Altair® AcuSolve® 2025

Altair AcuSolve User-Defined Functions Manual

Updated: 11/13/2024

Contents

AcuSolve User-Defined Functions Manual

1

Customization of AcuSolve allowing you to customize certain capabilities of the solver.

AcuSolve is a general purpose, pressure based incompressible and compressible flow solver. Its technology is based on the Galerkin/Least-Squares (GLS) finite element method. It uses unstructured meshes of tetrahedron, pyramid, wedge and hexahedron elements, with nodal-based field variables. Two steps are needed to use user-defined functions; build a user-defined function and reference it in the input file.

Any number of user-defined functions may be used in a given problem. As further explained below, the user-defined functions are compiled and linked into one or more dynamic shared libraries. The script acuMakeLib on Unix/Linux machines and acuMakeDll on Windows machines may be used for this purpose. A list of these libraries is then given to AcuSolve via the configuration option user_libraries. The solver then sequentially searches through these libraries for the user-defined functions referenced in the input file.

Input File

2

To access a user-defined function in AcuSolve, you must reference it in the input file.

For example, to access the user-defined function `usrMyViscosity()` in the viscosity model, the following command may be specified in the input file:

```
VISCOSITY_MODEL( "my viscosity" ) {
   type = user_function
   user_function = "usrMyViscosity"
   user_values = { 1.781e-5, 2, 3 }
   user_strings = { "string1", "string2" }
}
```

In this example, the viscosity model my viscosity has a *user_function* type. This type requires, as a minimum, the name of the user-programmed routine, which is `usrMyViscosity()` for the example above. The value of the *user_function* parameter is exactly the name of the function (routine) used in the C program written by you. The parameters *user_values* and *user_strings* are used to pass parameters to the user function. In this example, three floating-point numbers and two strings are passed to the function. The order of these parameters is preserved, for example, the first user value is 1.781e-5, the second is two, and the third is three.

User-defined functions may be used by any of the following categories of commands:

1. Multiplier Function: `MULTIPLIER_FUNCTION`
2. Mesh Motion: `MESH_MOTION`, `FLEXIBLE_BODY`
3. Body Force: `GRAVITY`, `MASS_HEAT_SOURCE`, `VOLUME_HEAT_SOURCE`, `MASS_SPECIES_SOURCE`, `VOLUME_SPECIES_SOURCE`
4. Material Model: `DENSITY_MODEL`, `SPECIFIC_HEAT_MODEL`, `VISCOSITY_MODEL`, `CONDUCTIVITY_MODEL`, `DIFFUSIVITY_MODEL`, `POROSITY_MODEL`, `EMISSIVITY_MODEL`, `SURFACE_TENSION_MODEL`, `SOLAR_RADIATION_MODEL`
5. Component Model: `FAN_COMPONENT`, `HEAT_EXCHANGER_COMPONENT`
6. Element Output `ELEMENT_OUTPUT`
7. Element Boundary Condition: `ELEMENT_BOUNDARY_CONDITION`
8. Nodal Boundary Condition: `NODAL_BOUNDARY_CONDITION`
9. Periodic Boundary Condition: `PERIODIC_BOUNDARY_CONDITION`

See the *AcuSolve Command Reference Manual* for examples of user-defined functions for each of the above commands.

Function Format

3

A user-defined function has the following form (in C).

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE(usrFunc) ;
Void usrFunc (
         UdfHd   udfHd, /* Opaque handle for accessing data  */
         Real*   outVec, /* Output vector                     */
         Integer nItems, /* Number of items in outVec         */
         Integer vecDim /* Vector dimension of outVec        */
 ) {
         ...
         outVec[0] = ... ;
 } /* end of usrFunc() */
```

The header file `acusim.h` defines the standard C header file, such as `stdio.h` and `stdarg.h`. In addition, the data types used by AcuSolve are defined. The relevant types are:

Integer	Type integer
Real	Type floating point
String	Type string
Void	Type void

The header file `udf.h` contains the definitions and declarations needed by the user function:

- The definitions of symbolic constants, such as `UDF_VISCOSITY` and `UDF_ELM_VELOCITY`. These constants are used to access data.
- The prototypes of support routines
- The macro `UDF_PROTOTYPE()`, which may be used to prototype the user function

In addition, it contains the useful macros min(a,b), max(a,b), and abs(a) for the computation of minimum, maximum, and absolute value of integers or floating point numbers.

The user-function name (generically called usrFunc above) may be any name supported in C that starts with the letters usr. This naming convention avoids any potential conflict in function name space.

Four arguments are passed to a user-defined function:

Table 1:

• udfHd:	This is an opaque handle (pointer) which contains the necessary information for accessing various data. All supporting routines require this argument. For example, to access the current time step pass udfHd to the function `udfGetTimeStep()`, as in:
	`Integer time_step ;`
	`...`

```
time_step = udfGetTimeStep( udfHd ) ;
```

- outVec:

This is the resulting vector of the user function. You must fill this vector before returning. On input this array contains all zeros.

- nItems:

This is the first dimension of outVec, which specifies the number of items that needs to be filled. The value of this parameter depends on command category. In particular:

Table 2:

Command Category	nItems Definition
Multiplier Function	1
Mesh Motion	1
Body Force	Number of Elements
Material Model	Number of Elements
Component Model	Number of Elements
Element Output	Number of Elements
Element Boundary Condition	Number of Surfaces
Nodal Boundary Condition	Number of Nodes
Periodic Boundary Condition	Number of Nodal Pairs

where the "Number of Elements" is the number of elements in a block, "Number of Nodes" is the number of nodes in a block, "Number of Surfaces" is the number of surfaces in a block, and "Number of Nodal Pairs" is the number of pairs of nodes in a block.

- vecDim:

This is the second dimension of outVec, which specifies the vector dimension of the particular data. The value of this parameter also depends on command category. In particular:

Table 3:

Command Category	vecDim Definition
Multiplier Function	1
Mesh Motion	12
Body Force (gravity)	3
Body Force (other)	1
Material Model	1
Component Model	1
Element Output	1
Element Boundary Condition (tangential_traction)	3
Element Boundary Condition (Other)	1
Nodal Boundary Condition	1
Periodic Boundary Condition	1

The fastest dimension of outVec is nItems, and not vecDim. For example, to fill in a constant gravity given via user values, you may write the following code:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE(usrGrav) ;
Void usrGrav (
         UdfHd   udfHd, /* Opaque handle for accessing data  */
         Real*   outVec, /* Output vector                      */
         Integer nItems, /* Number of items in outVec         */
         Integer vecDim /* Vector dimension of outVec        */
 ) {
         Integer dir ; /* a running direction index         */
         Integer elem ; /* a running element index           */
         Real*   usrVals ; /* user values                        */
         udfCheckNumUsrVals( udfHd, 3 ) ; /* check for error   */
         usrVals = udfGetUsrVals( udfHd ) ; /* get the user vals */
         for ( dir = 0 ; dir < vecDim ; dir++ ) {
               for ( elem = 0 ; elem < nItems ; elem++ ) {
                   outVec[elem+dir*nItems] = usrVals[dir] ;
               }
          }
```

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

The corresponding input file command may be:

```
GRAVITY( "my gravity" ) {
             type = user_function
             user_function = "usrGrav"
             user_values = { 0, 0, -9.81 }
}
```

AcuSolve only accesses user-defined functions that are written in C. To write a user-function in Fortran, you must write a C cover function via which Fortran routine(s) are called. This cover function is accessed by AcuSolve. For the above example the C file may contain:

```
#include "acusim.h"
#include "udf.h"
UDF_PROTOTYPE(usrGrav) ;
#define usrGravF usrgravf_          /* FORTRAN name conversion           */
Void usrGrav (
         UdfHd   udfHd,             /* Opaque handle for accessing data  */
         Real*   outVec,            /* Output vector                     */
         Integer nItems,            /* Number of items in outVec         */
         Integer vecDim             /* Vector dimension of outVec        */
) {
         Real*   usrVals ;          /* user values                        */
         udfCheckNumUsrVals( udfHd, 3 ) ; /* check for error               */
         usrVals = udfGetUsrVals( udfHd ) ; /* get the user vals           */
         usrGravF( outVec, usrVals, &nItems ) ;
} /* end of usrGrav() */
```

While the Fortran file may contain:

```
         subroutine usrGravF ( grav, usrVals, nElems )
         integer nElems
         real*8 grav (nElems, 3), usrVals (3)
         integer i, j
c
c ...   fill in the vector
c
           do j = 1, 3
              do i = 1, nElems
                 grav(i,j) = usrVals (j)
              enddo
            enddo
c
c         return
c
              return
              end
```

Note: You are responsible for Fortran/C name conversion.

Compile, Link and Run

4

In order for the solver to access the user-defined functions, the functions must be compiled and linked into a shared library. The script acuMakeLib (for Linux) and acuMakeDll (for Windows) may be used for this purpose.

Assume you have the files `usrCover.c` and `usrGrav.f` containing the C and Fortran functions given in the previous section. To compile and link into the shared library `libusr.so` simply issue the command:

```
acuMakeLib -src usrCover.c,usrGrav.f
```

This script writes the makefile, `Makefile`, and invokes make to compile and link the routines. The makefile has two targets: make all (or equivalently make install) and make clean.

Once the file is successfully compiled and linked, the solver may be invoked as

```
acuSolve -libs ./libusr.so
```

or through the use of AcuRun as

```
acuRun -libs ./libusr.so
```

Multiple user-defined functions may be provided by one or more libraries. To run the solver with multiple libraries, give a comma separated list of libraries to the configuration option -libs. For example, given the libraries `libgrav1.so` and `libgrav2.so` stored in directory `~/acusim_libs`, the solver may be invoked as

```
acuRun -libs ~/acusim_libs/libgrav1.so,~/acusim_libs/libgrav2.so
```

The libraries are searched sequentially in the order given for the user functions. Libraries that do not exist are ignored. To see what libraries and what routines are accessed by the solver, invoke the solver with the option -verbose 2.

Note: For most computer platforms, the functions within a given user shared library can access all functions in that library, plus all system and solver functions. However, they may not access functions in other user shared libraries. This behavior is a platform dependent.

Note: On most platforms the shared library has a `.so` extension, however on some it may be `.sl` (such as HP) or `.dll` (Windows).

Support Routines

5

Within a user-defined function or an external output code data may be accessed through support routines. These routines provide flexibility, longevity, efficiency and the ability to check for run-time errors.

The support routines fall into these categories:

- Basic Routines: General UDF routines, such as user values and setting error flags.
- Client/Server Routines: Used for passing data back and forth between AcuSolve and an external program.
- Global Routines: Access global solver data such as time step information.
- Element Routines: Access element data, typically at quadrature points, such as velocity and gradient of temperature.
- Element Boundary Condition Routines: Access surface data, typically at surface quadrature points, such as velocity and normal direction.
- Nodal Boundary Condition Routines: Access nodal data such as velocity and pressure.
- Nodal Initial Condition Routines: Access nodal initial condition data at the nodes, such as velocity and pressure.
- Periodic Boundary Condition Routines: Access nodal data, by pairs of nodes, such as velocity and pressure.
- Subdomain Routines: Access data by subdomain, which is done by an external output user function traversing all the data in a problem, and not by a user function that returns data to AcuSolve.

Not all support routines are available for each user function. The following tables provide the valid access combinations:

Table 4:

Routines	Multiplier Function	Mesh Motion	Body Force	Material
Basic	yes	yes	yes	yes
Client/Server	yes	yes	no	no
Global	yes	yes	yes	yes
Element	no	no	yes	yes
Element Boundary Conditions	no	no	no	no

Routines	Multiplier Function	Mesh Motion	Body Force	Material
Nodal Boundary Conditions	no	no	no	no
Nodal Initial Conditions	no	no	no	no
Periodic Boundary Conditions	no	no	no	no
Subdomain	no	no	no	no

Table 5:

Component	Element Output	Element Boundary Condition	Nodal Boundary Condition	Periodic Boundary Condition	Nodal Initial Condition
yes	yes	yes	yes	yes	yes
no	no	no	no	no	no
yes	yes	yes	yes	yes	yes
yes	yes	no	no	no	no
no	no	yes	no	no	no
no	no	no	yes	no	no
no	no	no	no	no	yes
no	no	no	no	yes	no
no	no	no	no	no	no

In addition, an external output code can access the Basic, Global, Element, Element Boundary Condition and Subdomain support routines.

Basic Routines

6

These routines are accessible by all user functions, and by external output codes.

This chapter covers the following:

udfGetType()

Get the type of the user-defined function.

Syntax

```
type = udfGetType( udfHd ) ;
```

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd

The opaque handle (pointer) which was passed to the user function.

Return Value

type *(integer)*

The returned **type** can have several different values, depending on the user-defined function type:

Returned Type Value	Meaning of Type Value
UDF_MULT_FUNCTION	Multiplier function.
UDF_NODAL_BC	Nodal boundary condition.
UDF_NODAL_BC_DIR	Nodal BC direction vector.
UDF_PERIODIC_BC	Periodic boundary condition.
UDF_ELEMENT_COMPONENT	Element component (fan or heat exchanger).
UDF_ELEMENT_BC	Element boundary condition.
UDF_VISCOSITY	Element viscosity.
UDF_CONDUCTIVITY	Element thermal conductivity.
UDF_DIFFUSIVITY	Element species diffusivity.
UDF_GRAVITY	Element gravity (generalized volumetric body force).
UDF_MASS_HEAT_SOURCE	Element heat source per unit mass.
UDF_VOLUME_HEAT_SOURCE	Element heat source per unit volume.
UDF_MASS_SPECIES_SOURCE	Element species source per unit mass.
UDF_VOLUME_SPECIES_SOURCE	Element species source per unit volume.

Returned Type Value	Meaning of Type Value
UDF_EMISSIVITY	Radiation surface emissivity.
UDF_DENSITY	Element density.
UDF_SPECIFIC_HEAT	Element enthalpy.
UDF_MESH_MOTION	Mesh motion.
UDF_ELEMENT_OUTPUT	Element output.
UDF_SURFACE_TENSION	Surface tension.
UDF_CONTACT_ANGLE	Contact angle.
UDF_FLEXIBLE_BODY	Flexible body.
UDF_VISCOELASTIC	Viscoelastic model.

Description

This routine returns the type of the user-defined function. This routine may be used to determine the expected return value of the function. Among other usages, this allows a single routine to be used for multiple objectives. For example,

```
Integer type ;
...
type = udfGetType( udfHd ) ;
UDF_PROTOTYPE(usrFunc) ;
if ( type == UDF_VISCOSITY ) {
    /* compute viscosity */
    ...
} else if ( type == UDF_CONDUCTIVITY ) {
    /* compute thermal conductivity */
    ...
} else {
udfSetError( udfHd, "Invalid type %d", type ) ;
}
```

Errors

This routine expects a valid *udfHd* as an argument. An invalid argument returns an error.

udfGetProcId()

Return the processor ID.

Syntax

```
nProc = udfGetProcId( udfHd ) ;
```

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

***nProc** (integer)*
: The rank of the processor that is calling the function.

Description

The returned integer is the rank of the processor that is calling the function. For example,

```
Integer nProc ;
...
nProc = udfGetProcId( udfHd ) ;
```

Errors

This routine expects a valid *udfHd* as an argument; an invalid argument returns an error.

udfGetName()

Return the user-given name of the input command associated with the user function.

Syntax

```
name = udfGetName( udfHd ) ;
```

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

name *(string)*
: The returned string value is the User-given name of the input command calling this user function.

Description

This routine returns the user-given name of the input command associated with the user function. This facilitates correlating with the input file. For example,

```
String user_name ;
...
user_name = udfGetName( udfHd ) ;
udfPrintMess( "Inside command with name <%s>", user_name ) ;
```

Errors

This routine expects a valid *udfHd* as an argument; an invalid argument returns an error.

udfCheckNumUsrVals()

Check the number of user values given in the input file.

Syntax

udfCheckNumUsrVals(udfHd, nUsrVals) ;

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

nUsrVals
: Number of user values that the function expects.

Return Value

None

Description

This routine verifies that the number of user values supplied in the input file is the same as the second argument. If it is not, an error message is issued and the solver exits. For example,

```
udfCheckNumUsrVals( udfHd, 2 ) ;
```

Errors

This routine expects a valid *udfHd*.

udfGetNumUsrVals()

Return the number of user values supplied in the input file.

Syntax

nUsrVals = udfGetNumUsrVals(udfHd) ;

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nUsrVals *(integer)*
: The number of user values given in the input file is returned as an integer.

Description

This routine returns the number of user values supplied in the input file. It is most useful when the number of parameters is not know a priori, such as for an interpolated curve fit. For example,

```
Integer nUsrVals ;
Integer i ;
Real x, y ;
Real* usrVals ;
...
nUsrVals = udfGetNumUsrVals( udfHd ) ;
usrVals = udfGetUsrVals( udfHd ) ;
for ( i = 0 ; i < nUsrVals ; i+=2 ) {
    x = usrVals[i] ;
    y = usrVals[i+1] ;
...
}
```

Errors

This routine expects a valid *udfHd*.

udfGetUsrVals()

Return the array of user-supplied values.

Syntax

```
usrVals = udfGetUsrVals( udfHd ) ;
```

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

usrVals *(real)*
: The return value is a pointer to one-dimensional real array of user values as given in the input file. This array has `udfGetNumUsrVals()` entries.

Description

This routine returns the real array of user-given parameters. The order of the parameters within the array is the same as in the input file. For example,

```
Real* usrVals ;
Real amp, omega ;
...
udfCheckNumUsrVals( udfHd, 2 ) ;
usrVals = udfGetUsrVals( udfHd ) ;
amp = usrVals[0] ;
omega = usrVals[1] ;
```

Errors

This routine expects a valid *udfHd*.

udfCheckNumUsrStrs()

Check the number of user strings given in the input file.

Syntax

udfCheckNumUsrStrs(udfHd, nUsrStrs) ;

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

nUsrStrs *(integer)*
: Number of user strings that the function expects.

Return Value

None

Description

This routine verifies that the number of user strings supplied in the input file is the same as the second argument. If it is not, an error message is issued and the solver exits. For example,

```
udfCheckNumUsrStrs( udfHd, 2 ) ;
```

Errors

This routine expects a valid *udfHd*.

udfGetNumUsrStrs()

Return the number of user strings supplied in the input file.

Syntax

```
nUsrStrs = udfGetNumUsrStrs( udfHd ) ;
```

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nUsrStrs *(integer)*
: The return value is the number of user strings given in the input file.

Description

This routine returns the number of user strings supplied in the input file. For example,

```
Integer nUsrStrs ;
...
nUsrStrs = udfGetNumUsrStrs( udfHd ) ;
if ( nUsrStrs != 2 ) {
   udfSetError( udfHd, "Invalid number of user strings %d", nUsrStrs ) ;
}
```

Errors

This routine expects a valid *udfHd*.

udfGetUsrStrs()

Return the array of user supplied strings.

Syntax

```
usrStrs = udfGetUsrStrs( udfHd ) ;
```

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd

The opaque handle (pointer) which was passed to the user function.

Return Value

usrStrs *(string)*

The return value is an array of user strings given in the input file. This array has `udfGetNumUsrStrs()` entries.

Description

This routine returns the array of user strings given in the input file. For example,

```
String* usrStrs ;
String inflowOsi ;     /* inflow surface output */
String outflowOsi ;    /* outflow surface output */
...
udfCheckNumUsrStrs( udfHd, 2 ) ;
usrStrs = udfGetUsrStrs( udfHd ) ;
inflowOsi = usrStrs[0] ;
outflowOsi = usrStrs[1] ;
```

Errors

This routine expects a valid *udfHd*.

udfCheckNumUsrHists()

Check the number of user history variables given in the input file.

Syntax

udfCheckNumUsrHists(udfHd, nUsrHists) ;

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

usrStrs
: The number of user history variables that the function expects.

Return Value

None.

Description

This routine verifies that the number of user history variables supplied in the input file is the same as the second argument. If it is not, an error message is issued and the solver exits. For example,

```
udfCheckNumUsrHists( udfHd, 2 ) ;
```

Errors

This routine expects a valid *udfHd*.

udfGetNumUsrHists()

Return the number of user history variables supplied in the input file.

Syntax

nUsrHists = udfGetNumUsrHists(udfHd) ;

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
The opaque handle (pointer) which was passed to the user function.

Return Value

nUsrHists *(integer)*
The returned value is the number of user history variables given in the input file.

Description

This routine returns the number of user history variables supplied in the input file. For example,

```
Integer nUsrHists ;
...
nUsrHists = udfGetNumUsrHists( udfHd ) ;
if ( nUsrHists != 2 ) {
   udfSetError( udfHd, "Invalid number of user history variables %d",
                nUsrHists ) ;
}
```

Errors

This routine expects a valid *udfHd*.

udfGetUsrHistsCurr()

Return the array of current history variables.

Syntax

```
usrHistsCurr = udfGetUsrHistsCurr( udfHd ) ;
```

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

usrHistsCurr *(real)*
: The return value is an array of current history variables. This array has `udfGetNumUsrHists()` entries.

Description

This routine returns the array of current history variables. For the first time step, this array is initialized to the array given in the input file. The array may be overwritten in the user function. It is saved at the end of the time step as the "previous" history variables for use during the next time step. The current history variables are also written to the restart file for continuity across restarts. For example,

```
Real* usrHistsCurr ;                    /* current step history variables */
Real* usrHistsPrev ;                    /* previous step history variables */
...
udfCheckNumUsrHists( udfHd, 2 ) ;
usrHistsCurr                            = udfGetUsrHistsCurr( udfHd ) ;
usrHistsPrev                            = udfGetUsrHistsPrev( udfHd ) ;
usrHistsCurr[0]                         = usrHistsPrev[0] + 1. ;
usrHistsCurr[1] = usrHistsPrev[1] + 2. ;
```

Errors

This routine expects a valid *udfHd*.

udfGetUsrHistsPrev()

Return the array of history variables from the previous time step.

Syntax

```
usrHistsPrev = udfGetUsrHistsPrev( udfHd ) ;
```

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

usrHistPrev *(real)*
: The return value is an array of history variables from the previous time step. This array has `udfGetNumUsrHists()` entries.

Description

This routine returns the array of history variables from the previous time step. For the first time step, this array is initialized to the array given in the input file. The array is read-only. At the end of the time step the "current" history variables are copied into the previous history variables. See `udfGetUsrHistsCurr` for an example.

Errors

This routine expects a valid *udfHd*.

udfFirstCall()

Determine whether this the first time a user-defined function is called.

Syntax

```
firstCall = udfFirstCall( udfHd ) ;
```

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

firstCall *(integer)*
: The return value is a flag indicating whether or not this is the first time this user function is called:

0	No, this is not the first time.
1	Yes, this is the first time.

Description

This routine determines if this is the first time a specified user-defined function is called. For example,

```
Integer firstCall ;
...
firstCall = udfFirstCall( udfHd ) ;
 if ( firstCall == 1 ) {
    /* possible initialization */
} else {
    /* the rest of the time */
}
```

Errors

This routine expects a valid *udfHd*.

udfFirstStep()

Determine whether this is the initial time step of this run.

Syntax

`firstStep = udfFirstStep( udfHd ) ;`

Type

AcuSolve User-Defined Function Basic

Parameters

`udfHd`

The opaque handle (pointer) which was passed to the user function.

Return Value

`firstStep` *(integer)*

The integer return value **`firstStep`** indicates whether or not this is the initial time step of this run:

0	No, this is not the first time.
1	Yes, this is the first time.

Description

This routine determines if this user-defined function is called during the first time step. For example,

```
Integer firstStep ;
...
firstStep = udfFirstStep( udfHd ) ;
if ( firstStep == 1 ) {
   /* possible initialization */
} else {
   /* the rest of the time */
}
```

Errors

This routine expects a valid *`udfHd`*.

udfSetError()

Print an error message and exit the solver.

Syntax

udfSetError(udfHd, format, ...) ;

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

format
: C printf() format.

...
: List of C variables.

Return Value

None.

Description

This routine allows the user to print out a message and exit the solver if an error is detected. The argument list is the same as that of fprintf(), except that here *udfHd* takes the place of the file descriptor. For example,

```
String user_name ;
...
user_name = udfGetName( udfHd ) ;
if ( strcmp(user_name,  "viscosity water") != 0 ) {
   /* compute viscosity for water */
   ...
} else if ( strcmp(user_name, "viscosity air") != 0 ) {
   /* compute viscosity for air */
   ...
} else {
   udfSetError( udfHd, "Invalid user name %s", user_name ) ;
}
```

Errors

This routine expects a valid *udfHd*.

udfPrintMess()

Print a message to the standard output, or `.log` file, if active.

Syntax

`udfPrintMess( udfHd, format, ... ) ;`

Type

AcuSolve User-Defined Function Basic

Parameters

`udfHd`
: The opaque handle (pointer) which was passed to the user function.

`format`
: C printf() format.

`...`
: List of C variables.

Return Value

None

Description

This routine prints a message to the standard output, or the `.log` file, if active. The argument list is the same as that of fprintf(, except that here *`udfHd`* takes the place of the file descriptor. For example,

```
String user_name ;
...
user_name = udfGetName( udfHd ) ;
udfPrintMess( udfHd, "user name is %s", user_name ) ;
```

Errors

This routine expects a valid *`udfHd`*.

udfPrintMessPrim()

Print a message to the standard output, or `.log` file, if active, for the primary processor/subdomain.

Syntax

`udfPrintMessPrim( udfHd, format, ... ) ;`

Type

AcuSolve User-Defined Function Basic

Parameters

`udfHd`
: The opaque handle (pointer) which was passed to the user function.

`format` ***(string)***
: C printf() format.

`...` ***(vararg)***
: List of C variables.

Return Value

None

Description

This routine prints a message to the standard output, or the `.log` file, if active, but only for the primary processor or subdomain. The argument list is the same as that of fprintf(, except that here *`udfHd`* takes the place of the file descriptor. For example,

```
udfPrintMessPrim( udfHd, "return value is %g", outVec[0] ) ;
```

Errors

This routine expects a valid *`udfHd`*.

udfSetSig()

Set a signal to alter the behavior of AcuSolve.

Syntax

```
udfSetSig( udfHd, name, value ) ;
```

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd

The opaque handle (pointer) which was passed to the user function.

name

integer

Symbolic name of the signal to be set

UDF_SIG_NO_CONV	Stagger not converged.
UDF_SIG_DONT_AMP_TIMEINC	Do not amplify time increment.
UDF_SIG_REDUCE_TIMEINC	Reduce time increment.
UDF_SIG_REDO_STEP	Redo time step.
UDF_SIG_MUST_REDO_STEP	Must redo time step.
UDF_SIG_FATAL	Fatal error, exit.
UDF_SIG_STOP	Stop at end of step.
UDF_SIG_OUTPUT	Output at end of step.
UDF_SIG_OUTPUT_RESIDUAL	Output nodal residual.
UDF_SIG_PARTIAL_UPDATE	Partially update solution.

value (real)

Value of the signal. Ignored except for UDF_SIG_PARTIAL_UPDATE name.

Return Value

None

Description

This routine allows you to send a signal to AcuSolve from a user-defined function. For example, to signal AcuSolve to stop at the end of the current time step:

```
udfSetSig( udfHd, UDF_SIG_STOP, 0 ) ;
```

Errors

- This routine expects a valid *udfHd*.
- *name* must be one of the supported names.

udfPrim()

Determine whether this the primary processor/subdomain.

Syntax

`primary = udfPrim( udfHd ) ;`

Type

AcuSolve User-Defined Function Basic

Parameters

`udfHd`

The opaque handle (pointer) which was passed to the user function.

Return Value

`primary` *(integer)*

The return value **`primary`** is a Flag indicating whether or not this is the primary processor:

0	No, this is not the primary processor.
1	Yes, this is the primary processor.

Description

This routine determines if the current processor/subdomain is the primary one. For example,

```
Integer primary ;
 ...
primary = udfPrim( udfHd ) ;
if ( primary == 1 ) {
   /* possible global calculation */
} else {
   /* other processors */
}
```

Errors

This routine expects a valid *`udfHd`*.

udfBcastVector()

Broadcast a vector to all processors from the primary processor/subdomain.

Syntax

udfBcastVector(udfHd, vec, nDims) ;

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

vec ***(real)***
: Pointer to vector of data.

nDims ***(integer)***
: Size of vector.

Return Value

None

Description

This routine broadcasts a vector to all processors from the primary processor. For example,

```
Real vec[3]     = { 1., 0., 0. } ;
Integer nDims   = 3 ;
udfBcastVector( udfHd, vec, nDims ) ;
```

Errors

This routine expects a valid *udfHd*.

udfSetUsrHandle()

Save the handle (pointer) to user data for future retrieval.

Syntax

udfSetUsrHandle(udfHd, name, usrHandle) ;

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

name *(string)*
: Name of the user data.

usrHandle *(real)*
: Pointer to user data.

Return Value

None

Description

This routine saves the handle (pointer) to user data for future retrieval. For example,

```
if ( udfFirstCall(udfHd) ) {
     vtkHd = (VtkAcuSolve*) malloc( sizeof(VtkAcuSolve) ) ;
     udfSetUsrHandle( udfHd, "VTK", (Data) vtkHd ) ;
}
```

Errors

This routine expects a valid *udfHd*.

udfGetUsrHandle()

Return the handle (pointer) to previously saved user data.

