
}
```

The FEEDBACK_CONDITION is linked to a heat source, that is, the input to the control system, such as a flux or volumetric heat source. The heat source undergoes adjustments based on the parameters of the FEEDBACK_CONDITION, for example, maintaining the desired temperature of 453.15K. As an example, a VOLUME_HEAT_SOURCE referencing a FEEDBACK_CONDITION is:

```
VOLUME_HEAT_SOURCE(  "HeatSource" ) {
    type = constant
    volume_heat_source = 5e07
    feedback_condition = "feedback_example"
}
```

Currently, FEEDBACK_CONDITION is only supported by the energy equation. The *input_variable* of type heat can only be a VOLUME_HEAT_SOURCE or SIMPLE_BOUNDARY_CONDITION.

FIELD_BOUNDARY_CONDITION

Specifies field element and nodal boundary conditions on a set of element faces referred to in SIMPLE_BOUNDARY_CONDITION.

Type

AcuSolve Command

Syntax

FIELD_BOUNDARY_CONDITION("name") {parameters...}

Qualifier

User-given name.

Parameters

type (enumerated) [=value]

Type of the boundary condition imposition.

value	Field nodal values.
flux	Field volume fraction flux values. Valid only for *multi_field*= eulerian_eulerian.

value (real) [=0.0]

Field nodal values to be imposed on nodes associated with surface elements referred to in SIMPLE_BOUNDARY_CONDITION. For *multi_field*=levelset, algebraic_eulerian, eulerian_eulerian, field volume fraction should be provided. However, for *multi_field*=advective_diffusive, the mass fraction value of the field should be given.

flux (real) [=0.0]

Field volume fraction flux to be imposed on surface elements referred to in SIMPLE_BOUNDARY_CONDITION. Valid only for *multi_field*= eulerian_eulerian and needed for dispersed field. For fluidization type problems, the flux value for solid phase should be set to zero in the inlet.

field (string) [no default]

Name of the field that value or flux is associated with. Field should be an active field in the problem. This should be given for all *multi_field* types.

multiplier_function (string) [=none]

User-given name of the multiplier function for scaling the boundary condition values. If none, no scaling is performed.

phasic_inflow_type (enumerated) [=cartesian_velocity]

Type of the inflow phasic velocity boundary conditions. Used with *multi_field*= eulerian_eulerian only.

cartesian_velocity	Velocity in cartesian coordinates. Requires *x_velocity*, *y_velocity*, and *z_velocity*.

cylindrical_velocity	Velocity in cylindrical coordinates. Requires *cylinder_axis*, *axial_velocity*, *radial_velocity*, and *tangential_velocity*.
spherical_velocity	Radial velocity component in spherical coordinates. The other two components are zero. Requires *sphere_center* and *radial_velocity*.
normal_velocity	Velocity component normal to surface. The other two components are zero. Requires *normal_velocity*.
mass_flux or mass	Integrated mass flux (equivalent to mass flow rate) and nodal scalars (except eddy viscosity).
flow_rate or flow	Integrated volume flux (equivalent to volumetric flow rate) and nodal scalars (except eddy viscosity).
average_velocity or ave_vel	Average velocity and nodal scalars (except eddy viscosity).

cylinder_axis (array) [={0,0,0; 0,1,0}]
Coordinates of two points that define the axis of the cylinder. Used with *phasic_inflow_type*= cylindrical_velocity type.

sphere_center (array) [={0,0,0}]
Coordinates of the center of the sphere. Used with *phasic_inflow_type*= spherical_velocity.

x_velocity or xvel (real) [=0]
Value of the x-velocity. Used with *phasic_inflow_type*= cartesian_velocity.

x_velocity_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the x-velocity boundary condition value. If none, no scaling is performed. Used with *phasic_inflow_type*= cartesian_velocity.

y_velocity or yvel (real) [=0]
Value of the y-velocity. Used with *phasic_inflow_type*= cartesian_velocity.

y_velocity_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the y-velocity boundary condition value. If none, no scaling is performed. Used with *phasic_inflow_type*= cartesian_velocity.

z_velocity or zvel (real) [=0]
Value of the z-velocity. Used with *phasic_inflow_type*= cartesian_velocity.

z_velocity_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the z-velocity boundary condition value. If none, no scaling is performed. Used with *phasic_inflow_type*= cartesian_velocity.

axial_velocity (real) [=0]
Value of the axial velocity. Used with *phasic_inflow_type*=cylindrical_velocity.

axial_velocity_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the axial velocity boundary condition value. If none, no scaling is performed. Used with *axial_velocity*.

radial_velocity *(real) [=0]*
Value of the radial velocity. Used with *phasic_inflow_type*= cylindrical_velocity.

radial_velocity_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the radial velocity boundary condition value. If none, no scaling is performed. Used with *radial_velocity*.

tangential_velocity *(real) [=0]*
Value of the tangential velocity. Used with *phasic_inflow_type*= cylindrical_velocity.

tangential_velocity_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the tangential velocity boundary condition value. If none, no scaling is performed. Used with *tangential_velocity*.

normal_velocity *(real) [=0]*
Value of the normal velocity. Used with *phasic_inflow_type*= normal_velocity.

normal_velocity_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the normal velocity boundary condition value. If none, no scaling is performed. Used with *normal_velocity*.

mass_flux *or* *mass* *(real) >0 [=1]*
Value of the integrated mass flux. Used with *phasic_inflow_type*= mass_flux.

mass_flux_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the integrated mass flux value. If none, no scaling is performed. Used with inflow type and *mass_flux* inflow type.

flow_rate *or* *flow* *(real) >0 [=1]*
Value of the integrated flow rate. Used with *phasic_inflow_type*= flow_rate.

flow_rate_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the integrated flow rate value. If none, no scaling is performed. Used with inflow type and *flow_rate* inflow type.

average_velocity *or* *ave_vel* *(real) >0 [=1]*
Value of the average velocity. Used with inflow type and *average_velocity* inflow type.

average_velocity_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the average velocity value. If none, no scaling is performed. Used with inflow type and *average_velocity* inflow type.

FREE_SURFACE

Specifies a free surface.

Type

AcuSolve Command

Syntax

FREE_SURFACE("name") {parameters...}

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral
six_node_triangle or tri6	Six-node triangle.

element_set or *elem_set (string) [no default]*

User-given name of the parent element set.

surfaces (array) [no default]

List of element surfaces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this command. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to the FREE_SURFACE command. This option is used in place of directly specifying these parameters. In the event that both of the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

surface_tension_model (string) [=none]

User-given name of the surface tension model. If none, surface tension is not modeled.

contact_angle_model (string) [=none]

User-given name of the contact angle model. Used when *surface_tension_model* is specified. If none, the default contact angle model is used.

Description

This command specifies the surfaces (element faces) that define a free surface and imposes the appropriate nodal boundary condition on the mesh motion on those surfaces. Optional surface tension and contact angle models may also be specified.

The free-surface surfaces are defined with respect to the elements of an element set. For example,

```
ELEMENT_SET( "interior" ) {
    shape                  = four_node_tet
    elements               = { 1, 8, 3, 4, 9 ;
                               3, 3, 4, 9, 5 ;
                               ... }
    ...
}
FREE_SURFACE( "water surface" ) {
    shape                  = three_node_triangle
    element_set            = "interior"
    surfaces               = { 1, 12, 9, 3, 4 ;
                               3, 52, 5, 3, 4 ; }
    surface_tension_model  = "water/air"
    contact_angle_model    = "water/air @ smooth aluminum"
}
```

specifies that two surfaces of the element set "interior" are on a free surface boundary, and that this free surface has certain properties specified by the *surface_tension_model* and *contact_angle_model* parameters.

There are two main forms of this command. The legacy version, or single topology version, of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number, of the parent element set, a unique, within this set, surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six

columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as free.srf:

```
1 12 9 3 4
3 52 5 3 4
```

and ready by:

```
FREE_SURFACE( "water surface" ) {
   shape       = three_node_triangle
   element_set = "interior"
   surfaces    = Read( "free.srf" )
...
}
```

The mixed topology form of the FREE_SURFACE command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the SURFACE_SET command, and are then referred to by the FREE_SURFACE command. For example, a collection of triangular and quadrilateral element faces can be defined using the following SURFACE_SET commands.

```
SURFACE_SET( "tri faces" ) {
    surfaces     = { 1, 1, 1, 2, 4 ;
                     2, 2, 3, 4, 6 ;
                     3, 3, 5, 6, 8 ; }
    shape        = three_node_triangle
    volume_set   = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
    surfaces     = { 1, 1, 1, 2, 4, 9 ;
                     2, 2, 3, 4, 6, 12 ;
                     3, 3, 5, 6, 8, 15 ; }
    shape        = four_node_quad
    volume_set   = "prisms"
```

Then, a single FREE_SURFACE command is defined that contains the tri and quad faces as follows:

```
SURFACE_INTEGRATED_CONDITION ( "water surface" ) {
   surface_sets       = {"tri_faces", "quad_faces"}
   ...
}
```

The list of surface sets can also be placed in a file, such as surface_sets.srfst:

```
tri faces
quad faces
```

and read using:

```
FREE_SURFACE ( "water surface" ) {
   surface_sets    = Read("surface_sets.srfst")
   ...
}
```

The mixed topology version of the FREE_SURFACE command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *surface_sets* and *surfaces* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

This command also automatically creates the following nodal (Dirichlet) boundary condition on the mesh velocity for all surface nodes:

$$u_{\text{mesh}} \bullet n = u_{\text{fluid}} \bullet n \qquad (156)$$

where u_{mesh} is the mesh velocity, u_{fluid} is the fluid velocity, and **n** is the surface normal at the nodes. The latter is the area-weighted average of the neighboring element surface normals. This particular averaging ensures that the mass conservation property of the continuity equation becomes a volume-conserving property of the mesh movement equation, which in turn is needed for mass conservation. The precedence of this boundary condition is automatically set to zero.

> **Note:** This is lower than the default value of one for boundary conditions created through the NODAL_BOUNDARY_CONDITION command.

The *surface_tension_model* parameter provides the surface tension information to the free surface. This parameter specifies the user-given name of a SURFACE_TENSION_MODEL command. If this parameter is not given the surface tension defaults to zero. If it is given, then in addition a contact angle at the side walls of the free surface may be specified with the *contact_angle_model* parameter. This parameter specifies the user given name of a CONTACT_ANGLE_MODEL command. If *contact_angle_model* is not given, the contact angle defaults to 90 degrees. See the SURFACE_TENSION_MODEL command for details.

The momentum equation also requires a boundary condition on the free surface. Normally, a constant-pressure element boundary condition should be imposed, which is all that is needed.

The side walls of a tank, or any surface that intersects with a free surface, require special care at the intersection. If a side wall has a no-slip velocity boundary condition, then the above boundary condition on the mesh motion will prevent the free surface from moving at the corner where it meets the wall. Thus, for just these corner nodes, the no-slip condition needs to be replaced by a slip velocity boundary condition in order to allow the correct free surface motion. A slip condition on mesh displacement is typically used for the entire side wall.

In order for a free surface to be physically stable, some type of restoring force is necessary. This is usually provided by an acceleration of the reference frame, or a body force such as gravity acting towards the interior of the fluid.

FWH_SURFACE_OUTPUT

Specifies a set of surfaces for integrated-surface output parameters for the Ffowcs-Williams Hawkings (FWH) method that is used by the AcuFwh post processing program to propagate acoustic sources to far field locations.

Type

AcuSolve Command

Syntax

FWH_SURFACE_OUTPUT("name") {parameters...}

Qualifier

User-given name.

Parameters

coordinates or *coord* *(array) [no default]*
: An array of sample points. Each row contains a unique identification number and the x, y and z coordinates of a point in the global xyz coordinate system.

gap *(real) >=0 [=0]*
: Tolerance for sample points falling outside of all elements.

shape *(enumerated) [no default]*
: Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilaterial.

partial_surfaces *(array) [no default]*
: List of surfaces with no references to parent elements or element sets.

quadrature *(enumerated) [=full]*
: Quadrature rule used for surface integration.

full	Full quadrature.
reduced	Reduced quadrature.

Description

This command specifies a set of surfaces for the FWH output data. Surface geometry and integration rules are given here, while the related parameters are specified in `FWH_OUTPUT` command. For example,

```
FWH_SURFACE_OUTPUT( "sample surfaces" ) {
   coordinates      = Read( "sample.crd" )
   partial_surfaces = Read( "sample.srf" )
   shape            = four_node_quad
   quadrature       = full
```

```
}
```

defines an acoustic surface output for use with AcuFwh.

A set of sample points is specified by the *coordinates* parameter, which is a four-column array corresponding to an identification number and the x, y and z coordinates of each sample point, one row per point. The identification number may be any arbitrary but unique (within the set) integer. Negative numbers are acceptable. The coordinates are in the global xyz coordinate system. This is the same format as the *coordinates* parameter of the COORDINATE command, but the sample points do not have to be a subset of the nodal coordinates nor related to them in any other way. The *coordinates* parameter is mandatory. Within AcuPrep, an element is found for each sample point and interpolation shape functions are computed. If a sample point falls outside all fluid elements, then the closest element face is found, up to a distance specified by *gap*. An error is issued if no element is found.

The *partial_surfaces* parameter contains the connectivity of the surfaces. This parameter is a multi-column array, but it does not have the same format as the corresponding parameter in other surface commands, for example SURFACE_OUTPUT. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has four columns, corresponding to a unique (within this set) surface number and the three nodes of the surface. For four_node_quad, partial_surfaces have five columns, corresponding to a surface number and the four nodes of the surface. One row per surface must be given. Since there is no parent element definition the three or four nodes of the surface must be ordered around the periphery of the surface. Also, unlike other surface commands this command does not support higher-order surfaces, that is, six-node triangles, or nodal quadrature. The nodes contained in the surface connectivity must all be defined in the coordinates.

FWH_OUTPUT

Specifies output parameters for the Ffowcs-Williams Hawkings (FWH) method that is used by the AcuFwh post processing program to propagate acoustic sources to far field locations.

Type

AcuSolve Command

Syntax

FWH_OUTPUT{parameters...}

Qualifier

None.

Parameters

reference_density or *ref_dens* *(real) [=1.225]*
: Reference density. Required to compute the output quantities.

reference_pressure or *ref_pres* *(real) [=0]*
: Reference pressure. Required to compute the output quantities.

reference_sound_speed or *sound* *(real) [=330]*
: Reference sound speed. Required to compute the output quantities.

source_output_frequency or *out_freq* *(integer) >=0 [=0]*
: Time step frequency at which to output FWH data. If zero, this option is ignored.

source_output_time_interval or *out_intv* *(real) >=0 [=0]*
: Time frequency at which to output the FWH data. If zero, this option is ignored.

num_saved_states or *save* *(integer) >=0 [=0]*
: Number of FWH data files to keep. If zero, all are kept.

Description

This command specifies the FWH output data properties to support Ffowcs-Williams Hawkings (FWH) post-processing. For example,

```
FWH_OUTPUT{
 reference_density           = 1.225
 reference_pressure          = 0
 reference_sound_speed       = 330
 source_output_frequency     = 1
 source_output_time_interval = 0
 num_saved_states            = 0
}
```

The parameters *reference_density*, *reference_pressure* and *reference_sound_speed* provide the required data for the computation of the output fields. An AcuSolve input file should only contain one FWH_OUTPUT command.

GUIDE_SURFACE

Specifies a surface for mesh nodes to follow.

Type

AcuSolve Command

Syntax

GUIDE_SURFACE("name") {parameters...}

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral
six_node_triangle or tri6	Six-node triangle.

element_set or elem_set (string) [no default]

User-given name of the parent element set.

surfaces (array) [no default]

List of element surfaces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this command. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to this command. This option is used in place of directly specifying these parameters. In the event that both of the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

split_internal_surfaces (boolean) [=off]

Option to automatically split the nodes on internal surfaces that require two boundary conditions, that is, cases that include baffles, sliding mesh or thermal interfaces.

type (enumerated) [=faceted]

Type of the guide surface.

faceted	Mesh nodes follow the surface as given.
smooth	Apply polynomial to surface node definition.

mesh_motion (string) [=none]
: User-given name of the MESH_MOTION command for specifying mesh displacement boundary conditions on all the nodes in this surface set. If none, no boundary conditions are imposed.

mesh_motion_precedence (integer) [=1]
: User-given precedence for mesh motion surfaces. High values of *mesh_motion_precedence* will take precedence over lower values for duplicate *mesh_motion* boundary conditions.

Description

This command specifies a set of surfaces (element faces) to be used to guide mesh nodes. One application of this is the free surface problem around a boat with curved surfaces. The commands to specify that the fluid nodes on the boat surface shall follow this surface may be given as:

```
ELEMENT_SET( "boat" ) {
    shape          = four_node_tet
    elements       = Read( "boat.cnn" )
    medium         = none
    mesh_motion    = none
}
GUIDE_SURFACE( "stationary boat surface" ) {
    shape          = three_node_triangle
    element_set    = "boat"
    surfaces       = { 1, 101, 1, 3, 11 ;
                       2, 102, 3, 5, 13 ; }
    type           = faceted
    mesh_motion    = none
}
NODAL_BOUNDARY_CONDITION( "guided mesh motion" ) {
    nodes          = Read( "boat.wall.nbc" )
    variable       = mesh_direction_displacement
    type           = guide_surface
    guide_surface  = "stationary boat surface"
}
NODAL_BOUNDARY_CONDITION( "x-velocity on boat" ) {
    nodes          = Read( "boat.wall.nbc" )
    variable       = x_velocity
    type           = zero
}
NODAL_BOUNDARY_CONDITION( "y-velocity on boat" ) {
    nodes          = Read( "boat.wall.nbc" )
    variable       = y_velocity
    type           = zero
}
NODAL_BOUNDARY_CONDITION( "z-velocity on boat" ) {
    nodes          = Read( "boat.wall.nbc" )
    variable       = z_velocity
    type           = zero
}
```

In this example two surfaces of the element set "boat" are used to define a guide surface. In general all surfaces are defined in relation to an element set. Here, an element set is needed for the sole purpose of defining geometry for the guide surface. To avoid creating extra equations, *medium* = none allows an element set to be completely inactive, or to have only mesh displacement boundary conditions on its nodes. An active element set, one with a medium other than none, may be used as well. The guide surface is then used in a special type of nodal boundary condition that allows the given mesh nodes to move but constrains them to the guide surface. A faceted type of guide surface means that the nodes

will follow these surfaces exactly as defined, as opposed to following a smoothed version. Each node in a boundary condition set with a guide_surface type is projected to the given guide surface. If this projection falls within a surface facet, then the normal to that facet is used for a slip boundary condition on the mesh displacement. If the projection falls outside of all the guide surface facets, then the closest one is used. Effectively, the guide surface is extended as far as necessary to cover all the mesh nodes.

There are two main forms of this command. The legacy version, or single topology version, of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number, of the parent element set, a unique, within this set, surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as boat.ebc:

```
1 101 1 3 11
2 102 3 5 13
```

and read by:

```
GUIDE_SURFACE( "boat surface" ) {
    shape        = three_node_triangle
    element_set  = "boat"
    surfaces     = Read( "boat.ebc" )
    ...
```

```
}
```

The mixed topology form of the GUIDE_SURFACE command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the SURFACE_SET command, and are then referred to by the GUIDE_SURFACE command. For example, a collection of triangular and quadrilateral element faces can be defined using the following SURFACE_SET commands.

```
SURFACE_SET( "tri faces" ) {
    surfaces       = { 1, 1, 1, 2, 4 ;
                       2, 2, 3, 4, 6 ;
                       3, 3, 5, 6, 8 ; }
    shape          = three_node_triangle
    volume_set     = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
    surfaces       = { 1, 1, 1, 2, 4, 9 ;
                       2, 2, 3, 4, 6, 12 ;
                       3, 3, 5, 6, 8, 15 ; }
    shape          = four_node_quad
    volume_set     = "prisms"
```

Then, a single GUIDE_SURFACE command is defined that contains the tri and quad faces as follows:

```
GUIDE_SURFACE ( "boat surface" ) {
    surface_sets       = {"tri_faces", "quad_faces"}
    ...
}
```

The list of surface sets can also be placed in a file, such as surface_sets.srfst:

```
tri faces
quad faces
```

and read using:

```
GUIDE_SURFACE ( "boat surface" ) {
   surface_sets       = Read("surface_sets.srfst")
    ...
}
```

The mixed topology version of the GUIDE_SURFACE command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *surface_sets* and *surfaces* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

In the above examples the boat is stationary. A rigid body motion can be added to the boat. For example, the forces of the ocean acting on the boat can be modeled by:

```
MESH_MOTION( "boat motion" ) {
    type                          = rigid_body_dynamic
    rigid_body_mass               = 100
```

```
    ...
    rigid_body_surface_outputs    = { "ocean: boat hull" }
}
SURFACE_OUTPUT( "ocean: boat hull" ) {
    ...
    integrated_output_frequency   = 1
}
GUIDE_SURFACE( "moving boat surface" ) {
    ...
    mesh_motion                   = "boat motion"
}
SIMPLE_BOUNDARY_CONDITION( "slip conditions for mesh and fluid" ) {
    ...
    type                          = slip
    mesh_displacement_type        = guide_surface
    guide_surface                 = "moving boat surface"
}
```

Alternatively, the *mesh_motion* parameter can be specified in the element set that the guide surface belongs to.

There are four different cases for the fluid velocity boundary condition on the guide surface:

- Fixed Surface, No-Slip Velocity: Zero velocity is imposed on all nodes using either NODAL_BOUNDARY_CONDITION commands or a SIMPLE_BOUNDARY_CONDITION command of type wall. The no-slip condition may not be imposed on the nodes at the free surface or else the free surface could not slide along the guide surface.
- Fixed Surface, Slip Velocity: The easiest way to handle this is to use a SIMPLE_BOUNDARY_CONDITION command with type slip. This will impose a zero boundary condition on the normal component of velocity, where the normal is taken from the guide surface and not the fluid mesh.
- Moving Surface, No-Slip Velocity: The guide surface is moving, or deforming for that matter, and the velocity is fixed to the body, that is, moving along with the body. Using a SIMPLE_BOUNDARY_CONDITION command of type wall causes the fluid velocity on the wall to match that of the guide surface. If in addition a mesh displacement type of guide_surface is specified in the same SIMPLE_BOUNDARY_CONDITION command then the normal component of the fluid velocity will match the normal component of the mesh velocity instead of the guide surface velocity. This preserves the fluid volume. An example for this case using NODAL_BOUNDARY_CONDITION commands is given in the description for that command.
- Moving Surface, Slip Velocity: Using a SIMPLE_BOUNDARY_CONDITION command of type slip and mesh displacement type of guide_surface causes the fluid velocity to slip properly on the moving wall while preserving the fluid volume, as in case three.

HEAT_EXCHANGER_COMPONENT

Specifies a heat exchanger component for an element set.

Type

AcuSolve Command

Syntax

HEAT_EXCHANGER_COMPONENT("name") {parameters...}

Qualifier

User-given name.

Parameters

type (enumerated) [=constant_coolant_heat_reject]

Type of heat exchanger model to use.

constant_coolant_heat_reject	Constant heat rejection. Requires *coolant_heat_reject*.
constant_coolant_temperature	Constant coolant temperature. Requires *coolant_temperature*.

shape (enumerated) [no default]

Shape of the surfaces.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral
six_node_triangle or tri6	Six-node triangle.

element_set or *elem_set (string) [no default]*

User-given name of the parent element set. This element set is turned into a heat exchanger component by this command.

surfaces (array) [no default]

List of component inlet element surfaces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this heat exchanger component. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to the HEAT_EXCHANGER_COMPONENT command. This option is used in place of directly specifying these parameters. In the event that both the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

coolant_heat_reject or *heat_reject* *(real) [=0]*
The amount of heat rejected by the heat exchanger. Used with constant_coolant_heat_reject type.

coolant_temperature *(real) [=0]*
The temperature of the heat exchanger coolant. Used with constant_coolant_temperature type.

coolant_heat_reject_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the amount of heat rejected by the heat exchanger. If none, no scaling is performed.

coolant_temperature_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the coolant temperature. If none, no scaling is performed.

coolant_flow_rate or *flow_rate* *(real) >=0 [=0]*
Flow rate of the coolant liquid.

coolant_flow_rate_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the flow rate of the coolant liquid. If none, no scaling is performed.

direction or *dir* *(array) [={1,0,0}]*
Direction of the flow through the heat exchanger.

thickness *(real) >0 [=1]*
Thickness of the heat exchanger: the distance from the inlet surface to the outlet surface.

upstream_distance *(real) >=0 [=0]*
Distance in front of the heat exchanger inlet surface at which the upstream temperature is sampled.

friction_type *(enumerated) [=constant]*
Type of the friction factor model.

constant or const	Constant friction factor. Requires *friction*.
kays_london	Kays-London friction factor. Requires *wet_min_area_ratio*, *core_friction_constant*, *core_friction_exponent*, *inlet_loss_coefficients* and *outlet_loss_coefficients*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *friction_curve_fit_values* and *friction_curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *friction_curve_fit_values* and *friction_curve_fit_variable*.
user_function or user	User-defined function. Requires *friction_user_function*, *friction_user_values*, and *friction_user_strings*.

friction *(real) [=0]*
Constant value of the friction factor. Used with constant friction type.

inlet_min_area_ratio *(real) >0 [=1]*
Ratio of inlet area to minimum flow area.

outlet_min_area_ratio *(real) >0 [=1]*
Ratio of outlet area to minimum flow area. This typically is equal to *inlet_min_area_ratio*

wet_min_area_ratio *(real) >0 [=1]*
Ratio of the wetted area to minimum flow area. Used with kays_london friction type.

core_friction_constant *(real) >=0 [=1]*
Constant coefficient of the core friction factor. Used with kays_london friction type.

inlet_loss_coefficients *(array) [={0.4,0.0,-0.4}]*
Loss coefficients for the inlet friction factor. This is a 3-component array corresponding to the constant, linear, and quadratic coefficients. Used with kays_london friction type.

outlet_loss_coefficients *(array) [={1,-2,1}]*
Loss coefficients for the outlet friction factor. This is a three component array corresponding to the constant, linear and quadratic coefficients. Used with kays_london friction type.

core_friction_exponent *(real) [=-1]*
Exponential coefficient of the core friction factor. Used with kays_london friction type.

friction_curve_fit_values *(array) [={0,1}]*
A two column array of independent-variable/friction-factor data values. Used with piecewise_linear and cubic_spline friction types.

friction_curve_fit_variable *(enumerated) [=axial_velocity]*
Independent variable of the friction factor curve fit. Used with piecewise_linear and cubic_spline friction types.

axial_velocity	Velocity in the direction of the component.

friction_user_function *(string) [no default]*
Name of the friction factor user-defined function. Used with user_function friction type.

friction_user_values *(array) [={}]*
Array of values to be passed to the friction factor user-defined function. Used with user_function friction type.

friction_user_strings *(list) [={}]*
Array of strings to be passed to the friction factor user-defined function. Used with user_function friction type.

friction_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the friction factor. If none, no scaling is performed.

transverse_friction_factor *(real) >=0 [=10]*
Fraction of the friction factor used for the transverse friction factor. Higher values yield higher degrees of unidirectional flow, but comes at the expense of stability.

effectiveness_type *(enumerated) [=constant]*
Type of the thermal effectiveness.

constant or const	Constant for the entire set. Requires *effectiveness*.

piecewise_bilinear or bilinear Piecewise bilinear curve fit. Requires *effectiveness_curve_fit_values*, *effectiveness_curve_fit_row_variable* and *effectiveness_curve_fit_column_variable*.

user_function or user User-defined function. Requires *effectiveness_user_function* and *effectiveness_user_strings*.

effectiveness (real) [=1]
The constant value of the thermal effectiveness. Used with constant effectiveness.

effectiveness_curve_fit_values (array) [={0,0;0,1}]
Effectiveness as a two dimensional curve fit table with two independent variables. Used with piecewise_bilinear effectiveness type.

effectiveness_curve_fit_row_variable (enumerated) [=coolant_flow_rate]
Independent variable of the rows of the effectiveness curve fit table. Used with piecewise_bilinear effectiveness type.

coolant_flow_rate Coolant flow rate.

effectiveness_curve_fit_column_variable (enumerated) [=axial_velocity]
Independent variable of the columns of the effectiveness curve fit table. Used with piecewise_bilinear effectiveness type.

axial_velocity Velocity in the direction of the component.

effectiveness_user_function (string) [no default]
Name of the user-defined function for the thermal effectiveness. Used with user_function effectiveness type.

effectiveness_user_values (array)[={}]
Array of values to be passed to the thermal effectiveness user-defined function. Used with user_function effectiveness type.

effectiveness_user_strings (list) [={}]
Array of strings to be passed to the thermal effectiveness user-defined function. Used with user_function effectiveness type.

effectiveness_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the thermal effectiveness. If none, no scaling is performed.

Description

This command specifies the parameters of a heat exchanger component. It also turns all the elements of an element set into a single heat exchanger. For example,

```
ELEMENT_SET( "radiator" ) {
    shape = four_node_tet
    elements = { 1, 1, 9, 8, 3 ;
                 2, 3, 8, 9, 5 ;
                 ... }
```

```
    ...
}
HEAT_EXCHANGER_COMPONENT( "radiator component" ) {
    shape                              = three_node_triangle
    element_set                        = "radiator"
    surfaces                           = { 1, 101, 1, 9, 8 ;
                                           2, 102, 8, 9, 5 ; }
    type                               = constant_coolant_heat_reject
    coolant_heat_reject                = 237443.5
    coolant_flow_rate                  = 4.6
    direction                          = { 1, 0, 0 }
    thickness                          = 0.1
    upstream_distance                  = 0.05
    friction_type                      = kays_london
    inlet_min_area_ratio               = 1.296
    outlet_min_area_ratio              = 1.296
    wet_min_area_ratio                 = 54.18
    core_friction_constant             = 23.505
    core_friction_exponent             = -.74
    inlet_loss_coefficients            = { 0.4, 0.0, -0.4 }
    outlet_loss_coefficients           = { 1.0, -2., 1.0 }
    transverse_friction_factor         = 100
    effectiveness_type                 = piecewise_bilinear
    effectiveness_curve_fit_values     = {
              0 ,  1.0000, 3.0000, 5.0000, 7.5000, 10.000, 13.500 ;
              0.4, 0.7487, 0.3302, 0.2511, 0.1731, 0.1411, 0.0987 ;
              1.1, 0.8877, 0.5422, 0.4301, 0.3021, 0.2524, 0.1822 ;
              1.7, 0.9123, 0.6321, 0.5332, 0.3923, 0.3387, 0.2487 ;
              2.6, 0.9344, 0.6953, 0.5962, 0.4693, 0.4023, 0.3084 ;
              3.5, 0.9452, 0.7233, 0.6287, 0.4976, 0.4344, 0.3329 ;
    effectiveness_curve_fit_row_variable = coolant_flow_rate
    effectiveness_curve_fit_column_variable = axial_velocity
}
```

turns the "radiator" element set into a heat exchanger component. The inlet to this heat exchanger is defined by the two surfaces (element faces) of elements one and two. The heat exchanger flows in the positive global x-direction. The heat exchanger transfers 237443.5 units of energy from the coolant to the flow. The coolant flow rate is 4.6. The component thickness is 0.1. The upstream temperature is sampled 0.05 coordinate units in front of the component inlet surface. The Kays-London model is used to incorporate the heat exchanger friction. A two dimensional curve fit is used to model the heat transfer effectiveness from the coolant to the flow.

There are two main forms of this command. The legacy version (or single topology version) of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad

Element Shape	**Surface Shape**
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number (of the parent element set), a unique (within this set) surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as `radiator.srf`:

```
1 101 1 9 8
2 102 8 9 5
```

and read by:

```
HEAT_EXCHANGER_COMPONENT( "radiator component" ) {
   shape        = three_node_triangle
   element_set  = "radiator"
   surfaces     = Read( "radiator.srf" )
   ...
}
```

The mixed topology form of the `HEAT_EXCHANGER_COMPONENT` command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the `SURFACE_SET` command, and are then referred to by the `HEAT_EXCHANGER_COMPONENT` command. For example, a collection of triangular and quadrilateral element faces can be defined using the following `SURFACE_SET` commands.

