ALTAIR

ONLY FORWARD

Altair® AcuSolve® 2025

AcuTrace Command Reference Manual

Updated: 11/13/2024

altair.com

Contents

AcuTrace Command Reference Manual

1

Commands of AcuTrace, a particle tracer that runs as a post-processor to or a co-processor with AcuSolve.

AcuTrace computes particle traces as a series of segments using a fifth-order time-discontinuous Galerkin (TDG) method with error control for solving ordinary differential equations. AcuTrace solves the particle motion and stretch evolution equations. These can be augmented by one or more user defined evolution equations. It computes traces for unsteady as well as steady flow fields, for flows with mesh motion as well as without, and for flows computed on meshes with interface surfaces. AcuTrace can also compute traces for flows on meshes with multiple reference frames in two ways; by converting flow velocities to the local reference frame or by treating the boundaries between stationary and rotating reference frames as interface surfaces and the flow as pseudotransient.

To solve a problem with AcuTrace, you must first run AcuSolve. This can be done concurrently. You must also create a trace input file. Once these two steps are complete, AcuTrace is invoked with the command `acuRunTrace`. See the AcuSolve Programs Reference Manual for further details.

The trace input file consists of one or more commands, each having zero or more parameters. These commands define the problem parameters, such as solution strategies, initial conditions and output.

The command style for AcuTrace, command format, command qualifier usage, parameter format, parameter operators and functions, is identical to the style used by AcuSolve. For these details, see the AcuSolve Command Reference Manual.

Global Commands

2

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

This chapter covers the following:

EQUATION

Specifies the equation systems present in the problem.

Type

AcuTrace Command

Syntax

`EQUATION {parameters}`

Qualifier

This command has no qualifier.

Parameters

`particle` *(enumerated) [=massless]*
Type of the particle motion equation.

massless	Massless particle equation.
finite_mass	Finite mass particle equation.

`stretch` *(enumerated) [=none]*
Type of the stretch equation.

none	No stretch equation present.
standard	Standard stretch equation.

`user_equations` *(list) [={}]*
List of user equations to be used.

`number_particle_components` *or* `num_part_comps` *(integer) >=0 [=0]*
Number of particle components.

Description

The `EQUATION` command specifies the existence and types of the equation systems and solution fields present in the problem. For example, to specify a problem with particle position using the massless model and stretch, you need:

```
EQUATION {
   particle = massless
   stretch  = standard
}
```

while to use the finite mass model instead, you need:

```
EQUATION {
   particle = finite_mass
   stretch  = standard
}
```

In the massless model, the velocity of a particle always equals the flow velocity at the particle's location. In the finite mass model, the particle velocity does not necessarily equal the flow velocity. Instead, the particles have mass and are subject to forces such as drag and gravity.

The massless particle and stretch equations are:

$$\frac{D\vec{x}_p}{Dt} = \vec{u}_f$$
$$\frac{D\vec{l}_p}{Dt} = \nabla\vec{u}_f{}^T\vec{l}_p \tag{1}$$

where $\frac{D}{Dt}$ denotes the derivative in the particle frame of reference, t is time, $\vec{x}_p$ is the particle position, $\vec{u}_f$ is the flow velocity, $\nabla\vec{u}_f T$ is the transpose of the velocity gradient, and $\vec{l}_p$ is the stretch vector.

If the parameter `turbulence_trace` in the `TRACE_PARAMETERS` command is set to on, the massless particle equation solved is not the above but instead

$$\frac{D\vec{x}_p}{Dt} = \vec{u}_f + \vec{u}_{turb} \tag{2}$$

where $\vec{u}_{turb}$ is a random perturbation to the particle velocity based on the eddy viscosity.

The finite mass particle equations are

$$\frac{D\vec{x}_p}{Dt} = \vec{u}_p$$
$$m_p\frac{D\vec{u}_p}{Dt} = \vec{F} \tag{3}$$

where $\vec{u}_p$ is the particle velocity, m_p is the particle mass, and $\vec{F}$ is the sum of the forces acting on the particle. If the parameter `turbulence_trace` in the `TRACE_PARAMETERS` command is set to on, then the values of $\vec{u}_f$ used in evaluating $\vec{F}$ are replaced by $\vec{u}_f+\vec{u}_{turb}$.

> **Note:** Particles are assumed to be spherical of constant (subgrid scale) size.

The particle and stretch equations can be augmented by one or more user equations. The user equations can either be evolution equations of the form

$$\frac{D\vec{s}_p}{Dt} = \vec{f}(\vec{s}_p, \vec{x}_p, t, \vec{l}_p, \vec{S}_p, \vec{U}_f; \vec{V}_{udf}) \tag{4}$$

where $\vec{s}_p$ is the set of user-defined particle variables in the current equation, $\vec{S}_p$ is the set of user-defined variables from the other user equations (if any), $\vec{U}_f$ is the set of flow variables, and $\vec{V}_{udf}$ is the set of user equation parameters; or evaluation equations of the form

$$\vec{s}_p = \vec{f}(\vec{s}_p, \vec{x}_p, t, \vec{l}_p, \vec{S}_p, \vec{U}_f; \vec{V}_{udf}) \tag{5}$$

The user equations present in the problem are specified by the *user_equations* parameter. Each user equation specified by the *user_equations* parameter must be defined by a USER_EQUATION command.

For example, evolution equations tparticle and cparticle for particle temperature and composition could be defined. To include these equations in a problem, one could have:

```
EQUATION {
...
   user_equations = {tparticle, cparticle}
}
USER_EQUATION( "tparticle" ) {
   user_function = "usrTparticle"
   num_variables = 1
...
}
USER_EQUATION( "cparticle" ) {
   user_function = "usrCparticle"
   num_variables = 2                          # 2 species example
...
}
```

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 particle position and stretch, you could have:

```
EQUATION {
   particle = massless
   stretch = standard
}
TIME_SEQUENCE {
   staggers = { "particle", "stretch" }
...
}
STAGGER( "particle" ) {
   equation = particle
...
}
STAGGER( "stretch" ) {
   equation = stretch
...
}
```

As with *particle* and *stretch* equations, the TIME_SEQUENCE and STAGGER commands must be used to request the solution for the specified user equations. For example, to solve for particle position, temperature, and composition, you could have:

```
EQUATION {
   particle       = massless
   user_equations = {tparticle, cparticle}
}
TIME_SEQUENCE {
   staggers       = { "particle", "tparticle", "cparticle"}
...
}
STAGGER( "particle" ) {
   equation = particle
...
}
```

```
STAGGER( "tparticle" ) {
   equation      = user_equation
   user_equation = "cparticle"
...
}
STAGGER( "cparticle" ) {
   equation      = user_equation
   user_equation = "tparticle"
...
}
USER_EQUATION( "tparticle" ) {
   user_function = "usrTparticle"
   num_variables = 1
...
}
USER_EQUATION( "cparticle" ) {
   user_function = "usrCparticle"
   num_variables = 2
...
}
```

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.

Particle components are a generalization of a marker variable; they can also be considered a generalization of species or composition. When *number_particle_components* is greater than zero, particle components are initialized in the `PARTICLE_SEED` command. The values of the particle components are constant in time but they may vary over the particles in the problem according to how they are initialized.

FLOW_FIELD

Specifies the flow field used in the problem.

Type

AcuTrace Command

Syntax

`FLOW_FIELD {parameters}`

Qualifier

This command has no qualifier.

Parameters

`flow_field_type` or *`mode`* *(enumerated) [=static]*

Type of AcuSolve flow data.

static or steady	Steady flow.
dynamic or transient	Transient flow.
cyclic_dynamic or cyclic	Cyclic flow.
pseudodynamic or pseudotransient	Pseudodynamic flow.
one_way_coupling	One way coupling with AcuSolve.
two_way_coupling	Two way coupling with AcuSolve.

`from_run` or *`run`* *(integer) [=0]*

Number of the AcuSolve run used to define the flow field. If *`from_run`* is set to 0, the last run in the working directory is assumed. Not used if *`flow_field_type`* is one_way_coupling or two_way_coupling.

`from_directory` or *`dir`* *(string) [="ACUSIM.DIR"]*

All internal files are read from or stored in this directory. This directory does not need to be on the same file system as the user-supplied input files.

`from_problem` or *`problem`* *(string) [no default]*

The name of the problem is specified via this option. This name is used to build internal file names and to generate output files. The names of all generated output files start with the problem name.

`from_time_step` or *`step`* *(integer) [=0]*

The time step used to define the flow data. If *`from_time_step`* is set to 0 the last time step of *`from_run`* in the working directory is assumed. Used only if *`flow_field_type`* is static or pseudodynamic.

`time_step_type` *(enumerated) [=all]*

The type of time step specification. Used only if *`flow_field_type`* is dynamic.

all	Use all available time steps.
time_step_series or series	Use a user-defined series of time steps.

`cyclic_time_step_type` *(enumerated) [=range]*
The type of time step specification. Used only if `flow_field_type` is cyclic_dynamic.

range	Use a range of time steps.
time_step_series or series	Use a user-defined series of time steps.

`time_step_series` *or* `steps` *(array) [={}]*
List of time step numbers defining the flow field. Used only if `flow_field_type` is dynamic and `time_step_type` is series or `flow_field_type` is cyclic_dynamic and `cyclic_time_step_type` is time_step_series.

`first_cyclic_time_step` *or* `first_step` *(integer) [=0]*
The first step number in a range of time steps. Used only if `flow_field_type` is cyclic_dynamic and `cyclic_time_step_type` is range.

`last_cyclic_time_step` *or* `last_step` *(integer) [=0]*
The last step number in a range of time steps. Used only if `flow_field_type` is cyclic_dynamic and `cyclic_time_step_type` is range.

`cyclic_end_time_steps` *(enumerated) [=include_first]*
The specification of how the end points of a cycle of steps are used. Used only if `flow_field_type` is cyclic_dynamic.

include_first	The flow data for the first step is used as the flow for the last step. The data for the last step is ignored.
include_last	The flow data for the last step is used as the flow for the first step. The data for the first step is ignored.
include_both	The flow data from both the first and the last steps are used.
average_both	The flow data at the cycle end points is the average of the first and last steps.

`mesh_motion` *(boolean) [=on]*
Flag specifying if mesh motion is active. Ignored if there is no mesh motion in the AcuSolve solution. Not used if `flow_field_type` is pseudodynamic.

`pseudodynamic_mesh_update` *(enumerated) [=max_angle]*
Flag specifying how often the mesh is rotated if `flow_field_type` is pseudodynamic. Flag specifying if mesh motion is active. Ignored if there is no mesh motion in the AcuSolve solution. Not used if `flow_field_type` is pseudodynamic.

`pseudodynamic_time_increment` *(real) [=1]*
The time between mesh update. Used only if `flow_field_type` is pseudodynamic and `pseudodynamic_mesh_update` is time_increment.

extended_flow_variables *(list) [={}]*
List of extended variables in the AcuSolve database accessible to user equations.

coupling_socket_port *(integer) [=20000]*
Code port number for establishing socket connection to AcuSolve. Used only if *flow_field_type* is one_way_coupling.

Description

The FLOW_FIELD command specifies the AcuSolve flow data used to advance the particle trace equations and defines how that data is used. AcuTrace can use flow data either from a completed AcuSolve simulation or from a concurrently running AcuSolve simulation.

For a completed AcuSolve simulation, there are up to four possible ways in which the flow data can be used.

When *flow_field_type* is steady, a single time step from a completed simulation is used to define a steady flow field with which to advance the particles. The particle simulation itself is of course transient because the particles are moving. By default, the last available step is used. This is generally the best practice, but any other step could be set with the *step* parameter, for example,

```
FLOW_FIELD {
   ...
   flow_field_type = static
   step            = 10
   ...
}
```

For dynamic flow, a set of time steps from an AcuSolve simulation is used to define a transient flow field. For particle times between time steps, the flow field is interpolated in time. Before the earliest time and after the last time, the flow field is extrapolated as constant in time. By default, all the steps from a transient solution are used. This is generally the best practice, but a subset of steps can be selected, for example,

```
FLOW_FIELD {
   ...
   flow_field_type = dynamic
   time_step_type  = series
   steps           = { 2, 4, 6, 8 }
   ...
}
```

The cyclic flow field type defines a transient flow field in a similar manner, except that the flow repeats in time after the last time step is reached. This option is useful for tracing particles in time-periodic, for example, cyclic, flows in which the particle trace time is much greater than the flow cycle time because a relatively short flow simulation can be used. When using this option, one should be careful that the flow is indeed cyclic within a reasonable tolerance and that the set of time steps truly represents one or more complete flow cycles.

When *cyclic_end_time_steps* equals include_first, include_last, or average_both, the cycle time is the last time in the cycle minus the first time in the cycle. When *cyclic_end_time_steps* equals include_both, the cycle time is the last time in the cycle minus the first time in the cycle plus the difference between the last two steps in the cycle.

For example, for the command

```
FLOW_FIELD {
   ...
   flow_field_type        = cyclic
   cyclic_time_step_type = series
   steps                  = { 1, 2, 3 }
   ...
}
```

and assuming the times of steps 1-3 are 10, 16, and 20, the cycle time is 20-10 = 10 when *cyclic_end_time_steps* equals include_first, include_last, or average_both, and 20-10 + (20 - 16) = 14 when *cyclic_end_time_steps* equals include_both.

When *cyclic_end_time_steps* equals include_first, include_last, or average_both, the flow fields at the first and last steps of the cycle are considered the same. When *cyclic_end_time_steps* equals include_first, the flow field from the first step is used; for the example above, the flow field used by AcuTrace would cycle 1, 2, 1, 2, and so on. When *cyclic_end_time_steps* equals include_last, the flow field from the last step is used; in the example, the flow field would cycle 2, 3, 2, 3, and so on. When *cyclic_end_time_steps* equals average_both, the flow field at the the cycle endpoint is the average of the fields at the first and last step; in the example, the flow field would cycle 2, average of 2 and 3, 2, average of 2 and 3, and so on. When *cyclic_end_time_steps* equals include_both, all steps in the cycle are used; in the example, the flow field would cycle 1, 2, 3, 1, 2, 3, and so on.

The AcuTrace times over the course of the cycle definition are used as is from the AcuSolve run. Outside the times of the cycle definition, the times map into the times in the cycle definition. For example, consider the following inputs:

```
FLOW_FIELD {
   ...
   flow_field_type        = cyclic
   cyclic_time_step_type = series
   steps                  = { 100, 200, 300, 400, 500 }
   cyclic_end_time_steps = include_both
   ...
}
```

and suppose that the times corresponding to these steps in the AcuSolve run are 10, 20, 30, 40, and 50. Then these same times are used in AcuTrace in the following way: at time 10 in the AcuTrace run, the flow field corresponds to the AcuSolve flow field at step 100, time 20 corresponds to step 200, and so on. Before time 10 or after time 50, the AcuSolve solution repeats: time 60 in the AcuTrace run uses the flow field from time 10 in the AcuSolve solution, time 70 uses the flow field from time 20, and so on.

Note: Time 0 uses the AcuSolve flow field at time 50.