Syntax

```
usrHandle = udfGetUsrHandle( udfHd, name ) ;
```

Type

AcuSolve User-Defined Function Basic

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

name (Void)*
: Name of the user data.

Return Value

***usrHandle** (real)*
: Pointer to user data.

Description

This routine returns the previously saved pointer to the specified user data. If the data is not found, then NULL is returned. For example,

```
VtkAcuSolve *vtkHd ;
...
vtkHd = udfGetUsrHandle ( udfHd, "VTK" ) ;
```

Errors

This routine expects a valid *udfHd*

Client/Server Routines

7

These routines are accessible only by multiplier function and mesh motion user functions.

This chapter covers the following:

udfOpenPipe()

Open a pipe to an external program.

Syntax

```
udfOpenPipe( udfHd, command ) ;
```

Type

AcuSolve User-Defined Function Client

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

command *(string)*
: Name and arguments of the external program.

Return Value

None

Description

This routine opens a client/server pipe to an external program, given by command. The external program must use the standard read/write calls to receive/send ASCII data. For example,

```
udfOpenPipe( udfHd, "rsh pelican usrPres.pl 4 1 0" ) ;
```

Errors

- This routine expects a valid *udfHd*.
- May only be called from a multiplier function or mesh motion user-defined function.

udfOpenPipePrim()

Open a pipe to an external program from main processor.

Syntax

udfOpenPipePrim(udfHd, command) ;

Type

AcuSolve User-Defined Function Client

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

command *(string)*
: Name and arguments of the external program.

Return Value

None

Description

This routine opens a client/server pipe to an external program, given by command. The external program must use the standard read/write calls to receive/send ASCII data. This function is the same as `udfOpenPipe()` except that only the main processor or subdomain opens the pipe, that is, only a single copy of the external program is executed. For example,

```
udfOpenPipePrim( udfHd, "rsh pelican usrPres.pl 4 1 0" ) ;
```

Errors

- This routine expects a valid *udfHd*
- May only be called from a multiplier function or mesh motion user-defined function.

udfWritePipe()

Write (send) a message to an external program.

Syntax

```
udfWritePipe( udfHd, format, ... ) ;
```

Type

AcuSolve User-Defined Function Client

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

format (string)
: C `printf()` format.

... (vararg)
: List of C variable.

Return Value

None

Description

This routine writes (sends) a message to the pipe opened by `udfOpenPipe` or `udfOpenPipePrim`. The argument list is the same as that of fprintf(), except that here *udfHd* takes the place of the file descriptor. For example,

```
udfWritePipe( udfHd, "pressure= %.16g", -trac[0] ) ;
```

Errors

- This routine expects a valid *udfHd*.
- May only be called from a multiplier function or mesh motion user-defined function.

udfReadPipe()

Read (receive) a line of data from an external program.

Syntax

udfreadPipe(udfHd, buffer, buffSize) ;

Type

AcuSolve User-Defined Function Client

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

buffer *(string)*
: Pointer to the line of data.

buffSize *(integer)*
: Number of bytes in the line of data.

Return Value

None

Description

This routine reads (receives) a message from the pipe opened by `udfOpenPipe` or `udfOpenPipePrim`. For example,

```
udfReadPipe( udfHd, &buffer, 1024 ) ;
```

Errors

- This routine expects a valid *udfHd*.
- May only be called from a multiplier function or mesh motion user-defined function.

Global Routines

8

These routines are accessible by all user functions, and by external output codes.

This chapter covers the following:

udfGetTimeStep()

Return the current time step.

Syntax

```
timeStep = udfGetTimeStep( udfHd ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

timeStep *(integer)*
: Current time step.

Description

This routine returns the current time step. For example,

```
Integer timeStep ;
...
timeStep = udfGetTimeStep( udfHd ) ;
```

Errors

This routine expects a valid *udfHd*.

udfGetTime()

Return the time of the current time step.

Syntax

```
time = udfGetTime( udfHd ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

time *(real)*
: Time of the current time step.

Description

This routine returns the time of the current time step. For example,

```
Real time ;
...
time = udfGetTime( udfHd ) ;
```

Errors

This routine expects a valid *udfHd*.

udfGetTimeAlpha()

Return the time at the alpha point of the generalized-alpha time marching algorithm.

Syntax

`timeAlpha = udfGetTimeAlpha( udfHd ) ;`

Type

AcuSolve User-Defined Function Global

Parameters

`udfHd`
: The opaque handle (pointer) which was passed to the user function.

Return Value

`timeAlpha` *(real)*
: Time at the alpha point of the generalized-alpha time marching algorithm.

Description

This routine returns the time at the alpha point of the generalized-alpha time marching algorithm. The element residual is evaluated at this time; hence if a material model or other element data is dependent on time, this function must be used to get the time. For example,

```
Real timeAlpha ;
...
timeAlpha = udfGetTimeAlpha( udfHd ) ;
```

Errors

This routine expects a valid *`udfHd`*.

udfGetTimeInc()

Return the current time increment.

Syntax

```
timeInc = udfGetTimeInc( udfHd ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

timeInc *(real)*
: Current time increment.

Description

This routine returns the current time increment. For example,

```
Real timeInc ;
...
timeInc = udfGetTimeInc( udfHd ) ;
```

Errors

This routine expects a valid *udfHd*.

udfGetResidualNorm()

Return the current residual norm of requested equation. This value forms the numerator of the residual ratio for the given equation.

Syntax

```
conv = udfGetResidualNorm( udfHd, eqnName) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

eqnName (integer)
: Symbolic name of the requested equation.

UDF_EQN_PRESSURE	Pressure.
UDF_EQN_VELOCITY	Velocity.
UDF_EQN_SPECIES	Species.
UDF_EQN_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_EQN_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_EQN_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_EQN_MESH_DISPLACEMENT	Mesh displacement.
UDF_EQN_RADIATION	Radiation.

Return Value

***conv** (real)*
: This routine returns a pointer containing the current convergence information. If the requested convergence information is not available, an assertion is given.

The dimension of the returned array depends on *eqnName* as follows:

eqnName	**Array Dimension**
UDF_EQN_PRESSURE	1
UDF_EQN_VELOCITY	1
UDF_EQN_SPECIES	`udfGetNumSpecs()`

eqnName	Array Dimension
UDF_EQN_EDDY_VISCOSITY	1
UDF_EQN_KINETIC_ENERGY	1
UDF_EQN_EDDY_FREQUENCY	1
UDF_EQN_MESH_DISLACEMENT	1
UDF_EQN_RADIATION	1

Description

This routine returns the current convergence level of the residual norm. For example,

```
Real* conv ;
Real presResNorm;
...
conv = udfGetResidualNorm( udfHd, UDF_EQN_PRESSURE) ;
presResNorm = conv[0];
```

Errors

This routine expects a valid *udfHd* and *eqnName*.

udfGetResidualRatio()

Return the current residual ratio of the requested equation.

Syntax

```
conv = udfGetResidualRatio( udfHd, eqnName) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

eqnName (integer)
: Symbolic name of the requested equation.

UDF_EQN_PRESSURE	Pressure.
UDF_EQN_VELOCITY	Velocity.
UDF_EQN_SPECIES	Species.
UDF_EQN_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_EQN_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_EQN_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_EQN_MESH_DISPLACEMENT	Mesh displacement.
UDF_EQN_RADIATION	Radiation.

Return Value

***conv** (real)*
: This routine returns a pointer containing the current convergence information. If the requested convergence information is not available, an assertion is given.

The dimension of the returned array depends on *eqnName* as follows:

eqnName	**Array Dimension**
UDF_EQN_PRESSURE	1
UDF_EQN_VELOCITY	1
UDF_EQN_SPECIES	`udfGetNumSpecs()`
UDF_EQN_EDDY_VISCOSITY	1

eqnName	Array Dimension
UDF_EQN_KINETIC_ENERGY	1
UDF_EQN_EDDY_FREQUENCY	1
UDF_EQN_MESH_DISLACEMENT	1
UDF_EQN_RADIATION	1

Description

This routine returns the current residual ratio convergence level. For example,

```
Real* conv ;
Real  presResRatio;
...
conv = udfGetResidualRatio( udfHd, UDF_EQN_PRESSURE) ;
presResRatio = conv[0];
```

Errors

This routine expects a valid *udfHd* and *eqnName*.

udfGetSolutionNorm()

Return the current solution norm of requested equation. This value forms the numerator of the solution ratio for the given equation.

Syntax

```
conv = udfGetSolutionNorm( udfHd, eqnName) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

eqnName (integer)
: Symbolic name of the requested equation.

UDF_EQN_PRESSURE	Pressure.
UDF_EQN_VELOCITY	Velocity.
UDF_EQN_SPECIES	Species.
UDF_EQN_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_EQN_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_EQN_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_EQN_MESH_DISPLACEMENT	Mesh displacement.
UDF_EQN_RADIATION	Radiation.

Return Value

conv*(real)*
: This routine returns a pointer containing the current convergence information. If the requested convergence information is not available, an assertion is given.

The dimension of the returned array depends on *eqnName* as follows:

eqnName	**Array Dimension**
UDF_EQN_PRESSURE	1
UDF_EQN_VELOCITY	1
UDF_EQN_SPECIES	`udfGetNumSpecs()`

eqnName	Array Dimension
UDF_EQN_EDDY_VISCOSITY	1
UDF_EQN_KINETIC_ENERGY	1
UDF_EQN_EDDY_FREQUENCY	1
UDF_EQN_MESH_DISLACEMENT	1
UDF_EQN_RADIATION	1

Description

This routine returns the current convergence information of the solution norm. For example,

```
Real* conv ;
Real  presSolNorm;
...
conv = udfGetSolutionNorm( udfHd, UDF_EQN_PRESSURE) ;
presSolNorm = conv[0];
```

Errors

This routine expects a valid *udfHd* and *eqnName*.

udfGetSolutionRatio()

Return the current solution ratio of the requested equation.

Syntax

```
conv = udfGetSolutionRatio( udfHd, eqnName) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

eqnName (integer)
: Symbolic name of the requested equation.

UDF_EQN_PRESSURE	Pressure.
UDF_EQN_VELOCITY	Velocity.
UDF_EQN_SPECIES	Species.
UDF_EQN_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_EQN_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_EQN_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_EQN_MESH_DISPLACEMENT	Mesh displacement.
UDF_EQN_RADIATION	Radiation.

Return Value

***conv** (real)*
: This routine returns a pointer containing the current convergence information. If the requested convergence information is not available, an assertion is given.

The dimension of the returned array depends on *eqnName* as follows:

eqnName	**Array Dimension**
UDF_EQN_PRESSURE	1
UDF_EQN_VELOCITY	1
UDF_EQN_SPECIES	`udfGetNumSpecs()`
UDF_EQN_EDDY_VISCOSITY	1

eqnName	Array Dimension
UDF_EQN_KINETIC_ENERGY	1
UDF_EQN_EDDY_FREQUENCY	1
UDF_EQN_MESH_DISLACEMENT	1
UDF_EQN_RADIATION	1

Description

This routine returns the current convergence information of the solution ratio. For example,

```
Real* conv ;
Real  presSolRatio;
...
conv = udfGetSolutionRatio( udfHd, UDF_EQN_PRESSURE) ;
presSolRatio = conv[0];
```

Errors

This routine expects a valid *udfHd* and *eqnName*.

udfHasFlow()

Does this problem contain flow equations?

Syntax

`flowFlag = udfHasFlow( udfHd ) ;`

Type

AcuSolve User-Defined Function Global

Parameters

`udfHd`
: The opaque handle (pointer) which was passed to the user function.

Return Value

`flowFlag` *(integer)*
: The return value is a flag indicating whether or not this problem has flow equations:

0	No
1	Yes

Description

This routine determines if this problem contains flow equations. For example,

```
Integer flowFlag ;
...
flowFlag = udfHasFlow( udfHd ) ;
if ( flowFlag == 1 ) {
/* Logic if flow equations exist */
...
)
```

Errors

This routine expects a valid *`udfHd`*.

udfHasTemp()

Does this problem contain a temperature equation?

Syntax

```
tempFlag = udfHasTemp( udfHd ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

tempFlag *(integer)*
: Flag indicating whether or not this problem has temperature equation.

The return value is a flag indicating whether or not this problem has temperature equations:

0	No
1	Yes

Description

This routine determines if this problem contains a temperature equation. For example,

```
Integer tempFlag ;
...
tempFlag = udfHasTemp( udfHd ) ;
if ( tempFlag == 1 ) {
/* Logic if temperature equations exist */
...
)
```

Errors

This routine expects a valid *udfHd*.

udfHasSpec()

Does this problem contain species equations?

Syntax

```
specFlag = udfHasSpec( udfHd ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

specFlag*(integer)*
: Flag indicating whether or not this problem has species equation(s).

The return value is a flag indicating whether or not this problem has species equations:

0	No
1	Yes

Description

This routine determines if this problem contains species equations. For example,

```
Integer specFlag ;
...
specFlag = udfHasSpec( udfHd ) ;
if ( specFlag == 1 ) {
/* Logic if species equations exist */
...
)
```

Errors

This routine expects a valid *udfHd*.

udfHasTurb()

Does this problem contain a turbulence equation?

Syntax

`turbFlag = udfHasTurb( udfHd ) ;`

Type

AcuSolve User-Defined Function Global

Parameters

`udfHd`

The opaque handle (pointer) which was passed to the user function.

Return Value

`turbFlag` *(integer)*

Flag indicating whether or not this problem has turbulence equation(s).

The return value is a flag indicating whether or not this problem has turbulence equations:

0	No
1	Yes

Description

This routine determines if this problem contains a turbulence equation. For example,

```
Integer turbFlag ;
...
turbFlag = udfHasTurb( udfHd ) ;
if ( turbFlag == 1 ) {
   /* Logic if turbulence equations exist */
   ...
)
```

Errors

This routine expects a valid *`udfHd`*.

udfHasAle()

Does this problem have mesh movement?

Syntax

```
aleFlag = udfHasAle( udfHd ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd

The opaque handle (pointer) which was passed to the user function.

Return Value

aleFlag *(integer)*

Flag indicating whether or not this problem has mesh movement.

The return value is a flag indicating whether or not this problem has mesh movement equations:

0	No
1	Yes

Description

This routine determines if this problem contains mesh movement. For example,

```
Integer aleFlag ;
...
aleFlag = udfHasAle( udfHd ) ;
if ( aleFlag == 1 ) {
   /* Logic if mesh movement exist */
   ...
)
```

Errors

This routine expects a valid *udfHd*.

udfGetNumSpecs()

Number of species transport equations in the problem.

Syntax

```
nSpecs = udfGetNumSpecs( udfHd ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nSpecs *(integer)*
: Number of species in the problem.

Description

This routine returns the number of species in the problem. For example,

```
Integer nSpecs ;
...
nSpecs = udfGetNumSpecs( udfHd ) ;
```

Errors

This routine expects a valid *udfHd*.

udfGetActSpecId()

The identification number of the current species being solved.

Syntax

specId = udfGetActSpecId(udfHd) ;

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

specId *(integer)*
: Identification number of the species being solved. If not within a species stagger, the return value is zero. The return value is one based.

Description

This routine returns the identification number of species being solved in the current stagger. For example,

```
Integer specId ;
...
specId = udfGetActSpecId( udfHd ) ;
```

Errors

This routine expects a valid *udfHd*.

udfGetOsiData()

Return data from an integrated surface output set.

Syntax

```
osiData = udfGetOsiData( udfHd, setName, dataName ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

setName (string)
: Name of the SURFACE_OUTPUT set.

dataName (integer)
: Symbolic name of the requested data.

UDF_OSI_AREA	Total area.
UDF_OSI_MASS_FLUX	Total mass flux.
UDF_OSI_MOMENTUM_FLUX	Total momentum flux.
UDF_OSI_TRACTION	Total traction.
UDF_OSI_MOMENT	Total moment.
UDF_OSI_HEAT_FLUX	Total heat flux.
UDF_OSI_CONVECTIVE_SPECIES	Total convective species flux.
UDF_OSI_VELOCITY	Average velocity.
UDF_OSI_PRESSURE	Average pressure.
UDF_OSI_TOTAL_PRESSURE	Average total pressure.
UDF_OSI_TEMPERATURE	Average temperature.
UDF_OSI_SPECIES	Average species.
UDF_OSI_EDDY_VISCOSITY	Average turbulence eddy viscosity.
UDF_OSI_KINETIC_ENERGY	Average turbulence kinetic energy.
UDF_OSI_EDDY_FREQUENCY	Average turbulence eddy frequency.
UDF_OSI_MESH_DISPLACEMENT	Average mesh displacement.

UDF_OSI_MESH_VELOCITY	Average mesh velocity.
UDF_OSI_TURBULENCE_YPLUS	Average y-plus of first node.
UDF_OSI_FILM_COEFFICIENT	Average heat transfer film coefficient.
UDF_OSI_AVE_DENSITY	Average density.
UDF_OSI_MAV_VELOCITY	Mass averaged velocity.
UDF_OSI_MAV_PRESSURE	Mass averaged pressure.
UDF_OSI_MAV_TOTAL_PRESSURE	Mass averaged total pressure.
UDF_OSI_MAV_TEMPERATURE	Mass averaged temperature.
UDF_OSI_MAV_SPECIES	Mass averaged species.
UDF_OSI_MAV_EDDY_VISCOSITY	Mass averaged viscosity.
UDF_OSI_MAV_KINETIC_ENERGY	Mass averaged kinetic energy.
UDF_OSI_MAV_EDDY_FREQUENCY	Mass averaged eddy frequency.
UDF_OSI_VEST	Average viscoelastic stress.
UDF_OSI_MAV_VEST	Mass averaged viscoelastic stress.
UDF_OSI_MFAV_VELOCITY	Mass flux average of velocity.
UDF_OSI_MFAV_PRESSURE	Mass flux average of pressure.
UDF_OSI_MFAV_TOTAL_PRESSURE	Mass flux average of total pressure.
UDF_OSI_MFAV_TEMPERATURE	Mass flux average of temperature.
UDF_OSI_MFAV_SPECIES	Mass flux average of species.
UDF_OSI_MFAV_EDDY_VISCOSITY	Mass flux average of turbulence eddy viscosity.
UDF_OSI_MFAV_KINETIC_ENERGY	Mass flux average of turbulence kinetic energy.
UDF_OSI_MFAV_EDDY_FREQUENCY	Mass flux average of turbulent eddy frequency.
UDF_OSI_MFAV_VEST	Mass flux average of viscoelastic stress.
UDF_OSI_BULK_TEMPERATURE	Bulk temperature.
UDF_OSI_PARTIAL_VOLUME	Partial volume.
UDF_OSI_WALL_SHEAR_STRESS	Wall shear stress.
UDF_OSI_CURR_AREA	Current total area.

UDF_OSI_CURR_MASS_FLUX Current total mass flux.

UDF_OSI_CURR_MOMENTUM_FLUX Current total momentum flux.

UDF_OSI_CURR_TRACTION Current total traction.

UDF_OSI_CURR_MOMENT Current total moment.

UDF_OSI_CURR_CONVECTIVE_TEMPERATURE Current total convective temperature flux.

UDF_OSI_CURR_HEAT_FLUX Current total heat flux.

UDF_OSI_CURR_CONVECTIVE_SPECIES Current total convective species flux.

UDF_OSI_CURR_SPECIES_FLUX Current total species flux.

UDF_OSI_CURR_VELOCITY Current average velocity.

UDF_OSI_CURR_PRESSURE Current average pressure.

UDF_OSI_CURR_TOTAL_PRESSURE Current average total pressure.

UDF_OSI_CURR_TEMPERATURE Current average temperature.

UDF_OSI_CURR_SPECIES Current average species.

UDF_OSI_CURR_EDDY_VISCOSITY Current average turbulence eddy viscosity.

UDF_OSI_CURR_KINETIC_ENERGY Current average turbulence kinetic energy.

UDF_OSI_CURR_EDDY_FREQUENCY Current average turbulence eddy frequency.

UDF_OSI_CURR_MESH_DISPLACEMENT Current average mesh displacement.

UDF_OSI_CURR_MESH_VELOCITY Current average mesh velocity.

UDF_OSI_CURR_TURBULENCE_YPLUS Current average y-plus of first node.

UDF_OSI_CURR_FILM_COEFFICIENT Current average heat transfer film coefficient.

UDF_OSI_CURR_AVE_DENSITY Current average density.

UDF_OSI_CURR_MAV_VELOCITY Current mass averaged velocity.

UDF_OSI_CURR_MAV_PRESSURE Current mass averaged pressure.

UDF_OSI_CURR_MAV_TOTAL_PRESSURE Current mass averaged total pressure.

UDF_OSI_CURR_MAV_TEMPERATURE Current mass averaged temperature.

UDF_OSI_CURR_MAV_SPECIES_FLUX Current mass averaged species flux.

UDF_OSI_CURR_MAV_EDDY_VISCOSITY Current mass averaged eddy viscosity.

UDF_OSI_CURR_MAV_KINETIC_ENERGY Current mass averaged kinetic energy.

UDF_OSI_CURR_MAV_EDDY_FREQUENCY Current mass averaged eddy frequency.

UDF_OSI_CURR_VEST Current average viscoelastic stress.

UDF_OSI_CURR_MAV_VEST Current mass averaged viscoelastic stress.

UDF_OSI_CURR_MFAV_VELOCITY Current mass flux average of velocity.

UDF_OSI_CURR_MFAV_PRESSURE Current mass flux average of pressure.

UDF_OSI_CURR_MFAV_TOTAL_PRESSURE Current mass flux average of total pressure.

UDF_OSI_CURR_MFAV_TEMPERATURE Current mass flux average of temperature.

UDF_OSI_CURR_MFAV_SPECIES Current mass flux average of species.

UDF_OSI_CURR_MFAV_EDDY_VISCOSITY Current mass flux average of turbulence eddy viscosity.

UDF_OSI_CURR_MFAV_KINETIC_ENERGY Current mass flux average of turbulence kinetic energy.

UDF_OSI_CURR_MFAV_EDDY_FREQUENCY Current mass flux average of turbulence eddy frequency.

UDF_OSI_CURR_MFAV_VEST Current mass flux average of viscoelastic stress.

UDF_OSI_CURR_BULK_TEMPERATURE Current bulk temperature.

UDF_OSI_CURR_PARTIAL_VOLUME Current partial volume.

UDF_OSI_CURR_WALL_SHEAR_STRESS Current wall shear stress.

Return Value

osiData *(string)*

Pointer to one-dimensional real array of the requested data. The dimension of the array depends on `dataName` as follows:

dataName	Array dimension
UDF_OSI_AREA	1
UDF_OSI_MASS_FLUX	1
UDF_OSI_MOMENTUM_FLUX	3
UDF_OSI_TRACTION	3
UDF_OSI_MOMENT	3
UDF_OSI_CONVECTIVE_TEMPERATURE	1

dataName	Array dimension
UDF_OSI_HEAT_FLUX	1
UDF_OSI_CONVECTIVE_SPECIES	`udfGetNumSpecs()`
UDF_OSI_SPECIES_FLUX	`udfGetNumSpecs()`
UDF_OSI_VELOCITY	3
UDF_OSI_PRESSURE	1
UDF_OSI_TOTAL_PRESSURE	1
UDF_OSI_TEMPERATURE	1
UDF_OSI_SPECIES	`udfGetNumSpecs()`
UDF_OSI_EDDY_VISCOSITY	1
UDF_OSI_KINETIC_ENERGY	1
UDF_OSI_EDDY_FREQUENCY	1
UDF_OSI_MESH_DISPLACEMENT	3
UDF_OSI_MESH_VELOCITY	3
UDF_OSI_TURBULENCE_YPLUS	1
UDF_OSI_FILM_COEFFICIENT	1
UDF_OSI_AVE_DENSITY	1
UDF_OSI_MAV_VELOCITY	3
UDF_OSI_MAV_PRESSURE	1
UDF_OSI_MAV_TOTAL_PRESSURE	1
UDF_OSI_MAV_TEMPERATURE	1
UDF_OSI_MAV_SPECIES	`udfGetNumSpecs()`
UDF_OSI_MAV_EDDY_VISCOSITY	1
UDF_OSI_MAV_KINETIC_ENERGY	1
UDF_OSI_MAV_EDDY_FREQUENCY	1
UDF_OSI_VEST	6
UDF_OSI_MAV_VEST	6

dataName	Array dimension
UDF_OSI_MFAV_VELOCITY	3
UDF_OSI_MFAV_PRESSURE	1
UDF_OSI_MFAV_TOTAL_PRESSURE	1
UDF_OSI_MFAV_TEMPERATURE	1
UDF_OSI_MFAV_SPECIES	`udfGetNumSpecs()`
UDF_OSI_MFAV_EDDY_VISCOSITY	1
UDF_OSI_MFAV_KINETIC_ENERGY	1
UDF_OSI_MFAV_EDDY_FREQUENCY	1
UDF_OSI_MFAV_VEST	6
UDF_OSI_BULK_TEMPERATURE	1
UDF_OSI_PARTIAL_VOLUME	1
UDF_OSI_WALL_SHEAR_STRESS	3
UDF_OSI_CURR_AREA	1
UDF_OSI_CURR_MASS_FLUX	1
UDF_OSI_CURR_MOMENTUM_FLUX	3
UDF_OSI_CURR_TRACTION	3
UDF_OSI_CURR_MOMENT	3
UDF_OSI_CURR_CONVECTIVE_TEMPERATU	1
UDF_OSI_CURR_HEAT_FLUX	1
UDF_OSI_CURR_CONVECTIVE_SPECIES	`udfGetNumSpecs()`
UDF_OSI_CURR_SPECIES_FLUX	`udfGetNumSpecs()`
UDF_OSI_CURR_VELOCITY	3
UDF_OSI_CURR_PRESSURE	1
UDF_OSI_CURR_TOTAL_PRESSURE	1
UDF_OSI_CURR_TEMPERATURE	1
UDF_OSI_CURR_SPECIES	`udfGetNumSpecs()`

dataName	Array dimension
UDF_OSI_CURR_EDDY_VISCOSITY	1
UDF_OSI_CURR_KINETIC_ENERGY	1
UDF_OSI_CURR_EDDY_FREQUENCY	1
UDF_OSI_CURR_MESH_DISPLACEMENT	3
UDF_OSI_CURR_MESH_VELOCITY	3
UDF_OSI_CURR_TURBULENCE_YPLUS	1
UDF_OSI_CURR_FILM_COEFFICIENT	1
UDF_OSI_CURR_AVE_DENSITY	1
UDF_OSI_CURR_MAV_VELOCITY	3
UDF_OSI_CURR_MAV_PRESSURE	1
UDF_OSI_CURR_MAV_TOTAL_PRESSURE	1
UDF_OSI_CURR_MAV_TEMPERATURE	1
UDF_OSI_CURR_MAV_SPECIES_FLUX	`udfGetNumSpecs()`
UDF_OSI_CURR_MAV_EDDY_VISCOSITY	1
UDF_OSI_CURR_MAV_KINETIC_ENERGY	1
UDF_OSI_CURR_MAV_EDDY_FREQUENCY	1
UDF_OSI_CURR_VEST	6
UDF_OSI_CURR_MAV_VEST	6
UDF_OSI_CURR_MFAV_VELOCITY	3
UDF_OSI_CURR_MFAV_PRESSURE	1
UDF_OSI_CURR_MFAV_TOTAL_PRESSURE	1
UDF_OSI_CURR_MFAV_TEMPERATURE	1
UDF_OSI_CURR_MFAV_SPECIES	`udfGetNumSpecs()`
UDF_OSI_CURR_MFAV_EDDY_VISCOSITY	1
UDF_OSI_CURR_MFAV_KINETIC_ENERGY	1
UDF_OSI_CURR_MFAV_EDDY_FREQUENCY	1

dataName	Array dimension
UDF_OSI_CURR_MFAV_VEST	6
UDF_OSI_CURR_BULK_TEMPERATURE	1
UDF_OSI_CURR_PARTIAL_VOLUME	1
UDF_OSI_CURR_WALL_SHEAR_STRESS	3

Description

This routine returns data from an integrated surface output set. For example,

```
Real* osiData ;
Real area, x_mom_flux, y_mom_flux, z_mom_flux, convSpec ;
Integer specId, nSpecs ;
...
osiData = udfGetOsiData( udfHd, "car body", UDF_OSI_AREA ) ;
area = osiData[0] ;
...
osiData = udfGetOsiData( udfHd, "car body", UDF_OSI_MOMENTUM_FLUX ) ;
x_mom_flux = osiData[0] ;
y_mom_flux = osiData[1] ;
z_mom_flux = osiData[2] ;
...
nSpecs = udfGetNumSpecs( udfHd ) ;
osiData = udfGetOsiData( udfHd, "car body", UDF_OSI_CONVECTIVE_SPECIES ) ;
for ( specId = 0 ; specId < nSpecs ; specId++ ) {
 convSpec = osiData[specId] ;
 ...
)
```

All quantities are evaluated at the end of the previous time step, except for the "current" variables (UDF_OSI_CURR_MASS_FLUX, UDF_OSI_CURR_MOMENTUM_FLUX, and UDF_OSI_CURR_TRACTION). These are evaluated at the end of every flow stagger in order to provide the latest data.

