CFD EXPERTS

Simulate the Future

Ansys Fluent Customization Manual

Table of Contents

List of Figures

List of Tables

Using This Manual

This preface is divided into the following sections:

1. The Contents of This Manual

User-defined functions (UDFs) allow you to customize Ansys Fluent and can significantly enhance its capabilities. Part 1 of the Ansys Fluent Customization Manual presents detailed information on how to write, compile, and use UDFs in Ansys Fluent. Examples have also been included, where available. General information about C programming basics is included in an appendix.

Learning to customize the Ansys Fluent user interface to build your own graphical user interface (GUI) will allow you to easily change critical data being used by your UDF at will. Part 2 of the Ansys Fluent Customization Manual will walk you through the process of creating a customized addition to the Ansys Fluent user interface through the use of Scheme macros. General information about Scheme programming basics is included in chapter 1 of part 2.

> **Important:**
>
> Under U.S. and international copyright law, ANSYS, Inc. is unable to distribute copies of the papers listed in the bibliography, other than those published internally by ANSYS, Inc. Use your library or a document delivery service to obtain copies of copyrighted papers.

Information in part 1 of this manual is presented in the following chapters:

- Overview of User-Defined Functions (UDFs) (p. 3), presents an introduction to User Defined Functions (UDFs).

- DEFINE Macros (p. 19), describes predefined DEFINE macros that you will use to define your UDF.

- Additional Macros for Writing UDFs (p. 305), describes additional predefined macros that you will use to define your UDF.

- Interpreting UDFs (p. 393), describes how to interpret the source file for your UDFs.

- Compiling UDFs (p. 399), describes how to compile the UDF source file, build a shared library from the resulting objects, and load the library into Ansys Fluent.

- Hooking UDFs to Ansys Fluent (p. 423), describes how to add, or hook, your UDF into the Ansys Fluent interface.

- Parallel Considerations (p. 577), describes how to use UDFs in a parallel computing environment.

- Examples (p. 611), presents examples of UDFs.

- Appendix A: C Programming Basics (p. 677), presents an introduction to the C programming language.

- , presents a series of `DEFINE` macro definitions for multiple categories.

- , presents a series of reference tables for multiphase-related `DEFINE` macros.

Information in part 2 of this manual is presented in the following chapters:

- , presents an introduction to Fluent GUI concepts such as limitations, the Scheme language, and RP variables.

- , describes the process of creating a dialog box to add interface elements to, and using tables to organize these elements.

- , presents each of the interface elements that can be added to a dialog box and explains the macros needed to use them.

- , describes the process of adding menus, submenus, and menu items to the Fluent ribbon that are used to open your dialog box.

- , presents a number of examples that encompass all of the concepts that are covered in the rest of part 2 of this manual.

- , describes mistakes that can be easily made throughout the GUI creation process and how to avoid them.

- , provides a reference table for all of the Ansys Fluent Scheme macros used throughout part 2 of this manual.

This document provides some basic information about the C programming language (Part 1 Appendix A) as it relates to user-defined functions in Ansys Fluent, and assumes that you are an experienced programmer in C. If you are unfamiliar with C, consult a C language reference guide (for example, [6] (p. 713), [9] (p. 713)) before you begin the process of writing UDFs and using them in your Ansys Fluent model.

This document also provides some basic information about the Scheme programming language (Part 2 Chapter 1) as it relates to Fluent interface macros, and assumes that you are an experienced programmer in Scheme. If you are unfamiliar with Scheme, consult a Scheme language reference guide (for example http://www.scheme.com/tspl4/) before you begin the process of writing GUI code for use with your Fluent UDF.

This document does not imply responsibility on the part of Ansys, Inc. for the accuracy or stability of solutions obtained using UDFs that are either user-generated or provided by ANSYS, Inc. Support for current license holders will be limited to guidance related to communication between a UDF and the Ansys Fluent solver. Other aspects of the UDF development process that include conceptual function design, implementation (writing C code), compilation and debugging of C source code, execution of the UDF, and function design verification will remain the responsibility of the UDF author.

UDF compiled libraries are specific to the computer architecture being used and the version of the Ansys Fluent executable being run and must be rebuilt any time Ansys Fluent is upgraded, your operating system changes, or the job is run on a different type of computer. Note that UDFs may need to be updated with new versions of Ansys Fluent.

2. Typographical Conventions

Several typographical conventions are used in this manual's text to help you find commands in the user interface.

- Different type styles are used to indicate graphical user interface items and text interface items. For example:

     Iso-Surface dialog box

     `surface/iso-surface` text command

- The text interface type style is also used when illustrating exactly what appears on the screen to distinguish it from the narrative text. In this context, user inputs are typically shown in boldface. For example,

```
solve/initialize/set-fmg-initialization

 Customize your FMG initialization:
   set the number of multigrid levels [5]

   set FMG parameters on levels ..

    residual reduction on level 1 is:  [0.001]
    number of cycles on level 1 is:  [10]  100

    residual reduction on level 2 is:  [0.001]
    number of cycles on level 2 is:  [50]  100
```

- Mini flow charts are used to guide you through the ribbon or the tree, leading you to a specific option, dialog box, or task page. The following tables list the meaning of each symbol in the mini flow charts.
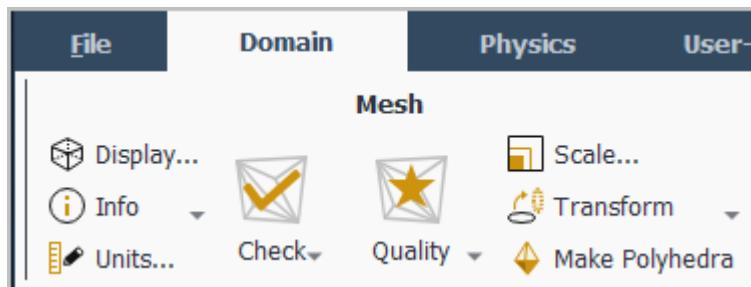
Table 1: Mini Flow Chart Symbol Descriptions

Symbol	Indicated Action
	Look at the ribbon
	Look at the tree
	Double-click to open task page
	Select from task page
	Right-click the preceding item

For example,

   Setting Up Domain → **Mesh** → **Transform** → **Translate...**
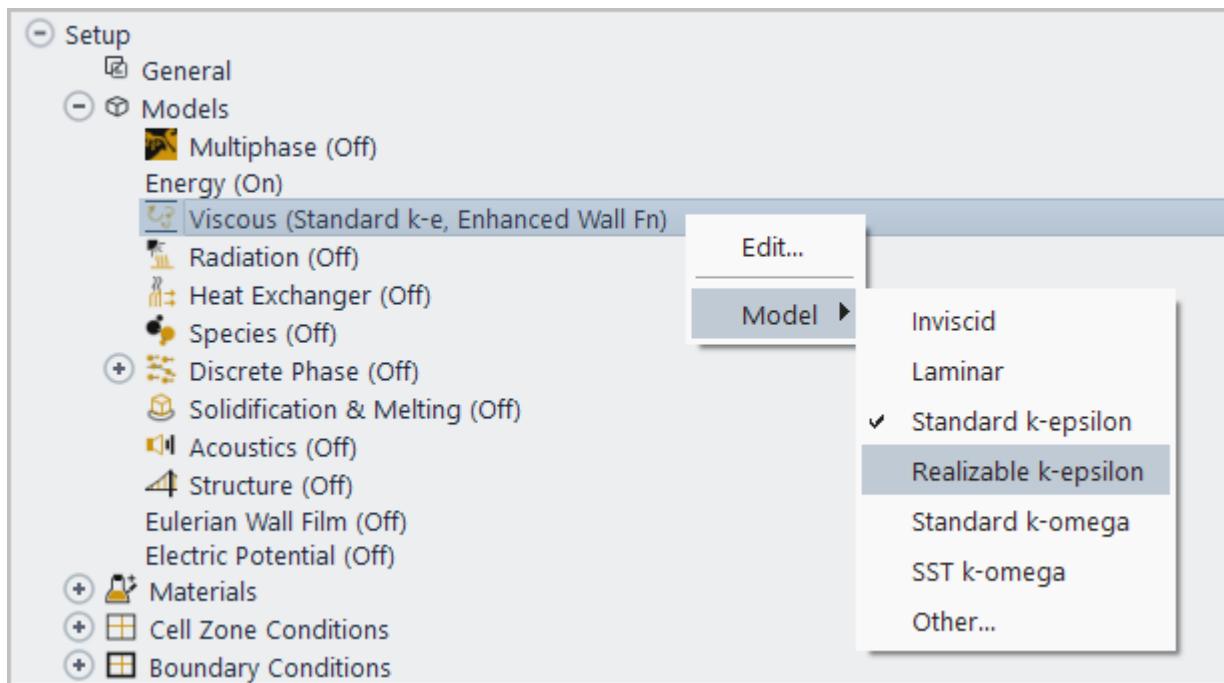
indicates selecting the **Setting Up Domain** ribbon tab, clicking **Transform** (in the **Mesh** group box) and selecting **Translate...**, as indicated in the figure below:

And

⊞ Setup → Models → Viscous 🖰 Model → Realizable k-epsilon
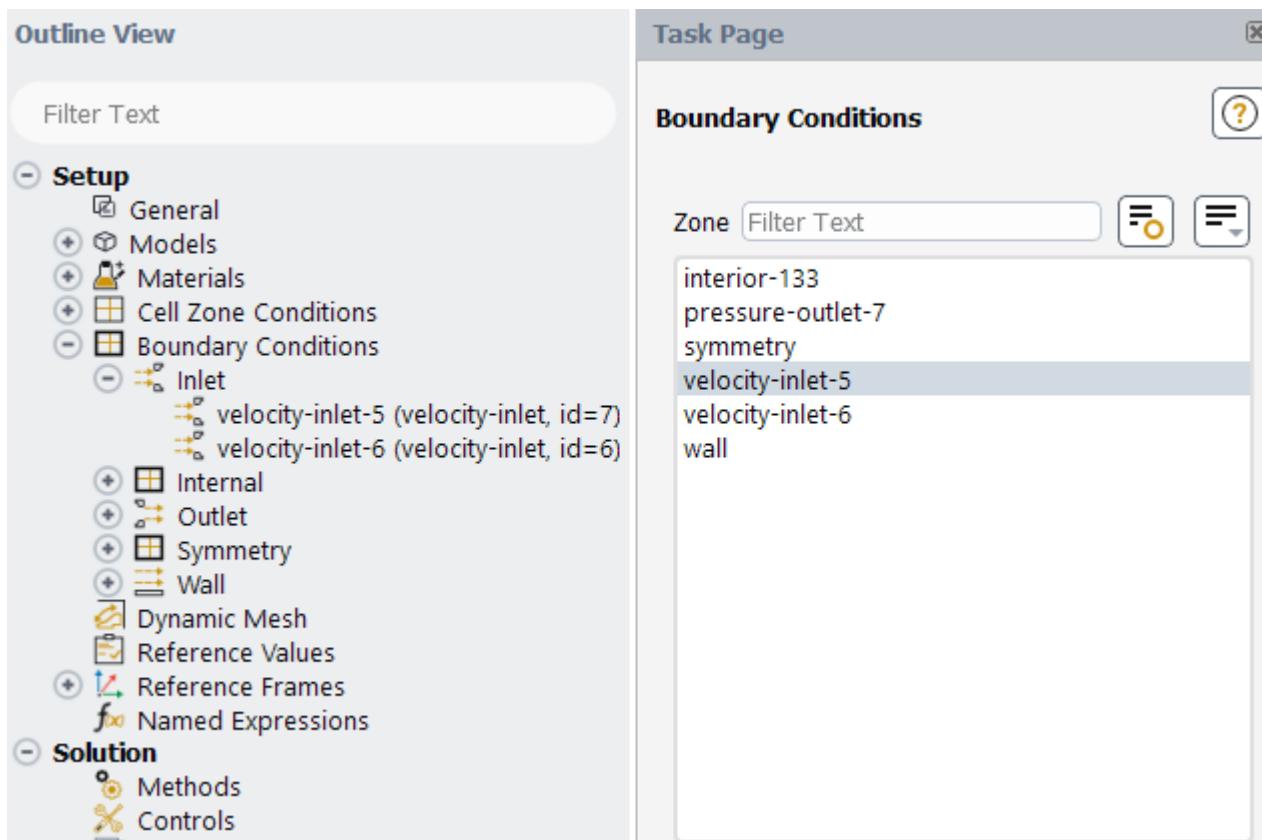
indicates expanding the **Setup** and **Models** branches, right-clicking **Viscous**, and selecting **Realizable k-epsilon** from the **Model** sub-menu, as shown in the following figure:

And

⊞ Setup → ✣ Boundary Conditions → ☰ velocity-inlet-5

indicates opening the task page as shown below:

In this manual, mini flow charts usually accompany a description of a dialog box or command, or a screen illustration showing how to use the dialog box or command. They show you how to quickly access a command or dialog box without having to search the surrounding material.

- In-text references to **File** ribbon tab selections can be indicated using a "/". For example **File/Write/Case...** indicates clicking the **File** ribbon tab and selecting **Case...** from the **Write** submenu (which opens the **Select File** dialog box).

3. Mathematical Conventions

- Where possible, vector quantities are displayed with a raised arrow (for example, $\vec{a}$, $\vec{A}$). Boldfaced characters are reserved for vectors and matrices as they apply to linear algebra (for example, the identity matrix, $\mathbf{I}$).

- The operator ∇, referred to as grad, nabla, or del, represents the partial derivative of a quantity with respect to all directions in the chosen coordinate system. In Cartesian coordinates, ∇ is defined to be

$$\frac{\partial}{\partial x}\vec{i} + \frac{\partial}{\partial y}\vec{j} + \frac{\partial}{\partial z}\vec{k} \tag{1}$$

∇ appears in several ways:

- The gradient of a scalar quantity is the vector whose components are the partial derivatives; for example,

$$\nabla p = \frac{\partial p}{\partial x} \vec{\imath} + \frac{\partial p}{\partial y} \vec{\jmath} + \frac{\partial p}{\partial z} \vec{k} \tag{2}$$

– The gradient of a vector quantity is a second-order tensor; for example, in Cartesian coordinates,

$$\nabla(\vec{v}) = \left(\frac{\partial}{\partial x} \vec{\imath} + \frac{\partial}{\partial y} \vec{\jmath} + \frac{\partial}{\partial z} \vec{k} \right) \left(v_x \vec{\imath} + v_j \vec{\jmath} + v_z \vec{k} \right) \tag{3}$$

This tensor is usually written as

$$\begin{pmatrix} \dfrac{\partial v_x}{\partial x} & \dfrac{\partial v_x}{\partial y} & \dfrac{\partial v_x}{\partial z} \\[2ex] \dfrac{\partial v_y}{\partial x} & \dfrac{\partial v_y}{\partial y} & \dfrac{\partial v_y}{\partial z} \\[2ex] \dfrac{\partial v_z}{\partial x} & \dfrac{\partial v_z}{\partial y} & \dfrac{\partial v_z}{\partial z} \end{pmatrix} \tag{4}$$

– The divergence of a vector quantity, which is the inner product between ∇ and a vector; for example,

$$\nabla \cdot \vec{v} = \frac{\partial v_x}{\partial x} + \frac{\partial v_y}{\partial y} + \frac{\partial v_z}{\partial z} \tag{5}$$

– The operator $\nabla \cdot \nabla$, which is usually written as ∇^2 and is known as the Laplacian; for example,

$$\nabla^2 T = \frac{\partial^2 T}{\partial x^2} + \frac{\partial^2 T}{\partial y^2} + \frac{\partial^2 T}{\partial z^2} \tag{6}$$

$\nabla^2 T$ is different from the expression $(\nabla T)^2$, which is defined as

$$(\nabla T)^2 = \left(\frac{\partial T}{\partial x} \right)^2 + \left(\frac{\partial T}{\partial y} \right)^2 + \left(\frac{\partial T}{\partial z} \right)^2 \tag{7}$$

• An exception to the use of ∇ is found in the discussion of Reynolds stresses in Turbulence in the *Fluent Theory Guide*, where convention dictates the use of Cartesian tensor notation. In this chapter, you will also find that some velocity vector components are written as u, v, and w instead of the conventional v with directional subscripts.

Part 1: Creating and Using User Defined Functions

Chapter 1: Overview of User-Defined Functions (UDFs)

This chapter contains an overview of user-defined functions (UDFs) and their usage in Ansys Fluent. UDF functionality is described in the following sections:

1.1. What is a User-Defined Function?

A user-defined function, or UDF, is a C or C++ function that can be dynamically loaded with the Ansys Fluent solver to enhance its standard features. For example, you can use a UDF to:

- Customize boundary conditions, material property definitions, surface and volume reaction rates, source terms in Ansys Fluent transport equations, source terms in user-defined scalar (UDS) transport equations, diffusivity functions, and so on.

- Adjust computed values on a once-per-iteration basis.

- Initialize of a solution.

- Perform asynchronous (on demand) execution of a UDF.

- Execute at the end of an iteration, upon exit from Ansys Fluent, or upon loading of a compiled UDF library.

- Enhance postprocessing.

- Enhance existing Ansys Fluent models (such as discrete phase model, multiphase mixture model, discrete ordinates radiation model).

UDFs are identified by a `.c` or `.cpp` extension (for example, `myudf.c`). One source file can contain a single UDF or multiple UDFs, and you can define multiple source files. See Appendix A: C Programming Basics (p. 677) for some basic information on C programming.

UDFs are defined using `DEFINE` macros provided by Ansys Fluent (see `DEFINE` Macros (p. 19)). They are coded using additional macros and functions (also supplied by Ansys Fluent) that access Ansys Fluent solver data and perform other tasks. See Additional Macros for Writing UDFs (p. 305) for details.

Every UDF must contain the `udf.h` file inclusion directive (`#include "udf.h"`) at the beginning of the source code file, which enables both the definition of `DEFINE` macros and other Ansys Fluent-provided macros and functions, and their inclusion in the compilation process. See Including the `udf.h` Header File in Your Source File (p. 6) for details.

Source files containing UDFs can be either interpreted or compiled in Ansys Fluent.