When *flow_field_type* is pseudodynamic, a single step from an AcuSolve simulation with multiple reference frames is used to create a time varying flow field with mesh motion. Before particles are advanced, the boundaries between element sets with different reference frames are converted to sliding interface surfaces. As the particle trace advances, the mesh, flow velocity, and other flow variables in the non-stationary reference frames are rotated in time according to the reference frame

definitions. The flow data so generated approximates the flow data that AcuSolve would produce for a truly dynamic simulation with mesh motion and sliding interface surfaces.

With pseudodynamic flow, the flow data and mesh do not update continuously but instead are updated at regular intervals, just as if the data were being read from a transient AcuSolve solution. If *pseudodynamic_mesh_update* equals time_increment, the update interval is simply the value of *pseudodynamic_time_increment*. If, on the other hand, *pseudodynamic_mesh_update* equals max_angle, the solution update interval is set to the time it takes the fastest rotating reference frame in the AcuSolve solution to rotate though an angle of *pseudodynamic_max_angle*. For example if there are two rotating reference frames in the AcuSolve solution with rotation speeds of 5 and 20 rpm, and *pseudodynamic_max_angle* equals 10, the update interval is set to 10/(360*20) * 60 = 1/12 second.

When *flow_field_type* is pseudodynamic, by default the last step of the simulation is used. This is generally the best practice. However, any other step could be set with the *step* parameter, for example,

```
FLOW_FIELD {
   ...
   flow_field_type = pseudodynamic
   step            = 10
   ...
}
```

The command

```
acuTrans -out -ts A -outv ,
```

will list the available steps in an AcuSolve run.

> **Note:** There is a "," at the end of the command.

The flow data from a transient concurrently running AcuSolve simulation is used when *flow_field_type* is one_way_coupling. The particle trace is identical to what AcuTrace would compute with a *flow_field_type* of dynamic if the AcuSolve simulation first ran to completion. The advantage in using a *flow_field type* of one_way_coupling is that the AcuSolve disk requirements for long running particle traces in transient flow fields can be greatly reduced. One way coupling is also called unidirectional coupling.

When *flow_field_type* is two_way_coupling, the flow data from a transient concurrently running AcuSolve simulation is used as well. In addition, particle source terms and values are sent from AcuTrace to AcuSolve. The source terms and values sent as well as how AcuSolve uses them are specified in the COUPLING_FIELDS command. See the COUPLING_FIELDS section of this manual for more details. Two way coupling is also called bidirectional coupling.

The AcuSolve flow nodal output accessible to user equations always includes flow velocity, pressure, temperature, eddy viscosity, species and velocity gradient. In addition, some or all of the available extended nodal output values can be made accessible through the *extended_flow_variables* parameter. For example,

extended_flow_variables = { "material_viscosity", "strain_rate_invariant_2" }

makes material viscosity and the second strain rate invariant available for user equations. To use extended output values in AcuTrace, you should make sure that the proper AcuSolve commands are

used in setting up the AcuSolve run providing the data to AcuTrace. The nodal output values available in an existing AcuSolve run can be determined by executing:

```
acuTrans -out -extout -to stats
```

By default, AcuTrace uses nodal mesh displacements if they are provided in the AcuSolve database. It is possible to ignore these, for example set them to 0, by setting the parameter *mesh_motion* to off. This is not recommended, however. If there are no mesh displacements, a value of on for *mesh_motion* is ignored. The *mesh_motion* parameter is ignored if the *flow_field_type* is pseudodynamic.

USER_EQUATION

Specifies a user equation in the problem.

Type

AcuTrace Command

Syntax

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

Qualifier

User-given name.

Parameters

`user_function` *(string) [no default]*
Name of the C function to use.

`num_variables` *or* *`num_vars`* *(integer) >=1 [1]*
Number of values returned by the equation, that is, the size of the output array of the equation.

`user_values` *(array) [={}]*
User-defined constant parameters supplied to the equation.

`user_strings` *(list) [={}]*
List of user-defined constant strings supplied to the equation.

`type` *(enumerated) [=evolve]*
Controls how the user equation is used in the particle trace. In both cases, *u* is the variable advanced and *f* is the user equation.

evolve	User equation is the right hand side of the evolution equation du/dt=f.
evaluate	User equation is the right hand side of u=f.

Description

The `USER_EQUATION` command defines an equation that can then be included in the particle trace simulation with the *`user_equations`* parameter of the `EQUATION` command. The full definition of a user equation consists a user-defined function written in the C programming language plus additional constants provided by the *`user_values`* and *`user_strings`* parameters. A single C function can therefore be used to define multiple user equations by using different values of these constants in multiple `USER_EQUATION` commands. See the AcuTrace User-Defined Function Manual for a detailed description of user-defined functions for AcuTrace.

In the following example, there are two user equations ener and temp. The first is used as the source term in an evolution equation for particle energy, and the second is used to compute the particle temperature from the particle energy. There are two constants provided by you: the product of the particle mass and the particle specific heat; and the thermal conductivity. The constants have values of

two and three here. The two C functions are named *usrEner* and *usrTemp*. The input commands may then be given as:

```
EQUATION {
...
   user_equations = {ener, temp}
}
USER_EQUATION( "ener" ) {
   user_function  = "usrEner"
   num_variables  = 1
   user_strings   = {}
   user_values    = {3,2}
   type           = evolve
}
USER_EQUATION( "temp" ) {
   user_function  = "usrTemp"
   num_variables  = 1
   user_strings   = {}
   user_values    = {2}
   type           = evaluate
}
```

where the user-defined functions *usrEner* and *usrTemp* may be implemented as:

```
#include "acusim.h"
#include "ufp.h"
UFP_PROTOTYPE(usrEner);
Void usrEner (
      UfpHd        ufpHd,          /* Opaque handle for accessing data
 */
      Integer      nItems,         /* Number of items in outVec
 */
      Integer      vecDim,         /* Vector dimension of outVec
 */
      Real*        outVec,         /* Output Vector
 */
)
{
   Real*           usrVals ;       /* user values
 */
   Real*           fluid ;         /* fluid temperature
 */
   Real*           particle ;      /* particle energy
 */
   Real            mpXcp ;         /* mass * c_p for particle
 */
   Real            cond ;          /* conductivity
 */
   Real*           jac ;           /* source jacobian
 */
   cond                = usrVals[0] ;
   mpXcp               = usrVals[1] ;
   t_fluid             = ufpGetFlowData( ufpHd, UFP_FLOW_TEMPERATURE ) ;
   h_particle          = ufpGetUdfData( ufpHd, 0 ) ;
   outVec[0]           = -cond * ( h_particle[0] / mpXcp - t_fluid[0] ) ;
   jac                 = ufpGetJac( ufpHd, UFP_JAC_UDF_VARIABLES ) ;
   jac[0]              = -cond / mpXcp ;
}
UFP_PROTOTYPE(usrTemp);
Void usrTemp (
```

```
   UfpHd            ufpHd,           /* Opaque handle for accessing data
 */
   Integer          nItems,          /* Number of items in outVec
 */
   Integer          vecDim,          /* Vector dimension of outVec
 */
   Real*            outVec,          /* Output Vector
 */
)
{
   Real*            usrVals ;        /* user values
 */
   Real*            h_particle ;     /* particle energy
 */
   Real             mpXcp ;          /* mass * c_p for particle
 */
   usrVals              = ufpGetUsrVals( ufpHd ) ;
   mpXcp                = usrVals[0] ;
   h_particle           = ufpGetUdfData( ufpHd, "ener" ) ;
   outVec[0]            = h_particle[0] / mpXcp ;
}
```

In order for AcuTrace to access these functions, the source file containing them must be compiled and linked into a shared library. The scripts AcuMakeLib, under Linux, and AcuMakeDll, under Windows, may be used for this purpose.

Assume the function implementations are in the file `usrTemp.c`. The functions are compiled and linked into a shared library `libusr.so` by issuing the command:

```
acuMakeLib -src usrTemp.c
```

The shared library `libusr.so` is automatically loaded by AcuTrace when it is run.

COUPLING_FIELDS

Specifies which AcuTrace fields couple with AcuSolve and how they couple when *flow_field_type* is two_way_coupling.

Type

AcuTrace Command

Syntax

COUPLING_FIELDS {parameters}

Qualifier

This command has no qualifier.

Parameters

coupling_iterations or *coup_iters* *(integer) >=1 [=1]*

Number of AcuTrace iterations per AcuSolve time step. An AcuTrace iteration is an advance of all the particles from the beginning to the end of the AcuSolve time step.

momentum_type *(enumerated) [=none]*

Type of momentum coupling to AcuSolve. Specifies what type of momentum data is sent to AcuSolve.

none	No coupling (no data sent).
flux	Momentum source terms are sent to AcuSolve.

momentum_flux_type *(enumerated) [=finite_mass]*

When *momentum_type* equals flux, specifies how the momentum source terms sent to AcuSolve are computed.

finite_mass	Momentum source terms are computed according to the finite mass model.

temperature_type *(enumerated) [=none]*

Type of temperature coupling to AcuSolve. Specifies what type of temperature data is sent to AcuSolve.

none	No coupling (no data sent).
flux	Temperature source terms are sent to AcuSolve.
value	Temperature values are sent to AcuSolve.

temperature_flux_type *(enumerated) [=user_equation]*

When *temperature_type* equals flux, specifies how the temperature source terms sent to AcuSolve are computed.

user_equation	Temperature source terms are computed in a user equation.

`temperature_value_type` *(enumerated) [=user_equation]*

When `temperature_type` equals value, specifies how the temperature values sent to AcuSolve are computed.

user_equation	Temperature values are computed in a user equation.
component	Temperature values are set equal to a particle component.

`temperature_user_equation` *(string) [No default]*

When `temperature_type` equals flux and `temperature_flux_type` equals user_equation or `temperature_type` equals value and `temperature_value_type` equals user_equation, specifies which AcuTrace user equation provides the temperature values sent to AcuSolve.

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

When `temperature_type` equals flux and `temperature_flux_type` equals user_equation or `temperature_type` equals value and `temperature_value_type` equals user_equation, specifies which term in the user equation source term is used. `temperature_user_index` is 0 based. It must be greater than or equal to 0 and strictly less than the number of variables in the user equation.

`temperature_component_index` *(integer) >=0 [=0}*

When `temperature_type` equals value and `temperature_value_type` equals component, specifies which particle component is used. `temperature_component_index` is 0 based. It must be greater than or equal to 0 and strictly less than the number of particle components.

`temperature_scaling_factor` *(real) [=1.0]*

When `temperature_type` equals value, the temperature values sent to AcuSolve are multiplied by this value before they are sent.

`species_1_type` *(enumerated) [=none]*

Type of species 1 coupling to AcuSolve. Specifies what type of species 1 data is sent to AcuSolve.

none	No coupling (no data sent).
flux	Species 1 source terms are sent to AcuSolve.
value	Species 1 values are sent to AcuSolve.

`species_1_flux_type` *(enumerated) [=user_equation]*

When `species_1_type` equals flux, specifies how the species 1 source terms sent to AcuSolve are computed.

user_equation

Species 1 source terms are computed in a user equation

`species_1_value_type` *(enumerated) [=component]*

When `species_1_type` equals value, specifies how the species 1 values sent to AcuSolve are computed.

user_equation

Species 1 values are computed in a user equation.

component
Species 1 values are set equal to a particle component.

`species_1_user_equation` *(string) [No default]*
When `species_1_type` equals flux and `species_1_flux_type` equals user_equation or `species_1_type` equals value and `species_1_value_type` equals user_equation, specifies which AcuTrace user equation provides the species 1 values sent to AcuSolve.

`species_1_user_index` *(integer) >=0 [=0]*
When `species_1_type` equals flux and `species_1_flux_type` equals user_equation or `species_1_type` equals value and `species_1_value_type` equals user_equation, specifies which term in the user equation source term is used. `species_1_user_index` is 0 based. It must be greater than or equal to 0 and strictly less than the number of variables in the user equation.

`species_1_component_index` *(integer) >=0 [=0]*
When `species_1_type` equals value and `species_1_value_type` equals component, specifies which particle component is used. `species_1_component_index` is 0 based. It must be greater than or equal to 0 and strictly less than the number of particle components.

`species_1_scaling_factor` *(real) [=1.0]*
When `species_1_type` equals value, the species 1 values sent to AcuSolve are multiplied by this value before they are sent.

`species_2_type` *(enumerated) [=none]*
Type of species 2 coupling to AcuSolve. Specifies what type of species 2 data is sent to AcuSolve.

none	No coupling (no data sent).
flux	Species 2 source terms are sent to AcuSolve.
value	Species 2 values are sent to AcuSolve.

`species_2_flux_type` *(enumerated) [=user_equation]*
When `species_2_type` equals flux, specifies how the species 2 source terms sent to AcuSolve are computed.

user equation	Species 2 source terms are computed in a user equation.

`species_2_value_type` *(enumerated) [=component]*
When `species_2_type` equals value, specifies how the species 2 values sent to AcuSolve are computed.

user equation	Species 2 values are computed in a user equation.
component	Species 2 values are set equal to a particle component.

`species_2_user_equation` *(string) [No default]*
When `species_2_type` equals flux and `species_2_flux_type` equals user_equation or `species_2_type` equals value and `species_2_value_type` equals user_equation, specifies which AcuTrace user equation provides the species 2 values sent to AcuSolve.

`species_2_user_index` *(integer) >=0 [=0]*
When `species_2_type` equals flux and `species_2_flux_type` equals user_equation or `species_2_type` equals value and `species_2_value_type` equals user_equation, specifies which

term in the user equation source term is used. *species_2_user_index* is 0 based. It must be greater than or equal to 0 and strictly less than the number of variables in the user equation.

species_2_component_index *(integer) >=0 [=0]*
When *species_2_type* equals value and *species_2_value_type* equals component, specifies which particle component is used. *species_2_component_index* is 0 based. It must be greater than or equal to 0 and strictly less than the number of particle components.

species_2_scaling_factor *(real) [=1.0]*
When *species_2_type* equals value, the species 2 values sent to AcuSolve are multiplied by this value before they are sent.

species_3_type *(enumerated) [=none]*
Type of species 3 coupling to AcuSolve. Specifies what type of species 3 data is sent to AcuSolve.

none	No coupling (no data sent).
flux	Species 3 source terms are sent to AcuSolve.
value	Species 3 values are sent to AcuSolve.

species_3_flux_type *(enumerated) [=user_equation]*
When *species_3_type* equals flux, specifies how the species 3 source terms sent to AcuSolve are computed.

user_equation	Species 3 source terms are computed in a user equation

species_3_value_type *(enumerated) [=component]*
When *species_3_type* equals value, specifies how the species 3 values sent to AcuSolve are computed.

user_equation	Species 3 values are computed in a user equation.
component	Species 3 values are set equal to a particle component.

species_3_user_equation *(string) [No default]*
When *species_3_type* equals flux and *species_3_flux_type* equals user_equation or *species_3_type* equals value and *species_3_value_type* equals user_equation, specifies which AcuTrace user equation provides the species 3 values sent to AcuSolve.

species_3_user_index *(integer) >=0 [=0]*
When *species_3_type* equals flux and *species_3_flux_type* equals user_equation or *species_3_type* equals value and *species_3_value_type* equals user_equation, specifies which term in the user equation source term is used. *species_3_user_index* is 0 based. It must be greater than or equal to 0 and strictly less than the number of variables in the user equation.

species_3_component_index *(integer) >=0 [=0]*
When *species_3_type* equals value and *species_3_value_type* equals component, specifies which particle component is used. *species_3_component_index* is 0 based. It must be greater than or equal to 0 and strictly less than the number of particle components.

species_3_scaling_factor *(real) [=1.0]*
When *species_3_type* equals value, the species 3 values sent to AcuSolve are multiplied by this value before they are sent.