Errors

- This routine expects a valid *udfHd*.
- *setName* must be a valid name.
- *dataName* must be one of the values given above.
- The problem must contain the equation system corresponding to the parameter being requested. For example, the problem must contain a heat equation in order for UDF_OSI_HEAT_FLUX to be available.

udfGetOssData()

Return the nodal statistics (minimum, maximum and standard deviation) from a surface output set.

Syntax

```
ossData = udfGetOssData( udfHd, setName, dataName ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

setName *(string)*
: Name of the `SURFACE_OUTPUT` set.

dataName *(integer)*
: Symbolic name of the requested data.

UDF_OSS_MIN_VELOCITY	Minimum velocity.
UDF_OSS_MIN_PRESSURE	Minimum pressure.
UDF_OSS_MIN_TOTAL_PRESSURE	Minimum total pressure.
UDF_OSS_MIN_TEMPERATURE	Minimum temperature.
UDF_OSS_MIN_SPECIES	Minimum species.
UDF_OSS_MIN_EDDY_VISCOSITY	Minimum turbulence eddy viscosity.
UDF_OSS_MIN_KINETIC_ENERGY	Minimum turbulence kinetic energy.
UDF_OSS_MIN_EDDY_FREQUENCY	Minimum turbulence eddy frequency.
UDF_OSS_MIN_MESH_DISPLACEMENT	Minimum mesh displacement.
UDF_OSS_MIN_MESH_VELOCITY	Minimum mesh velocity.
UDF_OSS_MIN_TURBULENCE_YPLUS	Minimum y-plus of first node.
UDF_OSS_MIN_FILM_COEFFICIENT	Minimum heat transfer film coefficient.
UDF_OSS_MIN_DENSITY	Minimum density.
UDF_OSS_MIN_VEST	Minimum viscoelastic stress.
UDF_OSS_MIN_WALL_SHEAR_STRESS	Minimum wall shear stress.
UDF_OSS_MAX_VELOCITY	Maximum velocity.

UDF_OSS_MAX_PRESSURE	Maximum pressure.
UDF_OSS_MAX_TOTAL_PRESSURE	Maximum total pressure.
UDF_OSS_MAX_TEMPERATURE	Maximum temperature.
UDF_OSS_MAX_SPECIES	Maximum species.
UDF_OSS_MAX_EDDY_VISCOSITY	Maximum turbulence eddy viscosity.
UDF_OSS_MAX_KINETIC_ENERGY	Maximum turbulence kinetic energy.
UDF_OSS_MAX_EDDY_FREQUENCY	Maximum turbulence eddy frequency.
UDF_OSS_MAX_MESH_DISPLACEMENT	Maximum mesh displacement.
UDF_OSS_MAX_MESH_VELOCITY	Maximum mesh velocity.
UDF_OSS_MAX_TURBULENCE_YPLUS	Maximum y-plus of first node.
UDF_OSS_MAX_FILM_COEFFICIENT	Maximum heat transfer film coefficient.
UDF_OSS_MAX_DENSITY	Maximum density.
UDF_OSS_MAX_VEST	Maximum viscoelastic stress.
UDF_OSS_MAX_WALL_SHEAR_STRESS	Maximum wall shear stress.
UDF_OSS_STD_VELOCITY	Standard deviation of velocity.
UDF_OSS_STD_PRESSURE	Standard deviation of pressure.
UDF_OSS_STD_TOTAL_PRESSURE	Standard deviation of total pressure.
UDF_OSS_STD_TEMPERATURE	Standard deviation of temperature.
UDF_OSS_STD_SPECIES	Standard deviation of species.
UDF_OSS_STD_EDDY_VISCOSITY	Standard deviation of turbulence eddy viscosity.
UDF_OSS_STD_KINETIC_ENERGY	Standard deviation of turbulence kinetic energy.
UDF_OSS_STD_EDDY_FREQUENCY	Standard deviation of turbulence eddy frequency.
UDF_OSS_STD_MESH_DISPLACEMENT	Standard deviation of mesh displacement.
UDF_OSS_STD_MESH_VELOCITY	Standard deviation of mesh velocity.
UDF_OSS_STD_TURBULENCE_YPLUS	Standard deviation of y-plus of first node.
UDF_OSS_STD_FILM_COEFFICIENT	Standard deviation of heat transfer film coefficient.
UDF_OSS_STD_DENSITY	Standard deviation of density.

UDF_OSS_STD_VEST	Standard deviation of viscoelastic stress.
UDF_OSS_STD_WALL_SHEAR_STRESS	Standard deviation of wall shear stress.
UDF_OSS_UNI_VELOCITY	Uniformity index of velocity.
UDF_OSS_UNI_PRESSURE	Uniformity index of pressure.
UDF_OSS_UNI_TOTAL_PRESSURE	Uniformity index of total pressure.
UDF_OSS_UNI_TEMPERATURE	Uniformity index of temperature.
UDF_OSS_UNI_SPECIES	Uniformity index of species.
UDF_OSS_UNI_EDDY_VISCOSITY	Uniformity index of turbulence eddy viscosity.
UDF_OSS_UNI_KINETIC_ENERGY	Uniformity index of turbulence kinetic energy.
UDF_OSS_UNI_EDDY_FREQUENCY	Uniformity index of turbulence eddy frequency.
UDF_OSS_UNI_MESH_DISPLACEMENT	Uniformity index of mesh displacement.
UDF_OSS_UNI_MESH_VELOCITY	Uniformity index of mesh velocity.
UDF_OSS_UNI_TURBULENCE_YPLUS	Uniformity index of y-plus of first node.
UDF_OSS_UNI_FILM_COEFFICIENT	Uniformity index of heat transfer film coefficient.
UDF_OSS_UNI_DENSITY	Uniformity index of density.
UDF_OSS_UNI_VEST	Uniformity index of viscoelastic stress.
UDF_OSS_UNI_WALL_SHEAR_STRESS	Uniformity index of wall shear stress.
UDF_OSS_CURR_MIN_VELOCITY	Current minimum velocity.
UDF_OSS_CURR_MIN_PRESSURE	Current minimum pressure.
UDF_OSS_CURR_MIN_TOTAL_PRESSURE	Current minimum total pressure.
UDF_OSS_CURR_MIN_TEMPERATURE	Current minimum temperature.
UDF_OSS_CURR_MIN_SPECIES	Current minimum species.
UDF_OSS_CURR_MIN_EDDY_VISCOSITY	Current minimum turbulence eddy viscosity.
UDF_OSS_CURR_MIN_KINETIC_ENERGY	Current minimum turbulence kinetic energy.
UDF_OSS_CURR_MIN_EDDY_FREQUENCY	Current minimum turbulence eddy frequency.
UDF_OSS_CURR_MIN_MESH_DISPLACEMENT	Current minimum mesh displacement.
UDF_OSS_CURR_MIN_MESH_VELOCITY	Current minimum mesh velocity.

UDF_OSS_CURR_MIN_TURBULENCE_YPLUS Current minimum y-plus of first node.

UDF_OSS_CURR_MIN_FILM_COEFFICIENT Current minimum heat transfer film coefficient.

UDF_OSS_CURR_MIN_DENSITY Current minimum density.

UDF_OSS_CURR_MIN_VEST Current minimum viscoelastic stress.

UDF_OSS_CURR_MIN_WALL_SHEAR_STRESS Current minimum wall shear stress.

UDF_OSS_CURR_MAX_VELOCITY Current maximum velocity.

UDF_OSS_CURR_MAX_PRESSURE Current maximum pressure.

UDF_OSS_CURR_MAX_TOTAL_PRESSURE Current maximum total pressure.

UDF_OSS_CURR_MAX_TEMPERATURE Current maximum temperature.

UDF_OSS_CURR_MAX_SPECIES Current maximum species.

UDF_OSS_CURR_MAX_EDDY_VISCOSITY Current maximum turbulence eddy viscosity.

UDF_OSS_CURR_MAX_KINETIC_ENERGY Current maximum turbulence kinetic energy.

UDF_OSS_CURR_MAX_EDDY_FREQUENCY Current maximum turbulence eddy frequency.

UDF_OSS_CURR_MAX_MESH_DISPLACEMENT Current maximum mesh displacement.

UDF_OSS_CURR_MAX_MESH_VELOCITY Current maximum mesh velocity.

UDF_OSS_CURR_MAX_TURBULENCE_YPLUS Current maximum y-plus of first node.

UDF_OSS_CURR_MAX_FILM_COEFFICIENT Current maximum heat transfer film coefficient.

UDF_OSS_CURR_MAX_DENSITY Current maximum density.

UDF_OSS_CURR_MAX_VEST Current maximum viscoelastic stress.

UDF_OSS_CURR_MAX_WALL_SHEAR_STRESS Current maximum wall shear stress.

UDF_OSS_CURR_STD_VELOCITY Current standard deviation of velocity.

UDF_OSS_CURR_STD_PRESSURE Current standard deviation of pressure.

UDF_OSS_CURR_STD_TOTAL_PRESSURE Current standard deviation of total pressure.

UDF_OSS_CURR_STD_TEMPERATURE Current standard deviation of temperature.

UDF_OSS_CURR_STD_SPECIES Current standard deviation of species.

UDF_OSS_CURR_STD_EDDY_VISCOSITY Current standard deviation of turbulence eddy viscosity.

UDF_OSS_CURR_STD_KINETIC_ENERGY Current standard deviation of turbulence kinetic energy.

UDF_OSS_CURR_STD_EDDY_FREQUENCY Current standard deviation of turbulence eddy frequency.

UDF_OSS_CURR_STD_MESH_DISPLACEMENT Current standard deviation of mesh displacement.

UDF_OSS_CURR_STD_MESH_VELOCITY Current standard deviation of mesh velocity.

UDF_OSS_CURR_STD_TURBULENCE_YPLUS Current standard deviation of y-plus of first node.

UDF_OSS_CURR_STD_FILM_COEFFICIENT Current standard deviation of heat transfer film coefficient.

UDF_OSS_CURR_STD_DENSITY Current standard deviation of density.

UDF_OSS_CURR_STD_VEST Current standard deviation of viscoelastic stress.

UDF_OSS_CURR_STD_WALL_SHEAR_STRESS Current standard deviation of wall shear stress.

UDF_OSS_CURR_UNI_VELOCITY Current uniformity index of velocity.

UDF_OSS_CURR_UNI_PRESSURE Current uniformity index of pressure.

UDF_OSS_CURR_UNI_TOTAL_PRESSURE Current uniformity index of total pressure.

UDF_OSS_CURR_UNI_TEMPERATURE Current uniformity index of temperature.

UDF_OSS_CURR_UNI_SPECIES Current uniformity index of species.

UDF_OSS_CURR_UNI_EDDY_VISCOSITY Current uniformity index of turbulence eddy viscosity.

UDF_OSS_CURR_UNI_KINETIC_ENERGY Current uniformity index of turbulence kinetic energy.

UDF_OSS_CURR_UNI_EDDY_FREQUENCY Current uniformity index of turbulence eddy frequency.

UDF_OSS_CURR_UNI_MESH_DISPLACEMENT Current uniformity index of mesh displacement.

UDF_OSS_CURR_UNI_MESH_VELOCITY Current uniformity index of mesh velocity.

UDF_OSS_CURR_UNI_TURBULENCE_YPLUS Current uniformity index of y-plus of first node.

UDF_OSS_CURR_UNI_FILM_COEFFICIENT Current uniformity index of heat transfer film coefficient.

UDF_OSS_CURR_UNI_DENSITY Current uniformity index of density.

UDF_OSS_CURR_UNI_VEST Current uniformity index of viscoelastic stress.

UDF_OSS_CURR_UNI_WALL_SHEAR_STRESS Current uniformity index of wall shear stress.

Return Value

osiData *(real)*

Pointer to one dimensional real array of the requested data. The dimension of the array depends on *dataName* as follows:

dataName	Array dimension
UDF_OSS_MIN_VELOCITY	3
UDF_OSS_MIN_PRESSURE	1
UDF_OSS_MIN_TOTAL_PRESSURE	1
UDF_OSS_MIN_TEMPERATURE	1
UDF_OSS_MIN_SPECIES	`udfGetNumSpecs()`
UDF_OSS_MIN_EDDY_VISCOSITY	1
UDF_OSS_MIN_KINETIC_ENERGY	1
UDF_OSS_MIN_EDDY_FREQUENCY	1
UDF_OSS_MIN_MESH_DISPLACEMENT	3
UDF_OSS_MIN_MESH_VELOCITY	3
UDF_OSS_MIN_TURBULENCE_YPLUS	1
UDF_OSS_MIN_FILM_COEFFICIENT	1
UDF_OSS_MIN_DENSITY	1
UDF_OSS_MIN_VEST	6
UDF_OSS_MIN_WALL_SHEAR_STRESS	3
UDF_OSS_MAX_VELOCITY	3
UDF_OSS_MAX_PRESSURE	1
UDF_OSS_MAX_TOTAL_PRESSURE	1
UDF_OSS_MAX_TEMPERATURE	1
UDF_OSS_MAX_SPECIES	`udfGetNumSpecs()`
UDF_OSS_MAX_EDDY_VISCOSITY	1
UDF_OSS_MAX_KINETIC_ENERGY	1
UDF_OSS_MAX_EDDY_FREQUENCY	1
UDF_OSS_MAX_MESH_DISPLACEMENT	3
UDF_OSS_MAX_MESH_VELOCITY	3

dataName	Array dimension
UDF_OSS_MAX_TURBULENCE_YPLUS	1
UDF_OSS_MAX_FILM_COEFFICIENT	1
UDF_OSS_MAX_DENSITY	1
UDF_OSS_MAX_VEST	1
UDF_OSS_MAX_WALL_SHEAR_STRESS	3
UDF_OSS_STD_VELOCITY	3
UDF_OSS_STD_PRESSURE	1
UDF_OSS_STD_TOTAL_PRESSURE	1
UDF_OSS_STD_TEMPERATURE	1
UDF_OSS_STD_SPECIES	`udfGetNumSpecs()`
UDF_OSS_STD_EDDY_VISCOSITY	1
UDF_OSS_STD_KINETIC_ENERGY	1
UDF_OSS_STD_EDDY_FREQUENCY	1
UDF_OSS_STD_MESH_DISPLACEMENT	3
UDF_OSS_STD_MESH_VELOCITY	3
UDF_OSS_STD_TURBULENCE_YPLUS	1
UDF_OSS_STD_FILM_COEFFICIENT	1
UDF_OSS_STD_DENSITY	1
UDF_OSS_STD_VEST	6
UDF_OSS_STD_WALL_SHEAR_STRESS	3
UDF_OSS_UNI_VELOCITY	3
UDF_OSS_UNI_PRESSURE	1
UDF_OSS_UNI_TOTAL_PRESSURE	1
UDF_OSS_UNI_TEMPERATURE	1
UDF_OSS_UNI_SPECIES	`udfGetNumSpecs()`
UDF_OSS_UNI_EDDY_VISCOSITY	1

dataName	Array dimension
UDF_OSS_UNI_KINETIC_ENERGY	1
UDF_OSS_UNI_EDDY_FREQUENCY	1
UDF_OSS_UNI_MESH_DISPLACEMENT	3
UDF_OSS_UNI_MESH_VELOCITY	3
UDF_OSS_UNI_TURBULENCE_YPLUS	1
UDF_OSS_UNI_FILM_COEFFICIENT	1
UDF_OSS_UNI_DENSITY	1
UDF_OSS_UNI_VEST	6
UDF_OSS_UNI_WALL_SHEAR_STRESS	3
UDF_OSS_CURR_MIN_VELOCITY	3
UDF_OSS_CURR_MIN_PRESSURE	1
UDF_OSS_CURR_MIN_TOTAL_PRESSURE	1
UDF_OSS_CURR_MIN_TEMPERATURE	1
UDF_OSS_CURR_MIN_SPECIES	`udfGetNumSpecs()`
UDF_OSS_CURR_MIN_EDDY_VISCOSITY	1
UDF_OSS_CURR_MIN_KINETIC_ENERGY	1
UDF_OSS_CURR_MIN_EDDY_FREQUENCY	1
UDF_OSS_CURR_MIN_MESH_DISPLACEMENT	3
UDF_OSS_CURR_MIN_MESH_VELOCITY	3
UDF_OSS_CURR_MIN_TURBULENCE_YPLUS	1
UDF_OSS_CURR_MIN_FILM_COEFFICIENT	1
UDF_OSS_CURR_MIN_DENSITY	1
UDF_OSS_CURR_MIN_VEST	6
UDF_OSS_CURR_MIN_WALL_SHEAR_STRESS	3
UDF_OSS_CURR_MAX_VELOCITY	3
UDF_OSS_CURR_MAX_PRESSURE	1

dataName	Array dimension
UDF_OSS_CURR_MAX_TOTAL_PRESSURE	1
UDF_OSS_CURR_MAX_TEMPERATURE	1
UDF_OSS_CURR_MAX_SPECIES	`udfGetNumSpecs()`
UDF_OSS_CURR_MAX_EDDY_VISCOSITY	1
UDF_OSS_CURR_MAX_KINETIC_ENERGY	1
UDF_OSS_CURR_MAX_EDDY_FREQUENCY	1
UDF_OSS_CURR_MAX_MESH_DISPLACEMENT	3
UDF_OSS_CURR_MAX_MESH_VELOCITY	3
UDF_OSS_CURR_MAX_TURBULENCE_YPLUS	1
UDF_OSS_CURR_MAX_FILM_COEFFICIENT	1
UDF_OSS_CURR_MAX_DENSITY	1
UDF_OSS_CURR_MAX_VEST	6
UDF_OSS_CURR_MAX_WALL_SHEAR_STRESS	3
UDF_OSS_CURR_STD_VELOCITY	3
UDF_OSS_CURR_STD_PRESSURE	1
UDF_OSS_CURR_STD_TOTAL_PRESSURE	1
UDF_OSS_CURR_STD_TEMPERATURE	1
UDF_OSS_CURR_STD_SPECIES	`udfGetNumSpecs()`
UDF_OSS_CURR_STD_EDDY_VISCOSITY	1
UDF_OSS_CURR_STD_KINETIC_ENERGY	1
UDF_OSS_CURR_STD_EDDY_FREQUENCY	1
UDF_OSS_CURR_STD_MESH_DISPLACEMENT	3
UDF_OSS_CURR_STD_MESH_VELOCITY	3
UDF_OSS_CURR_STD_TURBULENCE_YPLUS	1
UDF_OSS_CURR_STD_FILM_COEFFICIENT	1
UDF_OSS_CURR_STD_DENSITY	1

dataName	Array dimension
UDF_OSS_CURR_STD_VEST	6
UDF_OSS_CURR_STD_WALL_SHEAR_STRESS	3
UDF_OSS_CURR_UNI_VELOCITY	3
UDF_OSS_CURR_UNI_PRESSURE	1
UDF_OSS_CURR_UNI_TOTAL_PRESSURE	1
UDF_OSS_CURR_UNI_TEMPERATURE	1
UDF_OSS_CURR_UNI_SPECIES	`udfGetNumSpecs()`
UDF_OSS_CURR_UNI_EDDY_VISCOSITY	1
UDF_OSS_CURR_UNI_KINETIC_ENERGY	1
UDF_OSS_CURR_UNI_EDDY_FREQUENCY	1
UDF_OSS_CURR_UNI_MESH_DISPLACEMENT	3
UDF_OSS_CURR_UNI_MESH_VELOCITY	3
UDF_OSS_CURR_UNI_TURBULENCE_YPLUS	1
UDF_OSS_CURR_UNI_FILM_COEFFICIENT	1
UDF_OSS_CURR_UNI_DENSITY	1
UDF_OSS_CURR_UNI_VEST	6
UDF_OSS_CURR_UNI_WALL_SHEAR_STRESS	3

Description

This routine returns statistical data from a surface output set. For example,

```
Real* ossData ;
Real max_pres, min_x_mesh_disp, min_y_mesh_disp, min_z_mesh_disp, convSpec ;
Integer specId, nSpecs ;
...
ossData = udfGetOssData( udfHd, "car body", UDF_OSS_MAX_PRESSURE ) ;
max_pres = ossData[0] ;
...
ossData = udfGetOssData( udfHd, "car body", UDF_OSS_MIN_MESH_DISPLACEMENT ) ;
min_x_mesh_disp = ossData[0] ;
min_y_mesh_disp = ossData[1] ;
min_z_mesh_disp = ossData[2] ;
...
nSpecs = udfGetNumSpecs( udfHd ) ;
ossData = udfGetOssData( udfHd, "car body", UDF_OSS_STD_SPECIES ) ;
```

```
for ( specId = 0 ; specId < nSpecs ; specId++ ) {
 convSpec = osiData[specId] ;
 ...
)
```

This routine is used to compute the statistical data from a surface output set. The data is computed by extracting the solution values from all nodes in the surface set, then performing the statistical operation to produce in the maximum, minimum and standard deviation values. When extracting the UDF_OSS_CURR_ variables, the returned values reflect the solution updates at each nonlinear iteration. When extracting the UDF_OSS_ variables, the values from the previous timestep are returned. In the above example, maximum pressure, minimum mesh displacement and standard deviation of species are extracted.

Errors

- This routine expects a valid *udfHd*.
- *setName* must be a valid name.
- *dataName* must be one of the values given above.
- The problem must contain the equation system corresponding to the parameter being requested. For example, the problem must contain a heat equation in order for UDF_OSS_STD_TEMPERATURE to be available.

udfGetOriData()

Return data from an integrated radiation surface output set.

Syntax

```
oriData = udfGetOriData( udfHd, setName, dataName ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

setName (string)
: Name of the RADIATION_SURFACE set.

dataName (integer)
: Symbolic name of the requested data.

UDF_ORI_AREA	Total area.
UDF_ORI_HEAT_FLUX	Total radiation heat flux.
UDF_ORI_TEMPERATURE	Average temperature.
UDF_ORI_MEAN_RADIANT_TEMPERATURE	Mean radiant temperature.

Return Value

oriData *(real)*
: Pointer to one-dimensional real array of the requested data. The dimension of the array is 1 for all values of *dataName*.

Description

This routine returns data from an integrated radiation surface output set. For example,

```
Real* oriData ;
Real area, heat_flux ;
...
oriData = udfGetOriData( udfHd, "car body", UDF_ORI_AREA ) ;
area = oriData[0] ;
...
oriData = udfGetOriData( udfHd, "car body", UDF_ORI_HEAT_FLUX ) ;
heat_flux = oriData[0] ;
```

Errors

- This routine expects a valid *udfHd*.
- *setName* must be a valid name.
- *dataName* must be one of the values given above.

- The problem must contain the equation system corresponding to the parameter being requested. In particular, the problem must contain a radiation equation in order for the heat flux or temperature quantities to be available.

udfGetOeiData()

Return data from an integrated element output set.

Syntax

```
oeiData = udfGetOeiData( udfHd, setName, dataName ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

setName (string)
: Name of the ELEMENT_OUTPUT set.

dataName (integer)
: Symbolic name of the requested data.

UDF_OEI_VOLUME	Total volume.
UDF_OEI_VELOCITY	Average velocity.
UDF_OEI_ACCELERATION	Average acceleration.
UDF_OEI_PRESSURE	Average pressure.
UDF_OEI_TOTAL_PRESSURE	Average total pressure.
UDF_OEI_TEMPERATURE	Average temperature.
UDF_OEI_SPECIES	Average species.
UDF_OEI_EDDY_VISCOSITY	Average turbulence eddy viscosity.
UDF_OEI_KINETIC_ENERGY	Average turbulence kinetic energy.
UDF_OEI_EDDY_FREQUENCY	Average turbulence eddy frequency.
UDF_OEI_GRAD_VELOCITY	Average gradient of velocity.
UDF_OEI_GRAD_PRESSURE	Average gradient of pressure.
UDF_OEI_GRAD_TEMPERATURE	Average gradient of temperature.
UDF_OEI_GRAD_SPECIES	Average gradient of species.
UDF_OEI_STRESS	Average Cauchy stress.
UDF_OEI_HEAT_FLUX	Average heat flux.

UDF_OEI_SPECIES_FLUX Average species flux.

UDF_OEI_MESH_DISPLACEMENT Average mesh displacement.

UDF_OEI_MESH_VELOCITY Average mesh velocity.

UDF_OEI_USER_OUTPUT User defined output.

Return Value

oeiData *(real)*

Pointer to one or two-dimensional real array of the requested data. The dimensions of the array depend on `dataName` as follows. If the second (slower) dimension is one, then the array may be treated as a one-dimensional array. The x, y and z components of the gradient for gradient quantities are always contained in the second dimension. The six components of the Cauchy stress are, in order, xx, yy, zz, xy, yz and zx.

dataName	**First dimension**
UDF_OEI_VOLUME	1
UDF_OEI_VELOCITY	3
UDF_OEI_ACCELERATION	3
UDF_OEI_PRESSURE	1
UDF_OEI_TOTAL_PRESSURE	1
UDF_OEI_TEMPERATURE	1
UDF_OEI_SPECIES	`udfGetNumSpecs()`
UDF_OEI_EDDY_VISCOSITY	1
UDF_OEI_KINETIC_ENERGY	1
UDF_OEI_EDDY_FREQUENCY	1
UDF_OEI_GRAD_VELOCITY	3
UDF_OEI_GRAD_PRESSURE	1
UDF_OEI_GRAD_TEMPERATURE	1
UDF_OEI_GRAD_SPECIES	`udfGetNumSpecs()`
UDF_OEI_STRESS	6
UDF_OEI_HEAT_FLUX	3

dataName	First dimension
UDF_OEI_SPECIES_FLUX	`udfGetNumSpecs()`
UDF_OEI_MESH_DISPLACEMENT	3
UDF_OEI_MESH_VELOCITY	3
UDF_OEI_USER_OUTPUT	Determined by you

Description

This routine returns data from an output element integrated set. For example,

```
Real* oeiData ;
Real volume, ave_x_vel, ave_y_vel, ave_z_vel, spec_x, spec_y, spec_z ;
Real u_x, v_x, w_x, u_y, v_y, w_y, u_z, v_z, w_z ;
Integer specId, nSpecs ;
...
oeiData = udfGetOeiData( udfHd, "probe 1", UDF_OEI_VOLUME ) ;
volume = oeiData[0] ;
...
oeiData = udfGetOeiData( udfHd, "probe 1", UDF_OEI_VELOCITY ) ;
ave_x_vel = oeiData[0] ;
ave_y_vel = oeiData[1] ;
ave_z_vel = oeiData[2] ;
...
nSpecs = udfGetNumSpecs( udfHd ) ;
oeiData = udfGetOeiData( udfHd, "probe 1", UDF_OEI_GRAD_SPECIES )
for ( specId = 0 ; specId < nSpecs ; specId++ ) {
   spec_x = oeiData[0*nSpecs+specId] ;
   spec_y = oeiData[1*nSpecs+specId] ;
   spec_z = oeiData[2*nSpecs+specId] ;
   ...
}
...
```

```
oeiData = udfGetOeiData( udfHd, "probe 1", UDF_OEI_GRAD_VELOCITY )

u_x = oeiData[0] ;

v_x = oeiData[1] ;

w_x = oeiData[2] ;

u_y = oeiData[3] ;

v_y = oeiData[4] ;

w_y = oeiData[5] ;

u_z = oeiData[6] ;

v_z = oeiData[7] ;

w_z = oeiData[8] ;

...
```

Errors

- This routine expects a valid *udfHd*.
- *setName* must be a valid name.
- *dataName* must be one of the values given above.
- The problem must contain the equation system corresponding to the parameter being requested. For example, the problem must contain a heat equation in order for UDF_OEI_HEAT_FLUX to be available.

udfGetFanData()

Return data of a fan component.

Syntax

```
fanData = udfGetFanData( udfHd, setName, dataName ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

setName *(string)*
: Name of the FAN_COMPONENT set.

dataName *(integer)*
: Symbolic name of the requested data.

UDF_FAN_AREA	Inlet area.
UDF_FAN_MASS_FLUX	Inlet mass flux.
UDF_FAN_AVERAGE_VELOCITY	Mass-averaged inlet velocity.

Return Value

fanData*(real)*
: Pointer to one-dimensional real array of the requested data. The dimension of the array is one for all values of *dataName*.

Description

This routine returns data from a specified fan component. For example,

```
Real* fanData ;
Real area, mass, velocity;
...
fanData = udfGetFanData( udfHd, "fan", UDF_FAN_AREA ) ;
area = fanData[0] ;
fanData = udfGetFanData( udfHd, "fan", UDF_FAN_MASS_FLUX ) ;
mass = fanData[0] ;
fanData = udfGetFanData( udfHd, "fan", UDF_FAN_AVERAGE_VELOCITY ) ;
velocity = fanData[0] ;
```

Errors

- This routine expects a valid *udfHd*.
- *setName* must be a valid name.
- *dataName* must be one of the values given above.

udfGetHecData()

Return data of a heat exchanger component.