```
SURFACE_SET( "tri faces" ) {
    surfaces        = { 1, 1, 1, 2, 4 ;
                        2, 2, 3, 4, 6 ;
                        3, 3, 5, 6, 8 ; }
    shape           = three_node_triangle
    volume_set      = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
    surfaces        = { 1, 1, 1, 2, 4, 9 ;
                        2, 2, 3, 4, 6, 12 ;
                        3, 3, 5, 6, 8, 15 ; }
```

```
    shape           = four_node_quad
    volume_set      = "prisms"
```

Then, a single `HEAT_EXCHANGER_COMPONENT` command is defined that contains the tri and quad faces as follows:

```
HEAT_EXCHANGER_COMPONENT( "radiator component" ) {
   surface_sets       = {"tri_faces", "quad_faces"}
   ...
}
```

The list of surface sets can also be placed in a file, such as "`surface_sets.srfst`"

```
tri faces
quad faces
```

and read using:

```
HEAT_EXCHANGER_COMPONENT( "radiator component" ) {
    surface_sets       = Read("surface_sets.srfst")
    ...
}
```

The mixed topology version of the `HEAT_EXCHANGER_COMPONENT` command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *`surface_sets`* and *`surfaces`* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

A heat exchanger component is modeled by a "porous media" resistance in the momentum equations and a heat source in the energy equation. The pressure drop across a heat exchanger, in the axial direction, is given by:

$$\Delta P_{\text{axial}} = \frac{1}{2} f \rho \hat{u}^2 \tag{157}$$

where ΔP_{axial} is the pressure drop in the heat exchanger axial direction; the heat exchanger axial direction is specified by direction; f is the friction factor, given by *`friction_type`* and related parameters (see below); ρ is the density; $u=r_{\text{inlet}}u_{\text{axial}}$ is the axial velocity at the minimum cross-section area; r_{inlet} is the ratio of the inlet area to the minimum cross-section area, given by *`inlet_min_area_ratio`*; and u_{axial} is the axial component of the velocity.

To confine the flow to remain in the direction of the heat exchanger, pressure drops in the transverse directions of the heat exchanger component are introduced. The friction factor of these directions is scaled by *`transverse_friction_factor`*.

A volumetric heat source is added to the energy equation to model the heat source of the heat exchanger. This heat source is given by:

$$q = \frac{\rho c_p u_{\text{axial}} \varepsilon (T_{\text{hot}} - T_{\text{upstream}})}{L} \tag{158}$$

where q is the heat source per volume; cP is the specific heat at constant pressure; ε is the heat exchanger effectiveness, specified by *effectiveness_type* and related parameters (see below); T_{hot} is the "top water" temperature (that is, the temperature of the coolant); $T_{upstream}$ is the temperature upstream of the heat exchanger, taken at a distance from the inlet surface specified by *upstream_distance*; and L is the heat exchanger thickness, specified by *thickness*. The top-water temperature is determined (at each iteration) such that the total heat flux, Q, given by:

$$Q = \int q d\Omega \tag{159}$$

is equal to the heat exchanger heat rejection, specified by *coolant_heat_reject*. Here Ω is the heat exchanger volume.

A constant_coolant_heat_reject heat exchanger component adds a constant heat of *coolant_heat_reject* to the flow passing through the heat exchanger. The coolant flow is not modeled here; only its effect on the fluid flow is modeled.

Alternatively, a constant coolant temperature can be used,

```
HEAT_EXCHANGER_COMPONENT( "constant coolant temp radiator" ) {
    type                = constant_coolant_temperature
    coolant_temperature = 350.0
    ...
 }
```

The heat exchanger flow direction is given by *direction*, which is expressed in the global xyz coordinate system. The *direction* must be inward, normal to the heat exchanger inlet surface. This parameter is internally normalized.

thickness specifies the thickness (from the inlet to the outlet) of the heat exchanger. This thickness is assumed to be constant throughout the heat exchanger.

A constant friction factor applies a spatially constant friction factor. For example,

```
HEAT_EXCHANGER_COMPONENT( "constant friction radiator" ) {
  friction_type = constant
  friction      = 0.4
  ...
}
```

applies a constant friction factor of 0.4 to all the heat exchanger component elements.

The kays_london friction factor is given by:

$$f = f_{inlet} - f_{outlet} + f_{core} \tag{160}$$

where f_{inlet}, f_{outlet}, and f_{core} are, respectively, the inlet effect, exit effect, and core frictions. These are:

$$f_{\text{inlet}} = 1 + K_{\text{inlet}} - r_{\text{inlet}}^{-2}$$
$$f_{\text{outlet}} = 1 - K_{\text{outlet}} - r_{\text{oulet}}^{-2} \quad (161)$$
$$f_{\text{core}} = \alpha r_{\text{wet}} Re^{\beta}$$

Here K_{inlet} and K_{outlet} are the inlet and outlet loss coefficients, given by

$$K_{\text{inlet}} = a_0 + a_1 r_{\text{inlet}}^{-1} + a_2 r_{\text{inlet}}^{-2}$$
$$K_{\text{outlet}} = b_0 + b_1 r_{\text{oulet}}^{-1} + b_2 r_{\text{oulet}}^{-2}$$

The coefficients a_0, a_1, and a_2 are specified by *inlet_loss_coefficients*; the coefficients b_0, b_1, and b_2 are specified by *outlet_loss_coefficients*; r_{inlet} is the ratio of the inlet area to the minimum cross section area, specified by *inlet_min_area_ratio*; and r_{outlet} is the ratio of the outlet area to the minimum cross-section area, specified by *outlet_min_area_ratio*.

alpha and beta are the core friction coefficients, specified by *core_friction_constant* and *core_friction_exponent*, respectively; r_{wet} is the ratio of the wetted cross-section area to the minimum cross-section area, specified by *wet_min_area_ratio*; and the Reynolds number is defined by:

$$Re = \frac{\rho \tilde{u} D}{\mu} \quad (162)$$

where ρ is the density; $u=r_{\text{inlet}}u_{\text{axial}}$ is the axial velocity at the minimum cross-section area; u_{axial} is the axial component of the velocity; μ is the dynamic viscosity; and D is the hydraulic diameter given by:

$$D = \frac{4L}{r_{\text{wet}}} \quad (163)$$

L is the heat exchanger thickness, specified by *thickness*.

A piecewise_linear or cubic_spline type may be used to define the friction factor as a function of a single independent variable. For example,

```
HEAT_EXCHANGER_COMPONENT( "curve fit friction radiator" ) {
 friction_type             = piecewise_linear
 friction_curve_fit_values = { 0, 10.685 ;
                               5, 3.291 ;
                              10, 1.996 ;
                              15, 1.495 ;
                              20, 1.220 ;
                              30, 0.920 ;
                              40, 0.756 ;
                              50, 0.650 ; }
 friction_curve_fit_variable = axial_velocity
 ...
```

```
 }
```

defines the friction factor as a function of the axial velocity (that is, velocity in the direction of the heat exchanger). This curve fit is equivalent to the above Kays-London model, provided that the density is 1.225 and the dynamic viscosity is 1.781e-5. The *friction_curve_fit_values* parameter is a two-column array corresponding to the independent variable and the friction factor. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *friction_curve_fit_variable* falls outside of the curve fit limits.

The *friction_curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, such as `friction.fit`:

```
0 10.685
5 3.291
10 1.996
15 1.495
20 1.220
30 0.920
40 0.756
50 0.650
```

and read by:

```
HEAT_EXCHANGER_COMPONENT( "curve fit friction radiator" ) {
 friction_type               = piecewise_linear
 friction_curve_fit_values   = Read( "friction.fit" )
 friction_curve_fit_variable = axial_velocity
 ...
}
```

A friction factor of type user_function may be used to model more complex behaviors; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider the Kays-London friction factor of the above example. Then the input command may be given by:

```
HEAT_EXCHANGER_COMPONENT( "UDF friction radiator" ) {
 friction_type          = user_function
 friction_user_function = "usrFrictionExample"
 friction_user_values   = { 1,0,0, # flow direction
                            0.0629, # constant
                           10.6225, # proportional
                           -0.74 } # exponential
 ...
}
```

where the user-defined function "usrFrictionExample" may be implemented as follows:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE( usrFrictionExample ) ;                          /* function prototype
 */
Void usrFrictionExample (
    UdfHd        udfHd,                                        /* Opaque handle for
 accessing data */
    Real*        outVec,                                       /* Output vector */
    Integer      nItems,                                       /* Number of elements
 */
```

```
    Integer      vecDim                                          /* = 1 */
) {
    Integer      elem ;                                          /* an element counter
 */
    Real         aVel ;                                          /* axial velocity */
    Real         coef1 ;                                         /* constant
 coefficient */
    Real         coef2 ;                                         /* proportional
 coefficient */
    Real         coef3 ;                                         /* exponential
 coefficient */
    Real         xDir ;                                          /* x-component of
 direction vector */
    Real         yDir ;                                          /* y-component of
 direction vector */
    Real         zDir ;                                          /* z-component of
 direction vector */
    Real*        spec ;                                          /* specie field */
    Real*        usrVals ;                                       /* user values */
    Real*        vel ;                                           /* velocity field */
    Real*        xVel ;                                          /* x-velocity */
    Real*        yVel ;                                          /* y-velocity */
    Real*        zVel ;                                          /* z-velocity */
    udfCheckNumUsrVals( udfHd, 6 ) ;                             /* check for error */
    usrVals      = udfGetUsrVals( udfHd ) ;                      /* get the user vals
 */
    xDir         = usrVals[0] ;                                  /* x-direction */
    yDir         = usrVals[1] ;                                  /* y-direction */
    zDir         = usrVals[2] ;                                  /* z-direction */
    coef1        = usrVals[3] ;                                  /* const. coeff. */
    coef2        = usrVals[4] ;                                  /* prop. coeff. */
    coef3        = usrVals[5] ;                                  /* exp. coeff. */
    vel          = udfGetElmData( udfHd, UDF_ELM_VELOCITY ) ;    /* get velocity */
    xVel         = &vel[0*nItems] ;                              /* localized x-vel */
    yVel         = &vel[1*nItems] ;                              /* localized y-vel */
    zVel         = &vel[2*nItems] ;                              /* localized z-vel */
    for ( elem   = 0 ; elem < nItems ; elem++ ) {
        aVel     = xVel[elem] * xDir
                 + yVel[elem] * yDir
                 + zVel[elem] * zDir ;
    outVec[elem] = coef1 + coef2 * pow( aVel, coef3 ) ;
    }
} /* end of usrFrictionExample() */
```

The dimension of the returned friction vector, outVec, is the number of elements.

The friction factor in the transverse direction is controlled by *transverse_friction_factor*. Ideally, an infinite friction factor should be imposed in the transverse direction, resulting in a unidirectional flow through the heat exchanger. This proves, however, to be unstable. Therefore a multiple of the friction factor in the axial direction is imposed instead. The multiplier is specified by *transverse_friction_factor*; values from 10 to 100 are typically used.

The amount of heat transferred from the coolant to the flow depends on the effectiveness of the heat exchanger, as given in the above equations.

A constant effectiveness applies a spatially uniform effectiveness to all heat exchanger elements. For example,

```
HEAT_EXCHANGER_COMPONENT( "constant effectiveness radiator" ) {
  effectiveness_type = constant
```

```
  effectiveness      = 0.35
  ...
}
```

A piecewise_bilinear effectiveness defines a piecewise bilinear curve fit as a function of two independent variables, defined by *effectiveness_curve_fit_row_variable* and *effectiveness_curve_fit_column_variable*. The *effectiveness_curve_fit_values* parameter is a two dimensional array which defines the coefficients of the interpolation. The (zero , zero) entry of this array must be zero. The first row of the table defines the values of the independent column variable, while the first column of the table defines the values of the independent row variable. The values of both independent variables must be given in ascending order. For the above example, the first row (1.0000, ..., 13.500) defines the *coolant_flow_rate* values and the first column (0.4, ..., 3.5) defines the axial_velocity values. The values

```
0.7487 ... 0.0987
...
0.9452 ... 0.3329
```

are the effectiveness coefficients.

The coefficients may be read from a file. For the above example, the effectiveness coefficients may be placed in a file, such as `effectiveness.fit`:

```
0   1.0000 3.0000 5.0000 7.5000 10.000  13.500
0.4 0.7487 0.3302 0.2511 0.1731  0.1411  0.0987
1.1 0.8877 0.5422 0.4301 0.3021  0.2524  0.1822
1.7 0.9123 0.6321 0.5332 0.3923  0.3387  0.2487
2.6 0.9344 0.6953 0.5962 0.4693  0.4023  0.3084
3.5 0.9452 0.7233 0.6287 0.4976  0.4344  0.3329
```

and read by:

```
HEAT_EXCHANGER_COMPONENT( "radiator component" ) {
 effectiveness_curve_fit_values = Read( "effectiveness.fit" )
 ...
}
```

An effectiveness of type user_function allows for more complex behavior; see the *AcuSolve User-Defined Functions Manual* for a detailed description of user-defined functions.

For example, consider the case where the effectiveness is governed by a rational function of the fluid flow. Then the input command may be given by:

```
HEAT_EXCHANGER_COMPONENT( "UDF effectiveness radiator" ) {
    effectiveness_type           = user_function
    effectiveness_user_function  = "usrEffectivenessExample"
    effectiveness_user_values    = { 1,0,0, # flow direction
                                     0.6, # coefficient 1
                                     0.9, # coefficient 2
                                     1.0 } # coefficient 3
    ...
}
```

where the user-defined function "usrEffectivenessExample" may be implemented as follows:

```
#include "acusim.h"
```

```
#include "udf.h"
UDF_PROTOTYPE( usrEffectivenessExample ) ;                          /* function
 prototype */
Void usrEffectivenessExample (
    UdfHd           udfHd,                                          /* Opaque handle for
 accessing data */
    Real*           outVec,                                         /* Output vector */
    Integer         nItems,                                         /* Number of
 elements */
    Integer         vecDim                                          /* = 1 */
) {
    Integer         elem ;                                          /* an element
 counter */
    Real            aVel ;                                          /* axial velocity */
    Real            coef1 ;                                         /* rational function
 coefficient 1 */
    Real            coef2 ;                                         /* rational function
 coefficient 2 */
    Real            coef3 ;                                         /* rational function
 coefficient 3 */
    Real            xDir ;                                          /* x-component of
 direction vector */
    Real            yDir ;                                          /* y-component of
 direction vector */
    Real            zDir ;                                          /* z-component of
 direction vector */
    Real*           usrVals ;                                       /* user values */
    Real*           vel ;                                           /* velocity field */
    Real*           xVel ;                                          /* x-velocity */
    Real*           yVel ;                                          /* y-velocity */
    udfCheckNumUsrVals( udfHd, 6 ) ;                                /* check for error
 */
    usrVals         = udfGetUsrVals( udfHd ) ;                      /* get the user vals
 */
    xDir            = usrVals[0] ;                                  /* x-direction */
    yDir            = usrVals[1] ;                                  /* y-direction */
    zDir            = usrVals[2] ;                                  /* z-direction */
    coef1           = usrVals[3] ;                                  /* coeff. 1 */
    coef2           = usrVals[4] ;                                  /* coeff. 2 */
    coef3           = usrVals[5] ;                                  /* coeff. 3 */
    vel             = udfGetElmData( udfHd, UDF_ELM_VELOCITY ) ;    /* get velocity */
    xVel            = &vel[0*nItems] ;                              /* localized x-vel
 */
    yVel            = &vel[1*nItems] ;                              /* localized y-vel
 */
    zVel            = &vel[2*nItems] ; /* localized z-vel */
    for ( elem      = 0 ; elem < nItems ; elem++ ) {
        aVel        = xVel[elem] * xDir
                    + yVel[elem] * yDir
                    + zVel[elem] * zDir ;
    if ( aVel < 0 ) aVel = 0 ;
    outVec[elem] = (coef1 + coef2 * aVel) / (1. + coef3 * aVel) ;
    }
} /* end of usrEffectivenessExample() */
```

The dimension of the returned effectiveness vector, *outVec*, is the number of elements.

The *coolant_heat_reject_multiplier_function*, *coolant_flow_rate_multiplier_function*, *friction_multiplier_function*, and *effectiveness_multiplier_function* parameters may be used to uniformly scale the amount of heat rejected by the heat exchanger, coolant flow rate, friction factor and thermal effectiveness, respectively. The value of each of these parameters refers to the user given

name of a MULTIPLIER_FUNCTION command in the input file. For example, to impose a uniform friction factor as a function of time and a bilinear effectiveness as a different function of time, the following commands may be issued:

```
HEAT_EXCHANGER_COMPONENT( "radiator component" ) {
    shape                                       = three_node_triangle
    element_set                                 = "radiator"
    surfaces                                    = { 1, 101, 1, 9, 8 ;
                                                    2, 102, 8, 9, 5 ; }
    type                                        = constant_coolant_heat_reject
    coolant_heat_reject                         = 237443.5
    coolant_heat_reject_multiplier_function = none
    coolant_flow_rate                           = 4.6
    coolant_flow_rate_multiplier_function   = none
    direction                                   = { 1, 0, 0 }
    thickness                                   = 0.1
    upstream_distance                           = 0.05
    friction_type                               = constant
    friction                                    = 0.5
    friction_multiplier_function                = "time varying friction"
    inlet_min_area_ratio                        = 1.296
    outlet_min_area_ratio                       = 1.296
    transverse_friction_factor                  = 100
    effectiveness_type                          = piecewise_bilinear
    effectiveness_curve_fit_values              = {
                                                    0 ,  1.0000, 3.0000, 5.0000, 7.5000,
 10.000, 13.500 ;
                                                    0.4, 0.7487, 0.3302, 0.2511, 0.1731,
 0.1411, 0.0987 ;
                                                    1.1, 0.8877, 0.5422, 0.4301, 0.3021,
 0.2524, 0.1822 ;
                                                    1.7, 0.9123, 0.6321, 0.5332, 0.3923,
 0.3387, 0.2487 ;
                                                    2.6, 0.9344, 0.6953, 0.5962, 0.4693,
 0.4023, 0.3084 ;
                                                    3.5, 0.9452, 0.7233, 0.6287, 0.4976,
 0.4344, 0.3329 ;
    effectiveness_curve_fit_row_variable    = coolant_flow_rate
    effectiveness_curve_fit_column_variable = axial_velocity
    effectiveness_multiplier_function       = "time varying effectiveness"
}
MULTIPLIER_FUNCTION( "time varying friction" ) {
    type                                        = piecewise_linear
    curve_fit_values                            = { 0, 1.0 ;
                                                    10, 1.0 ;
                                                    20, 0.5 ;
                                                    40, 0.7 ;
                                                    80, 1.2 ; }
    curve_fit_variable                          = time
}
MULTIPLIER_FUNCTION( "time varying effectiveness" ) {
    type                                        = piecewise_linear
    curve_fit_values                            = { 0, 1.0 ;
                                                    10, 2.0 ;
                                                    20, 2.5 ;
                                                    40, 2.7 ;
                                                    80, 2.7 ; }
    curve_fit_variable                          = time
}
```

INTERFACE_SURFACE

Specifies an interface surface for a non-conformal mesh.

Type

AcuSolve Command

Syntax

```
INTERFACE_SURFACE("name") {parameters...}
```

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral
six_node_triangle or tri6	Six-node triangle.

element_set or elem_set (string) [no default]

User-given name of the parent element set.

surfaces (array) [no default]

List of element surfaces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this command. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to the INTERFACE_SURFACE command. This option is used in place of directly specifying these parameters. In the event that both of the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

split_internal_surfaces (boolean) [=off]

Option to automatically split the nodes on internal surfaces that require two boundary conditions, that is, cases that include baffles, sliding mesh or thermal interfaces.

gap_factor (real) >=0 [=1]

Nondimensional (with respect to the length of an element face) maximum gap allowed for two element faces to be in contact. No maximum if zero.

gap (real) >=0 [=0]

Dimensional maximum gap allowed for two element faces to be in contact. No maximum if zero.

crease_angle or *angle* *real >=0 <=180 [=90]*
: Maximum angle between face normals allowed for two element faces to be in contact. No maximum if zero.

external_heat_flux or *ext_heat* *(real) [=0]*
: The value of *external_heat_flux* is applied on each of the INTEFACE_SURFACE pairs. A positive value means that heat is added (source), while a negative value means that heat is removed (sink).

Description

This command defines an "interface surface" as a set of surfaces (element faces) with associated geometrical properties. Interface surfaces are used to connect pairs of elements that share (approximately) the same surface but are not conformal. That is, the corresponding element faces are not node-on-node but are in "contact." This allows "tied interface" and "sliding mesh" problems to be solved. If two interface surfaces are found to be in contact, a Discontinuous Galerkin formulation, as opposed to commonly used penalty methods, is used to form the residual on these surfaces. If an interface surface is not found to be in contact with any other interface surface, it may be subject to boundary conditions if *active_type* = no_interface is specified by a boundary condition command for this surface. The boundary condition commands that support this constraint are:

- NODAL_BOUNDARY_CONDITION
- ELEMENT_BOUNDARY_CONDITION
- SIMPLE_BOUNDARY_CONDITION
- TURBULENCE_WALL

See these commands for details on the *active_type* parameter.

AcuSolve internally combines all interface surfaces into one global set. Then each element face is checked against all the others for contact. This has three important implications:

- There are no "main" or "secondary" surfaces;
- You do not need to (and are not allowed to) specify pairs of interface surfaces that are to be checked for contact; and
- Element faces within a single interface surface set may be contact with each other.

The surfaces of an interface surface are defined with respect to the elements of an element set. For example,

```
ELEMENT_SET( "impeller region" ) {
    shape          = four_node_tet
    elements       = { ...
                       4, 2, 5, 6, 8 ;
                       5, 2, 6, 3, 5 ;
                       ... }
    ...
}
INTERFACE_SURFACE( "surface of impeller region" ) {
    shape          = three_node_triangle
    element_set    = "impeller region"
    surfaces       = { 4, 41, 2, 5, 6 ;
                       5, 51, 5, 6, 3 ; }
    gap_factor     = 0
    gap            = 0
    crease_angle   = 0
```

```
}
```

defines two surfaces of the element set "impeller region" as an interface surface with no constraints on finding contact surfaces.

There are two main forms of this command. The legacy version (or single topology version) of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number (of the parent element set), a unique (within this set) surface number, and the three nodes of the element face. For four_node_quad, *surfaces* have six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as region_surface.ebc:

```
4 41 2 5 6
5 51 5 6 3
```

and read by:

```
INTERFACE_SURFACE( "surface of impeller region" ) {
 shape = three_node_triangle
 element_set = "flow_elements"
 surfaces = Read( "region_surface.ebc" )
...
}
```

The mixed topology form of the INTERFACE_SURFACE command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the SURFACE_SET command and are then referred to by the INTERFACE_SURFACE command. For example, a collection of triangular and quadrilateral element faces can be defined using the following SURFACE_SET commands.

```
SURFACE_SET( "tri faces" ) {
    surfaces       = { 1, 1, 1, 2, 4 ;
                       2, 2, 3, 4, 6 ;
                       3, 3, 5, 6, 8 ; }
    shape          = three_node_triangle
    volume_set     = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
    surfaces       = { 1, 1, 1, 2, 4, 9 ;
                       2, 2, 3, 4, 6, 12 ;
                       3, 3, 5, 6, 8, 15 ; }
    shape          = four_node_quad
    volume_set     = "prisms"
```

Then, a single INTERFACE_SURFACE command is defined that contains the tri and quad faces as follows:

```
INTERFACE_SURFACE ( "surface of impeller region" ) {
    surface_sets       = {"tri_faces", "quad_faces"}
    ...
}
```

The list of surface sets can also be placed in a file, such as surface_sets.srfst:

```
tri faces
quad faces
```

and read using:

```
INTERFACE_SURFACE ( "surface of impeller region" ) {
    surface_sets       = Read("surface_sets.srfst")
    ...
}
```

The mixed topology version of the INTERFACE_SURFACE command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *surface_sets* and *surfaces* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

Residuals of the Discontinuous Galerkin formulation are formed at the quadrature points of the surfaces and integrated to the nodes of the element faces. The quadrature rule is inherited from the parent element set.

For each quadrature point of each interface surface element face, a search is made over all other such faces for the closest face. The original quadrature point is considered to be in contact with this face if three constraints are met:

- The distance between the point and the face is less than *gap*;
- The distance between the point and the face, normalized by the length of the face, is less than *gap_factor*; and
- The angle between the normals of the face and the original face is less than *crease_angle*.

Each constraint is ignored if the corresponding parameter is zero. If all three parameters are zero, then there are no constraints and each quadrature point is guaranteed to be in contact with another face.

To obtain the best solution quality at the non-conformal interface, at least one layer of structured mesh with uniform height and length on both sides of the interface should be created. As shown in the example below, each side of the interface between the blue and yellow domains has one layer of boundary layer mesh of identical grid size.

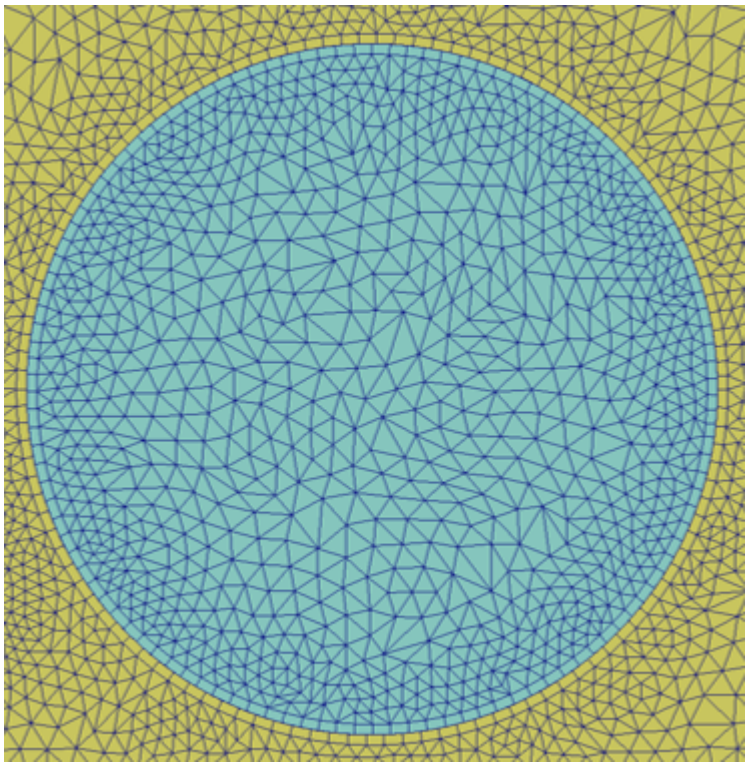


Figure 25:

INTERPOLATED_MOTION_SURFACE

Specifies a surface that is used to scale displacements of mesh nodes.

Type

AcuSolve Command

Syntax

INTERPOLATED_MOTION_SURFACE ("name") {parameters...}

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral
six_node_triangle or tri6	Six-node triangle.

element_set or *elem_set (string) [no default]*

User-given name of the parent element set.

surfaces (array) [no default]

List of element faces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this command. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to the INTERPOLATED_MOTION_SURFACE command. This option is used in place of directly specifying these parameters. In the event that both of the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

type (enumerated) [=faceted]

Type of the interpolated motion surface.

faceted	Interpolated motion surface is made up of the discrete points and faces contained within the input surfaces.

Description

This command specifies a faceted surface (a set of element faces) to be used in conjunction with a MESH_MOTION of *type* = interpolated. When using interpolated motion, each

INTERPOLATED_MOTION_SURFACE associated with a given MESH_MOTION command is collected into a single set. All nodes falling within the boundaries of that set are then interpolated to determine their weighted displacement based on the distance from their surrounding "driving" surfaces. For example, consider a simple 2D application consisting of a rigid cylinder embedded in a square domain with fictitious springs attached to it. As the flow passes over the cylinder, the forces cause the cylinder to translate in the stream wise and cross stream directions. Assigning a MESH_MOTION of *type* = interpolated to the nodes in the volume will linearly scale the displacement of the surrounding nodes as a function of distance between INTERPOLATED_MESH_MOTION surfaces. For example:

```
MESH_MOTION( "vibrating cylinder" ) {
    type                          = rigid_body_dynamic
    rigid_body_center             = { 0.05, 0.05, 0.00125; }
    rigid_body_mass               = 2.35619449e-007
    rigid_body_stiffness          = { 0.000654446111, 5.2355689e-005, 0.0, 0.0, 0.0,
 0.0; }
    rigid_body_surface_outputs    = { "cylinder surface" }
}
MESH_MOTION( "Interpolated Motion" ) {
    type                          = interpolated_motion
    interpolated_motion_surfaces  = { "cylinder surface", "box walls"}
}
INTERPOLATED_MOTION_SURFACE( "cylinder surface" ) {
    surfaces                      = Read( "MESH.DIR/cylinder_surface.ebc" )
    shape                         = four_node_quad
    element_set                   = "Air"
    type                          = faceted
}
INTERPOLATED_MOTION_SURFACE( "box walls" ) {
    surfaces                      = Read( "MESH.DIR/box_walls.ebc" )
    shape                         = four_node_quad
    element_set                   = "Air"
    type                          = faceted
}
ELEMENT_SET( "Air" ) {
    ...
    mesh_motion                   = "Interpolated Motion"
    mesh_motion_precedence        = 1
    ...
}
```

There are two main forms of this command. The legacy version (or single topology version) of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad

Element Shape	Surface Shape
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number (of the parent element set), a unique (within this set) surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as `walls.ebc`:

```
4 41 2 5 6
5 51 5 6 3
```

and read by:

```
INTERPOLATED_MOTION_SURFACE( "walls" ) {
    shape        = three_node_triangle
    element_set  = "flow_elements"
    surfaces     = Read( "region_surface.ebc" )
    ...
}
```

The mixed topology form of the `INTERPOLATED_MOTION_SURFACE` command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the `SURFACE_SET` command, and are then referred to by the `INTERPOLATED_MOTION_SURFACE` command. For example, a collection of triangular and quadrilateral element faces can be defined using the following `SURFACE_SET` commands.

```
SURFACE_SET( "tri faces" ) {
    surfaces    = { 1, 1, 1, 2, 4 ;
                    2, 2, 3, 4, 6 ;
                    3, 3, 5, 6, 8 ; }
    shape       = three_node_triangle
    volume_set  = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
    surfaces    = { 1, 1, 1, 2, 4, 9 ;
                    2, 2, 3, 4, 6, 12 ;
                    3, 3, 5, 6, 8, 15 ; }
```

```
    shape       = four_node_quad
    volume_set  = "prisms"
```

Then, a single `INTERPOLATED_MOTION_SURFACE` command is defined that contains the tri and quad faces as follows:

```
INTERPOLATED_MOTION_SURFACE ( "walls" ) {
     surface_sets       = {"tri_faces", "quad_faces"}
     ...
}
```

The list of surface sets can also be placed in a file, such as `surface_sets.srfst`:

```
tri faces
quad faces
```

and read using:

```
INTERPOLATED_MOTION_SURFACE ( "walls" ) {
    surface_sets       = Read("surface_sets.srfst")
    ...
}
```

The mixed topology version of the `INTERPOLATED_MOTION_SURFACE` command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *surface_sets* and *surfaces* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

LINE_SOURCE

Specifies sources on a line.

Type

AcuSolve Command

Syntax

LINE_SOURCE("name"){parameters...}

Qualifier

User-given name.

Parameters

lines (array) [no default]
A three-column array defining the line connectivity.

radius (real) >0 [=1]
Effective radius of the line.

friction_type (enumerated) [=none]
Type of the friction factor.

none	No friction.
constant or const	Constant for the entire set. Requires *friction*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *friction_curve_fit_values* and *friction_curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *friction_curve_fit_values* and *friction_curve_fit_variable*.

friction (real) >=0 [=1]
Constant value of the friction factor. Used with constant friction type.

friction_curve_fit_values (array) [={0,1}]
A two-column array of independent-variable/friction-factor data values. Used with piecewise_linear and cubic_spline friction types.

friction_curve_fit_variable (enumerated) [=normal_velocity]
Independent variable of the friction factor curve fit. Used with piecewise_linear and cubic_spline friction types.

normal_velocity	Velocity component normal to the line.

friction_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the friction factor. If none, no scaling is performed.

axial_friction_factor (real) >=0 [=10]
Fraction of the friction factor used for the axial friction factor.

volume_heat_source_type (enumerated) [=none]
Type of the heat source per unit volume.

none	No volume heat source.
constant or const	Constant for the entire set. Requires *volume_heat_source*.

volume_heat_source (real) >=0 [=1]
Constant value of the heat source per unit volume. Used with constant volume heat source type.

volume_heat_source_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the volume heat source. If none, no scaling is performed.

turbulence_source_type (enumerated) [=none]
Type of the turbulence source.

none	No turbulence source.
constant or const	Constant for the entire set. Requires *turbulence_source*.
piecewise_linear or linear	Piecewise linear curve fit. Requires *turbulence_source_curve_fit_values* and *turbulence_source_curve_fit_variable*.
cubic_spline or spline	Cubic spline curve fit. Requires *turbulence_source_curve_fit_values* and *turbulence_source_curve_fit_variable*.

turbulence_source (real) >=0 [=0]
Constant value of the turbulence source. Used with constant turbulence source type.

turbulence_source_curve_fit_values (array) [={0,1}]
A two-column array of independent-variable/turbulence-source data values. Used with piecewise_linear and cubic_spline turbulence source types.

turbulence_source_curve_fit_variable (enumerated) [=normal_velocity]
Independent variable of the turbulence source curve fit. Used with piecewise_linear and cubic_spline turbulence source types.

normal_velocity	Velocity component normal to the line.

turbulence_source_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the turbulence source. If none, no scaling is performed.

mesh_motion (string) [=none]
User-given name of the MESH_MOTION command for specifying mesh displacement boundary conditions on all the nodes in this line. If none, no boundary conditions are imposed.

Description

This command specifies momentum, heat, and turbulence sources on a line.