`species_4_type` *(enumerated) [=none]*
Type of species 4 coupling to AcuSolve. Specifies what type of species 4 data is sent to AcuSolve.

none	No coupling (no data sent).
flux	Species 4 source terms are sent to AcuSolve.
value	Species 4 values are sent to AcuSolve.

`species_4_flux_type` *(enumerated) [=user_equation]*
When `species_4_type` equals flux, specifies how the species 4 source terms sent to AcuSolve are computed.

user_equation
Species 4 source terms are computed in a user equation.

`species_4_value_type` *(enumerated) [=user_equation]*
When `species_4_type` equals value, specifies how the species 4 values sent to AcuSolve are computed.

user_equation	Species 4 values are computed in a user equation.
component	Species 4 values are set equal to a particle component.

`species_4_user_equation` *(string) [No default]*
When `species_4_type` equals flux and `species_4_flux_type` equals user_equation or `species_4_type` equals value and `species_4_value_type` equals user_equation, specifies which AcuTrace user equation provides the species 4 values sent to AcuSolve.

`species_4_user_index` *(integer) >=0 [=0]*
When `species_4_type` equals flux and `species_4_flux_type` equals user_equation or `species_4_type` equals value and `species_4_value_type` equals user_equation, specifies which term in the user equation source term is used. `species_4_user_index` is 0 based. It must be greater than or equal to 0 and strictly less than the number of variables in the user equation.

`species_4_component_index` *(integer) >=0 [=0]*
When `species_4_type` equals value and `species_4_value_type` equals component, specifies which particle component is used. `species_4_component_index` is 0 based. It must be greater than or equal to 0 and strictly less than the number of particle components.

`species_4_scaling_factor` *(real) [=1.0]*
When `species_4_type` equals value, the species 4 values sent to AcuSolve are multiplied by this value before they are sent.

`species_5_type` *(enumerated) [=none]*
Type of species 5 coupling to AcuSolve. Specifies what type of species 5 data is sent to AcuSolve.

none	No coupling (no data sent).
flux	Species 5 source terms are sent to AcuSolve.
value	Species 5 values are sent to AcuSolve.

`species_5_flux_type` *(enumerated) [=user_equation]*

When `species_5_type` equals flux, specifies how the species 5 source terms sent to AcuSolve are computed.

user_equation	Species 5 source terms are computed in a user equation.

`species_5_value_type` *(enumerated) [=component]*

When `species_5_type` equals value, specifies how the species 5 values sent to AcuSolve are computed.

user_equation	Species 5 values are computed in a user equation.
component	Species 5 values are set equal to a particle component.

`species_5_user_equation` *(string) [No default])*

When `species_5_type` equals flux and `species_5_flux_type` equals user_equation or `species_5_type` equals value and `species_5_value_type` equals user_equation, specifies which AcuTrace user equation provides the species 5 values sent to AcuSolve.

`species_5_user_index` *(integer) >=0 [=0]*

When `species_5_type` equals flux and `species_5_flux_type` equals user_equation or `species_5_type` equals value and `species_5_value_type` equals user_equation, specifies which term in the user equation source term is used. `species_5_user_index` is 0 based. It must be greater than or equal to 0 and strictly less than the number of variables in the user equation.

`species_5_component_index` *(integer) >=0 [=0)]*

When `species_5_type` equals value and `species_5_value_type` equals component, specifies which particle component is used. `species_5_component_index` is 0 based. It must be greater than or equal to 0 and strictly less than the number of particle components.

`species_5_scaling_factor` *(real) [=1.0]*

When `species_5_type` equals value, the species 5 values sent to AcuSolve are multiplied by this value before they are sent.

`species_6_type` *(enumerated) [=none]*

Type of species 6 coupling to AcuSolve. Specifies what type of species 6 data is sent to AcuSolve.

none	No coupling (no data sent).
flux	Species 6 source terms are sent to AcuSolve.
value	Species 6 values are sent to AcuSolve.

`species_6_flux_type` *(enumerated) [=user_equation]*

When `species_6_type` equals flux, specifies how the species 6 source terms sent to AcuSolve are computed.

user_equation	Species 6 source terms are computed in a user equation.

`species_6_value_type` *(enumerated) [=component]*

When `species_6_type` equals value, specifies how the species 6 values sent to AcuSolve are computed.

user_equation	Species 6 values are computed in a user equation.

component Species 6 values are set equal to a particle component.

`species_6_user_equation` *(string) [No default]*
When `species_6_type` equals flux and `species_6_flux_type` equals user_equation or `species_6_type` equals value and `species_6_value_type` equals user_equation, specifies which AcuTrace user equation provides the species 6 values sent to AcuSolve.

`species_6_user_index` *(integer) >=0 [=0)]*
When `species_6_type` equals flux and `species_6_flux_type` equals user_equation or `species_6_type` equals value and `species_6_value_type` equals user_equation, specifies which term in the user equation source term is used. `species_6_user_index` is 0 based. It must be greater than or equal to 0 and strictly less than the number of variables in the user equation.

`species_6_component_index` *(integer) >=0 [=0]*
When `species_6_type` equals value and `species_6_value_type` equals component, specifies which particle component is used. `species_6_component_index` is 0 based. It must be greater than or equal to 0 and strictly less than the number of particle components.

`species_6_scaling_factor` *(real) [=1.0]*
When `species_6_type` equals value, the species 6 values sent to AcuSolve are multiplied by this value before they are sent.

`species_7_type` *(enumerated) [=none]*
Type of species 7 coupling to AcuSolve. Specifies what type of species 7 data is sent to AcuSolve.

none No coupling (no data sent).

flux Species 7 source terms are sent to AcuSolve.

value Species 7 values are sent to AcuSolve.

`species_7_flux_type` *(enumerated) [=user_equation]*
When `species_7_type` equals flux, specifies how the species 7 source terms sent to AcuSolve are computed.

user_equation Species 7 source terms are computed in a user equation.

`species_7_value_type` *(enumerated) [=component]*
When `species_7_type` equals value, specifies how the species 7 values sent to AcuSolve are computed.

user_equation Species 7 values are computed in a user equation.

component Species 7 values are set equal to a particle component.

`species_7_user_equation` *(string) [No default]*
When `species_7_type` equals flux and `species_7_flux_type` equals user_equation or `species_7_type` equals value and `species_7_value_type` equals user_equation, specifies which AcuTrace user equation provides the species 7 values sent to AcuSolve.

`species_7_user_index` *(integer) >=0 [=0]*
When `species_7_type` equals flux and `species_7_flux_type` equals user_equation or `species_7_type` equals value and `species_7_value_type` equals user_equation, specifies which

term in the user equation source term is used. *species_7_user_index* is 0 based. It must be greater than or equal to 0 and strictly less than the number of variables in the user equation.

species_7_component_index *(integer) >=0 [=0]*
When *species_7_type* equals value and *species_7_value_type* equals component, specifies which particle component is used. *species_7_component_index* is 0 based. It must be greater than or equal to 0 and strictly less than the number of particle components.

species_7_scaling_factor *(real) [=1.0]*
When *species_7_type* equals value, the species 7 values sent to AcuSolve are multiplied by this value before they are sent.

species_8_type *(enumerated) [=none]*
Type of species 8 coupling to AcuSolve. Specifies what type of species 8 data is sent to AcuSolve.

none	No coupling (no data sent).
flux	Species 8 source terms are sent to AcuSolve.
value	Species 8 values are sent to AcuSolve.

species_8_flux_type *(enumerated) [=component]*
When *species_8_type* equals flux, specifies how the species 8 source terms sent to AcuSolve are computed.

user_equation	Species 8 source terms are computed in a user equation.

species_8_value_type (enumerated) [=user_equation]
When *species_8_type* equals value, specifies how the species 8 values sent to AcuSolve are computed.

user_equation	Species 8 values are computed in a user equation.
component	Species 8 values are set equal to a particle component.

species_8_user_equation *(string) [No default]*
When *species_8_type* equals flux and *species_8_flux_type* equals user_equation or *species_8_type* equals value and *species_8_value_type* equals user_equation, specifies which AcuTrace user equation provides the species 8 values sent to AcuSolve.

species_8_user_index *(integer) >=0 [=0]*
When *species_8_type* equals flux and *species_8_flux_type* equals user_equation or *species_8_type* equals value and *species_8_value_type* equals user_equation, specifies which term in the user equation source term is used. *species_8_user_index* is 0 based. It must be greater than or equal to 0 and strictly less than the number of variables in the user equation.

species_8_component_index *(integer) >=0 [=0]*
When *species_8_type* equals value and *species_8_value_type* equals component, specifies which particle component is used. *species_8_component_index* is 0 based. It must be greater than or equal to 0 and strictly less than the number of particle components.

species_8_scaling_factor *(real) [=1.0]*
When *species_8_type* equals value, the species 8 values sent to AcuSolve are multiplied by this value before they are sent.

`species_9_type` *(enumerated) [=none]*
Type of species 9 coupling to AcuSolve. Specifies what type of species 9 data is sent to AcuSolve.

none	No coupling (no data sent).
flux	Species 9 source terms are sent to AcuSolve.
value	Species 9 values are sent to AcuSolve.

`species_9_flux_type` *(enumerated) [=user_equation]*
When `species_9_type` equals flux, specifies how the species 9 source terms sent to AcuSolve are computed.

user_equation	Species 9 source terms are computed in a user equation.

`species_9_value_type` *(enumerated) [=component]*
When `species_9_type` equals value, specifies how the species 9 values sent to AcuSolve are computed.

user_equation	Species 9 values are computed in a user equation.
component	Species 9 values are set to a particle component.

`species_9_user_equation` *(string) [No default]*
When `species_9_type` equals flux and `species_9_flux_type` equals user_equation or `species_9_type` equals value and `species_9_value_type` equals user_equation, specifies which AcuTrace user equation provides the species 9 values sent to AcuSolve.

`species_9_user_index` *(integer) >=0 [=0]*
When `species_9_type` equals flux and `species_9_flux_type` equals user_equation or `species_9_type` equals value and `species_9_value_type` equals user_equation, specifies which term in the user equation source term is used. `species_9_user_index` is 0 based. It must be greater than or equal to 0 and strictly less than the number of variables in the user equation.

`species_9_component_index` *(integer) >=0 [=0]*
When `species_9_type` equals value and `species_9_value_type` equals component, specifies which particle component is used. `species_9_component_index` is 0 based. It must be greater than or equal to 0 and strictly less than the number of particle components.

`species_9_scaling_factor` *(real) [=1.0]*
When `species_9_type` equals value, the species 9 values sent to AcuSolve are multiplied by this value before they are sent.

Description

When the `flow_field_type` parameter in the `FLOW_FIELD` command is two_way_coupling, particle source terms and values are sent from AcuTrace to a concurrently running AcuSolve simulation. The source terms and values sent as well as how AcuSolve uses them are specified in the `COUPLING_FIELDS` command.

The set up for a coupled particle-flow problem with AcuSolve and AcuTrace also requires several general inputs in the AcuSolve input file regardless of the details of the coupling:

- Particle trace must be enabled:

```
EQUATION {
      ...
      particle_trace              = on
      ...
}
```

- The host on which AcuTrace runs must be specified in the `PARTICLE_TRACE` command:

```
PARTICLE_TRACE{
      ...
      socket_host                 = "avocet"
      ...
}
```

  In this example, AcuTrace runs on a host named avocet.

- Either `AUTO_SOLUTION_STRATEGY` must be used:

```
AUTO_SOLUTION_STRATEGY {
      ...
}
```

or a stagger for the particle trace must be defined and included as part of the `TIME_SEQUENCE` command:

```
TIME_SEQUENCE {
      ...
      staggers                    = { "flow",
                                  "particle_trace" }
      ...
}
STAGGER( "particle_trace" ) {
      equation                    = particle_trace
}
```

Two examples now follow. The first example illustrates flux, that is, source term, coupling. In this example, particles exchange energy via conductive heat transfer. The `COUPLING_FIELDS` command

```
COUPLING_FIELDS {
      temperature_type            = flux
      temperature_flux_type       = user_equation
      temperature_user_equation   = energy
}
```

specifies that the AcuTrace user equation energy provides a source term to the AcuSolve temperature equation.

The user equation energy must be used in an active AcuTrace stagger for the coupling to have effect. For example,

```
USER_EQUATION( "energy" ) {
      user_function               = "usrEnergy"
      num_variables               = 1
}
EQUATION {
      ...
      user_equations              = {energy}
```

```
      ...
}
STAGGER( energy ) {
      equation                    = user_equation
      user_equation               = "energy"
}
TIME_SEQUENCE {
      ...
      staggers                    = {particle,energy}
      ...
}
```

The C AcuTrace user function usrEnergy could be written as follows:

```
Void usrEnergy (
               UfpHd       ufpHd,
               Real*       outVec
               Integer     nItems,
               Integer     vecDim
)
{
      ...
      outVec[0] =  cond /  ( m_p * cp_p ) * e_particle[0] -
                   cond * t_fluid[0] ;
      ...
}
```

outVec[0] contains the source term for the particle energy; the source term provided to AcuSolve is this same value times -1.

See the AcuTrace User-Defined Function Reference Manual for more details on writing and using AcuTrace user-defined functions.

Another requirement in this example for the coupling to have effect is that the AcuSolve input file should have the temperature equation defined:

```
EQUATION {
      ...
      temperature                 = advective_diffusive
      ...
}
```

and used in an active stagger, for example:

```
TIME_SEQUENCE {
      ...
      staggers                    = { "flow",
                                  "temp_part" }
      ...
}
STAGGER( "temp_part" ) {
      ...
      equation                    = none
      staggers                    { "temperature",
                                  "particle_trace" }
      ...
}
STAGGER( "temperature" ) {
      equation                    = temperature
```

```
}
```

Our second example illustrates using value coupling as an alternative to AcuSolve species transport. In this example, particles are used to represent species 1 and, therefore, AcuTrace is effectively evolving species 1.

The number of particle components is set to 1 in the equation command:

```
EQUATION {
      ...
      number particle components = 1
      ...
}
```

The `COUPLING_FIELDS` command

```
COUPLING_FIELDS {
      ...
      species_1_type              = value
      species_1_value_type        = component
      ...
}
```

specifies that the value of the particle component couples to AcuSolve species 1. In other words, the particle component will be averaged over all the particles onto the AcuSolve mesh to provide AcuSolve with values of species 1. The components of the particles can be initialized as follows in the trace input file:

```
PARTICLE_SEED( "honey" ) {
      ...
      component_type              = constant
      constant_components         = { 1.0 }
      ...
}
PARTICLE_SEED( "tea" ) {
      ...
      component_type              = constant
      constant_components         = { 0.0 }
      ...
}
```

The component for the particles in seed group "honey" is set to 1, in other words, species 1 will equal 1 in regions containing these particles alone. Similarly, species 1 will equal 0 in regions containing particles from the second seed group alone.