- For interpreted UDFs, source files are interpreted and loaded directly at *run time* in a single-step process.

- For compiled UDFs, the process involves two separate steps. A shared object code library is first built, then loaded, into Ansys Fluent. See Interpreting UDFs (p. 393) and Compiling UDFs (p. 399).

After being interpreted or compiled, UDFs will become visible and selectable in Ansys Fluent dialog boxes, and can be hooked to a solver by choosing the function name in the appropriate dialog box. This process is described in Hooking UDFs to Ansys Fluent (p. 423).

In summary, UDFs:

- Are written in the C (version C99 or earlier) or C++ programming languages.

- Must be defined using `DEFINE` macros supplied by Ansys Fluent.

- Must have an include statement for the `udf.h` file.

- Use predefined macros and functions to access Ansys Fluent solver data and to perform other tasks.

- Are executed as interpreted or compiled functions.

- Are hooked to an Ansys Fluent solver using a graphical user interface dialog boxes.

> **Important:**
>
> You are encouraged to parallelize all UDFs that you create or modify, as described in Parallelizing Your Serial UDF (p. 585).

1.2. Limitations

UDFs have the following limitations:

- Although the UDF capability in Ansys Fluent can address a wide range of applications, it is not possible to address every application using UDFs. Not all solution variables or Ansys Fluent models can be accessed by UDFs. If you are unsure whether a particular problem can be handled using a UDF, contact your technical support engineer for assistance.

- UDFs use and return values specified in SI units, except in rare cases as noted (such as `UNIVERSAL_GAS_CONSTANT` (p. 391)).

- UDFs cannot be loaded if: Ansys Fluent is not launched in UNC mode and Host and Node processess may be launched on remote machines.

- You may need to update your UDF when you use a new version of Ansys Fluent.

- The default serial version of Fluent (also referred to as -t1) interacts with a host process and a single compute node process, and so all user-defined functions (UDFs) are required to be written for parallel usage.

> **Important:**
>
> If using a non-parallelized UDF from previous releases causes an abnormal termination, then the UDF needs to be parallelized. For details, see Parallelizing Your Serial UDF (p. 585).

1.3. Defining Your UDF Using `DEFINE` Macros

UDFs are defined using Ansys Fluent-supplied function declarations. These function declarations are implemented in the code as macros, and are referred to in this document as `DEFINE` (all capitals) macros. Definitions for `DEFINE` macros are contained in the `udf.h` header file (see Appendix B: `DEFINE` Macro Definitions (p. 693) for a listing). For a complete description of each `DEFINE` macro and an example of its usage, refer to `DEFINE` Macros (p. 19).

The general format of a `DEFINE` macro is:

```
DEFINE_MACRONAME(udf_name, passed-in variables)
```

where the first argument in the parentheses is the name of the UDF that you supply. Name arguments are case-sensitive and *must* be specified in lowercase. After the function has been interpreted or compiled, the name that you choose for your UDF will become visible and selectable in drop-down lists in Ansys Fluent.

> **Important:**
>
> The name for your UDF must be unique to avoid conflicts with Fluent internal functions of the same name. Otherwise, the Linker may choose the wrong function, resulting in a segmentation violation.

The second set of input arguments to the `DEFINE` macro are variables that are passed into your function from the Ansys Fluent solver.

For example, the macro:

```
DEFINE_PROFILE(inlet_x_velocity, thread, index)
```

defines a boundary profile function named `inlet_x_velocity` with two variables, `thread` and `index`, that are passed into the function from Ansys Fluent. These passed-in variables are the boundary condition zone ID (as a pointer to the `thread`) and the `index` identifying the variable that is to be stored. After the UDF has been interpreted or compiled, its name (`inlet_x_velocity`) will become

visible and selectable in drop-down lists in the appropriate boundary condition dialog box (for example, **Velocity Inlet**) in Ansys Fluent.

Important:

When using UDFs:

- All of the arguments to a `DEFINE` macro need to be placed on the same line in your source code. Splitting the `DEFINE` statement onto several lines will result in a compilation error.

- There must be no spaces between the macro (for example, `DEFINE_PROFILE`) and the first parenthesis of the arguments, as this will cause an error in Windows.

- Do not include a `DEFINE` macro statement (such as `DEFINE_PROFILE`) within a comment in your source code. This will cause a compilation error.

1.3.1. Including the `udf.h` Header File in Your Source File

The `udf.h` header file contains:

- Definitions for `DEFINE` macros

- `#include` compiler directives for C or C++ library function header files

- Header files (for example, `mem.h`) for other Ansys Fluent-supplied macros and functions.

You must, therefore, include the `udf.h` file at the beginning of *every* UDF source code file using the `#include` compiler directive:

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

For example, when `udf.h` is included in the source file containing the `DEFINE` statement from the previous section,

```
#include "udf.h"

 DEFINE_PROFILE(inlet_x_velocity, thread, index)
```

upon compilation, the macro will expand to

```
void inlet_x_velocity(Thread *thread, int index)
```

Important:

You do not need to put a copy of `udf.h` in your local folder when you compile your UDF. The Ansys Fluent solver automatically reads the `udf.h` file from the following folder after your UDF is compiled, for example:

path\ANSYS Inc\v212\fluent\fluent21.2.0\src\udf

---

where `path` is the folder in which you have installed Ansys Fluent (by default, the path is `C:\Program Files`).

1.4. Interpreting and Compiling UDFs

Source code files containing UDFs can be either interpreted or compiled in Ansys Fluent. In both cases the functions are compiled, but the way in which the source code is compiled and the code that results from the compilation process is different for the two methods. These differences are explained in the following sections:

1.4.1. Compiled UDFs

Compiled UDFs are built in the same way that the Ansys Fluent executable itself is built: a `Makefile` script is used to invoke the system C or C++ compiler to build an object code library. You initiate this action in the **Compiled UDFs** dialog box by clicking **Build**. The object code library contains the native machine language translation of your higher-level C or C++ source code. The shared library must then be loaded into Ansys Fluent at run time by a process called "dynamic loading". You initiate this action in the **Compiled UDFs** dialog box by clicking **Load**. The object libraries are specific to the computer architecture being used, as well as to the particular version of the Ansys Fluent executable being run. The libraries must, therefore, be rebuilt any time Ansys Fluent is upgraded, when the computer's operating system level changes, or when the job is run on a different type of computer.

In summary, compiled UDFs are compiled from source files using the graphical user interface, in a two-step process. The process involves the **Compiled UDFs** dialog box, where you first build a shared library object file from a source file, and then load the shared library that was just built into Ansys Fluent.

1.4.2. Interpreted UDFs

Interpreted UDFs are interpreted from source files using the graphical user interface, but in a single-step process. The process, which occurs at *runtime*, involves using the **Interpreted UDFs** dialog box, where you **Interpret** a source file.

Inside Ansys Fluent, the source code is compiled into an intermediate, architecture-independent machine code using a C preprocessor. This machine code then executes on an internal emulator, or interpreter, when the UDF is invoked. This extra layer of code incurs a performance penalty, but enables an interpreted UDF to be shared effortlessly between different architectures, operating systems, and Ansys Fluent versions. If execution speed does become an issue, an interpreted UDF can always be run in compiled mode without modification.

The interpreter that is used for interpreted UDFs does not have all of the capabilities of a standard C compiler (which is used for compiled UDFs) and does not support all parallel UDF macros. For a detailed summary of limitations, see Limitations (p. 394).

> **Note:**
>
> In some situations you may encounter an error with interpreted UDFs related to the C preprocessor (CPP) distributed with Fluent. As a workaround, you can enter the full path to the system's preprocessor in the **CPP Command Name** field in the **Interpreted UDFs** dialog box; for example, `/usr/bin/cpp` or `gcc -E` (Linux only).

1.4.3. Differences Between Interpreted and Compiled UDFs

The major difference between interpreted and compiled UDFs is that interpreted UDFs cannot access Ansys Fluent solver data using direct structure references; they can only indirectly access data through the use of Ansys Fluent-supplied macros. This can be significant if, for example, you want to introduce new data structures in your UDF.

Here is a summary of the differences between interpreted and compiled UDFs:

- Interpreted UDFs:

  - Are portable to other platforms

  - Can all be run as compiled UDFs

  - Do not require a C compiler

  - Are slower than compiled UDFs

  - Are restricted in the use of the supported C programming language.

  - Cannot be written in C++

  - Cannot be linked to compiled system or user libraries

  - Can access data stored in an Ansys Fluent structure *only* using a predefined macro (see Additional Macros for Writing UDFs (p. 305)).

  See Interpreting UDFs (p. 393) for details on interpreting UDFs in Ansys Fluent.

- Compiled UDFs:

  - Execute faster than interpreted UDFs

  - Are not restricted in their use of the supported C or C++ programming languages

  - Can call functions written in other languages (specifics are system- and compiler-dependent)

  - Cannot necessarily be run as interpreted UDFs if they contain certain elements of the C language that the interpreter cannot handle

  See Compiling UDFs (p. 399) for details on compiling UDFs in Ansys Fluent.

Thus, when deciding which type of UDF to use for your Ansys Fluent model:

- Use interpreted UDFs for small, straightforward functions

- Use compiled UDFs for complex functions that:

  - Have a significant CPU requirement (for example, a property UDF that is called on a per-cell basis every iteration)

  - Require access to a shared library.

1.5. Hooking UDFs to Your Ansys Fluent Model

After your UDF source file is interpreted or compiled, the function(s) contained in the interpreted code or shared library will appear in drop-down lists in dialog boxes, ready for you to activate or "hook" to your CFD model. See Hooking UDFs to Ansys Fluent (p. 423) for details on how to hook a UDF to Ansys Fluent.

1.6. Mesh Terminology
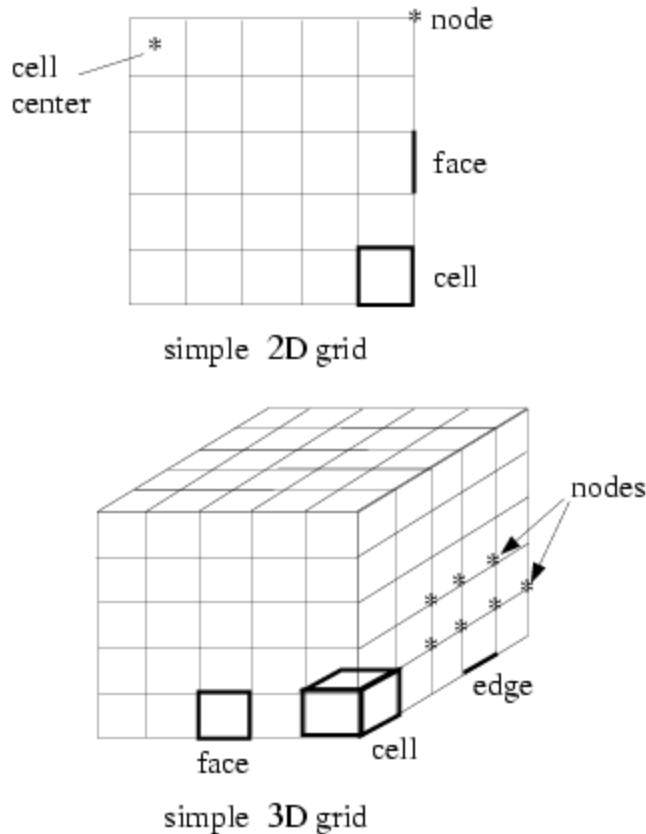
Most user-defined functions access data from an Ansys Fluent solver. Because solver data is defined in terms of mesh components, you will need to learn some basic mesh terminology before you can write a UDF.

A mesh is broken up into control volumes, or cells. Each cell is defined by a set of nodes, a cell center, and the faces that bound the cell (Figure 1.1: Mesh Components (p. 10)). Ansys Fluent uses internal data structures to define the domain(s) of the mesh; to assign an order to cells, cell faces, and nodes in a mesh; and to establish connectivity between adjacent cells.

A *thread* is a data structure in Ansys Fluent that is used to store information about a boundary or cell zone. *Cell threads* are groupings of cells, and *face threads* are groupings of faces. Pointers to thread data structures are often passed to functions and manipulated in Ansys Fluent to access the information about the boundary or cell zones represented by each thread. Each boundary or cell zone that you define in your Ansys Fluent model in a boundary conditions dialog box has an integer Zone ID that is associated with the data contained within the zone. You will not see the term "thread" in a dialog box in Ansys Fluent so you can think of a "zone" as being the same as a "thread" data structure when programming UDFs.

Cells and cell faces are grouped into zones that typically define the physical components of the model (for example, inlets, outlets, walls, fluid regions). A face will bound either one or two cells depending on whether it is a boundary face or an interior face. A domain is a data structure in Ansys Fluent that is used to store information about a collection of node, face threads, and cell threads in a mesh.

Figure 1.1: Mesh Components

node

A mesh point.

node thread

A grouping of nodes.

edge

A boundary of a face (3D).

face

A boundary of a cell (2D or 3D).

face thread

A grouping of faces.

cell

A control volume into which a domain is broken up.

cell center

The location where cell data is stored.

cell thread

A grouping of cells.

domain

A grouping of node, face, and cell threads.

1.7. Data Types in Ansys Fluent

In addition to standard C and C++ language data types such as `real`, `int`, and so on that you can use to define data in your UDF, there are Ansys Fluent-specific data types that are associated with solver data. These data types represent the computational units for a mesh (Figure 1.1: Mesh Components (p. 10)). Variables that are defined using these data types are typically supplied as arguments to `DEFINE` macros, as well as to other special functions that access Ansys Fluent solver data.

Some of the more commonly used Ansys Fluent data types are:

Node

A structure data type that stores data associated with a mesh point.

face_t

An integer data type that identifies a particular face within a face thread.

cell_t

An integer data type that identifies a particular cell within a cell thread.

Thread

A structure data type that stores data that is common to the group of cells or faces that it represents. In the `Thread` data type there is an array (`storage`) of pointers that each point to an array of cell or face values of a particular field variable (such as pressure, velocity, or gradients). Within that array of pointers, the index by which to identify a pointer to the array of (cell or face values of) a particular field variable is of the type `Svar`. For multiphase applications, there is a thread structure for each phase, as well as for the mixture. See Multiphase-specific Data Types (p. 16) for details.

Svar

An index used to identify a pointer in the `Thread` storage. All possible values for this index variable are given in the enumeration of that type in the file `src/storage/storage.h`. Note that some values are generated using macros like `SV_[COUPLED_]SOLUTION_VAR[_WITH_FC](...)`, `SV_UDS_I(...)` or `SV_UDSI_G(...)` and are therefore not found explicitly in the enumeration. If a pointer in the storage array of pointers still has the value `NULL`, memory for the corresponding field variable has not been allocated yet. A call to the function `Alloc_Storage_Vars(domain, SV_..., ..., SV_NULL);` can be used to change that allocation. The expression `if (NULLP(THREAD_STORAGE(t, SV_...)))` can be used to test whether the memory for a particular field variable has already been allocated on a given `Thread` or not.

Domain

A structure data type that stores data associated with a collection of node, face, and cell threads in a mesh. For single-phase applications, there is only a single domain structure. For multiphase applications, there are domain structures for each phase, the interaction between phases, as well as for the mixture. The mixture-level domain is the highest-level structure for a multiphase model. See Multiphase-specific Data Types (p. 16) for details.

Important:

All Ansys Fluent data types are case-sensitive.

When you use a UDF in Ansys Fluent, your function can access solution variables at individual cells or cell faces in the fluid and boundary zones. UDFs need to be passed appropriate arguments such as a thread reference (that is, a pointer to a particular thread) and the cell or face ID in order to enable individual cells or faces to be accessed. Note that a face ID or cell ID by itself does not uniquely identify the face or cell. A thread pointer is always required along with the ID to identify the thread to which the face (or cell) belongs.

Some UDFs are passed the cell index variable (`c`) as an argument (such as in `DEFINE_PROPERTY(my_function,c,t)`), or the face index variable (`f`) (such as in `DEFINE_UDS_FLUX(my_function,f,t,i)`). If the cell or face index variable (for example, `cell_t c, face_t f`) is not passed as an argument and is needed in the UDF, the variable is always available to be used by the function after it has been declared locally. See `DEFINE_UDS_FLUX` (p. 299) for an example.

The data structures that are passed to your UDF (as pointers) depend on the `DEFINE` macro you are using and the property or term you are trying to modify. For example, `DEFINE_ADJUST` UDFs are general-purpose functions that are passed a domain pointer (`d`) (such as in `DEFINE_ADJUST(my_function, d)`). `DEFINE_PROFILE` UDFs are passed a thread pointer (`t`) to the boundary zone to which the function is hooked, such as in `DEFINE_PROFILE(my_function, thread, i)`.

Some UDFs (such as `DEFINE_ON_DEMAND` functions) are not passed any pointers to data structures, while others are not passed the pointer the UDF needs. If your UDF needs to access a thread or domain pointer that is not *directly* passed by the solver through an argument, then you will need to use a special Ansys Fluent-supplied macro to obtain the pointer in your UDF. For example, `DEFINE_ADJUST` is passed only the domain pointer, so if your UDF needs a thread pointer, it will have to declare the variable locally and then obtain it using the special macro `Lookup_Thread`. An exception to this is if your UDF needs a thread pointer to loop over all of the cell threads or all the face threads in a domain (using `thread_c_loop(c,t)` or `thread_f_loop(f,t)`, respectively) and it is not passed to the `DEFINE` macro. Since the UDF will be looping over all threads in the domain, you will not need to use `Lookup_Thread` to get the thread pointer to pass it to the looping macro; you will just need to declare the thread pointer (and cell or face ID) locally before calling the loop. See `DEFINE_ADJUST` (p. 21) for an example.

As another example, if you are using `DEFINE_ON_DEMAND` (which is not passed any pointer argument) to execute an asynchronous UDF and your UDF needs a domain pointer, then the function will need to declare the domain variable locally and obtain it using `Get_Domain`. See `DEFINE_ON_DEMAND` (p. 33) for an example. Refer to Special Macros (p. 328) for details.