For example,

```
LINE_SOURCE( "example line source" ) {
   lines                                  = { 1001, 1, 2 ;
                                              1002, 3, 4 ;
                                              1003, 5, 6 ; }
   radius                                 = 1
   friction_type                          = constant
   friction                               = 1
   axial_friction_factor                  = 10
   friction_multiplier_function           = none
   volume_heat_source_type                = constant
   volume_heat_source                     = 2
   volume_heat_source_multiplier_function = none
   turbulence_source_type                 = constant
   turbulence_source                      = 0.01
   turbulence_source_multiplier_function  = none
   mesh_motion                            = none
}
```

creates sources along a line for the momentum, temperature, and turbulence equations. If any equation is not defined, then the corresponding source is ignored. The effective radius of the line is 1. The *lines* parameter is mandatory. It is a three-column array corresponding to a unique (within this set) line number, and the two nodes (in the global coordinates list) of a line segment. The lines may be read from a file. For the above example the lines may be placed in a file, such as `example.lin`:

```
1001      1         2
1002       3         4
1003      5         6
```

and read by:

```
LINE_SOURCE( "example line source" ) {
   lines         = Read( "example.lin" )
   ...
}
```

The force on the line is defined in terms of the pressure drop in the normal and axial directions:

$$\nabla p \bullet \boldsymbol{n} = -\rho f(u_n) u_n \left(\frac{2}{\pi R}\right)$$

$$\nabla p \bullet \boldsymbol{a} = -\rho f_a f(u_n) u_a \left(\frac{2}{\pi R}\right) \qquad (164)$$

where p is the fluid pressure; **a** is the direction parallel to the line; **n** is the direction of the fluid velocity projected to the plane normal to the line (that is, the "normal" direction); $f(u_n)$ is the friction factor, given by *friction_type* and related parameters (see below); $f_a f(u_n)$ is the axial friction factor; f_a is given by *axial_friction_factor*; R is the effective radius, given by radius; $u_n = \boldsymbol{u} \bullet \boldsymbol{n}$ is the normal velocity; $u_a = \boldsymbol{u} \bullet \boldsymbol{a}$ is the axial velocity; and **u** is the fluid velocity vector. The pressure drop is

integrated over the effective volume yielding a force on the fluid, which is applied only at the nodes on the line.

If *friction_type*=constant, the friction factor is given by *friction*.

A piecewise_linear or cubic_spline friction type may be used to define the friction factor as a function of a single independent variable. The only independent variable currently supported is the normal velocity. For example,

```
LINE_SOURCE( "curve fit friction factor" ) {
   friction_type               = piecewise_linear
   friction_curve_fit_values   = { 0, 10.685 ;
                                   5, 3.291 ;
                                  10, 1.996 ;
                                  15, 1.495 ;
                                  20, 1.220 ;
                                  30, 0.920 ;
                                  40, 0.756 ;
                                  50, 0.650 ; }
   friction_curve_fit_variable = normal_velocity
   ...
}
```

defines the friction factor as a function of the normal velocity (that is, velocity in the direction of the heat exchanger). The *friction_curve_fit_values* parameter is a two-column array corresponding to the independent variable and the friction factor. The independent variable values must be in ascending order. The limit point values of the curve fit are used when *friction_curve_fit_variable* falls outside of the curve fit limits.

The *friction_curve_fit_values* data may be read from a file. For the above example, the curve fit values may be placed in a file, such as `friction.fit`:

```
0    10.685
5     3.291
10    1.996
15    1.495
20    1.220
30    0.920
40    0.756
50    0.650
```

and read by:

```
LINE_SOURCE( "curve fit friction factor" ) {
   friction_type               = piecewise_linear
   friction_curve_fit_values   = Read( "friction.fit" )
   friction_curve_fit_variable = normal_velocity
   ...
}
```

A volumetric heat source for the temperature equation is given by *volume_heat_source*. This has the same units and effect as the `VOLUME_HEAT_SOURCE` command. The only difference is that here the heat source is integrated over the effective volume and applied only on the nodes of the line.

For the Spalart-Allmaras and Detached Eddy Simulation turbulence models, a turbulence line source is given by:

$$s = f_t(u_n)u_n^2\left(\frac{2}{\pi R}\right) \quad (165)$$

where $f_t(u_n)$ is the turbulence source, given by *turbulence_source_type* and related parameters; and s is an effective volumetric source term in the turbulence model equation. The latter is integrated over the effective volume and applied only on the nodes of the line.

If *turbulence_source_type*=constant, the turbulence source is given by *turbulence_source*. A piecewise_linear or cubic_spline turbulence source type may be used to define the turbulence source as a function of the normal velocity.

The *friction_multiplier_function*, *volume_heat_source_multiplier_function* and *turbulence_source_multiplier_function* parameters may be used to scale the friction factor, volume heat source, and turbulence source, respectively. The value of each of these parameters refers to the user given name of a MULTIPLIER_FUNCTION command in the input file.

The *mesh_motion* parameter is provided as a shortcut way to impose nodal boundary conditions on all three components of the mesh displacement. The parameter refers to a MESH_MOTION command where the motion is defined. Specifying *mesh_motion* is equivalent to providing three NODAL_BOUNDARY_CONDITION commands on the components of the mesh displacement, with each command referring to all the nodes in the line connectivity, setting *precedence*=0, and including the same *mesh_motion* parameter. If *mesh*=eulerian in the EQUATION command or *mesh_motion*=none then this parameter has no effect. For example, a line that rotates with a fan may be defined by:

```
LINE_SOURCE( "rotating fan" ) {
   mesh_motion        = "rotating fan"
   ...
}
MESH_MOTION( "rotating fan" ) {
   type               = rotation
   rotation_center    = { 0, 0, 0 }
   angular_velocity   = { 0, 3, 0 }
}
```

MESH_BOUNDARY_CONDITION

Specifies element and nodal boundary conditions for mesh displacement for a set of element faces.

Type

AcuSolve Command

Syntax

```
MESH_BOUNDARY_CONDITION("name") { parameters...}
```

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral
six_node_triangle or tri6	Six-node triangle.

element_set or elem_set (string) [no default]

User-given name of the parent element set.

surfaces (array) [no default]

List of element surfaces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this command. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to the MESH_BOUNDARY_CONDITION command. This option is used in place of directly specifying these parameters. In the event that both of the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

type (enumerated) [=fixed]

Type of displacement for the mesh on this surface..

fixed	Mesh fixed to the wall.
planar_slip	Mesh slips tangentially along the wall of a planar surface. An error is issued if the surface is not planar.
free_surface	Free surface.

mesh_motion	Mesh follows a mesh motion. Requires *mesh_motion*.
guide_surface	Mesh follows a guide surface. Requires *guide_surface*.
flexible_body	Mesh follows a flexible body. Requires *flexible_body*.
external_code	Mesh follows the coupled external code surface.

surface_tension_model *(string) [=none]*
User-given name of the surface tension model. If none, surface tension is not modeled. Used with *type* = free_surface.

contact_angle_model *(string) [=none]*
User-given name of the contact angle model. If none, default contact angle is used. Used with *type* = free_surface.

flexible_body *(string) [=none]*
User-given name of the FLEXIBLE_BODY command. Used with *type* = flexible_body.

nodal_x_modes *(array) [no default]*
Array of node number (first column) and x-component eigenvector values (last num_modes columns, where num_modes is given by the FLEXIBLE_BODY command referenced by *flexible_body*). Nodes must match those referenced in the surfaces. Used with *type* = flexible_body.

nodal_y_modes *(array) [no default]*
Array of node number (first column) and y-component eigenvectors values (last num_modes columns, where num_modes is given FLEXIBLE_BODY command referenced by *flexible_body*). Nodes must match those referenced in the surfaces. Used with *type* = flexible_body.

nodal_z_modes *(array) [no default]*
Array of node number (first column) and z-component eigenvectors values (last num_modes columns, where num_modes is given FLEXIBLE_BODY command referenced by *flexible_body*). Nodes must match those referenced in the surfaces. Used with *type* = flexible_body.

guide_surfaces *(list) [={}]*
List of user-given names of the GUIDE_SURFACE commands that the nodes of this MESH_BOUNDARY_CONDTION will be forced to track along. Used with *type* = guide_surface.

mesh_motion *(string) [=none]*
User-given name of the MESH_MOTION command for determining mesh displacement. Used with *type* = mesh_motion and *type* = flexible_body.

coupling_type *(enumerated) [=structural]*
Manner in which AcuSolve surfaces are coupled with the external code. Used with *type* = external_code.

structural	AcuSolve surface is coupled with either a line or surface element set in the external code. This option requires *mesh_coupling_type* = tied when coupling to line elements.

line	AcuSolve surface is coupled with a line element set in the external code. This option requires *mesh_coupling_type* = tied.
surface	AcuSolve surface is coupled with a surface element set in the external code.
rigid_body	AcuSolve surface is coupled with a rigid body in the external code.

coupling_direction (enumerated) [=both]
External code coupling direction for the surface. Used with *type* = external_code.

both	Exchanges information with the external code object in both directions. AcuSolve sends loading information to the external code and the external code sends displacements back to AcuSolve.
to_external_code	AcuSolve sends loading information to the external code, but no displacement information is returned.
from_external_code	AcuSolve does not send loading information to the external code, but displacement information is provided to AcuSolve at each step.

mesh_coupling_type (enumerated) [= tied]
Type of the mesh displacement boundary condition. Used with *type* = external_code.

tied	Fluid mesh tied to the solid.
slip	Fluid mesh slides on the solid.

external_code_tags (list) [={}]
Array of strings used to associate this fluid surface to a given solid surface. The null list matches to all solid surface and beams. Used with *type* = external_code.

rigid_body_name (string) [no default]
Name of the rigid body the external code surface is coupled with. Used with *coupling_type* = rigid_body and *type* = external_code.

gap_factor (real) >=0 [=1]
Non-dimensional (with respect to the length of an element face) maximum gap allowed for two element faces to be in contact. No maximum if zero.

gap (real) >= 0 [=]
Dimensional maximum gap allowed for two element faces to be in contact. No maximum if zero.

precedence (integer) [1]
Precedence of the nodal boundary condition defined here with respect to others elsewhere. Highest value has precedence on a per-variable basis.

Description

This command specifies the appropriate nodal and element boundary conditions for mesh displacement on a set of surfaces (element faces) for the given type of boundary. "Nodal boundary conditions on a surface" is shorthand for "nodal boundary conditions on all the nodes that comprise the surface." The surfaces of an element boundary condition are defined with respect to the elements of an element set. For example,

```
ELEMENT_SET( "flow elements" ) {
    shape          = four_node_tet
    elements       = { ...
                       4, 2, 5, 6, 8 ;
                       5, 2, 6, 3, 5 ;
                       ... }
    ...
}
MESH_BOUNDARY_CONDITION( "slip wall" ) {
    shape          = three_node_triangle
    element_set    = "flow elements"
    surfaces       = { 4, 41, 2, 5, 6 ;
                       5, 51, 5, 6, 3 ; }
    type = planar_slip
}
```

defines all the necessary boundary conditions for mesh displacement such that the nodes are allowed to slip tangentially along two surfaces of the element set "flow elements".

There are two main forms of this command. The legacy version (or single topology version) of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number (of the parent element set), a unique (within this set) surface number, and the three nodes of the element face. For four_node_quad, *surfaces*

has six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as `lower_wall.srf`:

```
4 41 2 5 6
5 51 5 6 3
```

and read by:

```
MESH_BOUNDARY_CONDITION( "slip wall" ) {
   shape        = three_node_triangle
   element_set  = "flow elements"
   surfaces     = Read( "lower_wall.srf" )
   type         = planar_slip
}
```

The mixed topology form of the `MESH_BOUNDARY_CONDITION` command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the `SURFACE_SET` command, and are then referred to by the `MESH_BOUNDARY_CONDITION` command. For example, a collection of triangular and quadrilateral element faces can be defined using the following `SURFACE_SET` commands.

```
SURFACE_SET( "tri faces" ) {
    surfaces       = { 1, 1, 1, 2, 4 ;
                       2, 2, 3, 4, 6 ;
                       3, 3, 5, 6, 8 ; }
    shape          = three_node_triangle
    volume_set     = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
    surfaces       = { 1, 1, 1, 2, 4, 9 ;
                       2, 2, 3, 4, 6, 12 ;
                       3, 3, 5, 6, 8, 15 ; }
    shape          = four_node_quad
    volume_set     = "prisms"
```

Then, a single `MESH_BOUNDARY_CONDITION` command is defined that contains the tri and quad faces as follows:

```
MESH_BOUNDARY_CONDITION ( "slip wall" ) {
   surface_sets       = {"tri_faces", "quad_faces"}
   ...
}
```

The list of surface sets can also be placed in a file, such as `surface_sets.srfst`:

```
tri faces
quad faces
```

and read using:

```
MESH_BOUNDARY_CONDITION ( "slip wall" ) {
   surface_sets       = Read("surface_sets.srfst")
   ...
}
```

The mixed topology version of the MESH_BOUNDARY_CONDITION command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *surface_sets* and *surfaces* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

The MESH_BOUNDARY_CONDITION command centralizes the options associated with mesh movement from the SIMPLE_BOUNDARY_CONDITION, FREE_SURFACE, and EXTERNAL_CODE commands. This command is intended to act as the primary mechanism for specifying control of the mesh movement. For backward compatibility, the mesh movement related options from the SIMPLE_BOUNDARY_CONDITION, FREE_SURFACE, and EXTERNAL_CODE commands are still supported. AcuPrep internally creates a table of constraints for each element face/node on the boundaries and identifies conflicts within the mesh motion boundary conditions. These conflicts may arise due to by the combinations of MESH_BOUNDARY_CONDITION and the legacy commands that provide similar functionality. In the case of conflicting constraints that have equal precedence values, the MESH_BOUNDARY_CONDITION is given priority. If the precedence values are not equal, then the higher precedence constraint is enforced. AcuPrep also checks for compatibility of the mesh motion boundary conditions on each node with the boundary conditions applied for the physics through the SIMPLE_BOUNDARY_CONDITION command. The following table is used when checking these compatibilities:

SIMPLE_BC type	**MESH_BOUNDARY_CONDITION type**						
-	fixed	planar_slip	mesh_motio	free_surface	guide_surfac	flexible_bod	external_code
wall	OK	OK	OK	Error	OK	OK	OK
inflow	OK	OK	OK	Error	OK	Warning	Warning
outflow	OK	OK	OK	Error	OK	Warning	Warning
slip	OK	OK	OK	Error	OK	OK	OK
symmetry	OK	OK	Error	Error	Error	Error	Error
far_field	OK	OK	OK	Error	OK	Warning	Warning
free_surface	Error	Error	Error	OK	Error	Error	Error

The *type* parameter of the MESH_BOUNDARY_CONDITION determines the behavior of mesh on the given surface. To prevent any motion of the mesh on the surface, *type* = fixed is used. Other behaviors may be obtained by setting *type* to free_surface, guide_surface, flexible_body, mesh_motion or

external_code. The behavior of a free surface wave, including surface tension and a contact angle model, can be simulated by using the *type* = free_surface as follows:

```
MESH_BOUNDARY_CONDITION( "water surface" ) {
    surface_sets          = {"tri faces","quad faces"}
    type                  = free_surface
    surface_tension_model = "water/air"
    contact_angle_model   = "water/air @ smooth aluminum"
}
```

For the case of a simulation involving free surface waves impinging against the moving hull of a ship, it is necessary to ensure that the water touching the ship follows the profile of the ship's hull as it moves in the ocean. This requires the use of *type* = guide_surface. When using this option, the nodes slip along the guide surfaces given by the *guide_surfaces* parameter. The movement of the guide surface itself is then controlled by a MESH_MOTION command of *type* = rigid_body. For example:

```
MESH_MOTION( "boat motion" ) {
    type               = rigid_body_dynamic
    rigid_body_mass    = 100
    ...
}
GUIDE_SURFACE( "ship hull" ) {
    surface_sets       = {"tri faces","quad faces"}
    type               = faceted
    mesh_motion        = "boat motion"
}
MESH_BOUNDARY_CONDITION( "water/ship hull" ) {
    ...
    type               = guide_surface
    guide_surfaces     = {"ship hull"}
}
```

A *mesh_displacement_type* = flexible_body is used to specify mesh motion of a deforming surface as part of a Practical Fluid-Structure Interaction (P-FSI) problem. For example, the FLEXIBLE_BODY command can be used to define the parameters of a structure undergoing linear deformation. The deformation of the structure is then transferred to the wetted surfaces of the fluid using MESH_BOUNDARY_CONDITION. This is illustrated in the following example:

```
FLEXIBLE_BODY( "flexible body" ) {
 equation      = mesh_displacement
 num_modes     = 5
 ...
}
MESH_BOUNDARY_CONDITION( "flexible surface" ) {
 ...
 type          = flexible_body
 flexible_body = "flexible body"
 nodal_x_modes = Read( "channel.wet_surf.xmd" )
 nodal_y_modes = Read( "channel.wet_surf.ymd" )
 nodal_z_modes = Read( "channel.wet_surf.zmd" )
}
```

where the files channel.wet_surf.xmd, channel.wet_surf.ymd, and channel.wet_surf.zmd correspond to the nodal modes in the x, y, and z-directions. The format of these files is the same as *nodal_modes* in the NODAL_BOUNDARY_CONDITION command. See also the FLEXIBLE_BODY command for details. The nodes in these files must match those referenced in the MESH_BOUNDARY_CONDITION. If the

problem has a deforming mesh (that is, mesh is set to arbitrary_lagrangian_eulerian or specified in the EQUATION command), and *type* = flexible_body and *wall_velocity_type* = match_mesh_velocity on a given set of nodes (the default), then the given velocity conditions are ignored. Instead, the velocity is set to match the velocity of the mesh.

Note: AcuPev may be used to obtain the nodal mode files.

Mesh displacement boundary conditions may also be specified with *type* = mesh_motion. The value of this parameter refers to the user-given name of a MESH_MOTION command in the input file. This command is evaluated, resulting in a mesh displacement vector. The *mesh_motion* parameter has an effect only if there is an active mesh equation and type is set to mesh_motion or flexible_body . The same parameter may be specified in the parent ELEMENT_SET command, but the one here takes precedence unless precedence is set to a value less than zero. For example,

```
MESH_MOTION( "moving body" ) {
    type        = rigid_body
    ...
}
MESH_BOUNDARY_CONDITION( "moving wall" ) {
    ...
    type        = mesh_motion
    mesh_motion = "moving body"
}
```

A flexible body may be combined with a rotating mesh. To model a deformable impeller with sliding interfaces (for example a wind turbine) the following commands may be used:

```
FLEXIBLE_BODY( "Blade A" ) {
    equation          = mesh_displacement
    ...
    surface_outputs   = { "Blade A" }
}
MESH_MOTION( "Rotation" ) {
    type              = rotation
    ...
    rotation_variable = time
}
MESH_BOUNDARY_CONDITION( "Blade A" ) {
    ...
    type = flexible_body
    flexible_body     = "Blade A"
    nodal_x_modes     = Read( "BladeA.xmd" )
    nodal_y_modes     = Read( "BladeA.ymd" )
    nodal_z_modes     = Read( "BladeA.zmd" )
    mesh_motion       = "Rotation"
}
```

The *type* = external_code option allows AcuSolve to extract the displacement of the surface from an external code. For example, to model a flexible surface that is part of a directly couple Fluid-Structure Interaction problem (DC-FSI), the following commands would be applied:

```
EXTERNAL_CODE{
    communication      = socket
    socket_initiate    = off
    socket_host        = "localhost"
    socket_port        = 10000
```

```
    coupling_type      = structural
}
MESH_BOUNDARY_CONDITION( "Deforming wall" ) {
    ...
    type               = external_code
    coupling_type      = structural
    gap                = 0
    gap_factor         = 0
    external_code_tags = {}
}
```

The fluid forces, moments, and heat fluxes are communicated from all of the surfaces defined in the MESH_BOUNDARY_CONDITION commands to an external solid/structural code defined by the EXTERNAL_CODE command. Mesh displacements and temperatures are returned from the external code to the same surfaces. For each quadrature point of each external code surface element face, a search is made over all appropriate solid model faces in the external code. The surfaces are assumed to be in contact if the following conditions are met:

- The distance between the point and the face is less than *gap*; and
- The distance between the point and the face, normalized by the length of the face, is less than *gap_factor*.

The *gap*/*gap_factor* constraint is ignored if the corresponding parameter is zero. If both parameters are zero, then there are no constraints and each quadrature point is guaranteed to be in contact with another face. Fluid forces are actually applied to the solid model only if the fluid quadrature point is determined to be in contact with a solid face. Similarly, nodal boundary conditions from the solid model are applied only if the solid model node is in contact with the external code surface.

The search over solid surfaces may be constrained through the use of tags. For example,

```
MESH_BOUNDARY_CONDITION( "Riser" ) {
    ...
    type               = external_code
    coupling_type      = structural
    gap                = 0
    gap_factor         = 0
    external_code_tags = {"Riser outer wall"}
}
MESH_BOUNDARY_CONDITION( "Drill" ) {
    ...
    type               = external_code
    coupling_type      = structural
    gap                = 0
    gap_factor         = 0
    external_code_tags = {}
}
```

In this case, the "Riser" surface will only be matched to solid surfaces tagged with "Riser outer wall" while the "Drill" surface can be matched with anything.

A *coupling_type* = structural indicates that all of the surfaces participating in the exchange between AcuSolve and the external code are of a single, consistent element type (that is line/beam elements or surface elements). For example, if a simulation contains five MESH_BOUNDARY_CONDITION, *type* = external_code commands, they must all be coupled to line elements or surface elements in the structural model. Prior to the release of AcuSolve 13.0, it was only possible to couple with a single type and this option is therefore supplied for backward compatibility. Starting in AcuSolve 13.0, it is possible

to couple with surface elements, line elements and rigid body surfaces within a single simulation. This is accomplished by explicitly defining the *coupling_type* parameter for each MESH_BOUNDARY_CONDITION of *type* = external_code command that is issued.

> **Note:** If a *coupling_type* = structural is used in the same simulation in which a *coupling_type* = line, *coupling_type* = structural or *coupling_type* = rigid_body is specified, an assertion will be issued and the code will exit.

This *type* = external_code option determines all of the boundary conditions on the wetted surface (that is, all the points that are determined to be in contact) if precedence is sufficiently high. The parameter *mesh_displacement_type* may be either slip, allowing the mesh to slide against the solid (such as in a half-filled tank); or tied, where the fluid mesh displacement is tied to the solid. For slip, the solid surface acts just like a guide surface; see the description of guide_surface type in the NODAL_BOUNDARY_CONDITION command for details about the algorithm used.

PARTICLE_SURFACE

Specifies a surface for particle tracking.

Type

AcuSolve Command

Syntax

PARTICLE_SURFACE("name") {parameters...}

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral
six_node_triangle or tri6	Six-node triangle.

element_set or *elem_set (string) [no default]*

User-given name of the parent element set.

surfaces (array) [no default]

List of element surfaces.

type (enumerated) [=wall]

Type of the boundary surface.

wall	Impenetrable wall.
inflow or in	Inflow surface. Currently not used..
outflow or out	Outflow surface. Currently not used.

Description

This command specifies the surfaces (element faces) that comprise a particular kind of surface for particle tracking. Currently particle tracking is done as part of a post-processing module. The only type supported is wall, which is used by the particle tracker to prevent particles from penetrating the given surface. It is not necessary to use a PARTICLE_SURFACE command for surfaces specified in a SIMPLE_BOUNDARY_CONDITION command with *type* = wall, *type* = slip, or *type* = symmetry.

The particle surfaces are defined with respect to the elements of an element set. For example,

```
ELEMENT_SET( "interior" ) {
    shape        = four_node_tet
    elements     = { 1, 8, 3, 4, 9 ;
```

```
                      3, 3, 4, 9, 5 ;
                      ... }
    ...
}
PARTICLE_SURFACE( "wall" ) {
    shape        = three_node_triangle
    element_set  = "interior"
    surfaces     = { 1, 12, 9, 3, 4 ;
                     3, 52, 5, 3, 4 ; }
    type         = wall
}
```

specifies that two surfaces of the element set "interior" are on a wall and that no particles are allowed to penetrate this wall.

There are two main forms of this command. The legacy version (or single topology version) of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number (of the parent element set), a unique (within this set) surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as wall.srf:

```
1 12 9 3 4
3 52 5 3 4
```

and read by:

```
PARTICLE_SURFACE ( "no-slip wall" ) {
   shape       = three_node_triangle
   element_set = "interior"
   surfaces    = Read( "free.srf" )
   ...
}
```

The mixed topology form of the PARTICLE_SURFACE command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the SURFACE_SET command, and are then referred to by the PARTICLE_SURFACE command. For example, a collection of triangular and quadrilateral element faces can be defined using the following SURFACE_SET commands.

```
SURFACE_SET( "tri faces" ) {
    surfaces    = { 1, 1, 1, 2, 4 ;
                    2, 2, 3, 4, 6 ;
                    3, 3, 5, 6, 8 ; }
    shape       = three_node_triangle
    volume_set  = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
    surfaces    = { 1, 1, 1, 2, 4, 9 ;
                    2, 2, 3, 4, 6, 12 ;
                    3, 3, 5, 6, 8, 15 ; }
    shape       = four_node_quad
    volume_set  = "prisms"
```

Then, a single PARTICLE_SURFACE command is defined that contains the tri and quad faces as follows:

```
PARTICLE_SURFACE ( "no-slip wall" ) {
    surface_sets       = {"tri_faces", "quad_faces"}
    ...
}
```

The list of surface sets can also be placed in a file, such as surface_sets.srfst:

```
tri faces
quad faces
```

and read using:

```
PARTICLE_SURFACE ( "no-slip wall" ) {
    surface_sets       = Read("surface_sets.srfst")
    ...
}
```

The mixed topology version of the PARTICLE_SURFACE command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *surface_sets* and *surfaces* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

A wall type specifies that this surface is impenetrable by particles. This specification is necessary because small numerical errors in particle tracking, or physical effects like inertia, may lead to particles intersecting a wall and becoming lost.

Specification of a particle surface via this command has no effect on the execution of AcuSolve. The only purpose of this command is to provide geometrical information to post-processing modules that perform particle tracking.

RADIATION_SURFACE

Specifies radiative boundary conditions for the enclosure, *p1_model* and *discrete_ordinate* radiation models.

For the enclosure model, a radiation heat flux condition is applied to the surface. For the *p1_model* the condition specifies the emissivity of the surface used by the Marshak boundary condition. For the *discrete_ordinate* model, the condition specifies surface properties, such as emissivity and diffuse fraction. The emissivity is used to account for radiation emission from the surface. For example, $\varepsilon\sigma T_w^4/\pi$, where ε is the emissivity, σ is Stefan-Boltzmann constant, and T_w is the wall temperature.

Type

AcuSolve Command

Syntax

```
RADIATION_SURFACE("name") {parameters...}
```

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral
six_node_triangle or tri6	Six-node triangle.

element_set or elem_set (string) [no default]

User-given name of the parent element set.

surfaces (array) [no default]

List of element surfaces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this command. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to the RADIATION_SURFACE command. This option is used in place of directly specifying these parameters. In the event that both of the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

type (enumerated) [=wall]
Type of the boundary surface. For the enclosure and *p1_model* radiation models, the type can be either wall or opening. For the *discrete_ordinate* radiation model the type can be wall, opening or radiation_interface.

auto	Automatic radiation surface treatment to determine whether a surface is treated as type = wall or type = opening. Used with all radiation models.
wall	Wall. Requires an *emissivity_model* for all radiation models and agglomeration for enclosure.
opening	Opening. Requires *emissivity_model* and *opening_temperature*.
radiation_interface	Radiation_interface. Enables the transmission and reflection of radiative intensity at an interface between two participating media. Available when *radiation* = discrete_ordinates.

radiation_interface_type (enumerated) [=internal]
The type of radiation_interface is specified through *radiation_interface_type*. The parameter has two values: internal and external. If *radiation_interface_type* = internal, there must be a participating media on each side of an interface, that is, the material for each adjacent ELEMENT_SET has a radiation_model. For this condition, the transfer of radiative intensity is both transmitted and reflected at the interface. If *radiation_interface_type* = external, there is a participating media on one side of an interface. The radiative intensity in the medium surrounding the model is calculated using the mathematical model. The external radiation interface is used with *external_emissivity_model* and *external_temperature*. The *radiation_interface_type* requires radiation_interface.

external_emissivity_model (string) [=none]
User-given name of the external emissivity model of an exterior facing surface if *radiation_interface_type* = external.

external_temperature (real)>=0 [= 273.15]
Defines the temperature of the fluid surrounding the domain if *radiation_interface_type* = external.

external_temperature_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the external temperature. If none, no scaling is performed.

specular_ordinate_averaging (enumerated) [=one_ordinate]
The specular ordinate averaging parameter is used to determine the direction of the specular ordinate. Two averging methods are available: one_ordinate and three_ordinates. If *specular_ordinate_averaging* = one_ordinate, this method is to search for the closest specular ordinate direction, if *specular_ordinate_averaging* = three_ordinates, the specular ordinate direction is calculated by averaging the three closest ordinate directions. This parameter is used for specular interfaces when the diffusion fraction is less than 1.0. Requires *radiation_interface*. Available when *radiation* = discrete_ordinates.

emissivity_model (string) [no default]
User-given name of the emissivity model. Used with wall and opening types.

opening_temperature or temp (real) >=0 [=273.15]
Opening temperature. Used with opening type.

opening_temperature_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the opening temperature. If none, no scaling is performed. Used with opening type.

agglomeration (boolean) [=on]
Flag specifying whether to agglomerate the surface elements. Used with wall type. A value of on requires *max_agglomeration_surfaces*, *max_agglomeration_angle* and *max_agglomeration_radius*.

max_agglomeration_surfaces (integer) >=0 [=25]
Maximum number of surfaces in one agglomeration. Used with on agglomeration. If zero, this option is ignored.

max_agglomeration_angle or angle (real) >=0 <=180 [=10]
Maximum angle between surfaces allowed in any agglomeration. Used with on agglomeration.

max_agglomeration_radius or radius (real) >=0 [=0.25]
Maximum radius of agglomeration. Used with on agglomeration. If zero, this option is ignored.

integrated_output_frequency or intg_freq (integer) >=0 [=1]
Time step frequency at which to output the integrated radiation heat flux. If zero, this option is ignored.

integrated_output_time_interval or intg_intv (real) >=0 [=0]
Time frequency at which to output the integrated radiation heat flux. If zero, this option is ignored.

nodal_output_frequency or nodal_freq (integer) >=0 [=0]
Time step frequency at which to output radiation heat flux at the nodes of the surface. If zero, this option is ignored.

nodal_output_time_interval or nodal_intv (real) >=0 [=0]
Time frequency at which to output radiation heat flux at the nodes of the surface. If zero, this option is ignored.

diffused_fraction (real) [=1.0]
Diffused fraction defines the proportion of reflected radiation intensity at a surface that is diffused.

Description

This command specifies a radiation heat flux condition on a set of surfaces (element faces). This condition is coupled to all other radiation surfaces. The RADIATION command provides a detailed description of this coupling.

The surfaces of a radiation surface are defined with respect to the elements of an element set. For example,

```
ELEMENT_SET( "interior" ) {
    shape                       = four_node_tet
    elements                    = { 1, 8, 3, 4, 9 ;
```

```
                                    3, 3, 4, 9, 5 ;
                                    ... }
    ...
}
RADIATION_SURFACE( "wall" ) {
    type                          = wall
    shape                         = three_node_triangle
    element_set                   = "interior"
    surfaces                      = { 1, 12, 9, 3, 4 ;
                                      3, 52, 5, 3, 4 ; }
    emissivity_model              = "emissivity"
    integrated_output_frequency   = 2
}
```

specifies a radiation heat flux condition to be applied to two surfaces of the element set "interior" using the emissivity model "emissivity", and the integral of the radiation heat flux is to be output every two steps.