Syntax

```
hecData = udfGetHecData( udfHd, setName, dataName ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

setName *(string)*
: Name of the `HEAT_EXCHANGER_COMPONENT` set.

dataName *(integer)*
: Symbolic name of the requested data.

UDF_HEC_AREA	Inlet area.
UDF_HEC_MASS_FLUX	Inlet mass flux.
UDF_HEC_AVERAGE_VELOCITY	Mass-averaged inlet velocity.
UDF_HEC_AVERAGE_TEMPERATURE	Mass-averaged inlet temperature.
UDF_HEC_COOLANT_TEMPERATURE	Top water temperature.
UDF_HEC_COOLANT_HEAT	Top water heat rejection.

Return Value

hecData *(real)*
: Pointer to one-dimensional real array of the requested data. The dimension of the array is one for all values of *dataName*.

Description

This routine returns data from a specified heat exchanger component. For example,

```
Real* hecData ;
Real area, mass, velocity, temp, topTemp;
...
hecData = udfGetHecData( udfHd, "radiator 1", UDF_HEC_AREA ) ;
area = hecData[0] ;
hecData = udfGetHecData( udfHd, "radiator 1", UDF_FAN_MASS_FLUX ) ;
mass = hecData[0] ;
hecData = udfGetHecData( udfHd, "radiator 1", UDF_FAN_AVERAGE_VELOCITY ) ;
velocity = hecData[0] ;
hecData = udfGetHecData( udfHd, "radiator 1", UDF_FAN_AVERAGE_TEMPERATURE ) ;
```

```
temp = hecData[0] ;
hecData = udfGetHecData( udfHd, "radiator 1", UDF_FAN_COOLANT_TEMPERATURE ) ;
topTemp = hecData[0] ;
```

Errors

- This routine expects a valid *udfHd*.
- *setName* must be a valid name.
- *dataName* must be one of the values given above.

udfGetEDEMData()

Return the data for a variable.

Syntax

```
data = udfGetEDEMData( udfHd, dataName ) ;
```

Type

AcuSolve-EDEM coupling user-defined drag, lift and torque models.

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName *(integer)*
: Symbolic name of the requested data.

UDF_EDEM_FLOW_DENSITY	Flow density.
UDF_EDEM_FLOW_VISCOSITY	Flow viscosity.
UDF_EDEM_FLOW_POROSITY	Flow porosity.
UDF_EDEM_FLOW_VELOCITY	Flow velocity.
UDF_EDEM_FLOW_VELOCITY_CURL	Flow velocity curl.
UDF_EDEM_PARTICLE_VOLUME	Particle volume.
UDF_EDEM_PARTICLE_ANGLE	Particle angle.
UDF_EDEM_PARTICLE_POSITION	Particle position.
UDF_EDEM_PARTICLE_VELOCITY	Particle velocity.
UDF_EDEM_PARTICLE_VELOCITY_CURL	Particle velocity curl.
UDF_EDEM_DRAG_COEF	Drag coefficient.
UDF_EDEM_PAXIS_VELOCITY	pAxis * slipVel * slipVel.
UDF_EDEM_DRAG	Drag.
UDF_EDEM_LIFT	Lift.
UDF_EDEM_PARTICLE_QUATERNION	Particle quaternion.

Return Value

data *(Real*)*

Pointer to one, two, or three dimensional real array of the requested data. The dimensions of the array depend on *dataName* as follows. If the third (slowest) dimension of the array is equal to one, then the array may be treated as two dimensional. If the second dimension is likewise one, then the array is one dimensional.

Table 6:

dataName	First Dimension	Second Dimension
UDF_EDEM_FLOW_DENSITY	nItems	1
UDF_EDEM_FLOW_VISCOSITY	nItems	1
UDF_EDEM_FLOW_POROSITY	nItems	1
UDF_EDEM_FLOW_VELOCITY	nItems	3
UDF_EDEM_FLOW_VELOCITY_C	nItems	3
UDF_EDEM_PARTICLE_VOLUME	nItems	1
UDF_EDEM_PARTICLE_ANGLE	nItems	1
UDF_EDEM_PARTICLE_POSITIO	nItems	3
UDF_EDEM_PARTICLE_VELOCIT	nItems	3
UDF_EDEM_PARTICLE_VELOCIT	nItems	3
UDF_EDEM_DRAG_COEF	nItems	1
UDF_EDEM_PAXIS_VELOCITY	nItems	3
UDF_EDEM_DRAG	nItems	3
UDF_EDEM_LIFT	nItems	3
UDF_EDEM_PARTICLE_QUATERN	nItems	4

Description

This routine returns the requested solution data. For example,

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

/*============================================================================
 *
 * Prototype the function
 *
```

```
 *=============================================================================
 */

UDF_PROTOTYPE( usrEDEMDrag ) ;

/*=============================================================================
 *
 * "usrEDEMDrag":  calculate EDEM particle drags
 *
 *  Arguments:
 *    udfHd     - opaque handle for accessing information
 *    outVec    - output vector
 *    nItems    - number of items in outVec (=1 in this case)
 *    vecDim    - vector dimension of outVec (=1 in this case)
 *
 *  Input file parameters:
 *    user_values = { ErgunA, ErgunB }
 *    user_strings = { "" }
 *
 *=============================================================================
 */

Void usrEDEMDrag(           UdfHd  udfHd,
                            Real*  outVec,
                            Integer         nItems,
                            Integer         vecDim )
{

    Real beta ;
    Real betaArea ;
    Real dens ;
    Real visc ;
    Real vol ;
    Real diam ;
    Real* scalar ;
    Real* flowVel ;
    Real* particleVel ;
    Real slipVelMag ;
    Real slipVel[3] ;
    Real poros ;
    Real CD ;
    Real Re ;
    Real ErgunA ;
    Real ErgunB ;
    Real*       usrVals ;
    String*     usrStrs ;
    Real carrierDens ;

/*-----------------------------------------------------------------------------
 * Get the user data
 *-----------------------------------------------------------------------------
 */
    udfCheckNumUsrVals(   udfHd,  2  ) ;
    udfCheckNumUsrStrs(   udfHd,  1  ) ;

    usrVals = udfGetUsrVals(        udfHd                          ) ;
    usrStrs = udfGetUsrStrs(        udfHd                          ) ;
    ErgunA = usrVals[0] ;
    ErgunB = usrVals[1] ;

/*-----------------------------------------------------------------------------
 * Get particle and flow data at current particle location
 *-----------------------------------------------------------------------------
```

```
 */
    scalar = udfGetEDEMData( udfHd, UDF_EDEM_FLOW_DENSITY ) ;
    dens = scalar[0] ;

    scalar = udfGetEDEMData( udfHd, UDF_EDEM_PARTICLE_VOLUME) ;
    vol = scalar[0] ;

    scalar = udfGetEDEMData( udfHd, UDF_EDEM_FLOW_VISCOSITY ) ;
    visc = scalar[0] ;

    scalar = udfGetEDEMData( udfHd, UDF_EDEM_FLOW_POROSITY ) ;
    poros = scalar[0] ;

    flowVel = udfGetEDEMData( udfHd, UDF_EDEM_FLOW_VELOCITY ) ;

    particleVel = udfGetEDEMData( udfHd, UDF_EDEM_PARTICLE_VELOCITY) ;

    diam = pow( 1.909859317102744 * vol, 0.3333333333333333  ) ;
    if ( vol <= 0 ) {
 udfSetError( udfHd, "Error from \"%s\" UDF, EDEM particle volume"
   " should bigger than zero", usrStrs[0] ) ;
    }
/*---------------------------------------------------------------------------
 * Calculate drag with Gidaspow model
 *---------------------------------------------------------------------------
 */
    slipVel[0] = flowVel[0] - particleVel[0] ;
    slipVel[1] = flowVel[1] - particleVel[1] ;
    slipVel[2] = flowVel[2] - particleVel[2] ;

    slipVelMag = pow( slipVel[0], 2 )
  + pow( slipVel[1], 2 )
  + pow( slipVel[2], 2 ) ;
    slipVelMag = pow( slipVelMag, 0.5 ) ;

    Re = poros * dens * slipVelMag * diam / visc ;

    if ( poros >= 0.8 ) {

       /* Wen-Yu */
 CD = 24. / Re * ( 1.0 + 0.15 * pow( Re, 0.687 ) ) ;
 if ( Re == 0 ) {
     CD = 0.0 ;
 }
 if ( Re > 1000. ) {
     CD = 0.44 ;
 }
 beta = 0.75 * CD * pow( poros,-1.65) * dens * slipVelMag ;

    } else {
 /* Ergun */
 beta = ErgunA * ( 1.0 - poros ) * visc / ( poros * diam )
      + ErgunB * dens * slipVelMag ;
    }

    betaArea = beta * vol / diam ;

/*---------------------------------------------------------------------------
 * Push the drag force to AcuSolve
 *---------------------------------------------------------------------------
 */
    outVec[0] = betaArea * slipVel[0] ;
    outVec[1] = betaArea * slipVel[1] ;
```

```
    outVec[2] = betaArea * slipVel[2] ;

} /* end of usrEDEMDrag() */

/*===========================================================================
 *
 * Prototype the function
 *
 *===========================================================================
 */

UDF_PROTOTYPE( usrEDEMLift ) ;


/*===========================================================================
 *
 * "usrEDEMLift":  calculate EDEM particle lift
 *
 *  Arguments:
 *    udfHd     - opaque handle for accessing information
 *    outVec    - output vector
 *    nItems    - number of items in outVec (=1 in this case)
 *    vecDim    - vector dimension of outVec (=1 in this case)
 *
 *  Input file parameters:
 *    user_values = { ErgunA, ErgunB }
 *    user_strings = { "" }
 *
 *===========================================================================
 */

Void usrEDEMLift(       UdfHd  udfHd,
                        Real*  outVec,
                        Integer         nItems,
                        Integer         vecDim )
{

    Real beta ;
    Real betaArea ;
    Real dens ;
    Real visc ;
    Real vol ;
    Real diam ;
    Real* scalar ;
    Real* flowVel ;
    Real* particleVel ;
    Real* particleVelCurl ;
    Real* flowVelCurl ;
    Real slipVelMag ;
    Real slipVel[3] ;
    Real slipCurl[3] ;
    Real slipVelXflowVelCurl[3] ;
    Real slipCurlXslipVel[3] ;
    Real poros ;
    Real CD ;
    Real Re ;
    Real flowVelCurlMag ;
    Real slipCurlMag ;
    Real SaffmanCoef ;
    Real MagnusCoef ;
    Real CLS ;
    Real betaLS ;
    Real CLM ;
```

```
    Real betaLM ;
    Real gamma ;
    Real*       usrVals ;
    String*     usrStrs ;
    Real carrierDens ;

/*-----------------------------------------------------------------------------
 * Get the user data
 *-----------------------------------------------------------------------------
 */
    udfCheckNumUsrVals(   udfHd,  2  ) ;
    udfCheckNumUsrStrs(   udfHd,  1  ) ;

    usrVals = udfGetUsrVals(        udfHd                          ) ;
    usrStrs = udfGetUsrStrs(        udfHd                          ) ;
    SaffmanCoef = usrVals[0] ;
    MagnusCoef = usrVals[1] ;

/*-----------------------------------------------------------------------------
 * Get particle and flow data at current particle location
 *-----------------------------------------------------------------------------
 */
    scalar = udfGetEDEMData( udfHd, UDF_EDEM_FLOW_DENSITY ) ;
    dens = scalar[0] ;

    scalar = udfGetEDEMData( udfHd, UDF_EDEM_PARTICLE_VOLUME) ;
    vol = scalar[0] ;

    scalar = udfGetEDEMData( udfHd, UDF_EDEM_FLOW_VISCOSITY ) ;
    visc = scalar[0] ;

    scalar = udfGetEDEMData( udfHd, UDF_EDEM_FLOW_POROSITY ) ;
    poros = scalar[0] ;

    flowVel = udfGetEDEMData( udfHd, UDF_EDEM_FLOW_VELOCITY ) ;
    flowVelCurl = udfGetEDEMData( udfHd, UDF_EDEM_FLOW_VELOCITY_CURL) ;

    particleVel = udfGetEDEMData( udfHd, UDF_EDEM_PARTICLE_VELOCITY) ;

    particleVelCurl = udfGetEDEMData( udfHd, UDF_EDEM_PARTICLE_VELOCITY_CURL ) ;

    diam = pow( 1.909859317102744 * vol, 0.3333333333333333  ) ;

    if ( vol <= 0 ) {
 udfSetError( udfHd, "Error from \"%s\" UDF, EDEM particle volume"
   " should bigger than zero", usrStrs[0] ) ;
    }
/*-----------------------------------------------------------------------------
 * Calculate lift with Saffman Magnus
 *-----------------------------------------------------------------------------
 */
    slipVel[0] = flowVel[0] - particleVel[0] ;
    slipVel[1] = flowVel[1] - particleVel[1] ;
    slipVel[2] = flowVel[2] - particleVel[2] ;

    slipVelMag = pow( slipVel[0], 2 )
  + pow( slipVel[1], 2 )
  + pow( slipVel[2], 2 ) ;
    slipVelMag = pow( slipVelMag, 0.5 ) ;

    flowVelCurlMag = pow( flowVelCurl[0], 2 )
   + pow( flowVelCurl[1], 2 )
   + pow( flowVelCurl[2], 2 ) ;
```

```
    flowVelCurlMag = pow( flowVelCurlMag, 0.5 ) ;

    slipVelXflowVelCurl[0] = slipVel[1] * flowVelCurl[2] - slipVel[2] *
 flowVelCurl[1]  ;
    slipVelXflowVelCurl[1] = slipVel[2] * flowVelCurl[0] - slipVel[0] *
 flowVelCurl[2]  ;
    slipVelXflowVelCurl[2] = slipVel[0] * flowVelCurl[1] - slipVel[1] *
 flowVelCurl[0]  ;

    slipCurl[0] = 0.5 * flowVelCurl[0] - particleVelCurl[0] ;
    slipCurl[1] = 0.5 * flowVelCurl[1] - particleVelCurl[1] ;
    slipCurl[2] = 0.5 * flowVelCurl[2] - particleVelCurl[2] ;

    slipCurlMag = pow( slipCurl[0], 2 )
  + pow( slipCurl[1], 2 )
  + pow( slipCurl[2], 2 ) ;
    slipCurlMag = pow( slipCurlMag, 0.5 ) ;


    slipCurlXslipVel[0] = slipVel[2] * slipCurl[1] - slipVel[1] * slipCurl[2]  ;
    slipCurlXslipVel[1] = slipVel[0] * slipCurl[2] - slipVel[2] * slipCurl[0]  ;
    slipCurlXslipVel[2] = slipVel[1] * slipCurl[0] - slipVel[0] * slipCurl[1]  ;


    Re = poros * dens * slipVelMag * diam / visc ;
    CLS = 0.0 ;
    betaLS = 0.0 ;
    if ( flowVelCurlMag > 0. && slipVelMag > 0. ) {
 gamma = flowVelCurlMag * diam / (2*slipVelMag) ;
 if ( Re <= 40. ) {
     CLS = ( 1. - 0.3314 * pow(gamma,0.5) ) * exp(-0.1*Re)
  + 0.3314 * pow(gamma,0.5) ;
 } else {
     CLS = 0.0524 * pow(gamma*Re,0.5) ;
 }
        betaLS = SaffmanCoef * CLS * pow(diam,2) * pow(visc*dens,0.5)
                          / pow(flowVelCurlMag,0.5) ;
    }

    CLM = 0.0 ;
    betaLM = 0.0 ;
    if ( slipVelMag > 0. && slipCurlMag > 0. ) {
 if ( Re <= 1. ) {
        CLM = diam * slipCurlMag / slipVelMag ;
 } else if ( Re > 1. ) {
     CLM = diam * slipCurlMag / slipVelMag
  * ( 0.178 + 0.822 * pow(Re,-0.522) ) ;
 }
        betaLM = MagnusCoef * CLM * pow(slipVelMag,2) * 3.141592653589793
  * pow(diam,2) * dens / ( slipCurlMag*slipVelMag ) ;
    }

/*---------------------------------------------------------------------------
 * Push the lift force to AcuSolve
 *---------------------------------------------------------------------------
 */
    outVec[0] = betaLS * slipVelXflowVelCurl[0] + betaLM * slipCurlXslipVel[0] ;
    outVec[1] = betaLS * slipVelXflowVelCurl[1] + betaLM * slipCurlXslipVel[1] ;
    outVec[2] = betaLS * slipVelXflowVelCurl[2] + betaLM * slipCurlXslipVel[2] ;

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

Errors

- This routine expects a valid udfHd.
- This routine may only be called within an AcuSolve-EDEM coupling drag, lift and torque model user functions.
- `dataName` must be one of the values given above.
- The problem must contain the equation associated with the requested data.

udfGetMfData()

Return data of a multiplier function.

Syntax

`mfData = udfGetMfData( udfHd, mfName) ;`

Type

AcuSolve User-Defined Function Global

Parameters

`udfHd`
: The opaque handle (pointer) which was passed to the user function.

`mfName` *(string)*
: Name of the multiplier function.

Return Value

`mfData` *(real)*
: Value of the requested multiplier function.

Description

This routine returns the current value of the specified multiplier function. For example,

```
Real mfData ;
...
mfData = udfGetMfData( udfHd, "my mult func" ) ;
```

Errors

- This routine expects a valid *`udfHd`*.
- *`mfName`* must be a valid name.

udfSetMfData()

Set the value of a multiplier function.

Syntax

udfSetMfData(udfHd, mfName, mfData) ;

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

mfName *(string)*
: Name of the multiplier function.

mfData *(real)*
: Value of the multiplier function.

Return Value

None.

Description

This routine sets the value of the specified multiplier function. It may only be used for multiplier functions that were created with modifiable type. For example,

```
udfSetMfData( udfHd, "my mult func", 2. ) ;
```

Errors

- This routine expects a valid *udfHd*.
- *mfName* must be a valid name.
- Type of the multiplier function being set must be modifiable.

udfGetFbdData()

Return data from a flexible body set.

Syntax

```
fbdData = udfGetFbdData( udfHd, cmdName, dataName ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

cmdName *(string)*
: Name of the `FLEXIBLE_BODY` command.

dataName *(integer)*
: Symbolic name of the requested data.

UDF_FBD_PREV_DISPLACEMENT	Displacement at step n.
UDF_FBD_PREV_VELOCITY	Velocity at step n.
UDF_FBD_CURR_DISPLACEMENT	Displacement at step n+1.
UDF_FBD_CURR_VELOCITY	Velocity at step n+1.
UDF_FBD_INTERNAL_FORCE	Projection of fluid forces on the body.
UDF_FBD_EXTERNAL_FORCE	User-given external forces.

Return Value

fbdData *(real)*
: Pointer to one dimensional real array of the requested data. The dimension of the array is num_modes (as given by the `FLEXIBLE_BODY` command) for all values of *dataName*.

Description

This routine returns data from a flexible body model. For example,

```
char* name ; /* name of the UDF */

Integer i ; /* a running index */

Real* disp ; /* displacement (curr or prev) */

Real* dispC ; /* next displacement */

Real* vel ; /* velocity (curr or prev) */
```

```
Real* velC ; /* next velocity */
Real* extForce ; /* external force */
Real* intForce ; /* internal force */
name = udfGetName( udfHd ) ;
dispC = outVec ;
velC = outVec + nItems ;
/* If the first time step */
   if ( udfFirstStep(udfHd) ) {
      disp = udfGetFbdData( udfHd, name, UDF_FBD_CURR_DISPLACEMENT ) ;
      vel = udfGetFbdData( udfHd, name, UDF_FBD_CURR_VELOCITY ) ;
      for ( i = 0 ; i < nItems ; i++ ) {
          dispC[i] = disp[i] ;
          velC[i] = vel[i] ;
      }
      return ;
   }
/* Otherwise, get the data from the external program */
   intForce = udfGetFbdData( udfHd, name, UDF_FBD_INTERNAL_FORCE ) ;
   extForce = udfGetFbdData( udfHd, name, UDF_FBD_EXTERNAL_FORCE ) ;
   disp = udfGetFbdData( udfHd, name, UDF_FBD_PREV_DISPLACEMENT ) ;
   vel = udfGetFbdData( udfHd, name, UDF_FBD_PREV_VELOCITY ) ;
   for ( i = 0 ; i < nItems ; i++ ) {
          dispC[i] = ... ;
          velC[i] = ... ;
    }
```

Errors

- This routine expects a valid *udfHd*.
- *cmdName* must be a valid name.
- *dataName* must be one of the values given above.

udfGetMmoRgdData()

Return data from a rigid body mesh motion.

Syntax

```
mmoRgdData = udfGetMmoRgdData( udfHd, mmoRgdName, dataName ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

mmoRgdName *(string)*
: Name of the MESH_MOTION command (must be of rigid_body type).

dataName *(integer)*
: Symbolic name of the requested data.

UDF_MMO_RGD_PREV_DISPLACEMENT Displacement at step n.

UDF_MMO_RGD_PREV_VELOCITY Velocity at step n.

UDF_MMO_RGD_PREV_ROTATION Rotation at step n.

UDF_MMO_RGD_PREV_ANG_VELOCITY Angular velocity at step n.

UDF_MMO_RGD_CURR_DISPLACEMENT Displacement at step n+1.

UDF_MMO_RGD_CURR_VELOCITY Velocity at step n+1.

UDF_MMO_RGD_CURR_ROTATION Rotation at step n+1.

UDF_MMO_RGD_CURR_ANG_VELOCITY Angular velocity at step n+1.

UDF_MMO_RGD_EXTERNAL_FORCE External force at step n+1.

UDF_MMO_RGD_EXTERNAL_MOMENT External moment at step n+1.

Return Value

mmoRgdData *(real)*
: Pointer to one dimensional real array of the requested data. The dimension of the array is three for all values of *dataName*.

Description

This routine returns data from a rigid body mesh motion. For example,

```
Void usrUdfMmo(
```

```
   UdfHd udfHd, /* Opque handle for accessing data */
   Real* outVec, /* Output vector */
   Integer nItems, /* Number of items in outVec */
   Integer        vecDim /* Vector dimension of outVec */
)  {
   Real* rotOrg ; /* rotation center */
   Real* rot ;        /* rotation */
   Real* trans ; /* displacement */
   String*        usrStrs ; /* user strings */
   String rgdMmoName ; /* name of rigid body mmo */
   Integer timeStep ;/* time step */
   Integer i ; /* running index */


   udfCheckNumUsrStrs( udfHd, 1 ) ;
   udfCheckNumUsrVals( udfHd, 3 ) ;


   timeStep = udfGetTimeStep( udfHd ) ;
   usrStrs = udfGetUsrStrs( udfHd ) ;
   rgdMmoName = usrStrs[0] ;
   rotOrg = udfGetUsrVals( udfHd ) ;
   rot = udfGetMmoRgdData( udfHd, rgdMmoName, UDF_MMO_RGD_CURR_ROTATION ) ;
   trans = udfGetMmoRgdData( udfHd, rgdMmoName, UDF_MMO_RGD_CURR_DISPLACEMENT) ;
   udfPrintMess( udfHd, "time step = %d, rot = (%f,%f,%f), disp = (%f,%f,%f)",
           timeStep, rot[0], rot[1], rot[2],
           trans[0], trans[1], trans[2]        ) ;
   udfBuildMmo( udfHd, rot, rotOrg, trans, outVec ) ;


   for ( i = 0 ; i < 12 ; i++ ) {
       udfPrintMess( udfHd, "mmoMtx[%d] = %f", i, outVec[i] );
   }
} /* end of usrUdfMmo */
```

Errors

- This routine expects a valid *udfHd*.
- *mmoRgdName* must be a valid name.
- *dataName* must be one of the values given above.

udfGetMmoRgdJac()

Return storage for rigid body external force or moment derivatives with respect to displacement or rotation.

Syntax

mmoRgdJac = udfGetMmoRgdJac(udfHd, mmoRgdName, dataName) ;

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque (pointer) which was passed to the user function.

mmoRgdName *(string)*
: Name of the MESH_MOTION command (must be of type = rigid_body).

dataName *(integer)*
: Symbolic name of the requested data.

UDF_MMO_RGD_JAC_DISPLACEMENT Derivative of force with respect to displacement.

UDF_MMO_RGD_JAC_ROTATION Derivative of moment with respect to rotation.

Return Value

mmoRGDJac *(real)*
: Pointer to one-dimensional storage array. The dimension of the array is nine for three components of the derivatives of *dataName*.

Description

This function will return a pointer to the Jacobian at quadrature points. The Jacobian is the derivatives of the UDF type (force or moment) with respect to the *dataName* which is displacement or rotation. Nine components of this pointer array should be assigned explicitly in this function.

```
Void usrMmoRgdExtFrc( UdfHd udfHd )
{
   Real dFrc1d1 ; /* derivetive of force to Dis. */
   Real dFrc1d2 ; /* derivetive of force to Dis. */
   Real dFrc1d3 ; /* derivetive of force to Dis. */
   Real dFrc2d1 ; /* derivetive of force to Dis. */
   Real dFrc2d2 ; /* derivetive of force to Dis. */
   Real dFrc2d3 ; /* derivetive of force to Dis. */
   Real dFrc3d1 ; /* derivetive of force to Dis. */
   Real dFrc3d2 ; /* derivetive of force to Dis. */
   Real dFrc3d3 ; /* derivetive of force to Dis. */

   udfCheckNumUsrVals( udfHd, 3 ) ;
   udfCheckNumUsrStrs( udfHd, 1 ) ;
   usrStrs = udfGetUsrstrs ( udfHd ) ;
   rgdMmoName = usrStrs[0] ;
```

```
   mmoRgdDisJac = udfGetMmoRgdJac( udfHd,
                                   rgdMmoName,
                                   UDF_MMO_RGD_JAC_DISPLACEMENT ) ;
   mmoRgdRotJac = udfGetMmoRgdJac( udfHd,
                                   rgdMmoName,
                                   UDF_MMO_RGD_JAC_ROTATION ) ;

   mmoRgdDisJac[0] = dFrc1d1 ;
   mmoRgdDisJac[1] = dFrc1d2 ;
   mmoRgdDisJac[2] = dFrc1d3 ;

   mmoRgdDisJac[3] = dFrc2d1 ;
   mmoRgdDisJac[4] = dFrc2d2 ;
   mmoRgdDisJac[5] = dFrc2d3 ;

   mmoRgdDisJac[6] = dFrc3d1 ;
   mmoRgdDisJac[7] = dFrc3d2 ;
   mmoRgdDisJac[8] = dFrc3d3 ;

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

Errors

None.

udfHasUgd()

Has this user global data been set?

Syntax

```
ugdFlag = udfHasUgd( udfHd, ugd ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

ugd (string)
: Name of the user global data variable.

Return Value

***ugdFlag** (integer)*
: Flag indicating whether or not this ugd has been set.

0	No
1	Yes

Description

This routine checks whether or not the specified user global data has been set. For example,

```
ugdFlag = udfHasUgd( udfHd, "nCols" ) ;
```

Errors

- This routine expects a valid *udfHd*.
- *ugd* must exist.

udfCheckUgd()

Check the existence of a user global data.

Syntax

udfCheckUgd(udfHd, ugd) ;

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

ugd (string)
: Name of the user global data variable.

Return Value

None

Description

This routine verifies that the specified user global data is defined. If it is not, an error message is issued and the solver exits. For example,

```
udfCheckUgd( udfHd, "nCols" ) ;
```

Errors

None

udfSetUgdData()

Set the value of a user global data.

Syntax

```
udfSetUgdData( udfHd, ugd, ugdData ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

ugd *(string)*
: Name of the user global data variable.

ugdData *(real)*
: Value of the user global data.

Return Value

None

Description

This routine sets the value of the specified user global data. The user global data is created if it does not exist. For example,

```
udfSetUgdData( udfHd, "nCols", 2. ) ;
```

Errors

- This routine expects a valid *udfHd*.
- *ugd* must exist.
- May only be called from a multiplier function or a mesh motion user-defined function.

udfGetUgdData()

Return the value of a user global data.

Syntax

ugdData = udfGetUgdData(udfHd, ugd) ;

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

ugd (string)
: Name of the user global data variable.

Return Value

***ugdData** (real)*
: Value of the user global data.

Description

This routine returns the value of the specified user global data. If the user global data has not been set, then it is set with a value of zero. For example,

```
Real ugdData ;
...
ugdData = udfGetUgdData( udfHd, "nCols" ) ;
```

Errors

- This routine expects a valid *udfHd*.
- *ugd* must exist.

udfGetGlobalVector()

Return a user global vector.

Syntax

vec = udfGetGlobalVector(udfHd, name, nDims) ;

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

name (char)*
: Pointer to name of the global vector.

nDims (integer)
: Dimension.