1.8. UDF Calling Sequence in the Solution Process

UDFs are called at predetermined times in the Ansys Fluent solution process. However, they can also be executed asynchronously (or "on demand") using a `DEFINE_ON_DEMAND` UDF. If a `DEFINE_EXECUTE_AT_END` UDF is used, then Ansys Fluent calls the function at the end of an iteration. A `DEFINE_EXECUTE_AT_EXIT` is called at the end of an Ansys Fluent session while a `DEFINE_EXECUTE_ON_LOADING` is called whenever a UDF compiled library is loaded. Understanding the context in which UDFs are called within Ansys Fluent's solution process may be important when you begin the process of writing UDF code, depending on the type of UDF you are writing. The solver contains call-outs that are linked to user-defined functions that you write. Knowing the sequencing of function calls within an iteration in the Ansys Fluent solution process can help you determine which data are current and available at any given time.

For more information, see the following:
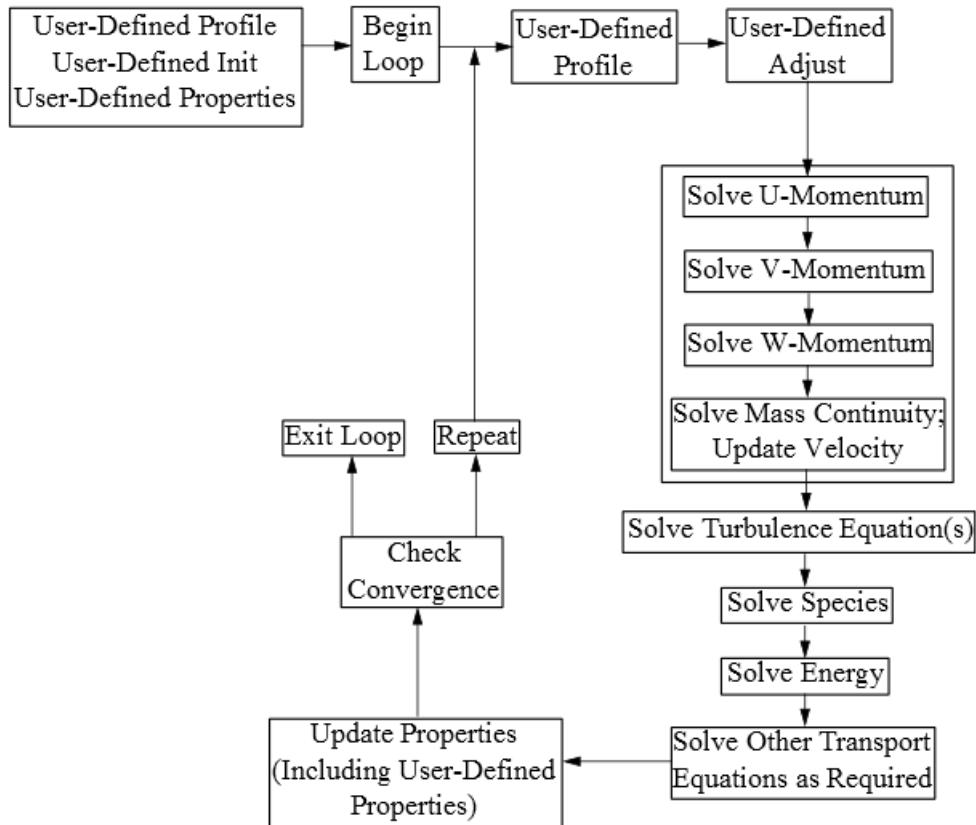
1.8.1. Pressure-Based Segregated Solver
1.8.2. Pressure-Based Coupled Solver
1.8.3. Density-Based Solver

1.8.1. Pressure-Based Segregated Solver

The solution process for the pressure-based segregated solver (Figure 1.2: Solution Procedure for the Pressure-Based Segregated Solver (p. 14)) begins with a two-step initialization sequence that is executed outside of the solution iteration loop. This sequence begins by initializing equations to user-entered (or default) values taken from the Ansys Fluent user interface. Next, `PROFILE` UDFs are called, followed by a call to `INIT` UDFs. Initialization UDFs overwrite initialization values that were previously set.

The solution iteration loop begins with the execution of `ADJUST` UDFs (note that this is true for the first iteration loop only; for subsequent iteration loops, `PROFILE` UDFs are executed prior to `ADJUST` UDFs). Next, momentum equations for u, v, and w velocities are solved sequentially, followed by mass continuity and velocity updates. Subsequently, the energy and species equations are solved, followed by turbulence and other scalar transport equations, as required. Note that `PROFILE` and `SOURCE` UDFs are called by each "Solve" routine for the variable currently under consideration (for example, species, velocity).
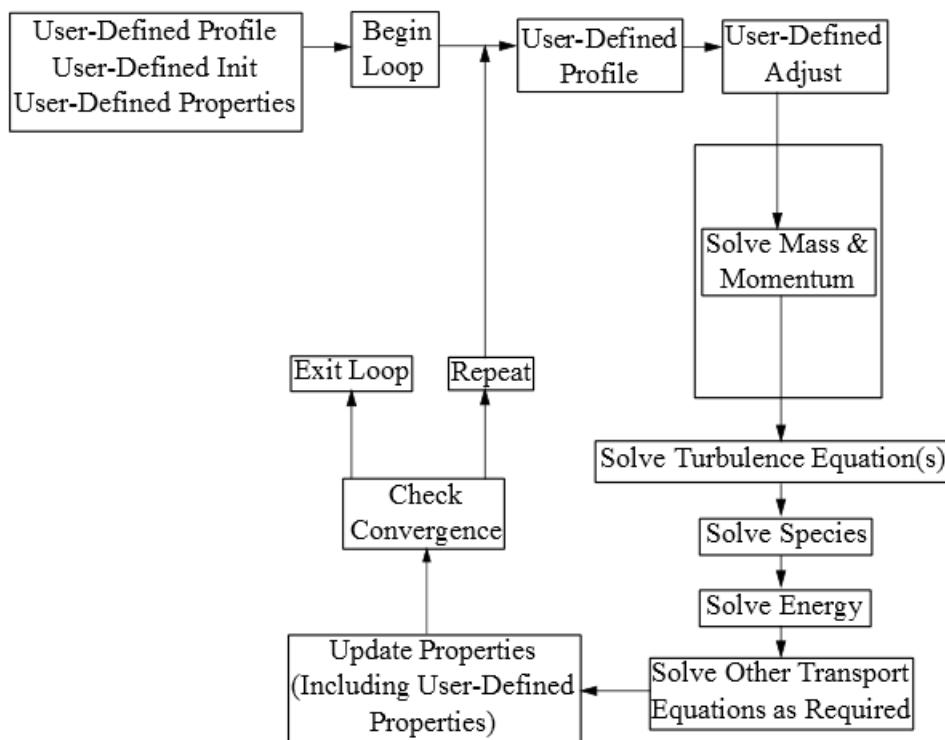
After the conservation equations, properties are updated, including `PROPERTY` UDFs. Thus, if your model involves the gas law, for example, the density will be updated at this time using the updated temperature (and pressure and/or species mass fractions). A check for either convergence or additional requested iterations is done, and the loop either continues or stops.

Figure 1.2: Solution Procedure for the Pressure-Based Segregated Solver

1.8.2. Pressure-Based Coupled Solver

The solution process for the pressure-based coupled solver (Figure 1.3: Solution Procedure for the Pressure-Based Coupled Solver (p. 15)) begins with a two-step initialization sequence that is executed outside of the solution iteration loop. This sequence begins by initializing equations to user-entered (or default) values taken from the Ansys Fluent user interface. Next, `PROFILE` UDFs are called, followed by a call to `INIT` UDFs. Initialization UDFs overwrite initialization values that were previously set.
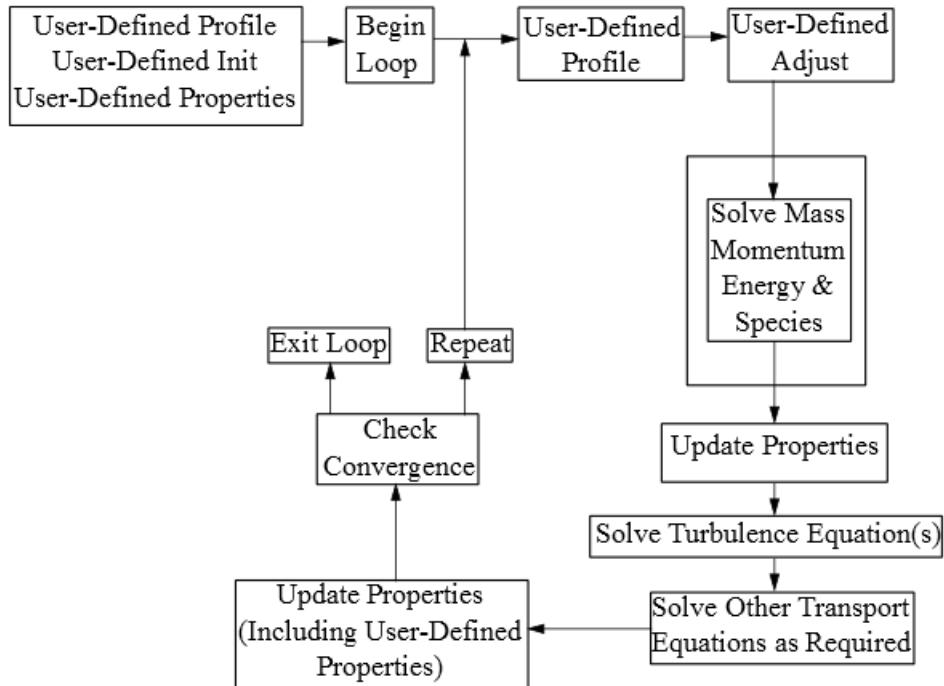
The solution iteration loop begins with the execution of `ADJUST` UDFs (note that this is true for the first iteration loop only; for subsequent iteration loops, `PROFILE` UDFs are executed prior to `ADJUST` UDFs). Next, Ansys Fluent solves the governing equations of continuity and momentum in a coupled fashion, which is simultaneously as a set, or vector, of equations. Energy, species transport, turbulence, and other transport equations as required are subsequently solved sequentially, and the remaining process is the same as the pressure-based segregated solver.

Figure 1.3: Solution Procedure for the Pressure-Based Coupled Solver

1.8.3. Density-Based Solver

As is the case for the other solvers, the solution process for the density-based solver (Figure 1.4: Solution Procedure for the Density-Based Solver (p. 16)) begins with a two-step initialization sequence that is executed outside the solution iteration loop. This sequence begins by initializing equations to user-entered (or default) values taken from the Ansys Fluent user interface. Next, `PROFILE` UDFs are called, followed by a call to `INIT` UDFs. Initialization UDFs overwrite initialization values that were previously set.

The solution iteration loop begins with the execution of `ADJUST` UDFs (note that this is true for the first iteration loop only; for subsequent iteration loops, `PROFILE` UDFs are executed prior to `ADJUST` UDFs). Next, Ansys Fluent solves the governing equations of continuity and momentum, energy, and species transport in a coupled fashion, which is simultaneously as a set, or vector, of equations. Turbulence and other transport equations as required are subsequently solved sequentially, and the remaining process is the same as the pressure-based segregated solver.

Figure 1.4: Solution Procedure for the Density-Based Solver

1.9. Special Considerations for Multiphase UDFs

In many cases, the UDF source code that you will write for a single-phase flow will be the same as for a multiphase flow. For example, there will be no differences between the C or C++ code for a single-phase boundary profile (defined using `DEFINE_PROFILE`) and the code for a multiphase profile, assuming that the function is accessing data *only* from the phase-level domain to which it is hooked in the graphical user interface. If your UDF is *not* explicitly passed a pointer to the thread or domain structure that it requires, you will need to use a special multiphase-specific macro (for example, `THREAD_SUB_THREAD`) to retrieve it. This is discussed in Additional Macros for Writing UDFs (p. 305).

See Appendix B:`DEFINE` Macro Definitions (p. 693) for a complete list of general-purpose `DEFINE` macros and multiphase-specific `DEFINE` macros that can be used to define UDFs for multiphase model cases.

1.9.1. Multiphase-specific Data Types
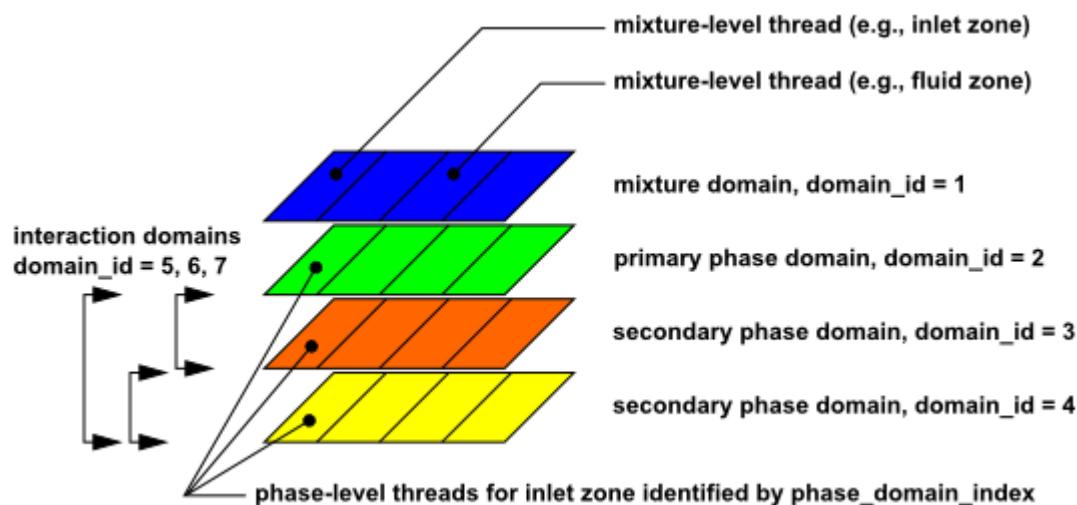
In addition to the Ansys Fluent-specific data types presented in Data Types in Ansys Fluent (p. 11), there are special thread and domain data structures that are specific to multiphase UDFs. These data types are used to store properties and variables for the mixture of all of the phases, as well as for each individual phase when a multiphase model (Mixture, VOF, or Eulerian) is used.

In a multiphase application, the top-level domain is referred to as the *superdomain*. Each phase occupies a domain referred to as a *subdomain*. A third domain type, the *interaction* domain, is introduced to allow for the definition of phase interaction mechanisms. When mixture properties and variables are needed (a sum over phases), the superdomain is used for those quantities, while the subdomain carries the information for individual phases.

In single-phase, the concept of a mixture is used to represent the sum over all the species (components), while in multiphase it represents the sum over all the phases. This distinction is important, because Ansys Fluent has the capability of handling multiphase multi-components, where, for example, a phase can consist of a mixture of species.

Because solver information is stored in thread data structures, threads must be associated with the superdomain as well as with each of the subdomains. That is, for each cell or face thread defined in the superdomain, there is a corresponding cell or face thread defined for each subdomain. Some of the information defined in one thread of the superdomain is shared with the corresponding threads of each of the subdomains. Threads associated with the superdomain are referred to as 'superthreads', while threads associated with the subdomain are referred to as phase-level threads, or 'subthreads'. The domain and thread hierarchy are summarized in Figure 1.5: Domain and Thread Structure Hierarchy (p. 17).

Figure 1.5: Domain and Thread Structure Hierarchy

Figure 1.5: Domain and Thread Structure Hierarchy (p. 17) introduces the concept of the `domain_id` and `phase_domain_index`. The `domain_id` can be used in UDFs to distinguish the superdomain from the primary and secondary phase-level domains. The superdomain (mixture domain) `domain_id` is always assigned the value of `1`. Interaction domains are also identified with the `domain_id`. The `domain_id` elements are not necessarily ordered sequentially, as shown in Figure 1.5: Domain and Thread Structure Hierarchy (p. 17).

The `phase_domain_index` can be used in UDFs to distinguish between the primary and secondary phase-level threads. `phase_domain_index` is always assigned the value of `0` for the primary phase-level thread.

The data structures that are passed to a UDF depend on the multiphase model that is enabled, the property or term that is being modified, the `DEFINE` macro that is used, and the domain that is to be affected (mixture or phase). To better understand this, consider the differences between the Mixture and Eulerian multiphase models. In the Mixture model, a single momentum equation is solved for a mixture whose properties are determined from the sum of its phases. In the Eulerian model, a momentum equation is solved for each phase. Ansys Fluent enables you to directly specify a momentum source for the mixture of phases (using `DEFINE_SOURCE`) when the mixture model is used, but not for the Eulerian model. For the latter case, you can specify momentum sources for the individual phases. Hence, the multiphase model, as well as the term being modified by the UDF, determines which domain or thread is required.

UDFs that are hooked to the mixture of phases are passed superdomain (or mixture-level) structures, while functions that are hooked to a particular phase are passed subdomain (or phase-level) structures. DEFINE_ADJUST and DEFINE_INIT UDFs are hardwired to the mixture-level domain. Other types of UDFs are hooked to different phase domains. For your convenience, Appendix C: Quick Reference Guide for Multiphase DEFINE Macros (p. 699) contains a list of multiphase models in Ansys Fluent and the phase on which UDFs are specified for the given variables. From this information, you can infer which domain structure is passed from the solver to the UDF.

Chapter 2: `DEFINE` Macros

This chapter contains descriptions of predefined `DEFINE` macros that you will use to define your UDF.

The chapter is organized in the following sections:

2.1. Introduction

`DEFINE` macros are predefined macros provided by Ansys, Inc. that must be used to define your UDF. A listing and discussion of each `DEFINE` macro is presented below. (Refer to Defining Your UDF Using `DEFINE` Macros (p. 5) for general information about `DEFINE` macros.) Definitions for `DEFINE` macros are contained within the `udf.h` file. For your convenience, they are provided in Appendix B: `DEFINE` Macro Definitions (p. 693).