In the AcuSolve input file, the `EQUATION` command specifies that there is a single species,

```
EQUATION {
      ...
      species_transport           = advective_diffusive
      num_species                 = 1
      ...
}
```

but that a particle trace stagger is to be used instead of a species stagger:

```
TIME_SEQUENCE {
      ...
```

```
      staggers                    = { "flow",
                                  "particle_trace" }
      ...
}
```

The parameter *coupling_iterations* specifies the number of AcuTrace iterations per AcuSolve time step. A single AcuTrace iteration advances all the particles from the beginning to the end of the AcuSolve time step. Ideally, the value of *coupling_iterations* should equal to the number of times the particle trace stagger is invoked during an AcuSolve time step.

For example, suppose the AcuSolve inputs include following time sequence and stagger parameters:

```
TIME_SEQUENCE {
      ...
      staggers                    = { "flow",
                                  "temp_part" }
      ...
}
STAGGER( "temp_part" ) {
      equation                    = none
      min_stagger_iterations      = 3
      max_stagger_iterations      = 3
      staggers                    = { "temperature",
                                  "particle_trace" }
}
STAGGER( "particle_trace" ) {
      equation                    = particle_trace
}
```

Here AcuSolve will invoke the particle trace stagger three times per AcuSolve time step; therefore, the value of *coupling_iterations* should be three. If the value of *coupling_iterations* is smaller than three, the two and three invocations of the particle trace stagger will have no effect on the AcuSolve solution. A value of *coupling_iterations* greater than three is effectively ignored since AcuSolve will invoke AcuTrace exactly three times.

If the AUTO_SOLUTION_STRATEGY is used by AcuSolve, AcuSolve will invoke AcuTrace only once, so the value of *coupling_iterations* is effectively one.

FINITE_MASS

Specifies the finite mass model parameters.

Type

AcuTrace Command

Syntax

`FINITE_MASS {parameters}`

Qualifier

This command has no qualifier.

Parameters

`drag_law_type` or `drag_law` (enumerated)[=standard]

Type of drag law.

zero or none	No drag force.
simple_stokes_law or simple	Simple Stokes law.
stokes_law or stokes	Stokes law.
standard_drag_law or standard	Standard drag law.

`drag_coefficient_model` or `cd_model` (enumerated)[=standard]

Type of drag coefficient model. Used only when `drag_law_type` is standard.

constant	Constant drag coefficient.
standard	Standard drag coefficient model.

`drag_coefficient` or `cd` (real) [=0.0]

Drag coefficient. Used only when `drag_law_type` is standard and `drag_coefficient_model` is constant or when `drag_law_type` is simple.

`faxen_drag_force` or `faxen_drag` (boolean) [=on]

Flag specifying if Faxen force is used. Used only when `drag_law_type` is standard or stokes.

`viscosity_model` or `mu_model` (enumerated) [=flow]

Flag specifying how to obtain the values of material viscosity used to compute the Reynolds number. Used only when `drag_law_type` is standard.

use_flow_values or flow	Obtain the values from the AcuSolve database.
constant	Use a constant value.

`constant_viscosity` *or* `mu` *(real) [=0.0]*
Material viscosity used in calculations of the Reynolds number. Used only when `drag_law_type` is standard and `viscosity_model` is constant.

`density_model` *or* `rho_model` *(enumerated)[=flow]*
Flag specifying how to obtain the values of fluid density used in force calculations.

use_flow_values or flow	Obtain the values from the AcuSolve database.
constant	Use a constant value.

`constant_density` *or* `rho_fluid` *(real) [=0.0]*
Fluid density used in force calculations. Used only when `density_model` is constant.

`pressure_force` *or* `pressure` *(boolean) [=on]*
Flag specifying if pressure force is used.

`tau_force` *or* `tau` *(boolean) [=on]*
Flag specifying if viscous stress force is used.

`virtual_mass_force` *or* `virtual_mass` *(boolean) [=on]*
Flag specifying if virtual mass force is used.

`faxen_virtual_mass_force` *or* `faxen_virtual_mass` *(boolean) [=on]*
Flag specifying if Faxen virtual mass force is used. Used only when `faxen_drag` is on.

`constant_gravity` *or* `gravity` *(array) [={0,0,0}]*
Acceleration due to gravity. This value should equal what is used in AcuSolve.

`centrifugal` *(boolean) [=on]*
Flag specifying if the centrifugal acceleration is included in rotating reference frames. This value should equal what is used in AcuSolve.

`coriolis` *(boolean) [=on]*
Flag specifying if the coriolis acceleration is included in rotating reference frames. This value should equal what is used in AcuSolve.

`angular_acceleration` *or* `angular_acc` *(boolean) [=on]*
Flag specifying if the angular acceleration is included in rotating reference frames. This value should equal what is used in AcuSolve.

`wall_type` *or* `type` *(enumerated) [=reflect]*
Wall type. Applies only to particle surfaces of type wall, slip, or symmetry not specified in a `FINITE_MASS_BOUNDARY_CONDITION` command.

reflect	Particles reflect at surface.
stop or trap	Particles stop at surface but remain active.
terminate or escape	Particles stop at surface and become inactive.

`wall_en_type` *or* `en_type` *(enumerated) [=constant]*
Type of normal coefficient of restitution. Applies only to particle surfaces of type wall, slip, or symmetry not specified in a `FINITE_MASS_BOUNDARY_CONDITION` command and only if `wall_type` is reflect.

constant	Coefficient is constant.
piecewise_linear or linear	Coefficient is a piecewise linear function of the normal component of the impact velocity.
cubic_spline or spline	Coefficient is a cubic spline function of the normal component of the impact velocity.

`constant_wall_en` *or* `wall_en` *[=1.0]*
Normal coefficient of restitution. Applies only to particle surfaces of type wall, slip, or symmetry not specified in a `FINITE_MASS_BOUNDARY_CONDITION` command and only if `wall_type` is reflect and `wall_en_type` is constant.

`wall_en_curve_fit_values` *or* `en_values` *[no default]*
A two-column array of normal-velocity/normal-coefficient-of-restitution data values. Used when `wall_en_type` is piecewise_linear or cubic_spline. Applies only to particle surfaces of type wall, slip, or symmetry not specified in a `FINITE_MASS_BOUNDARY_CONDITION` command and only if `wall_type` is reflect.

`wall_et_type` *or* `et_type` *(enumerated) [=constant]*
Type of tangential coefficient of restitution. Applies only to particle surfaces of type wall, slip, or symmetry not specified in a `FINITE_MASS_BOUNDARY_CONDITION` command and only if `wall_type` is reflect.

constant	Coefficient is constant.
piecewise_linear or linear	Coefficient is a piecewise linear function of the normal component of the impact velocity.
cubic_spline or spline	Coefficient is a cubic spline function of the normal component of the impact velocity.

`constant_wall_et` *or* `wall_et` *[=1.0]*
Tangential coefficient of restitution. Applies only to particle surfaces of type wall, slip, or symmetry not specified in a `FINITE_MASS_BOUNDARY_CONDITION` command and only if `wall_type` is reflect and `wall_en_type` is constant.

`wall_et_curve_fit_values` *or* `et_values` *[no default]*
A two-column array of normal-velocity/tangential-coefficient-of-restitution data values. Used when `wall_et_type` is piecewise_linear or cubic_spline. Applies only to particle surfaces of type wall, slip, or symmetry not specified in a `FINITE_MASS_BOUNDARY_CONDITION` command and only if `wall_type` is reflect.

Description

The `FINITE_MASS` command specifies the forces acting on the particles and the default particle/wall interaction.

The forces acting on a particle in AcuTrace are

$$\vec{F} = \vec{F}_D + \vec{F}_p + \vec{F}_\tau + \vec{F}_{VM} + \vec{F}_g \tag{6}$$

where $\vec{F}_D$ is the drag force, $\vec{F}_p$ the pressure force, $\vec{F}_\tau$ the viscous stress force, $\vec{F}_{VM}$ the virtual mass force, and $\vec{F}_g$ the gravity force. (The Basset force and other forces such as the Saffman lift force are not currently accounted for.) The parameters `drag_law_type`, `pressure_force`, `tau_force`, `virtual_mass_force`, and `constant_gravity` determine which of these are active; by default, all forces are active.

> **Note:** The default value of `constant_gravity` is the zero vector; this should be set to the same value used in the AcuSolve run providing the flow data for the trace.

The drag and the virtual mass forces can include Faxen correction terms accounting for nonuniformity effects. The inclusion of these terms is controlled by the parameters `faxen_drag_force` and `faxen_virtual_mass`. These terms are included by default. Generally, none of the force-related parameters need to be modified with the exception of `constant_gravity`.

By default (`drag_law_type` is standard and `drag_coefficient_model` is standard) the drag force (without the Faxen correction) is $\vec{F}_D = -C_D \frac{\pi}{8} \rho_f d_p^2 \left|\vec{u}_p - \vec{u}_f\right| \left(\vec{u}_p - \vec{u}_f\right)$ where ρ_f is the density of the fluid and d_p the diameter of the particle. The value of C_D the coefficient of drag, depends on the relative Reynolds number, $\mathrm{Re} = \frac{\rho_f d_p \left|\vec{u}_p - \vec{u}_f\right|}{\mu}$ where μ is the material viscosity of the fluid:

$$
\begin{aligned}
C_D &= \frac{24}{Re} && Re < 1 \\
&= \frac{24}{Re}\left(1 + \frac{Re^{\frac{2}{3}}}{6}\right) && 1 \le Re \le 1000 \\
&= .424 && Re > 1000
\end{aligned}
\tag{7}
$$

When `drag_law_type` is standard and `drag_coefficient_model` is constant, C_D is equal to the value of `drag_coefficient` regardless of the value of Re. When `drag_law_type` is stokes_law, $\vec{F}_D = -3\pi\mu d_p\left(\vec{u}_p - \vec{u}_f\right)$ (i.e. $C_D = \frac{24}{\mathrm{Re}}$) regardless of the value of Re. When `drag_law_type` is simple_stokes, $\vec{F}_D = -C_D\left(\vec{u}_p - \vec{u}_f\right)$ where C_D is equal to the value of `drag_coefficient`. When `drag_law_type` is zero, $\vec{F}_D$ is the zero vector.

The other forces are

$$
\begin{aligned}
\vec{F}_p &= V_p \nabla p \\
\vec{F}_\tau &= V_p \nabla \cdot \tau \\
\vec{F}_{VM} &= \rho_f \frac{V_p}{2}\left(\frac{D\vec{u}_f}{Dt} - \frac{D\vec{u}_p}{Dt}\right) \\
\vec{F}_g &= m_p \vec{g}
\end{aligned}
\tag{8}
$$

where V_p is the volume of the particle p the fluid pressure, τ the viscous stress tensor of the fluid, and $\frac{D\vec{u}_f}{Dt}$ the material derivative of the fluid velocity. $\vec{g}$ is given by the value of `constant_gravity`.

If *pressure_force*, *tau_force*, or *virtual_mass_force* are off, $\vec{F}_p$, $\vec{F}_\tau$, or $\vec{F}_{VM}$ respectively, are set to 0. Similarly, if *faxen_drag_force* or *faxen_virtual_mass_force* are off, the corresponding Faxen correction is set to 0.

The calculation of the drag and the virtual mass forces requires values for the fluid density and, in the case of drag, material viscosity. It is highly recommended that these values be obtained from the AcuSolve database, for example,

```
density_model        = use_flow_values
viscosity_model      = use_flow_values
```

However, these values are in the AcuSolve database only if derived quantity output is enabled in AcuSolve. This output is enabled only if the *output_frequency* parameter in the DERIVED_QUANTITY_OUTPUT command has a non-zero value, for example,

```
DERIVED_QUANTITY_OUTPUT {
...
   output_frequency    = 1000
...
}
```

Moreover, if *density_model* = use_flow_values or *viscosity_model* = use_flow_values, AcuTrace requires that the *output_frequency* parameters in the AcuSolve NODAL_OUTPUT and DERIVED_QUANTITY_OUTPUT commands have the same value, for example,

```
DERIVED_NODAL_OUTPUT {
...
   output_frequency    = 1000
...
}
DERIVED_QUANTITY_OUTPUT {
...
   output_frequency    = 1000
...
}
```

If AcuTrace is run with *density_model* or *viscosity_model* equal to use_flow_values and derived quantity output was not enabled in the AcuSolve run, AcuTrace will print an error message and stop. AcuTrace will also stop if *density_model* or *viscosity_model* equals use_flow_values and there is a mismatch in the values of *output_frequency* in AcuSolve. In either of these cases, the two options are either to rerun AcuSolve with the proper inputs or to use

```
density_model        = constant
viscosity_model      = constant
```

and suitable values of *constant_density* and *constant_viscosity*.

The default particle-wall boundary conditions and interaction parameters are also set by the FINITE_MASS command. Only the interactions at surfaces of type wall, slip, or symmetry are affected by the FINITE_MASS command. Moreover, the settings for a specific surface can be set in a FINITE_MASS_BOUNDARY_CONDITION command, in which case all the settings in the FINITE_MASS command, including default vales, are ignored for that surface.

AcuTrace allows three different types of interaction when a particle hits a wall (here "wall" refers to a surface of type wall, slip, or symmetry):

- the particle reflects off the wall (`wall_type` = reflect)
- the particle stops but continues to be actively involved in the particle trace (`wall_type` = stop)
- the particle trace terminates (`wall_type` = terminate)

When a particle reflects off a wall, the normal and tangential components of its velocity $\vec{u}_n$ and $\vec{u}_t$ (in the wall frame of reference) are given by

$$\vec{u}_n = -e_n \vec{u}_{n,i}$$
$$\vec{u}_t = e_t \vec{u}_{t,i} \qquad (9)$$

where e_n and e_t are the normal and tangential coefficients of restitution, and $\vec{u}_{n,i}$ and $\vec{u}_{t,i}$ are the incident values of the normal and tangential components of the particle velocity. e_n and e_t always lie between 0 and 1.

Each coefficient can be specified as a constant, a piecewise linear function of the magnitude of the incident normal velocity, or a cubic spline function of the magnitude of the incident normal velocity.

Note: AcuTrace clips the values of the coefficients so that they lie between 0 and 1.

In the first example below, the normal and tangential coefficients of restitution have constant values of 1.0:

```
wall_en_type          = constant
wall_et_type          = constant
wall_en               = 1.0
wall_et               = 1.0
```

In the next example, the normal and tangential coefficients of restitution have values of .1, .5, and .9 for incident normal velocity magnitudes of 1, 10, and 100, respectively. Linear interpolation is used for velocity magnitudes between 1 and 100; constant extrapolation is used for magnitudes less than 1 or greater than 100:

```
wall_en_type          = piecewise_linear
wall_et_type          = piecewise_linear
en_values             = { 1.0, 0.1 ;
                          10.0, 0.5 ;
                          100.0, 1.0 }
et_values             = { 1.0, 0.1 ;
                          10.0, 0.5 ;
                          100.0, 1.0 }
```

If instead

```
wall_en_type          = cubic_spline
wall_et_type           = cubic_spline
```

cubic spline interpolants are used for velocity magnitudes between 1 and 100.