Return Value

***vec** (real)*
: Pointer to the global vector.

Description

This routine returns a pointer to a named user global vector. For example, the user_function usrMotionRigidBody() computes and stores a motion vector in the global vector named "motion". For example,

```
MULTIPLIER_FUNCTION( "Fairing" ) {
  type = user_function
  user_function = "usrMotionRigidBody"
  ...
}
```

where usrMotionRigidBody() has the form:

```
Void usrMotionRigidBody( UdfHd udfHd,
   Real* outVec,
   Integer nItems,
   Integer vecDim )
{
   Real xDisp ; /* displacement in x */
   Real yDisp ; /* displacement in y */
   Real zRot ; /* rotation in z */
   ...
   /* Calculate displacements and rotation */
   ...
   /* Store displacements and rotation in user global vector */
   vec = udfGetGlobalVector(udfHd, "motion", 3 ) ;
   vec[0] = xDisp ;
```

```
   vec[1] = yDsip ;
   vec[2] = zRot ;
   ...
}
```

The motion vector can be retrieved and used in a nodal boundary condition as follows:

```
NODAL_BOUNDARY_CONDTION( "x_motion" ) {
  type = user_function
  user_function = "usrMotionNBC"
  user_values = { 1, 0.00150704, 0., -1., 3., .2, -2., 2., .2, .2 }
  nodes = Read( "MESH.DIR/all_nodes.nbc" )
  variable = mesh_x_displacement
}
```

where the user-defined function usrMotionNBC may be implemented as:

```
Void usrMotionNBC( UdfHd udfHd, /* opaque handle for accessing information */
                Real* outVec, /* output vector */
                Integer nItems, /* No. of items in outVec (=nNodes here) */
                Integer vecDim ) /* vector dimension of outVec (=1 here) */
{
   Integer dir ; /* direction being computed */
   Integer i ; /* a running index */
   Real* crd ; /* nodal coordinates */
   Real ct ; /* cos( zRot ) - 1 */
   Real dx ; /* change in x */
   Real dy ; /* change in y */
   Real r ; /* distance to CM */
   Real rWidth ; /* r width */
   Real st ; /* sin( zRot ) */
   Real* usrVals ; /* user supplied values */
   Real* vec ; /* computed motion */
   Real xDisp ; /* displacement in x */
   Real xMax ; /* max x location */
   Real xMin ; /* min x location */
   Real xOrg ; /* center of mass */
   Real xs ; /* x scale factor */
   Real xWidth ; /* x width */
   Real x ; /* x location of the point */
   Real yDisp ; /* displacement in y */
   Real yMax ; /* max y location */
   Real yMin ; /* min y location */
   Real yOrg ; /* center of mass */
   Real ys ; /* y scale factor */
   Real yWidth ; /* y width */
   Real y ; /* y location of the point */
   Real zRot ; /* rotation in z */
/* Get the user data. Expected format:
   * user_values =
   * { <direction>, <xOrg>, <yOrg>, <xMin>, <xMax>, <xWidth>,
   * <yMin>, <yMax>, <yWidth>, <rWidth> }
   */
   udfCheckNumUsrVals( udfHd, 10 ) ;
   usrVals = udfGetUsrVals( udfHd ) ;
   dir = (Integer) usrVals[0] ;
   xOrg = usrVals[1] ;
   yOrg = usrVals[2] ;
   xMin = usrVals[3] ;
   xMax = usrVals[4] ;
   xWidth = usrVals[5] ;
   yMin = usrVals[6] ;
```

```
   yMax = usrVals[7] ;
   yWidth = usrVals[8] ;
   rWidth = usrVals[9] ;
/* Get the motion */
   vec = udfGetGlobalVector( udfHd, "motion", 3 ) ;
   xDisp = vec[0] ;
   yDisp = vec[1] ;
   zRot = vec[2] ;
/* Get the coordinates */
   crd = udfGetNbcRefCrd( udfHd ) ;
/* Compute the motion */
   for ( i = 0 ; i < nItems ; i++ ) {
         x = crd[0*nItems+i] - xOrg ;
         y = crd[1*nItems+i] - yOrg ;
         xs = 1 ;
         ys = 1 ;
         r = sqrt( x * x + y * y ) / rWidth ;
         r = max( 1, r ) ;
         r = pow( r, 0.7 ) ;
         if ( x > +xWidth ) { xs = 1 - (x - xWidth) / (xMax - xWidth) ; }
         if ( x < -xWidth ) { xs = 1 - (x + xWidth) / (xMin + xWidth) ; }
         if ( y > +yWidth ) { ys = 1 - (y - yWidth) / (yMax - yWidth) ; }
         if ( y < -yWidth ) { ys = 1 - (y + yWidth) / (yMin + yWidth) ; }
         ct = cos( zRot / r ) ;
         st = sin( zRot / r ) ;
         dx = xs * ( xDisp + ct * x - st * y - x ) ;
         dy = ys * ( yDisp + st * x + ct * y - y ) ;
         if ( dir == 1 ) {
         outVec[i] = dx ;
         } else {
         outVec[i] = dy ;
         }
   }
} /* end of usrMotionNBC() */
```

Errors

This routine expects a valid *udfHd*.

udfBuildMmo()

Build the mesh motion transformation matrix.

Syntax

udfBuildMmo(udfHd, rot, rotOrg, trans, mmoMtx) ;

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

rot (real)
: Rotation vector.

rotOrg (real)
: Origin of rotation.

trans (real)
: Translation vector.

mmoMts (real)
: Mesh motion transformation matrix.

Return Value

None

Description

This routine builds the mesh motion transformation matrix from rotation and translation vectors. For example,

```
Void
usrRotTransMmo( UdfHd udfHd, /* opaque handle for accessing information */
                Real* outVec, /* output vector */
                Integer nItems, /* No. of items in outVec (=1 here) */
                Integer vecDim ) /* vector dimension of outVec (=12 here) */
{
    Integer i ; /* a running index */
    Real angle ; /* rot/trans angle */
    Real rotOmega ; /* rotation frequency */
    Real trnOmega ; /* translation frequency */
    Real trnAmp ; /* translation amplitude */
    Real rotOrg[3] ; /* origin of rotation */
    Real rot[3] ; /* rotation angle */
    Real trans[3] ; /* translation */
    Real mmoMtx[12] ; /* transformation matrix */
    Real time ; /* current time */
    Real sa ; /* sine of angle */
    Real* usrVals ; /* user supplied values */
```

```
/* Get global data */
     time = udfGetTime( udfHd ) ;

/* Get the user data */
    udfCheckNumUsrVals( udfHd, 6 ) ; /* check for error */
    usrVals = udfGetUsrVals( udfHd ) ; /* get the user vals */

    rotOrg[0] = usrVals[0] ; /* get x-origin of rotation */
    rotOrg[1] = usrVals[1] ; /* get y-origin of rotation */
    rotOrg[2] = usrVals[2] ; /* get z-origin of rotation */
    rotOmega = usrVals[3] ; /* get rotation frequency */
    trnOmega = usrVals[4] ; /* get translation frequency */
    trnAmp = usrVals[5] ; /* get translation amplitude */

/* Construct rotation and translation vectors */
    angle = time * rotOmega * 2 * 3.1415926535897931E+00 ;
    rot[0] = 0.0 ;
    rot[1] = 0.0 ;
    rot[2] = angle ;

    angle = time * trnOmega * 2 * 3.1415926535897931E+00 ;
    sa = sin( angle ) ;
    trans[0] = 0.0 ;
    trans[1] = 0.0 ;
    trans[2] = trnAmp * sa ;

/* Build rotation matrix */
    udfBuildMmo( udfHd, rot, rotOrg, trans, mmoMtx ) ;

/* Output vector */
    for ( i = 0 ; i < 12 ; i++ ) {
          outVec[i] = mmoMtx[i] ;
    }
} /* end of usrRotTransMmo() */
```

Errors

This routine expects a valid *udfHd*.

udfGetGlobalHistsCurr1()

Get the current global history variables.

Syntax

```
vec = udfGetGlobalHistsCurr1( udfHd, nSize, id1 ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

nSize *(integer)*
: Size of vector.

id1 *(integer)*
: index.

Return Value

vec *(real)*
: Pointer to the global vector.

Description

This routine returns a pointer to *id1*-th current global history variables in size *nSize* vector. For example,

```
Real* vec ;
...
vec = udfGetGlobalHistsCurr1( udfHd, nSize, id1 )
```

Errors

This routine expects a valid *udfHd*.

udfGetGlobalHistsPrev1()

Get the previous global history variables.

Syntax

```
vec = udfGetGlobalHistsPrev1( udfHd, nSize, id1 ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

nSize *(integer)*
: Size of vector.

id1 *(integer)*
: index.

Return Value

vec *(real)*
: Pointer to the global vector.

Description

This routine returns a pointer to *id1*-th previous global history variables in size *nSize* vector. For example,

```
Real* vec ;
...
vec = udfGetGlobalHistsPrev1( udfHd, nSize, id1 )
```

Errors

This routine expects a valid *udfHd*.

udfGetGlobalHistsCurr2()

Get the current global history variables.

Syntax

```
vec = udfGetGlobalHistsCurr2( udfHd, nSize, id1, id2 ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

nSize *(integer)*
: Size of vector.

id1 *(integer)*
: index 1.

id2 *(integer)*
: index 2.

Return Value

vec *(real)*
: Pointer to the global vector.

Description

This routine returns a pointer to *(id1*, *id2*)-th current global history variables in size *nSize* vector. For example,

```
Real* vec ;
...
vec = udfGetGlobalHistsCurr2( udfHd, nSize, id1, id2 )
```

Errors

This routine expects a valid *udfHd*.

udfGetGlobalHistsPrev2()

Get the previous global history variables.

Syntax

```
vec = udfGetGlobalHistsPrev2( udfHd, nSize, id1, id2 ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

nSize *(integer)*
: Size of vector.

id1 *(integer)*
: index 1.

id2 *(integer)*
: index 2.

Return Value

vec *(real)*
: Pointer to the global vector

Description

This routine returns a pointer to (*id1*, *id2*)-th previous global history variables in size *nSize* vector. For example,

```
Real* vec ;
...
vec = udfGetGlobalHistsPrev2( udfHd, nSize, id1, id2 )
```

Errors

This routine expects a valid *udfHd*.

udfGetGlobalHistsCurr3()

Get the current global history variables.

Syntax

```
vec = udfGetGlobalHistsCurr3( udfHd, nSize, id1, id2, id3 ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

nSize *(integer)*
: Size of vector.

id1 *(integer)*
: index 1.

id2 *(integer)*
: index 2.

id3 *(integer)*
: index 3.

Return Value

vec *(real)*
: Pointer to the global vector

Description

This routine returns a pointer to the *(id1*, *id2*, *id3*)-th current global history variables in size *nSize* vector. For example,

```
Real* vec ;
...
vec = udfGetGlobalHistsCurr3( udfHd, nSize, id1, id2, id3 )
```

Errors

This routine expects a valid *udfHd*.

udfGetGlobalHistsPrev3()

Get the previous global history variables.

Syntax

```
vec = udfGetGlobalHistsPrev3( udfHd, nSize, id1, id2, id3 ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

nSize *(integer)*
: Size of vector.

id1 *(integer)*
: index 1.

id2 *(integer)*
: index 2.

id3 *(integer)*
: index 3.

Return Value

vec *(real)*
: Pointer to the global vector.

Description

This routine returns a pointer to (*id1*, *id2*, *id3*)-th previous global history variables in size *nSize* vector. For example,

```
Real* vec ;
...
vec = udfGetGlobalHistsPrev3( udfHd, nSize, id1, id2, id3 )
```

Errors

This routine expects a valid *udfHd*.

udfGetNumSds()

Return the number of subdomains.

Syntax

```
nSds = udfGetNumSds( udfHd ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nSds *(integer)*
: Number of subdomains.

Description

The older function `udfGetNSds` is replaces with this user function. This routine returns the number of subdomains. For example,

```
Integer nSds ;
...
nSds = udfGetNumSds( udfHd ) ;
```

Errors

This routine expects a valid *udfHd*.

udfGetSdId()

Return the ID of the subdomain.

Syntax

```
sdId = udfGetSdId( udfHd ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

sdId *(integer)*
: ID of subdomain, from 0 to `udfGetNumSds(udfHd)-1`.

Description

This routine returns the ID of subdomain. For example,

```
Integer sdId ;
...
sdId = udfGetSdId( udfHd ) ;
```

Errors

This routine expects a valid *udfHd*.

udfGetLastStepFlag()

Is this the last time step?

Syntax

```
lastStepFlag = udfGetLastStepFlag( udfHd ) ;
```

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

lastStepFlag *(integer)*
: Flag indicating whether or not this is the last time step.

0	No
1	Yes

Description

This routine determines if this is the last time step. For example,

```
Integer lastStepFlag ;
...
lastStepFlag = udfGetLastStepFlag( udfHd ) ;
if ( lastStepFlag == 1 ) {
   /* Logic if this is the last time step */
   ...
}
```

Errors

This routine expects a valid *udfHd*.

udfMeanConv()

Compute the mean and confidence of a signal.

Syntax

udfMeanConv(udfHd, sInput, timeScale, skipTime, attn, sMean, sError) ;

Type

AcuSolve User-Defined Function Global

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

sInput *(real)*
: Input signal at the time t.

timeScale *(real)*
: Time scale of a signal. The formula, 2.0 * dt / *timeScale*, is used to decide the cut off frequency for low-pass filtering.

skipTime *(real)*
: Time to skip accumulating mean and error in the beginning of iterations.

attn *(real)*
: Attenuation for filtering. The value should be less or equal to unity.

sMean *(real)*
: Returned mean value.

sError *(real)*
: Returned convergence error value. This error value has the same dimension as the original signal. *sInput*.

Return Value

None

Description

This routine computes the mean and confidence of a signal. For example,

```
Integer timeScale = 0.1 ;
Integer skipTime = 1.0 ;
Integer attn = 0.1 ;
...
while ( sError > sErrorTole ) {
   ...
   udfMeanConv( udfHd, sInput, timeScale, skipTime, attn, sMean, sError ) ;
   ...
}
```

Errors

This routine expects a valid `udfHd`.

Element Routines

9

These routines are accessible only by the Body Force, Material Model and Component Model user functions, and by external output codes.

This chapter covers the following:

udfGetElmType()

Return the type of the element set.

Syntax

```
type = udfGetElmType (udfHd) ;
```

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

type *(integer)*
: Type of the element set.

Description

This routine returns the type of the element set. For example,

```
Integer type ;
...
type = udfGetElmType( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

udfGetElmQuadType()

Return the type of the quadrature rule for the element set.

Syntax

quadType = udfGetElmQuadType(udfHd) ;

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

quadType *(integer)*
: Type of the quadrature rule for the element set.

Description

This routine returns the type quadrature rule of the element set. For example,

```
Integer quadType ;
...
quadType = udfGetElmQuadType( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

udfGetElmNQuads()

Return the number of quadrature points of the quadrature rule for the element set.

Syntax

```
nQuads = udfGetElmNQuadType( udfHd ) ;
```

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nQuads *(integer)*
: Number of quadrature points of the quadrature rule for the element set.

Description

This routine returns the number of quadrature points of the quadrature rule for the element set. For example,

```
Integer nQuads ;
...
nQuads = udfGetElmNQuads( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

udfGetElmQuadId()

Return the quadrature ID of the quadrature rule for the element set.

Syntax

```
quadId = udfGetElmQuadId( udfHd ) ;
```

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

QuadId *(integer)*
: Quadrature ID of the quadrature rule for the element set.

Description

This routine returns the quadrature ID of the quadrature rule for the element set. For example,

```
Integer quadId ;
...
quadId = udfGetElmQuadId( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

udfGetElmTime()

Return the current time.

Syntax

```
time = udfGetElmTime( udfHd ) ;
```

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

time *(real)*
: Current time.

Description

This routine returns the current time. For example,

```
Real time ;
...
time = udfGetElmTime( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

udfGetElmIds()

Return the user element numbers.

Syntax

```
elemIds = udfGetElmIds( udfHd ) ;
```

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

elemIds *(Integer*)*
: Pointer to one-dimensional integer array of user element numbers. This array has a dimension of number of elements, nItems.

Description

This routine returns the array of user element numbers. This is the first array column of the parameter elements of the command `ELEMENT_SET` in the input file. For example,

```
Integer* elemIds ;
Integer usrElem, elem ;
...
elemIds = udfGetElmIds( udfHd ) ;
for ( elem = 0 ; elem < nItems ; elem++ ) {
   usrElem = elemIds[elem] ;
 ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

udfGetElmCrd()

Return the element coordinates at the quadrature point.

Syntax

```
crd = udfGetElmCrd( udfHd ) ;
```

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

crd *(Real*)*
: Pointer to two dimensional real array of element coordinates. The first (fastest) dimension of the array is the number of elements, nItems, and the second (slower) dimension is three, for the x, y and z coordinates.

Description

This routine returns the element coordinates at the quadrature point. If mesh displacement is active, the returned coordinates are for the current (deformed) configuration. For example,

```
Real* crd ;
Real x, y, z ;
Integer elem ;
...
crd = udfGetElmCrd( udfHd ) ;
for ( elem = 0 ; elem < nItems ; elem++ ) {
   x = crd[0*nItems+elem] ;
   y = crd[1*nItems+elem] ;
   z = crd[2*nItems+elem] ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

udfGetElmWDetJ()

Return the weighted determinant of the element Jacobian at the quadrature point.

Syntax

```
wghtDetJac = udfGetElmWDetJ( udfHd ) ;
```

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

wghtDetJac *(Real*)*
: Pointer to one dimensional real array of the weighted determinant of the Jacobian at the element quadrature point. The array dimension is the number of elements, nItems.

Description

This routine returns the weighted determinant of the element Jacobian at the quadrature point. The unit of this quantity is volume. The sum over the quadrature points is the volume of the element. For example,

```
Real* wghtDetJac ;
Real wDetJ ;
Integer elem ;
...
wghtDetJac = udfGetElmWDetJ( udfHd ) ;
for ( elem = 0 ; elem < nItems ; elem++ ) {
  wDetJ = wghtDetJac[elem] ;
 ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

udfGetElmCovar()

Return the element covariant at the quadrature point.

Syntax

```
covar = udfGetElmCovar( udfHd ) ;
```

Type

User Defined Element

Parameters

udfHd

The opaque handle (pointer) which was passed to the user function.

Return Value

covar *(Real*)*

Pointer to two dimensional real array of the element covariant. The first (fastest) dimension of the array is the number of elements, nItems, and the second (slower) dimension is six, for the xx, yy, zz, xy, yz and zx components.

Description

This routine returns the element covariant at the quadrature point. The element covariant is given by

$$g_{ij} = \frac{\partial \xi_\alpha}{\partial x_i}\frac{\partial \xi_\alpha}{\partial x_j} \tag{1}$$

where $\xi_\alpha \ (\alpha = 1, 2, 3)$ is the local parent element coordinate and $x_i \ (i = 1, 2, 3)$ is the global coordinate. The latter is for the current (deformed) configuration if mesh displacement is active. For example,

```
Real* covar ;
Real xx_covar, yy_covar, zz_covar ;
Real xy_covar, yz_covar, zx_covar ;
Integer elem ;
...
covar = udfGetElmCovar( udfHd ) ;
for ( elem = 0 ; elem < nItems ; elem++ ) {
   xx_covar = covar[0*nItems+elem] ;
   yy_covar = covar[1*nItems+elem] ;
   zz_covar = covar[2*nItems+elem] ;
   xy_covar = covar[3*nItems+elem] ;
   yz_covar = covar[4*nItems+elem] ;
   zx_covar = covar[5*nItems+elem] ;
   ...
}
```

Errors

- This routine expects a valid `udfHd`.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

udfGetElmContvar()

Return the element contravariant at the quadrature point.

Syntax

```
contvar = udfGetElmContvar( udfHd ) ;
```

Type

User Defined Element

Parameters

udfHd

The opaque handle (pointer) which was passed to the user function.

Return Value

contvar *(Real*)*

Pointer to two dimensional real array of the element contravariant. The first (fastest) dimension of the array is the number of elements, nItems, and the second (slower) dimension is six, for the xx, yy, zz, xy, yz and zx components.

Description

This routine returns the element contravariant at the quadrature point. The element contravariant is given by

$$g^{ij} = \frac{\partial x_j}{\partial \xi_\alpha}\frac{\partial x_j}{\partial \xi_\alpha} \tag{2}$$

where $\xi_\alpha \ (\alpha = 1, 2, 3)$ is the local parent element coordinate and $x_i \ (i = 1, 2, 3)$ is the global coordinate. The latter is for the current (deformed) configuration if mesh displacement is active. For example,

Examples

```
Real* contvar ;
Real xx_contvar, yy_contvar, zz_contvar ;
Real xy_contvar, yz_contvar, zx_contvar ;
Integer elem ;
...
contvar = udfGetElmContvar( udfHd ) ;
for ( elem = 0 ; elem < nItems ; elem++ ) {
  xx_contvar = contvar[0*nItems+elem] ;
  yy_contvar = contvar[1*nItems+elem] ;
  zz_contvar = contvar[2*nItems+elem] ;
  xy_contvar = contvar[3*nItems+elem] ;
  yz_contvar = contvar[4*nItems+elem] ;
  zx_contvar = contvar[5*nItems+elem] ;
  ...
}
```

Errors

- This routine expects a valid `udfHd`.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

udfGetElmData()

Return the element data at the quadrature point.

Syntax

```
data = udfGetElmData( udfHd, dataName ) ;
```

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName *(integer)*
: Symbolic name of the requested data.

UDF_ELM_VELOCITY	Velocity.
UDF_ELM_ACCELERATION	Acceleration.
UDF_ELM_PRESSURE	Pressure.
UDF_ELM_TEMPERATURE	Temperature.
UDF_ELM_DENSITY	Density.
UDF_ELM_SPECIES	Species.
UDF_ELM_VISCOSITY	Viscosity.
UDF_ELM_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_ELM_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_ELM_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_ELM_GRAD_VELOCITY	Gradient of velocity.
UDF_ELM_GRAD_PRESSURE	Gradient of pressure.
UDF_ELM_GRAD_TEMPERATURE	Gradient of temperature.
UDF_ELM_GRAD_SPECIES	Gradient of species.
UDF_ELM_MESH_DISPLACEMENT	Mesh displacement.
UDF_ELM_MESH_VELOCITY	Mesh velocity.
UDF_ELM_MESH_GRAD_DISPLACEMENT	Gradient of mesh displacement.

UDF_ELM_MESH_GRAD_VELOCITY Gradient of mesh velocity.

UDF_ELM_TURBULENCE_Y Distance to nearest turbulence wall.

UDF_ELM_TURBULENCE_YPLUS Turbulence y+ based on distance to nearest turbulence wall and shear at that wall.

UDF_ELM_VISCOELASTIC Viscoelastic.

Return Value

data *(Real*)*

Pointer to one, two, or three dimensional real array of the requested data. The dimensions of the array depend on *dataName* as follows. If the third (slowest) dimension of the array is equal to one, then the array may be treated as two dimensional. If the second dimension is likewise one, then the array is one dimensional. The x, y, z components of the gradient for gradient quantities are always contained in the last nontrivial dimension.

Table 7:

dataName	**First Dimension**	**Second Dimension**	**Third Dimension**
UDF_ELM_VELOCITY	nItems	3	1
UDF_ELM_ACCLERAT	nItems	3	1
UDF_ELM_PRESSURE	nItems	1	1
UDF_ELM_TEMPERAT	nItems	1	1
UDF_ELM_DENSITY	nItems	1	1
UDF_ELM_SPECIES	nItems	`udfGetNumSpecs()`	1
UDF_ELM_VISCOSIT	nItems	1	1
UDF_ELM_EDDY_VIS	nItems	1	1
UDF_ELM_KINETIC_	nItems	1	1
UDF_ELM_EDDY_FRE	nItems	1	1
UDF_ELM_GRAD_VE	nItems	3	3
UDF_ELM_GRAD_PR	nItems	3	1
UDF_ELM_GRAD_TE	nItems	3	1
UDF_ELM_GRAD_SP	nItems	`udfGetNumSpecs()`	3
UDF_ELM_MESH_DIS	nItems	3	1

dataName	First Dimension	Second Dimension	Third Dimension
UDF_ELM_MESH_VE	nItems	3	1
UDF_ELM_MESH_GR	nItems	3	3
UDF_ELM_MESH_GR	nItems	3	3
UDF_ELM_TURBULEI	nItems	1	1
UDF_ELM_TURBULEI	nItems	1	1
UDF_ELM_VISCOELA	nItems	6	1

Description

This routine returns the requested solution data. For example,

```
Real* data ;
Real* grad_x ;
Real* grad_y ;
Real* grad_z ;
Real* s_x ;
Real* s_y ;
Real* s_z ;
Real u, v, w ;
Real u_x, v_x, w_x, u_y, v_y, w_y, u_z, v_z, w_z ;
Real spec_x, spec_y, spec_z ;
Integer elem, nSpecs, specId ;
...
data = udfGetElmData( udfHd, UDF_ELM_VELOCITY ) ;
for ( elem = 0 ; elem < nItems ; elem++ ) {
  u = data[0*nItems+elem] ;
  v = data[1*nItems+elem] ;
  w = data[2*nItems+elem] ;
  ...
}
...
data = udfGetElmData( udfHd, UDF_ELM_GRAD_VELOCITY ) ;
for ( elem = 0 ; elem < nItems ; elem++ ) {
   u_x = data[0*nItems+elem] ;
   v_x = data[1*nItems+elem] ;
   w_x = data[2*nItems+elem] ;
   u_y = data[3*nItems+elem] ;
   v_y = data[4*nItems+elem] ;
   w_y = data[5*nItems+elem] ;
   u_z = data[6*nItems+elem] ;
   v_z = data[7*nItems+elem] ;
   w_z = data[8*nItems+elem] ;
   ...
}
...
data = udfGetElmData( udfHd, UDF_ELM_GRAD_SPECIES ) ;
nSpecs = udfGetNumSpecs( udfHd ) ;
for ( specId = 0 ; specId < nSpecs ; specId++ ) {
   for ( elem = 0 ; elem < nItems ; elem++ ) {
     /* 3 components of gradient of species */
     spec_x = data[nItems*(0*nSpecs+specId)+elem] ;
```

```
      spec_y = data[nItems*(1*nSpecs+specId)+elem] ;
      spec_z = data[nItems*(2*nSpecs+specId)+elem] ;
      ...
   }
}
/* Alternative coding of the above. */
/* grad_? are pointers to 2-D arrays, and */
/* s_? are pointers to 1-D arrays. */
data = udfGetElmData( udfHd, UDF_ELM_GRAD_SPECIES ) ;
nSpecs = udfGetNumSpecs( udfHd ) ;
grad_x = &data[0*nItems*nSpecs] ;
grad_y = &data[1*nItems*nSpecs] ;
grad_z = &data[2*nItems*nSpecs] ;
for ( specId = 0 ; specId < nSpecs ; specId++ ) {
   s_x = &grad_x[nItems*specId] ;
   s_y = &grad_y[nItems*specId] ;
   s_z = &grad_z[nItems*specId] ;
   for ( elem = 0 ; elem < nItems ; elem++ ) {
      /* 3 components of gradient of species */
      spec_x = s_x[elem] ;
      spec_y = s_y[elem] ;
      spec_z = s_z[elem] ;
      ...
   }
}
```

The quantities UDF_ELM_TURBULENCE_Y and UDF_ELM_TURBULENCE_YPLUS are based on the distance to the nearest turbulence wall. These are defined by `TURBULENCE_WALL` and `SIMPLE_BOUNDARY_CONDITION type=wall` commands. UDF_ELM_TURBULENCE_YPLUS also uses the shear at these walls. The data for these two quantities are current (computed at the end of each stagger) while the other quantities are computed only at the end of each time step.

Errors

- This routine expects a valid *`udfHd`*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.
- *`dataName`* must be one of the values given above.
- The problem must contain the equation associated with the requested data.

udfGetElmJac()

Return the array storage for the element Jacobian at the quadrature point.

Syntax

```
jac = udfGetElmJac( udfHd, dataName ) ;
```

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName (integer)
: Symbolic name of the independent variable of the Jacobian.

UDF_ELM_JAC_VELOCITY	Velocity.
UDF_ELM_JAC_PRESSURE	Pressure.
UDF_ELM_JAC_TEMPERATURE	Temperature.
UDF_ELM_JAC_SPECIES	Species.
UDF_ELM_JAC_STRAIN_INVARIANT_2	Second invariant of strain rate tensor.

Return Value

***jac** (Real*)*
: Pointer to multi-dimensional real array of the element Jacobian. The first (fastest) dimension of the array is the number of elements, nItems, the second dimension is one or three for the components of the UDF type, and the third (slowest) dimension is one or three for the components of the given independent variable.

Description

This routine returns the array storage for the element Jacobian at the quadrature point. The Jacobian is defined here as the derivative of the UDF type with respect to the independent variable given by *dataName*. This Jacobian becomes part of the left-hand side of the solution strategy only. Therefore, it does not affect the final solution itself, but it may be very important to achieving a robust strategy to obtain the solution.

If the specified combination of type and independent variable is not supported, then a NULL is returned. If the combination is supported but the particular problem does not contain the appropriate physics, then the array storage is returned but not used. This feature is convenient for certain types of restart problems. For instance, if the type is gravity and *dataName* is UDF_ELM_JAC_TEMPERATURE, then storage for a two dimensional array (nItems, 3) is returned for you to fill with the appropriate data. But this

data will not be used if there is no temperature equation specified in the problem. However, if such an equation is specified on restart, then the data is used by the solver.