For each of the `DEFINE` macros listed in this chapter, a source code example of a UDF that utilizes it is provided, where available. Many of the examples make extensive use of other macros presented in Additional Macros for Writing UDFs (p. 305). Note that not all of the examples in the chapter are complete functions that can be executed as stand-alone UDFs in Ansys Fluent. Examples are intended to demonstrate `DEFINE` macro usage only.

Special care must be taken for some UDFs that will be run in serial or parallel Ansys Fluent. See Parallel Considerations (p. 577) for details.

> **Important:**
>
> - You must place all of the arguments to a `DEFINE` macro on the same line in your source code. Splitting the `DEFINE` statement onto several lines will result in a compilation error.
>
> - Make sure that there are no spaces between the macro (such as `DEFINE_PROFILE`) and the first parenthesis of the arguments, as this will cause an error in Windows.
>
> - Do not include a `DEFINE` macro statement (such as `DEFINE_PROFILE`) within a comment in your source code. This will cause a compilation error.

2.2. General Purpose `DEFINE` Macros

The `DEFINE` macros presented in this section implement general solver functions that are independent of the model(s) you are using in Ansys Fluent. Table 2.1: Quick Reference Guide for General Purpose DEFINE Macros (p. 20) provides a quick reference guide to these `DEFINE` macros, the functions they are used to define, and the dialog boxes where they are activated or "hooked" to Ansys Fluent. Definitions of each `DEFINE` macro are contained in `udf.h` can be found in Appendix B:`DEFINE` Macro Definitions (p. 693).

Table 2.1: Quick Reference Guide for General Purpose DEFINE Macros

Function	`DEFINE` Macro	Dialog Box Activated In
manipulates variables	DEFINE_ADJUST	**User-Defined Function Hooks**
time step size (for time-dependent solutions)	DEFINE_DELTAT	**Run Calculation** task page
executes at end of iteration	DEFINE_EX-ECUTE_AT_END	**User-Defined Function Hooks**
executes at end of an Ansys Fluent session	DEFINE_EX-ECUTE_AT_EXIT	**User-Defined Function Hooks**
executes from a user-defined Scheme routine	DEFINE_EX-ECUTE_FROM_GUI	N/A
executes when a UDF library is loaded	DEFINE_EX-ECUTE_ON_LOADING	N/A
executes after a case file is read	DEFINE_EX-ECUTE_AFTER_CASE	N/A
executes after a data file is read	DEFINE_EX-ECUTE_AFTER_DATA	N/A
initializes variables	DEFINE_INIT	**User-Defined Function Hooks**
executes asynchronously	DEFINE_ON_DEMAND	**Execute On Demand**

Function	**`DEFINE` Macro**	**Dialog Box Activated In**
returns a value for a user defined report definition	`DEFINE_REPORT_DEFINI-TION_FN`	**User Defined Report Definition**
reads/writes variables to legacy case and data files	`DEFINE_RW_FILE`	**User-Defined Function Hooks**
reads/writes variables to CFF case and data files	`DEFINE_RW_HDF_FILE`	**User-Defined Function Hooks**

2.2.1. `DEFINE_ADJUST`

2.2.1.1. Description

`DEFINE_ADJUST` is a general-purpose macro that can be used to adjust or modify Ansys Fluent variables that are *not* passed as arguments. For example, you can use `DEFINE_ADJUST` to modify flow variables (for example, velocities, pressure) and compute integrals. You can also use it to integrate a scalar quantity over a domain and adjust a boundary condition based on the result. A function that is defined using `DEFINE_ADJUST` executes at every iteration and is called at the beginning of every iteration before transport equations are solved. For an overview of the Ansys Fluent solution process which shows when a `DEFINE_ADJUST` UDF is called, refer to Figure 1.2: Solution Procedure for the Pressure-Based Segregated Solver (p. 14), Figure 1.3: Solution Procedure for the Pressure-Based Coupled Solver (p. 15), and Figure 1.4: Solution Procedure for the Density-Based Solver (p. 16).

2.2.1.2. Usage

`DEFINE_ADJUST` (name, d)

Argument Type	**Description**
`symbol name`	UDF name.
`Domain *d`	Pointer to the domain over which the adjust function is to be applied. The domain argument provides access to all cell and face threads in the mesh. For multiphase flows, the pointer that is passed to the function by the solver is the mixture-level domain.

Function returns

`void`

There are two arguments to `DEFINE_ADJUST`: `name` and `d`. You supply `name`, the name of the UDF. `d` is passed by the Ansys Fluent solver to your UDF.

2.2.1.3. Example 1

The following UDF, named `my_adjust`, integrates the turbulent dissipation over the entire domain using `DEFINE_ADJUST`. This value is then displayed in the console. The UDF is called once every iteration. It can be executed as an interpreted or compiled UDF in Ansys Fluent.

```
/********************************************************************
   UDF for integrating turbulent dissipation and displaying it in the
   console
 ********************************************************************/

#include "udf.h"

DEFINE_ADJUST(my_adjust,d)
{
   Thread *t;
   /* Integrate dissipation. */
   real sum_diss=0.;
   cell_t c;

   thread_loop_c(t,d)
   {
    begin_c_loop(c,t)
     sum_diss += C_D(c,t)*
     C_VOLUME(c,t);
    end_c_loop(c,t)
   }

   printf("Volume integral of turbulent dissipation: %g\n", sum_diss);
}
```

2.2.1.4. Example 2

The following UDF, named `adjust_fcn`, specifies a user-defined scalar as a function of the gradient of another user-defined scalar, using `DEFINE_ADJUST`. The function is called once every iteration. It is executed as a compiled UDF in Ansys Fluent.

```
/********************************************************************
   UDF for defining user-defined scalars and their gradients
 ********************************************************************/

#include "udf.h"

DEFINE_ADJUST(adjust_fcn,d)
{
   Thread *t;
   cell_t c;
   real K_EL = 1.0;

   /* Do nothing if gradient isn't allocated yet. */
   if (! Data_Valid_P())
     return;

   thread_loop_c(t,d)
     {
     if (FLUID_THREAD_P(t))
       {
        begin_c_loop_all(c,t)
          {
          C_UDSI(c,t,1) +=
                 K_EL*NV_MAG2(C_UDSI_G(c,t,0))*C_VOLUME(c,t);
          }
        end_c_loop_all(c,t)
       }
     }
}
```

2.2.1.5. Hooking an Adjust UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_ADJUST` is interpreted (Interpreting UD-Fs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `adjust_fcn`) will become visible and selectable via the **User-Defined Function Hooks** dialog box in Ansys Fluent. Note that you can hook multiple adjust functions to your model. See Hooking `DEFINE_ADJUST` UDFs (p. 423) for details.

2.2.2. `DEFINE_DELTAT`

2.2.2.1. Description

`DEFINE_DELTAT` is a general-purpose macro that you can use to control the size of the time step during the solution of a transient problem. Note that this macro can be used only if **User-Defined Function** is selected from the **Type** drop-down list in the **Run Calculation** task page in Ansys Fluent.

2.2.2.2. Usage

`DEFINE_DELTAT` (name, d)

Argument Type	Description
`symbol name`	UDF name.
`Domain *d`	Pointer to domain over which the time stepping control function is to be applied. The domain argument provides access to all cell and face threads in the mesh. For multiphase flows, the pointer that is passed to the function by the solver is the mixture-level domain.

Function returns

`real`

There are two arguments to `DEFINE_DELTAT`: `name` and `domain`. You supply `name`, the name of the UDF. `domain` is passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the `real` value of the physical time step and return it to the solver.

2.2.2.3. Example

The following UDF, named `mydeltat`, is a simple function that shows how you can use `DEFINE_DELTAT` to change the value of the time step in a simulation. First, `CURRENT_TIME` is used to get the value of the current simulation time (which is assigned to the variable `flow_time`). Then, for the first `0.5` seconds of the calculation, a time step of `0.1` is set. A time step of `0.2` is set for the remainder of the simulation. The time step variable is then returned to the solver. See Time-Dependent Macros (p. 380) for details on `CURRENT_TIME`.

```
/**********************************************************************
   UDF that changes the time step value for a time-dependent solution
 **********************************************************************/
 #include "udf.h"
```

```
DEFINE_DELTAT(mydeltat,d)
{
  real time_step;
  real flow_time = CURRENT_TIME;
  if (flow_time < 0.5)
      time_step = 0.1;
  else
      time_step = 0.2;
  return time_step;
}
```

2.2.2.4. Hooking an Adaptive Time Step UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DELTAT` is interpreted (Interpreting UD-Fs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (e.g,. `mydeltat`) will become visible and selectable in the **Run Calculation** task page in Ansys Fluent. See Hooking `DEFINE_DELTAT` UDFs (p. 425) for details.

2.2.3. `DEFINE_EXECUTE_AT_END`

2.2.3.1. Description

`DEFINE_EXECUTE_AT_END` is a general-purpose macro that is executed at the end of an iteration in a steady-state run, or at the end of a time step in a transient run. You can use `DEFINE_EX-ECUTE_AT_END` when you want to calculate flow quantities at these particular times. Note that you do not have to specify whether your execute-at-end UDF gets executed at the end of a time step or the end of an iteration. This is done automatically when you select the steady or unsteady time method in your Ansys Fluent model.

2.2.3.2. Usage

`DEFINE_EXECUTE_AT_END` (name)

Argument Type	Description
`symbol name`	UDF name.

Function returns

`void`

There is only one argument to `DEFINE_EXECUTE_AT_END`: `name`. You supply `name`, the name of the UDF. Unlike `DEFINE_ADJUST`, `DEFINE_EXECUTE_AT_END` is not passed a domain pointer. Therefore, if your function requires access to a domain pointer, then you will need to use the utility `Get_Domain(ID)` to explicitly obtain it (see Domain Pointer (`Get_Domain`) (p. 330) and the example below). If your UDF requires access to a phase domain pointer in a multiphase solution, then it will need to pass the appropriate phase ID to `Get_Domain` in order to obtain it.

2.2.3.3. Example

The following UDF, named `execute_at_end`, integrates the turbulent dissipation over the entire domain using `DEFINE_EXECUTE_AT_END` and displays it in the console at the end of the current iteration or time step. It can be executed as an interpreted or compiled UDF in Ansys Fluent.

```
/**********************************************************************
   UDF for integrating turbulent dissipation and displaying it in the
   console at the end of the current iteration or time step
 **********************************************************************/

#include "udf.h"

DEFINE_EXECUTE_AT_END(execute_at_end)
{

   Domain *d;
   Thread *t;
   /* Integrate dissipation. */
   real sum_diss=0.;
   cell_t c;
   d = Get_Domain(1);   /* mixture domain if multiphase */

 thread_loop_c(t,d)
    {
      if (FLUID_THREAD_P(t))
        {
         begin_c_loop(c,t)
            sum_diss += C_D(c,t) * C_VOLUME(c,t);
         end_c_loop(c,t)
        }
     }

   printf("Volume integral of turbulent dissipation: %g\n", sum_diss);
   fflush(stdout);
}
```

2.2.3.4. Hooking an Execute-at-End UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_EXECUTE_AT_END` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `execute_at_end`) will become visible and selectable via the **User-Defined Function Hooks** dialog box in Ansys Fluent. Note that you can hook multiple end-iteration functions to your model. See Hooking `DEFINE_EXECUTE_AT_END` UDFs (p. 426) for details.

2.2.4. `DEFINE_EXECUTE_AT_EXIT`

2.2.4.1. Description

`DEFINE_EXECUTE_AT_EXIT` is a general-purpose macro that can be used to execute a function at the end of an Ansys Fluent session.

2.2.4.2. Usage

`DEFINE_EXECUTE_AT_EXIT` (name)

Argument Type	Description
`symbol name`	UDF name.

Function returns

`void`

There is only one argument to `DEFINE_EXECUTE_AT_EXIT`: `name`. You supply `name`, the name of the UDF.

2.2.4.3. Hooking an Execute-at-Exit UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_EXECUTE_AT_EXIT` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible and selectable via the **User-Defined Function Hooks** dialog box in Ansys Fluent. Note that you can hook multiple at-exit UDFs to your model. For details, see Hooking `DEFINE_EXECUTE_AT_EXIT` UDFs (p. 427).

2.2.5. `DEFINE_EXECUTE_FROM_GUI`

2.2.5.1. Description

`DEFINE_EXECUTE_FROM_GUI` is a general-purpose macro that you can use to define a UDF which is to be executed from a user-defined graphical user interface (GUI). For example, a C function that is defined using `DEFINE_EXECUTE_FROM_GUI` can be executed whenever a button is clicked in a user-defined GUI. Custom GUI components (dialog boxes, buttons, and so on) are defined in Ansys Fluent using the Scheme language.

2.2.5.2. Usage

`DEFINE_EXECUTE_FROM_GUI` (`name`, `libname`, `mode`)

Argument Type	Description
`symbol name`	UDF name.
`char *libname`	name of the UDF library that has been loaded in Ansys Fluent.
`int mode`	an integer passed from the Scheme program that defines the user-defined GUI.

Function returns

`void`

There are three arguments to `DEFINE_EXECUTE_FROM_GUI`: name, libname, and mode. You supply name, the name of the UDF. The variables `libname` and `mode` are passed by the Ansys Fluent solver to your UDF. The integer variable `mode` is passed from the Scheme program which defines the user-defined GUI, and represent the possible user options available from the GUI dialog box. A different C function in UDF can be called for each option. For example, the user-defined GUI

dialog box may have a number of buttons. Each button may be represented by different integers, which, when clicked, will execute a corresponding C function.

Important:

`DEFINE_EXECUTE_FROM_GUI` UDFs must be implemented as compiled UDFs, and there can be only one function of this type in a UDF library.

2.2.5.3. Example

The following UDF, named `reset_udm`, resets all user-defined memory (UDM) values when a reset button on a user-defined GUI dialog box is clicked. The clicking of the button is represented by 0, which is passed to the UDF by the Ansys Fluent solver.

```
/*********************************************************
   UDF called from a user-defined GUI dialog box to reset
   all user-defined memory locations
 *********************************************************/
#include "udf.h"

DEFINE_EXECUTE_FROM_GUI(reset_udm, myudflib, mode)
{
    Domain *domain = Get_Domain(1); /* Get domain pointer */
    Thread *t;
    cell_t c;
    int i;

    /* Return if mode is not zero */
    if (mode != 0) return;

    /* Return if no User-Defined Memory is defined in Ansys Fluent */
    if (n_udm == 0) return;

    /* Loop over all cell threads in domain */
    thread_loop_c(t, domain)
    {
/* Loop over all cells */
      begin_c_loop(c, t)
        {
          /* Set all UDMs to zero */
          for (i = 0; i < n_udm; i++)
            {
                C_UDMI(c, t, i) = 0.0;
            }
        }
      end_c_loop(c, t);
    }
}
```

2.2.5.4. Hooking an Execute From GUI UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_EXECUTE_FROM_GUI` is compiled (Compiling UDFs (p. 399)), the function will *not* need to be hooked to Ansys Fluent through any graphics dialog boxes. Instead, the function will be searched automatically by the Ansys Fluent solver when the execution of the UDF is requested (that is, when a call is made from a user-defined Scheme program to execute a C function).

2.2.6. `DEFINE_EXECUTE_ON_LOADING`

2.2.6.1. Description

`DEFINE_EXECUTE_ON_LOADING` is a general-purpose macro that can be used to specify a function that executes as soon as a compiled UDF library is loaded in Ansys Fluent. This is useful when you want to initialize or set up UDF models when a UDF library is loaded. (Alternatively, if you save your case file when a shared library is loaded, then the UDF will execute whenever the case file is subsequently read.)

Compiled UDF libraries are loaded using either the **Compiled UDFs** or the **UDF Library Manager** dialog box (see Load and Unload Libraries Using the UDF Library Manager Dialog Box (p. 418)). An `EXECUTE_ON_LOADING` UDF is the best place to reserve user-defined scalar (UDS) and user-defined memory (UDM) for a particular library (`Reserve_User_Scalar_Vars` (p. 355) and Reserving UDM Variables Using `Reserve_User_Memory_Vars` (p. 360)) as well as set UDS and UDM names (`Set_User_Scalar_Name` (p. 354) and `Set_User_Memory_Name` (p. 357)).

> **Important:**
>
> `DEFINE_EXECUTE_ON_LOADING` UDFs can be executed only as compiled UDFs.

> **Note:**
>
> Using the `DEFINE_EXECUTE_ON_LOADING` UDF for renaming DPM scalars is not re-commended because it may cause an abnormal termination of Ansys Fluent while reading a DPM case with such a UDF. This occurs because the `DEFINE_EX-ECUTE_ON_LOADING` type UDF is executed prior to the DPM model being enabled. Instead, you should use the `DEFINE_EXECUTE_AFTER_CASE` UDF to accomplish this task.

2.2.6.2. Usage

`DEFINE_EXECUTE_ON_LOADING` (name, libname)

Argument Type	Description
`symbol name`	UDF name.
`char *libname`	compiled UDF library name.

Function returns

`void`

There are two arguments to `DEFINE_EXECUTE_ON_LOADING`: `name` and `libname`. You supply a name for the UDF which will be used by Ansys Fluent when reporting that the `EXECUTE_ON_LOAD-ING` UDF is being run. The `libname` is set by Ansys Fluent to be the name of the library (for example, `libudf`) that you have specified (by entering a name or keeping the default `libudf`). `libname` is passed so that you can use it in messages within your UDF.