There are two main forms of this command. The legacy version (or single topology version) of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For *three_node_triangle*, this parameter has five columns, corresponding to the element number, of the parent element set, a unique, within this set surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as `wall.srf`:

```
1 12 9 3 4
3 52 5 3 4
```

and read by:

```
RADIATION_SURFACE ( "no-slip wall" ) {
    shape        = three_node_triangle
    element_set  = "interior"
    surfaces     = Read( "wall.srf" )
    ...
}
```

The mixed topology form of the `RADIATION_SURFACE` command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *`surface_sets`* parameter. The element faces are first created in the input file using the `SURFACE_SET` command, and are then referred to by the `RADIATION_SURFACE` command. For example, a collection of triangular and quadrilateral element faces can be defined using the following `SURFACE_SET` commands.

```
SURFACE_SET( "tri faces" ) {
   surfaces       = { 1, 1, 1, 2, 4 ;
                      2, 2, 3, 4, 6 ;
                      3, 3, 5, 6, 8 ; }
   shape          = three_node_triangle
   volume_set     = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
   surfaces       = { 1, 1, 1, 2, 4, 9 ;
                      2, 2, 3, 4, 6, 12 ;
                      3, 3, 5, 6, 8, 15 ; }
   shape          = four_node_quad
   volume_set     = "prisms"
```

Then, a single `RADIATION_SURFACE` command is defined that contains the tri and quad faces as follows:

```
RADIATION_SURFACE ( "no-slip wall" ) {
    surface_sets       = {"tri_faces", "quad_faces"}
    ...
}
```

The list of surface sets can also be placed in a file, such as `surface_sets.srfst`:

```
tri faces
quad faces
```

and read using:

```
RADIATION_SURFACE ( "no-slip wall" ) {
   surface_sets       = Read("surface_sets.srfst")
   ...
}
```

The mixed topology version of the RADIATION_SURFACE command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *surface_sets* and *surfaces* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

For the enclosure model, all data from all RADIATION_SURFACE commands are pre-processed to form view factors. Since it is usually too expensive to store and process radiation heat transfer between a pair of each element faces, the surface data are "agglomerated" to reduce the number of view factors. Several parameters are provided to help control the agglomeration. For example,

```
RADIATION_SURFACE( "wall" ) {
    ...
    type                          = wall
    emissivity_model              = "emissivity"
    agglomeration                 = on
    max_agglomeration_surfaces    = 100
    max_agglomeration_angle       = 20
    max_agglomeration_radius      = 0.5
    diffused_fraction             = 0.9
}
```

specifies that each agglomeration contains no more than 100 surfaces, the angles between the outward normals of these surfaces are no more than 20 degrees, and the radius of the agglomerated surface no greater than 50 percent of the radius of the entire surface set. In addition to these constraints, each agglomeration must contain only one emissivity model. Several parameters must be the same across all radiation surfaces; these are given in the RADIATION command. When accuracy is more important than the cost of computing the view factors, for example, a small, hot surface, *agglomeration* should be set to off. In this case the radiation heat flux will be computed for each element face in the set.

The opening type provides a method of fully enclosing a fluid domain that is not be completely surrounded by walls. This type is appropriate for inlets, outlets, and surfaces that approximate infinity in external flows. The primary assumption is that the surface is at a single given temperature of *opening_temperature*. This assumption allows the entire set to be combined into one agglomerated facet, so *agglomeration* and associated parameters are ignored. An opening is typically modeled as a black body, with an emissivity of one. However, other emissivity models may be used with an opening.

The *opening_temperature_multiplier_function* parameter may be used to scale the opening temperature. For example,

```
RADIATION_SURFACE( "inlet" ) {
    ...
    type                                    = opening
    emissivity_model                        = "black body"
    opening_temperature                     = 1
    opening_temperature_multiplier_function = "inlet temperature"
}
MULTIPLIER_FUNCTION( "inlet temperature" ) {
    type                                    = cubic_spline
    curve_fit_values                        = { 0, 295 ;
                                                12, 312 ;
                                                24, 320 ; }
    curve_fit_variable                      = time
}
```

```
EMISSIVITY_MODEL( "black body" ) {
    type                                = constant
    emissivity                          = 1
}
```

If either *integrated_output_frequency* or *integrated_output_time_interval* is non-zero, the surface integral of the radiation heat flux will be output at the end of the run. If both are zero, no integrated radiation heat flux data is written to disk.

Similarly, if either *nodal_output_frequency* or *nodal_output_time_interval* is non-zero, the nodal values of the radiation heat flux will be output at the end of the run. If both are zero, no nodal radiation heat flux data is written to disk.

Run times may not coincide with *integrated_output_time_interval* or *nodal_output_time_interval*. In these cases, the corresponding data are output for every time step which passes through a multiple of *output_time_interval* or *nodal_output_time_interval*.

Once the surface quantities have been written to disk, they can be translated to other formats using the AcuTrans program and other post-processing modules; see the *AcuSolve Programs Reference Manual* for details.

For the *discete_ordinate* radiation model, exchange of radiative intensity occurs at the interface between participating media. For this case the RADIATION_SURFACE type is set to radiation_interface and the *radiation_interface_type* can be either internal or external. For the internal *radiation_interface_type*, the condition is applied between two participating media (two media both with a MATERIAL_RADIATION_MODEL). For the external *radiation_interface_type*, the RADIATION_SURFACE is applied to the boundary of a semi-transparent media (has a MATERIAL_RADIATION_MODEL) and models radiative exchange with the surrounding environment. The exchange of radiative intensity depends on the diffused_fraction of the interface and the refractive indices of the media. The different scenarios are described below.

SIMPLE_BOUNDARY_CONDITION

Specifies element and nodal boundary conditions on a set of element faces.

Type

AcuSolve Command

Syntax

```
SIMPLE_BOUNDARY_CONDITION("name") {parameters...}
```

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral
six_node_triangle or tri6	Six-node triangle.

element_set or *elem_set (string) [no default]*

User-given name of the parent element set.

surfaces (array) [no default]

List of element surfaces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this simple boundary condition. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to the SIMPLE_BOUNDARY_CONDITION command. This option is used in place of directly specifying these parameters. In the event that both of the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

type (enumerated) [=wall]

Type of the boundary surface.

wall	No-slip solid wall.
auto_wall	Automatic wall treatment.
inflow	Inflow boundary.

outflow	Outflow boundary.
slip	Slip surface.
symmetry	Plane of symmetry.
far_field	Either inflow or outflow depending on specified velocity.
free_surface	Free surface.

inflow_type (enumerated) [=velocity]

Type of the inflow boundary conditions. Used with inflow type.

velocity or vel	Nodal velocity and scalars.
pressure or pres	Weak pressure and nodal scalars.
stagnation_pressure or stag_pres	Weak stagnation pressure and nodal scalars.
mass_flux or mass	Integrated mass flux (equivalent to mass flow rate) and nodal scalars (except eddy viscosity).
flow_rate or flow	Integrated volume flux (equivalent to volumetric flow rate) and nodal scalars (except eddy viscosity).
average_velocity or ave_vel	Average velocity and nodal scalars (except eddy viscosity).
gravity_wave or wave	Nodal velocity, field and scalars for wave generator.
atmospheric	Atmospheric boundary layer.
velocity_pressure_temperature	All three variables (velocity, pressure, temperature) needed to specify supersonic inflow boundary condition. Used with *compressible_navier_stokes* equation.
mach_pressure_temperature	Another choice of the supersonic inflow boundary condition is Mach, pressure, and temperature. Used with *compressible_navier_stokes* equation.
phasic	Used to set phasic velocity for *multi_field*=eulerian_eulerian. If used, all velocity values for individual phases should be set from *field_boundary_conditions*.

humidity_type (enumerated) [=relative_humidity]

Type of humidity specified for the boundary condition. Used with *inflow_type* for the humid-air model.

relative_humidity	
dewpoint_temperature	

mass_fraction

relative_humidity (real) [=0]
Value of the relative humidity in percentage basis.

dewpoint_temperature (real) [=0]
Value of the *dewpoint_temperature*.

mass_fraction (list)[ ]
When *humidity_type* = mass_fration, *field_boundary_conditions* are used to specify mass fraction values of two fields.

outflow_type (enumerated) [=auto_pressure]
Type of the outflow boundary conditions. Used with outflow type.

auto_pressure	The user specified pressure is used in subsonic flow, while the user specified pressure is ignored if it is lower than the computed pressure on the outflow surface.
fix_pressure	Pressure is specified by you.
free_pressure	The user specified pressure is ignored.

back_flow_conditions (boolean) [=off]
Flag specifying whether to specify nodal boundary conditions for scalar variables on nodes on outflow surfaces where there is flow into the fluid domain. Used with outflow and far_field types.

temperature_back_flow_type or temp_back (enumerated) [=value]
How the constant temperature boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires temperature.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
bulk	Bulk average of solution over entire outflow surface.
exiting_bulk or exit_bulk	Bulk average of solution over subset of outflow surface that has flow exiting the domain.

species_1_back_flow_type or species_1_back (enumerated) [=value]
How the constant species_1 boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires species_1.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.

mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

species_2_back_flow_type or *species_2_back* *(enumerated) [=value]*

How the constant species_2 boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires species_s.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

species_3_back_flow_type or *species_3_back* *(enumerated) [=value]*

How the constant species_3 boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires species_3.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

species_4_back_flow_type or *species_4_back* *(enumerated) [=value]*

How the constant species_4 boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires species_4.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

species_5_back_flow_type or *species_5_back* *(enumerated) [=value]*

How the constant species_5 boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires species_5.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

species_6_back_flow_type or *species_6_back* *(enumerated) [=value]*

How the constant species_6 boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires species_6.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

species_7_back_flow_type or *species_7_back* *(enumerated) [=value]*

How the constant species_7 boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires species_7.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

species_8_back_flow_type or *species_8_back* *(enumerated) [=value]*

How the constant species_8 boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires species_8.

area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

species_9_back_flow_type or *species_9_back* *(enumerated) [=value]*
How the constant species_9 boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires species_9.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

field_back_flow_type or *field_back* *(enumerated) [=value]*
How the constant field boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires *field_boundary_conditions*.

field_boundary_conditions *(list) [no default]*
List of field boundary conditions that are applied at this boundary. Only used for type inflow and outflow. For inflow type, the values are applied to the fields directly. For outflow type, they are applied only if *back_flow_conditions*=on on the nodes where the flow is inbound.

viscoelastic_xx_back_flow_type or *xxvest_back* *(enumerated) [=value]*
How the constant viscoelastic_xx_stress boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires *viscoelastic_xx_stress*.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

viscoelastic_yy_back_flow_type or *yyvest_back* *(enumerated) [=value]*

How the constant viscoelastic_yy_stress boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires *viscoelastic_yy_stress*.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

viscoelastic_zz_back_flow_type or *zzvest_back* *(enumerated) [=value]*

How the constant *viscoelastic_zz_stress* boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires *viscoelastic_zz_stress*.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

viscoelastic_xy_back_flow_type or *xyvest_back* *(enumerated) [=value]*

How the constant viscoelastic_xy_stress boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires *viscoelastic_xy_stress*.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

viscoelastic_yz_back_flow_type or *yzvest_back* *(enumerated) [=value]*

How the constant viscoelastic_yz_stress boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires *viscoelastic_yz_stress*.

area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

viscoelastic_zx_back_flow_type or *zxvest_back* *(enumerated) [=value]*
How the constant viscoelastic_zx_stress boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires *viscoelastic_zx_stress*.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

eddy_viscosity_back_flow_type or *eddy_back* *(enumerated) [=value]*
How the constant eddy_viscosity boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires *eddy_viscosity*.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

kinetic_energy_back_flow_type or *tke_back* *(enumerated) [=value]*
How the constant turbulence_kinetic_energy boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires *kinetic_energy*.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.

mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

eddy_frequency_back_flow_type or *tomega_back* *(enumerated) [=value]*

How the constant turbulence eddy_frequency boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires *eddy_frequency*.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

dissipation_rate_back_flow_type or *teps_back* *(enumerated) [=value]*

How the constant turbulence dissipation_rate boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires *dissipation_rate*.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

intermittency_back_flow_type or *tintc_back* *(enumerated) [=value]*

How the constant turbulence transition intermittency boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires *intermittency*.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

transition_re_theta_back_flow_type or *treth_back* *(enumerated) [=value]*
How the constant turbulence dissipation_rate boundary condition is defined under back flow conditions. Used with outflow and far_field types with *back_flow_conditions*=on.

value	Specified value. Requires *transition_re_theta*.
area_average or area_ave	Average of solution over entire outflow surface.
exiting_area_average or exit_area_ave	Area average of solution over subset of outflow surface that has flow exiting the domain.
mass_flux_average	Mass flux average of solution over entire outflow surface.
exiting_mass_flux_average	Mass flux average of solution over subset of outflow surface that has flow exiting the domain.

precedence *(integer) (=1)*
Precedence of the nodal boundary conditions defined here with respect to others defined elsewhere. Highest value has precedence on a per-variable basis.

reference_frame *(string) [=none]*
User-given name of the reference frame for transforming velocity boundary conditions. If none, or if there are no velocity boundary conditions, no transformation takes place. Used with wall, inflow, slip, and symmetry types.

reference_frame_type *(enumerated) [=direct]*
Type of reference frame. Used with inflow type, wall type, auto_wall type and farfield type.

direct	Defined explicitly through the *reference_frame* identifier.
inherited	*reference_frame* is inherited from parent element set. Requires auto_wall type.

inflow_velocity_type *(enumerated) [=cartesian]*
Type of the inflow velocity. Used with inflow type and velocity inflow type.

cartesian	Velocity in cartesian coordinates. Requires *x_velocity*, *y_velocity*, and *z_velocity*.
cylindrical	Velocity in cylindrical coordinates. Requires *cylinder_axis*, *axial_velocity*, *radial_velocity*, and *tangential_velocity*.
spherical	Radial velocity component in spherical coordinates, other two components are zero. Requires *sphere_center* and *radial_velocity*.
normal	Velocity component normal to surface, other two components are zero. Requires *normal_velocity*.

waves list *(={})*
List of wave boundary conditions to generate waves at the inflow surface. Used with inflow type and gravity_wave inflow type.

heavy_fluid_velocity (array) (={0,0,0})
Velocity of the heavier fluid when used for a multiphase problem with two fluids and a stable interface. Used with inflow type and gravity_wave inflow_type.

light_fluid_velocity (array) (={0,0,0})
Velocity of the lighter fluid when used for a multiphase problem with two fluids and a stable interface. Used with inflow type and gravity_wave inflow type.

wave_damping (boolean) (=on)
Flag specifying whether to apply wave damping near the outflow surface. If off, no wave damping is applied near the outflow surface. Used with outflow type.

wall_velocity_type (enumerated) [=match_mesh_velocity]
Type of the wall velocity. Used with wall type.

zero	Velocity is zero.
match_mesh_velocity	Velocity matches velocity of the mesh.
cartesian	Velocity in cartesian coordinates. Requires *x_velocity*, *y_velocity*, and *z_velocity*.
cylindrical	Velocity in cylindrical coordinates. Requires *cylinder_axis*, *axial_velocity*, *radial_velocity*, and *tangential_velocity*.
spherical	Radial velocity component in spherical coordinates, other two components are zero. Requires *sphere_center* and *radial_velocity*.
normal	Velocity component normal to surface, other two components are zero. Requires *normal_velocity*.

cylinder_axis (array) [={0,0,0; 0,1,0}]
Coordinates of two points that define the axis of the cylinder. Used with inflow type and velocity inflow type and cylindrical inflow velocity type, or with wall type and cylindrical wall velocity type.

sphere_center (array) [={0,0,0}]
Coordinates of the center of the sphere. Used with inflow type and velocity inflow type and spherical inflow velocity type, or with wall type and spherical wall velocity type.

x_velocity or xvel (real) [=0]
Value of the x-velocity. Used with inflow type and velocity inflow type and cartesian inflow velocity type, or with wall type and cartesian wall velocity type.

x_velocity_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the x-velocity boundary condition value. If none, no scaling is performed. Used with *x_velocity*.

y_velocity or yvel (real) [=0]
Value of the y-velocity. Used with inflow type and velocity inflow type and cartesian inflow velocity type, or with wall type and cartesian wall velocity type.

y_velocity_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the y-velocity boundary condition value. If none, no scaling is performed. Used with *y_velocity*.

z_velocity or zvel (real) [=0]
Value of the z-velocity. Used with inflow type and velocity inflow type and cartesian inflow velocity type, or with wall type and cartesian wall velocity type.

z_velocity_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the z-velocity boundary condition value. If none, no scaling is performed. Used with *z_velocity*.

axial_velocity (real) [=0]
Value of the axial velocity. Used with inflow type and velocity inflow type and cylindrical inflow velocity type, or with wall type and cylindrical wall velocity type.

axial_velocity_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the axial velocity boundary condition value. If none, no scaling is performed. Used with *axial_velocity*.

radial_velocity (real) [=0]
Value of the radial velocity. Used with inflow type and velocity inflow type and cylindrical and spherical inflow velocity types, or with wall type and cylindrical and spherical wall velocity types.

radial_velocity_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the radial velocity boundary condition value. If none, no scaling is performed. Used with *radial_velocity*.

tangential_velocity (real) [=0]
Value of the tangential velocity. Used with inflow type and velocity inflow type and cylindrical inflow velocity type, or with wall type and cylindrical wall velocity type.

tangential_velocity_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the tangential velocity boundary condition value. If none, no scaling is performed. Used with *tangential_velocity*.

normal_velocity (real) [=0]
Value of the normal velocity. Used with inflow type and velocity inflow type and normal inflow velocity type, or with wall type and normal wall velocity type.

normal_velocity_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the normal velocity boundary condition value. If none, no scaling is performed. Used with *normal_velocity*.

mass_flux or mass (real) >0 [=1]
Value of the integrated mass flux. Used with inflow type and mass_flux inflow type.

mass_flux_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the integrated mass flux value. If none, no scaling is performed. Used with inflow type and mass_flux inflow type.

flow_rate or flow (real) >0 [=1]
Value of the integrated flow rate. Used with inflow type and flow_rate inflow type.

flow_rate_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the integrated flow rate value. If none, no scaling is performed. Used with inflow type and flow_rate inflow type.

average_velocity or *ave_vel (real) >0 [=1]*
Value of the average velocity. Used with inflow type and average_velocity inflow type.

average_velocity_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the average velocity value. If none, no scaling is performed. Used with inflow type and average_velocity inflow type.

inflow_mach_number_type (enumerated) [=cartesian]
Type of the inflow mach number. Used with inflow type and *mach_pressure_temperature* type.

cartesian	Mach number in cartesian coordinates.
normal	Mach number component normal to surface, the other two components are zero.

mach_number (real) [=0]
Value of the Mach number. Used with inflow type and *compressible_navier_stokes*.

mach_number_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the mach number boundary condition value. If none, no scaling is performed. Used with *mach_number*.

mach_number_dir (array) [={1,0,0}]
Mach number direction. Used with inflow type, *mach_pressure_temperature inflow_type* and *inflow_mach_number_type* as well as *far_field* type.

pressure or *pres (real) [=0]*
Value of the pressure. Used with outflow, far_field, and free_surface types, and inflow type with pressure inflow type.

pressure_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the pressure boundary condition value. If none, no scaling is performed. Used with pressure.

pressure_loss_factor (real) >=0 [=0]
Coefficient of a pressure loss term added/subtracted to outflow/inflow pressure boundary conditions. Used with outflow, far_field, and free_surface types, and inflow type with pressure inflow type.

pressure_loss_factor_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the pressure loss factor. Used with outflow, far_field, and free_surface types, and inflow type with pressure inflow type. If none, no scaling is performed.

hydrostatic_pressure (boolean) [=off]
Flag specifying whether to add hydrostatic pressure to pressure and stagnation pressure boundary conditions. Used with outflow, far_field, and free_surface types, and inflow type with pressure and stagnation_pressure inflow types.

hydrostatic_pressure_origin (array) [={0,0,0}]
Coordinates of any location where the hydrostatic pressure is zero. Used with *hydrostatic_pressure*=on.

stagnation_pressure or stag_pres (real) [=0]
Value of the stagnation pressure. Used with inflow type and stagnation_pressure inflow type.

stagnation_pressure_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the stagnation pressure boundary condition value. If none, no scaling is performed. Used with *stagnation_pressure*.

temperature_type or temp_type (enumerated) [=flux]
Type of the thermal boundary condition. Used with wall type.

none	No thermal boundary condition.
value	Nodal boundary condition.
flux	Heat flux boundary condition.

temperature or temp (real) [=0]
Value of the temperature. Used with inflow type, and wall type with value temperature type. Also used with outflow and far_field types when *back_flow_conditions* = on.

temperature_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the temperature boundary condition value. If none, no scaling is performed. Used with *temperature*.

temperature_coupling_type (enumerated) [= tied]
Type of the temperature boundary condition used for thermal simulations for the nodes/elements of this SIMPLE_BOUNDARY_CONDITION command that match up with elements/nodes of a MESH_BOUNDARY_CONDITION of *type*=external_code. In the event that no matching MESH_BOUNDARY_CONDITION of *type*=external_code is found for the elements within this SIMPLE_BOUNDARY_CONDITION command, this parameter is ignored.

none	No temperature coupling with external code.
tied	Temperature coupling with external code.

heat_flux or heat (real) [=0]
Value of the heat flux. Used with wall type and flux temperature type.

heat_flux_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the heat flux boundary condition value. If none, no scaling is performed. Used with *heat_flux*.

convective_heat_coefficient (real) [=0]
Value of the convective heat coefficient. Used with wall type and flux temperature type.

convective_heat_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the convective heat coefficient value. If none, no scaling is performed. Used with *convective_heat_coefficient*.

convective_heat_reference_temperature (real) [=273.15]
Value of the reference temperature associated with the convective heat coefficient. Used with wall type and flux temperature type.

reference_temperature_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the convective_heat_reference temperature. If none, no scaling is performed.

nucleate_boiling (boolean) [=off]
Flag specifying whether to activate single-phase nucleate boiling on walls. If off, single-phase nucleate-boiling is not applied. Used with wall type.

species_1_type or spec1_type (enumerated) [=flux]
Type of the species_1 boundary condition. Used with wall type.

value	Nodal boundary condition.
flux	Flux boundary condition.

species_1 or spec1 (real) [=0]
Value of the species_1. Used with inflow type, and wall type with value species_1 type. Also used with outflow and far_field types when *back_flow_conditions* = on.

species_1_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the species_1 boundary condition value. If none, no scaling is performed. Used with *species_1*.

species_1_flux (real) [=0]
Value of the species_1 flux. Used with wall type and species_1 type.

species_1_flux_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the species_1 flux boundary condition value. If none, no scaling is performed. Used with *species_1_flux*.

species_2_type or spec2_type (enumerated) [=flux]
Type of the species_2 boundary condition. Used with wall type.

value	Nodal boundary condition.
flux	Flux boundary condition.

species_2 or spec2 (real) [=0]
Value of the species_2. Used with inflow type, and wall type with value species_2 type. Also used with outflow and far_field types when *back_flow_conditions* = on.

species_2_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the species_2 boundary condition value. If none, no scaling is performed. Used with *species_2*.

species_2_flux (real) [=0]
Value of the species_2 flux. Used with wall type and flux species_2 type.

species_2_flux_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the species_2_flux boundary condition value. If none, no scaling is performed. Used with *species_2_flux*.

species_3_type or *spec3_type* *(enumerated) [=flux]*
Type of the species_3 boundary condition. Used with wall type.

value	Nodal boundary condition.
flux	Flux boundary condition.

species_3 or *spec3* *(real) [=0]*
Value of the species_3. Used with inflow type, and wall type with value species_3 type. Also used with outflow and far_field types when *back_flow_conditions* = on.

species_3_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the species_3 boundary condition value. If none, no scaling is performed. Used with *species_3*.

species_3_flux *(real) [=0]*
Value of the species_3 flux. Used with wall type and flux species_3 type.

species_3_flux_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the species_3_flux boundary condition value. If none, no scaling is performed. Used with *species_3_flux*.

species_4_type or *spec4_type* *(enumerated) [=flux]*
Type of the species_4 boundary condition. Used with wall type.

value	Nodal boundary condition.
flux	Flux boundary condition.

species_4 or *spec4* *(real) [=0]*
Value of the species_4. Used with inflow type, and wall type with value species_4 type. Also used with outflow and far_field types when *back_flow_conditions* = on.

species_4_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the species_4 boundary condition value. If none no scaling is performed. Used with *species_4*.

species_4_flux *(real) [=0]*
Value of the species_4 flux. Used with wall type and flux species_4 type.

species_4_flux_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the species_4_flux boundary condition value. If none, no scaling is performed. Used with *species_4_flux*.

species_5_type or *spec5_type* *(enumerated) [=flux]*
Type of the species_5 boundary condition. Used with wall type.

value	Nodal boundary condition.
flux	Flux boundary condition.

species_5 or *spec5* *(real) [=0]*
Value of the species_5. Used with inflow type, and wall type with value species_5 type. Also used with outflow and far_field types when *back_flow_conditions* = on.

species_5_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the species_5 boundary condition value. If none, no scaling is performed. Used with *species_5*.

species_5_flux (real) [=0]
Value of the species_5 flux. Used with wall type and flux species_5 type.

species_5_flux_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the species_5 flux boundary condition value. If none, no scaling is performed. Used with *species_5_flux*.

species_6_type or spec6_type (enumerated) [=flux]
Type of the species_6 boundary condition. Used with wall type.

value	Nodal boundary condition.
flux	Flux boundary condition.

species_6 or spec6 (real) [=0]
Value of the species_6. Used with inflow type, and wall type with value species_6 type. Also used with outflow and far_field types when *back_flow_conditions* = on.

species_6_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the species_6 boundary condition value. If none, no scaling is performed. Used with *species_6*.

species_6_flux (real) [=0]
Value of the species_6 flux. Used with wall type and flux species_6 type.

species_6_flux_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the species_6 flux boundary condition value. If none, no scaling is performed. Used with *species_6_flux*.

species_7_type or spec7_type (enumerated) [=flux]
Type of the species_7 boundary condition. Used with wall type.

value	Nodal boundary condition.
flux	Flux boundary condition.

species_7 or spec7 (real) [=0]
Value of the species_7. Used with inflow type, and wall type with value species_7 type. Also used with outflow and far_field types when *back_flow_conditions*=on.

species_7_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the species_7 boundary condition value. If none, no scaling is performed. Used with *species_7*.

species_7_flux (real) [=0]
Value of the species_7 flux. Used with wall type and flux species_7 type.

species_7_flux_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the species_7 flux boundary condition value. If none, no scaling is performed. Used with *species_7_flux*.

species_8_type or *spec8_type* *(enumerated) [=flux]*
Type of the species_8 boundary condition. Used with wall type.

value	Nodal boundary condition.
flux	Flux boundary condition.

species_8 or *spec8* *(real) [=0]*
Value of the species_8. Used with inflow type, and wall type with value species_8 type. Also used with outflow and far_field types when *back_flow_conditions* = on.

species_8_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the species_8 boundary condition value. If none no scaling is performed. Used with *species_8*.

species_8_flux *(real) [=0]*
Value of the species_8 flux. Used with wall type and flux species_8 type.

species_8_flux_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the species_8 flux boundary condition value. If none, no scaling is performed. Used with *species_8_flux*.

species_9_type or *spec9_type* *(enumerated) [=flux]*
Type of the species_9 boundary condition. Used with wall type.

value	Nodal boundary condition.
flux	Flux boundary condition.

species_9 or *spec9* *(real) [=0]*
Value of the species_9. Used with inflow type, and wall type with value species_9 type. Also used with outflow and far_field types when *back_flow_conditions* = on.

species_9_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the species_9 boundary condition value. If none, no scaling is performed. Used with *species_9*.

species_9_flux *(real) [=0]*
Value of the species_9 flux. Used with wall type and flux species_9 type.

species_9_flux_multiplier_function *(string) [=none]*
User-given name of the multiplier function for scaling the species_9 flux boundary condition value. If none, no scaling is performed. Used with *species_9_flux*.

maintain_ambient_turbulence or *amb_turb* *(boolean) [=off]*
Flag specifying whether to maintain inflow turbulence intensity throughout the domain. Valid for all two-equation turbulence models. Used with inflow type, velocity inflow_type and *farfield* type.

synthetic_turbulence *(boolean) [=off]*
Flag specifying whether to activate the synthetic turbulence feature at inflow.
Used with velocity, atmospheric inflow type. This command is only activated when *synthetic_turbulence_input_type* in *turbulence_model_parameters* is set to k_omega (or k_epsilon).

turbulence_input_type (enumerated) [=direct]

Type of turbulence input at inflow. Used with inflow type, with velocity, pressure, stagnation_pressure, *inflow_type* and farfield type.

direct	Direct specification of turbulence quantities. Requires associated turbulent value associated with selected turbulence model.
auto	Automatic definition for turbulence input. Valid for all turbulence models.
intensity_length_scale or tilen	Specifies turbulence quantities based on turbulence_intensity and length_scale. Valid for all turbulence models.
intensity_viscosity_ratio or *tivr*	Specifies turbulence quantities based on turbulence_intensity and viscosity_ratio. Valid for two-equation turbulence models.
viscosity_ratio or *vr*	Specifies turbulence quantities based on viscosity_ratio. Valid for the spalart_allmaras turbulence model.

turbulence_flow_type (enumerated) [=internal]

Type of turbulence flow type at inflow. Used with inflow type, with velocity, pressure, *stagnation_pressure*, *inflow_type* and farfield type.

internal	Specifies turbulent state at inflow for internal flows. Used with *turbulence_input_type* auto, intensity_length_scale and intensity_viscosity_ratio.
external	Specifies turbulent state at inflow for external flows. Used with *turbulence_input_type* auto, intensity_length_scale and intensity_viscosity_ratio.

turbulence_intensity_type (enumerated) [=auto]

Type of turbulence intensity at inflow. Used with inflow type and velocity, pressure, and *stagnation_pressure* inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

auto	Automatic definition for turbulence intensity. Valid for all turbulence models.
value	User specified turbulence intensity at inflow.
low	Specifies turbulence intensity for low turbulence inflow simulations.
medium	Specifies turbulence intensity for medium turbulence inflow simulations.
high	Specifies turbulence intensity for high turbulence inflow simulations.

percent_turbulence_intensity (real) [=0.1]
Value of the turbulence intensity at the inflow. Used with inflow type and velocity, pressure, and stagnation_pressure inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

turbulence_velocity_scale (real) [=0.0]
Value of the inflow velocity scale used to compute turbulent properties. Used with inflow type and velocity, pressure, and stagnation_pressure inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

turbulence_length_scale (real) [=0.0]
Value of the inflow length scale used to compute turbulent properties. Used with inflow type and velocity, pressure, and stagnation_pressure inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

turbulence_viscosity_ratio (real) [=0.1]
Value of the inflow turbulence viscosity ratio used to compute turbulent properties. Used with inflow type and velocity, pressure, and stagnation_pressure inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

viscoelastic_xx_stress or xxvest [=0]
Value of the viscoelastic xx stress component. Used with inflow type and velocity, pressure, and stagnation_pressure inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

viscoelastic_xx_stress_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the viscoelastic xx stress boundary condition value. If none, no scaling is performed. Used with *viscoelastic_xx_stress*.

viscoelastic_yy_stress or yyvest [=0]
Value of the viscoelastic yy stress component. Used with inflow type and velocity, pressure, and stagnation_pressure inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

viscoelastic_yy_stress_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the viscoelastic yy stress boundary condition value. If none, no scaling is performed. Used with *viscoelastic_yy_stress*.

viscoelastic_zz_stress or zzvest [=0]
Value of the viscoelastic zz stress component. Used with inflow type and velocity, pressure, and stagnation_pressure inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

viscoelastic_zz_stress_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the viscoelastic zz stress boundary condition value. If none, no scaling is performed. Used with *viscoelastic_zz_stress*.

viscoelastic_xy_stress or xyvest [=0]
Value of the viscoelastic xy stress component. Used with inflow type and velocity, pressure, and stagnation_pressure inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

viscoelastic_xy_stress_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the viscoelastic xy stress boundary condition value. If none, no scaling is performed. Used with *viscoelastic_xy_stress*.

viscoelastic_yz_stress or yzvest [=0]
Value of the viscoelastic yz stress component. Used with inflow type and velocity, pressure, and stagnation_pressure inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

viscoelastic_yz_stress_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the viscoelastic yz stress boundary condition value. If none, no scaling is performed. Used with *viscoelastic_yz_stress*.

viscoelastic_zx_stress or zxvest [=0]
Value of the viscoelastic zx stress component. Used with inflow type and velocity, pressure, and stagnation_pressure inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

viscoelastic_zx_stress_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the viscoelastic zx stress boundary condition value. If none, no scaling is performed. Used with *viscoelastic_zx_stress*.

eddy_viscosity or eddy (real) [=0]
Value of the kinematic eddy viscosity. Used with inflow type and velocity, pressure, and stagnation_pressure inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

eddy_viscosity_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the eddy viscosity boundary condition value. If none, no scaling is performed. Used with *eddy_viscosity*.

kinetic_energy or tke (real) [=0]
Value of the turbulent kinetic energy. Used with inflow type and velocity, pressure, and stagnation_pressure inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

kinetic_energy_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the turbulent kinetic energy boundary condition value. If none, no scaling is performed. Used with *kinetic_energy*.

eddy_frequency or tomega (real) [=0]
Value of the turbulent eddy frequency. Used with inflow type and velocity, pressure, and stagnation_pressure inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

eddy_frequency_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the turbulent eddy frequency boundary condition value. If none, no scaling is performed. Used with *eddy_frequency*.

dissipation_rate or teps (real) [=0]
Value of the turbulent dissipation rate. Used with inflow type and velocity, pressure, and stagnation_pressure inflow types. Also used with outflow and far_field types when *back_flow_conditions* = on.

dissipation_rate_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the turbulent dissipation rate boundary condition value. If none, no scaling is performed. Used with *dissipation_rate*.

turbulence_wall_type (enumerated) [=wall_function]
Type of the turbulence wall modeling. Used with wall type.

none	No wall modeling. When this option is set, the surface does not participate in the wall distance calculation for turbulence and no wall modeling is performed.
low_reynolds_number or low_re	Low Reynolds number damping functions.
wall_function or func	Turbulence wall function.
running_average_wall_function	Turbulence wall function based on running average variables.