FINITE_MASS_BOUNDARY_CONDITION

Specifies the finite mass boundary conditions and interaction parameters for a particle surface.

Type

AcuTrace Command

Syntax

`FINITE_MASS_BOUNDARY_CONDITION ("name") {parameters}`

Qualifier

User-given name.

Parameters

`particle_surface` *or* `surface` *(string) [no default]*
Name of the particle surface.

`wall_type` *or* `type` *(enumerated) [=reflect]*
Wall type.

reflect	Particles reflect at surface.
stop or trap	Particles stop at surface but remain active.
terminate or escape	Particles stop at surface and become inactive.

`wall_en_type` *or* `en_type` *(enumerated) [=constant]*
Type of normal coefficient of restitution. Applies only if `wall_type` is reflect.

constant	Coefficient is constant.
piecewise_linear or linear	Coefficient is a piecewise linear function of the normal component of the impact velocity.
cubic_spline or spline	Coefficient is a cubic spline function of the normal component of the impact velocity.

`constant_wall_en` *or* `wall_en` *[=1.0]*
Normal coefficient of restitution. Applies only if `wall_type` is reflect and `wall_en_type` is constant.

`wall_en_curve_fit_values` *or* `en_values` *[no default]*
A two-column array of normal-velocity/normal-coefficient-of-restitution data values. Used when `wall_en_type` is piecewise_linear or cubic_spline. Applies only if `wall_type` is reflect.

`wall_et_type` *or* `et_type` *(enumerated) [=constant]*
Type of tangential coefficient of restitution. Applies only if `wall_type` is reflect.

constant	Coefficient is constant.

piecewise_linear or linear	Coefficient is a piecewise linear function of the normal component of the impact velocity.
cubic_spline or spline	Coefficient is a cubic spline function of the normal component of the impact velocity.

`constant_wall_et` or `wall_et` [=1.0]
Tangential coefficient of restitution. Applies only if *`wall_type`* is reflect and *`wall_en_type`* is constant.

`wall_et_curve_fit_values` or `et_values` [no default]
A two-column array of normal-velocity/tangential-coefficient-of-restitution data values. Used when *`wall_et_type`* is piecewise_linear or cubic_spline. Applies if *`wall_type`* is reflect.

Description

The `FINITE_MASS_BOUNDARY_CONDITION` command specifies the particle/wall interaction at a given AcuSolve particle surface of type wall, slip, or symmetry. An AcuSolve particle surface is a surface that is named either in a `PARTICLE_SURFACE` command or in a `SIMPLE_BOUNDARY_CONDITION` command.

For example, if either the command

```
PARTICLE_SURFACE( "upper_wall" ) {
   type = wall
   ...
}
```

or the command

```
SIMPLE_BOUNDARY_CONDITION( "upper_wall" ) {
   type = wall
   ...
}
```

appears in the AcuSolve input file for the AcuSolve run used by AcuTrace, *`upper_wall`* is an AcuSolve particle surface of type wall and the command

```
FINITE_MASS_BOUNDARY_CONDITION( "upper_wall" ) {
   particle_surface = "upper_wall"
   ...
}
```

specifies the particle/wall boundary condition and interaction parameters at particle surface *`upper_wall`*. Moreover, the wall interaction parameters set in the `FINITE_MASS` command (*`wall_type`*, *`wall_en_type`*, *`constant_wall_en`*, *`wall_en_curve_fit_values`*, *`wall_et_type`*, *`constant_wall_et`*, and *`wall_et_curve_fit_values`*) are ignored for *`upper_wall`*. Only the parameters in the `FINITE_MASS_BOUNDARY_CONDITION` command, including any default parameter values, apply to particle surface *`upper_wall`*.

Note: In this example any unique name can be used in the `FINITE_MASS_BOUNDARY_CONDITION` command, not just *`upper_wall`*; for example,

```
FINITE_MASS_BOUNDARY_CONDITION( "finite mass upper wall parameters" ) {
```

```
   particle_surface = upper_wall
   ...
}
```

AcuTrace allows three different types of interaction when a particle hits a wall (here "wall" refers to a surface of type wall, slip, or symmetry):

- the particle reflects off the wall (`wall_type` = reflect)
- the particle stops but continues to be actively involved in the particle trace (`wall_type` = stop)
- the particle trace terminates (`wall_type` = terminate)

When a particle reflects off a wall, the normal and tangential components of its velocity, $\vec{u}_n$ and $\vec{u}_t$ (in the wall frame of reference) are given by

$$\begin{aligned} \vec{u}_n &= -e_n \vec{u}_{n,i} \\ \vec{u}_t &= e_t \vec{u}_{t,i} \end{aligned} \tag{10}$$

where e_n and e_t are the normal and tangential coefficients of restitution, and $\vec{u}_{n,i}$ and $\vec{u}_{t,i}$ are the incident values of the normal and tangential components of the particle velocity. e_n and e_t always lie between 0 and 1.

Each coefficient can be specified as a constant, a piecewise linear function of the magnitude of the incident normal velocity, or a cubic spline function of the magnitude of the incident normal velocity.

Note: AcuTrace clips the values of the coefficients so that they lie between 0 and 1.

In the first example below, the normal and tangential coefficients of restitution have constant values of 1.0:

```
wall_en_type        = constant
wall_et_type        = constant
wall_en             = 1.0
wall_et             = 1.0
```

In the next example, the normal and tangential coefficients of restitution have values of .1, .5, and .9 for incident normal velocity magnitudes of 1, 10, and 100, respectively. Linear interpolation is used for velocity magnitudes between 1 and 100; constant extrapolation is used for magnitudes less than 1 or greater than 100:

```
wall_en_type        = piecewise_linear
wall_et_type        = piecewise_linear
en_values           = { 1.0, 0.1 ;
                        10.0, 0.5 ;
                        100.0, 1.0 }
et_values           = { 1.0, 0.1 ;
                        10.0, 0.5 ;
                        100.0, 1.0 }
```

If instead

```
wall_en_type        = cubic_spline
wall_et_type         = cubic_spline
```

cubic spline interpolants are used for velocity magnitudes between 1 and 100.

Solution Strategy Commands

3

The solution strategy of a problem is specified by the commands in this chapter.

This chapter covers the following:

Either use the `AUTO_SOLUTION_STRATEGY` command, which issues all the other commands in this chapter, 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 a trace input file. All others are optional; if they are not given, their default values are used.

AUTO_SOLUTION_STRATEGY

Automatically creates a solution strategy by issuing all the other commands in this chapter.

Type

AcuTrace Command

Syntax

`AUTO_SOLUTION_STRATEGY {parameters}`

Qualifier

This command has no qualifier.

Parameters

`max_time` *(real) >=0 [=0]*
: Final time of the particle trace. The trace of an individual particle will terminate when its trace time reaches this value. The trace may terminate earlier due to other criteria. If zero, this option is ignored.

`max_segments` *(integer) >=0 [= 100000]*
: Maximum number of segments in the trace of any one particle. The trace of an individual particle will terminate when the number of segments in its trace reaches this value. The trace may terminate earlier due to other criteria. If zero, this option is ignored.

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 it is. `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 `EQUATION` command are used to determine what equations are being solved. Then the appropriate `TIME_SEQUENCE` and `STAGGER` commands are issued. These commands are also saved in a file called `problem.pa.inc`, where problem is specified by the*`problem`* parameter in the `FLOW_FIELD` command. The parameters from these commands may be overwritten afterwards by manually issuing the commands. On the next run, `problem.pa.inc` may be included (modified or not) instead of `AUTO_SOLUTION_STRATEGY`.

For example, the simplest form of this command for particle tracing without stretch is

```
EQUATION {
   particle = massless
}
AUTO_SOLUTION_STRATEGY {
}
```

For particle tracing with stretch the command is

```
EQUATION {
```

```
   particle = massless
   stretch  = standard
}
AUTO_SOLUTION_STRATEGY {
}
```

For particle tracing with stretch and user equations ener and temp the command is

```
EQUATION {
   particle = massless
   stretch  = standard
   user_equations = {ener, temp}
}
AUTO_SOLUTION_STRATEGY {
}
```

One can discover the stagger names used in the staggers parameter of the `TIME_SEQUENCE` command by examining the `problem.pa.inc` file.

TIME_SEQUENCE

Specifies the time stepping and staggering strategy.

Type

AcuTrace Command

Syntax

`TIME_SEQUENCE {parameters}`

Qualifier

This command has no qualifier.

Parameters

`max_time` *(real) >=0 [=0]*
: Final time of the particle trace. The trace of an individual particle will terminate when its trace time reaches this value. The trace may terminate earlier due to other criteria. If zero, this option is ignored.

`max_segments` *(integer) >=0 [=10000]*
: Maximum number of segments in the trace of any one particle. The trace of an individual particle will terminate when the number of segments in its trace reaches this value. The trace may terminate earlier due to other criteria. If 0, 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.

`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 [=1]*
: 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.

`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` have 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.

Description

This command specifies the time stepping and stagger iteration strategy and parameters.

A time marching method is used to advance the particle trace of each individual particle. Staggers are used to solve for a subset of the equations present in the problem; a stagger for the particle equation must always be present. With exception of a stagger for stretch equal standard, 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); the standard stretch equation is solved by a direct, non-iterative method. 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
      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 `TRACE_PARAMETERS` command.

The loop over the time steps terminates when one of the following occurs:

- User signals termination
- A fatal error occurs
- All particles have become "inactive", for example, they have:
  - reached either a trace time equal to *`max_time`* or a segment count equal to *`max_segments`*, or
  - have reached a trace time equal to the maximum value of all time cuts when time cut output is the only output requested, or
  - left the flow domain through an outflow surface, or
  - stopped at a solid boundary of the flow domain, or

- stopped at a sliding interface of the flow domain

Particles for which none of the above hold are said to be "active".

The loop over the staggers terminates when one of the following occurs:

- *max_stagger_iterations* stagger iterations are performed.
- at least *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 for particle position, stretch, and a user equation ener, one may specify:

```
TIME_SEQUENCE {
   staggers = { "particle", "stretch", "ener" }
}
STAGGER( "particle" ) {
   equation = particle
...
}
STAGGER( "stretch" ) {
   equation = stretch
...
}
STAGGER( "enth" ) {
   equation = user_equation
   user_equation = "ener"
...
}
```

Here the particle stagger is solved first for the particle equations, then the stretch stagger is solved for the stretch equation and then the ener stagger is solved for the user-defined energy equation. AcuTrace computes particle traces as a series of segments using fifth-order time-discontinuous Galerkin (TDG) with error control. Given a segment start point, a stagger for *equation* = particle computes the endpoint of a single segment of the trace of a single particle.

A stagger may be repeated multiple times. For example,

```
TIME_SEQUENCE {
   staggers = { "ener", "particle", "ener", "stretch" }
   ...
}
```

solves the ener stagger before and after the particle stagger.

The only parameters that currently affect the solution are *max_time*, *max_segments*, and *staggers*. The other parameters are reserved for future use. Currently, changing the values of the other parameters will not change the solution obtained.

There is currently no feedback between the particle and the stretch equations, nor from the user equations to either the particle or stretch equations. Generally, if there are N user equations, eqn1, ... eqnN, and these equations are not coupled, the time sequence and accompanied staggers can simply be:

```
TIME_SEQUENCE {
   staggers = { "particle", "stretch", "eqn1", ..., "eqnN" }
      }
```

```
STAGGER( "particle" ) {
   equation = particle
...
      }
STAGGER( "stretch" ) {
   equation = stretch
...
      }
STAGGER( "eqn1" ) {
   equation = user_equation
   user_equation = "eqn1"
...
      }
...
STAGGER( "eqnN" ) {
   equation = user_equation
   user_equation  = "eqnN"
 ...
}
```

If two or more user equations are coupled, it is best to have a single stagger for their coupled solution. See the description of the `STAGGER` command.

STAGGER

Specifies a stagger for the solution of an equation.

Type

AcuTrace Command

Syntax

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

Qualifier

User-given name.

Parameters

`equation` *(enumerated) [=none]*
Equation to be solved.

none	No equation solved.
particle	Particle motion equation.
stretch	Stretch equation.
user_equation	User-defined equation.

`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 for this stagger. If 0, this option is ignored.

`convergence_tolerance` *or* `conv_tol` *(real) >0 [=1.e-6]*
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, that is, the LHS is not updated. Is this its own entry or a subset of the one above?

`Nonlinear` *(boolean) [=on]*
Flag specifying whether or not to use the nonlinear solver. Ignored if `equation` is particle (`nonlinear` is always on) or `stretch` (`nonlinear` is always off.)

`user_equation` *(string) [no default]*
User equation to use if `equation` is user_equation.