2.2.6.3. Example 1

The following simple UDF named `report_version`, prints a message on the console that contains the version and release number of the library being loaded.

```
#include "udf.h"

 static int version = 1;
 static int release = 2;

 DEFINE_EXECUTE_ON_LOADING(report_version, libname)
 {
     Message("\nLoading %s version %d.%d\n",libname,version,release);
 }
```

2.2.6.4. Example 2

The following source code contains two UDFs. The first UDF is an `EXECUTE_ON_LOADING` function that is used to reserve three UDMs (using `Reserve_User_Memory_Vars`) for a library and set unique names for the UDM locations (using `Set_User_Memory_Name`). The second UDF is an `ON_DEMAND` function that is used to set the values of the UDM locations after the solution has been initialized. The `ON_DEMAND` UDF sets the initial values of the UDM locations using `udm_offset`, which is defined in the on-loading UDF. Note that the on demand UDF must be executed *after* the solution is initialized to reset the initial values for the UDMs. See Reserving UDM Variables Using `Reserve_User_Memory_Vars` (p. 360) and `Set_User_Memory_Name` (p. 357) for more information on reserving and naming UDMs.

```
/*********************************************************************
 This file contains two UDFs: an execute on loading UDF that reserves three UDMs
 for libudf and renames the UDMs to enhance postprocessing,
 and an on-demand UDF that sets the initial value of the UDMs.
 *********************************************************************/
#include "udf.h"

 #define NUM_UDM 3
static int udm_offset = UDM_UNRESERVED;

 DEFINE_EXECUTE_ON_LOADING(on_loading, libname)
 {
     if (udm_offset == UDM_UNRESERVED) udm_offset =
         Reserve_User_Memory_Vars(NUM_UDM);

     if (udm_offset == UDM_UNRESERVED)
         Message("\nYou need to define up to %d extra UDMs in GUI and "
             "then reload current library %s\n", NUM_UDM, libname);
     else
       {
         Message("%d UDMs have been reserved by the current "
             "library %s\n",NUM_UDM, libname);

         Set_User_Memory_Name(udm_offset,"lib1-UDM-0");
         Set_User_Memory_Name(udm_offset+1,"lib1-UDM-1");
         Set_User_Memory_Name(udm_offset+2,"lib1-UDM-2");
       }
     Message("\nUDM Offset for Current Loaded Library = %d",udm_offset);
 }

 DEFINE_ON_DEMAND(set_udms)
 {
     Domain *d;
     Thread *ct;
     cell_t c;
     int i;
```

```
    d=Get_Domain(1);

    if(udm_offset != UDM_UNRESERVED)
      {
         Message("Setting UDMs\n");

         for (i=0;i<NUM_UDM;i++)
           {
               thread_loop_c(ct,d)
                 {
                     begin_c_loop(c,ct)
                       {
                         C_UDMI(c,ct,udm_offset+i)=3.0+i/10.0;
                       }
                     end_c_loop(c,ct)
                 }
           }
      }
    else
       Message("UDMs have not yet been reserved for library 1\n");
  }
```

2.2.6.5. Hooking an Execute On Loading UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_EXECUTE_ON_LOADING` is compiled (Compiling UDFs (p. 399)), the function will *not* need to be hooked to Ansys Fluent through any graphics dialog boxes. Instead, Ansys Fluent searches the newly loaded library for any UDFs of the type `EXECUTE_ON_LOADING`, and will automatically execute them in the order they appear in the library.

2.2.7. `DEFINE_EXECUTE_AFTER_CASE/DATA`

2.2.7.1. Description

`DEFINE_EXECUTE_AFTER_CASE` and `DEFINE_EXECUTE_AFTER_DATA` are general-purpose macros that can be used to specify a function that executes after the case and/or data file is read in Ansys Fluent. This is useful because it provides access to UDF functions after the case and/or data file is read.

Compiled UDF libraries are loaded using either the **Compiled UDFs** or the **UDF Library Manager** dialog box (see Load and Unload Libraries Using the UDF Library Manager Dialog Box (p. 418)).

> **Important:**
>
> `DEFINE_EXECUTE_AFTER_CASE` and `DEFINE_EXECUTE_AFTER_DATA` UDFs can be executed only as compiled UDFs.

2.2.7.2. Usage

`DEFINE_EXECUTE_AFTER_CASE` (name, libname) or

`DEFINE_EXECUTE_AFTER_DATA` (name, libname)

Argument Type	Description
symbol name	UDF name.

Argument Type	Description
`char *libname`	compiled UDF library name.

Function returns

`void`

There are two arguments to `DEFINE_EXECUTE_AFTER_CASE` and `DEFINE_EX-ECUTE_AFTER_DATA`: `name` and `libname`. You supply a name for the UDF which will be used by Ansys Fluent when reporting that the `EXECUTE_AFTER_CASE` or `EXECUTE_AFTER_DATA` UDF is being run. The `libname` is set by Ansys Fluent to be the name of the library (for example, `libudf`) that you have specified (by entering a name or keeping the default `libudf`). `libname` is passed so that you can use it in messages within your UDF.

2.2.7.3. Example

The following simple UDF named `after_case` and `after_data`, prints a message to the console that contains the name of the library being loaded.

```
#include "udf.h"

DEFINE_EXECUTE_AFTER_CASE(after_case, libname)
{
    Message("EXECUTE_AFTER_CASE called from $s\n", libname);
 }

DEFINE_EXECUTE_AFTER_DATA(after_data, libname)
{
    Message("EXECUTE_AFTER_DATA called from $s\n", libname);
}
```

2.2.7.4. Hooking an Execute After Reading Case and Data File UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_EXECUTE_AFTER_CASE` or `DEFINE_EX-ECUTE_AFTER_DATA` is compiled (Compiling UDFs (p. 399)), the function will *not* need to be hooked to Ansys Fluent through any graphics dialog boxes. Instead, Ansys Fluent searches the newly loaded library for any UDFs of the type `EXECUTE_AFTER_CASE` or `EXECUTE_AFTER_DATA`, and will automatically execute them in the order they appear in the library.

2.2.8. `DEFINE_INIT`

2.2.8.1. Description

`DEFINE_INIT` is a general-purpose macro that you can use to specify a set of initial values for your solution. `DEFINE_INIT` accomplishes the same result as patching, but does it in a different way, by means of a UDF. A `DEFINE_INIT` function is executed once per initialization and is called immediately after the default initialization is performed by the solver. Since it is called after the flow field is initialized, it is typically used to set initial values of flow quantities. For an overview of the Ansys Fluent solution process which shows when a `DEFINE_INIT` UDF is called, refer to Figure 1.2: Solution Procedure for the Pressure-Based Segregated Solver (p. 14), Figure 1.3: Solution

2.2.8.2. Usage

DEFINE_INIT (name, d)

Argument Type	Description
symbol name	UDF name.
Domain *d	Pointer to the domain over which the initialization function is to be applied. The domain argument provides access to all cell and face threads in the mesh. For multiphase flows, the pointer that is passed to the function by the solver is the mixture-level domain.

Function returns

void

There are two arguments to DEFINE_INIT: name and d. You supply name, the name of the UDF. d is passed from the Ansys Fluent solver to your UDF.

2.2.8.3. Example

The following UDF, named my_init_func, initializes flow field variables in a solution. It is executed once, at the beginning of the solution process. The function can be executed as an interpreted or compiled UDF in Ansys Fluent.

```
/************************************************************************
   UDF for initializing flow field variables
 ************************************************************************/

#include "udf.h"

DEFINE_INIT(my_init_func,d)
{
   cell_t c;
   Thread *t;
   real xc[ND_ND];

   /* loop over all cell threads in the domain */
   thread_loop_c(t,d)
     {

        /* loop over all cells */
        begin_c_loop_all(c,t)
          {
             C_CENTROID(xc,c,t);
             if (sqrt(ND_SUM(pow(xc[0] - 0.5,2.),
                     pow(xc[1] - 0.5,2.),
                     pow(xc[2] - 0.5,2.))) < 0.25)
               C_T(c,t) = 400.;
             else
               C_T(c,t) = 300.;
          }
        end_c_loop_all(c,t)
```

```
      }
   }
```

The macro `ND_SUM(a,b,c)` computes the sum of the first two arguments (2D) or all three arguments (3D). It is useful for writing functions involving vector operations so that the same function can be used for 2D and 3D. For a 2D case, the third argument is ignored. See Additional Macros for Writing UDFs (p. 305) for a description of predefined macros such as `C_CENTROID` and `ND_SUM`.

2.2.8.4. Hooking an Initialization UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_INIT` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `my_init_func`) will become visible and selectable via the **User-Defined Function Hooks** dialog box in Ansys Fluent. Note that you can hook multiple init functions to your model. See Hooking `DEFINE_INIT` UDFs (p. 429) for details.

2.2.9. `DEFINE_ON_DEMAND`

2.2.9.1. Description

`DEFINE_ON_DEMAND` is a general-purpose macro that you can use to specify a UDF that is executed "on demand" in Ansys Fluent, rather than having Ansys Fluent call it automatically during the calculation. Your UDF will be executed immediately, after it is activated, but it is not accessible while the solver is iterating. Note that the domain pointer `d` is not explicitly passed as an argument to `DEFINE_ON_DEMAND`. Therefore, if you want to use the domain variable in your on-demand function, you will need to first retrieve it using the `Get_Domain` utility provided by Ansys Fluent (shown in the example below). See Domain Pointer (`Get_Domain`) (p. 330) for details on `Get_Domain`.

2.2.9.2. Usage

`DEFINE_ON_DEMAND (name)`

Argument Type	Description
`symbol name`	UDF name.

Function returns

`void`

There is only one argument to `DEFINE_ON_DEMAND`: `name`. You supply `name`, the name of the UDF.

2.2.9.3. Example

The following UDF, named `on_demand_calc`, computes and prints the minimum, maximum, and average temperatures for the current data field. It then computes a temperature function

$$f(T) = \frac{T - T_{min}}{T_{max} - T_{min}}$$

(2.1)

and stores it in user-defined memory location 0 (which is allocated as described in Cell Macros (p. 308)). After you hook the on-demand UDF (as described in Hooking `DEFINE_ON_DEMAND` UDFs (p. 431)), the field values for $f(T)$ will be available in drop-down lists in postprocessing dialog boxes in Ansys Fluent. You can select this field by choosing **User Memory 0** in the **User-Defined Memory...** category. If you write a data file after executing the UDF, the user-defined memory field will be saved to the data file. This source code can be interpreted or compiled in Ansys Fluent.

```
/************************************************************************
   UDF to calculate temperature field function and store in
   user-defined memory. Also print min, max, avg temperatures.
 ************************************************************************/
#include "udf.h"

DEFINE_ON_DEMAND(on_demand_calc)
{
  Domain *d; /* declare domain pointer since it is not passed as an
       argument to the DEFINE macro */
  real tavg = 0.;
  real tmax = 0.;
  real tmin = 0.;
  real temp,volume,vol_tot;
  Thread *t;
  cell_t c;
  d = Get_Domain(1);  /* Get the domain using Ansys Fluent utility */

  /* Loop over all cell threads in the domain */
  thread_loop_c(t,d)
     {

     /* Compute max, min, volume-averaged temperature */

     /* Loop over all cells */
     begin_c_loop(c,t)
       {
          volume = C_VOLUME(c,t);  /* get cell volume */
          temp = C_T(c,t);    /* get cell temperature */

          if (temp < tmin || tmin == 0.) tmin = temp;
          if (temp > tmax || tmax == 0.) tmax = temp;

          vol_tot += volume;
          tavg += temp*volume;

       }
   end_c_loop(c,t)

   tavg /= vol_tot;

   printf("\n Tmin = %g  Tmax = %g  Tavg = %g\n",tmin,tmax,tavg);

   /* Compute temperature function and store in user-defined memory*/
   /*(location index 0)                    */

   begin_c_loop(c,t)
     {
        temp = C_T(c,t);
        C_UDMI(c,t,0) = (temp-tmin)/(tmax-tmin);
     }
   end_c_loop(c,t)

   }
}
```

`Get_Domain` is a macro that retrieves the pointer to a domain. It is necessary to get the domain pointer using this macro since it is not explicitly passed as an argument to `DEFINE_ON_DEMAND`. The function, named `on_demand_calc`, does not take any explicit arguments. Within the function

body, the variables that are to be used by the function are defined and initialized first. Following the variable declarations, a looping macro is used to loop over each cell thread in the domain. Within that loop another loop is used to loop over all the cells. Within the inner loop, the total volume and the minimum, maximum, and volume-averaged temperature are computed. These computed values are printed to the Ansys Fluent console. Then a second loop over each cell is used to compute the function $f(T)$ and store it in user-defined memory location 0. Refer to Additional Macros for Writing UDFs (p. 305) for a description of predefined macros such as `C_T` and `begin_c_loop`.

2.2.9.4. Hooking an On-Demand UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_ON_DEMAND` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `on_demand_calc`) will become visible and selectable in the **Execute On Demand** dialog box in Ansys Fluent. See Hooking `DEFINE_ON_DEMAND` UDFs (p. 431) for details.

2.2.10. `DEFINE_REPORT_DEFINITION_FN`

2.2.10.1. Description

`DEFINE_REPORT_DEFINITION_FN` is a macro that you can use to specify a UDF for reporting a single-valued expression that can be plotted, written, or printed to the console as a report definition.

2.2.10.2. Usage

The `DEFINE_REPORT_DEFINITION_FN` macro takes a single argument, *name*, as the name of the function you are creating. Your user-defined function should return a real value that will be the value of the report definition.

Note:

If you need to access the value of another report definition, use the `Get_Report_Definition_Values` (p. 388) macro.

`DEFINE_REPORT_DEFINITION_FN (name)`

Argument Type	Description
`symbol name`	name of the user defined report definition function

Function returns

`real`

2.2.10.3. Example

This function computes the volumetric flow rate at an inlet, demonstrating the ability to perform operations on an input variable and output the result for plotting, printing, or writing to a file as part of a user-defined report definition.

```
#include "udf.h"
DEFINE_REPORT_DEFINITION_FN(volume_flow_rate_inlet)
{
  real inlet_velocity = Get_Input_Parameter("vel_in");
  real inlet_area   =  0.015607214;
  real volumeFlow = inlet_velocity*inlet_area;
  return volumeFlow;
}
```

Note:

This example uses the `Get_Input_Parameter` API, indicating that this Fluent case file has the inlet velocity specified as an input parameter.

2.2.10.4. Hooking a User Defined Report Definition to Ansys Fluent

After the UDF that you have defined using `DEFINE_REPORT_DEFINITION_FN` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `on_demand_calc`) will become visible and selectable in the **User Defined Report Definition** dialog box in Fluent. See User Defined Report Definition Function Hooking in the *Fluent User's Guide* for details.

2.2.11. `DEFINE_RW_FILE`

2.2.11.1. Description

`DEFINE_RW_FILE` is a general-purpose macro that you can use to specify customized information that is to be written to a legacy case or data file (that is, a `.cas` or `.dat` file), or read from a legacy case or data file. You can save and restore custom variables of any data type (for example, integer, real, CXBoolean, structure) using `DEFINE_RW_FILE`. It is often useful to save dynamic information (for example, number of occurrences in conditional sampling) while your solution is being calculated, which is another use of this function. Note that the read order and the write order must be the same when you use this function.

When using the `DEFINE_RW_FILE`, you should be attentive to which parts of the function should be done by the host, and which by the compute nodes. For details, see Compiler Directives (p. 587).

Important:

Starting in Ansys Fluent version 18.2, the `DEFINE_RW_FILE` macro is not supported for serial UDFs in the following circumstances:

- In a compiled UDF on Windows

- In an interpreted UDF on any platform

As a workaround for such unsupported combinations, you can parallelize the UDF (as described in Parallelizing Your Serial UDF (p. 585)).

2.2.11.2. Usage

`DEFINE_RW_FILE` (name, fp)

Argument Type	Description
`symbol name`	UDF name.
`FILE *fp`	Pointer to the legacy file you are reading or writing.

Function returns

`void`

There are two arguments to `DEFINE_RW_FILE`: `name` and `fp`. You supply `name`, the name of the UDF. `fp` is passed from the solver to the UDF.

2.2.11.3. Example

The following C source code listing contains examples of functions that write information to a legacy data file and read it back. These functions are concatenated into a single source file that can be interpreted or compiled in Ansys Fluent.

```
/*************************************************************************
   UDFs that increment a variable, write it to a legacy data file
   and read it back in
 *************************************************************************/
#include "udf.h"

int kount = 0; /* define global variable kount */

DEFINE_ADJUST(demo_calc,d)
{
   kount++;
   printf("kount = %d\n",kount);
}
DEFINE_RW_FILE(writer,fp)
{
   printf("Writing UDF data to legacy data file...\n");
#if !RP_NODE
   fprintf(fp,"%d",kount); /* write out kount to legacy data file */
#endif
}
DEFINE_RW_FILE(reader,fp)
{
   printf("Reading UDF data from legacy data file...\n");
#if !RP_NODE
   fscanf(fp,"%d",&kount); /* read kount from legacy data file */
#endif
}
```

At the top of the listing, the integer `kount` is defined and initialized to zero. The first function (`demo_calc`) is an `ADJUST` function that increments the value of `kount` at each iteration, since the `ADJUST` function is called once per iteration. (See `DEFINE_ADJUST` (p. 21) for more information about `ADJUST` functions.) The second function (`writer`) instructs Ansys Fluent to write the

current value of `kount` to the legacy data file, when the legacy data file is saved. The third function (`reader`) instructs Ansys Fluent to read the value of `kount` from the legacy data file, when the data file is read.

The functions work together as follows. If you run your calculation for, say, 10 iterations (`kount` has been incremented to a value of 10) and save the legacy data file, then the current value of `kount` (10) will be written to your legacy data file. If you read the data back into Ansys Fluent and continue the calculation, `kount` will start at a value of 10 and will be incremented at each iteration. Note that you can save as many static variables as you want, but you must be sure to read them in the same order in which they are written.

2.2.11.4. Hooking a Read/Write Legacy Case or Data File UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_RW_FILE` is interpreted (Interpreting UD-Fs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `writer`) will become visible and selectable via the **User-Defined Function Hooks** dialog box in Ansys Fluent. Note that you can hook multiple read/write functions to your model. See Hooking `DEFINE_RW_FILE` and `DEFINE_RW_HDF_FILE` UDFs (p. 432) for details.

2.2.12. `DEFINE_RW_HDF_FILE`

2.2.12.1. Description

`DEFINE_RW_HDF_FILE` is a general-purpose macro that you can use to read or write customized information from or to case or data files written in the Common Fluids Format (CFF), that is, `.cas.h5` or `.dat.h5` files. You can read and write custom datasets of real values as well as attributes of various types.

2.2.12.2. Usage

`DEFINE_RW_HDF_FILE (name, filename)`

Argument Type	Description
`symbol name`	UDF name.
`char *filename`	Name of the CFF file you are reading or writing.