roughness_height (real) >=0 [=0]
Average wall roughness height. Used with low_reynolds_number, wall_function, and running_average_wall_function turbulence wall types.

granular_wall_specularity (real) >=0, <=1 [=1]
Specularity for granular flows. Used to control the interaction between solid particles and wall. free-slip (=0.0), no-slip (=1.0), partial slip (>0, <1).

wall_function_heat_flux_factor >=1 [=1]
Constant factor used to scale the turbulent thermal conductivity within the first element off the wall. Used with *turbulence_wall_type* = wall_function and running_average_wall_function .

non_reflecting_factor (real) >=0 [=0]
Amount of non-reflecting modification. If zero, there is no effect. If one, waves can pass through the boundary without reflection. Used with pressure variable with outflow type and *mass_flux* variable with inflow type. Turning on both non_reflecting_bc_running_average_field and running_average specifies the use of running average fields to enhance the performance of non-reflective boundary conditions when *flow* = navier_stokes is used. When *flow* = compressible_navier_stokes is set, the running_average option does not need to be turned on.

surface_tension_model (string) [=none]
User-given name of the surface tension model. If none, surface tension is not modeled. Used with free_surface type. This parameter has been migrated to the MESH_BOUNDARY_CONDITION command.

contact_angle_model (string) [=none]
User-given name of the contact angle model. If none, the default contact angle model is used. Used with *surface_tension_model* and *multi_field*=levelset, levelset_bfecc, and eulerian_eulerian as well as wall_type.

mesh_displacement_type (enumerated) [=fixed]
Type of the wall for mesh displacement boundary conditions. Not used with free_surface type. This parameter has been migrated to the MESH_BOUNDARY_CONDITION command.

none	No mesh displacement boundary conditions.

fixed	Mesh fixed to the wall.
slip	Mesh slips tangentially along the wall.
flexible_body	Mesh follows a flexible body. Requires flexible_body.
guide_surface	Mesh follows a guide surface. Requires *guide_surface*.

flexible_body (string) [=none]
User-given name of the FLEXIBLE_BODY command. Used with flexible_body mesh displacement type. This parameter has been migrated to the MESH_BOUNDARY_CONDITION command.

nodal_x_modes (array) [no default]
Array of node number (first column) and x-component eigenvector values (last num_modes columns, where num_modes is given by the FLEXIBLE_BODY command referenced by *flexible_body*). Nodes must match those referenced in the surface. Used with flexible_body mesh displacement type. This parameter has been migrated to the MESH_BOUNDARY_CONDITION command.

nodal_y_modes (array) [no default]
Array of node number (first column) and y-component eigenvector values (last num_modes columns, where num_modes is given by the FLEXIBLE_BODY command referenced by *flexible_body*). Nodes must match those referenced in the surface. Used with flexible_body mesh displacement type. This parameter has been migrated to the MESH_BOUNDARY_CONDITION command.

nodal_z_modes (array) [no default]
Array of node number (first column) and z-component eigenvector values (last num_modes columns, where num_modes is given by the FLEXIBLE_BODY command referenced by *flexible_body*). Nodes must match those referenced in the surface. Used with flexible_body mesh displacement type. This parameter has been migrated to the MESH_BOUNDARY_CONDITION command.

guide_surface (string) [=none]
User-given name of the GUIDE_SURFACE command. Used with guide_surface mesh displacement type. This parameter has been migrated to the MESH_BOUNDARY_CONDITION command.

mesh_motion (string) [=none]
User-given name of the MESH_MOTION command for determining mesh displacement boundary conditions. If none, this parameter has no effect. Used with fixed and flexible_body mesh displacement types. This parameter has been migrated to the MESH_BOUNDARY_CONDITION command.

split_internal_surfaces (boolean) [=off]
Option to automatically split the nodes on internal surfaces that require two boundary conditions, that is, cases that include baffles, sliding mesh or thermal interfaces.

gap_factor (real) >=0 [=1]
Non-dimensional (with respect to the length of an element face) maximum gap allowed for two element faces to be in contact. No maximum if zero.

gap (real) >= 0 [=0]
Dimensional maximum gap allowed for two element faces to be in contact. No maximum if zero.

crease_angle or angle (real) >=0 <=180 [=90]
Maximum angle between face normals allowed for two element faces to be in contact. No maximum if zero.

active_type (enumerated) [=all]
Type of the active flag. Determines which surfaces in this set will have boundary conditions imposed by this command.

all	All surfaces in this set are active.
none	No surface in this set is active.
no_interface	Only surfaces that are not in an interface surface set or do not find a contact surface of an appropriate medium are active.

atmospheric_roughness_type (enumerated) (=suburb)
Type of the surface roughness for atmospheric boundary layer. Provides typical terrain types and user specified input.

user_value	User specification of roughness height.
ocean	Use roughness height for ocean (0.001m).
low_grass	Use roughness height for low grass (0.02m).
high_grass	Use roughness height for high grass (0.06m).
forest	Use roughness height for forest (1.0m).
suburb	Use roughness height for suburb (0.3m).
town	Use roughness height for town (1.0m).
city	Use roughness height for city (2.5m).

atmospheric_reference_velocity_type (enumerated) (=friction_velocity)
Type of the reference velocity for atmospheric boundary layer.

friction_velocity	Use ground friction velocity.
sample_height_velocity	Use height and corresponding velocity.

atmospheric_ground_roughness (real) (=0.001)
Value of roughness height if *atmospheric_roughness_type* = user_value.

atmospheric_friction_velocity (real) (=1.0)
Value of friction velocity if *atmospheric_reference_velocity_type* = friction_velocity.

atmospheric_sample_velocity (real) (=5.0)
Value of reference velocity if *atmospheric_reference_velocity_type* = sample_height_velocity.

atmospheric_sample_height (real) (=10.0)
Value of reference height if *atmospheric_reference_velocity_type* = sample_height_velocity.

atmospheric_ground_origin (array) (={0,0,0})
Array of origin location for atmospheric boundary layer, specified in the global xyz coordinate system.

atmospheric_ground_normal_dir (array) (={0,0,1})
Array of ground normal direction for atmospheric boundary layer, specified in the global xyz coordinate system.

atmospheric_flow_dir (array) (={1,0,0})
Array of flow direction for atmospheric boundary layer, specified in the global xyz coordinate system.

electric_potential_type or elecp_type (enumerated) [=flux]
Type of the electric potential/current boundary condition. Used with wall type.

none	No electric potential boundary condition.
value	Nodal electric potential boundary condition.
flux	Current density or current boundary condition. Current density is the default condition.

electric_potential or elecP (real) [=0]
Value of the electric potential. The default value is 0V or the ground potential.

electric_potential_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the electric potential boundary condition value. If none, no scaling is performed. Used with *electric_potential*.

electric_current_density (real) [=0]
Value of the electric current density normal to a surface. The default value is 0 A/m^2.

electric_current_density_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the electric current density boundary condition value. If none, no scaling is performed. Used with *electric_current_density*.

electric_current (real) [=0]
Value of the current at a surface. The default value is 0 A. If current is defined the value of current density is ignored.

electric_current_multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the electric current boundary condition value. If none, no scaling is performed. Used with *electric_current*.

electric_current_from_module (boolean) [=off]
Calculates the boundary current automatically based on the current supplied to a battery module (multiple battery cells).

battery_tab_connection_surface (string) [=no default]
User-given name of the BATTERY_COMPONENT_MODEL to which this surface belongs. The *component_type* must be either a tab_positive or tab_negative.

feedback_condition (string) [=no default]

User-given name of the FEEDBACK_CONDITION. This condition enables control of a surface heat flux on a SIMPLE_BOUNDARY_CONDITION based on a control variable at a point defined in the FEEDBACK_CONDITION.

topology_design_wall (enumerated) [=off]

Type of the topology wall boundary condition. Used with wall type.

off	The design topology value is free at the wall surface.
on	The design topology value is set to 1, representing a solid, when *filter_wall_type* = constant in the DESIGN_VARIABLES_FIELD command. If *filter_wall_type* = free, the *toplogy_design_wall* parameter is inactive.

Description

This command specifies the appropriate nodal and element boundary conditions on a set of surfaces (element faces) for the given type of boundary. "Nodal boundary conditions on a surface" is shorthand for "nodal boundary conditions on all the nodes that comprise the surface." The surfaces of an element boundary condition are defined with respect to the elements of an element set. For example,

```
ELEMENT_SET( "flow elements" ) {
    shape = four_node_tet
    elements                                = { ...
                                                4, 2, 5, 6, 8 ;
                                                5, 2, 6, 3, 5 ;
                                                ... }
 ...
}
SIMPLE_BOUNDARY_CONDITION( "heated wall" ) {
    type                                    = wall
    shape                                   = three_node_triangle
    element_set                             = "flow elements"
    surfaces                                = { 4, 41, 2, 5, 6 ;
                                                5, 51, 5, 6, 3 ; }
    wall_velocity_type                      = match_mesh_velocity
    temperature_type                        = flux
    heat_flux                               = 50
    convective_heat_coefficient             = 0
    convective_heat_reference_temperature = 300
    turbulence_wall_type                    = wall_function
    roughness_height                        = 0.001
}
```

defines all the necessary boundary conditions for a no-slip wall with an imposed heat flux of 50 on two surfaces of the element set "flow elements".

There are two main forms of this command. The legacy version, or single topology version, of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number, of the parent element set, a unique, within this set, surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as `heated_wall.ebc`:

```
4 41 2 5 6
5 51 5 6 3
```

and read by:

```
SIMPLE_BOUNDARY_CONDITION( "heated wall" ) {
 type        = wall
 shape       = three_node_triangle
 element_set = "flow elements"
 surfaces    = Read( "heated_wall.ebc" )
 ...
}
```

The mixed topology form of the `SIMPLE_BOUNDARY_CONDITION` command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the `SURFACE_SET` command, and are then referred to by the `SIMPLE_BOUNDARY_CONDITION` command. For example, a collection of triangular and quadrilateral element faces can be defined using the following `SURFACE_SET` commands.

```
SURFACE_SET( "tri faces" ) {
    surfaces    = { 1, 1, 1, 2, 4 ;
                    2, 2, 3, 4, 6 ;
```

```
                    3, 3, 5, 6, 8 ; }
    shape       = three_node_triangle
    volume_set  = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
    surfaces    = { 1, 1, 1, 2, 4, 9 ;
                    2, 2, 3, 4, 6, 12 ;
                    3, 3, 5, 6, 8, 15 ; }
    shape       = four_node_quad
    volume_set  = "prisms"
```

Then, a single `SIMPLE_BOUNDARY_CONDITION` command is defined that contains the tri and quad faces as follows:

```
SIMPLE_BOUNDARY_CONDITION( "heated wall" ) {
   surface_sets       = {"tri_faces", "quad_faces"}
   ...
}
```

The list of surface sets can also be placed in a file, such as `surface_sets.srfst`:

```
tri faces
quad faces
```

and read using:

```
SIMPLE_BOUNDARY_CONDITION( "heated wall" ) {
   surface_sets       = Read("surface_sets.srfst")
   ...
}
```

The mixed topology version of the `SIMPLE_BOUNDARY_CONDITION` command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *`surface_sets`* and *`surfaces`* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

The kind of boundary to which boundary conditions are applied is given by *`type`*. A separate `SIMPLE_BOUNDARY_CONDITION` command must be issued for each type of boundary, but not for each variable. It is expected that this command will be used instead of `NODAL_BOUNDARY_CONDITION` and `ELEMENT_BOUNDARY_CONDITION` commands for the vast majority of boundary conditions. The latter commands should be needed only for complex boundary conditions on individual variables. Internally to AcuSolve, each `SIMPLE_BOUNDARY_CONDITION` command is replaced by the appropriate `NODAL_BOUNDARY_CONDITION`, `ELEMENT_BOUNDARY_CONDITION` and `TURBULENCE_WALL` commands.

The element boundary (flux) conditions are applied at the quadrature points of the surfaces. The quadrature rule is inherited from the parent element set. Nodal boundary conditions are applied at the three, four, or six nodes of each surface.

Several types of boundary surfaces are available. For each type, nodal or flux boundary conditions are defined as appropriate for every variable. If the problem does not solve for one of these variables, then its boundary condition is ignored. The set of options is limited in order to keep this command simple.

The *precedence* parameter applies to all variables with nodal boundary conditions created by this command. If more than one nodal boundary condition is specified for a given variable on a given node, then the boundary condition with the highest value of *precedence* takes precedence. This may be useful when one variable has a nodal boundary condition too complex for this command. Instead of issuing boundary condition commands for every variable, it is recommended that this command be used followed by a NODAL_BOUNDARY_CONDITION command with a higher *precedence* value for that one variable. See the NODAL_BOUNDARY_CONDITION command for details about precedence tie-breaking. The concept of precedence is not supported for element boundary conditions so an ELEMENT_BOUNDARY_CONDITION command cannot be used to overwrite any flux conditions defined here. Trying to do so will result in a warning message and unpredictable behavior.

By default, all velocity boundary conditions are specified with respect to the fixed (or "laboratory") reference frame. The velocity boundary conditions on a rotating wall may be non-trivial. However, the *reference_frame* parameter provides a convenient way of specifying these conditions. Instead of the fixed frame, velocity boundary conditions are applied in the given frame. In many cases, values of zero could then be used; for example, boundary conditions on a rotating solid body could be defined using the wall type. See the REFERENCE_FRAME command for more details. The *reference_frame* parameter has no effect on variables other than velocity components.

For all types of surfaces except free_surface, nodal boundary conditions on the mesh displacement are defined. For a fixed mesh displacement type, all three components of the mesh displacement vector are set to zero. For slip and symmetry, just the component normal to the surface is zero.

A simple boundary condition of type wall imposes nodal boundary conditions in the given reference frame on the x-velocity, y-velocity, z-velocity, and eddy viscosity variables. Nodal mesh displacement boundary conditions are imposed as mentioned above. Either nodal or flux boundary conditions, determined by *temperature_type*, *species_1_type*, and so on, are imposed on temperature and all species variables. The nodal boundary conditions are specified by the *temperature*, *species_1*, and so on, parameters. The element boundary conditions are specified by the *heat_flux*, *convective_heat_coefficient*, *convective_heat_reference_temperature*, *species_1_flux*, and so on, parameters. Both regular and convective heat fluxes are included. All boundary conditions on the temperature and species may be scaled by their corresponding multiplier function parameters. Defaults are used for unspecified variables.

The convective and radiation heat fluxes are given by:

$$q_{conv} = h(T - T_{ref})$$

$$q_{rad} = h[(T + T_{off})^4 - (T_{ref} + T_{off})^4] \quad (178)$$

where h is the value of the element boundary condition; T is the temperature; T_{ref} is the reference temperature, given by *reference_temperature*; and T_{off} is the offset to convert to an absolute temperature, given by the *absolute_temperature_offset* parameter of the EQUATION command.

The *turbulence_wall_type* parameter determines the type of turbulence wall modeling: low Reynolds number, wall function, or wall function based on average field variables. The wall function may be modified by specifying an average wall roughness height, given by *roughness_height*. See the *type* and *roughness_height* parameters of the TURBULENCE_WALL command for details. An example for a wall with an imposed heat flux is given above.

The velocity boundary condition for a wall type may be specified in six different ways, as determined by *wall_velocity_type*. A value of *match_mesh_velocity* is the default and requires no further data. If the problem has a deforming mesh, that is, *mesh* is set to arbitrary_lagrangian_eulerian or specified in the EQUATION command, and *mesh_displacement_type* is set to guide_surface or flexible_body, see below, or *mesh_motion* is active, then the velocity is set equal to the velocity of the mesh. Otherwise, the wall velocity type reverts to zero, which sets the velocity to zero in the given reference frame. The other four wall velocity types allow the wall velocity to be given in different local coordinate systems. A wall velocity type of cartesian uses *x_velocity*, *y_velocity*, and *z_velocity* to impose boundary conditions in the global coordinate system, for example,

```
SIMPLE_BOUNDARY_CONDITION( "heated wall" ) {
   ...
   type               = wall
   wall_velocity_type = cartesian
   x_velocity         = 1
   y_velocity         = 0
   z_velocity         = 0
   ...
}
```

A wall velocity type of cylindrical sets boundary conditions on the axial, radial, and tangential components of the velocity in a cylindrical coordinate system:

```
SIMPLE_BOUNDARY_CONDITION( "heated wall" ) {
   type                = wall
   ...
   wall_velocity_type  = cylindrical
   cylinder_axis       = {0, 0, 0;
                          0, 1, 0}
   axial_velocity      = 1
   radial_velocity     = 0
   tangential_velocity = 0
   ...
}
```

where the axial direction is from the first to the second point in *cylinder_axis*, the radial direction is the outward normal from this line segment to the given node, and the tangential direction is determined by the right hand rule from the axial direction to the radial direction.

A wall velocity type of spherical imposes a given boundary condition on the radial component of the velocity in a spherical coordinate system and zero boundary conditions on the other two components:

```
SIMPLE_BOUNDARY_CONDITION( "heated wall" ) {
   type               = wall
   ...
   wall_velocity_type = spherical
   sphere_center      = {0, 0, 0}
   radial_velocity    = 1
   ...
}
```

where the radial direction is outward from *sphere_center* to the given node.

A wall velocity type of normal imposes a given boundary condition on the component of the velocity normal to the surface and zero boundary conditions on the other two components:

```
SIMPLE_BOUNDARY_CONDITION( "heated wall" ) {
   type               = wall
   ...
   wall_velocity_type = normal
   normal_velocity    = 1
   ...
}
```

where the normal direction is taken as positive inward towards the fluid domain.

The boundary conditions on the last four wall velocity types may be scaled by their corresponding multiplier functions, given by *x_velocity_multiplier_function* through *normal_velocity_multiplier_function*.

A simple boundary condition of type auto_wall simplifies the set-up of CFD models containing complex internal boundaries. All internal surfaces except for inflow, outflow, symmetry, slip and far-field, are processed automatically by AcuPrep when *type* = auto_wall. AcuPrep processes the auto_wall command by first identifying whether the surface is internal or external and then constructing a table of information regarding the parent volume set of each surface set. Once this information is obtained, the surface sets are prepared for the supported scenarios. When auto_wall is used, *reference_frame_type* and *mesh_displacement_type* will be set to be inherited. When interfaces are located within the CFD domain for mesh motion applications, auto_wall will automatically split interface surfaces, create the interface surface mechanism with *gap* and *gap_factor* appropriately and assign the boundary conditions. The following is an example of auto_wall input:

```
SIMPLE_BOUNDARY_CONDITION("auto wall"){
    ...
    type = auto_wall
    temperature_type = value
    temperature = 300
    turbulence_wall_type = wall_function
    gap = 0.0
    gap_factor = 0.0
    ....
}
```

The naming convention for the automatic generated surface outputs are as follows:

AUTO elelemtSetName1 wall

AUTO elelemtSetName1 interface

AUTO elelemtSetName1 internal

AUTO elelemtSetName2 internal

AUTO elelemtSetName2 wall

The scenarios of auto_wall are summarized in the following table:

Table 2:

Scenarios of auto_wall	Examples
Solid-Solid, Same Motion	Electronics cooling assembly with board +components attached.
Solid-Solid, Different Motion	Spinning solid cylinder embedded within a solid box.
Fluid-Solid, Same Motion	Conjugate heat transfer in a pipe (fluid/solid stationary or moving at same rate).
Fluid-Solid, Different Motion	Solid wheel (smooth tread) rotating in a stationary fluid volume.
Fluid-Solid, Fluid Reference Frame	Conjugate heat transfer analysis of spinning pipe. Solid exterior of pipe modeled using mesh motion, fluid interior of pipe modeled using a reference frame.
Fluid-Fluid, Same Motion	Internal surface in a pipe that is created for output/visualization purposes only.
Fluid-Fluid, Different Motion	Blower example with rotating impeller region and stationary outer region.
Solid-Void	Conjugate heat transfer in a pipe.
Fluid-Void, No Motion	External face of pipe flow case.
Fluid-Void, Motion	External face of pipe flow where the pipe is retained via MRF or mesh motion.

A simple boundary condition of type inflow allows the specification of several different kinds of common inflow boundary conditions. The particular kind is specified by *inflow_type*. For all inflow types, nodal boundary conditions are imposed as mentioned above. A velocity inflow type imposes nodal boundary conditions on the velocity, temperature, eddy viscosity, and all species variables. No element boundary conditions are imposed. The boundary conditions on the scalar variables are specified by the temperature parameters, plus an optional scaling by the corresponding *temperature_multiplier_function* parameters. Boundary conditions on the velocity may be specified in different coordinate systems, as determined by *inflow_velocity_type*. Values of cartesian, cylindrical, spherical, and normal are supported. These are used in the same way as for *wall_velocity_type*, see above.

Note: *normal_velocity* needs to be positive for flow into the fluid domain when normal is used. Defaults are used for unspecified variables.

For example, a constant-flow inlet with a time-varying temperature may be specified by:

```
SIMPLE_BOUNDARY_CONDITION( "inlet condition" ) {
    ...
    type                              = inflow
    inflow_type                       = velocity
    inflow_velocity_type              = cartesian
    x_velocity                        = 1
    eddy_viscosity                    = 1.e-03
    temperature                       = 300
    temperature_multiplier_function = "time varying"
}
MULTIPLIER_FUNCTION( "time varying" ) {
    type                              = piecewise_linear
    curve_fit_values                  = { 0, 1.0 ;
                                          10, 1.0 ;
                                          20, 1.1 ;
                                          40, 1.2 ;
                                          80, 1.5 ; }
    curve_fit_variable                = time
}
```

Inflow types of pressure and stagnation_pressure impose element boundary conditions on the pressure and stagnation pressure, respectively. The values used are given by the *pressure* and *stagnation_pressure* parameters, scaled by their respective multiplier function parameter *stagnation_pressure_multiplier_function* scales only the pressure part of the stagnation pressure. No boundary condition is imposed on any component of the velocity. The eddy viscosity as well as temperature and all species variables are specified, as in the velocity inflow type described above. Since it tends to be more stable, *stagnation_pressure* is preferred over *pressure*, as opposed to outlets where specifying the pressure is more stable. For example,

```
SIMPLE_BOUNDARY_CONDITION( "pressure inlet" ) {
   ...
   type                = inflow
   inflow_type         = stagnation_pressure
   stagnation_pressure = 1000
   eddy_viscosity      = 1.e-03
   temperature         = 300
}
```

An inflow type of mass_flux creates a normal velocity profile across the inflow surface such that the integrated mass flux is equal to *mass_flux*, scaled by *mass_flux_multiplier_function*. The velocity profile is based on either laminar or turbulent fully-developed arbitrary cross section pipe flow, depending on the equations specified. For turbulent flows, an eddy viscosity profile based on the same fully-developed flow is also imposed. Surface nodes with zero velocity, as imposed by other boundary conditions, are considered to be the walls of the inflow surface and as such are used to compute the profile. The normal velocity profile and the eddy viscosity profile are imposed as nodal boundary conditions. The two tangential velocity components are set to zero. A flow_rate inflow type is the same as mass_flux, except the parameters *flow_rate* and *flow_rate_multiplier_function* are used, and the quantity specified is the integrated volume flux. An average_velocity inflow type is also the same as *mass_flux*, except the parameters *average_velocity* and *average_velocity_multiplier_function* are used, and the quantity specified is the average normal velocity. For mass_flux (mass flow rate), flow_rate (volumetric flow rate), and average_velocity inflow types the inflow surface should be planar or close to it, otherwise the meaning of normal velocity becomes vague and the results unpredictable.

The temperature and all species variables are specified, as in the velocity inflow type described above. For example,

```
SIMPLE_BOUNDARY_CONDITION( "profiled inlet" ) {
   ...
   type          = inflow
   inflow_type   = mass_flux
   mass_flux     = 25
   temperature   = 300
}
```

For a supersonic inflow boundary condition, you need to specify all three variables. The simplest choice is primitive variables: velocity, pressure and temperature. Another choice might be: Mach number, pressure and temperature. In order to specify these conditions, the following two commands are introduced:

```
SIMPLE_BOUNDARY_CONDITION("superSonicInlet") {
   type  = inflow
   inflow_type = velocity_pressure_temperature

   or
   inflow_type = mach_pressure_temperature
}
```

When the specified inflow condition is subsonic, pressure is ignored, and only velocity or Mach number and temperature are used as the boundary conditions. If Mach < 1, velocity is computed by the given Mach number and temperature, while pressure is ignored. If Mach > 1, velocity is computed by the given Mach number and temperature, while pressure and temperature are fixed.

> **Note:** Both *inflow_type* = velocity_pressure_temperature and mach_pressure_temperature commands are only available for *flow* = compressible_navier_stokes.

A type outflow simple boundary condition normally, see below for the exception, imposes nodal boundary conditions on just the mesh displacement variables, see above. Element boundary conditions of type free are imposed on the tangential traction, turbulence flux, heat flux, and all the species fluxes. An element boundary condition of type outflow is imposed on the mass flux. A pressure element boundary condition is specified by the pressure parameter, plus an optional scaling by the *pressure_multiplier_function* parameter. For example, a typical outlet can be specified by:

```
SIMPLE_BOUNDARY_CONDITION( "outlet" ) {
 ...
 type     = outflow
 pressure = 0
}
```

Pressure loss for inflow, outflow, and far_field types may be modeled through the *pressure_loss_factor* parameter. The following term is added to the pressure element boundary condition, if one exists:

$$\frac{1}{2}\mathrm{sgn}(\boldsymbol{u}\bullet\boldsymbol{n})k\rho(\boldsymbol{u}\bullet\boldsymbol{n})^2 \qquad (179)$$

where k is given by *pressure_loss_factor*; ρ is the density; **u** is the velocity; **n** is the outward-pointing normal to the surface; and sgn**u**• **n** is -1 for inflow boundaries and +1 for outflow boundaries. If *pressure_multiplier_function* is given, it is applied to the pressure variable before this term is added. The pressure loss factor may be scaled with *pressure_loss_factor_multiplier_function*. For example,

```
SIMPLE_BOUNDARY_CONDITION( "outflow" ) {
   ...
   type                                      = outflow
   pressure                                  = 0
   pressure_loss_factor                      = 100
   pressure_loss_factor_multiplier_function = "pLoss_TMF"
   ...
}
MULTIPLIER_FUNCTION( "pLoss_TMF" ) {
   type                                    = piecewise_linear
   curve_fit_values                        = { 1   , 1 ;
                                               10 , 0 }
   curve_fit_variable                      = time_step
}
```

The hydrostatic pressure term may be added to pressure and stagnation pressure element boundary conditions by setting *hydrostatic_pressure*=on. This term is given by

$$\rho(\boldsymbol{x}-\boldsymbol{x}_0)\bullet\boldsymbol{g} \tag{180}$$

where ρ is the density; **x** is the current coordinate vector; **x**0 is any coordinate vector where the hydrostatic pressure is zero, given by *hydrostatic_pressure_origin*; and **g** is the gravity vector given by *body_force* in the parent ELEMENT_SET command. If there are no pressure or stagnation pressure element boundary conditions, then *hydrostatic_pressure* has no effect. If *pressure_multiplier_function* or *stagnation_pressure_multiplier_function* is given, it is applied before this term is added. If gravitational force is modeled using the GRAVITY command, then *hydrostatic_pressure*=on should be set for most boundaries where pressure or stagnation pressure boundary conditions are active in order to properly account for the hydrostatic pressure. This is most commonly needed for outflow boundaries. For free surface boundaries *hydrostatic_pressure* is normally set to off in order to ensure that the imposed pressure is constant along the entire surface. If this pressure or the nominal height of the free surface is nonzero, then *hydrostatic_pressure_origin* may be adjusted for any other boundary so that its pressure boundary condition evaluated at the nominal surface height equals the pressure imposed on the free surface. That is, $p_{\mathrm{BC}}+\rho(\boldsymbol{x}_{\mathrm{NOM}}-\boldsymbol{x}_0)\bullet\boldsymbol{g}=p_{\mathrm{FS}}$, where $\boldsymbol{x}_{\mathrm{NOM}}$ is a point on the nominal free surface, p_{BC} is the given pressure on a surface with *hydrostatic_pressure*=on, and p_{FS} is the given pressure on the free surface. For the following example, assume the nominal free surface height is $z_{\mathrm{NOM}}=2$:

```
GRAVITY( "gravity" ) {
    type                        = constant
    gravity                     = { 0, 0, -9.81 }
}
BODY_FORCE( "body force" ) {
    gravity                     = "gravity"
    ...
}
ELEMENT_SET( "flow elements" ) {
    body_force                  = "body force"
```

```
    ...
}
SIMPLE_BOUNDARY_CONDITION( "free surface" ) {
    type                          = free_surface
    ...
    pressure                      = 10
    hydrostatic_pressure          = off
}
SIMPLE_BOUNDARY_CONDITION( "outlet" ) {
    type                          = outflow
    ...
    pressure                      = 10
    hydrostatic_pressure          = on
    hydrostatic_pressure_origin = { 0, 0, 2 }
}
```

Reflections of waves from inflow and outflow boundaries can be reduced by setting *non_reflecting_factor* to a positive value. This parameter applies to inflow types with *inflow_type* = mass_flux and to outflow types. For inflows only the mass flux element boundary condition is modified and for outflows only the pressure element boundary condition is modified. It is ignored for all other cases. Also, there is no effect for the constant density model unless a positive *isothermal_compressibility* value is specified.

Setting *non_reflecting_factor* = 0 has no effect; the given element boundary condition is imposed. Using a value of one gives the full effect; waves pass through the boundary without reflection but at the cost of the variable drifting away from its specified value. Other values yield a blend between these effects. Values greater than one for *non_reflecting_factor* may lead to instabilities for the mass flux condition. If running average fields are defined, that is, *running_average* = on in the EQUATION command, then an enhanced non-reflecting boundary condition is used, leading to less reflection for the same value of *non_reflecting_factor*. The *non_reflecting_bc_running_average_field* needs to be turned on for this option. When *flow* = compressible_navier_stokes, the running_average option does not need to be turned on; the running average files are automatically used in non- reflecting boundary conditions.

A type slip simple boundary condition imposes a zero nodal boundary condition on the normal velocity in the given reference frame. An element boundary condition of type free is imposed on the pressure, which is important for curved surfaces. A plane of symmetry can also be specified with this type.

A type symmetry simple boundary condition is mathematically equivalent to type slip. The difference is that since a plane of symmetry must be flat a single normal is computed for the whole surface. The geometry of the given surface is checked and appropriate warning or error messages are issued if it is not flat. It is highly recommended that symmetry be used instead of slip for all surfaces that are planes of symmetry.

A type far_field boundary condition contains both a velocity-based inflow condition and a pressure-based outflow condition. At each point on the boundary the angle between the outward normal to the surface and the user-specified velocity is used to determine whether to impose the inflow or outflow conditions. The AcuSolve input for is a combination of the inflow and outflow types:

```
SIMPLE_BOUNDARY_CONDITION( "far field" ) {
    ...
    type           = far_field
    x_velocity     = 1
    y_velocity     = 0
    z_velocity     = 0
```

```
    temperature     = 0
    eddy_viscosity = 1.e-3
    pressure        = 0
}
```

Sometimes there is "back flow" over a portion of an outflow or far field boundary where the velocity is into the fluid domain. This may lead to instability of the temperature, species, viscoelastic stress, kinetic energy, eddy frequency, and eddy viscosity variables. Enabling *back_flow_conditions* allows nodal boundary conditions to be specified for these variables only on nodes where there is flow re-entering the domain. The *temperature_back_flow_type*, *species_1_back_flow_type* ,..., *species_9_back_flow_type*, *viscoelastic_xx_stress_back_flow_type*, ..., *viscoelastic_zx_stress_back_flow_type*, *kinetic_energy_back_flow_type*, *eddy_frequency_back_flow_type*, and *eddy_viscosity_back_flow_type* parameters specify how the constant nodal boundary conditions are to be calculated. A back flow type of value uses nodal boundary conditions specified by the *temperature*, *species_1*,..., *species_9*, *viscoelastic_xx_stress*, ..., *viscoelastic_zx_stress*, *kinetic_energy*, *eddy_frequency*, and *eddy_viscosity* parameters and their corresponding multiplier function parameters. Instead of specifying a value, various solution averages may be used. A back flow type of area_average averages the solution variable over the entire outflow surface and uses the result for the nodal boundary condition. The area averaged value for the scalar ϕ is defined using the following relationship:

$$\phi_A = \frac{\int \phi dA}{\int dA} \tag{181}$$

where A is the area.