`staggers` *(list) [={}]*
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 = { "particle" }
}
STAGGER( "particle" ) {
   equation = particle
}
STAGGER( "stretch" ) {
   equation = stretch
}
```

The particle stagger is directly referenced, therefore its equation is solved. The stretch stagger is not referenced, therefore its equation is not solved.

A stagger may also be indirectly referenced through another (referenced) stagger. For example,

```
TIME_SEQUENCE {
   staggers = { "particle", "reaction" }
}
STAGGER( "reaction" ) {
   equation                = none
   min_stagger_iterations = 2
   max_stagger_iterations = 2
   staggers                = { "temperature", "composition" }
}
STAGGER( "particle" ) {
   equation = particle
...
}
STAGGER( "temperature" ) {
   equation      = user_equation
   user_equation = temperature
...
}
STAGGER( "composition" ) {
   equation      = user_equation
   user_equation = composition
...
}
```

This option does provide a powerful mechanism for building a custom nonlinear solution strategy. It is currently needed only for solving coupled user equations, as in the example just above, where the user equations temperature and composition are coupled.

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 one or more `PARTICLE_SEED` commands or one or more `USER_EQUATION_INITIAL_CONDITION` commands.

Generally speaking, 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. Particle traces are computed as a series of segments using fifth-order time-discontinuous Galerkin (TDG) with error control. Given a segment start point, a particle stagger computes the endpoint of a single segment of the trace of a single particle using a nonlinear iterative solver. A stretch stagger, on the other hand, is always solved with a linear, non-iterative, direct update. User equation staggers can also be solved in this manner if appropriate, although generally a nonlinear iterative solver is used.

For all nonlinear iterative solutions, 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.

TRACE_PARAMETERS

Specifies parameters controlling the computation of the particle path.

Type

AcuTrace Command

Syntax

`TRACE_PARAMETERS {parameters}`

Qualifier

This command has no qualifier.

Parameters

`element_crossing` or *`elem_cross`* *(boolean) [=off]*
: Specifies whether or not a single particle trace segment can cross an element boundary.

`max_segment_length` or *`max_seglen`* *(real) >= 0 [=0]*
: Maximum length of a segment. If 0, this value imposes no maximum.

`max_segment_coordinate_increment` or *`max_coord_inc`* *(real) >=0 [=0.5]*
: Maximum local coordinate segment length allowed (segment length as a fraction of the size of the element.) If 0, this value imposes no maximum.

`max_segment_time_increment` or *`max_dt`* *(real) >=0 [=0.0]*
: Maximum time step per segment. If 0, this value imposes no maximum.

`max_turning_angle` or *`max_angle`* *(real) >=0 [=15]*
: Maximum turn angle in degrees of the particle velocity from the previous segment. If 0, this value imposes no maximum.

`turbulence_trace` or *`turb`* *(boolean) [=off]*
: Specifies whether or not the particle integration accounts for the effects of turbulence.

Description

The `TRACE_PARAMETERS` command specifies parameters controlling the computation of the particle path.

Particle traces are computed by AcuTrace as a series of segments using fifth-order time-discontinuous Galerkin (TDG) with error control. In the absence of any other restrictions, the end point of single particle segment can be anywhere in the element containing the segment start point or anywhere in any of the neighboring elements sharing a face with the element of origin. There are four such neighbors for tetrahedral elements, five for pyramidal and wedge elements, and six for hexahedral elements. This restriction in effect imposes a time increment restriction on the particle advance similar to a CFL=1 condition on the flow solver. *`max_segment_time_increment`* is the maximum value this time increment can have. The values of the *`max_segment_length`*, *`max_segment_coordinate_increment`*, and *`max_turning_angle`* parameters further restrict the time increment of a single particle segment.

A value of off for *`element_crossing`* further restricts the time step by forcing the particle segment endpoint to be in the element, or on the element face of the element, containing the segment starting

point. The particle trace is computed more accurately if `element_crossing` is off, but the computation time will be about 50 percent greater than if `element_crossing` is on.

If `turbulence_trace` equals on, the effect of turbulence is modeled by randomly perturbing the particle velocity as a function of the local eddy viscosity. The result is a statistically correct lateral diffusion of the particle paths from the path given by the unperturbed flow velocity field alone. This process is repeatable, that is, successive runs of AcuTrace will yield the same trajectories.

Particle Data Commands

4

The particle data commands are outlined in this chapter.

This chapter covers the following:

There needs to be at least one `PARTICLE_SEED` command. As many `PARTICLE_SEED` commands as necessary are allowed. Each unique `PARTICLE_SEED` command defines a distinct set of particles. A `USER_EQUATION_INITIAL_CONDITION` command is allowed only for pairings of particle seed groups and user equations. Each of the pairings defined by the particle seed commands and the user equations specified by the *`user_equations`* parameter in the `EQUATION` command requires a corresponding `USER_EQUATION_INITIAL_CONDITION` command. If no equations are specified by the *`user_equations`* parameter in the `EQUATION` command, no `USER_EQUATION_INITIAL_CONDITION` commands are needed.

PARTICLE_SEED

Specifies the initial conditions for a set of particles.

Type

AcuTrace Command

Syntax

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

Qualifier

User-given name.

Parameters

`marker` (integer) [=0]
A marker value assigned to all particles in this set.

`seed_ids_type` or `id_type` (enumerated) [=user]
Type of seed position specification.

user	Use the seed ids provided by you if available, otherwise, use global seed ids.
global	Use global seed ids.
local	Use local seed ids.

`coordinates_type` or `crd_type` (enumerated) [=per_seed]
Type of seed position specification.

per_seed or seeds	Use a list of seed ids and positions (id, x, y, z).
volume_random or vol_random	Seeds randomly distributed in an element set.
volume_uniform or vol_uniform	Seeds uniformly distributed in an element set.
surface_random or surf_random	Seeds randomly distributed on a surface.
surface_uniform or surf_uniform	Seeds uniformly distributed on a surface.
surface_flux_weighted or surf_flux_weighted	Seeds randomly distributed on a surface in a flux weighted manner.
region_random or reg_random	Seeds randomly distributed in a rectangular region.

region_uniform or reg_uniform	Seeds uniformly distributed in a rectangular region.

`seed_coordinates` or `coord` (array) [no default]
List of seed ids and positions. Each row contains an integer seed id followed by the three real values, x,y,z, of the seed position. Used only if `coordinates_type` is per_seed.

`number_of_seeds` or `num_seeds` (integer) [=1]
Number of seeds. Ignored if `coordinates_type` is per_seed.

`region_bounding_box` or `region` (array) [={0,0,0;1,1,1}]
Lower and upper corners of the region. Used only if `coordinates_type` is region_random or region_uniform.

`particle_surface` or `surface` (string) [no default]
Name of the particle surface. Used only if `coordinates_type` is surface_random, surface_uniform, or surface_flux_weighted.

`particle_surface_offset` or `surface_offset` (real) [=0]
Offset of seed position from the particle surface as a fraction of the element length. Used only if `coordinates_type` is surface_random, surface_uniform, or surface_flux_weighted.

`element_set` or `elem_set` (string) [no default]
Name of the element set. Used only if `coordinates_type` is volume_random or volume_uniform.

`density_type` (enumerated) [=constant]
Type of seed density specification.

constant	All seeds assigned the same density.
per_seed	Use a list of densities, one density per seed.
random	Seed densities are randomly assigned.

`seed_densities` or `densities` (array) [no default]
List of seed densities, one per seed. Used only if `density_type` is per_seed.

`constant_density` (real) >0.0 [=1.0]
Density assigned to all seeds. Used only if `density_type` is constant.

`density_random_bounds` or `density_rand_bounds` (array) [no default]
Upper and lower bounds used to assign the random density initial conditions. Used only if `density_type` is random.

radius_type (enumerated) [=constant]
Type of seed radius specification.

constant	All seeds assigned the same radius.
per_seed	Use a list of radii, one density per seed.
random	Seed radii are randomly assigned.

seed_radii or radii (array) (no default)
List of seed radii, one per seed. Used only if `radius_type` is `per_seed`.

constant_radius real >0.0 (=1.0)
Radius assigned to all seeds. Used only if `radius_type` is `constant`.

radius_random_bounds or radius_rand_bounds (array) (no default)
Upper and lower bounds used to assign the random density initial conditions. Used only if `density_type` is random.

`velocity_type` *(enumerated) [=use_flow_velocity]*
Type of seed velocity specification.

use_flow_velocity or use_flow	Use the flow velocity at the seed location.
constant	All seeds assigned the same velocity.
zero	All seeds assigned a zero velocity.
per_seed	Use a list of velocities, one velocity per seed.
random	Seed velocities are randomly assigned.

`particle_velocity_multiplier` *(real) >=0.0, <=1.0 [=1.0]*
When `velocity_type` equals use_flow_velocity, the particle velocity is set to the flow velocity at the seed location multiplied by `particle_velocity_multiplier`. Used only if `velocity_type` is use_flow_velocity.

`seed_velocity` *or* `seed_vel` *(array) [no default]*
List of seed velocities, one per seed. Used only if `velocity_type` is per_seed.

`constant_velocity` *or* `vel` *(array) [={0,0,0}]*
Velocity assigned to all seeds. Used only if `velocity_type` is constant.

`velocity_random_bounds` *or* `vel_rand_bounds` *(array) [no default]*
Upper and lower bounds used to assign the random velocity initial conditions. Used only if `velocity_type` is random.

`time_type` *(enumerated) [=zero]*
Type of seed time specification.

zero	All seeds assigned a time of 0.
constant	All seeds assigned an identical time.
per_seed	Use a list of times, one time per seed.
emission_times	Use a list of times. A copy of all the seeds are emitted at each time.

`seed_time` *or* `time` *(real) [=0]*
Time assigned to all seeds. Used only if `time_type` is constant.

`seed_times` *or* `times` *(array) [no default]*
List of seed times, one per seed. Used only if `time_type` is per_seed.

`emission_time_type` *(enumerated) [=series]*
Type of emission seed time specification. The number of particles in the simulation will be the number of seeds times the number of emission times.

time_series or series	Use a list of emission times.
time_interval or interval	Specify times by start and stop times and a time interval.

`emission_times` *(array) [no default]*
List of emission times. Used only if `emission_time_type` is time_series.

`emission_start_time` *or* `etime_start` *(real) [=0]*
First emission time. Used only if `emission_time_type` is interval.

`emission_stop_time` *or* `etime_stop` *(real) [=0]*
Last emission time. Used only if `emission_time_type` is interval.

`emission_time_interval` *or* `etime_interval` *(real) [=0]*
Time interval between successive emission times. Used only if `emission_time_type` is interval. When `emission_time_type` is interval, the emission times always include `emission_start_time` and `emission_stop_time` regardless of the value of `emission_time_interval`.

`stretch_type` *(enumerated) [=constant]*
Type of seed stretch specification.

constant	All seeds assigned the same stretch vector.
random	Stretch vectors are randomly assigned.
per_seed	Use a list of stretch vectors.

`constant_stretch` *(array) [={1,0,0}]*
Stretch vector assigned to all seeds. Used only if `stretch_type` is constant.

`seed_stretch` *(array) [no default}*
List of stretch vectors, one per seed. Used only if `stretch_type` is per_seed.

`random_stretch_length` *(real) [=1]*
Value of random stretch length. Used only if `stretch_type` is random.

`component_type` *(enumerated) [=none]*
Type of seed component specification.

none	No components are assigned.
constant	All seeds assigned the same component vector.
random	Component vectors are randomly assigned.
per_seed	Use a list of stretch vectors.
distributed	All components of each seed have value 0 excepting a randomly selected one with a value of 1.

`constant_components` *(array) [no default]*
: If `component_type` is constant, the component vector assigned to all seeds. If `component_type` is distributed, the array is used to construct bins, one for each component. Using these bins, seeds are randomly assigned one component with a value of 1.

`seed_components` *or* `seed_comp` *(array) [no default]*
: List of components vectors, one per seed. Used only if `component_type` is per_seed.

`component_random_bounds` *or* `comp_rand_bounds` *(array) [no default]*
: Upper and lower bounds used to assign the random component initial conditions. The array should have two rows, one for each bound. The number of columns in each row equal must the number of particle components. Use only if `component_type` is random.

`turbulence_random_seed_type` *(enumerated) [=constant]*
: Type of turbulence random seed specification.

constant	All particle seeds assigned the same random seed.
per_seed	Used a list of random seeds.

`constant_turbulence_random_seed` *or* `turb_seed` *(integer) [=1]*
: Turbulence random seed assigned to all particle seeds. The random seed actually used for each particle is this value plus the internal seed id. Used only if `turbulence_random_seed_type` is constant.

`turbulence_random_seeds` *(array) [no default]*
: List of turbulence random seeds, one per seed. The random seed actually used for the particles are these values added to the internal seed ids. Used only if `turbulence_random_seed_type` is per_seed.

Description

The `PARTICLE_SEED` command defines a set of particles and initial conditions for those particles. The initial conditions that can be set by the `PARTICLE_SEED` command are

- particle ids
- particle set marker
- position (coordinates)
- radius
- density
- velocity
- time
- stretch vector
- turbulence random seed

User equation initial conditions for a set of particles are defined by the `USER_EQUATION_INITIAL_CONDITION` command.

There needs to be at least one `PARTICLE_SEED` command in a trace input file, but as many `PARTICLE_SEED` commands as necessary are allowed. A unique set of particles is defined for each

unique qualifier name used in a `PARTICLE_SEED` command. For example, two separate sets of particles, four particles in all, are defined if the following two commands appear in the input file:

```
PARTICLE_SEED( "seed_group_1" ) {
   coordinates_type = per_seed
   seed_coordinates = { 11, -0.05, 0.1, 0.0;
                        12, -0.05, 0.2, 0.0
                      }
}
PARTICLE_SEED( "seed_group_2" ) {
   coordinates_type = per_seed
   seed_coordinates = { 21, -0.05, 0.1, 0.0;
                        22, -0.05, 0.2, 0.0
                      }
}
```

Particle coordinates are assigned explicitly only if *coordinates_type* equals per_seed, in which case the number of seeds in the particle set equals the number of particle id and positions provided by the *seed_coordinates* parameter. In the example above, there are two particles in each particle set. If the coordinates of a seed fall outside the AcuSolve flow domain, the seed is ignored and does not participate in the particle trace. Particle ids are assigned explicitly only if *coordinates_type* equals per_seed and *seed_ids_type* equals user.

- If *coordinates_type* equals per_seed but *seed_ids_type* does not equal user, the seed id values in the *seed_coordinates* parameter are ignored, and the particle ids are automatically assigned based on the value of *seed_ids_type*.
- If *coordinates_type* does not equal per_seed and *seed_ids_type* does equal user, *seed_ids_type* is reset to global.
- If *seed_ids_type* equals global, the particles are assigned a unique id between 1 and the total number of seeds in all the particle seed sets.
- If *seed_ids_type* equals local, the particles are assigned an id between 1 and the number of seeds specified in the current `PARTICLE_SEED` command.

In the example above, the four seeds are assigned seed ids of 11, 12, 21, and 22. If the parameter *seed_ids_type* is set to local in both particle seed commands, the four seeds will have ids of 1, 2, 1, and 2. If the parameter *seed_ids_type* is set to global in both particle seed commands, the four seeds will have ids of 1, 2, 3, and 4. The parameter *seed_ids_type* does not need to have the same value in all the seed groups. For example, if *seed_ids_type* is set to user in the first seed group and global in the second, the four seeds will have ids of 11, 12, 3, and 4.

If *coordinates_type* does not equal per_seed, the number of seeds is specified, and the particle coordinates are automatically assigned based on other command parameters:

- If *fluid_elements* is the name of an AcuSolve element set, for example, the command

  ```
  ELEMENT_SET( "fluid_elements" ) {
     ...
  }
  ```

appears in the AcuSolve input file for the AcuSolve run used by AcuTrace, the following `PARTICLE_SEED` command initializes a set of particles with initial positions randomly distributed throughout the AcuSolve element set *fluid_elements*:

```
PARTICLE_SEED( "seeds" ) {
```

```
   ...
   coordinates_type = volume_random
   element_set      = "fluid_elements"
   number_of_seeds  = 20
   ...
}
```

- If inflow is the name of an AcuSolve particle surface, for example, if either the command

```
PARTICLE_SURFACE( "inflow" ) {
   ...
}
```

or the command

```
SIMPLE_BOUNDARY_CONDITION( "inflow" ) {
   ...
}
```

appears in the AcuSolve input file for the AcuSolve run used by AcuTrace, the following `PARTICLE_SEED` command initializes a set of particles with initial positions randomly distributed on the AcuSolve particle surface inflow:

```
PARTICLE_SEED( "seeds" ) {
   ...
   coordinates_type = surface_random
   particle_surface = "inflow"
   number_of_seeds  = 20
   ...
}
```

while the following `PARTICLE_SEED` command initializes a set of particles with initial positions randomly distributed in a mass flux weighted manner on the AcuSolve particle surface inflow:

```
PARTICLE_SEED( "seeds" ) {
   ...
   coordinates_type = surface_flux_weighted
   particle_surface = "inflow"
   number_of_seeds  = 20
   ...
}
```

- A set of particles with positions randomly distributed in a rectangular region is initialized by the command

```
PARTICLE_SEED( "seeds" ) {
  ...
   coordinates_type    = region_random
   region_bounding_box = {0,1,2;3,4,5}
   number_of_seeds     = 20
   ...
}
```

When *`coordinates_type`* equals region_random, particle positions that fall outside the flow domain are discarded. If this happens, the number of particles initialized is less than the value of *`number_of_seeds`*.

- Values of *coordinates_type* equal to volume_uniform, surface_uniform, and region_uniform currently result in the same seeding as volume_random, surface_uniform, and region_random, respectively.

For *coordinates_type* of surface_uniform, surface_random, and surface_flux_weighted, the parameter *particle_surface_offset* defines an offset for the initial particle positions from the surface as a fraction of the representative element length. In other words, when *particle_surface_offset* is 0, the positions are directly on the surface whereas for a non-zero value they are moved into the element adjacent to the surface. This is particularly useful if the surface is a no-slip wall. In this case, if the *particle_surface_offset* is 0, the particles will never move since the velocity at the wall is 0 for all time. For a non-zero value, the particles do move because they are put into the fluid next to the wall.

Particle masses are not input directly. Instead, particle densities and radii are specified. The particle masses are then initialized to the particle density times the particle volume. The particle radius and density inputs affect the particle trace only if the finite mass particle equation is used. If the massless particle equation is used, the particle densities and radii are still assigned to the particles but have no effect on the particle trace.

The initial particle densities can be assigned in one of three ways:

- as a constant over the seed set, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   density_type      = constant
   constant_density  = 1.0
   ...
}
```

- per seed, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   density_type   = per_seed
   seed_densities = { 0.9, 0.7, 1.2, 1.3, 1.05 }
   ...
}
```

- randomly, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   desnity_type          = random
   density_random_bounds = {0.5, 1.5 }
   ...
}
```

When *density_type* equals random, the density of each seed is randomly assigned a value between the values of *density_random_bounds*, here, between 0.5 and 1.5.