Function returns

`void`

There are two arguments to `DEFINE_RW_HDF_FILE`: `name` and `filename`. You supply `name`, the name of the UDF. `filename` is passed from the solver to the UDF. Because of the structured nature of the Common Fluids Format, a series of helper functions are made available for performing the low-level read/write operations. These are detailed in Helper Functions (p. 39).

2.2.12.3. Helper Functions

CFF files use a compressed binary format. In order to make reading and writing these files easier, a series of helper functions are made available for use in your UDF. The following terminology is used in the descriptions of the helper functions:

Datasets

can be thought of as arrays that store data. Typically the data are stored as real values.

Groups

are analogous to directories on a file system. A group can contain other groups, datasets, or a combination of both.

Attributes

are name/value pairs that assigned to datasets or groups and are generally used for storing metadata. For example, a dataset may have an attribute for the number of elements in the dataset.

Link

is an umbrella term that may refer to groups or datasets, but not to attributes.

Path

is a string representation of a link in the CFF file. The path of the root group of the file is denoted by `/`. Any link in the file can be identified by specifying the absolute path to the link (for example, `/path/to/link`).

The following functions are made available by including `hdfio.h` in your UDF source file and are used to perform CFF reading and writing. Note that all functions prepend `/user` to the provided `path` parameter. Thus, all user-defined data written or accessed by these functions is stored under the group `/user`.

Write_Complete_User_Dataset

#include "hdfio.h"

void **Write_Complete_User_Dataset** (*filename*, *path*, *ptr*, *nelems*);

char* *filename* ;

```
char* path ;
real* ptr ;
size_t nelems ;
```

filename

> Name of the CFF file to which data is to be written. In a DEFINE_RW_HDF_FILE macro, this is generally passed in by Fluent.

path

> Path in the file at which the dataset will be written.

ptr

> Pointer to the data that will be written. Note that the data type must be real.

nelems

> Number of data elements to be written from a node.

This is the basic function for writing data to a CFF file. The function assumes that data is only in nodes and is already arranged sequentially according to node rank. Therefore, data in node0 is written first followed by data in node1, and so on. A new dataset is created at *path* and the size of the created dataset is the sum of *nelems* across all nodes.

Write_Partial_User_Dataset

#include "hdfio.h"

void **Write_Partial_User_Dataset** (*filename*, *path*, *ptr*, *nelems*, *datasetsize*, *datasetoffset*);

```
char* filename ;
char* path ;
real* ptr ;
size_t nelems ;
size_t datasetsize ;
size_t datasetoffset ;
```

filename

> Name of the CFF file to which data is to be written. In a DEFINE_RW_HDF_FILE macro, this is generally passed in by Fluent.

path

> Path in the file at which the dataset will be written.

ptr

> Pointer to the data that will be written. Note that the data type must be real.

nelems

> Number of data elements to be written from a node.

datasetsize

> Total size of the dataset into which the data will be written. This must be larger than the sum of `nelems` across all nodes.

datasetoffset

> The offset within the dataset at which writing will start.

This function can be used when it is desirable to write data to the same dataset through multiple calls. The dataset will be created the first time and data will be written into the dataset in subsequent calls from each compute node.

Write_User_Attributes

#include "hdfio.h"

```
void Write_User_Attributes (filename, path, name, mpttype, value, ...,
NULL );

char* filename ;
char* path ;
char* name ;
MPT_Datatype mpttype ;
type value ;
... ;
NULL   ;
```

filename

> Name of the CFF file.

path

> Path to the link in the file for which the attributes will be written.

name

> Name of the attribute.

mpttype

> The datatype of the attribute value. This can be one of MPT_SHORT, MPT_INT, MPT_LONG, MPT_LONG_LONG, MPT_SIZE_T, MPT_DOUBLE

value

> Value of the attribute. The `type` of the attribute should be short, int, long, long long, size_t, or double according to the value of `mpttype`

...

Multiple attributes can be specified by including additional triplets of *name*, *mpttype*, and *value*.

NULL

the list of parameters must be terminated with NULL.

Read_Complete_User_Dataset

#include "hdfio.h"

void **Read_Complete_User_Dataset** (*filename*, *path*, *ptr*, *nelems*);

char* *filename* ;
char* *path* ;
real* *ptr* ;
size_t *nelems* ;

filename

Name of the CFF file from which data is to be read. In a DEFINE_RW_HDF_FILE macro, this is generally passed in by Fluent.

path

Path in the file from which the dataset will be read.

ptr

Pointer to the location where the read data will be stored. Note that the data type is assumed to be real.

nelems

Number of data elements to be read into a node.

This is the basic function for reading data from an CFF file. As in the Write_Complete_User_Data function, the data is read sequentially according to node rank. Therefore, data that is read first fills node0 followed by node1, and so on.

Read_Partial_User_Dataset

#include "hdfio.h"

void **Read_Partial_User_Dataset** (*filename*, *path*, *ptr*, *nelems*, *datasetoffset*);

char* *filename* ;
char* *path* ;
real* *ptr* ;
size_t *nelems* ;
size_t *datasetoffset* ;

filename

> Name of the CFF file from which data is to be read. In a DEFINE_RW_HDF_FILE macro, this is generally passed in by Fluent.

path

> Path in the file from which the dataset will be read.

ptr

> Pointer to the location where the read data will be stored. Note that the data type is assumed to be real.

nelems

> Number of data elements to be read into a node.

datasetoffset

> The offset within the dataset at which reading will start.

This function can be used to read a portion of a dataset from a CFF file.

Read_User_Attributes

#include "hdfio.h"

```
void Read_User_Attributes (filename, path, name, mpttype, ptr, ...,  NULL
);

char* filename ;
char* path ;
char* name ;
MPT_Datatype mpttype ;
type* ptr ;
... ;
NULL   ;
```

filename

> Name of the CFF file.

path

> Path to the link in the file for which the attributes will be read.

name

> Name of the attribute to read.

mpttype

> The datatype of the attribute value. This can be one of MPT_SHORT, MPT_INT, MPT_LONG, MPT_LONG_LONG, MPT_SIZE_T, MPT_DOUBLE.

ptr

Pointer to the location in which to store the attribute value. The pointer `type` should be short, int, long, long long, size_t, or double according to the value of `mpttype`

...

Multiple attributes can be read by including additional triplets of `name`, `mpttype`, and `ptr`.

NULL

the list of parameters must be terminated with `NULL`.

Get_Next_User_Link_Name

#include "hdfio.h"

```
void Get_Next_User_Link_Name (filename, path, prev_name, next_name);

char* filename ;
char* path ;
char* prev_name ;
char* next_name ;
```

filename

Name of the CFF file.

path

Path to a group in the file under which the links reside.

prev_name

Name of a link under the group at `path`. The name of the next link after `prev_name` (based on order of creation in the document) will be returned in `next_name`. A value of NULL can be used to get the first link within a group.

next_name

The name of the link following `prev_name` is returned in `next_name`. If `prev_name` is the last link in the group, an empty string will be returned.

This function is typically used to iterate over the links under a group. The iteration is started by passing NULL as the value for `prev_name`. Iteration continues by using the value returned in `next_name` as the value of `prev_name` in the next iteration. The iteration is complete when an empty string is returned in `next_name`.

2.2.12.4. Examples

```
/*
 * UDF that writes a complete dataset to a CFF file, sets an attribute for the dataset
 * and then reads it back.  These demonstrate the use of the following helper functions:
 *    Write_Complete_User_Dataset
 *    Write_User_Attributes
```

```
 *      Read_Complete_User_Dataset
 *      Read_User_Attributes
 */
#include "udf.h"
#include "hdfio.h"

#define ELEM_COUNT 50

DEFINE_RW_HDF_FILE(write_complete_dataset, filename)
{
  size_t i, nelems = ELEM_COUNT;
  real* ptr = NULL;
  char* path = "/test/complete_data";


  /* Assign equal number of elements in all the nodes
   * and fill with some values. Write a dataset with
   * all the values at the end. Also write an attribute
   * with total number of elements.
   */
#if RP_NODE
  ptr = (real*)CX_Malloc(sizeof(real) * nelems);
  for (i = 0; i < nelems; ++i) {
    ptr[i] = (real)((myid + 1) * i);
  }
#endif
  Write_Complete_User_Dataset(filename, path, ptr, nelems);
  Write_User_Attributes(filename, path,
                        "totalElems", MPT_SIZE_T, (size_t)(nelems * compute_node_count),
                        NULL
                        );
  if (NNULLP(ptr)) {
    CX_Free(ptr);
  }
}


DEFINE_RW_HDF_FILE(read_complete_dataset, filename)
{
  size_t nelems = 0, totalElems = 0;
  real* ptr = NULL;
  char* path = "/test/complete_data";

  /* Read complete dataset and check the elements. */
  Read_User_Attributes(filename, path,
                        "totalElems", MPT_SIZE_T, &totalElems,
                        NULL
                        );
#if RP_NODE
  nelems = totalElems / compute_node_count;
  ptr = (real*)CX_Malloc(sizeof(real) * nelems);
#endif
  Read_Complete_User_Dataset(filename, path, ptr, nelems);
  if (NNULLP(ptr)) {
    CX_Free(ptr);
  }
}
```

2.2.12.5. Hooking a Read/Write CFF Case or Data File UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_RW_HDF_FILE is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first DEFINE macro argument (for example, write_complete_dataset) will become visible and selectable via the **User-Defined Function Hooks** dialog box in Ansys Fluent. Note that you can hook multiple read/write functions to your model. See Hooking DEFINE_RW_FILE and DEFINE_RW_HDF_FILE UDFs (p. 432) for details.

2.3. Model-Specific `DEFINE` Macros

The `DEFINE` macros presented in this section are used to set parameters for a particular model in Ansys Fluent. Table 2.2: Quick Reference Guide for Model-Specific DEFINE Functions (p. 47) – Table 2.6: Quick Reference Guide for Model-Specific DEFINE Functions MULTIPHASE ONLY (p. 52) provides a quick reference guide to the `DEFINE` macros, the functions they are used to define, and the dialog boxes where they are activated in Ansys Fluent. Definitions of each `DEFINE` macro are listed in `udf.h`. For your convenience, they are listed in Appendix B: `DEFINE` Macro Definitions (p. 693).

Table 2.2: Quick Reference Guide for Model-Specific DEFINE Functions

Function	`DEFINE` Macro	Dialog Box Activated In
anisotropic thermal conductivity	`DEFINE_ANISOTROPIC_CON-DUCTIVITY`	**Create/Edit Materials**
mixing constant	`DEFINE_CPHI`	**User-Defined Function Hooks**
homogeneous net mass reaction rate for all species, integrated over a time step	`DEFINE_CHEM_STEP`	**User-Defined Function Hooks**
species mass or UDS diffusivity	`DEFINE_DIFFUSIVITY`	**Create/Edit Materials**
diffusive reflectivity for discrete ordinates (DO) model	`DEFINE_DOM_DIFFUSE_ RE-FLECTIVITY`	**User-Defined Function Hooks**

Function	**DEFINE Macro**	**Dialog Box Activated In**
source for DO model	DEFINE_DOM_SOURCE	**User-Defined Function Hooks**
specular reflectivity for DO model	DEFINE_DOM_SPECULAR_ REFLECTIVITY	**User-Defined Function Hooks**
electrochemical reaction kinetics parameter	DEFINE_EC_KINETICS_ PARAMETER	**Reaction**
electrochemical reaction rate	DEFINE_EC_RATE	**User-Defined Function Hooks**
reaction rates for the EDC model	DEFINE_EDC_MDOT	**Species Model**
scales for the EDC model	DEFINE_EDC_SCALES	**Species Model**
emissivity weighting factor for the radiative transfer equation of the non-gray P-1 model and the non-gray DO model	DEFINE_EMISSIVITY_ WEIGHTING_FACTOR	**User-Defined Function Hooks**
variation of scalar dissipation, mean mixture fraction grid, and mean progress variable grid for flamelet generation	DEFINE_FLAMELET_PARA- METERS	**Species Model**
cell zone motion components in a moving reference frame, moving mesh, or solid motion simulation	DEFINE_ZONE_MOTION	cell zone condition
gray band absorption coefficient for DO model	DEFINE_GRAY_BAND_ABS_ COEFF	**Create/Edit Materials**
weighted-sum-of-gray-gases model (WSGGM) absorption coefficient	DEFINE_WSGGM_ABS_COEFF	**Create/Edit Materials**
soot absorption coefficient	DEFINE_WSGGM_ABS_COEFF	**Create/Edit Materials**
wall heat flux	DEFINE_HEAT_FLUX	**User-Defined Function Hooks**
ignition time source	DEFINE_IGNITE_SOURCE	**User-Defined Function Hooks**
inputs (such as heat transfer coefficients, heat flux augmentation/suppression factors, reference velocity, and temperature) for the semi-mechanistic boiling model	DEFINE_MASS_TR_PROP- ERTY	**User-Defined Function Hooks**

Function	**DEFINE Macro**	Dialog Box Activated In
homogeneous net mass reaction rate for all species	`DEFINE_NET_REAC-` `TION_RATE`	**User-Defined Function Hooks**

Table 2.3: Quick Reference Guide for Model-Specific DEFINE Functions–Continued

Function	**DEFINE Macro**	Dialog Box Activated In
NOx formation rates (for Thermal NOx, Prompt NOx, Fuel NOx, and N_2O Pathways) and upper limit for temperature PDF	`DEFINE_NOX_RATE`	**NOx Model**
discharge coefficient in a perforated wall	`DEFINE_PERFORATED_CD`	**User-Defined Function Hooks**
PDF lookup table	`DEFINE_PDF_TABLE`	**User-Defined Function Hooks**
particle surface reaction rate	`DEFINE_PR_RATE`	**User-Defined Function Hooks**
Prandtl numbers	`DEFINE_PRANDTL`	**Viscous Model**
species mass fraction	`DEFINE_PROFILE`	boundary condition (for example, **Velocity Inlet**)
velocity at a boundary	`DEFINE_PROFILE`	boundary condition
pressure at a boundary	`DEFINE_PROFILE`	boundary condition
temperature at a boundary	`DEFINE_PROFILE`	boundary condition
mass flux at a boundary	`DEFINE_PROFILE`	boundary condition
target mass flow rate for pressure outlet	`DEFINE_PROFILE`	**Pressure Outlet**
turbulence kinetic energy	`DEFINE_PROFILE`	boundary condition (for example, **Velocity Inlet**)
turbulence dissipation rate	`DEFINE_PROFILE`	boundary condition
specific dissipation rate	`DEFINE_PROFILE`	boundary condition
porosity	`DEFINE_PROFILE`	boundary condition
viscous resistance	`DEFINE_PROFILE`	boundary condition
inertial resistance	`DEFINE_PROFILE`	boundary condition
porous resistance direction vector	`DEFINE_PROFILE`	boundary condition
user-defined scalar boundary value	`DEFINE_PROFILE`	boundary condition

Function	**DEFINE** Macro	Dialog Box Activated In
internal emissivity	DEFINE_PROFILE	boundary condition

Table 2.4: Quick Reference Guide for Model-Specific DEFINE Functions–Continued

Function	**DEFINE** Macro	Dialog Box Activated In
wall thermal conditions (heat flux, heat generation rate, temperature, heat transfer coefficient, external emissivity, external radiation and free stream temperature)	DEFINE_PROFILE	boundary condition
shell layer heat generation rate	DEFINE_PROFILE	**Shell Conduction Layers**
wall radiation (internal emissivity, irradiation)	DEFINE_PROFILE	boundary condition
wall momentum (shear stress x, y, z components swirl component, moving wall velocity components, roughness height, roughness constant)	DEFINE_PROFILE	boundary condition
wall species mass fractions	DEFINE_PROFILE	boundary condition
wall user-defined scalar boundary value	DEFINE_PROFILE	boundary condition
wall discrete phase boundary value	DEFINE_PROFILE	boundary condition
density (as function of temperature)	DEFINE_PROPERTY	**Create/Edit Materials**
density (as function of pressure for compressible liquids)	DEFINE_PROPERTY	**Create/Edit Materials**
viscosity	DEFINE_PROPERTY	**Create/Edit Materials**
mass diffusivity	DEFINE_PROPERTY	**Create/Edit Materials**
thermal conductivity	DEFINE_PROPERTY	**Create/Edit Materials**
thermal diffusion coefficient	DEFINE_PROPERTY	**Create/Edit Materials**

Table 2.5: Quick Reference Guide for Model-Specific DEFINE Functions–Continued

Function	**DEFINE** Macro	Dialog Box Activated In
absorption coefficient	DEFINE_PROPERTY	**Create/Edit Materials**
scattering coefficient	DEFINE_PROPERTY	**Create/Edit Materials**
laminar flame speed	DEFINE_PROPERTY	**Create/Edit Materials**
rate of strain	DEFINE_PROPERTY	**Create/Edit Materials**

Function	**`DEFINE` Macro**	**Dialog Box Activated In**
speed of sound function	`DEFINE_PROPERTY`	**Create/Edit Materials**
user-defined mixing law for mixture materials (density viscosity, thermal conductivity)	`DEFINE_PROPERTY`	**Create/Edit Materials**
reacting channel inlet boundary conditions	`DEFINE_REACTING_CHAN-NEL_BC`	**Reacting Channel Model**
reacting channel solver	`DEFINE_REACTING_CHAN-NEL_SOLVER`	**User-Defined Function Hooks**
species characteristic time for reaching chemical equilibrium	`DEFINE_RELAX_TO_EQUI-LIBRIUM`	**User-Defined Function Hooks**
blending function for the Stress-Blended Eddy Simulation (SBES) model	`DEFINE_SBES_BF`	**Viscous Model**
scattering phase function	`DEFINE_SCAT_PHASE_FUNC`	**Create/Edit Materials**
solar intensity	`DEFINE_SOLAR_INTENSITY`	**Radiation Model**
back diffusion	`DEFINE_SOLIDIFICA-TION_PARAMS`	**Solidification and Melting**
mushy zone	`DEFINE_SOLIDIFICA-TION_PARAMS`	**Solidification and Melting**
soot nucleation, surface growth, and oxidation rates for soot mass fraction equation	`DEFINE_SOOT_MASS_RATES`	**Soot Model**
soot nucleation, surface growth, and oxidation rates for soot moment equations	`DEFINE_SOOT_MOM_RATES`	**Soot Model**
soot nucleation and coagulation rates for soot nuclei equation	`DEFINE_SOOT_NUCLE-ATION_RATES`	**Soot Model**
soot oxidation rate	`DEFINE_SOOT_OXIDA-TION_RATE`	**Soot Model**
soot precursor	`DEFINE_SOOT_PRECURSOR`	**Soot Model**
mass source	`DEFINE_SOURCE`	cell zone condition
momentum source	`DEFINE_SOURCE`	cell zone condition
energy source	`DEFINE_SOURCE`	cell zone condition
turbulence kinetic energy source	`DEFINE_SOURCE`	cell zone condition
turbulence dissipation rate source	`DEFINE_SOURCE`	cell zone condition
species mass fraction source	`DEFINE_SOURCE`	cell zone condition
user-defined scalar source	`DEFINE_SOURCE`	cell zone condition