A back flow type of exiting_area_average averages the solution variable only over the portions of the outflow surface where flow is exiting. This is defined by the following relationship:

$$\phi_{AE} = \frac{\int H_f \phi dA}{\int H_f dA} \tag{182}$$

where H_f is the heaviside step function defined as:

$$H_f = \begin{cases} 1 & U \cdot n > 0 \\ 0 & otherwise \end{cases} \tag{183}$$

A back flow type of bulk applies only to the temperature variable, and is defined as:

$$T_B = \frac{\int \rho U \cdot n H(T) dA}{\int \rho U \cdot n C_p dA} \tag{184}$$

where U is the velocity vector, ρ is the density, n is the surface normal vector, H is the enthalpy, Cp is the specific heat, and A is the area. A back flow type of exiting_bulk is also available for the temperature variable. When utilizing this option, the bulk temperature is averaged over only the portions of the surface where flow is re-entering the domain. The value used for the nodal constraint on temperature is computed as follows:

$$T_{BE} = \frac{\int \rho U \cdot n H(T) H_f dA}{\int \rho U \cdot n Cp H_f dA} \quad (185)$$

Back flow types of mass_flux_average and exiting_mass_flux_average are analogous to the bulk and exiting_bulk options for temperature, but utilize mass flux averaging and apply to all scalars except temperature. In the case of mass_flux_average, mass flux weighting is used, and the value is averaged over the entire outflow surface. The relationship used to compute the scalar value becomes:

$$\phi_M = \frac{\int \rho U \cdot n \phi dA}{\int \rho U \cdot n dA} \quad (186)$$

A back flow type of exiting_mass_flux_average is also available for all scalars except temperature. When utilizing this option, the scalar value is mass flux averaged over only the portions of the surface where flow is re-entering the domain. The value used for the nodal constraint on the scalar is computed as follows:

$$\phi_{ME} = \frac{\int \rho U \cdot n \phi H_f dA}{\int \rho U \cdot n H_f dA} \quad (187)$$

If the velocity is strictly out of the fluid domain over the entire outflow boundary then *back_flow_conditions* has no effect. For example,

```
SIMPLE_BOUNDARY_CONDITION( "outlet" ) {
    ...
    type                          = outflow
    pressure                      = 0
    back_flow_conditions          = on
    temperature_back_flow_type    = value
    temperature                   = 300.
    eddy_viscosity_back_flow_type = mass_flux_average
    species_1_back_flow_type      = exiting_mass_flux_average
}
```

A type free_surface simple boundary condition imposes nodal boundary conditions on the normal component of the mesh velocity. A pressure element boundary condition is specified by the pressure parameter, plus an optional scaling by the *pressure_multiplier_function* parameter.

The *surface_tension_model* parameter provides the surface tension information for type free_surface simple boundary conditions. This parameter specifies the user-given name of a SURFACE_TENSION_MODEL command. If this parameter is not given the surface tension defaults to

zero. If it is given, then in addition a contact angle at the side walls of the free surface may be specified with the *contact_angle_model* parameter. This parameter specifies the user-given name of a CONTACT_ANGLE_MODEL command. If *contact_angle_model* is not given, the contact angle defaults to 90 degrees. See the SURFACE_TENSION_MODEL command for details.

There are two primary mechanisms by which constraints can be applied to the displacement of the mesh for moving mesh simulations. The legacy approach uses the SIMPLE_BOUNDARY_CONDITION command to assign constraints for the transport variables governing the physics of the simulation, for example velocity field, turbulence fields and species equations, as well as the movement of the mesh, while the newer approach relies on the MESH_BOUNDARY_CONDITION command to control the motion of nodes. The MESH_BOUNDARY_CONDITION command enables separation of the physics of the problem from the mesh displacement controls to provide simplified case set up in the graphical user interface. The recommended modeling guideline is to use the SIMPLE_BOUNDARY_CONDITION command to assign constraints to the physics, but use the MESH_BOUNDARY_CONDITION to assign constraints to the motion of the nodes. The full set of mesh motion constraints associated with the SIMPLE_BOUNDARY_CONDITION command is still supported to provide full backward compatibility with legacy input files. However, it is recommended that these constraints be migrated to the MESH_BOUNDARY_CONDITION command.

The MESH_BOUNDARY_CONDITION command centralizes the options associated with mesh movement from the SIMPLE_BOUNDARY_CONDITION, free_surface, and EXTERNAL_CODE commands. This command is intended to act as the primary mechanism for specifying control of the mesh movement. For backward compatibility, the mesh movement related options from the SIMPLE_BOUNDARY_CONDITION, free_surface, and EXTERNAL_CODE commands are still supported by internally creating a MESH_BOUNDARY_CONDITION command for every SIMPLE_BOUNDARY_CONDITION command that is present in the input file. AcuPrep internally creates a table of constraints for each element face/node on the boundaries and identifies conflicts within the mesh motion boundary conditions. These conflicts may arise due to by the combinations of MESH_BOUNDARY_CONDITION and the legacy commands that provide similar functionality. In the case of conflicting constraints that have equal precedence values, the MESH_BOUNDARY_CONDITION is given priority. If the precedence values are not equal, then the higher precedence constraint is enforced. AcuPrep also checks for compatibility of the mesh motion boundary conditions on each node with the boundary conditions applied for the physics through the SIMPLE_BOUNDARY_CONDITION command. The following table is used when checking these compatibilities:

SIMPLE_BC type	**MESH_BOUNDARY_CONDITION type**						
-	fixed	planar_slip	mesh_motio	free_surface	guide_surfac	flexible_bod	external_code
wall	OK	OK	OK	Error	OK	OK	OK
inflow	OK	OK	OK	Error	OK	Warning	Warning
outflow	OK	OK	OK	Error	OK	Warning	Warning
slip	OK	OK	OK	Error	OK	OK	OK
symmetry	OK	OK	Error	Error	Error	Warning	Error
far_field	OK	OK	OK	Error	OK	Error	Warning

SIMPLE_BC type	**MESH_BOUNDARY_CONDITION type**						
free_surface	Error	Error	Error	OK	Error	Warning	Error

The mesh displacement boundary conditions defined above for every type is zero displacement with respect to the surface; that is, mesh nodes move with the surface. This corresponds to *mesh_displacement_type* = fixed, the default. Other behaviors may be obtained by setting *mesh_displacement_type* to none, slip, guide_surface, or flexible_body. The first of these removes all mesh displacement boundary conditions from the set. The second removes these boundary conditions and replaces them with a single condition: zero normal displacement with respect to the surface. The third is similar to slip, but the mesh displacement boundary conditions are such that the nodes slip along a guide surface given by the *guide_surface* parameter, which specifies a user-given name of a GUIDE_SURFACE command in the input file. Refer to the GUIDE_SURFACE command for further information and an example. The last of these, *mesh_displacement_type* = flexible_body, is used to specify mesh motion as part of a Fluid-Structure Interaction (FSI) problem. For example,

```
SIMPLE_BOUNDARY_CONDITION( "flexible surface" ) {
    ...
    type                    = wall
    mesh_displacement_type  = flexible_body
    flexible_body           = "flexible body"
    nodal_x_modes           = Read( "channel.wet_surf.xmd" )
    nodal_y_modes           = Read( "channel.wet_surf.ymd" )
    nodal_z_modes           = Read( "channel.wet_surf.zmd" )
}
FLEXIBLE_BODY( "flexible body" ) {
    equation               = mesh_displacement
    num_modes              = 5
 ...
}
```

where the files channel.wet_surf.xmd, channel.wet_surf.ymd, and channel.wet_surf.zmd correspond to the nodal modes in the x, y, and z-directions. The format of these files the same as nodal_modes in the NODAL_BOUNDARY_CONDITION command. See also the FLEXIBLE_BODY command for details. The nodes in these files must match those referenced in *surfaces*. If the problem has a deforming mesh (that is, *mesh* is set to arbitrary_lagrangian_eulerian or specified in the EQUATION command), and *mesh_displacement_type* = flexible_body and *wall_velocity_type* = match_mesh_velocity (the default), then the given velocity conditions are ignored. Instead, the velocity is set to match the velocity of the mesh.

Note: AcuPev may be used to obtain the nodal mode files.

Mesh displacement boundary conditions may also be specified with *mesh_motion*. The value of this parameter refers to the user-given name of a MESH_MOTION command in the input file. This command is evaluated resulting in a mesh displacement vector. The *mesh_motion* parameter has an effect only if there is an active mesh equation and *mesh_displacement_type* is set to fixed or flexible_body.

The same parameter may be specified in the parent `ELEMENT_SET` command, but the one here takes precedence unless precedence is set to less than zero. For example,

```
SIMPLE_BOUNDARY_CONDITION( "moving wall" ) {
   ...
   type                   = wall
   precedence             = 1
   surfaces               = Read( "wall.ebc" )
   mesh_displacement_type = fixed
   mesh_motion            = "moving body"
}
MESH_MOTION( "moving body" ) {
   type                   = rigid_body
   ...
}
```

A flexible body may be combined with a rotating mesh. For example, to model a deformable impeller with sliding interfaces, for example, a wind turbine, the following commands may be used:

```
FLEXIBLE_BODY( "BladeA" ) {
    equation               = mesh_displacement
    ...
    surface_outputs        = { "BladeA" }
}
MESH_MOTION( "Rotation" ) {
    type                   = rotation
    ...
    rotation_variable      = time
}
SIMPLE_BOUNDARY_CONDITION( "BladeA" ) {
    ...
    mesh_displacement_type = flexible_body
    flexible_body          = "BladeA"
    nodal_x_modes          = Read( "BladeA.xmd" )
    nodal_y_modes          = Read( "BladeA.ymd" )
    nodal_z_modes          = Read( "BladeA.zmd" )
    mesh_motion            = "Rotation"
}
```

In some circumstances it is necessary to "turn off" or "deactivate" previously-defined boundary conditions. This is accomplished by setting the active flag through *active_type*. A value of all is the default and means that all surfaces in the set are active (active flag = 1) and will have the boundary conditions described above imposed on them. A value of none means that no surface is active (active flag = 0).

A no_interface active type is used to automatically turn on and off boundary conditions in the presence of interfaces. Boundary conditions are applied (active flag = 1) if the surface does not belong to an `INTERFACE_SURFACE` set or if it does not find a contact surface of the appropriate medium. That is, each surface and boundary condition will fall into one of five possible cases:

- The surface is not in any `INTERFACE_SURFACE` set; active flag = 1.
- The surface is in an `INTERFACE_SURFACE` set, but it does not find a contact surface; active flag = 1.
- The surface finds a contact surface of fluid medium; active flag = 0.
- The surface finds a contact surface of solid/shell medium, and the boundary condition variable is defined on the solid/shell side, currently only temperature and the heat flux variables,; active flag = 0.

- The surface finds a contact surface of solid/shell medium, and the boundary condition variable is not defined on the solid/shell side, that is, it is not temperature or one of the heat flux variables,; active flag = 1.

Turbulence inflow quantities are specified according to the *turbulence_input_type* and turbulence model dependent properties. Legacy usage for direct specification of the *eddy_viscosity* variable for the spalart_allmaras based turbulence models or *kinetic_energy* and *eddy_frequency* for the sst and k_omega based turbulence models is accomplished with *turbulence_input_type* set to direct. As follows:

```
SIMPLE_BOUNDARY_CONDITION( "far field" ) {
    ...
    type                    = far_field
    x_velocity              = 1
    y_velocity              = 0
    z_velocity              = 0
    turbulence_input_type   = direct
    eddy_viscosity          = 1.e-3
}
SIMPLE_BOUNDARY_CONDITION( "inflow" ) {
    ...
    type                   = inflow
    inflow_type            = velocity
    inflow_velocity_type   = normal
    normal_velocity        = 31.1# m/sec
    turbulence_input_type  = direct
    kinetic_energy         = 0.9067 # m2/sec2
    eddy_frequency         = 2.55E+5
}
```

The following input scenarios are also supported for *turbulence_input_type* = direct:

- *turbulence_input_type* = auto, used to automatically assign turbulent state based on selected turbulence equation, *turbulence_intensity_type* and *turbulence_flow_type*. Valid for all turbulence models.
- *turbulence_input_type* = intensity_length_scale, where the turbulent state at the inflow is approximated by the *turbulence_intensity_type* and length_scale. Valid for all turbulence models.
- *turbulence_input_type* = intensity_viscosity_ratio, where the turbulent state at the inflow is approximated by the *turbulence_intensity_type* and viscosity_ratio. Valid for all two-equation turbulence models.
- *turbulence_input_type* = viscosity_ratio, where the turbulent state at the inflow is approximated by the viscosity_ratio. Valid only for the spalart_allmaras based turbulence models.

The following relationships are used to compute the turbulent quantities for each turbulence model:

Spalart Allmaras turbulent eddy viscosity $$\nu_t = \sqrt{\frac{3}{2}} l \bar{u} I \quad (188)$$

SST, k-ω, Standard k-ε, Realizable k-ε, RNG k-ε turbulent kinetic energy

$$k = \frac{3}{2}(\bar{u}\, I)^2 \tag{189}$$

SST, k-ω eddy frequency $$\omega = \frac{C_\mu k}{\beta \nu}$$ or

$$\omega = \frac{\sqrt{k}}{l} \tag{190}$$

Standard k-ε, Realizable k-ε, RNG k-ε dissipation rate $$\varepsilon = C_\mu \frac{k^{3/2}}{l}$$ or

$$\varepsilon = \frac{C_\mu k^2}{\beta \nu} \tag{191}$$

Where ν_t is the turbulent kinematic viscosity, l is the turbulence length scale, I is the turbulence intensity, $\bar{u}$ is the velocity scale, k is the turbulent kinetic energy, C_μ=0.09, β is the ratio of turbulent to molecular viscosity, and ε is the dissipation rate. Given these relations, $l = C_\mu k^{3/2}/\varepsilon$ for the standard, RNG, realizable $k-\varepsilon$ models, while $l = \sqrt{k}/\omega$ for the $k-\omega$ and SST models. The following example illustrates the turbulence specifications when a spalart_allmaras based turbulence models is active:

```
SIMPLE_BOUNDARY_CONDITION( "farfield" ) {
  ...
  type                      = far_field
  x_velocity                = 14.5 # m/sec
  turbulence_input_type     = auto
  turbulence_flow_type      = external
  turbulence_intensity_type = auto
}
```

The following example illustrates the turbulence specifications for *turbulence_flow_type* = internal when a spalart_allmaras based turbulence model is active, using *turbulence_input_type* = intensity_length_scale:

```
SIMPLE_BOUNDARY_CONDITION( "internal_inflow" ) {
   ...
   type                        = inflow
   inflow_type                 = velocity
   inflow_velocity_type        = cartesian
   x_velocity                  = 14.5 # m/sec
   turbulence_input_type       = intensity_length_scale
   turbulence_flow_type        = internal
   turbulence_intensity_type   = auto
   turbulence_length_scale     = 0.1 #m
}
```

The following example illustrates the turbulence specifications when one of the two-equation based turbulence models is active, using *turbulence_input_type* = intensity_viscosity_ratio:

```
SIMPLE_BOUNDARY_CONDITION( "two_equation_farfield" ) {
    ...
    type = far_field
    x_velocity                    = 0.999847 # m/sec
    y_velocity                    = 0.017452 # m/sec
    z_velocity                    = 0.0           # m/sec
    pressure                      = 0.0           # N/m2
    turbulence_input_type         = intensity_viscosity_ratio
    turbulence_intensity_type     = value
    percent_turbulence_intensity  = 0.025
    turbulence_viscosity_ratio    = 3.0
}
```

The following example illustrates the turbulence specifications when the Spalart Allmaras based turbulence models are active, using *turbulence_input_type* = viscosity_ratio:

```
SIMPLE_BOUNDARY_CONDITION( "one_equation_farfield" ) {
    ...
    type                        = far_field
    x_velocity                  = 14.5 # m/sec
    turbulence_input_type       = viscosity_ratio
    turbulence_viscosity_ratio = 3.0
}
```

The following example illustrates the turbulence specifications when one of the two-equation based turbulence models are active, using *turbulence_input_type* = auto and the *turbulence_intensity_type* = low:

```
SIMPLE_BOUNDARY_CONDITION( "two_equation_pressure" ) {
    ...
    type                        = inflow
    inflow_type                 = pressure
    pressure                    = 101325.0 #Pa
    turbulence_input_type       = auto
    turbulence_intensity_type   = low
    turbulence_velocity_scale   = 2.0
}
```

The turbulence specifications are turbulence model dependent and change if the simulation includes a turbulence transition model. The following tables outline the available options for each turbulence model selection with or without the inclusion of a transition model when using inflow type, with velocity inflow_type or farfield type:

Table 3: Spalart Allmaras, DDES-SA

turbulence_input_type	direct
Input Requirements	eddy_viscosity
turbulence_intensity_type	N/A N/A
turbulence_flow_type	N/A

turbulence_input_type	auto
Input Requirements	*turbulence_flow_type*
turbulence_intensity_type	N/A
turbulence_flow_type	internal external
turbulence_viscosity_ratio	40.0 3.0

turbulence_input_type	intensity_length_scale
Input Requirements	turbulence_intensity_type length_scale
turbulence_intensity_type	auto low/medium/high value N/A
Input Requirements	turbulence_flow_type turbulence_flow_type

	percent_turbulence_intensity length_scale
turbulence_flow_type	internal external internal external N/A
percent_turbulence_intensity	4.0 0.1 1.0/4.0/10.0 0.1/0.5/1.0 As specified

turbulence_input_type	viscosity_ratio
Input Requirements	turbulence_viscosity_ratio
turbulence_viscosity_ratio	As specified

Table 4: Spalart Allmaras - Gamma/Gamma-Re-Theta

Input Requirements	*turbulence_intensity_type*
turbulence_intensity_type	auto low/medium/high value
percent_turbulence_intensity	0.1 0.1/0.5/1.0 As specified

Table 5: SST, K-ω, K-ε, BSL K-ω, DDES-SST

turbulence_input_type	direct

Input Requirements	kinetic_energy eddy_frequency dissipation_rate
turbulence_intensity_type	N/A
turbulence_flow_type	N/A

turbulence_input_type	auto	
Input Requirements	*turbulence_intensity_type*	*turbulence_flow_type*
turbulence_intensity_typ	auto low/medium/high value	
Input Requirements	*turbulence_flow_type* *turbulence_flow_type* *percent_turbulence_intensity*	
turbulence_flow_type	internal external internal external N/A	
percent_turbulence_inte	4.0 0.5 1.0/4.0/10.0 0.1/0.5/1.0 As specified	

turbulence_input_type	*intensity_length_scale*	
Input Requirements	*turbulence_intensity_type*	*length_scale*
turbulence_intensity_type	auto	

	low/medium/high value	
Input Requirements	*turbulence_flow_type* *turbulence_flow_type* *percent_turbulence_intensity*	*length_scale*
turbulence_flow_type	internal external internal external N/A	
percent_turbulence_intensi	4.0 0.5 1.0/4.0/10.0 0.1/0.5/1.0 As specified	

turbulence_intensity_type	*intensity_viscosity_ratio*
Input Requirements	*turbulence_intensity_type* viscosity_ratio
turbulence_intensity_type	auto low/medium/high value
Input Requirements	*turbulence_flow_type* *turbulence_flow_type* *percent_turbulence_intensity*
turbulence_flow_type	internal external internal external N/A

percent_turbulence_intensit	4.0 0.5 1.0/4.0/10.0 0.1/0.5/1.0 As specified

SST-Gamma/Gamma-Re-Theta

turbulence_input_type	direct
Input Requirements	kinetic_energy eddy_frequency dissipation rate
turbulence_intensity_type	N/A
turbulence_flow_type	N/A

turbulence_intensity_type	*intensity_viscosity_ratio*
Input Requirements	*turbulence_intensity_type* viscosity_ratio
turbulence_intensity_type	auto low/medium/high value
Input Requirements	*turbulence_flow_type* *turbulence_flow_type* *percent_turbulence_intensity* *viscosity_ratio*
turbulence_flow_type	internal external internal external N/A

percent_turbulence_intensity	4.0 0.5 1.0/4.0/10.0 0.1/0.5/1.0 As specified

turbulence_input_type	*intensity_length_scale*
Input Requirements	*turbulence_intensity_type length_scale*
turbulence_intensity_type	auto low/medium/high value
Input Requirements	*turbulence_flow_type* *turbulence_flow_type* *percent_turbulence_intensity* *length_scale*
turbulence_flow_type	internal external internal external N/A
percent_turbulence_intensit	4.0 0.5 1.0/4.0/10.0 0.1/0.5/1.0 As specified

turbulence_intensity_type	*intensity_viscosity_ratio*
Input Requirements	*turbulence_intensity_type*

	viscosity_ratio
turbulence_intensity_type	auto low/medium/high value
Input Requirements	*turbulence_flow_type* *turbulence_flow_type* *percent_turbulence_intensity* *viscosity_ratio*
turbulence_flow_type	internal external internal external N/A
percent_turbulence_intensity	4.0 0.5 1.0/4.0/10.0 0.1/0.5/1.0 As specified

Atmospheric Boundary Layer Inflow

When the inflow type is set to atmospheric, inlet velocity profile and turbulence inflow quantities are generated automatically. Inlet velocity is prescribed via log law while inlet turbulence properties are computed according to the chosen turbulence model. Necessary inputs are surface roughness and reference velocity. Below are two examples of how to set inflow atmospheric boundary layer:

```
SIMPLE_BOUNDARY_CONDITION( "abl_inflow" ) {
   ...
   type                                        = inflow
   inflow_type                                 = atmospheric
   atmospheric_roughness_type                  = user_value
   atmospheric_ground_roughness                = 0.001
   atmospheric_reference_velocity_type         = friction_velocity
   atmospheric_friction_velocity               = 1.0
}
```

```
SIMPLE_BOUNDARY_CONDITION( "abl_inflow" ) {
    ...
    type                                       = inflow
```

```
    inflow_type                                 = atmospheric
    atmospheric_roughness_type                  = suburb
    atmospheric_reference_velocity_type         = sample_height_velocity
    atmospheric_sample_velocity                 = 5.0
    atmospheric_sample_height                   = 10.0
}
```

Corresponding velocity and turbulence boundary conditions at inflow will be generated automatically. There should be an empirical ratio (30) between inflow roughness and standard wall roughness. For example, if atmospheric boundary layer inflow has a roughness of 0.001m, on the ground the standard turbulence wall function should have a roughness of 0.03m.

Topology Optimization

Simple boundary conditions are automatically applied on the adjoint velocity and pressure for topology optimization for minimum mechanical energy. Aiming at the design of ducts and manifolds and generally for internal flows, you can only have *type*=inflow, *type*=outflow or *type*=wall or auto_wall. For inflow the choices are: *inflow_type*=velocity, *inflow_type*=flow_rate or *inflow_type*=mass_flux.

SOLAR_RADIATION_SURFACE

Specifies a solar radiation heat flux condition on a surface.

Type

AcuSolve Command

Syntax

SOLAR_RADIATION_SURFACE("name") {parameters...}

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral
six_node_triangle or tri6	Six-node triangle.

element_set or *elem_set (string) [no default]*

User-given name of the parent element set.

surfaces (array) [no default]

List of element surfaces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this command. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to the SOLAR_RADIATION_SURFACE command. This option is used in place of directly specifying these parameters. In the event that both of the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

side (enumerated) [=default]

Side of the surface that receives solar heat flux.

default	Side is based on the element medium.
inward	Side facing into the element.
outward	Side facing away from the element.
both	Both sides.

solar_radiation_model (string) [no default]
User-given name of the solar radiation model.

Description

This command specifies a solar radiation heat flux condition on a set of surfaces (element faces). This condition is coupled to all other solar radiation surfaces, but not to radiation surfaces or radiation element boundary conditions. The SOLAR_RADIATION command provides a detailed description of this coupling.

The surfaces of a solar radiation surface are defined with respect to the elements of an element set. For example,

```
ELEMENT_SET( "interior" ) {
    shape                   = four_node_tet
    medium                  = fluid
    elements                = { 1, 8, 3, 4, 9 ;
                                3, 3, 4, 9, 5 ;
                                ... }
 ...
}
SOLAR_RADIATION_SURFACE( "wall" ) {
    shape                   = three_node_triangle
    element_set             = "interior"
    surfaces                = { 1, 12, 9, 3, 4 ;
                                3, 52, 5, 3, 4 ; }
    side = default
    solar_radiation_model   = "solar reflectivity"
}
```

specifies a solar radiation heat flux condition to be applied to the default side of each of two surfaces of the element set "interior" using the solar radiation model "solar reflectivity".

There are two main forms of this command. The legacy version (or single topology version) of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad

Element Shape	**Surface Shape**
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number (of the parent element set), a unique (within this set) surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as `wall.srf`:

```
1 12 9 3 4
3 52 5 3 4
```

and read by:

```
SOLAR_RADIATION_SURFACE ( "wall" ) {
   shape        = three_node_triangle
   element_set  = "interior"
   surfaces     = Read( "wall.srf" )
   ...
}
```

The mixed topology form of the `SOLAR_RADIATION_SURFACE` command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the `SURFACE_SET` command, and are then referred to by the `SOLAR_RADIATION_SURFACE` command. For example, a collection of triangular and quadrilateral element faces can be defined using the following `SURFACE_SET` commands.

```
SURFACE_SET( "tri faces" ) {
   surfaces        = { 1, 1, 1, 2, 4 ;
                       2, 2, 3, 4, 6 ;
                       3, 3, 5, 6, 8 ; }
   shape           = three_node_triangle
   volume_set      = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
   surfaces        = { 1, 1, 1, 2, 4, 9 ;
                       2, 2, 3, 4, 6, 12 ;
                       3, 3, 5, 6, 8, 15 ; }
   shape           = four_node_quad
   volume_set      = "prisms"
```

Then, a single SOLAR_RADIATION_SURFACE command is defined that contains the tri and quad faces as follows:

```
SOLAR_RADIATION_SURFACE ( "wall" ) {
   surface_sets       = {"tri_faces", "quad_faces"}
   ...
}
```

The list of surface sets can also be placed in a file, such as surface_sets.srfst:

```
tri faces
quad faces
```

and read using:

```
SOLAR_RADIATION_SURFACE ( "wall" ) {
   surface_sets       = Read("surface_sets.srfst")
   ...
}
```

The mixed topology version of the SOLAR_RADIATION_SURFACE command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *surface_sets* and *surfaces* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

The *side* parameter specifies which side of the surface is active. If the solar radiation comes from the opposite side then the surface does not interact with this radiation, that is, the surface is completely transparent. An inward side faces towards the interior of its parent element, and an outward side faces away from the element. A value of both means that both the inward and outward sides are active. A default side is chosen based on the *medium* parameter of the ELEMENT_SET command. For a fluid medium, default is the same as inward, and for *solid* and *shell* media, default is the same as outward.

For example, a fluid-only thermal simulation of the interior of a car may include the effects of solar radiation falling on the roof and reflecting around on the inside with the following commands:

```
VOLUME_SET("interior"){
    shape                   = eight_node_brick
    elements                = { 1, 1, 2, 3, 4, 5, 6, 7, 8 ;
                                ... }
}
SURFACE_SET("roof"){
    shape                   = four_node_quad
    surfaces                = { 1, 1, 1, 2, 3, 4 ;
                                ... }
    volume_set              = "interior"
}
ELEMENT_SET( "air" ) {
    medium                  = fluid
    volume_sets             = {"interior"}
    ...
}
SOLAR_RADIATION_SURFACE( "interior side" ) {
    surface_sets            = {"roof"}
```

```
    side                        = inward
    solar_radiation_model       = "headliner cloth"
}
SOLAR_RADIATION_SURFACE( "exterior side" ) {
    surface_sets                = {"roof"}
    side                        = outward
    solar_radiation_model       = "exterior paint"
}
SOLAR_RADIATION_MODEL( "headliner cloth" ) {
    specular_transmissivity     = 0.0
    diffuse_transmissivity      = 0.0
    specular_reflectivity       = 0.05
    diffuse_reflectivity        = 0.4
}
SOLAR_RADIATION_MODEL( "exterior paint" ) {
    specular_transmissivity     = 0.0
    diffuse_transmissivity      = 0.0
    specular_reflectivity       = 0.6
    diffuse_reflectivity        = 0.1
}
```

The surface of the interior fluid elements that lie against the inside of the roof are split into inward and outward pointing sides. The inward pointing side models the properties of the headliner cloth, and the outward pointing side models the exterior paint on the roof. This avoids directly modeling the roof itself. However, it may be important to do so if the thermal resistance of the roof is significant.

In the roof example a both side could have been used if both solar radiation models were the same, but otherwise there is only one good way to model the roof. In other physical problems there may be a variety of ways of modeling solar radiation. If this is so, then modeling both sides of all possible surfaces is normally the most accurate model, but may be only slightly more accurate than simpler models at significantly greater cost and complexity. For example, consider modeling a glass windshield with one layer of brick elements. Since reflections occur on both sides of the glass for solar radiation coming from either direction, the most accurate model for one of these elements may look like:

```
VOLUME_SET("glass"){
    shape                       = eight_node_brick
    elements                    = { 1, 1, 2, 3, 4, 5, 6, 7, 8 ;
                                    ... }
}
SURFACE_SET("windshield"){
    shape                       = four_node_quad
    surfaces                    = { 1, 1, 1, 2, 3, 4 ;
                                    ... }
    volume_set                  = "glass"
}
ELEMENT_SET( "glass" ) {
    medium                      = solid
    volume_sets                 = {"glass"}
    ...
}
SOLAR_RADIATION_SURFACE( "both windshield surfaces" ) {
    surface_sets                = {"glass"}
    side                        = both
    solar_radiation_model       = "windshield, one surface"
}
SOLAR_RADIATION_MODEL( "windshield, one surface" ) {
    specular_transmissivity     = 0.9
    specular_reflectivity       = 0.1
}
```

But for thin windshields, there is little loss of accuracy by lumping reflections from both surfaces onto one as follows:

```
SOLAR_RADIATION_SURFACE( "lumped windshield surfaces" ) {
    ...
    side                        = both
    solar_radiation_model       = "windshield, both surfaces"
}
SOLAR_RADIATION_MODEL( "windshield, both surfaces" ) {
    specular_transmissivity     = 0.81
    specular_reflectivity       = 0.19
}
```

Another way to model this is to use both surfaces but only one side from each:

```
SOLAR_RADIATION_SURFACE( "both windshield surfaces, cheaper model" ) {
   ...
   side                   = outward
   solar_radiation_model = "windshield, both surfaces"
}
```

For all three of the above models there would be no solar radiation surfaces on the fluid elements on either side of the windshield. When a fluid element is adjacent to an unmodeled solid wall, such a surface would normally be created and assigned the solar radiation properties of the solid wall, as in the roof example.

All data from all `SOLAR_RADIATION_SURFACE` commands are preprocessed to form the solar heat flux for each of several given points in time (see the `SOLAR_RADIATION` command). This data is then interpolated for each time step during the simulation.

Since the solar radiation data is preprocessed, its content is written to disk. Once the problem has run, its values for all solved time steps can be translated to other formats using the AcuTrans program and other post-processing modules; see the *AcuSolve Programs Reference Manual* for details.

SURFACE_INTEGRATED_CONDITION

Specifies a condition on the surface integral of a solution variable.

Type

AcuSolve Command

Syntax

```
SURFACE_INTEGRATED_CONDITION("name") {parameters...}
```

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral
six_node_triangle or tri6	Six-node triangle.

element_set or elem_set (string) [no default]

User-given name of the parent element set.

surfaces (array) [no default]

List of element surfaces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this command. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to the SURFACE_INTEGRATED_CONDITION command. This option is used in place of directly specifying these parameters. In the event that both of the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

variable or var (enumerated) [no default]

Boundary condition variable.

mass_flux or mass	Mass flux.
temperature or temp	Temperature.
species_1 or spec1	Species 1.
species_2 or spec2	Species 2.

species_3 or spec3	Species 3.
species_4 or spec4	Species 4.
species_5 or spec5	Species 5.
species_6 or spec6	Species 6.
species_7 or spec7	Species 7.
species_8 or spec8	Species 8.
species_9 or spec9	Species 9.
field	Field.
kinetic_energy or tke	Turbulence kinetic energy.
eddy_frequency or tomega	Turbulence eddy frequency.
dissipation_rate or teps	Dissipation rate.
intermittency or tintc	Intermittency.
transition_re_theta or treth	Transition Re-theta.

type (enumerated) [=constant]
Type of the boundary surface.

constant or const	Constant value. Requires *constant_value*.

constant_value or *value (real) [=0]*
Constant value of the boundary condition. Used with constant type.

field (string) [no default]
Value of the boundary condition for field. Used with *variable* field.

multiplier_function (string) [=none]
User-given name of the multiplier function for scaling the boundary condition values. If none, no scaling is performed.

Description

This command specifies a condition on the surface integral of a solution variable. Bulk (mass-averaged) definitions are used for the temperature and species variables. The bulk temperature on a surface is defined by:

$$T_{bulk} = \frac{\int_S T\boldsymbol{n} \bullet (\rho \boldsymbol{u}) dA}{\int_S \boldsymbol{n} \bullet (\rho \boldsymbol{u}) dA} \qquad (192)$$

where T is the temperature, ρ is the density, **u** is the velocity, **n** is the outward-pointing normal to the surface, and S is the surface defined by *surfaces*. Similar definitions are used for the species variables. The mass_flux variable is given by the denominator of the above equation.