The initial values of the particle radii can also be assigned in one of three ways:

- as constant over the seed set, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   radius_type      = constant
```

```
   constant_radius  = 0.0001
   ...
}
```

- per seed, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   radius_type  = per_seed
   seed_radii   = { 1.1e-4, 9.0e-5, 1.2e-4, 0.85e-4, 1.01e-4 }
   ...
}
```

- randomly, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   density_type         = random
   desity_random_bounds = { 9.0e-5, 1.1e-4 }
   ...
}
```

When *radius_type* equals random, the radius of each seed is randomly assigned a value between the values of *radius_random_bounds*, here, between 9.0e-5 and 1.1e-4.

> **Note:** The radius and density of a particle is constant in time. The mass of a particle is therefore constant as well.

Particle velocity initialization is relevant only when the finite mass particle equation is used. If the massless particle equation is used, particle velocity inputs are ignored because the particle velocity always equals the flow velocity.

Particle velocities can be initialized in one of six ways:

- to zero for all seeds in the seed set, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   velocity_type   = zero
   ...
}
```

- to the flow velocity at the seed location, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   velocity_type                = use_flow_velocity
   particle_velocity_multiplier = 1.0
   ...
}
```

- to a constant times the flow velocity, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   velocity_type                = use_flow_velocity
   particle_velocity_multiplier = 0.9
```

```
   ...
}
```

- as a constant over the seed set, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   velocity_type             = constant
   constant_velocity         = { 0.7, 0.9, 1.1 }
   ...
}
```

- per seed, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   velocity_type  = per_seed
   seed_velocity  = { 1.0,  1.1,  0.8 ;
                      0.85, 0.95, 1.3 ;
                      1.35, 1.1,  0.65 ;
                      0.7,  0.8,  1.4
                    }
   ...
}
```

- randomly, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   velocity_type           = random
   velocity_random_bounds  = { 0.65,  0.75,  0.6 ;
                               1.4, 1.2, 1.5
                             }
   ...
}
```

When *velocity_type* equals random, each component of the velocity of a seed is randomly assigned a value between the values of *velocity_random_bounds*. In the example above, the x-component is randomly assigned a value between 0.65 and 1.4, the y-component between .75 and 1.2, and the z-component between .6 and 1.5.

Initial particle times can be set in one of four ways:

- An initial time of 0 is assigned to all the particles in a particle set if *time_type* is zero.
- To assign the same non-zero constant time to all the particles, use, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   time_type = constant
   time      = 20.0
   ...
}
```

- To assign a unique time for each particle, use, for example:

```
PARTICLE_SEED( "seeds" ) {
   ...
   time_type = per_seed
```

```
   times      = Read( "seed_times" )
   ...
}
```

The number of values specified by the *times* must equal the number of seeds in the set.

- To assign a series of emission times to all the particles in the current set, use, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   time_type          = emission_times
   emission_time_type = time_series
   emission_times     = {
                        0.0 ;
                        1.0;
                        2.0;
                        3.0
                        }
   ...
}
```

The total number of particles initialized when *time_type* equals emission_times equals the number of seeds times the number of emission times. For example, for the command

```
PARTICLE_SEED( "seeds" ) {
   coordinates_type = per_seed
   seed_coordinates = {
                      1, -0.05, 0.1, 0.0;
                      2, -0.05, 0.2, 0.0
                      }
   ...
   time_type                   = emission_times
   emission_time_type          = time_interval
   emission_start_time         = 0.0
   emission_stop_time          = 3.0
   emission_time_interval      = 1.0
   ...
}
```

there are eight particles in all. There are four particles at (-0.05, 0.1, 0.0) with initial times of 0.0, 1.0, 2.0, and 3.0, and four particles at (-0.05, 0.2, 0.0) with the same four initial times.

Initial stretch vectors can be assigned in one of three ways:

- a constant over the seed set, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   stretch_type     = constant
   constant_stretch = { 1.0, 0.0, 1.0 }
   ...
}
```

- per seed, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   stretch_type = per_seed
   seed_stretch = { 1, 0, 0; 0, 1, 0; 1, 1, 1; 0, 0, 1 }
   ...
```

```
}
```

- randomly, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   stretch_type          = random
   random_stretch_length = 1.0
   ...
}
```

When *stretch_type* equals per_seed, the number of stretch vectors (four in the example shown) must equal the number of seeds in the seed set. When *stretch_type* equals random, each component of each stretch vector is randomly assigned a value between 0 and the value of *random_stretch_length*.

The initial values of the particle component vectors can be assigned in one of four ways. In the following example, it is assumed that *number_particle_components* is set to five in the EQUATION command.

- a constant over the seed set, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   component_type      = constant
   constant_components = { 0.0, 0.0, 1.0, 1.0, 0.0 }
   ...
}
```

- per seed, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   component_type  = per_seed
   seed_components = { 1, 0, 0, 0, 0; 0, 1, 0, 0, 0; 0, 0, 1, 0, 0 }
   ...
}
```

- randomly, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   component_type          = random
   component_random_bounds = {0.0, 2.0, 4.0, 3.0, 4.0 ;
                              1.0, 3.0, 5.0, 4.0, 5.0
                              }
   ...
}
```

- randomly distributed over the seeds, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   component_type      = distributed
   constant_components = { 1.0, 2.0, 1.0, 0.5, 1.5 }
   ...
}
```

When *component_type* equals constant, all seeds in the seed set are assigned the same component vector, (0.0, 0.0, 1.0, 1.0, 0.0) in the example shown. When *component_type* equals per_seed, the

number of initial component vectors, three in the example shown, must equal the number of seeds in the seed set. When *component_type* equals random, each component of each seed is randomly assigned a value between the corresponding values of *component_random_bounds*. Here, the first components all lie between 0.0 and 1.0, the second between 2.0 and 3.0, and so on.

When *component_type* equals distributed, the values of *constant_components* represent bin widths. For the example shown, five bins are constructed: 0.0 to 1.0; 1.0 to 3.0; 3.0 to 4.0; 4.0 to 4.5; 4.5 to 6. For each seed, a random number is drawn between the lower bound of the lowest bin and the upper bound of the highest bin. That number is then used to determine a bin number. The component for that bin is set to 1; the components for the other bins are set to 0. For the example show, random numbers are drawn between zero and six. If the random draw for the first seed is 4.1, the first seed is assigned a component vector of (0.0, 0.0, 0.0, 1.0, 0.0) because 4.1 falls in the fourth bin, that is, it is between 4.0 and 4.5. If the draw for the second seed is 2.7, the second seed is assigned a component vector of (0.0, 1.0, 0.0, 0.0, 0.0) because 2.7 falls in the second bin, that is, it is between 1.0 and 3.0. This process is repeated for all the seeds in the seed group.

> **Note:** The particle component vectors retain their initial values throughout the particle trace, that is, they are constant in time.

The initial turbulent seed can be assigned in one of two ways:

- a constant over the seed set, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   turbulence_random_seed_type = constant
   turbulence_random_seed      = 1.0
   ...
}
```

- per seed, for example,

```
PARTICLE_SEED( "seeds" ) {
   ...
   turbulence_random_seed_type = per_seed
   turbulence_random_seeds     = { 1; 3; 99; -3; 17 }
   ...
}
```

When *turbulence_random_seed_type* equals per_seed, the number of turbulent seeds, five in the example shown, must equal the number of seeds in the seed set.

USER_EQUATION_INITIAL_CONDITION

Specifies the initial user equation values for a particle set and a single user equation.

Type

AcuTrace Command

Syntax

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

Qualifier

User-given name.

Parameters

`particle_seed` or *`seed`* *(string) [no default]*
: The name of the particle set. Must be a qualifier used in one of the `PARTICLE_SEED` commands.

`user_equation` *(string) [no default]*
: The name of the user equation. Must be a qualifier used in one of the `USER_EQUATION` commands.

`type` *(enumerated) [=constant]*
: The type of initial condition specification.

constant	Assign the same initial condition to all seeds.
per_seed	Use a list of initial conditions.
random	Randomly assign initial condition.

`constant_values` *(array) [no default]*
: User equation initial condition assigned to all seeds. The size of the array must equal the number of variables in the user equation. Use only if *`type`* is constant.

`seed_values` *(array) [no default]*
: List of initial condition vectors, one per seed. The number of columns in each row must the number of variables in the user equation. The number of rows must equal the number of seeds in the seed group. Use only if *`type`* is per_seed.

`random_bounds` *(array) [no default]*
: Upper and lower bounds used to assign the random initial conditions. The array should have two rows, one for each bound. The number of columns in each row equal must the number of variables in the user equation. Use only if *`type`* is random.

Description

Initial conditions must be assigned for every user equation that appears in the *`user_equations`* parameter of the `EQUATION` command. The initial conditions for each such equation are assigned per seed set. The `USER_EQUATION_INITIAL_CONDITION` command sets the initial conditions for the pairing of a single user equation with a single particle seed set. Every pairing of a user equation appearing in the *`user_equations`* parameter and a particle seed set must have an associated

`USER_EQUATION_INITIAL_CONDITION` command. For example, if there are two user equations, *`user1`* and *`user2`*, and three seed sets, seed1, seed2, and seed3, for example,

```
EQUATION {
   ...
   user_equations = {user1, user2}
}
PARTICLE_SEED( "seed1" ) {
   ...
}
PARTICLE_SEED( "seed2" ) {
   ...
}
PARTICLE_SEED( "seed3" ) {
   ...
}
```

there must be six `USER_EQUATION_INITIAL_CONDITION` commands:

```
USER_EQUATION_INITIAL_CONDITION( "seed1_user1" ) {
   particle_seed = "seed1"
   user_equation = "user1"
   ...
}
USER_EQUATION_INITIAL_CONDITION( "seed1_user2" ) {
   particle_seed = "seed1"
   user_equation = "user2"
   ...
}
USER_EQUATION_INITIAL_CONDITION( "seed2_user1" ) {
   particle_seed = "seed2"
   user_equation = "user1"
   ...
}
USER_EQUATION_INITIAL_CONDITION( "seed2_user2" ) {
   particle_seed = "seed2"
   user_equation = "user2"
   ...
}
USER_EQUATION_INITIAL_CONDITION( "seed3_user1" ) {
   particle_seed = "seed3"
   user_equation = "user1"
   ...
}
USER_EQUATION_INITIAL_CONDITION( "seed3_user2" ) {
   particle_seed = "seed3"
   user_equation = "user2"
   ...
}
```

The user given names, seed1_user1, and so on, do not need to follow the convention used in this example, as long as each seed set and user equation pairing is assigned a unique name.

Initial conditions can be assigned in one of three ways. To illustrate, assume seed set seed1 has three seeds and user function *`user1`* has four variables:

- a constant over the seed set, for example,

```
USER_EQUATION_INITIAL_CONDITION( "seed1_user1" ) {
   particle_seed   = "seed1"
   user_equation   = "user1"
```

```
   type             = constant
   constant_values = { 1.0, 2.0, 3.0, 4.0 }
}
```

- per seed, for example,

```
USER_EQUATION_INITIAL_CONDITION( "seed1_user1" ) {
   particle_seed = "seed1"
   user_equation = "user1"
   type          = per_seed
   constant_values  = {    1.0, 2.0, 3.0, 4.0 ;
                        1.1, 2.3, 2.9, 3.7 ;
                        1.2, 2.4, 3.1, 4.2
                     }
}
```

- randomly, for example,

```
USER_EQUATION_INITIAL_CONDITION( "seed1_user1" ) {
   particle_seed = "seed1"
   user_equation = "user1"
   type          = random
   random_bounds = {    1.0, 1.9, 2.8, 3.5 ;
                        2.0, 3.5, 4.1, 4.7
                     }
}
```

When *type* equals per_seed, the number of initial value vectors, three in the example shown, must equal the number of seeds in the seed set. When *type* equals random, each component of each user equation value is randomly assigned a value between the corresponding bounds. Here, the first components all lie between 1.0 and 2.0, the second between 1.9 and 3.5, and so on.

A USER_EQUATION_INITIAL_CONDITION command is allowed but not required for pairings of particle seed groups with user equations that do not appear in the *user_equations* parameter of the EQUATION command. For example, in the following:

```
EQUATION {
   ...
   user_equations = {user1}
}
PARTICLE_SEED( "seed1" ) {
   ...
}
USER_EQUATION( "user1" ) {
   ...
}
USER_EQUATION( "user2" ) {
   ...
}
```

the command

```
USER_EQUATION_INITIAL_CONDITION( "seed1_user2" ) {
   particle_seed = "seed1"
   user_equation = "user2"
   ...
}
```

allowed but not required. In fact, since *user2* is not listed by the *user_equations* parameter, the command USER_EQUATION_INITIAL_CONDITION("seed1_user2") command has no effect.

A pairing of a seed set and a user equation can only appear in one USER_EQUATION_INITIAL_CONDITION command. If the same pairing of seed set and user equation appears in more than one USER_EQUATION_INITIAL_CONDITION command, an error will be reported, for example, if the following two commands are used:

```
USER_EQUATION_INITIAL_CONDITION( "seed1_user2_a" ) {
   particle_seed = "seed1"
   user_equation = "user2"
   ...
}
USER_EQUATION_INITIAL_CONDITION( "seed1_user2_b" ) {
   particle_seed = "seed1"
   user_equation = "user2"
   ...
}
```

Output Commands

5

The output of a problem is specified by the commands in this chapter.

This chapter covers the following:

All four commands have a boolean parameter *active* controlling whether the output type (trace, time cut, Poincare, interpolate) is active or not. At least one output type must be active, but otherwise any number of output types can be active. All AcuTrace output files are written to the directory given by the *working_directory* parameter of the FLOW_FIELD command. By default, this directory is ACUSIM.DIR.

The file format for output is specified by a format parameter. The available formats are

- bin_rec
- binary
- ascii

The bin_rec format is the default. The other two formats are provided for backward compatibility with an older version of AcuTrace. Trace, timecut, and Poincare output written in the bin_rec format can be converted to a number of useful formats (Fieldview, EnSight, and so on) by the AcuTransTrace command. It is recommended that the bin_rec format be used when any of these three output types are active. INTERPOLATE_OUTPUT is a legacy output type provided for backward capability. The file format is specified by the AcuRunTrace file_format command option.

TRACE_OUTPUT

Specifies parameters for the output of path line segment endpoints.

Type

AcuTrace Command

Syntax

`TRACE_OUTPUT {parameters}`

Qualifier

This command has no qualifier.

Parameters

`active` *(boolean) [=off]*
: Flag specifying if trace output is active.

`output_frequency` *(integer) [=1]*
: Segment frequency at which to output the endpoints.

`flow_state_data` *or* *`flow_state`* *(boolean) [=off]*
: Flag specifying whether to include the flow state in the output.

`flow_gradient_data` *or* *`flow_gradient`* *(boolean) [=off]*
: Flag specifying whether to include gradients of the flow state in the output.

Description

Particle traces are computed by AcuTrace as a series of segments. Trace output records the endpoints of all the segments. The endpoints for all particles are recorded. The parameter *`output_frequency`* specifies how often to record the endpoints. For example,

```
TRACE_OUTPUT {
   active            = on
   output_frequency  = 5
   ...
}
```

specifies that every fifth segment endpoint is recorded.

The *`active`*, *`flow_state_data`*, and *`flow_gradient_data`* parameters control what information is recorded. No trace output is recorded unless *`active`* = on. By default, all particle outputs and no flow outputs are recorded when trace output is active.

Particle values are the state of the particle, that is, its position, time, particle velocity, and so on. The flow outputs are found by interpolating the AcuSolve flow field to the current position and time of the particle.

Particle outputs always include

- Seed ID

- Coordinates
- Time
- Particle velocity
- Element ID
- Element set ID
- Marker
- Particle velocity magnitude
- Trace length

They also include

- Particle mass
- Particle density
- Particle radius

if the parameter *particle* equals finite_mass model in the EQUATION command,

- Component values

if the parameter *number_particle_components* in the EQUATION command is non-zero,

- User equation values

if one or more user equations are specified in the *user_equations* parameter of the EQUATION command.

- Turbulence random seed

if the parameter *turbulent_trace* equals on in the TRACE_PARAMETERS command and

- Stretch vector
- Stretch magnitude
- Log stretch magnitude
- Stretch rate magnitude

if the parameter *stretch* does not equal none in the EQUATION command.

Flow state values always include

- Flow velocity
- Flow pressure
- Flow velocity magnitude
- Flow strain rate magnitude

They also include

- Flow temperature
- Flow species
- Flow eddy viscosity

if these are available in the AcuSolve database.

Flow gradient values currently include only

- Gradient of flow velocity

> **Note:** For the massless particle motion equation, the particle and the flow velocity fields are identical except possibly for particle positions in elements next to a wall.

TIME_CUT_OUTPUT

Specifies parameters for the output of time cuts.

Type

AcuTrace Command

Syntax

`TIME_CUT_OUTPUT {parameters}`

Qualifier

This command has no qualifier.

Parameters

`active` *(boolean) [=off]*
Flag specifying if time cut output is active.

`time_cut_type` *or* `tcut_type` *(enumerated) [=time_series]*
Type of time cut specification.

time_series or series	Use a user-defined series of times.
time_interval	Use end points and an interval.

`time_cuts` *or* `tcuts` *(array) [={}]*
List of time values. Used only if `time_cut_type` is time_series.

`time_cut_start_time` *or* `tcut_start` *(real) [=0]*
Initial time cut value. Used only if `time_cut_type` is time_interval.

`time_cut_stop_time` *or* `tcut_stop` *(real) [=0]*
Final time cut value. Used only if `time_cut_type` is time_interval.

`time_cut_interval` *or* `tcut_interval` *(real) [=0]*
Interval between successive time cuts. Used only if `time_cut_type` is time_interval.

`flow_state_data` *or* `flow_state` *(boolean) [=off]*
Flag specifying whether to include the flow state in the output.

`flow_gradient_data` *or* `flow_gradient` *(boolean) [=off]*
Flag specifying whether to include gradients of the flow state in the output.

Description

Particle traces are computed by AcuTrace as a series of segments. Time cut output is recorded for all active particles at the time cuts. The particle path is interpolated in time between the segment endpoints steps on either side of the time cut.

Time cuts can be specified either by a list or by range and an interval. For example, the command

```
TIME_CUT_OUTPUT {
   active               = on
```

```
   time_cut_type        = time_series
   time_cuts            = { 1.1, 3.2, 7.3, 13.4 }
   ...
}
```

specifies time cut output at times of 1.1, 3.2, 7.3, and 13.4. The command

```
TIME_CUT_OUTPUT {
   active                = on
   time_cut_type         = time_interval
   time_cut_start_time   = 0.
   time_cut_stop_time    = 1000.
   time_cut_interval     = 10.
   ...
}
```

specifies that time cut output be written every 10 time units beginning at time zero and ending at time 1000. When *time_cut_type* equals time_interval, time cut values of *time_cut_start_time* and *time_cut_stop_time* are always used regardless of the value of *time_cut_interval*.

Time cuts that occur earlier than any of the particle start times or after all the particles have become inactive are ignored. It is therefore always safe to provide more than enough time cut values. If TIME_CUT_OUTPUT is the only active output type, the particle trace terminates when a time equal to the largest time cut is reached by all the particles.

The *active*, *flow_state_data*, and *flow_gradient_data* parameters control what information is recorded. No trace output is recorded unless *active* = on. By default, all particle outputs and no flow outputs are recorded when trace output is active.

Particle values are the state of the particle, that is, its position, time, particle velocity, and so on. The flow outputs are found by interpolating the AcuSolve flow field to the current position and time of the particle.

Particle outputs always include

- Seed ID
- Coordinates
- Time
- Particle velocity
- Element ID
- Element set ID
- Marker
- Particle velocity magnitude
- Trace length

They also include

- Particle mass
- Particle density
- Particle radius

if the parameter *particle* equals finite_mass model in the EQUATION command,

- Component values

if the parameter *number_particle_components* in the EQUATION command is non-zero,

- User equation values

if one or more user equations are specified by the *user_equations* parameter of the EQUATION command.

- Turbulence random seed

if the parameter *turbulent_trace* equals on in the TRACE_PARAMETERS command and

- Stretch vector
- Stretch magnitude
- Log stretch magnitude
- Stretch rate magnitude

if the parameter *stretch* does not equal none in the EQUATION command.

Flow state values always include

- Flow velocity
- Flow pressure
- Flow velocity magnitude
- Flow strain rate magnitude

They also include

- Flow temperature
- Flow species
- Flow eddy viscosity

if these are available in the AcuSolve database.

Flow gradient values currently include only

- Gradient of flow velocity

> **Note:** For the massless particle motion equation, the particle and the flow velocity fields are identical except possibly for particle positions in elements next to a wall.

POINCARE_OUTPUT

Specifies parameters for the output of the Poincare plane sections.

Type

AcuTrace Command

Syntax

`POINCARE_OUTPUT {parameters}`

Qualifier

This command has no qualifier.

Parameters

`active` *(boolean) [=off]*
: Flag specifying if Poincare output is active.

`poincare_sections` *or* `psections` *(array) [={}]*
: Definition of the Poincare section rectangles. Each row is of the form x1, y1, z1,x2 ,y2 ,z2, x3, y3, z3 and defines a rectangle by the prescription given below.

`flow_state_data` *or* `flow_state` *(boolean) [=off]*
: Flag specifying whether to include the flow state in the output.

`flow_gradient_data` *or* `flow_gradient` *(boolean) [=off]*
: Flag specifying whether to include gradients of the flow state in the output.

Description

Particle traces are computed by AcuTrace as a series of segments. Poincare plane output is written only when a particle path crosses through and inside of a Poincare section rectangle. The particle path is interpolated between the segment endpoints that are on either side of the rectangle.

The Poincare section rectangles are defined by three points as follows: the first two points define one edge of the rectangle, and the third point is projected to the closest plane that is normal to this edge and passes through one of the first two points. The fourth point is constructed to finish the rectangle. In the simplest case, the three points can be three of the vertices of the rectangle.

For example, the following command defines two Poincare section rectangles, one with vertices at (.5587,-10.0, -10.0), (.5587,-10.0, 10.0), (.5587,10.0, -10.0), and (.5587,10.0, 10.0), and the other with vertices at (-.0499,-10.0, -10.0), (-.0499,-10.0, 10.0), (-.0499,10.0, -10.0), and (-.0499,10.0, 10.0):

```
POINCARE_OUTPUT {
   active                 = on
   poincare_sections      = { .5587, -10.0, -10.0,
                              .5587, -10.0, 10.0,
                              .5587, 10.0, -10.0 ;
                             -.0499, -10.0, -10.0,
                             -.0499, -10.0, 10.0,
                             -.0499, 10.0, -10.0 ;
```

```
                                    }
    ...
}
```

The *active*, *flow_state_data*, and *flow_gradient_data* parameters control what information is recorded. No trace output is recorded unless *active* = on. By default, all particle outputs and no flow outputs are recorded when trace output is active.

Particle values are the state of the particle, that is, its position, time, particle velocity, and so on. The flow outputs are found by interpolating the AcuSolve flow field to the current position and time of the particle.

Particle outputs always include

- Seed ID
- Coordinates
- Time
- Particle velocity
- Element ID
- Element set ID
- Marker
- Particle velocity magnitude
- Trace length

They also include

- Particle mass
- Particle density
- Particle radius

if the parameter *particle* equals finite_mass model in the EQUATION command,

- Component values

if the parameter *number_particle_components* in the EQUATION command is non-zero,

- User equation values

if one or more user equations are specified by the *user_equations* parameter of the EQUATION command.

- Turbulence random seed

if the parameter *turbulent_trace* equals on in the TRACE_PARAMETERS command and

- Stretch vector
- Stretch magnitude
- Log stretch magnitude
- Stretch rate magnitude

if the parameter *stretch* does not equal none in the EQUATION command.

Flow state values always include

- Flow velocity
- Flow pressure
- Flow velocity magnitude
- Flow strain rate magnitude

They also include

- Flow temperature
- Flow species
- Flow eddy viscosity

if these are available in the AcuSolve database.

Flow gradient values currently include only

- Gradient of flow velocity

> **Note:** For the massless particle motion equation, the particle and the flow velocity fields are identical except possibly for particle positions in elements next to a wall.

INTERPOLATE_OUTPUT

Specifies parameters for the output of particle and flow values at the particle seed locations.

Type

AcuTrace Command

Syntax

`INTERPOLATE_OUTPUT {parameters}`

Qualifier

This command has no qualifier.

Parameters

`active` *(boolean) [=off]*
: Flag specifying if interpolate output is active.

Description

Interpolate output is a legacy output type provided for backward capability.

Interpolate output does not require any particle tracing. It records the values of the particle and flow outputs at the particle seed locations at the earliest seed time. The output is written to one or more files with extension `.pin` in the working directory.

Interpolate output can be useful for determining which particle seeds are in the flow domain. Only seeds that are in the flow domain are recorded in the `.pin` files. To use interpolate output for this purpose, it is recommended that format be set to `ascii` and that AcuRunTrace be run in serial mode in which case a single `.pin` file is written.

The *`active`*, *`flow_state_data`*, and *`flow_gradient_data`* parameters control what information is recorded. No trace output is recorded unless *`active`* = on. By default, all particle outputs and no flow outputs are recorded when trace output is active.

Particle values are the state of the particle, that is, its position, time, particle velocity, and so on. The flow outputs are found by interpolating the AcuSolve flow field to the current position and time of the particle.

Particle outputs always include

- Seed ID
- Coordinates
- Time
- Particle velocity
- Element ID
- Element set ID
- Marker
- Particle velocity magnitude

- Trace length

They also include

- Particle mass
- Particle density
- Particle radius

if the parameter *particle* equals finite_mass model in the EQUATION command,

- Component values

if the parameter *number_particle_components* in the EQUATION command is non-zero,

- User equation values

if one or more user equations are specified by the *user_equations* parameter of the EQUATION command.

- Turbulence random seed

if the parameter *turbulent_trace* equals on in the TRACE_PARAMETERS command and

- Stretch vector
- Stretch magnitude
- Log stretch magnitude
- Stretch rate magnitude

if the parameter *stretch* does not equal none in the EQUATION command.

Flow state values always include

- Flow velocity
- Flow pressure
- Flow velocity magnitude
- Flow strain rate magnitude

They also include

- Flow temperature
- Flow species
- Flow eddy viscosity

if these are available in the AcuSolve database.

Flow gradient values currently include only

- Gradient of flow velocity

> **Note:** For the massless particle motion equation, the particle and the flow velocity fields are identical except possibly for particle positions in elements next to a wall.

Functional Commands

6

The functional commands, except for `AUTO_SOLUTION_STRATEGY`, are outlined in this chapter.

This chapter covers the following:

Functional commands are differentiated from declarative commands in that they are executed immediately; see the AcuSolve Command Reference Manual for further explanation. The purpose of the last three commands is to facilitate writing input files.

RUN

Processes the input data for the particle trace solver.

Type

AcuTrace 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

The purpose and function of this command is identical to the AcuSolve command of the same name. Refer to the AcuSolve Command Reference Manual for details.

INCLUDE

Includes the contents of an external file.

Type

AcuTrace Command

Syntax

`INCLUDE {parameters}`

Qualifier

This command has no qualifier.

Parameters

`file` *(string) [no default]*
Name of the input files to be included.

Description

The purpose and function of this command is identical to the AcuSolve command of the same name. Refer to the AcuSolve Command Reference Manual for details.

ASSIGN

Assigns a value to a variable.

Type

AcuTrace 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

The purpose and function of this command is identical to the AcuSolve command of the same name. Refer to the AcuSolve Command Reference Manual for details.

QUIT

Terminates parsing the trace input file.

Type

AcuTrace Command

Syntax

`QUIT {parameters}`

Qualifier

This command has no qualifier.

Parameters

This command has no parameters.

Description

The purpose and function of this command is identical to the AcuSolve command of the same name. Refer to the AcuSolve Command Reference Manual for details.

Intellectual Property Rights Notice

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

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

S

T

U