Function	**DEFINE** Macro	Dialog Box Activated In
P1 radiation model source	DEFINE_SOURCE	cell zone condition
spark kernel volume shape	DEFINE_SPARK_GEOM (R14.5 spark model only)	**Set Spark Ignition**
specific heat and sensible enthalpy	DEFINE_SPECIFIC_HEAT	**Create/Edit Materials**
surface reaction rate	DEFINE_SR_RATE	**User-Defined Function Hooks**
thickened flame model	DEFINE_THICKENED_FLAME_ MODEL	**User-Defined Function Hooks**
transition correlation numbers	DEFINE_TRANS	**Viscous Model**
time-varying profiles	DEFINE_TRANSIENT_PRO- FILE	cell zone condition
turbulent premixed source	DEFINE_TURB_PREMIX_ SOURCE	**User-Defined Function Hooks**
turbulent Schmidt number	DEFINE_TURB_SCHMIDT	**Viscous Model**
turbulent viscosity	DEFINE_TURBULENT_ VIS- COSITY	**Viscous Model**
volume reaction rate	DEFINE_VR_RATE	**User-Defined Function Hooks**
wall function	DEFINE_WALL_FUNCTIONS	**Viscous Model**
wall nodal force	DEFINE_WALL_NOD- AL_FORCE	**Wall**
wall nodal displacement	DEFINE_WALL_NODAL_DISP	**Wall**

Table 2.6: Quick Reference Guide for Model-Specific DEFINE Functions MULTIPHASE ONLY

Function	**DEFINE** Macro	Dialog Box Activated In
volume fraction (all multiphase models)	DEFINE_PROFILE	boundary condition
contact angle (VOF)	DEFINE_PROFILE	**Wall** boundary condition
heat transfer coefficient (Eulerian)	DEFINE_PROPERTY	**Phase Interaction**
surface tension coefficient (VOF)	DEFINE_PROPERTY	**Phase Interaction**
cavitation surface tension coefficient (Mixture)	DEFINE_PROPERTY	**Phase Interaction**
cavitation vaporization pressure (Mixture)	DEFINE_PROPERTY	**Phase Interaction**

Function	DEFINE Macro	Dialog Box Activated In
particle or droplet diameter (Mixture)	DEFINE_PROPERTY	**Create/Edit Materials**
diameter (Eulerian, Mixture)	DEFINE_PROPERTY	**Secondary Phase**
solids pressure (Eulerian, Mixture)	DEFINE_PROPERTY	**Secondary Phase**
radial distribution (Eulerian, Mixture)	DEFINE_PROPERTY	**Secondary Phase**
elasticity modulus (Eulerian, Mixture)	DEFINE_PROPERTY	**Secondary Phase**
viscosity (Eulerian, Mixture)	DEFINE_PROPERTY	**Secondary Phase**
temperature (Eulerian, Mixture)	DEFINE_PROPERTY	**Secondary Phase**
bulk viscosity (Eulerian)	DEFINE_PROPERTY	**Secondary Phase**
frictional viscosity (Eulerian)	DEFINE_PROPERTY	**Secondary Phase**
frictional pressure (Eulerian)	DEFINE_PROPERTY	**Secondary Phase**
frictional modulus (Eulerian)	DEFINE_PROPERTY	**Secondary Phase**
granular viscosity (Eulerian)	DEFINE_PROPERTY	**Secondary Phase**
granular bulk viscosity (Eulerian)	DEFINE_PROPERTY	**Secondary Phase**
granular conductivity (Eulerian)	DEFINE_PROPERTY	**Secondary Phase**
temperature source (Eulerian, Mixture)	DEFINE_SOURCE	boundary condition

2.3.1. `DEFINE_ANISOTROPIC_CONDUCTIVITY`

2.3.1.1. Description

You can use `DEFINE_ANISOTROPIC_CONDUCTIVITY` to specify the conductivity matrix in a solid, in order to simulate anisotropic thermal conductivity. For details about user-defined anisotropic conductivity, see User-Defined Anisotropic Thermal Conductivity in the User's Guide.

2.3.1.2. Usage

`DEFINE_ANISOTROPIC_CONDUCTIVITY (name, c, t, dmatrix)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *t`	Pointer to the cell thread on which the anisotropic conductivity is to be applied.
`Real dmatrix[ND_ND] [ND_ND]`	Anisotropic conductivity matrix to be filled in by the user.

Function returns

```
void
```

There are four arguments to DEFINE_ANISOTROPIC_CONDUCTIVITY: name, c, t, and dmatrix. You supply name, the name of the UDF. c, t, and dmatrix are variables that are passed by the Ansys Fluent solver to your UDF. Your function will compute the conductivity matrix for a single cell and fill dmatrix with it. Note that anisotropic conductivity UDFs are called by Ansys Fluent from within a loop on cell threads. Consequently, your UDF will not need to loop over cells in a thread since Ansys Fluent is doing it outside of the function call.

2.3.1.3. Example

The following UDF, named cyl_ortho_cond computes the anisotropic conductivity matrix for a cylindrical shell that has different conductivities in radial, tangential, and axial directions. This function can be executed as a compiled UDF.

```
/****************************************************************************
 UDF for defining the anisotropic conductivity matrix for a cylindrical shell
 ****************************************************************************/
#include "udf.h"

/* Computation of anisotropic conductivity matrix for
* cylindrical orthotropic conductivity */

/* axis definition for cylindrical conductivity */
static const real origin[3] = {0.0, 0.0, 0.0};
static const real axis[3]   = {0.0, 0.0, 1.0};

/* conductivities in radial, tangential and axial directions */
static const real cond[3] = {1.0, 0.01, 0.01};

DEFINE_ANISOTROPIC_CONDUCTIVITY(cyl_ortho_cond,c,t,dmatrix)
{
   real x[3][3]; /* principal direction matrix for cell in cartesian coords. */
   real xcent[ND_ND];
   real R;
   C_CENTROID(xcent,c,t);
  NV_VV(x[0],=,xcent,-,origin);
 #if RP_3D
    NV_V(x[2],=,axis);
 #endif
 #if RP_3D
    R = NV_DOT(x[0],x[2]);
    NV_VS(x[0],-=,x[2],*,R);
 #endif
  R = NV_MAG(x[0]);
  if (R > 0.0)
     NV_S(x[0],/=,R);
 #if RP_3D
  N3V_CROSS(x[1],x[2],x[0]);
 #else
    x[1][0] = -x[0][1];
    x[1][1] = x[0][0];
 #endif
  /* dmatrix is computed as xT*cond*x */
  dmatrix[0][0] = cond[0]*x[0][0]*x[0][0]
     + cond[1]*x[1][0]*x[1][0]
 #if RP_3D
   + cond[2]*x[2][0]*x[2][0]
 #endif
  ;
  dmatrix[1][1] = cond[0]*x[0][1]*x[0][1]
    + cond[1]*x[1][1]*x[1][1]
 #if RP_3D
```

```
       + cond[2]*x[2][1]*x[2][1]
    #endif
      ;
     dmatrix[1][0] = cond[0]*x[0][1]*x[0][0]
       + cond[1]*x[1][1]*x[1][0]
    #if RP_3D
       + cond[2]*x[2][1]*x[2][0]
    #endif
    ;
     dmatrix[0][1] = dmatrix[1][0];

    #if RP_3D
     dmatrix[2][2] = cond[0]*x[0][2]*x[0][2]
       + cond[1]*x[1][2]*x[1][2]
       + cond[2]*x[2][2]*x[2][2]
       ;
     dmatrix[0][2] = cond[0]*x[0][0]*x[0][2]
       + cond[1]*x[1][0]*x[1][2]
       + cond[2]*x[2][0]*x[2][2]
       ;
     dmatrix[2][0] = dmatrix[0][2];
    dmatrix[1][2] = cond[0]*x[0][1]*x[0][2]
       + cond[1]*x[1][1]*x[1][2]
       + cond[2]*x[2][1]*x[2][2]
       ;
     dmatrix[2][1] = dmatrix[1][2];
    #endif
  }
```

2.3.1.4. Hooking an Anisotropic Conductivity UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_ANISOTROPIC_CONDUCTIVITY` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `cyl_ortho_cond`) will become visible and selectable in the **Create/Edit Materials** dialog box in Ansys Fluent. See Hooking `DEFINE_ANISOTROPIC_CONDUCTIVITY` UDFs (p. 436) for details.

2.3.2. `DEFINE_CHEM_STEP`

2.3.2.1. Description

You can use `DEFINE_CHEM_STEP` to specify the change in mass fraction due to homogeneous reaction over a time step:

$$Y_i^{\Delta t} = Y_i^0 + \int_0^{\Delta t} \frac{dY_i}{dt} dt \tag{2.2}$$

where Y_i^0 is the initial mass fraction of species i, t is time, Δt is the given time step, and $\frac{dY_i}{dt}$ is the net rate of change of the i [th] species mass fraction. $Y_i^{\Delta t}$ is i [th] species mass fraction at the end of the integration.

`DEFINE_CHEM_STEP` UDFs are used for the laminar finite-rate (with the stiff chemistry solver), EDC and PDF Transport models.

2.3.2.2. Usage

`DEFINE_CHEM_STEP (name, c, t, p, num_p, n_spe, dt, pres, temp, yk)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index of current particle.
`Thread *t`	Pointer to cell thread for particle.
`Particle *p`	Pointer to `Particle` data structure that represents a particle used by the PDF transport model.
`int num_p`	Not Used.
`int n_spec`	Number of volumetric species.
`double *dt`	Time step.
`double *pres`	Pointer to pressure.
`double *temp`	Pointer to temperature.
`double *yk`	Pointer to array of initial species mass fractions.

Function returns

`void`

There are nine arguments to `DEFINE_CHEM_STEP`: `name`, `c`, `p`, `num_p`, `n_spe`, `dt`, `pres`, `temp`, and `yk`. You supply `name`, the name of the UDF. `c`, `p`, `n_spe`, `dt`, `pres`, `temp`, and `yk` are variables that are passed by the Ansys Fluent solver to your UDF. `num_p` is not used by the function and can be ignored. The output of the function is the array of mass fractions `yk` *after* the integration step. The initial mass fractions in array `yk` are overwritten.

2.3.2.3. Example

The following UDF, named `user_chem_step`, assumes that the net volumetric reaction rate is the expression,

$$\frac{dY_k}{dt} = 1/N_{spe} - Y_k \tag{2.3}$$

where N_{spe} is the number of species.

An analytic solution exists for the integral of this ODE as,

$$Y_k^{\Delta t} = \left(Y_k^0 - 1/N_{spe}\right)exp\left(-\Delta t\right) + 1/N_{spe} \tag{2.4}$$

```
/***************************************************
   Example UDF that demonstrates DEFINE_CHEM_STEP
   **************************************************/
#include "udf.h"

DEFINE_CHEM_STEP(user_chem_step,cell,thread,particle,nump,nspe,dt,pres,temp,yk)
{
  int i;
  double c = 1./(double)nspe;
  double decay = exp(-(*dt));
```

```
    for(i=0;i<n_spe;i++)
        yk[i] = (yk[i]-c)*decay + c;
}
```

2.3.2.4. Hooking a Chemistry Step UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_CHEM_STEP is interpreted (Interpreting UD-Fs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first DEFINE macro argument (for example, user_chem_step) will become visible and select-able in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking DEFINE_CHEM_STEP UDFs (p. 438) for details.

2.3.3. DEFINE_CPHI

2.3.3.1. Description

You can use DEFINE_CPHI to set the value of the mixing constant C_ϕ (see Equation 7.120 and Equation 7.122 in the Theory Guide for details). It is useful for modeling flows where C_ϕ departs substantially from its default value of 2, which occurs at low Reynolds and/or high Schmidt numbers.

2.3.3.2. Usage

DEFINE_CPHI (name, c, t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread.

Function returns

real

There are three arguments to DEFINE_CPHI: name, c, and t. You supply name, the name of the UDF. c and t are passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the real value of the mixing constant (C_ϕ) and return it to the solver.

2.3.3.3. Hooking a Mixing Constant UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_CPHI is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first DEFINE macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent whenever the Composition PDF Transport model is enabled. See Hooking DEFINE_CPHI UDFs (p. 439) for details.

2.3.4. `DEFINE_CORNER_FLOW_CORRECTION_CCORNER`

2.3.4.1. Description

You can use **`DEFINE_CORNER_FLOW_CORRECTION_CCORNER`** to specify a custom coefficient for C_{corner} which is used in the corner flow correction term to influence the strength of the corner flow correction if needed for a specific flow.

2.3.4.2. Usage

DEFINE_CORNER_FLOW_CORRECTION_CCORNER (name, c, t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Index that identifies the cell on which the corner flow correction coefficient C_{corner} is to be applied.
Thread *t	Pointer to cell thread.

Function returns

real

There are three arguments to these functions: **`name`**, **`c`**, and **`t`**. You specify the name of the UDF. **`c`** and **`t`** are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF returns the real value for the corner flow correction coefficient C_{corner}.

2.3.4.3. Example

In the following example, a zonal approach is demonstrated. Depending on the x-coordinate, the corner flow correction coefficient C_{corner} has different values.

```
#include "udf.h"
DEFINE_CORNER_FLOW_CORRECTION_CCORNER(corner_flow_ccorner, c, t)
 {
    real corner_flow_correction_ccorner;
    real xc[ND_ND];
    C_CENTROID(xc,c,t);
    if (xc[0] > 2.0)
      corner_flow_correction_ccorner = 1.1;
    else
      corner_flow_correction_ccorner = 1.0;
    return corner_flow_correction_ccorner;
 }
```

2.3.4.4. Hooking a UDF for Corner Flow Correction Coefficient to Ansys Fluent

After the UDF that you have defined using DEFINE_CORNER_FLOW_CORRECTION_CCORNER is interpreted or compiled, the name of the argument that you supplied as the DEFINE macro argument (for example, corner_flow_ccorner) will become visible and selectable in the Viscous Model dialog box in the drop-down menu of the model coefficient.

2.3.5. `DEFINE_CURVATURE_CORRECTION_CCURV`

2.3.5.1. Description

You can use `DEFINE_CURVATURE_CORRECTION_CCURV` to specify a custom function for the coefficient C_{curv} which is used in the curvature correction term to influence the strength of the curvature correction, if needed for a specific flow. For more information, see Curvature Correction for the Spalart-Allmaras and Two-Equation Models in the *Fluent Theory Guide*.

The coefficient should be positive and in the case of a user defined function, the provided value of C_{curv} is automatically limited by max(C_{curv},0).

2.3.5.2. Usage

`DEFINE_CURVATURE_CORRECTION_CCURV` (name, c, t)

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index that identifies the cell on which the curvature correction coefficient C_{curv} is to be applied.
`Thread *t`	Pointer to cell thread.

Function returns

`real`

There are 3 arguments for `DEFINE_CURVATURE_CORRECTION_CCURV`: `name`, `c`, and `t`. You supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the real value for the curvature correction coefficient C_{curv}.

2.3.5.3. Example

In the following example, a zonal approach is demonstrated: Depending on the x-coordinate, the curvature correction coefficient C_{curv} has different values.

```
#include "udf.h"
DEFINE_CURVATURE_CORRECTION_CCURV(user_curvcor_ccurv, c, t)
{
  real curvature_correction_ccurv;
  real xc[ND_ND];
  C_CENTROID(xc,c,t);
  if (xc[0] > 2.0)
    curvature_correction_ccurv = 1.1;
  else
    curvature_correction_ccurv = 1.0;
  return curvature_correction_ccurv;
}
```

2.3.5.4. Hooking a UDF for Curvature Correction Coefficient to Ansys Fluent

After the UDF that you have defined using `DEFINE_CURVATURE_CORRECTION_CCURV` is inter-
preted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name that you specified
in the `DEFINE` macro argument (for example, `user_curvcor_ccurv`) will become visible and
selectable in the **Viscous Model** dialog box in the drop-down menu for the curvature correction
coefficient.

2.3.6. `DEFINE_DIFFUSIVITY`

2.3.6.1. Description

You can use `DEFINE_DIFFUSIVITY` to specify the diffusivity for the species transport equations
(for example, mass diffusivity) or for user-defined scalar (UDS) transport equations. For details about
UDS diffusivity, see User-Defined Scalar (UDS) Diffusivity in the User's Guide.

2.3.6.2. Usage

`DEFINE_DIFFUSIVITY (name, c, t, i)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *t`	Pointer to cell thread on which the diffusivity function is to be applied.
`int i`	Index that identifies the species or user-defined scalar.

Function returns

`real`

There are four arguments to `DEFINE_DIFFUSIVITY`: `name`, `c`, `t`, and `i`. You supply `name`, the
name of the UDF. `c`, `t`, and `i` are variables that are passed by the Ansys Fluent solver to your UDF.
Your UDF will need to compute the diffusivity *only* for a single cell and return the `real` value to
the solver.

Note that diffusivity UDFs are called by Ansys Fluent from within a loop on cell threads. Consequently,
your UDF will not need to loop over cells in a thread since Ansys Fluent is doing it outside of the
function call.