The surface is defined by a set of surfaces (element faces) with respect to the elements of an element set. For example,

```
ELEMENT_SET( "flow elements" ) {
    shape          = four_node_tet
    elements       = { ...
                       4, 2, 5, 6, 8 ;
                       5, 2, 6, 3, 5 ;
                       ... }
    ...
}
SURFACE_INTEGRATED_CONDITION( "total mass flux at inlet" ) {
    shape          = three_node_triangle
    element_set    = "flow elements"
    surfaces       = { 4, 41, 2, 5, 6 ;
                       5, 51, 5, 6, 3 ; }
    variable       = mass_flux
    type           = contant
    constant_value = 12
}
```

defines a surface integrated condition, with a constant value of 12 for the integral of mass flux over two surfaces of the element set "flow elements".

There are two main forms of this command. The legacy version (or single topology version) of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number (of the parent element set), a unique (within

this set) surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as `inlet.ebc`:

```
4 41 2 5 6
5 51 5 6 3
```

and read by:

```
SURFACE_INTEGRATED_CONDITION( "inlet" ) {
 shape          = three_node_triangle
 element_set    = "flow elements"
 surfaces       = Read( "inlet.ebc" )
 variable       = mass_flux
 type           = constant
 constant_value = 12.
}
```

The mixed topology form of the `SURFACE_INTEGRATED_CONDITION` command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the `SURFACE_SET` command, and are then referred to by the `SURFACE_INTEGRATED_CONDITION` command. For example, a collection of triangular and quadrilateral element faces can be defined using the following `SURFACE_SET` commands.

```
SURFACE_SET( "tri faces" ) {
    surfaces    = { 1, 1, 1, 2, 4 ;
                    2, 2, 3, 4, 6 ;
                    3, 3, 5, 6, 8 ; }
    shape       = three_node_triangle
    volume_set  = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
    surfaces    = { 1, 1, 1, 2, 4, 9 ;
                    2, 2, 3, 4, 6, 12 ;
                    3, 3, 5, 6, 8, 15 ; }
    shape       = four_node_quad
    volume_set  = "prisms"
```

Then, a single `SURFACE_INTEGRATED_CONDITION` command is defined that contains the tri and quad faces as follows:

```
SURFACE_INTEGRATED_CONDITION ( "inlet" ) {
   surface_sets       = {"tri_faces", "quad_faces"}
   ...
}
```

The list of surface sets can also be placed in a file, such as `surface_sets.srfst`:

```
tri faces
quad faces
```

and read using:

```
SURFACE_INTEGRATED_CONDITION ( "inlet" ) {
    surface_sets       = Read("surface_sets.srfst")
    ...
}
```

The mixed topology version of the `SURFACE_INTEGRATED_CONDITION` command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *surface_sets* and *surfaces* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

The variable to which the integrated condition applies is given by *variable*. If the problem does not solve for a field associated with this variable, the condition is simply ignored. A separate `SURFACE_INTEGRATED_CONDITION` command must be issued for each variable and type.

The quadrature rule for the surface integral is inherited from the parent element set.

A constant type of surface integrated condition specifies that the surface integral of the given variable is equal to *constant_value*, as in the above example.

The *multiplier_function* parameter may be used to uniformly scale the surface integrated condition values. The value of this parameter refers to the user-given name of a `MULTIPLIER_FUNCTION` command in the input file. For example, to impose a condition on the integrated mass flux as a function of time, the following commands may be issued:

```
SURFACE_INTEGRATED_CONDITION( "time-varying mass flux" ) {
    ...
    variable            = mass_flux
    type                = constant
    constant_value      = 2.5
    multiplier_function = "time varying"
}
MULTIPLIER_FUNCTION( "time varying" ) {
    type                = piecewise_linear
    curve_fit_values = { 0, 0.0 ;
                         10, 1.0 ;
                         20, 0.5 ;
                         40, 0.7 ;
                         80, 1.2 ; }
    curve_fit_variable  = time
}
```

SURFACE_OUTPUT

Specifies output parameters for a boundary surface.

Type

AcuSolve Command

Syntax

```
SURFACE_OUTPUT("name") {parameters...}
```

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral
six_node_triangle or tri6	Six-node triangle.

element_set or *elem_set (string) [no default]*

User-given name of the parent element set.

surfaces (array) [no default]

List of element surfaces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this command. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to this command. This option is used in place of directly specifying these parameters. In the event that both of the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

integrated_output_frequency or *intg_freq (integer) >=0 [=1]*

Time step frequency at which to output integrated surface quantities. If zero, this option is ignored.

integrated_output_time_interval or *intg_intv (real) >=0 [=0]*

Time frequency at which to output integrated surface quantities. If zero, this option is ignored.

statistics_output_frequency or *stat_freq (integer) >=0 [=1]*

Time step frequency at which to output statistics of surface nodal quantities. If zero, this option is ignored.

statistics_output_time_interval or *stat_intv* *(real) >=0 [=0]*
: Time frequency at which to output statistics of surface nodal quantities. If zero, this option is ignored.

nodal_output_frequency or *nodal_freq* *(integer) >=0 [=0]*
: Time step frequency at which to output surface quantities at the nodes of the surface. If zero, this option is ignored.

nodal_output_time_interval or *nodal_intv* *(real) >=0 [=0]*
: Time frequency at which to output surface quantities at the nodes of the surface. If zero, this option is ignored.

num_saved_states or *save* *(integer) >=0 [=0]*
: Number of solution steps to retain on disk. If zero, all are kept.

Description

This command specifies a set of boundary surfaces (element faces) whose surface quantities are to be output. For example,

```
ELEMENT_SET( "duct" ) {
    shape                              = eight_node_brick
    elements                           = { 1, 1, 2, 4, 3, 11, 12, 14, 13 ;
                                           2, 3, 4, 6, 5, 13, 14, 16, 15 ;
                                           ... }
    ...
}
SURFACE_OUTPUT( "lower wall" ) {
    shape                              = four_node_quad
    element_set                        = "duct"
    surfaces                           = { 1, 101, 1, 3, 11, 13 ;
                                           2, 102, 3, 5, 13, 15 ; }
    integrated_output_frequency        = 2
    integrated_output_time_interval    = 15.5
    nodal_output_frequency             = 100
    nodal_output_time_interval         = 0
}
```

defines an element boundary having two surfaces of the element set "duct". In this example, two types of surface quantities are output. The integrals of the surface quantities are output at the end of every other time step, every 15.5 units of run time and at the last time step. The surface quantities are also output for each node of the surface every 100 time steps and at the last time step.

There are two main forms of this command. The legacy version (or single topology version) of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle

Element Shape	Surface Shape
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number (of the parent element set), a unique (within this set) surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as `lower_wall.srf`:

```
1 101 1 3 11 13
2 102 3 5 13 15
```

and read by:

```
SURFACE_OUTPUT( "lower wall" ) {
   shape         = four_node_quad
   element_set   = "duct"
   surfaces      = Read( "lower_wall.srf" )
   ...
}
```

The mixed topology form of the `SURFACE_OUTPUT` command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the `SURFACE_SET` command, and are then referred to by the `SURFACE_OUTPUT` command. For example, a collection of triangular and quadrilateral element faces can be defined using the following `SURFACE_SET` commands.

```
SURFACE_SET( "tri faces" ) {
    surfaces        = { 1, 1, 1, 2, 4 ;
                        2, 2, 3, 4, 6 ;
                        3, 3, 5, 6, 8 ; }
    shape           = three_node_triangle
    volume_set      = "tetrahedra"
```

```
}
SURFACE_SET( "quad faces" ) {
    surfaces         = { 1, 1, 1, 2, 4, 9 ;
                         2, 2, 3, 4, 6, 12 ;
                         3, 3, 5, 6, 8, 15 ; }
    shape            = four_node_quad
    volume_set       = "prisms"
```

Then, a single `SURFACE_OUTPUT` command is defined that contains the tri and quad faces as follows:

```
SURFACE_OUTPUT ( "lower wall" ) {
   surface_sets       = {"tri_faces", "quad_faces"}
   ...
}
```

The list of surface sets can also be placed in a file, such as `surface_sets.srfst`:

```
tri faces
quad faces
```

and read using:

```
SURFACE_OUTPUT ( "lower wall" ) {
   surface_sets       = Read("surface_sets.srfst")
   ...
}
```

The mixed topology version of the `SURFACE_OUTPUT` command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *surface_sets* and *surfaces* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

Once the surface quantities have been written to disk, they can be translated to other formats using the AcuTrans program and other post-processing modules; see the *AcuSolve Programs Reference Manual* for details.

If either *integrated_output_frequency* or *integrated_output_time_interval* is non-zero, the integrals of the surface quantities will be output at the end of the run. If both are zero, no integrated quantity is written to disk.

Similarly, if either *nodal_output_frequency* or *nodal_output_time_interval* is non-zero, the nodal quantities of an output surface will be output at the end of the run. If both are zero, no nodal quantity is written to disk.

Run times may not coincide with *integrated_output_time_interval* or *nodal_output_time_interval*. In these cases, the corresponding data are output for every time step which passes through a multiple of *output_time_interval* or *nodal_output_time_interval*.

The surface does not have to lie on an outer boundary of the problem; it is allowed to be an "interior" surface. However, you are responsible for ensuring that all the interior elements specified in *surface* are on the same side of the interior surface. Otherwise, inappropriate cancellation will occur. Surface output quantities calculated for interior surfaces will be conservative, just as they are for exterior boundaries.

The *num_saved_states* parameter indicates the number of surface output steps to save. Once the (*num_saved_states* + 1)th step is written, the first one is removed. When running in parallel, once all files are written (in parallel), then the old ones are removed (in parallel). This parameter only applies to *surface_nodal_output*. It does not apply to surface integrated quantities.

SURFACE_SET

Specifies topological information for a set of surface elements.

Type

AcuSolve Command

Syntax

SURFACE_SET ("name") { parameters...}

Qualifier

User-given name.

Parameters

type (enumerated) [default]
Type of surface set.

default
Mesh topology are given in *shape*, *volume_set* and *surfaces* parameters.

dynamic
Mesh topology will be set in AcuPrep. Used with THERMAL_Shell only.

shape (enumerated) [no default]
Shape of the elements. For *type* = default only.

three_node_triangle or tri3 Three-node triangle.

four_node_quad or quad4 Four-node quadrilateral.

six_node_triangle or tri6 Six-node triangle.

surfaces (array) [no default]
A set of surface element connectivity. For *type* = default only.

volume_set or *vol_set (string) [no default]*
User-given name of the parent volume set. For *type* = default only.

dynamic_type (enumerated) [no default]
Type of dynamic *surface_set*.

shell_inner
If this option is selected, this will be used for the inner *surface_set* of the AcuPrep created THERMAL_SHELL.

shell_outer
If this option is selected, this will be used for the outer *surface_set* of the AcuPrep created THERMAL_SHELL.

reference_surface_set (string) [no default]
: Name of the *surface_set* used in THERMAL_SHELL. *Surface_set* adjacent to this one will be called *shell_inner* while *surface_set* on the other side of the THERMAL_SHELL will be *shell_outer*. Valid for *type* = dynamic only.

Description

The purpose of the SURFACE_SET command is to simply act as a container that is later referenced by other commands that take surface elements as an input. All elements in a given SURFACE_SET have the same shape and parent VOLUME_SET, but the SURFACE_SET container enables subsequent commands to accept mixed topologies. For example, consider a simulation which contains tetrahedra and prism elements. These surface elements are read into separate SURFACE_SET containers, which reference their parent volume sets:

```
VOLUME_SET( "tetrahedra" ) {
    elements       = { 1, 1, 2, 4, 3 ;
                       2, 3, 4, 6, 5 ;
                       3, 5, 6, 8, 7 ; }
    shape          = four_node_tet
}
VOLUME_SET( "prisms" ) {
    elements       = { 1, 1, 2, 4, 9, 10, 11 ;
                       2, 3, 4, 6, 12, 13, 14 ;
                       3, 5, 6, 8, 15, 16, 17 ; }
    shape          = six_node_wedge
}
SURFACE_SET( "tetrahedra surfaces" ) {
    surfaces       = { 1, 1, 1, 2, 4 ;
                       2, 2, 3, 4, 6 ;
                       3, 3, 5, 6, 8 ; }
    shape          = three_node_triangle
    volume_set     = "tetrahedra"
}
SURFACE_SET( "prism surfaces" ) {
    surfaces       = { 1, 1, 1, 2, 4, 9 ;
                       2, 2, 3, 4, 6, 12 ;
                       3, 3, 5, 6, 8, 15 ; }
    shape          = four_node_quad
    volume_set     = "prisms"
}
```

The surface sets can then be referenced in commands that accept surface elements as arguments. For example, the following SIMPLE_BOUNDARY_CONDITION command can be used to define a profiled inlet velocity field that spans tetrahedra and prism elements.

```
SIMPLE_BOUNDARY_CONDITION("profiled inlet velocity"){
   surface_sets     = {"tetrahedra surfaces","prism surfaces"}
   type             = inflow
   inflow_type      = average_velocity
   average_velocity = 1.0
}
```

The surfaces may also be placed in a file, such as tet_surfaces.srf

```
1      1         1           2              4
2      2         3           4              6
3      3         5           6              8
```

and read into the SURFACE_SET command:

```
SURFACE_SET( "tetrahedra surfaces" ) {
   surfaces    = Read("tet_surfaces.srf")
   shape       = three_node_triangle
   volume_set  = "tetrahedra"
}
```

The *volume_set*, *shape*, and *surfaces* parameters of the SURFACE_SET command are mandatory. *shape* specifies the shape (topology) of the surface elements in the set. *volume_set* provides the name of the volume set containing the parent volume element for each surface element. *surfaces* provides the surface element connectivity. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number (of the parent volume element), a unique, within this set, surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six columns, corresponding to the parent volume element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the parent volume element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent volume element definition.

TURBULENCE_WALL

Specifies a turbulent boundary layer wall.

Type

AcuSolve Command

Syntax

```
TURBULENCE_WALL("name") {parameters...}
```

Qualifier

User-given name.

Parameters

shape (enumerated) [no default]

Shape of the surfaces in this set.

three_node_triangle or tri3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral.
six_node_triangle or tri6	Six-node triangle.

element_set or *elem_set (string) [no default]*

User-given name of the parent element set.

surfaces (array) [no default]

List of element surfaces.

surface_sets (list) [={}]

List of surface set names (strings) to use in this command. When using this option, the connectivity, shape, and parent element of the surfaces are provided by the surface set container and it is unnecessary to specify the *shape*, *element_set* and *surfaces* parameters directly to the TURBULENCE_WALL command. This option is used in place of directly specifying these parameters. In the event that both of the *surface_sets* and *surfaces* parameters are provided, the full collection of surface elements is read and a warning message is issued. The *surface_sets* option is the preferred method to specify the surface elements. This option provides support for mixed element topologies and simplifies pre-processing and post-processing.

type (enumerated) [=low_reynolds_number]

Type of the turbulence wall modeling.

low_reynolds_number or low_re	Low Reynolds number damping functions.
wall_function or func	Turbulence wall function.
running_average_wall_function	Turbulence wall function based on running average field.

roughness_height (real) >=0 [=0]
: Average wall roughness height. Used with low_reynolds_number, wall_function, and running_average_wall_function type.

wall_function_friction_factor >=1 [=1]
: Constant factor used to scale the turbulence friction within the first element off the wall. Used with *turbulence_wall_type* = wall_function and running_average_wall_function.

wall_function_heat_flux_factor >=1 [=1]
: Constant factor used to scale the turbulent thermal conductivity within the first element off the wall. Used with *type* = wall_function and running_average_wall_function.

active_type (enumerated) [=all]
: Type of the active flag. Determines which surfaces in this set will be active as turbulence boundary layer walls.

all	All surfaces in this set are active.
none	No surface in this set is active.
no_interface	Only surfaces that are not in an interface surface set or do not find a contact surface of an appropriate medium are active.

Description

This command specifies the surfaces (element faces) bounding a turbulent boundary layer.

The turbulence wall surfaces are defined with respect to the elements of an element set. For example,

```
ELEMENT_SET( "interior" ) {
    shape             = four_node_tet
    elements          = { 1, 8, 3, 4, 9 ;
                          3, 3, 4, 9, 5 ;
                          ... }
    ...
}
TURBULENCE_WALL( "no-slip wall" ) {
    shape             = three_node_triangle
    element_set       = "interior"
    surfaces          = { 1, 12, 9, 3, 4 ;
                          3, 52, 5, 3, 4 ; }
    type              = wall_function
    roughness_height  = 0.001
    active_type       = all
}
```

specifies that two surfaces of the element set "interior" are on a no-slip boundary, that a turbulence wall function with an average wall roughness height of 0.001 is to be used to capture the turbulent boundary layer, and that both surfaces are active.

There are two main forms of this command. The legacy version (or single topology version) of the command relies on the use of the *surfaces* parameter to define the surfaces. When using this form of the command, all surfaces within a given set must have the same shape, and it is necessary to include both the *element_set* and *shape* parameters in the command. *shape* specifies the shape of the surface. This shape must be compatible with the shape of the "parent" element set whose user-given

name is provided by *element_set*. The element set shape is specified by the shape parameter of the ELEMENT_SET command. The compatible shapes are:

Element Shape	Surface Shape
four_node_tet	three_node_triangle
five_node_pyramid	three_node_triangle
five_node_pyramid	four_node_quad
six_node_wedge	three_node_triangle
six_node_wedge	four_node_quad
eight_node_brick	four_node_quad
ten_node_tet	six_node_triangle

The *surfaces* parameter contains the faces of the element set. This parameter is a multi-column array. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has five columns, corresponding to the element number (of the parent element set), a unique (within this set) surface number, and the three nodes of the element face. For four_node_quad, *surfaces* has six columns, corresponding to the element number, a surface number, and the four nodes of the element face. For six_node_triangle, *surfaces* has eight columns, corresponding to the element number, a surface number, and the six nodes of the element face. One row per surface must be given. The three, four, or six nodes of the surface may be in any arbitrary order, since they are reordered internally based on the parent element definition.

The surfaces may be read from a file. For the above example, the surfaces may be placed in a file, such as wall.srf:

```
1 12 9 3 4
3 52 5 3 4
```

and read by:

```
TURBULENCE_WALL( "no-slip wall" ) {
 shape       = three_node_triangle
 element_set = "interior"
 surfaces    = Read( "free.srf" )
 ...
}
```

The mixed topology form of the TURBULENCE_WALL command provides a more powerful and flexible mechanism for defining the surfaces. Using this form of the command, it is possible to define a collection of surfaces that contains different element shapes. This is accomplished through the use of the *surface_sets* parameter. The element faces are first created in the input file using the SURFACE_SET command, and are then referred to by the TURBULENCE_WALL command. For example, a collection of triangular and quadrilateral element faces can be defined using the following SURFACE_SET commands.

```
SURFACE_SET( "tri faces" ) {
   surfaces     = { 1, 1, 1, 2, 4 ;
```

```
                    2, 2, 3, 4, 6 ;
                    3, 3, 5, 6, 8 ; }
   shape         = three_node_triangle
   volume_set    = "tetrahedra"
}
SURFACE_SET( "quad faces" ) {
    surfaces     = { 1, 1, 1, 2, 4, 9 ;
                     2, 2, 3, 4, 6, 12 ;
                     3, 3, 5, 6, 8, 15 ; }
    shape        = four_node_quad
    volume_set   = "prisms"
```

Then, a single `TURBULENCE_WALL` command is defined that contains the tri and quad faces as follows:

```
TURBULENCE_WALL ( "no-slip wall" ) {
   surface_sets  = {"tri_faces", "quad_faces"}
   ...
}
```

The list of surface sets can also be placed in a file, such as `surface_sets.srfst`:

```
tri faces
quad faces
```

and read using:

```
TURBULENCE_WALL ( "no-slip wall" ) {
   surface_sets       = Read("surface_sets.srfst")
   ...
}
```

The mixed topology version of the `TURBULENCE_WALL` command is preferred. This version provides support for multiple element topologies within a single instance of the command and simplifies pre-processing and post-processing. In the event that both the *surface_sets* and *surfaces* parameters are provided in the same instance of the command, the full collection of surface elements is read and a warning message is issued. Although the single and mixed topology formats of the commands can be combined, it is strongly recommended that they are not.

A low_reynolds_number type specifies that the low Reynolds number damping functions are to be used in modeling the boundary layer. This model integrates the turbulence flow all the way to the boundary. As a result, this option is more accurate, provided that sufficient mesh density is used. For the Spalart-Allmaras turbulence model, y^+ (defined below) of the first mesh point must be less than 10 (preferably 5); otherwise, gross errors in traction, heat flux, and mass transfer may result.

A wall_function type specifies that the turbulence "law of the wall" is to be assumed near the wall. Turbulence wall functions allow for coarser meshes. For the Spalart-Allmaras turbulence model, for example, y^+ of the first mesh point may be as large as 300. On the other hand, the law of the wall applies only to simple turbulent boundary layers. The implemented wall function is valid only for non-separated flows with mild pressure gradients.

Unlike some wall function implementations there is no minimum y^+ requirement in AcuSolve. For small values of y^+ the two types become equivalent. The velocity law of the wall is given by:

$$U^+ = y^+, \qquad y^+ \le 5$$
$$U^+ = \frac{1}{k_a}\log(y^+) + B_{\log}, \quad y^+ \ge 30 \tag{193}$$

with a transition region for 5 < y^+ < 30 . These two parts are referred to as the "viscous sublayer" and the "logarithmic layer". The non dimensional distance to the wall y^+ and velocity U^+ are defined by:

$$y^+ = \frac{yu^*}{\nu}$$
$$U^+ = \frac{U}{u^*} \tag{194}$$

where γ is the kinematic viscosity, $u^* = \sqrt{\tau_w/\rho}$ is the turbulent friction velocity, τ_w is the wall shear, ρ is the density, U is the velocity parallel to the wall, and y is the normal distance to the wall. The logarithmic layer requires two experimentally determined parameters: $k_a = 0.41$ and $B_{\log} = 5.5$.

The temperature law of the wall is similar:

$$T^+ = Pr\, y^+, \qquad y^+ \le 5$$
$$T^+ = \frac{Pr_t}{k_a}\log(y^+) + A(Pr), \quad y^+ \ge 30 \tag{195}$$

plus a transition region. The non dimensional temperature is defined by

$$T^+ = \frac{T_w - T}{T^*} \tag{196}$$

where, in addition to the velocity parameters, T is the temperature, T_w is the wall temperature, $T^* = q_w/(\rho c_p u^*)$ is the friction temperature, q_w is the wall heat flux, and c_p is the heat capacity. The molecular Prandtl number is $Pr = c_p\mu/k$, where μ is the dynamic viscosity and k is the thermal conductivity. The turbulent Prandtl number is modeled as a constant: the default is $Pr_t = 0.91$, but can be changed in `CONDUCTIVITY_MODEL`. The curve fit for the intercept A is:

$$A(Pr) = (3.85Pr^{1/3} - 1.3)^2 + 2.12\log(Pr) \tag{197}$$

The species law of the wall has the same form as for temperature:

$$\phi^+ = Sc\, y^+, \qquad y^+ \le 5$$
$$\phi^+ = \frac{Sc_t}{k_a}\log(y^+) + A(Sc), \quad y^+ \ge 30 \tag{198}$$

plus a transition region. The nondimensional species is defined by

$$\phi^+ = \frac{\phi_w - \phi}{\phi^*} \tag{199}$$

where, in addition to the parameters defined above, Φ is the species, ϕ_w is the wall species, $\phi^* = \theta_w/(\rho u^*)$ is the friction species, and θ_w is the wall species flux. The molecular Schmidt number is $Sc = \nu/D$, where D is the species diffusivity. The turbulent Schmidt number is modeled as a constant: the default is Sct = 0.91, but can be changed in `DIFFUSIVITY_MODEL`. The function A is the same as for the temperature law of the wall.

For simple attached turbulent boundary layers the required distance from the wall of the first mesh point can be estimated given a desired y^+ value. The Reynolds number is

$$\mathrm{Re}_x = \frac{U_\infty x}{\nu} \tag{200}$$

where x is the streamwise distance from the beginning of the boundary layer and U_∞ is the freestream velocity. An approximate correlation for the wall shear is given by:

$$\frac{\tau_w}{\rho U_\infty^{\,2}} = 0.0296\,\mathrm{Re}_x^{-0.2} \tag{201}$$

From this the distance to the first node may be derived:

$$y = 5.81 y^+ \left(\frac{\nu}{U_\infty}\right)\mathrm{Re}_x^{\,0.1} \tag{202}$$

The extremely weak dependence on Reynolds number allows this procedure to work well with only a rough estimate of the Reynolds number. For pipe flow a similar relationship valid for $\mathrm{Re}_d < 10^5$ may be derived:

$$y = 5.03 y^+ \left(\frac{\nu}{U_a}\right)\mathrm{Re}_d^{\,0.125} \tag{203}$$

where U_a is the average pipe velocity and the Reynolds number Re_d is based on U_a and the pipe diameter.

Normally, designing a mesh based on the turbulent velocity profile is sufficient to resolve the temperature and species boundary layers as well.

Specification of a turbulence wall via this command does not enforce a no-slip condition on the given surface. The no-slip velocity and proper conditions for the turbulence variables must be separately imposed using `NODAL_BOUNDARY_CONDITION` commands. Wall roughness may be modeled by specifying an average wall roughness height for equivalent sand grains, given by *`roughness_height`*.

In some circumstances it is necessary to "turn off" or "deactivate" a previously-defined turbulence boundary layer wall. This is accomplished by setting the active flag through active_type. A value of all is the default and means that all surfaces in the set are active (active flag = 1) as turbulence boundary layer walls. A value of none means that no surface is active (active flag = 0).

A no_interface active type is used to automatically turn on and off each surface's status as a turbulence boundary layer wall in the presence of interfaces. Each surface is such a wall (active flag = 1) if the surface does not belong to an `INTERFACE_SURFACE` set or if it does not find a contact surface of fluid medium. That is, each surface will fall into one of four possible cases:

- The surface is not in any `INTERFACE_SURFACE` set; active flag = 1.
- The surface is in an `INTERFACE_SURFACE` set, but it does not find a contact surface; active flag = 1.
- The surface finds a contact surface of fluid medium; active flag = 0.
- The surface finds a contact surface of solid/shell medium; active flag = 1.

VAO_ELEMENT_OUTPUT

Specifies a set of elements for vibro-acoustics integrated-element output.

Type

AcuSolve Command

Syntax

VAO_ELEMENT_OUTPUT("name") {parameters...}

Qualifier

User-given name.

Parameters

coordinates or *coord* *(array) [no default]*
: An array of sample points used to construct the elements defined by the *elements_array* parameter. Each row contains a unique identification number and the x, y and z coordinates of a point in the global xyz coordinate system.

gap *(real) >=0 [=0]*
: Tolerance for sample points falling outside of all elements.

shape *(enumerated) [no default]*
: Topology of the elements in this set.

four_node_tet or tet4	Four-node tetrahedron.
five_node_pyramid or pyramid5	Five-node pyramid.
six_node_wedge or wedge6	Six-node wedge (prism).
eight_node_brick or hex8	Eight-node brick (hexahedron).

elements *(array) [no default]*
: List of element connectivity.

quadrature *(enumerated) [=full]*
: Quadrature rule used for volume integration.

full	Full quadrature.
reduced	Reduced quadrature.

Description

This command specifies a set of elements for the element-integrated vibro-acoustics output data. Element geometry and integration rules are given here, while the output quantities and related parameters are specified in the VAO_OUTPUT command. For example,

```
VAO_ELEMENT_OUTPUT( "sample elements" ) {
 coordinates = Read( "sample.crd" )
 elements    = Read( "sample.cnn" )
 shape       = eight_node_brick
 quadrature  = full
}
```

defines the vibro-acoustic output on a set of elements specified by sample.cnn.

A set of sample points is specified by the *coordinates* parameter, which is a four-column array corresponding to an identification number and the x, y and z coordinates of each sample point, one row per point. The identification number may be any arbitrary but unique (within the set) integer. Negative numbers are acceptable. The coordinates are in the global xyz coordinate system. This is the same format as the *coordinates* parameter of the COORDINATE command, but the sample points do not have to be a subset of the nodal coordinates nor related to them in any other way. The *coordinates* parameter is mandatory. Within AcuPrep, an element is found for each sample point and interpolation shape functions are computed. If a sample point falls outside all fluid elements, then the closest element face is found, up to a distance specified by *gap*. An error is issued if no element is found.

The *elements* parameter defines the element connectivity and has the same format as the corresponding parameter in ELEMENT_SET command. However, the elements defined here need not have any relationship to any other elements or element sets in the CFD model. Unlike ELEMENT_SET this command does not support higher-order elements (that is, ten-node tetrahedra) or nodal quadrature.

VAO_OUTPUT

Specifies output parameters for vibro-acoustics output.

Type

AcuSolve Command

Syntax

VAO_OUTPUT{parameters...}

Qualifier

None.

Parameters

reference_density or *ref_dens* *(real) [=1.225]*
: Reference density. Required to compute the output quantities.

reference_pressure or *ref_pres* *(real) [=0]*
: Reference pressure. Required to compute the output quantities.

reference_sound_speed or *sound* *(real) [=330]*
: Reference sound speed. Required to compute the output quantities.

source_output_frequency or *out_freq* *(integer) >=0 [=0]*
: Time step frequency at which to output VAO data. If zero, this option is ignored.

source_output_time_interval or *out_intv* *(real) >=0 [=0]*
: Time frequency at which to output the VAO data. If zero, this option is ignored.

num_saved_states or *save* *(integer) >=0 [=0]*
: Number of VAO data files to keep. If zero, all are kept.

Description

This command specifies the vibro-acoustics output data properties. For example,

```
VAO_OUTPUT{
 reference_density           = 1.225
 reference_pressure          = 0
 reference_sound_speed       = 330
 source_output_frequency     = 1
 source_output_time_interval = 0
 num_saved_states            = 0
}
```

The parameters *reference_density*, *reference_pressure* and *reference_sound_speed* provide the required data for the computation of the output fields. An AcuSolve input file should only contain one VAO_OUTPUT command.

VAO_SURFACE_OUTPUT

Specifies a set of surfaces for integrated-surface output parameters for vibro-acoustics.

Type

AcuSolve Command

Syntax

VAO_SURFACE_OUTPUT("name") {parameters...}

Qualifier

User-given name.

Parameters

coordinates or *coord* *(array) [no default]*

An array of sample points used to construct the surfaces defined by the *partial_surfaces* parameter. Each row contains a unique identification number and the x, y and z coordinates of a point in the global xyz coordinate system.

gap *(real) >=0 [=0]*

Tolerance for sample points falling outside of all elements.

shape *(enumerated) [no default]*

Shape of the surfaces in this set.

three_node_triangle or tria3	Three-node triangle.
four_node_quad or quad4	Four-node quadrilateral.

partial_surfaces *(array) [no default]*

List of surfaces with no references to parent elements or element sets.

quadrature *(enumerated) [=full]*

Quadrature rule used for volume integration.

full	Full quadrature.
reduced	Reduced quadrature.

Description

This command specifies a set of surfaces for the surface-integrated vibro-acoustics output data. Surface geometry and integration rules are given here, while the related parameters are specified in the VAO_OUTPUT command. For example,

```
VAO_SURFACE_OUTPUT( "sample surfaces" ) {
   coordinates      = Read( "sample.crd" )
   partial_surfaces = Read( "sample.srf" )
   shape            = four_node_quad
   quadrature       = full
```

```
}
```

defines a vibro-acoustics surface output.

A set of sample points is specified by the *coordinates* parameter, which is a four-column array corresponding to an identification number and the x, y and z coordinates of each sample point, one row per point. The identification number may be any arbitrary but unique (within the set) integer. Negative numbers are acceptable. The coordinates are in the global xyz coordinate system. This is the same format as the *coordinates* parameter of the COORDINATE command, but the sample points do not have to be a subset of the nodal coordinates nor related to them in any other way. The *coordinates* parameter is mandatory. Within AcuPrep, an element is found for each sample point and interpolation shape functions are computed. If a sample point falls outside all fluid elements, then the closest element face is found, up to a distance specified by *gap*. An error is issued if no element is found.

The *partial_surfaces* parameter defines the connectivity of the surfaces. This parameter is a multi-column array, but it does not have the same format as the corresponding parameter in other surface commands, for example, SURFACE_OUTPUT. The number of columns depends on the shape of the surface. For three_node_triangle, this parameter has four columns, corresponding to a unique (within this set) surface number and the three nodes of the surface. four_node_quad, *partial_surfaces* has five columns, corresponding to a surface number and the four nodes of the surface. One row per surface must be given. Since there is no parent element definition, the three or four nodes of the surface must be ordered around the periphery of the surface. Also unlike other surface commands this command does not support higher-order surfaces, that is, six-node triangles, or nodal quadrature. The nodes contained in the surface connectivity must all be defined by the *coordinates* parameter of this command.

VOLUME_SET

Specifies topological information for a set of volumetric elements.

Type

AcuSolve Command

Syntax

VOLUME_SET ("name") { parameters...}

Qualifier

User-given name.

Parameters

Shape (enumerated) [no default]

Shape of the elements.

four_node_tet **or** ***tet4***	Four-node tetrahedron.
five_node_pyramid **or** ***pyramid5***	Five-node pyramid.
six_node_wedge **or** ***wedge6***	Six-node wedge (prism).
eight_node_brick **or** ***hex8***	Eight-node brick (hexahedron).
ten_node_tet **or** ***tet10***	Ten-node tetrahedron.

elements (array) [no default]

A set of element connectivity.