The returned array is three dimensional in general. The first (fastest) dimension of the array is the number of elements, nItems. The second dimension is three for a UDF type of gravity and one for all other types. The third (slowest) dimension is three for a *dataName* of `UDF_ELM_JAC_VELOCITY` and one for all others. If either the second or third dimension is one then the array may be treated as two dimensional. If both are one, then the array is one-dimensional.

For example, a temperature-dependent gravity UDF may be specified by:

```
Void usrGrav( UdfHd udfHd,
              Real* outVec,
              Integer nItems,
              Integer vecDim )
{
   Real grav0 ; /* expansivity */
   Integer i ; /* a running index */
   Real* usrVals ; /* user supplied values */
   Real* temp ; /* temperature */
   Real* gravJacTemp ; /* partial grav / partial temp */
   Real* xJacTemp ; /* x-gravJacTemp */
   Real* yJacTemp ; /* y-gravJacTemp */
   Real* zJacTemp ; /* z-gravJacTemp */
   Real* xGrav ; /* x-gravity */
   Real* yGrav ; /* y-gravity */
   Real* zGrav ; /* z-gravity */
/*-----------------------------------------------------------------------
* Get the parameters
*-----------------------------------------------------------------------
*/
   udfCheckNumUsrVals( udfHd, 1 ) ;
   usrVals = udfGetUsrVals( udfHd ) ;
   grav0 = usrVals[0] ;
   temp = udfGetElmData( udfHd, UDF_ELM_TEMPERATURE ) ;
   gravJacTemp = udfGetElmJac( udfHd, UDF_ELM_JAC_TEMPERATURE ) ;
   if ( gravJacTemp == Nil(Real*) ) {
      udfSetError( udfHd, "Invalid dataName" ) ;
   }
   xGrav = &outVec[0*nItems] ;
   yGrav = &outVec[1*nItems] ;
   zGrav = &outVec[2*nItems] ;
   xJacTemp = &gravJacTemp[0*nItems] ;
   yJacTemp = &gravJacTemp[1*nItems] ;
   zJacTemp = &gravJacTemp[2*nItems] ;
/*-----------------------------------------------------------------------
* Compute the gravity and its Jacobian with respect to temperature
*-----------------------------------------------------------------------
*/
for ( i = 0 ; i < nItems ; i++ ) {
   xGrav[i] = 0 ;
   yGrav[i] = -grav0 + temp[i] ;
   zGrav[i] = 0 ;
   xJacTemp[i] = 0 ;
   yJacTemp[i] = 1 ;
   zJacTemp[i] = 0 ;
   }
} /*
```

Errors

- This routine expects a valid `udfHd`.
- This routine may only be called within a Body Force, Material Model or Component Model user function.
- `dataName` must be one of the values given above.

udfGetElmAuxCrd()

Return the auxiliary element coordinates at the quadrature point.

Syntax

```
crd = udfGetElmAuxCrd( udfHd ) ;
```

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

crd *(Real*)*
: Pointer to two dimensional real array of auxiliary element coordinates. The first (fastest) dimension of the array is the number of elements, nItems, and the second (slower) dimension is three, for the x, y and z coordinates.

Description

This routine returns the auxiliary element coordinates at the quadrature point. If mesh displacement is active, the returned coordinates are for the current (deformed) configuration. For example,

```
Real* crd ;
Real x, y, z ;
Integer elem ;
...
crd = udfGetElmAuxCrd( udfHd ) ;
for ( elem = 0 ; elem < nItems ; elem++ ) {
    x = crd[0*nItems+elem] ;
    y = crd[1*nItems+elem] ;
    z = crd[2*nItems+elem] ;
    ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

udfGetElmAuxData()

Return auxiliary element data at the quadrature point.

Syntax

```
data = udfGetElmAuxData( udfHd, dataName ) ;
```

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName *(integer)*
: Symbolic name of the independent variable of the Jacobian.

UDF_ELM_VELOCITY	Velocity.
UDF_ELM_ACCELERATION	Acceleration.
UDF_ELM_PRESSURE	Pressure.
UDF_ELM_TEMPERATURE	Temperature.
UDF_ELM_SPECIES	Species.
UDF_ELM_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_ELM_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_ELM_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_ELM_GRAD_VELOCITY	Gradient of velocity.
UDF_ELM_GRAD_PRESSURE	Gradient of pressure.
UDF_ELM_GRAD_TEMPERATURE	Gradient of temperature.
UDF_ELM_GRAD_SPECIES	Gradient of species.
UDF_ELM_MESH_DISPLACEMENT	Mesh displacement.
UDF_ELM_MESH_VELOCITY	Mesh velocity.
UDF_ELM_MESH_GRAD_DISPLACEMENT	Gradient of mesh displacement.
UDF_ELM_MESH_GRAD_VELOCITY	Gradient of mesh velocity.
UDF_ELM_TURBULENCE_Y	Distance to nearest turbulence wall.

UDF_ELM_TURBULENCE_YPLUS Turbulence y+ based on distance to nearest turbulence wall and shear at that wall.

UDF_ELM_VISCOELASTIC Viscoelastic.

Return Value

data*(Real*)*

Pointer to one, two, or three dimensional real array of the requested data. The dimensions of the array depend on `dataName` as follows. If the third (slowest) dimension of the array is equal to one, then the array may be treated as two dimensional. If the second dimension is likewise one, then the array is one dimensional. The x, y, z components of the gradient for gradient quantities are always contained in the last nontrivial dimension.

dataName	**First Dimension**	**Second Dimension**	**Third Dimension**
UDF_ELM_VELOCITY	nItems	3	1
UDF_ELM_ACCLERATION	nItems	3	1
UDF_ELM_PRESSURE	nItems	1	1
UDF_ELM_TEMPERATURE	nItems	1	1
UDF_ELM_SPECIES	nItems	`udfGetNumSpecs()`	1
UDF_ELM_EDDY_VISCOSI	nItems	1	1
UDF_ELM_KINETIC_ENERG	nItems	1	1
UDF_ELM_EDDY_FREQUEN	nItems	1	1
UDF_ELM_GRAD_VELOCIT	nItems	3	3
UDF_ELM_GRAD_PRESSUR	nItems	3	1
UDF_ELM_GRAD_TEMPERAT	nItems	3	1
UDF_ELM_GRAD_SPECIES	nItems	`udfGetNumSpecs()`	3
UDF_ELM_MESH_DISPLACEM	nItems	3	1
UDF_ELM_MESH_VELOCIT	nItems	3	1
UDF_ELM_MESH_GRAD_DISPLA	nItems	3	3
UDF_ELM_MESH_GRAD_VELC	nItems	3	3
UDF_ELM_TURBULENCE_	nItems	1	1

dataName	First Dimension	Second Dimension	Third Dimension
UDF_ELM_TURBULENCE_YPl	nItems	1	1
UDF_ELM_VISCOELASTIC	nItems	6	1

Description

This routine returns the requested solution data from the auxiliary element, where the quadrature point is the same as that of the current element. For example,

```
Real* data ;
Real* grad_x ;
Real* grad_y ;
Real* grad_z ;
Real* s_x ;
Real* s_y ;
Real* s_z ;
Real u, v, w ;
Real u_x, v_x, w_x, u_y, v_y, w_y, u_z, v_z, w_z ;
Real spec_x, spec_y, spec_z ;
Integer elem, nSpecs, specId ;
...
/* velocity at the aux element */
data = udfGetElmAuxData( udfHd, UDF_ELM_VELOCITY ) ;
for ( elem = 0 ; elem < nItems ; elem++ ) {
   u = data[0*nItems+elem] ;
   v = data[1*nItems+elem] ;
   w = data[2*nItems+elem] ;
   ...
}
...
/* gradient of velocity at the aux element */
data = udfGetElmAuxData( udfHd, UDF_ELM_GRAD_VELOCITY ) ;
for ( elem = 0 ; elem < nItems ; elem++ ) {
   u_x = data[0*nItems+elem] ;
   v_x = data[1*nItems+elem] ;
   w_x = data[2*nItems+elem] ;
   u_y = data[3*nItems+elem] ;
   v_y = data[4*nItems+elem] ;
   w_y = data[5*nItems+elem] ;
   u_z = data[6*nItems+elem] ;
   v_z = data[7*nItems+elem] ;
   w_z = data[8*nItems+elem] ;
   ...
}
...
data = udfGetElmAuxData( udfHd, UDF_ELM_GRAD_SPECIES ) ;
nSpecs = udfGetNumSpecs( udfHd ) ;
 for ( specId = 0 ; specId < nSpecs ; specId++ ) {
   for ( elem = 0 ; elem < nItems ; elem++ ) {
   /* 3 components of gradient of species at the aux element */
   spec_x = data[nItems*(0*nSpecs+specId)+elem] ;
   spec_y = data[nItems*(1*nSpecs+specId)+elem] ;
   spec_z = data[nItems*(2*nSpecs+specId)+elem] ;
   ...
  }
```

```
}
```

The quantities UDF_ELM_TURBULENCE_Y and UDF_ELM_TURBULENCE_YPLUS are based on the distance to the nearest turbulence wall. These are defined by `TURBULENCE_WALL` and `SIMPLE_BOUNDARY_CONDITION`type=wall commands. UDF_ELM_TURBULENCE_YPLUS also uses the shear at these walls.

Errors

- This routine expects a valid *`udfHd`*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.
- *`dataName`* must be one of the values given above.
- The problem must contain the equation associated with the requested data.

udfGetElmRafData()

Return element running average field data at the quadrature point.

Syntax

```
data = udfGetElmRafData( udfHd, dataName ) ;
```

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName *(integer)*
: Symbolic name of the requested data.

UDF_ELM_VELOCITY	Velocity.
UDF_ELM_ACCELERATION	Acceleration.
UDF_ELM_PRESSURE	Pressure.
UDF_ELM_TEMPERATURE	Temperature.
UDF_ELM_SPECIES	Species.
UDF_ELM_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_ELM_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_ELM_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_ELM_GRAD_VELOCITY	Gradient of velocity.
UDF_ELM_GRAD_PRESSURE	Gradient of pressure.
UDF_ELM_GRAD_TEMPERATURE	Gradient of temperature.
UDF_ELM_GRAD_SPECIES	Gradient of species.
UDF_ELM_MESH_DISPLACEMENT	Mesh displacement.
UDF_ELM_MESH_VELOCITY	Mesh velocity.
UDF_ELM_MESH_GRAD_DISPLACEMENT	Gradient of mesh displacement.
UDF_ELM_MESH_GRAD_VELOCITY	Gradient of mesh velocity.
UDF_ELM_TURBULENCE_Y	Distance to nearest turbulence wall.

UDF_ELM_TURBULENCE_YPLUS Turbulence y+ based on distance to nearest turbulence wall and shear at that wall.

UDF_ELM_VISCOELASTIC Viscoelastic.

Return Value

data *(Real*)*

Pointer to one, two, or three dimensional real array of the requested data. The dimensions of the array depend on `dataName` as follows. If the third (slowest) dimension of the array is equal to one, then the array may be treated as two dimensional. If the second dimension is likewise one, then the array is one dimensional. The x, y, z components of the gradient for gradient quantities are always contained in the last nontrivial dimension.

dataName	**First Dimension**	**Second Dimension**	**Third Dimension**
UDF_ELM_VELOCITY	nItems	3	1
UDF_ELM_ACCLERAT	nItems	3	1
UDF_ELM_PRESSURE	nItems	1	1
UDF_ELM_TEMPERAT	nItems	1	1
UDF_ELM_SPECIES	nItems	`udfGetNumSpecs()`	1
UDF_ELM_EDDY_VIS	nItems	1	1
UDF_ELM_KINETIC_	nItems	1	1
UDF_ELM_EDDY_FRE	nItems	1	1
UDF_ELM_GRAD_VE	nItems	3	3
UDF_ELM_GRAD_PR	nItems	3	1
UDF_ELM_GRAD_TE	nItems	3	1
UDF_ELM_GRAD_SP	nItems	`udfGetNumSpecs()`	3
UDF_ELM_MESH_DIS	nItems	3	1
UDF_ELM_MESH_VE	nItems	3	1
UDF_ELM_MESH_GR	nItems	3	3
UDF_ELM_MESH_GR	nItems	3	3
UDF_ELM_TURBULEN	nItems	1	1
UDF_ELM_TURBULEN	nItems	1	1

dataName	First Dimension	Second Dimension	Third Dimension
UDF_ELM_VISCOELA	nItems	6	1

Description

This routine returns the requested solution data. It is used exactly the same way as `udfGetElmData`, but returns the running average field versions of the data.

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.
- *dataName* must be one of the values given above.
- The problem must contain the equation associated with the requested data.
- *running_average* must be turned on in the `EQUATION` command.

udfGetElmName()

Return the user-given name of the element set.

Syntax

```
name = udfGetElmName( udfHd ) ;
```

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

name *(string)*
: User-given name of the element set.

Description

This routine returns the user-given name of the element set associated with the user function. This facilitates correlating with the input file. For example,

```
String user_name ;
...
user_name = udfGetElmName( udfHd ) ;
udfPrintMess( "Inside element set with name <%s>", user_name ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

udfGetElmMedium()

Return the medium of the element set.

Syntax

```
name = udfGetElmMedium( udfHd ) ;
```

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

name *(integer)*
: Symbolic name corresponding to the medium of the element set.

PRM_MED_FLUID	Fluid.
PRM_MED_SOLID	Solid.
PRM_MED_SHELL	Shell.

Description

This routine returns the symbolic name for the medium (fluid, solid, or shell) of the element set associated with the user function. For example,

```
Integer medium ;
...
medium = udfGetElmMedium( udfHd ) ;
if ( medium == PRM_MED_FLUID ) {
...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function, or from an external code.

udfGetElmNElems()

Return the number of elements in the element set.

Syntax

nElems = udfGetElmNElems(udfHd) ;

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nElems *(integer)*
: Number of elements in the element set.

Description

This routine returns the number of elements in the element set. This is the number of rows of the parameter elements of the command `ELEMENT_SET` in the input file. For example,

```
Integer nElems ;
...
nElems = udfGetElmNElems( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

udfGetElmNElemNodes()

Return the number of element nodes for the element set.

Syntax

nElemNodes = udfGetElmNElemNodes(udfHd) ;

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nElemNodes *(integer)*
: Number of element nodes in the element set.

Description

This routine returns the number of element nodes for the element set. This depends only on the topology (test, wedges, and so on) of the elements in the element set. For example,

```
Integer nElemNodes ;
...
nElemNodes = udfGetElmNElemNodes( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

udfGetElmCnn()

Return the element connectivity.

Syntax

elemCnn = udfGetElmCnn(udfHd) ;

Type

User Defined Element

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

elemCnn *(Integer*)*
: Pointer to two dimensional integer array of element connectivity. The first (fastest) dimension of the array is the number of elements in the element set, nItems, and the second (slower) dimension is the number of element nodes, nElemNodes.

Description

This routine returns the array of element connectivity. This is the array, without the first column, of the parameter elements of the command `ELEMENT_SET` in the input file. For example,

```
Integer* elemCnn ;
Integer elemNode, elem, node, nElemNodes ;
...
nElemNodes = udfGetElmNElemNodes( udfHd ) ;
elemCnn = udfGetElmCnn( udfHd ) ;
for ( elem = 0 ; elem < nItems ; elem++ ) {
  for ( elemNode = 0 ; elemNode < nElemNodes ; elemNode++ ) {
    node = elemCnn[elemNode*nItems+elem] ;
    ...
   }
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Body Force, Material Model or Component Model user function.

Element Boundary Condition Routines

10

These support routines are accessible only by the Element Boundary Condition user functions, and by external output codes.

This chapter covers the following:

udfGetEbcType()

Return the type of the surface set.

Syntax

`type = udfGetEbcType( udfHd ) ;`

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

`udfHd`
: The opaque handle (pointer) which was passed to the user function.

Return Value

`type` *(integer)*
: Type of the element set.

Description

This routine returns the type of the surface set. For example,

```
Integer type ;
...
type = udfGetEbcType( udfHd ) ;
```

Errors

- This routine expects a valid *`udfHd`*.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcQuadType()

Return the type of the quadrature rule for the surface set.

Syntax

quadType = udfGetEbcQuadType(udfHd) ;

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

quadType *(integer)*
: Type of the quadrature rule for the surface set.

Description

This routine returns the quadrature type of the surface set. For example,

```
Integer quadType ;
...
quadType = udfGetEbcQuadType( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcNQuads()

Return the number of quadrature points of the quadrature rule for the surface set.

Syntax

```
nQuads = udfGetEbcNQuads( udfHd ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nQuads *(integer)*
: Number of quadrature points of the quadrature rule for the surface set.

Description

This routine returns the number of quadrature points of the quadrature rule for the surface set. For example,

```
Integer nQuads ;
...
nQuads = udfGetEbcNQuads( udfHd )
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcQuadId()

Return the quadrature point ID of the quadrature rule for the surface set.

Syntax

```
quadId = udfGetEbcQuadId( udfHd ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

QuadId *(integer)*
: Quadrature ID of the quadrature rule for the surface set.

Description

This routine returns the quadrature point ID of the quadrature rule for the surface set. For example,

```
Integer quadId ;
...
quadId = udfGetEbcQuadId( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcTime()

Return the current time.

Syntax

```
time = udfGetEbcTime( udfHd ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

time *(real)*
: Current time.

Description

This routine returns the current time. For example,

```
Real time ;
...
time = udfGetEbcTime( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcIds()

Return element boundary condition user surface numbers.

Syntax

`surfIds = udfGetEbcIds( udfHd ) ;`

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

`udfHd`
: The opaque handle (pointer) which was passed to the user function.

Return Value

`surfIds` *(Integer*)*
: Pointer to one dimensional integer array of user surface numbers. The array dimension is the number of surfaces, nItems.

Description

This routine returns the array of user surface numbers. This is the second array column of the parameter surfaces of the command `ELEMENT_BOUNDARY_CONDITION` in the input file. For example,

```
Integer* surfIds ;
Integer usrSurf, surf ;
...
surfIds = udfGetEbcIds( udfHd ) ;
for ( surf = 0 ; surf < nItems ; surf++ ) {
   usrSurf = surfIds[surf] ;
   ...
}
```

Errors

- This routine expects a valid *`udfHd`*.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcCrd()

Return coordinates of the surface quadrature point.

Syntax

```
crd = udfGetEbcCrd( udfHd ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

crd *(Real*)*
: Pointer to two dimensional real array of surface coordinates. The first (fastest) dimension of the array is the number of surfaces, nItems, and the second (slower) dimension is three, for the x, y and z coordinates.

Description

This routine returns the coordinates of the surface quadrature point. If mesh displacement is active, the returned coordinates are for the current (deformed) configuration. For example,

```
Real* crd ;
Real x, y, z ;
Integer surf ;
...
crd = udfGetEbcCrd( udfHd ) ;
for ( surf = 0 ; surf < nItems ; surf++ ) {
  x = crd[0*nItems+surf] ;
  y = crd[1*nItems+surf] ;
  z = crd[2*nItems+surf] ;
  ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcWDetJ()

Return the weighted determinant of the surface Jacobian at the surface quadrature point.

Syntax

```
wghtDetJac = udfGetEbcWDetJ( udfHd ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

wghtDetJac *(Real*)*
: Pointer to one dimensional real array of the weighted determinant of the surface Jacobian at the surface quadrature point. The array dimension is the number of surfaces, nItems.

Description

This routine returns the weighted determinant of the surface Jacobian at the surface quadrature point. If mesh displacement is active, the returned weighted determinant of the surface Jacobian is for the current (deformed) configuration. The unit of this quantity is area (as opposed to the analogous element quantity, where the unit is volume). For example,

```
Real* wghtDetJac ;
Real wDetJ ;
Integer surf ;
...
wghtDetJac = udfGetEbcWDetJ( udfHd ) ;
for ( surf = 0 ; surf < nItems ; surf++ ) {
   wDetJ = wghtDetJac[surf] ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcNormDir()

Return the outward normal direction of the surface at the surface quadrature point.

Syntax

```
normDir = udfGetEbcNormDir( udfHd ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

normDir *(Real*)*
: Pointer to two dimensional real array of the outward normal direction of the surface at the surface quadrature point. The first (fastest) dimension of the array is the number of surfaces, nItems, and the second (slower) dimension is three, for the x, y and z directions.

Description

This routine returns the outward normal direction of the surface at the surface quadrature point. For example,

```
Real* normDir ;
Real x_dir, y_dir, z_dir ;
Integer surf ;
...
normDir = udfGetEbcNormDir( udfHd ) ;
for ( surf = 0 ; surf < nItems ; surf++ ) {
   x_dir = normDir[0*nItems+surf] ;
   y_dir = normDir[1*nItems+surf] ;
   z_dir = normDir[2*nItems+surf] ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcCovar()

Return the surface covariant at the surface quadrature point.

Syntax

```
covar = udfGetEbcCovar( udfHd ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd

The opaque handle (pointer) which was passed to the user function.

Return Value

covar *(Real*)*

Pointer to two dimensional real array of the covariant. The first (fastest) dimension of the array is the number of surfaces, nItems, and the second (slower) dimension is six, for the xx, yy, zz, xy, yz and zx components.

Description

This routine returns the surface covariant at the surface quadrature point. The surface covariant is given by

$$g_{ij} = \frac{\partial \xi_\alpha}{\partial x_i}\frac{\partial \xi_\alpha}{\partial x_j} \tag{3}$$

where $\xi_\alpha \ (\alpha = 1, 2, 3)$ is the local parent element coordinate (with being normal to the surface) and $x_i \ (i = 1, 2, 3)$ is the global coordinate. The latter is for the current (deformed) configuration if mesh displacement is active. For example,

```
Real* covar ;
Real xx_covar, yy_covar, zz_covar ;
Real xy_covar, yz_covar, zx_covar ;
Integer surf ;
...
covar = udfGetEbcCovar( udfHd ) ;
for ( surf = 0 ; surf < nItems ; surf++ ) {
  xx_covar = covar[0*nItems+surf] ;
  yy_covar = covar[1*nItems+surf] ;
  zz_covar = covar[2*nItems+surf] ;
  xy_covar = covar[3*nItems+surf] ;
  yz_covar = covar[4*nItems+surf] ;
  zx_covar = covar[5*nItems+surf] ;
  ...
}
```

Errors

- This routine expects a valid `udfHd`.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcContvar()

Return the surface contravariant at the surface quadrature point.

Syntax

```
contvar = udfGetEbcContvar( udfHd ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd

The opaque handle (pointer) which was passed to the user function.

Return Value

contvar *(Real*)*

Pointer to two dimensional real array of the contravariant. The first (fastest) dimension of the array is the number of surfaces, nItems, and the second (slower) dimension is six, for the xx, yy, zz, xy, yz and zx components.

Description

This routine returns the surface contravariant at the surface quadrature point. The surface contravariant is given by

$$g^{ij} = \frac{\partial x_i}{\partial \xi_\alpha}\frac{\partial x_j}{\partial \xi_\alpha} \tag{4}$$

where $\xi_\alpha \ (\alpha = 1, 2, 3)$ is the local parent element coordinate (with ξ_3 being normal to the surface) and $x_i \ (i = 1, 2, 3)$ is the global coordinate. The latter is for the current (deformed) configuration if mesh displacement is active. For example,

```
Real* contvar ;
Real xx_contvar, yy_contvar, zz_contvar ;
Real xy_contvar, yz_contvar, zx_contvar ;
Integer surf ;
...
contvar = udfGetEbcContvar( udfHd ) ;
for ( surf = 0 ; surf < nItems ; surf++ ) {
   xx_contvar = contvar[0*nItems+surf] ;
   yy_contvar = contvar[1*nItems+surf] ;
   zz_contvar = contvar[2*nItems+surf] ;
   xy_contvar = contvar[3*nItems+surf] ;
   yz_contvar = contvar[4*nItems+surf] ;
   zx_contvar = contvar[5*nItems+surf] ;
```

```
    ...
}
```

Errors

- This routine expects a valid `udfHd`.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcJac()

Return the array storage for the surface element Jacobian at the quadrature point.

Syntax

```
jac = udfGetEbcJac( udfHd, dataName ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName (integer)
: Symbolic name of the independent variable of the Jacobian.

UDF_EBC_JAC_VELOCITY	Velocity.
UDF_EBC_JAC_PRESSURE	Pressure.
UDF_EBC_JAC_TEMPERATURE	Temperature.
UDF_EBC_JAC_SPECIES	Species.

Return Value

data *(Real*)*
: Pointer to multi-dimensional real array of the surface element Jacobian. The first (fastest) dimension of the array is the number of elements, nItems, the second dimension is one or three for the components of the UDF type, and the third (slowest) dimension is one or three for the components of the given independent variable.

Description

This routine returns the array storage for the surface element Jacobian at the quadrature point. The Jacobian is defined here as the derivative of the UDF type with respect to the independent variable given by *dataName*. This Jacobian becomes part of the left-hand side of the solution strategy only. Therefore, it does not affect the final solution itself, but it may be very important to achieving a robust strategy to obtain the solution.

If the specified combination of type and independent variable is not supported, then a NULL is returned. If the combination is supported but the particular problem does not contain the appropriate physics, then the array storage is returned but not used. This feature is convenient for certain types of restart problems. For instance, if the type is gravity and *dataName* is UDF_EBC_JAC_TEMPERATURE, then storage for a two dimensional array (nItems, three) is returned for you to fill with the appropriate data. But this data will not be used if there is no temperature equation specified in the problem. However, if such an equation is specified on restart, then the data is used by the solver.

The returned array is three-dimensional in general. The first (fastest) dimension of the array is the number of elements, nItems. The second dimension is three for a UDF for the tangential traction and one for all other variables. The third (slowest) dimension is three for a *dataName* of UDF_EBC_JAC_VELOCITY and one for all others. If either the second or third dimension is one then the array may be treated as two-dimensional, as in the above example. If both are one, then the array is one-dimensional.

This function is used very similarly to `udfGetElmJac`.

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.
- *dataName* must be one of the values given above.

udfGetEbcData()

Return data at the surface quadrature point.

Syntax

```
data = udfGetEbcData( udfHd, dataName ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName (integer)
: Symbolic name of the requested data.

UDF_EBC_VELOCITY	Velocity.
UDF_EBC_ACCELERATION	Acceleration.
UDF_EBC_PRESSURE	Pressure.
UDF_EBC_TEMPERATURE	Temperature.
UDF_EBC_SPECIES	Species.
UDF_EBC_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_EBC_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_EBC_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_EBC_GRAD_VELOCITY	Gradient of velocity.
UDF_EBC_GRAD_PRESSURE	Gradient of pressure.
UDF_EBC_GRAD_TEMPERATURE	Gradient of temperature.
UDF_EBC_GRAD_SPECIES	Gradient of species.
UDF_EBC_MESH_DISPLACEMENT	Mesh displacement.
UDF_EBC_MESH_VELOCITY	Mesh velocity.

Return Value

data *(Real*)*
: Pointer to one, two, or three dimensional real array of the requested data. The dimensions of the array depend on *dataName* as follows. If the third (slowest) dimension of the array is equal to one,

then the array may be treated as two dimensional. If the second dimension is likewise one, then the array is one dimensional. The x, y, z components of the gradient for gradient quantities are always contained in the last nontrivial dimension.

dataName	First Dimension	Second Dimension	Third Dimension
UDF_EBC_VELOCITY	nItems	3	1
UDF_EBC_ACCLERAT	nItems	3	1
UDF_EBC_PRESSURE	nItems	1	1
UDF_EBC_TEMPERAT	nItems	1	1
UDF_EBC_SPECIES	nItems	`udfGetNumSpecs()`	1
UDF_EBC_EDDY_VIS	nItems	1	1
UDF_EBC_KINETIC_	nItems	1	1
UDF_EBC_EDDY_FRE	nItems	1	1
UDF_EBC_GRAD_VE	nItems	3	3
UDF_EBC_GRAD_PR	nItems	3	1
UDF_EBC_GRAD_TE	nItems	3	1
UDF_EBC_GRAD_SP	nItems	`udfGetNumSpecs()`	3
UDF_EBC_MESH_DIS	nItems	3	1
UDF_EBC_MESH_VE	nItems	3	1

Description

This routine returns the requested solution data at the surface quadrature point. For example,

```
Real* data ;
Real u, v, w ;
Real u_x, v_x, w_x, u_y, v_y, w_y, u_z, v_z, w_z ;
Real spec_x, spec_y, spec_z ;
Integer surf, nSpecs, specId ;
...
data = udfGetEbcData( udfHd, UDF_EBC_VELOCITY ) ;
for ( surf = 0 ; surf < nItems ; surf++ ) {
   u = data[0*nItems+surf] ;
   v = data[1*nItems+surf] ;
   w = data[2*nItems+surf] ;
   ...
}
...
data = udfGetEbcData( udfHd, UDF_EBC_GRAD_VELOCITY ) ;
for ( surf = 0 ; surf < nItems ; surf++ ) {
```

```
   u_x = data[0*nItems+surf] ;
   v_x = data[1*nItems+surf] ;
   w_x = data[2*nItems+surf] ;
   u_y = data[3*nItems+surf] ;
   v_y = data[4*nItems+surf] ;
   w_y = data[5*nItems+surf] ;
   u_z = data[6*nItems+surf] ;
   v_z = data[7*nItems+surf] ;
   w_z = data[8*nItems+surf] ;
   ...
}
...
data = udfGetEbcData( udfHd, UDF_EBC_GRAD_SPECIES ) ;
nSpecs = udfGetNumSpecs( udfHd ) ;
for ( specId = 0 ; specId < nSpecs ; specId++ ) {
   for ( surf = 0 ; surf < nItems ; surf++ ) {
      /* 3 components of gradient of species */
      spec_x = data[nItems*(0*nSpecs+specId)+surf] ;
      spec_y = data[nItems*(1*nSpecs+specId)+surf] ;
      spec_z = data[nItems*(2*nSpecs+specId)+surf] ;
      ...
   }
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.
- *dataName* must be one of the values given above.
- The problem must contain the equation associated with the requested data.

udfGetEbcRafData()

Return running average field data at the surface quadrature point.