Description

The purpose of the VOLUME_SET command is to simply act as a container that is later referenced by other commands that take volume elements as an input. All elements in a given VOLUME_SET have the same shape. The VOLUME_SET container enables subsequent commands to accept mixed topologies. For example, consider a simulation which contains tetrahedra and prism elements. These volume elements are read into separate VOLUME_SET containers, and are then referenced in a single ELEMENT_SET command as follows:

```
VOLUME_SET( "tetrahedra" ) {
    elements     = { 1, 1, 2, 4, 3 ;
                     2, 3, 4, 6, 5 ;
                     3, 5, 6, 8, 7 ; }
    shape        = four_node_tet
}
VOLUME_SET( "prisms" ) {
    elements     = { 1, 1, 2, 4, 9, 10, 11 ;
                     2, 3, 4, 6, 12, 13, 14 ;
                     3, 5, 6, 8, 15, 16, 17 ; }
    shape        = six_node_wedge
```

```
}
ELEMENT_SET( "fluid elements" ) {
   volume_sets    = { "tetrahedra", "prisms" }
   medium         = fluid
   ...
}
```

The *elements* and *shape* parameters of the VOLUME_SET command are mandatory. *shape* specifies the shape (topology) of the elements in the set. *elements* provides the element connectivity. This parameter is a multi column array, with one row per element. The first column corresponds to the element numbers, which are unique numbers within each volume set. Negative values are acceptable. The remaining columns are the element nodes. The number of element nodes and their order must match the shape of the volume set. The element node numbers must be valid numbers, as given by the COORDINATE command.

A four-node tetrahedron element requires four nodes ordered with the local node numbering order shown below:

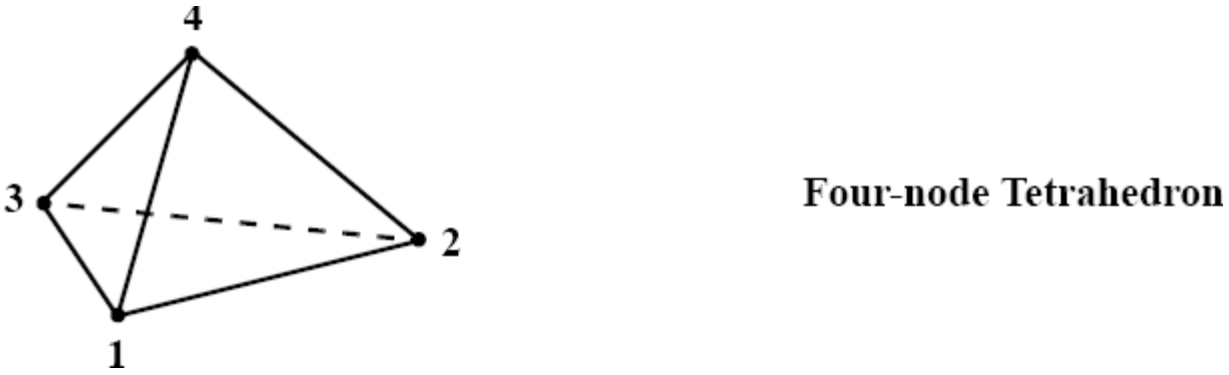


Figure 27: Four-node Tetrahedron Element

A five-node pyramid element requires five nodes ordered with the local node numbering order shown below:

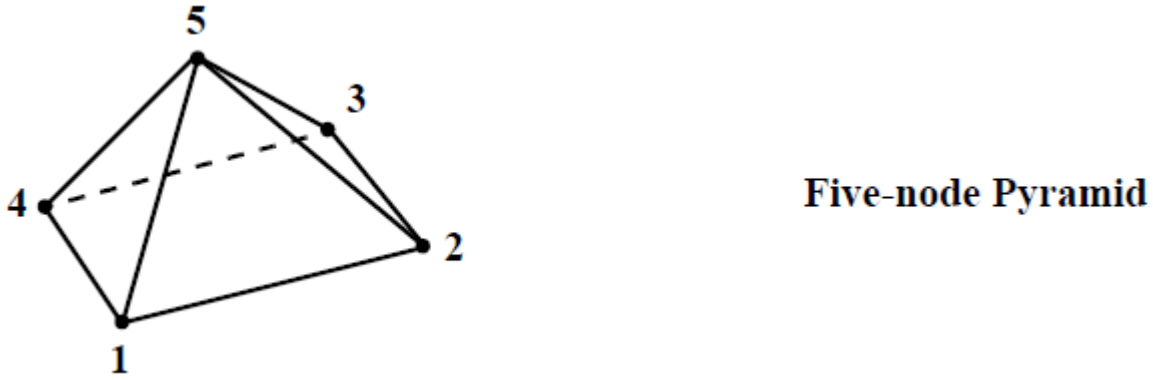
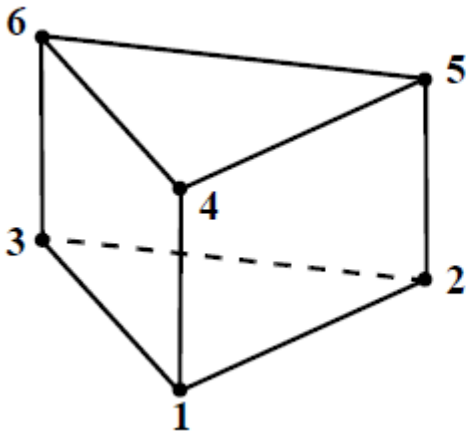


Figure 28: Five-node Pyramid Element

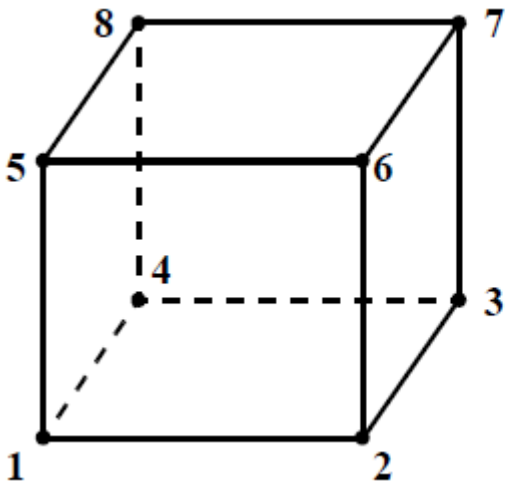
A six-node wedge (or prism) element requires six nodes ordered with the local node numbering order shown below:



Six-node Wedge

Figure 29: Six-node Wedge Element

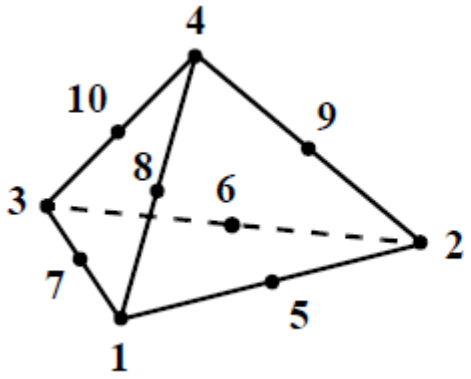
An eight-node brick (or hexahedron) element requires eight nodes ordered with the local node numbering order shown below:



Eight-node Brick

Figure 30: Eight-node Brick Element

A ten-node tetrahedron element requires ten nodes ordered with the local node numbering order shown below:



Ten-node Tetrahedron

Figure 31: Ten-node Tetrahedron Element

The element nodes may also be read from a file. For the above example, the element nodes may be placed in "tetrahedra.cnn":

```
1       1          2          4           3
2       3          4          6           5
3       5          6          8           7
```

and "prisms.cnn":

```
1       1          2          4           9          10
   11
2       3          4          6          12          13
   14
3       5          6          8          15          16
   17
```

then read from the file in the volume set commands:

```
VOLUME_SET( "tetrahedra" ) {
   elements  = Read("tetrahedra.cnn")
shape        = four_node_tet
}
VOLUME_SET( "prisms" ) {
   elements  = Read("prisms.cnn")
shape        = six_node_wedge
}
```

Each `VOLUME_SET` command must be referenced in an `ELEMENT_SET` command in order to have the elements participate in the solution. For example, it is not possible to define a `VOLUME_SET` strictly for `ELEMENT_OUTPUT` purposes without using it in an `ELEMENT_SET` command.

Functional Commands

14

The Functional commands, except for AUTO_SOLUTION_STRATEGY, are outlined in this chapter.

This chapter covers the following:

- RUN (p. 699)
- RESTART (p. 701)
- INCLUDE (p. 705)
- ASSIGN (p. 706)
- QUIT (p. 707)

Functional commands are differentiated from declarative commands in that they are executed immediately; see Command Format. The purpose of the last three commands is to facilitate writing input files.

RUN

Processes the input data for the solver.

Type

AcuSolve Command

Syntax

`RUN {parameters}`

Qualifier

This command has no qualifier.

Parameters

`ignore_input_error` *(boolean) [=off]*

Flag specifying whether to ignore any errors that occur while reading the preceding commands. If off and an error has occurred, this command is not processed.

Description

This functional command processes all the solver data that has been read through the preceding declarative commands. This command must be placed after all other commands. No command may follow the `RUN` command.

For example,

```
ANALYSIS {
  title = "test problem"
}
...
RUN {
}
```

When the `RUN` command is encountered in the input file, the program systematically processes each of the declarative commands that have preceded it. The program performs a number of input error checks, including checks for node and element numbering consistencies, surface definitions, solution strategy consistencies, material model and body force cross-referencing, and so on. It checks the boundary conditions for completeness and consistency. If there are any potential boundary condition problems, it generates the file `problem.bc_warnings`, where `problem` is the name of the problem. The `RUN` command also performs a number of pre-processing operations. Finally, it decomposes the domain for parallel processing and outputs the data in a number of internal files, ready to be read by the core solver.

If an error occurs during any of these operations, it is reported. The program will then attempt to continue to process the remaining data, with the hope of capturing all other potential input errors. Once completed, however, the program does not generate any solver file, and it terminates with an error.

If an error has occurred in the input file prior to the `RUN` command, the `RUN` command is not processed, by default. This behavior may be overwritten by setting the *`ignore_input_error`* parameter to on:

```
RUN {
   ignore_input_error = on
}
```

This option may be used to find other input errors that are only detected through the `RUN` command.

RESTART

Restarts from a previously solved problem.

Type

AcuSolve Command

Syntax

`RESTART {parameters}`

Qualifier

This command has no qualifier.

Parameters

`from_problem` or *`problem`* *(string) [no default]*
Name of the problem from which to restart. If not specified, the name of the current problem is used.

`from_directory` or *`dir`* *(string) [no default]*
Name of the working directory from which to access the restart files. If not specified, the name of the current working directory is used. An absolute or relative path to the working directory of the previous run is allowed. The path should be surrounded by quotes in the input deck.

`from_run` or *`run`* *(integer) >=0 [=0]*
Run ID from which to restart the problem. If 0, the program restarts from the latest available run.

`from_time_step` or *`step`* *(integer) >=0 [=0]*
Time step from which to restart the problem. If 0, the program restarts from the latest available time step.

`reset_time_step` *(boolean) [=off]*
Flag specifying whether to reset the time step and time to zero.

`reset_time_increment` *(boolean) [=off]*
Flag specifying whether to reset the time increment.

`type` *(enumerated) [=incremental]*
Type of restart to perform.

incremental	Performs an incremental restart where all settings from the previous run are retained with the exception of those that are present after the `RESTART{}` command in the input file that is being processed.
full	Performs a full restart where no commands from the previous run are retained. The only data that is carried forward across the restart is solution data associated with nodal results, rigid bodies, flexible bodies, and so on. When using this type of

restart, the input file must contain a fully populated set of commands followed by the RESTART command.

Description

This functional command restarts the problem from a previously solved problem. For example,

```
RESTART { }
```

loads in the restart data of the last time step of the last run of the current problem in the current working directory.

The *from_problem* parameter specifies the name of the problem from which to restart. If it is unspecified, the name of the current problem is used. The current problem name is provided by the command line option, problem; see the *AcuSolve Programs Reference Manual* for a description of command line options.

The *from_directory* parameter specifies the working directory of the problem from which to restart. This directory may be either relative to the current directory or an absolute address. If the *from_directory* parameter is unspecified, the working directory of the current problem is used. This directory is given by the command line option working_directory; see the *AcuSolve Programs Reference Manual* for a description of command line options.

Every time the solver is run, a new run identification number is assigned to the problem. Within each run there might be one or more restart time steps. The *from_run* and *from_time_step* parameters are used to load the appropriate restart data. If both parameters are non-zero, then the proper file is loaded. If the restart data for such a run and time step does not exist, an error occurs. One or both of these parameters may be set to 0, indicating the use of the latest values. To be specific, if *from_run* is non-zero, but *from_time_step* is zero, then the latest restart time step of the given run is loaded. If *from_time_step* is non-zero, but *from_run* is zero, the latest run ID with the given time step is loaded. If both parameters are zero, then the latest run with the latest time step is loaded. In all cases, if no restart data are found, an error occurs.

The simplest input file with the restart option is

```
RESTART {
type = incremental
}
RUN { }
```

Here, the latest restart data of the current problem is loaded. Then the data are prepared for the solver by the RUN command.

Once the restart data are read, one or more problem parameters may be changed. In principal, all parameters may be changed at the restart, with the exception of the number of nodes given in the nodal coordinates.

For example, assume that the original problem included all boundary conditions, material models, and other needed data for solving the temperature field, but only the flow equations were solved. At restart, the solution of the flow equations may be used to solve for the temperature field, by modifying or adding TIME_SEQUENCE and STAGGER commands, as shown in the following input file:

```
RESTART {
```

```
   type      = incremental
}
TIME_SEQUENCE {
   staggers  = { "temp" }
}
STAGGER( "temp" ) {
   equation  = temperature
   ...
}
RUN {
}
```

The `RESTART` command sets the unknown nodal fields to those written by the `RESTART_OUTPUT` command.

These solutions may be overwritten by including a `NODAL_INITIAL_CONDITION` command after the `RESTART` command. For example,

```
RESTART {
   type         = incremental
}
NODAL_INITIAL_CONDITION( temperature ) {
   nodal_values = Read( "temp.new" )
}
RUN {
}
```

replaces the restart temperature values with those in the file `temp.new`.

Setting *`reset_time_step`* to on sets both the step number and time to zero. The default behavior is to use the step number and time from the previous solution.

A restart of *`type`* = full deviates from one of *`type`* = incremental in the sense that it does not retain any of the commands from the previous run. Instead, it requires you to fully specify the new run in the current input file, and simply extracts the solution information from the previous run. When using this command, it is typically placed at the end of the input file. For example, the following sequence of input commands can be used to run a transient simulation by restarting from a steady state case.

```
ANALYSIS {
    title                    = "test problem"
    type                     = transient
}
EQUATION {
    flow                     = navier_stokes
    turbulence               = spalart_allmaras
}
AUTO_SOLUTION_STRATEGY{
    initial_time_increment   =  .001
    min_stagger_iterations   = 2
    max_stagger_iterations   = 3
}
COORDINATE{
    coordinates = Read("MESH.DIR/testModel.crd")
}
SIMPLE_BOUNDARY_CONDITION("wall"){
    ...
}
...
RESTART{
```

```
    type                        = full
    reset_time_step             = on
    reset_time_increment        = on
}
RUN {}
```

Care must be used when exercising functional commands in combination with the RESTART command. The specific instances of INCLUDE, ASSIGN and AUTO_SOLUTION_STRATEGY commands are not recorded in the file that contains the primary run information. Instead, they are evaluated immediately, and the values they produce are stored along with the data describing the run. This has important implications when restarting from a run that contains commands such as AUTO_SOLUTION_STRATEGY. When you rely on the AUTO_SOLUTION_STRATEGY command to set the solution strategy, it should be explicitly re-issued whenever a RESTART is used and the parameters impacting it are changed. For example, if a new equation is added to the restart case, it is necessary to have a new instance of the AUTO_SOLUTION_STRATEGY command in the restart file to properly populate all of the solution strategy commands. For example, a species equation can be added upon restart using the following set of commands:

```
RESTART {
    type                = incremental
}
EQUATION{
    species_transport   = advective_diffusive
}
AUTO_SOLUTION_STRATEGY{}
NODAL_INITIAL_CONDITION( species_1 ) {
    default_value       = .5
}
SIMPLE_BOUNDARY_CONDITION("Inflow"){
    species_1_type      = value
    species_1           = .25
RUN {
}
```

You should also bear in mind that if you modified any of the commands resulting from the issuance of AUTO_SOLUTION_STRATEGY in the initial run, you must also do so in the restart file. Otherwise, the issuance of the AUTO_SOLUTION_STRATEGY command in the restart file will overwrite the values that were modified in the original run.

INCLUDE

Includes the contents of an external file.

Type

AcuSolve Command

Syntax

`INCLUDE {parameters}`

Qualifier

This command has no qualifier.

Parameters

`file` (string) [no default]
Name of the input file to be included.

Description

This functional command includes the content of an external file. For example,

```
ANALYSIS {
   title = "test problem"
}
...
INCLUDE {
  file = "material-model/air"
}
RUN {
}
```

At the time it is first read, the `INCLUDE` command is replaced by the contents of the `material-model/air` file.

The *`file`* parameter points to the file to be included. This file may be either a relative file address, as in the above example, or an absolute file address, which starts with a "/".

This command is typically used for storing commonly used parameters (commands), such as a library of material models. It is also used for storing subsets of an input file which may be changed frequently.

ASSIGN

Assigns a value to a variable.

Type

AcuSolve Command

Syntax

ASSIGN {parameters}

Qualifier

This command has no qualifier.

Parameters

`variable` or `var` (string) [no default]
Variable name.

`value` (real) [=0]
Value assigned to the variable.

Description

This functional command assigns a value to a variable to be used in expressions defining integer, real or array parameters. Details can be found at Parameter Operators and Functions.

For example,

```
ASSIGN {
 variable        = MPH
 value           = 1609.344 / 3600 # mile/hr --> m/sec
}
NODAL_BOUNDARY_CONDITION( "inflow x-velocity" ) {
 variable        = x_velocity
 type            = constant
 constant_value  = 60 * MPH
 ...
}
```

The variable names are case insensitive; therefore, the double quotes that are used for string parameters may be omitted, as in the above example.

QUIT

Terminates parsing the input file.

Type

AcuSolve Command

Syntax

`QUIT {parameters}`

Qualifier

This command has no qualifier.

Parameters

This command has no parameters.

Description

This functional command terminates reading (parsing) the input file.

For example,

```
ANALYSIS {
 title = "test problem"
}
...
QUIT # check the commands read up to this point
...
```

This command is typically used for debugging.

Note: The `RUN` command, which is typically placed at the end of a file, also terminates the parsing of the input file. `EXIT` may also be used as an alternative command name for `QUIT`.

Intellectual Property Rights Notice

Copyright © 1986-2024 Altair Engineering Inc. All Rights Reserved.

This Intellectual Property Rights Notice is exemplary, and therefore not exhaustive, of the intellectual property rights held by Altair Engineering Inc. or its affiliates. Software, other products, and materials of Altair Engineering Inc. or its affiliates are protected under laws of the United States and laws of other jurisdictions.

In addition to intellectual property rights indicated herein, such software, other products, and materials of Altair Engineering Inc. or its affiliates may be further protected by patents, additional copyrights, additional trademarks, trade secrets, and additional other intellectual property rights. For avoidance of doubt, copyright notice does not imply publication. Copyrights in the below are held by Altair Engineering Inc. or its affiliates. Additionally, all non-Altair marks are the property of their respective owners. If you have any questions regarding trademarks or registrations, please contact marketing and legal.

This Intellectual Property Rights Notice does not give you any right to any product, such as software, or underlying intellectual property rights of Altair Engineering Inc. or its affiliates. Usage, for example, of software of Altair Engineering Inc. or its affiliates is governed by and dependent on a valid license agreement.

Altair® HyperWorks®, a Design & Simulation Platform

Altair® AcuSolve® ©1997-2024

Altair® Activate® ©1989-2024

Altair® Automated Reporting Director™ ©2008-2022

Altair® Battery Damage Identifier™ ©2019-2024

Altair® CFD™ ©1990-2024

Altair Compose® ©2007-2024

Altair® ConnectMe™ ©2014-2024

Altair® DesignAI™ ©2022-2024

Altair® DSim™ ©2024

Altair® EDEM™ ©2005-2024

Altair® EEvision™ ©2018-2024

Altair® ElectroFlo™ ©1992-2024

Altair Embed® ©1989-2024

Altair Embed® SE ©1989-2024

Altair Embed®/Digital Power Designer ©2012-2024

Altair Embed®/eDrives ©2012-2024

Altair Embed® Viewer©1996-2024

Altair® e-Motor Director™©2019-2024

Altair® ESAComp® ©1992-2024

Altair® expertAI™ ©2020-2024

Altair® Feko® ©1999-2024

Altair® FlightStream® ©2017-2024

Altair® Flow Simulator™ ©2016-2024

Altair® Flux® ©1983-2024

Altair® FluxMotor® ©2017-2024

Altair® GateVision PRO™ ©2002-2024

Altair® Geomechanics Director™ ©2011-2022

Altair® HyperCrash® ©2001-2023

Altair® HyperGraph® ©1995-2024

Altair® HyperLife® ©1990-2024

Altair® HyperMesh® ©1990-2024

Altair® HyperMesh® CFD ©1990-2024

Altair® HyperMesh ® NVH ©1990-2024

Altair® HyperSpice™ ©2017-2024

Altair® HyperStudy® ©1999-2024

Altair® HyperView® ©1999-2024

Altair® HyperView Player® ©2022-2024

Altair® HyperWorks® ©1990-2024

Altair® HyperWorks® Design Explorer ©1990-2024

Altair® HyperXtrude® ©1999-2024

Altair® Impact Simulation Director™ ©2010-2022

Altair® Inspire™ ©2009-2024

Altair® Inspire™ Cast ©2011-2024

Altair® Inspire™ Extrude Metal ©1996-2024

Altair® Inspire™ Extrude Polymer ©1996-2024

Altair® Inspire™ Form ©1998-2024

Altair® Inspire™ Mold ©2009-2024

Altair® Inspire™ PolyFoam ©2009-2024

Altair® Inspire™ Print3D ©2021-2024

Altair® Inspire™ Render ©1993-2024

Altair® Inspire™ Studio ©1993-2024

Altair® Material Data Center™ ©2019-2024

Altair® Material Modeler™ ©2019-2024

Altair® Model Mesher Director™ ©2010-2024

Altair® MotionSolve® ©2002-2024

Altair® MotionView® ©1993-2024

Altair® Multi-Disciplinary Optimization Director™ ©2012-2024

Altair® Multiscale Designer® ©2011-2024

Altair® newFASANT™ ©2010-2020

Altair® nanoFluidX® ©2013-2024

Altair® NLVIEW® ©2018-2024

Altair® NVH Director™ ©2010-2024

Altair® NVH Full Vehicle™ ©2022-2024

Altair® NVH Standard™ ©2022-2024

Altair® OmniV™ ©2015-2024

Altair® OptiStruct® ©1996-2024

Altair® physicsAI™ ©2021-2024

Altair® PollEx™ ©2003-2024

Altair® PollEx™ for ECAD ©2003-2024

Altair® PSIM™ ©1994-2024

Altair® Pulse™ ©2020-2024

Altair® Radioss® ©1986-2024

Altair® romAI™ ©2022-2024

Altair® RTLvision PRO™ ©2002-2024

Altair® S-CALC™ ©1995-2024

Altair® S-CONCRETE™ ©1995-2024

Altair® S-FRAME® ©1995-2024

Altair® S-FOUNDATION™ ©1995-2024

Altair® S-LINE™ ©1995-2024

Altair® S-PAD™ ©1995-2024

Altair® S-STEEL™ ©1995-2024

Altair® S-TIMBER™ ©1995-2024

Altair® S-VIEW™ ©1995-2024

Altair® SEAM® ©1985-2024

Altair® shapeAI™ ©2021-2024

Altair® signalAI™ ©2020-2024

Altair® Silicon Debug Tools™ ©2018-2024

Altair® SimLab® ©2004-2024

Altair® SimLab® ST ©2019-2024

Altair® SimSolid® ©2015-2024

Altair® SpiceVision PRO™ ©2002-2024

Altair® Squeak and Rattle Director™ ©2012-2024

Altair® StarVision PRO™ ©2002-2024

Altair® Structural Office™ ©2022-2024

Altair® Sulis™ ©2018-2024

Altair® Twin Activate® ©1989-2024

Altair® UDE™ ©2015-2024

Altair® ultraFluidX® ©2010-2024

Altair® Virtual Gauge Director™ ©2012-2024

Altair® Virtual Wind Tunnel™ ©2012-2024

Altair® Weight Analytics™ ©2013-2022

Altair® Weld Certification Director™ ©2014-2024

Altair® WinProp™ ©2000-2024

Altair® WRAP™ ©1998-2024

Altair® HPCWorks®, a HPC & Cloud Platform

Altair® Allocator™ ©1995-2024

Altair® Access™ ©2008-2024

Altair® Accelerator™ ©1995-2024

Altair® Accelerator™ Plus ©1995-2024

Altair® Breeze™ ©2022-2024

Altair® Cassini™ ©2015-2024

Altair® Control™ ©2008-2024

Altair® Desktop Software Usage Analytics™ (DSUA) ©2022-2024

Altair® FlowTracer™ ©1995-2024

Altair® Grid Engine® ©2001, 2011-2024

Altair® InsightPro™ ©2023-2024

Altair® Hero™ ©1995-2024

Altair® Liquid Scheduling™ ©2023-2024

Altair® Mistral™ ©2022-2024

Altair® Monitor™ ©1995-2024

Altair® NavOps® ©2022-2024

Altair® PBS Professional® ©1994-2024

Altair® PBS Works™ ©2022-2024

Altair® Simulation Cloud Suite (SCS) ©2024

Altair® Software Asset Optimization (SAO) ©2007-2024

Altair® Unlimited™ ©2022-2024

Altair® Unlimited Data Analytics Appliance™ ©2022-2024

Altair® Unlimited Virtual Appliance™ ©2022-2024

Altair® RapidMiner®, a Data Analytics & AI Platform

Altair® AI Hub ©2023-2024

Altair® AI Edge™ ©2023-2024

Altair® AI Cloud ©2022-2024

Altair® AI Studio ©2023-2024

Altair® Analytics Workbench™ ©2002-2024

Altair® Graph Lakehouse™ ©2013-2024

Altair® Graph Studio™ ©2007-2024

Altair® Knowledge Hub™ ©2017-2024

Altair® Knowledge Studio® ©1994-2024

Altair® Knowledge Studio® for Apache Spark ©1994-2024

Altair® Knowledge Seeker™ ©1994-2024

Altair® IoT Studio™ ©2002-2024

Altair® Monarch® ©1996-2024

Altair® Monarch® Classic ©1996-2024

Altair® Monarch® Complete™ ©1996-2024

Altair® Monarch® Data Prep Studio ©2015-2024

Altair® Monarch Server™ ©1996-2024

Altair® Panopticon™ ©2004-2024

Altair® Panopticon™ BI ©2011-2024

Altair® SLC™ ©2002-2024

Altair® SLC Hub™ ©2002-2024

Altair® SmartWorks™ ©2002-2024

Altair® RapidMiner® ©2001-2024

Altair One® ©1994-2024

Altair® CoPilot™ ©2023-2024

Altair® License Utility™ ©2010-2024

Altair® TheaRender® ©2010-2024

OpenMatrix™ ©2007-2024

OpenPBS® ©1994-2024

OpenRadioss™ ©1986-2024

October 7, 2024

Technical Support

Altair's support resources include engaging learning materials, vibrant community forums, intuitive online help resources, how-to guides, and a user-friendly support portal.

Visit Customer Support to learn more about our support offerings.

Index

A

AcuSolve command reference manual introduction *6*
ALGEBRAIC_MULTIGRID_PARAMETERS *147*
alternative command organization *30*
ANALYSIS *36*
array *19*
ascii files *25*
ASSIGN *706*
AUTO_SOLUTION_STRATEGY *170*

B

BATTERY_COMPONENT_MODEL *456*
BATTERY_MODEL *270*
BATTERY_MODULE *457*
BATTERY_PACK *470*
BATTERY_THERMAL_RUNAWAY_MODEL *473*
binary files *27*
body force commands *225*
BODY_FORCE *226*
boolean *14*

C

CAA_ELEMENT_OUTPUT *477*
CAA_OUTPUT *479*
CAA_SURFACE_OUTPUT *488*
command format *7*
command qualifier *10*
CONDUCTIVITY_MODEL *280*
CONSTRAINT *39*
CONTACT_ANGLE_MODEL *291*
CONVERGENCE_CHECK_PARAMETERS *177*
converting between ascii, binary files *28*
COORDINATE *377*

D

declarative commands *9*
DENSITY_MODEL *293*
DERIVED_QUANTITY_OUTPUT *379*
DESIGN VARIABLE *41*
DESIGN VARIABLES FIELD *42*
DIFFUSIVITY_MODEL *300*

E

ELECTRICAL_RESISTIVITY_MODEL *305*
element data commands *454*
ELEMENT_BOUNDARY_CONDITION *490*
ELEMENT_OUTPUT *505*
ELEMENT_SET *509*
EMISSIVITY_MODEL *308*
enumerated *13*
EQUATION *44*
ERROR_ESTIMATOR_OUTPUT *380*
EXTERNAL_CODE *61*
EXTERNAL_CODE_SURFACE *526*
EXTERNAL_OUTPUT *383*

F

FAN_COMPONENT *532*
FEEDBACK_CONDITION *544*
FIELD *314*
FIELD_BOUNDARY_CONDITION *65*, *547*
FIELD_INTERACTION_MODEL *315*
file data format, reading data from files *23*
file extensions, relevance *29*
FLEXIBLE_BODY *66*
FREE_SURFACE *550*
functional commands *8*, *698*
FWH_OUTPUT *556*
FWH_SURFACE_OUTPUT *554*

G

GAS_KINETIC_MODEL *312*
global commands *35*
GRAVITY *229*
GUIDE_SURFACE *557*

H

HEAT_EXCHANGER_COMPONENT *562*

I

IMMERSED_BOUNDARY_SURFACE *78*
INCLUDE *705*
integer *15*
INTERFACE_SURFACE *576*
INTERPOLATED_MOTION_SURFACE *581*

L

LINE_SOURCE *585*
LINEAR_SOLVER_PARAMETERS *191*
list *17*

M

MASS_HEAT_SOURCE *236*
MASS_SPECIES_SOURCE *242*
material model commands *269*
MATERIAL_MODEL *329*
MATERIAL_RADIATION_MODEL *334*
MESH_BOUNDARY_CONDITION *590*
MESH_MOTION *80*
MOMENTUM_SOURCE *248*
MULTI_FIELD_MODEL *336*
MULTIPLIER_FUNCTION *97*

N

nodal data commands *376*
NODAL SHAPES *107*
NODAL_BOUNDARY_CONDITION *402*
NODAL_INITIAL_CONDITION *389*
NODAL_OUTPUT *429*
NODAL_RESIDUAL_OUTPUT *427*

O

OBJECTIVE *109*
OPTIMIZATION *110*

P

parameter format *11*
parameter operators, functions *20*
PARTICLE_SURFACE *600*
PARTICLE_TRACE *113*
PERIODIC_BOUNDARY_CONDITION *431*
POROSITY_MODEL *339*

Q

QUIT *707*

R

RADIATION *114*
RADIATION_SURFACE *604*

real *16*
REFERENCE_FRAME *116*
RESPONSE VARIABLE *119*
RESTART *701*
RESTART_OUTPUT *446*
ROTATION_FORCE *254*
RUN *699*
RUNNING_AVERAGE_OUTPUT *448*

S

SIMPLE_BOUNDARY_CONDITION *611*
SOLAR_RADIATION *126*
SOLAR_RADIATION_MODEL *347*
SOLAR_RADIATION_SURFACE *663*
solution strategy commands *146*
SPECIFIC_HEAT_MODEL *351*
STAGGER *198*
string *12*
SURFACE_INTEGRATED_CONDITION *669*
SURFACE_OUTPUT *674*
SURFACE_SET *679*
SURFACE_TENSION_MODEL *358*

T

THERMAL_SHELL *131*
TIME_ INCREMENT *210*
TIME_AVERAGE_OUTPUT *449*
TIME_HISTORY_OUTPUT *451*
TIME_INTEGRATION *215*
TIME_SEQUENCE *219*
TURBULENCE_MODEL_PARAMETERS *133*
TURBULENCE_WALL *682*

U

USER_GLOBAL_DATA *142*

V

VAO_ELEMENT_OUTPUT *689*
VAO_OUTPUT *691*
VAO_SURFACE_OUTPUT *692*
VISCOELASTIC_MODEL *361*
VISCOSITY_MODEL *365*
VOLUME_HEAT_SOURCE *256*
VOLUME_SET *694*
VOLUME_SPECIES_SOURCE *263*

W

WAVE_BOUNDARY_CONDITION *143*
WAVE_DAMPING_SOURCE *145*