Syntax

```
data = udfGetEbcRafData( udfHd, dataName ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName (integer)
: Symbolic name of the requested data.

UDF_EBC_VELOCITY	Velocity.
UDF_EBC_ACCELERATION	Acceleration.
UDF_EBC_PRESSURE	Pressure.
UDF_EBC_TEMPERATURE	Temperature.
UDF_EBC_SPECIES	Species.
UDF_EBC_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_EBC_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_EBC_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_EBC_GRAD_VELOCITY	Gradient of velocity.
UDF_EBC_GRAD_PRESSURE	Gradient of pressure.
UDF_EBC_GRAD_TEMPERATURE	Gradient of temperature.
UDF_EBC_GRAD_SPECIES	Gradient of species.
UDF_EBC_MESH_DISPLACEMENT	Mesh displacement.
UDF_EBC_MESH_VELOCITY	Mesh velocity.

Return Value

data *(Real*)*
: Pointer to one, two, or three dimensional real array of the requested data. The dimensions of the array depend on *dataName* as follows. If the third (slowest) dimension of the array is equal to one,

then the array may be treated as two dimensional. If the second dimension is likewise one, then the array is one dimensional. The x, y, z components of the gradient for gradient quantities are always contained in the last nontrivial dimension.

dataName	First Dimension	Second Dimension	Third Dimension
UDF_EBC_VELOCITY	nItems	3	1
UDF_EBC_ACCLERAT	nItems	3	1
UDF_EBC_PRESSURE	nItems	1	1
UDF_EBC_TEMPERAT	nItems	1	1
UDF_EBC_SPECIES	nItems	`udfGetNumSpecs()`	1
UDF_EBC_EDDY_VIS	nItems	1	1
UDF_EBC_KINETIC_	nItems	1	1
UDF_EBC_EDDY_FRE	nItems	1	1
UDF_EBC_GRAD_VE	nItems	3	3
UDF_EBC_GRAD_PR	nItems	3	1
UDF_EBC_GRAD_TE	nItems	3	1
UDF_EBC_GRAD_SP	nItems	`udfGetNumSpecs()`	3
UDF_EBC_MESH_DIS	nItems	3	1
UDF_EBC_MESH_VE	nItems	3	1

Description

This routine returns the requested solution data at the surface quadrature point. It is used exactly the same way as, but returns the running average field versions of the data.

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.
- *dataName* must be one of the values given above.
- The problem must contain the equation associated with the requested data.
- *running_average* must be turned on in the `EQUATION` command.

udfGetEbcName()

Return the user-given name of the boundary element set.

Syntax

```
name = udfGetEbcName( udfHd ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

name *(string)*
: User-given name of the boundary element set.

Description

This routine returns the user-given name of the boundary element set associated with the user function. This facilitates correlating with the input file. For example,

```
String user_name ;
...
user_name = udfGetEbcName( udfHd ) ;
udfPrintMess( "Inside boundary element set with name <%s>", user_name ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcMedium()

Return the medium of the boundary element set.

Syntax

```
name = udfGetEbcMedium( udfHd ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

name *(integer)*
: Symbolic name corresponding to the medium of the boundary element set.

PRM_MED_FLUID	Fluid.
PRM_MED_SOLID	Solid.
PRM_MED_SHELL	Shell.

Description

This routine returns the medium (fluid, solid or shell) of the boundary element set associated with the user function. For example,

```
Integer medium ;
...
medium = udfGetEbcMedium( udfHd ) ;
if ( medium == PRM_MED_FLUID ) {
...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcNElems()

Return the number of elements in the boundary element set.

Syntax

```
nElems = udfGetEbcNElems( udfHd ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nElems *(integer)*
: Number of elements in the boundary element set.

Description

This routine returns the number of elements in the boundary element set. This is the number of rows of the parameter elements of the command `ELEMENT_BOUNDARY_CONDITION` in the input file. For example,

```
Integer nElems ;
...
nElems = udfGetEbcNElems( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcNElemNodes()

Return the number of element nodes for the boundary element set.

Syntax

nElemNodes = udfGetEbcNElemNodes(udfHd) ;

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nElemNodes *(integer)*
: Number of element nodes for the boundary element set.

Description

This routine returns the number of element nodes for the boundary element set. This depends only on the topology (tets, wedges, and so on) of the elements in the boundary element set. For example,

```
Integer nElemNodes ;
...
nElemNodes = udfGetEbcNElemNodes( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.

udfGetEbcCnn()

Return the element connectivity.

Syntax

```
elemCnn = udfGetEbcCnn( udfHd ) ;
```

Type

AcuSolve User-Defined Example Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

elemCnn *(Integer*)*
: Pointer to two dimensional integer array of element connectivity. The first (fast) dimension of the array is the number of elements in the boundary element set, nItems, and the second (slow) dimension is the number of element nodes, nElemNodes.

Description

This routine returns the array of element connectivity. This is the array, without the first column, of the parameter elements of the command `ELEMENT_BOUNDARY_CONDITION` in the input file. For example,

```
Integer* elemCnn ;
Integer elemNode, elem, node, nElemNodes ;
...
nElemNodes = udfGetEbcNElemNodes( udfHd ) ;
elemCnn = udfGetEbcCnn( udfHd ) ;
for ( elem = 0 ; elem < nItems ; elem++ ) {
   for ( elemNode = 0 ; elemNode < nElemNodes ; elemNode++ ) {
       node = elemCnn[elemNode*nItems+elem] ;
       ...
   }
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Element Boundary Condition user function.

Nodal Boundary Condition Routines

11

These support routines are accessible only by the Nodal Boundary Condition user functions.

This chapter covers the following:

udfGetNbcIds()

Return the nodal boundary condition user node numbers.

Syntax

```
nodeIds = udfGetNbcIds( udfHd ) ;
```

Type

AcuSolve User-Defined Nodal Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nodeIds *(Integer*)*
: Pointer to one dimensional integer array of user node numbers. This array has a dimension of number of nodes, nItems.

Description

This routine returns the array of user node numbers. For example,

```
Integer* nodeIds ;
Integer usrNode, node ;
...
nodeIds = udfGetNbcIds( udfHd ) ;
for ( node = 0 ; node < nItems ; node++ ) {
   usrNode = nodeIds[node] ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Nodal Boundary Condition user function.

udfGetNbcCrd()

Return the nodal coordinates for the nodal boundary condition.

Syntax

```
crd = udfGetNbcCrd( udfHd ) ;
```

Type

AcuSolve User-Defined Nodal Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

crd *(Real*)*
: Pointer to two dimensional real array of nodal coordinates. The first (fast) dimension of the array is the number of nodes, nItems, and the second (slow) dimension is three, for the x, y and z coordinates.

Description

This routine returns the nodal coordinates. If mesh displacement is active, the returned coordinates are for the current (deformed) configuration. For example,

```
Real* crd ;
Real x, y, z ;
Integer node ;
...
crd = udfGetNbcCrd( udfHd ) ;
for ( node = 0 ; node < nItems ; node++ ) {
   x = crd[0*nItems+node] ;
   y = crd[1*nItems+node] ;
   z = crd[2*nItems+node] ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Nodal Boundary Condition user function.

udfGetNbcRefCrd()

Return the initial condition nodal coordinates for the nodal boundary condition.

Syntax

```
crd = udfGetNbcRefCrd( udfHd ) ;
```

Type

AcuSolve User-Defined Nodal Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

crd *(Real*)*
: Pointer to two dimensional real array of initial condition nodal coordinates. The first (fast) dimension of the array is the number of nodes, nItems, and the second (slow) dimension is three, for the x, y and z coordinates.

Description

This routine returns the initial condition nodal coordinates. If mesh displacement is active, the returned coordinates include the initial conditions for the mesh displacement; that is, the initial deformed configuration. For example,

```
Real* crd ;
Real x, y, z ;
Integer node ;
...
crd = udfGetNbcRefCrd( udfHd ) ;
for ( node = 0 ; node < nItems ; node++ ) {
   x = crd[0*nItems+node] ;
   y = crd[1*nItems+node] ;
   z = crd[2*nItems+node] ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Nodal Boundary Condition user function.

udfGetNbcData()

Return nodal solution data for the nodal boundary condition.

Syntax

```
data = udfGetNbcData( udfHd, dataName ) ;
```

Type

AcuSolve User-Defined Nodal Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName *(integer)*
: Symbolic name of the requested data.

UDF_NBC_VELOCITY	Velocity.
UDF_NBC_ACCELERATION	Acceleration.
UDF_NBC_PRESSURE	Pressure.
UDF_NBC_TEMPERATURE	Temperature.
UDF_NBC_DENSITY	Density.
UDF_NBC_SPECIES	Species.
UDF_NBC_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_NBC_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_NBC_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_NBC_EDDY_TIME	Turbulence eddy time.
UDF_NBC_TRANSITION_RETHETA	Transition Re-Theta.
UDF_NBC_INTERMITTENCY	Transition intermittency.
UDF_NBC_MESH_DISPLACEMENT	Mesh displacement.
UDF_NBC_MESH_VELOCITY	Mesh velocity.
UDF_NBC_TURBULENCE_Y	Distance to nearest turbulence wall.
UDF_NBC_TURBULENCE_YPLUS	Turbulence y+ based on distance to nearest turbulence wall and shear at that wall.

UDF_NBC_VISCOELASTIC Viscoelastic.

Return Value

data *(Real*)*

Pointer to one or two dimensional real array of the requested data. The dimensions of the array depend on `dataName` as follows. If the second dimension is one, then the array may be treated as one dimensional.

dataName	First Dimension	Second Dimension
UDF_NBC_VELOCITY	nItems	3
UDF_NBC_ACCLERATION	nItems	3
UDF_NBC_PRESSURE	nItems	1
UDF_NBC_TEMPERATURE	nItems	1
UDF_NBC_SPECIES	nItems	`udfGetNumSpecs()`
UDF_NBC_EDDY_VISCOSITY	nItems	1
UDF_NBC_MESH_DISPLACEM	nItems	3
UDF_NBC_MESH_VELOCITY	nItems	3
UDF_NBC_KINETIC_ENERGY	nItems	1
UDF_NBC_EDDY_FREQUENCY	nItems	1
UDF_NBC_EDDY_TIME	nItems	1
UDF_NBC_TRANSITION_RETH	nItems	1
UDF_NBC_INTERMITTENCY	nItems	1
UDF_NBC_TURBULENCE_Y	nItems	1
UDF_NBC_TURBULENCE_YPLU	nItems	1
UDF_NBC_VISCOELASTIC	nItems	6

Description

This routine returns the requested nodal solution data. For example,

```
Real* data ;
Real u, v, w ;
Real spec ;
Integer node, nSpecs, specId ;
```

```
...
data = udfGetNbcData( udfHd, UDF_NBC_VELOCITY ) ;
for ( node = 0 ; node < nItems ; node++ ) {
  u = data[0*nItems+node] ;
  v = data[1*nItems+node] ;
  w = data[2*nItems+node] ;
  ...
}
...
data = udfGetNbcData( udfHd, UDF_NBC_SPECIES ) ;
nSpecs = udfGetNumSpecs( udfHd ) ;
for ( specId = 0 ; specId < nSpecs ; specId++ ) {
  for ( node = 0 ; node < nItems ; node++ ) {
    spec = data[specId*nItems+node] ;
    ...
  }
}
```

The quantities UDF_NBC_TURBULENCE_Y and UDF_NBC_TURBULENCE_YPLUS are based on the distance to the nearest turbulence wall. These are defined by `TURBULENCE_WALL` and `SIMPLE_BOUNDARY_CONDITION`type=wall commands. UDF_NBC_TURBULENCE_YPLUS also uses the shear at these walls.

Errors

- This routine expects a valid *`udfHd`*.
- This routine may only be called within an Nodal Boundary Condition user function.
- *`dataName`* must be one of the values given above.
- The problem must contain the equation associated with the requested data.

udfCheckNbcNumAuxs()

Check the number of auxiliary nodes given in the input file.

Syntax

udfCheckNbcNumAuxs(udfHd, nAuxs) ;

Type

AcuSolve User-Defined Nodal Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

nAuxs (integer)
: Number of auxiliary nodes that the function expects.

Return Value

None.

Description

This routine verifies that the number of auxiliary nodes supplied in the input file is the same as the second argument. If it is not, an error message is issued and the solver exits. For example,

```
udfCheckNbcNumAuxs( udfHd, 3 ) ;
```

Errors

- This routine expects a valid *udfHd*.

udfGetNbcNumAuxs()

Return the number of auxiliary nodes given in the input file.

Syntax

```
nAuxs = udfGetNbcNumAuxs( udfHd ) ;
```

Type

AcuSolve User-Defined Nodal Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nAuxs *(integer)*
: Number of auxiliary nodes given in the input file.

Description

This routine returns the number of auxiliary nodes given in the input file. For example,

```
Integer nAuxs ;
...
nAuxs = udfGetNbcNumAuxs( udfHd ) ;
```

Errors

This routine expects a valid *udfHd*.

udfGetNbcAuxIds()

Return the nodal boundary condition user auxiliary node numbers.

Syntax

```
auxNodeIds = udfGetNbcAuxIds( udfHd ) ;
```

Type

AcuSolve User-Defined Nodal Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

auxNodeIds *(Integer*)*
: Pointer to two dimensional integer array of user auxiliary node numbers. The first (fast) dimension of the array is the number of nodes, nItems, and the second (slow) dimension is the number of auxiliary nodes, nAuxs.

Description

This routine returns the array of user auxiliary node numbers. This array is defined by the *auxiliary_nodes* parameter of the NODAL_BOUNDARY_CONDITION command, starting with the second column. For example,

```
Integer* auxNodeIds ;
Integer usrAuxNode0, usrAuxNode1, usrAuxNode2, node, aux ;
...
udfCheckNbcNumAuxs( udfHd, 3 ) ;
auxNodeIds = udfGetNbcAuxIds( udfHd ) ;
for ( node = 0 ; node < nItems ; node++ ) {
  usrAuxNode0 = auxNodeIds[0*nItems+node] ;
  usrAuxNode1 = auxNodeIds[1*nItems+node] ;
  usrAuxNode2 = auxNodeIds[2*nItems+node] ;
  ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Nodal Boundary Condition user function.
- The corresponding command must have a valid *auxiliary_nodes* parameter.

udfGetNbcAuxCrd()

Return the nodal coordinates for the nodal boundary condition at the auxiliary nodes.

Syntax

```
auxCrd = udfGetNbcAuxCrd( udfHd ) ;
```

Type

AcuSolve User-Defined Nodal Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

auxCrd *(Real*)*
: Pointer to three dimensional real array of nodal coordinates at the auxiliary nodes. The first (fastest) dimension of the array is the number of nodes, nItems, the second dimension is the number of auxiliary nodes, nAuxs, and the third (slowest) dimension is three, for the x, y and z coordinates. If nAuxs is one, then the array may be treated as two dimensional.

Description

This routine returns the nodal coordinates at the auxiliary nodes. If mesh displacement is active, the returned coordinates are for the current (deformed) configuration. For example,

```
Real* auxCrd ;
Real x0, y0, z0, x1, y1, z1, x2, y2, z2;
Integer node, nAuxs=3 ;
...
udfCheckNbcNumAuxs( udfHd, nAuxs ) ;
auxCrd = udfGetNbcAuxCrd( udfHd ) ;
for ( node = 0 ; node < nItems ; node++ ) {
  x0 = auxCrd[nItems*(0*nAuxs+0)+node] ;
  y0 = auxCrd[nItems*(3*nAuxs+0)+node] ;
  z0 = auxCrd[nItems*(6*nAuxs+0)+node] ;
  x1 = auxCrd[nItems*(0*nAuxs+1)+node] ;
  y1 = auxCrd[nItems*(3*nAuxs+1)+node] ;
  z1 = auxCrd[nItems*(6*nAuxs+1)+node] ;
  x2 = auxCrd[nItems*(0*nAuxs+2)+node] ;
  y2 = auxCrd[nItems*(3*nAuxs+2)+node] ;
  z2 = auxCrd[nItems*(6*nAuxs+2)+node] ;
  ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Nodal Boundary Condition user function.
- The corresponding command must have a valid *auxiliary_nodes* parameter.

udfGetNbcAuxRefCrd()

Return the initial condition nodal coordinates for the nodal boundary condition at the auxiliary nodes.

Syntax

```
auxCrd = udfGetNbcAuxRefCrd( udfHd ) ;
```

Type

AcuSolve User-Defined Nodal Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

auxCrd *(Real*)*
: Pointer to three dimensional real array of reference nodal coordinates at the auxiliary nodes. The first (fastest) dimension of the array is the number of nodes, nItems, the second dimension is the number of auxiliary nodes, nAuxs, and the third (slowest) dimension is three, for the x, y and z coordinates. If nAuxs is one, then the array may be treated as two dimensional.

Description

This routine returns the initial condition nodal coordinates at the auxiliary nodes. If mesh displacement is active, the returned coordinates include the initial conditions for the mesh displacement; that is, the initial deformed configuration. For example,

```
Real* auxCrd ;
Real x, y, z ;
Integer node ;
...
udfCheckNbcNumAuxs( udfHd, 1 ) ;
auxCrd = udfGetNbcAuxRefCrd( udfHd ) ;
for ( node = 0 ; node < nItems ; node++ ) {
  x = auxCrd[0*nItems+node] ;
  y = auxCrd[1*nItems+node] ;
  z = auxCrd[2*nItems+node] ;
  ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Nodal Boundary Condition user function.
- The corresponding command must have a valid *auxiliary_nodes* parameter.

udfGetNbcAuxData()

Return nodal solution data for the nodal boundary condition at the auxiliary nodes.

Syntax

```
auxdata = udfGetNbcAuxData( udfHd, dataName ) ;
```

Type

AcuSolve User-Defined Nodal Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName *(integer)*
: Symbolic name of the requested data.

UDF_NBC_VELOCITY	Velocity.
UDF_NBC_ACCELERATION	Acceleration.
UDF_NBC_PRESSURE	Pressure.
UDF_NBC_TEMPERATURE	Temperature.
UDF_NBC_DENSITY	Density.
UDF_NBC_SPECIES	Species.
UDF_NBC_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_NBC_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_NBC_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_NBC_EDDY_TIME	Turbulence eddy time.
UDF_NBC_TRANSITION_RETHETA	Transition Re-Theta.
UDF_NBC_INTERMITTENCY	Transition intermittency.
UDF_NBC_MESH_DISPLACEMENT	Mesh displacement.
UDF_NBC_MESH_VELOCITY	Mesh velocity.
UDF_NBC_TURBULENCE_Y	Distance to nearest turbulence wall.
UDF_NBC_TURBULENCE_YPLUS	Turbulence y+ based on distance to nearest turbulence wall and shear at that wall.

Return Value

auxdata *(Real*)*

Pointer to three dimensional real array of the requested data at the auxiliary nodes. The dimensions of the array depend on `dataName` as follows. nAuxs is the number of auxiliary nodes. If either the second or third dimension is one, then the array may be treated as two dimensional. If both are one, then the array may be treated as one dimensional.

dataName	First Dimension	Second Dimension	Third Dimension
UDF_NBC_VELOCIT	nItems	nAuxs	3
UDF_NBC_ACCLERA	nItems	nAuxs	3
UDF_NBC_PRESSUR	nItems	nAuxs	1
UDF_NBC_TEMPERA	nItems	nAuxs	1
UDF_NBC_SPECIES	nItems	nAuxs	`udfGetNumSpecs()`
UDF_NBC_EDDY_VI	nItems	nAuxs	1
UDF_NBC_KINETIC_	nItems	nAuxs	1
UDF_NBC_EDDY_FR	nItems	nAuxs	1
UDF_NBC_EDDY_TI	nItems	nAuxs	1
UDF_NBC_TRANSIT	nItems	nAuxs	1
UDF_NBC_INTERMI	nItems	nAuxs	1
UDF_NBC_MESH_DI	nItems	nAuxs	3
UDF_NBC_MESH_VE	nItems	nAuxs	3
UDF_NBC_TURBULE	nItems	nAuxs	1
UDF_NBC_TURBULE	nItems	nAuxs	1

Description

This routine returns the requested nodal solution data at the auxiliary nodes. For example,

```
Real* auxData ;
Real u, v, w ;
Real spec ;
Integer node, nSpecs, specId ;
...
udfCheckNbcNumAuxs( udfHd, 1 ) ;
auxData = udfGetNbcAuxData( udfHd, UDF_NBC_VELOCITY ) ;
for ( node = 0 ; node < nItems ; node++ ) {
```

```
  u = auxData[0*nItems+node] ;
  v = auxData[1*nItems+node] ;
  w = auxData[2*nItems+node] ;
  ...
}
...
auxData = udfGetNbcAuxData( udfHd, UDF_NBC_SPECIES ) ;
nSpecs = udfGetNumSpecs( udfHd ) ;
for ( specId = 0 ; specId < nSpecs ; specId++ ) {
  for ( node = 0 ; node < nItems ; node++ ) {
     spec = auxData[specId*nItems+node] ;
     ...
  }
}
```

The quantities UDF_NBC_TURBULENCE_Y and UDF_NBC_TURBULENCE_YPLUS are based on the distance to the nearest turbulence wall. These are defined by `TURBULENCE_WALL` and `SIMPLE_BOUNDARY_CONDITION`type=wall commands. UDF_NBC_TURBULENCE_YPLUS also uses the shear at these walls.

Errors

- This routine expects a valid *`udfHd`*.
- This routine may only be called within an Nodal Boundary Condition user function.
- *`dataName`* must be one of the values given above.
- The problem must contain the equation associated with the requested data.
- The corresponding command must have a valid *`auxiliary_nodes`* parameter.

udfGetNbcRafData()

Return nodal running average field solution data for the nodal boundary condition.

Syntax

```
data = udfGetNbcRafData( udfHd, dataName ) ;
```

Type

AcuSolve User-Defined Nodal Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName *(integer)*
: Symbolic name of the requested data.

UDF_NBC_VELOCITY	Velocity.
UDF_NBC_ACCELERATION	Acceleration.
UDF_NBC_PRESSURE	Pressure.
UDF_NBC_TEMPERATURE	Temperature.
UDF_NBC_SPECIES	Species.
UDF_NBC_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_NBC_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_NBC_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_NBC_EDDY_TIME	Turbulence eddy time.
UDF_NBC_TRANSITION_RETHETA	Transition Re-Theta.
UDF_NBC_INTERMITTENCY	Transition intermittency.
UDF_NBC_MESH_DISPLACEMENT	Mesh displacement.
UDF_NBC_MESH_VELOCITY	Mesh velocity.
UDF_NBC_TURBULENCE_Y	Distance to nearest turbulence wall.
UDF_NBC_TURBULENCE_YPLUS	Turbulence y+ based on distance to nearest turbulence wall and shear at that wall.
UDF_NBC_VISCOELASTIC	Viscoelastic.

Return Value

data *(Real*)*

Pointer to one or two dimensional real array of the requested data. The dimensions of the array depend on *dataName* as follows. If the second dimension is one, then the array may be treated as one dimensional.

dataName	First Dimension	Second Dimension
UDF_NBC_VELOCITY	nItems	3
UDF_NBC_ACCLERATION	nItems	3
UDF_NBC_PRESSURE	nItems	1
UDF_NBC_TEMPERATURE	nItems	1
UDF_NBC_SPECIES	nItems	`udfGetNumSpecs()`
UDF_NBC_EDDY_VISCOSITY	nItems	1
UDF_NBC_KINETIC_ENERGY	nItems	1
UDF_NBC_EDDY_FREQUENCY	nItems	1
UDF_NBC_EDDY_TIME	nItems	1
UDF_NBC_TRANSITION_RETH	nItems	1
UDF_NBC_INTERMITTENCY	nItems	1
UDF_NBC_MESH_DISPLACEM	nItems	3
UDF_NBC_MESH_VELOCITY	nItems	3
UDF_NBC_TURBULENCE_Y	nItems	1
UDF_NBC_TURBULENCE_YPLU	nItems	1
UDF_NBC_VISCOELASTIC	nItems	6

Description

This routine returns the requested nodal solution data. It is used exactly the same way as `udfGetNbcData`, but returns the running average field versions of the data.

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an Nodal Boundary Condition user function.
- *dataName* must be one of the values given above.

- The problem must contain the equation associated with the requested data.
- *running_average* must be turned on in the EQUATION command.

udfCheckNbcNumUsrVals()

Check the number of user values given in the input file.

Syntax

udfCheckNbcNumUsrVals(udfHd, nUsrVals) ;

Type

AcuSolve User-Defined Nodal Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

nUsrVals *(integer)*
: Number of user values that the function expects.

Return Value

None.

Description

This routine verifies that the number of user values supplied in the input file is the same as the second argument. If it is not, an error message is issued and the solver exits. For example,

```
udfCheckNbcNumUsrVals( udfHd, 2 ) ;
```

Errors

This routine expects a valid *udfHd*.

udfGetNbcNumUsrVals()

Return the number of user values supplied in the input file.

Syntax

```
nUsrVals = udfGetNbcNumUsrVals( udfHd ) ;
```

Type

AcuSolve User-Defined Nodal Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nUsrVals *(integer)*
: Number of user values given in the input file.

Description

This routine returns the number of user values supplied in the input file. It is most useful when the number of parameters is not know a priori, such as for an interpolated curve fit. For example,

```
Integer nUsrVals ;
Integer i ;
Real x, y ;
Real* usrVals ;
...
nUsrVals = udfGetNbcNumUsrVals( udfHd ) ;
usrVals = udfGetNbcUsrVals( udfHd ) ;
for ( i = 0 ; i < nUsrVals ; i+=2 ) {
   x = usrVals[i] ;
   y = usrVals[i+1] ;
   ...
}
```

Errors

This routine expects a valid *udfHd*.

udfGetNbcUsrVals()

Return the array of user supplied values.

Syntax

```
nUsrVals = udfGetNbcUsrVals( udfHd ) ;
```

Type

AcuSolve User-Defined Nodal Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

usrVals *(Real*)*
: Pointer to one-dimensional real array of user values as given in the input file. This array has `udfGetNbcNumUsrVals()` entries.

Description

This routine returns the real array of user-given parameters. The order of the parameters within the array is the same as in the input file. For example,

```
Real* usrVals ;
Real amp, omega ;
...
udfCheckNbcNumUsrVals( udfHd, 2 ) ;
usrVals = udfGetNbcUsrVals( udfHd ) ;
amp = usrVals[0] ;
omega = usrVals[1] ;
```

Errors

This routine expects a valid *udfHd*.

Nodal Initial Condition Routines

12

These support routines are accessible only by the Nodal Initial Condition user functions.

This chapter covers the following:

udfGetNicIds()

Return the nodal initial condition user node numbers.

Syntax

```
nodeIds = udfGetNicIds( udfHd ) ;
```

Type

AcuSolve User-Defined Name Initial Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nodeIds *(Integer*)*
: Pointer to one dimensional integer array of user node numbers. This array has a dimension of number of nodes, nItems.

Description

This routine returns the array of user node numbers. For example,

```
Integer* nodeIds ;
Integer usrNode, node ;
...
nodeIds = udfGetNicIds( udfHd ) ;
for ( node = 0 ; node < nItems ; node++ ) {
 usrNode = nodeIds[node] ;
 ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Nodal Initial Condition user function.

udfGetNicCrd()

Return the nodal coordinates for the nodal initial condition.

Syntax

```
crd = udfGetNicCrd( udfHd ) ;
```

Type

AcuSolve User-Defined Name Initial Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

crd *(Real*)*
: Pointer to two dimensional real array of nodal coordinates. The first (fastest) dimension of the array is the number of nodes, nItems, and the second (slowest) dimension is three for the x, y and z coordinates.

Description

This routine returns the nodal coordinates. If mesh displacement is active, the returned coordinates are for the current (deformed) configuration. For example,

```
Real* crd ;
Real x, y, z ;
Integer node ;
...
crd = udfGetNicCrd( udfHd ) ;
for ( node = 0 ; node < nItems ; node++ ) {
   x = crd[0*nItems+node] ;
   y = crd[1*nItems+node] ;
   z = crd[2*nItems+node] ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Nodal Initial Condition user function.

udfGetNicRefCrd()

Return the initial condition nodal coordinates for the nodal initial condition.

Syntax

```
crd = udfGetNicRefCrd( udfHd ) ;
```

Type

AcuSolve User-Defined Name Initial Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

crd *(Real*)*
: Pointer to two dimensional real array of nodal coordinates. The first (fastest) dimension of the array is the number of nodes, nItems, and the second (slowest) dimension is three for the x, y and z coordinates.

Description

This routine returns the initial condition nodal coordinates. If mesh displacement is active, the returned coordinates include the initial conditions for the mesh displacement; that is, the initial deformed configuration. For example,

```
Real* crd ;
Real x, y, z ;
Integer node ;
...
crd = udfGetNicRefCrd( udfHd ) ;
for ( node = 0 ; node < nItems ; node++ ) {
 x = crd[0*nItems+node] ;
 y = crd[1*nItems+node] ;
 z = crd[2*nItems+node] ;
 ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Nodal Initial Condition user function.

udfGetNicData()

Return nodal solution data for the nodal initial condition.

Syntax

```
data = udfGetNicData( udfHd, dataName ) ;
```

Type

AcuSolve User-Defined Name Initial Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName (integer)
: Symbolic name of the requested data.

UDF_NIC_VELOCITY	Velocity.
UDF_NIC_ACCELERATION	Acceleration.
UDF_NIC_PRESSURE	Pressure.
UDF_NIC_TEMPERATURE	Temperature.
UDF_NIC_SPECIES	Species.
UDF_NIC_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_NIC_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_NIC_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_NIC_MESH_DISPLACEMENT	Mesh displacement.
UDF_NIC_MESH_VELOCITY	Mesh velocity.
UDF_NIC_VISCOELASTIC	Viscoelastic stress.

Return Value

data *(Real*)*
: Pointer to one or two dimensional real array of the requested data. The dimensions of the array depend on *dataName* as follows. If the second dimension is one, then the array may be treated as one dimensional.

dataName	First Dimension	Second Dimension
UDF_NIC_VELOCITY	nItems	3

dataName	First Dimension	Second Dimension
UDF_NIC_ACCELERATION	nItems	3
UDF_NIC_PRESSURE	nItems	1
UDF_NIC_TEMPERATURE	nItems	1
UDF_NIC_SPECIES	nItems	`udfGetNumSpecs()`
UDF_NIC_EDDY_VISCOSITY	nItems	1
UDF_NIC_MESH_DISPLACEMENT	nItems	3
UDF_NIC_MESH_VELOCITY	nItems	3
UDF_NIC_KINETIC_ENERGY	nItems	1
UDF_NIC_EDDY_FREQUENCY	nItems	1
UDF_NBC_TURBULENCE_Y	nItems	1
UDF_NBC_TURBULENCE_YPLUS	nItems	1
UDF_NIC_VISCOELASTIC	nItems	6

Description

This routine returns the requested nodal solution data. For example,

```
Real* data ;
Real u, v, w ;
Real spec ;
Integer node, nSpecs, specId ;
...
data = udfGetNicData( udfHd, UDF_NIC_VELOCITY ) ;
for ( node = 0 ; node < nItems ; node++ ) {
   u = data[0*nItems+node] ;
   v = data[1*nItems+node] ;
   w = data[2*nItems+node] ;
  ...
}
...
data = udfGetNicData( udfHd, UDF_NIC_SPECIES ) ;
nSpecs = udfGetNumSpecs( udfHd ) ;
for ( specId = 0 ; specId < nSpecs ; specId++ ) {
   for ( node = 0 ; node < nItems ; node++ ) {
      spec = data[specId*nItems+node] ;
      ...
   }
}
```

Errors

- This routine expects a valid *udfHd*.

- This routine may only be called within a Nodal Initial Condition user function.
- `dataName` must be one of the values given above.
- The problem must contain the equation associated with the requested data.

Periodic Boundary Condition Routines

13

These support routines are accessible only by the Periodic Boundary Condition user functions.

This chapter covers the following:

udfGetPbcIds()

Return the periodic boundary condition user node-pair numbers.

Syntax

```
pairIds = udfGetPbcIds( udfHd ) ;
```

Type

AcuSolve User-Defined Problem Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

pairIds *(Integer*)*
: Pointer to one dimensional integer array of user node-pair numbers. This array has a dimension of number of node-pairs, nItems.

Description

This routine returns the array of user node-pair numbers. For example,

```
Integer* pairIds ;
Integer usrPair, pair ;
...
PairIds = udfGetPbcIds( udfHd ) ;
for ( pair = 0 ; pair < nItems ; pair++ ) {
   usrPair = pairIds[pair] ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Periodic Boundary Condition user function.

udfGetPbcCrd()

Return the nodal coordinates for the periodic boundary condition.

Syntax

```
crd = udfGetPbcCrd( udfHd ) ;
```

Type

AcuSolve User-Defined Problem Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

crd *(Real*)*
: Pointer to three dimensional real array of nodal coordinates. The first (fastest) dimension of the array is the number of node-pairs, nItems, the second dimension is three for the x, y and z coordinates, and the third (slowest) is two for each of the two nodes in a node-pair.

Description

This routine returns the nodal coordinates. If mesh displacement is active, the returned coordinates are for the current (deformed) configuration. For example,

```
Real* crd ;
Real x_1, y_1, z_1, x_2, y_2, z_2 ;
Integer pair ;
...
crd = udfGetPbcCrd( udfHd ) ;
for ( pair = 0 ; pair < nItems ; pair++ ) {
   /* first node in pair */
   x_1 = crd[0*nItems+pair] ;
   y_1 = crd[1*nItems+pair] ;
   z_1 = crd[2*nItems+pair] ;
   /* second node in pair */
   x_2 = crd[3*nItems+pair] ;
   y_2 = crd[4*nItems+pair] ;
   z_2 = crd[5*nItems+pair] ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within a Periodic Boundary Condition user function.

udfGetPbcData()

Return solution data for pairs of nodes for the periodic boundary condition.

Syntax

```
data = udfGetPbcData( udfHd, dataName ) ;
```

Type

AcuSolve User-Defined Problem Boundary Condition

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName *(integer)*
: Symbolic name of the requested data.

UDF_PBC_VELOCITY	Velocity.
UDF_PBC_ACCELERATION	Acceleration.
UDF_PBC_PRESSURE	Pressure.
UDF_PBC_TEMPERATURE	Temperature.
UDF_PBC_SPECIES	Species.
UDF_PBC_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_PBC_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_PBC_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_PBC_MESH_DISPLACEMENT	Mesh displacement.
UDF_PBC_MESH_VELOCITY	Mesh velocity.

Return Value

data *(Real*)*
: Pointer to two or three dimensional real array of the requested data. The dimensions of the array depend on *dataName* as follows. nItems is the number of node-pairs. If the third dimension is one, then the array may be treated as two dimensional. The last non-trivial dimension is two for each of the two nodes in a node-pair.

dataName	**First Dimension**	**Second Dimension**	**Third Dimension**
UDF_PBC_VELOCITY	nItems	3	2

dataName	First Dimension	Second Dimension	Third Dimension
UDF_PBC_ACCLERAT	nItems	3	2
UDF_PBC_PRESSURE	nItems	2	1
UDF_PBC_TEMPERAT	nItems	2	1
UDF_PBC_SPECIES	nItems	`udfGetNumSpecs()`	2
UDF_PBC_EDDY_VIS	nItems	2	1
UDF_PBC_KINETIC_I	nItems	2	1
UDF_PBC_EDDY_FRE	nItems	2	1
UDF_PBC_MESH_DIS	nItems	3	2
UDF_PBC_MESH_VEI	nItems	3	2

Description

This routine returns the requested solution data for the node-pairs. For example,

```
Real* data ;
Real u_1, v_1, w_1, u_2, v_2, w_2 ;
Real spec_1, spec_2 ;
Integer pair, nSpecs, specId ;
...
data = udfGetPbcData( udfHd, UDF_PBC_VELOCITY ) ;
for ( pair = 0 ; pair < nItems ; pair++ ) {
   /* velocity at first node */
   u_1 = data[0*nItems+pair] ;
   v_1 = data[1*nItems+pair] ;
   w_1 = data[2*nItems+pair] ;
   /* velocity at second node */
   u_2 = data[3*nItems+pair] ;
   v_2 = data[4*nItems+pair] ;
   w_2 = data[5*nItems+pair] ;
   ...
}
...
data = udfGetPbcData( udfHd, UDF_PBC_SPECIES ) ;
nSpecs = udfGetNumSpecs( udfHd ) ;
for ( specId = 0 ; specId < nSpecs ; specId++ ) {
   for ( pair = 0 ; pair < nItems ; pair++ ) {
     /* species at first node */
     spec_1 = data[specId*nItems+pair] ;
     /* species at second node */
     spec_2 = data[(nSpecs+specId)*nItems+pair] ;
     ...
   }
}
```

Errors

- This routine expects a valid `udfHd`.
- This routine may only be called within a Periodic Boundary Condition user function.
- `dataName` must be one of the values given above.
- The problem must contain the equation associated with the requested data.

Subdomain Routines

14

These support routines are accessible only within external output codes.

This chapter covers the following:

udfGetNumSdNodes()

Return the number of nodes in the subdomain.

Syntax

```
nNodes = udfGetNumSdNodes( udfHd ) ;
```

Type

AcuSolve User-Defined Subdomain Routine

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nNodes *(integer)*
: Number of nodes in the subdomain.

Description

This routine returns the number of nodes in the current subdomain. For example,

```
Integer* nNodes ;
...
nNodes = udfGetNumSdNodes( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an external code.

udfGetSdUsrIds()

Return the nodal user IDs for the nodes in the subdomain.

Syntax

```
usrIds = udfGetSdUsrIds( udfHd ) ;
```

Type

AcuSolve User-Defined Subdomain Routine

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

usrIds *(Integer*)*
: Pointer to one dimensional integer array of user IDs.

Description

This routine returns the array of user IDs for the nodes in the current subdomain. For example,

```
Integer* usrIds ;
Integer nNodes, node, usrNode ;
...
nNodes = udfGetNSdNodes( udfHd ) ;
usrIds = udfGetSdUsrIds( udfHd ) ;
for ( node = 0 ; node < nNodes ; node++ ) {
   usrNode = usrIds[node] ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an external code.

udfGetSdCrd()

Return the nodal coordinates for the nodes in the subdomain.

Syntax

```
crd = udfGetSdCrd( udfHd ) ;
```

Type

AcuSolve User-Defined Subdomain Routine

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

crd *(Real*)*
: Pointer to two dimensional real array of nodal coordinates. The first (fast) dimension of the array is the number of nodes, and the second (slow) dimension is three, for the x, y and z coordinates.

Description

This routine returns the nodal coordinates for the subdomain. If mesh displacement is active, the returned coordinates are for the current (deformed) configuration. For example,

```
Real* crd ;
Real x, y, z ;
Integer node, nNodes ;
...
nNodes = udfGetNSdNodes( udfHd ) ;
crd = udfGetSdCrd( udfHd ) ;
for ( node = 0 ; node < nNodes ; node++ ) {
   x = crd[0*nNodes+node] ;
   y = crd[1*nNodes+node] ;
   z = crd[2*nNodes+node] ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an external code.

udfGetSdRefCrd()

Return the initial condition nodal coordinates for the nodes in the subdomain.

Syntax

```
crd = udfGetSdRefCrd (udfHd) ;
```

Type

AcuSolve User-Defined Subdomain Routine

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

crd *(Real*)*
: Pointer to two dimensional real array of initial condition nodal coordinates. The first (fast) dimension of the array is the number of nodes, and the second (slow) dimension is three, for the x, y and z coordinates.

Description

This routine returns the initial condition nodal coordinates for the subdomain. If mesh displacement is active, the returned coordinates include the initial conditions for the mesh displacement; that is, the initial deformed configuration. For example,

```
Real* crd ;
Real x, y, z ;
Integer node, nNodes ;
...
nNodes = udfGetNSdNodes( udfHd ) ;
crd = udfGetSdRefCrd( udfHd ) ;
for ( node = 0 ; node < nNodes ; node++ ) {
   x = crd[0*nNodes+node] ;
   y = crd[1*nNodes+node] ;
   z = crd[2*nNodes+node] ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an external code.

udfGetNumSdDataNames()

Return the number of data names in the subdomain.

Syntax

```
nNames = udfGetNumSdDataNames( udfHd ) ;
```

Type

AcuSolve User-Defined Subdomain Routine

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nNames *(integer)*
: Number of data names in the subdomain.

Description

This routine returns the number of data names in the current subdomain. For example,

```
Integer nNames ;
...
nNames = udfGetNumSdDataNames( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an external code.

udfGetSdDataName()

Return a data variable name in the subdomain.

Syntax

```
dataName = udfGetSdDataName( udfHd, dataId ) ;
```

Type

AcuSolve User-Defined Subdomain Routine

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataId *(integer)*
: Index from 0 to nNames -1.

Return Value

dataName *(integer)*
: Symbolic name corresponding to the data index.

UDF_SD_VELOCITY	Velocity.
UDF_SD_ACCELERATION	Acceleration.
UDF_SD_PRESSURE	Pressure.
UDF_SD_TEMPERATURE	Temperature.
UDF_SD_SPECIES	Species.
UDF_SD_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_SD_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_SD_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_SD_MESH_DISPLACEMENT	Mesh displacement.
UDF_SD_MESH_VELOCITY	Mesh velocity.
UDF_SD_TURBULENCE_Y	Distance to nearest turbulence wall.
UDF_SD_TURBULENCE_YPLUS	Turbulence y+ based on distance to nearest turbulence wall and shear at that wall.

Description

This routine returns a symbolic data name in the current subdomain. For example,

```
Integer nNames, dataName, dataId ;
...
nNames = udfGetNSdDataNames( udfHd ) ;
for ( dataId = 0 ; dataId < nNames ; dataId++ ) {
 dataName = udfGetSdDataName( udfHd, dataId ) ;
 if ( dataName == UDF_SD_VELOCITY ) {
  ...
 }
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an external code.
- *dataId* must be from 0 to nNames-1.

udfGetSdDataDim()

Return the dimension of a data variable.

Syntax

```
dataDim = udfGetSdDataDim( udfHd, dataName ) ;
```

Type

AcuSolve User-Defined Subdomain Routine

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName *(integer)*
: Symbolic name corresponding to the data index.

UDF_SD_VELOCITY	Velocity.
UDF_SD_ACCELERATION	Acceleration.
UDF_SD_PRESSURE	Pressure.
UDF_SD_TEMPERATURE	Temperature.
UDF_SD_SPECIES	Species.
UDF_SD_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_SD_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_SD_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_SD_MESH_DISPLACEMENT	Mesh displacement.
UDF_SD_MESH_VELOCITY	Mesh velocity.
UDF_SD_TURBULENCE_Y	Distance to nearest turbulence wall.
UDF_SD_TURBULENCE_YPLUS	Turbulence y+ based on distance to nearest turbulence wall and shear at that wall.

Return Value

dataDim *(integer)*
: Dimension of the data variable.

Description

This routine returns the dimension of a variable associated with the given symbolic data name. For example,

```
Integer dataDim ;
...
dataDim = udfGetSdDataDim( udfHd, UDF_SD_VELOCITY ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an external code.
- *dataName* must be one of the values given above.
- The subdomain must contain the equation associated with the data variable.

udfGetSdDataType()

Return the type of a data variable.

Syntax

```
datatype = udfGetSdDataType( udfHd, dataName ) ;
```

Type

AcuSolve User-Defined Subdomain Routine

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

dataName *(integer)*
: Symbolic name of a data variable.

UDF_SD_VELOCITY	Velocity.
UDF_SD_ACCELERATION	Acceleration.
UDF_SD_PRESSURE	Pressure.
UDF_SD_TEMPERATURE	Temperature.
UDF_SD_SPECIES	Species.
UDF_SD_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_SD_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_SD_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_SD_MESH_DISPLACEMENT	Mesh displacement.
UDF_SD_MESH_VELOCITY	Mesh velocity.
UDF_SD_TURBULENCE_Y	Distance to nearest turbulence wall.
UDF_SD_TURBULENCE_YPLUS	Turbulence y+ based on distance to nearest turbulence wall and shear at that wall.

Return Value

dataType *(integer)*
: Symbolic type corresponding to the data name.

UDF_SD_TYPE_SCALAR	Scalar.
UDF_SD_TYPE_VECTOR	Vector.

Description

This routine returns the type (scalar or vector) of a variable associated with the given symbolic data name. For example,

```
Integer dataType ;
...
dataType = udfGetSdDataType( udfHd, UDF_SD_VELOCITY )
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an external code.
- *dataName* must be one of the values given above.
- The subdomain must contain the equation associated with the data variable.

udfGetSdData()

Return the data for a variable.

Syntax

```
data = udfGetSdData( udfHd, dataName ) ;
```

Type

AcuSolve User-Defined Subdomain Routine

Parameters

udfHd
The opaque handle (pointer) which was passed to the user function.

dataName *(integer)*
Symbolic name of a data variable.

UDF_SD_VELOCITY	Velocity.
UDF_SD_ACCELERATION	Acceleration.
UDF_SD_PRESSURE	Pressure.
UDF_SD_TEMPERATURE	Temperature.
UDF_SD_SPECIES	Species.
UDF_SD_EDDY_VISCOSITY	Turbulence eddy viscosity.
UDF_SD_KINETIC_ENERGY	Turbulence kinetic energy.
UDF_SD_EDDY_FREQUENCY	Turbulence eddy frequency.
UDF_SD_MESH_DISPLACEMENT	Mesh displacement.
UDF_SD_MESH_VELOCITY	Mesh velocity.
UDF_SD_TURBULENCE_Y	Distance to nearest turbulence wall.
UDF_SD_TURBULENCE_YPLUS	Turbulence y+ based on distance to nearest turbulence wall and shear at that wall.

Return Value

data *(Real*)*
Pointer to two dimensional real array of the requested data variable. The first (fast) dimension of the array is the number of nodes, and the second (slow) dimension is the dimension of the variable.

Description

This routine returns the data for a variable associated with the given symbolic data name. For example,

```
Integer nNodes, dataDim, node ;
Real u, v, w ;
Real* data ;
...
nNodes = udfGetNSdNodes( udfHd ) ;
dataDim = udfGetSdDataDim( udfHd, UDF_SD_VELOCITY ) ;
data = udfGetSdData( udfHd, UDF_SD_VELOCITY ) ;
for ( node = 0 ; node < nNodes ; node++ ) {
   u = data[0*nNodes+node] ;
   v = data[1*nNodes+node] ;
   w = data[2*nNodes+node] ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an external code.
- *dataName* must be one of the values given above.
- The subdomain must contain the equation associated with the data variable.

udfGetSdNElms()

Return the number of element sets in the subdomain.

Syntax

```
nElms = udfGetSdNElms( udfHd ) ;
```

Type

AcuSolve User-Defined Subdomain Routine

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nElms *(integer)*
: Number of element sets in the subdomain.

Description

This routine returns the number of element sets in the current subdomain. For example,

```
Integer nElms ;
...
nElms = udfGetSdNElms( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an external code.

udfSetSdElmId()

Set the ID of an interior element set.

Syntax

udfSetSdElmId(udfHd, elmId) ;

Type

AcuSolve User-Defined Subdomain Routine

Parameters

udfHd
The opaque handle (pointer) which was passed to the user function.

elmId (integer)
Element set index.

Return Value

None.

Description

This routine sets the index of an interior element set. For example,

```
Integer nElms, elmId ;
...
nElms = udfGetNSdElms( udfHd ) ;
for ( elmId = 0 ; elmId < nElms ; elmId++ ) {
   udfSetSdElmId( udfHd, elmId ) ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an external code.
- *elmId* must be from 0 to nElms - 1.

udfGetSdNEbcs()

Return the number of boundary element sets in the subdomain.

Syntax

```
nEbcs = udfGetSdNEbcs( udfHd ) ;
```

Type

AcuSolve User-Defined Subdomain Routine

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

Return Value

nEbcs *(integer)*
: Number of boundary element sets in the subdomain.

Description

This routine returns the number of boundary element sets in the current subdomain. For example,

```
Integer nEbcs ;
...
nEbcs = udfGetSdNEbcs( udfHd ) ;
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an external code.

udfSetSdEbcId()

Set the ID of a boundary element set.

Syntax

udfSetSdEbcId(udfHd, ebcId) ;

Type

AcuSolve User-Defined Subdomain Routine

Parameters

udfHd
: The opaque handle (pointer) which was passed to the user function.

ebcId *(integer)*
: Boundary element set index.

Return Value

None.

Description

This routine sets the index of a boundary element set. For example,

```
Integer nEbcs, ebcId ;
...
nEbcs = udfGetNSdEbcs( udfHd ) ;
for ( ebcId = 0 ; ebcId < nEbcs ; ebcId++ ) {
   udfSetSdEbcId( udfHd, ebcId ) ;
   ...
}
```

Errors

- This routine expects a valid *udfHd*.
- This routine may only be called within an external code.
- *ebcId* must be from 0 to nEbcs - 1.

Intellectual Property Rights Notice

This Intellectual Property Rights Notice is exemplary, and therefore not exhaustive, of the intellectual property rights held by Altair Engineering Inc. or its affiliates. Software, other products, and materials of Altair Engineering Inc. or its affiliates are protected under laws of the United States and laws of other jurisdictions.

In addition to intellectual property rights indicated herein, such software, other products, and materials of Altair Engineering Inc. or its affiliates may be further protected by patents, additional copyrights, additional trademarks, trade secrets, and additional other intellectual property rights. For avoidance of doubt, copyright notice does not imply publication. Copyrights in the below are held by Altair Engineering Inc. or its affiliates. Additionally, all non-Altair marks are the property of their respective owners. If you have any questions regarding trademarks or registrations, please contact marketing and legal.

This Intellectual Property Rights Notice does not give you any right to any product, such as software, or underlying intellectual property rights of Altair Engineering Inc. or its affiliates. Usage, for example, of software of Altair Engineering Inc. or its affiliates is governed by and dependent on a valid license agreement.

Altair® HyperWorks®, a Design & Simulation Platform

Altair® AcuSolve® ©1997-2024

Altair® Activate® ©1989-2024

Altair® Automated Reporting Director™ ©2008-2022

Altair® Battery Damage Identifier™ ©2019-2024

Altair® CFD™ ©1990-2024

Altair Compose® ©2007-2024

Altair® ConnectMe™ ©2014-2024

Altair® DesignAI™ ©2022-2024

Altair® DSim™ ©2024

Altair® EDEM™ ©2005-2024

Altair® EEvision™ ©2018-2024

Altair® ElectroFlo™ ©1992-2024

Altair Embed® ©1989-2024

Altair Embed® SE ©1989-2024

Altair Embed®/Digital Power Designer ©2012-2024

Altair Embed®/eDrives ©2012-2024

Altair Embed® Viewer©1996-2024

Altair® e-Motor Director™©2019-2024

Altair® ESAComp® ©1992-2024

Altair® expertAI™ ©2020-2024

Altair® Feko® ©1999-2024

Altair® FlightStream® ©2017-2024

Altair® Flow Simulator™ ©2016-2024

Altair® Flux® ©1983-2024

Altair® FluxMotor® ©2017-2024

Altair® GateVision PRO™ ©2002-2024

Altair® Geomechanics Director™ ©2011-2022

Altair® HyperCrash® ©2001-2023

Altair® HyperGraph® ©1995-2024

Altair® HyperLife® ©1990-2024

Altair® HyperMesh® ©1990-2024

Altair® HyperMesh® CFD ©1990-2024

Altair® HyperMesh ® NVH ©1990-2024

Altair® HyperSpice™ ©2017-2024

Altair® HyperStudy® ©1999-2024

Altair® HyperView® ©1999-2024

Altair® HyperView Player® ©2022-2024

Altair® HyperWorks® ©1990-2024

Altair® HyperWorks® Design Explorer ©1990-2024

Altair® HyperXtrude® ©1999-2024

Altair® Impact Simulation Director™ ©2010-2022

Altair® Inspire™ ©2009-2024

Altair® Inspire™ Cast ©2011-2024

Altair® Inspire™ Extrude Metal ©1996-2024

Altair® Inspire™ Extrude Polymer ©1996-2024

Altair® Inspire™ Form ©1998-2024

Altair® Inspire™ Mold ©2009-2024

Altair® Inspire™ PolyFoam ©2009-2024

Altair® Inspire™ Print3D ©2021-2024

Altair® Inspire™ Render ©1993-2024

Altair® Inspire™ Studio ©1993-2024

Altair® Material Data Center™ ©2019-2024

Altair® Material Modeler™ ©2019-2024

Altair® Model Mesher Director™ ©2010-2024

Altair® MotionSolve® ©2002-2024

Altair® MotionView® ©1993-2024

Altair® Multi-Disciplinary Optimization Director™ ©2012-2024

Altair® Multiscale Designer® ©2011-2024

Altair® newFASANT™ ©2010-2020

Altair® nanoFluidX® ©2013-2024

Altair® NLVIEW® ©2018-2024

Altair® NVH Director™ ©2010-2024

Altair® NVH Full Vehicle™ ©2022-2024

Altair® NVH Standard™ ©2022-2024

Altair® OmniV™ ©2015-2024

Altair® OptiStruct® ©1996-2024

Altair® physicsAI™ ©2021-2024

Altair® PollEx™ ©2003-2024

Altair® PollEx™ for ECAD ©2003-2024

Altair® PSIM™ ©1994-2024

Altair® Pulse™ ©2020-2024

Altair® Radioss® ©1986-2024

Altair® romAI™ ©2022-2024

Altair® RTLvision PRO™ ©2002-2024

Altair® S-CALC™ ©1995-2024

Altair® S-CONCRETE™ ©1995-2024

Altair® S-FRAME® ©1995-2024

Altair® S-FOUNDATION™ ©1995-2024

Altair® S-LINE™ ©1995-2024

Altair® S-PAD™ ©1995-2024

Altair® S-STEEL™ ©1995-2024

Altair® S-TIMBER™ ©1995-2024

Altair® S-VIEW™ ©1995-2024

Altair® SEAM® ©1985-2024

Altair® shapeAI™ ©2021-2024

Altair® signalAI™ ©2020-2024

Altair® Silicon Debug Tools™ ©2018-2024

Altair® SimLab® ©2004-2024

Altair® SimLab® ST ©2019-2024

Altair® SimSolid® ©2015-2024

Altair® SpiceVision PRO™ ©2002-2024

Altair® Squeak and Rattle Director™ ©2012-2024

Altair® StarVision PRO™ ©2002-2024

Altair® Structural Office™ ©2022-2024

Altair® Sulis™ ©2018-2024

Altair® Twin Activate® ©1989-2024

Altair® UDE™ ©2015-2024

Altair® ultraFluidX® ©2010-2024

Altair® Virtual Gauge Director™ ©2012-2024

Altair® Virtual Wind Tunnel™ ©2012-2024

Altair® Weight Analytics™ ©2013-2022

Altair® Weld Certification Director™ ©2014-2024

Altair® WinProp™ ©2000-2024

Altair® WRAP™ ©1998-2024

Altair® HPCWorks®, a HPC & Cloud Platform

Altair® Allocator™ ©1995-2024

Altair® Access™ ©2008-2024

Altair® Accelerator™ ©1995-2024

Altair® Accelerator™ Plus ©1995-2024

Altair® Breeze™ ©2022-2024

Altair® Cassini™ ©2015-2024

Altair® Control™ ©2008-2024

Altair® Desktop Software Usage Analytics™ (DSUA) ©2022-2024

Altair® FlowTracer™ ©1995-2024

Altair® Grid Engine® ©2001, 2011-2024

Altair® InsightPro™ ©2023-2024

Altair® Hero™ ©1995-2024

Altair® Liquid Scheduling™ ©2023-2024

Altair® Mistral™ ©2022-2024

Altair® Monitor™ ©1995-2024

Altair® NavOps® ©2022-2024

Altair® PBS Professional® ©1994-2024

Altair® PBS Works™ ©2022-2024

Altair® Simulation Cloud Suite (SCS) ©2024

Altair® Software Asset Optimization (SAO) ©2007-2024

Altair® Unlimited™ ©2022-2024

Altair® Unlimited Data Analytics Appliance™ ©2022-2024

Altair® Unlimited Virtual Appliance™ ©2022-2024

Altair® RapidMiner®, a Data Analytics & AI Platform

Altair® AI Hub ©2023-2024

Altair® AI Edge™ ©2023-2024

Altair® AI Cloud ©2022-2024

Altair® AI Studio ©2023-2024

Altair® Analytics Workbench™ ©2002-2024

Altair® Graph Lakehouse™ ©2013-2024

Altair® Graph Studio™ ©2007-2024

Altair® Knowledge Hub™ ©2017-2024

Altair® Knowledge Studio® ©1994-2024

Altair® Knowledge Studio® for Apache Spark ©1994-2024

Altair® Knowledge Seeker™ ©1994-2024

Altair® IoT Studio™ ©2002-2024

Altair® Monarch® ©1996-2024

Altair® Monarch® Classic ©1996-2024

Altair® Monarch® Complete™ ©1996-2024

Altair® Monarch® Data Prep Studio ©2015-2024

Altair® Monarch Server™ ©1996-2024

Altair® Panopticon™ ©2004-2024

Altair® Panopticon™ BI ©2011-2024

Altair® SLC™ ©2002-2024

Altair® SLC Hub™ ©2002-2024

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