2.3.6.3. Example

The following UDF, named `mean_age_diff`, computes the diffusivity for the mean age of air using
a user-defined scalar. Note that the mean age of air calculations do not require that energy, radiation,
or species transport calculations have been performed. You will need to set `uds-0 = 0.0` at all inlets
and outlets in your model. This function can be executed as an interpreted or compiled UDF.

```
/***********************************************************************
   UDF that computes diffusivity for mean age using a user-defined
```

```
    scalar.
*********************************************************************/

#include "udf.h"

DEFINE_DIFFUSIVITY(mean_age_diff,c,t,i)
{
    return C_R(c,t) * 2.88e-05 + C_MU_EFF(c,t) / 0.7;
}
```

2.3.6.4. Hooking a Diffusivity UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DIFFUSIVITY` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name that you specified in the `DEFINE` macro argument (for example, `mean_age_diff`) will become visible and selectable in the **Create/Edit Materials** dialog box in Ansys Fluent. See Hooking `DEFINE_DIFFUSIVITY` UDFs (p. 442) for details.

2.3.7. `DEFINE_DOM_DIFFUSE_REFLECTIVITY`

2.3.7.1. Description

You can use `DEFINE_DOM_DIFFUSE_REFLECTIVITY` to modify the inter-facial reflectivity computed by Ansys Fluent at diffusely reflecting semi-transparent walls, based on the refractive index values. During execution, a `DEFINE_DOM_DIFFUSE_REFLECTIVITY` function is called by Ansys Fluent for each semi-transparent wall and also for each band (in the case of a non-gray discrete ordinates (DO) model). Therefore the function can be used to modify diffuse reflectivity and diffuse transmissivity values at the interface.

2.3.7.2. Usage

`DEFINE_DOM_DIFFUSE_REFLECTIVITY` (name, t, nb, n_a, n_b, diff_ref_a, diff_tran_a, diff_ref_b, diff_tran_b)

Important:

Note that all of the arguments to a `DEFINE` macro need to be placed on the same line in your source code. Splitting the `DEFINE` statement onto several lines will result in a compilation error.

Argument Type	Description
`symbol name`	UDF name.
`Thread *t`	Pointer to the thread on which the discrete ordinate diffusivity function is to be applied.
`int nb`	Band number (needed for the non-gray discrete ordinates (DO) model).
`real n_a`	Refractive index of medium a.
`real n_b`	Refractive index of medium b.

Argument Type	Description
`real *diff_ref_a`	Diffuse reflectivity at the interface facing medium a.
`real *diff_tran_a`	Diffuse transmissivity at the interface facing medium a.
`real *diff_ref_b`	Diffuse reflectivity at the interface facing medium b.
`real *diff_tran_b`	Diffuse transmissivity at the interface facing medium b.

Function returns

`void`

There are nine arguments to `DEFINE_DOM_DIFFUSE_REFLECTIVITY`: `name`, `t`, `nb`, `n_a`, `n_b`, `diff_ref_a`, `diff_tran_a`, `diff_ref_b`, and `diff_tran_b`. You supply `name`, the name of the UDF. `t`, `nb`, `n_a`, `n_b`, `diff_ref_a`, `diff_tran_a`, `diff_ref_b`, and `diff_tran_b` are variables that are passed by the Ansys Fluent solver to your UDF.

2.3.7.3. Example

The following UDF, named `user_dom_diff_refl`, modifies diffuse reflectivity and transmissivity values on both the sides of the interface separating medium a and b. The UDF is called for all the semi-transparent walls and prints the value of the diffuse reflectivity and transmissivity values for side a and b.

> **Important:**
>
> Note that in the example that follows, the `DEFINE_DOM_DIFFUSE_REFLECTIVITY` statement is broken up into two lines for the sake of readability. In your source file, you must make sure that the `DEFINE` statement is on one line only.

```
/* UDF to print the diffuse reflectivity and transmissivity
 at semi-transparent walls*/

#include "udf.h"

DEFINE_DOM_DIFFUSE_REFLECTIVITY(user_dom_diff_refl,t,nband,n_a,n_b,diff_ref_a,diff_tran_a,diff_ref_b,diff_tran
{
  printf("diff_ref_a=%f diff_tran_a=%f \n", *diff_ref_a, *diff_tran_a);
  printf("diff_ref_b=%f diff_tran_b=%f \n", *diff_ref_b, *diff_tran_b);
}
```

2.3.7.4. Hooking a Discrete Ordinates Model (DOM) Diffuse Reflectivity UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DOM_DIFFUSE_REFLECTIVITY` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `user_dom_diff_refl`) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent.

2.3.8. DEFINE_DOM_SOURCE

2.3.8.1. Description

You can use DEFINE_DOM_SOURCE to modify the emission term, which is the first term on the right hand side in Equation 5.74 or Equation 5.75 in the Theory Guide, as well as the scattering term (second term on the right hand side of either equation) in the radiative transport equation for the discrete ordinates (DO) model.

2.3.8.2. Usage

DEFINE_DOM_SOURCE (name, c, t, ni, nb, emission, in_scattering, abs_coeff, scat_coeff)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread.
int ni	Direction represented by the solid angle.
int nb	Band number (needed for the non-gray discrete ordinates (DO) model).
real *emission	Pointer to emission term in the radiative transport equation (Equation 5.74 in the Theory Guide)
real *in_scattering	Pointer to scattering term in the radiative transport equation (Equation 5.75 in the Theory Guide)
real *abs_coeff	Pointer to absorption coefficient.
real *scat_coeff	Pointer to scattering coefficient.

Function returns

void

There are nine arguments to DEFINE_DOM_SOURCE: name, t, c, ni, nb, emission, in_scattering, abs_coeff, and scat_coeff. You supply name, the name of the UDF. c, ni, nb, emission, in_scattering, abs_coeff, and scat_coeff are variables that are passed by the Ansys Fluent solver to your UDF. DEFINE_DOM_SOURCE is called by Ansys Fluent for each cell.

2.3.8.3. Example

In the following UDF, named dom, the emission term present in the radiative transport equation is modified. The UDF is called for all the cells and increases the emission term by 5%.

```
/* UDF to alter the emission source term in the DO model */

 #include "udf.h"
```

```
DEFINE_DOM_SOURCE(dom,c,t,ni,nb,emission,in_scattering,abs_coeff,scat_coeff)
{
  /* increased the emission by 5% */

  *emission *= 1.05;

}
```

2.3.8.4. Hooking a DOM Source UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DOM_SOURCE` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `dom`) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. Note that you can hook multiple discrete ordinate source term functions to your model. See Hooking `DEFINE_DOM_SOURCE` UDFs (p. 446) for details.

2.3.9. `DEFINE_DOM_SPECULAR_REFLECTIVITY`

2.3.9.1. Description

You can use `DEFINE_DOM_SPECULAR_REFLECTIVITY` to modify the inter-facial reflectivity of specularly reflecting semi-transparent walls. You may want to do this if the reflectivity is dependent on other conditions that the standard boundary condition does not allow for (see Specular Semi-Transparent Walls of the Theory Guide for more information). During Ansys Fluent execution, the same UDF is called for all the faces of the semi-transparent wall, for each of the directions.

2.3.9.2. Usage

`DEFINE_DOM_SPECULAR_REFLECTIVITY` (name, f, t, nband, n_a, n_b, ray_direction, en, internal_reflection, specular_reflectivity, specular_transmissivity)

Important:

Note that all of the arguments to a `DEFINE` macro need to be placed on the same line in your source code. Splitting the `DEFINE` statement onto several lines will result in a compilation error.

Argument Type	Description
`symbol name`	UDF name.
`face_t f`	Face index.
`Thread *t`	Pointer to face thread on which the specular reflectivity function is to be applied.
`int nband`	Band number (needed for non-gray discrete ordinates (DO) model).
`real n_a`	Refractive index of medium a.
`real n_b`	Refractive index of medium b.

Argument Type	Description
`real ray_direction`	Direction vector (s) defined in Equation 5.92 in the Theory Guide.
`real en`	Interface normal vector (n) defined in Equation 5.92 in the Theory Guide.
`int internal_reflection`	Variable used to flag the code that total internal reflection has occurred.
`real *specular_reflectivity`	Specular reflectivity for the given direction s.
`real *specular_transmissivity`	Specular transmissivity for the given direction s.

Function returns

`void`

There are eleven arguments to `DEFINE_DOM_SPECULAR_REFLECTIVITY`: name, f, t, nband, n_a, n_b, ray_direction, en, internal_reflection, specular_reflectivity, and specular_transmissivity. You supply `name`, the name of the UDF. `f`, `t`, `nband`, `n_a`, `n_b`, `ray_direction`, `en`, `internal_reflection`, `specular_reflectivity`, and `specular_transmissivity` are variables that are passed by the Ansys Fluent solver to your UDF.

2.3.9.3. Example

In the following UDF, named `user_dom_spec_refl`, specular reflectivity and transmissivity values are altered for a given ray direction s at face `f`.

Important:

Note that in the example that follows, the `DEFINE_DOM_SPECULAR_REFLECTIVITY` statement is broken up into three lines for the sake of readability. In your source file, you must make sure that the `DEFINE` statement is on one line only.

```
/* UDF to alter the specular reflectivity and transmissivity, at
   semi-transparent walls, along direction s at face f */

#include "udf.h"

DEFINE_DOM_SPECULAR_REFLECTIVITY(user_dom_spec_refl,f,t,nband,n_a,n_b,ray_direction,en,internal_reflection,spe
 {
   real angle, cos_theta;
   real PI = 3.141592;
   cos_theta = NV_DOT(ray_direction, en);
   angle = acos(cos_theta);
   if (angle > 0.785398  && angle < 1.047198)
      {
        *specular_reflectivity = 0.3;
        *specular_transmissivity = 0.7;
      }
 }
```

2.3.9.4. Hooking a Discrete Ordinates Model (DOM) Specular Reflectivity UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_DOM_SPECULAR_REFLECTIVITY is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first DEFINE macro argument (for example, user_dom_spec_refl) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent.

See Hooking DEFINE_DOM_SPECULAR_REFLECTIVITY UDFs (p. 447) for details.

2.3.10. DEFINE_EC_KINETICS_PARAMETER

2.3.10.1. Description

You can use DEFINE_EC_KINETICS_PARAMETER to customize any of the kinetics parameters (anodic and cathodic transfer coefficient, exchange current density, and equilibrium potential) in the Butler-Volmer equation. The UDF can be hooked up from the **Reaction** dialog box.

> **Important:**
>
> DEFINE_EC_KINETICS_PARAMETER UDFs can be executed only as compiled UDFs.

2.3.10.2. Usage

DEFINE_EC_KINETICS_PARAMETER(name, f, fthread)

Argument Type	Description
symbol name	UDF name.
face_t f	Index that identifies a face within the given thread.
Thread *fthread	Pointer to face thread on which the surface rate reaction is to be applied

Function returns

double

There are three arguments to DEFINE_EC_KINETICS_PARAMETER: name, f, and fthread. You supply name, the name of the UDF. f and fthread are variables that are passed by the Ansys Fluent solver to your UDF. After your UDF is compiled and linked, the name that you have chosen for your function will become visible and selectable in the graphical user interface in Ansys Fluent. Your UDF will return a double value to the kinetics parameter which you have hooked up to.

2.3.10.3. Example - Electrochemical Reaction Kinetics Parameter Using UDF

The following compiled UDF, named ec_parameter_Eeq, computes temperature-dependent equilibrium potential through the UDF approach. You can use this type of UDF to compute any of the four kinetics parameters used in the electrochemical reaction (anodic transfer coefficient/anodic

Tafel slope, cathodic transfer coefficient/cathodic Tefel slope, exchange current density, and equilibrium potential).

```
/*********************************************************************
Custom electrochemical reaction kinetics parameter UDF
*********************************************************************/
#include "udf.h"
DEFINE_EC_KINETICS_PARAMETER(ec_parameter_Eeq, f, fthread)
{
  real T, Eeq;
  T = F_T(f, fthread);
  Eeq = 0.8 + 0.0001 * T;
  return Eeq;
}
```

2.3.10.4. Hooking an Electrochemical Reaction Kinetics Parameter UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_EC_KINETICS_PARAMETER` is compiled and loaded (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `ec_parameter_Eeq`) will become visible and selectable after you select the **user-defined** option for a kinetics parameter in the Reaction dialog box. See Hooking `DEFINE_EC_KINETICS_PARAMETER` UDFs (p. 451) for details.

2.3.11. `DEFINE_EC_RATE`

2.3.11.1. Description

You can use `DEFINE_EC_RATE` to specify a custom electrochemical reaction rate. A custom electrochemical reaction rate function defined using this macro will overwrite the default reaction rate specified in the **Create/Edit Materials** dialog box.

2.3.11.2. Usage

`DEFINE_EC_RATE (name, f, t, r, V, i, didV, Eeq)`

Argument Type	Description
`symbol name`	UDF name.
`face_t f`	Index that identifies a face within the given thread.
`Thread *t`	Pointer to face thread on which the electrochemical reaction is to be applied.
`Reaction *r`	Pointer to data structure for the reaction.
`double V`	Potential difference between electrode and electrolyte $V=\varphi_{ed}-\varphi_{el}$.
`double *i`	Pointer to the electric current density, i.
`double *didV`	Pointer to the derivative of current density with respect to potential difference V, $\frac{di}{dV}$.
`double *Eeq`	Pointer to the Equilibrium potential

Function returns

```
void
```

There are eight arguments to DEFINE_EC_RATE: name, f, t, r, V, i, didV, and Eeq. You supply name, the name of the UDF. f, t, r, and V are variables that are passed by the Ansys Fluent solver to your UDF. After your UDF is compiled and linked, the name that you have chosen for your function will become visible and selectable in the graphical user interface in Ansys Fluent. Your UDF will need to set three values referenced by the real pointers i didV, and Eeq.

2.3.11.3. Example - Electrochemical Reaction Rate Using UDF

The following compiled UDF, named user_ec_rate, implements the solver default Butler-Volmer equation through the UDF approach. You can customize a term (for example, the exchange current density i0 or the equilibrium potential Eeq) or the whole rate formula.

```
/******************************************************************
 Custom electrochemical reaction rate UDF
 ******************************************************************/
#include "udf.h"
DEFINE_EC_RATE(user_ec_rate, f, fthread, r, V, current, didV, Eeq)
{
  double alpha_a, alpha_c;
  double io;
  double T = F_T(f,fthread);
  double arg1, arg2;
  cxboolean tafelmethod = r->tafelmethod;
  int i;
  double eta;
  io      = update_echem_value(r->Pio, f, fthread);
  alpha_a = update_echem_value(r->Palpha_a, f, fthread);
  alpha_c = update_echem_value(r->Palpha_c, f, fthread);
  *Eeq    = update_echem_value(r->PEeq, f, fthread);

  if (tafelmethod)
  {
    alpha_a = 2.303 * UNIVERSAL_GAS_CONSTANT * 298.15 /(alpha_a * FARADAY_CONSTANT);
    alpha_c = 2.303 * UNIVERSAL_GAS_CONSTANT * 298.15 /(alpha_c * FARADAY_CONSTANT);
  }
  eta = V - *Eeq;
  for(i = 0; i<r->n_reactants; i++)
    if( ABS( r->exp_reactant[i] ) > SMALL_S )
    {
      int ni = r->reactant[i];
      io *= pow((F_YI(f,fthread,ni)/MAX(r->yi_ref[ni],SMALL) + 1.0e-20), r->exp_reactant[i]);
    }
  for(i = 0; i<r->n_products; i++)
    if( ABS( r->exp_product[i] ) > SMALL_S )
    {
      int ni = r->product[i];
      io *= pow((F_YI(f,fthread,ni)/MAX(r->yi_ref[ni],SMALL) + 1.0e-20), r->exp_product[i]);
    }
  arg1 = FARADAY_CONSTANT / (UNIVERSAL_GAS_CONSTANT*T);
  arg2 = arg1*eta;
  *current = io*( exp( arg2*alpha_a ) - exp( -arg2*alpha_c ) );
  *didV = io*( arg1*alpha_a*exp( arg2*alpha_a ) + arg1*alpha_c*exp( -arg2*alpha_c ) );

  /* If multiple electrochemical reactions are used, you can define rate for each reaction
     using the following if-statement */
  /*
  if (STREQ(r->name, "reaction-1"))
  {
    ...
  }
  else if (STREQ(r->name, "reaction-2"))
```

```
    {
    ...
    }
  }
*/
}
```

2.3.11.4. Hooking an Electrochemical Reaction Rate UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_EC_RATE` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `user_ec_rate`) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking `DEFINE_EC_RATE` UDFs (p. 449) for details.

2.3.12. `DEFINE_EDC_MDOT`

2.3.12.1. Description

In Ansys Fluent, the Eddy Dissipation Concept (EDC) model closes the species transport equations according to the terms given in The Eddy-Dissipation-Concept (EDC) Model in the *Fluent Theory Guide*. You can use the `DEFINE_EDC_MDOT` UDF to modify the reaction rate for the EDC model as shown in Equation 7.30 in the *Fluent Theory Guide*. Equation 7.30 can be rewritten as follows:

$$R_i = \frac{\rho \left(\xi^* \right)^2}{\tau^* \left[1 - \left(\xi^* \right)^3 \right]} \left(Y_i^* - Y_i \right) = \frac{\rho \dot{m}}{\tau^*} \left(Y_i^* - Y_i \right) \tag{2.5}$$

This UDF allows you to use your own formulation to calculate the term $\dot{m}$ and the time scale τ^* in Equation 2.5 (p. 69).

2.3.12.2. Usage

`DEFINE_EDC_MDOT` (name, c, t, mdot, calc_tau, tau)

Important:

- Note that all the arguments of a `DEFINE` macro need to be placed on the same line in your source code. Splitting the `DEFINE` statement into several lines will result in a compilation error.

- `DEFINE_EDC_MDOT` functions can be executed only as compiled UDFs.

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell or face index.
`Thread *t`	Pointer to the cell or face thread.
`real *mdot`	Pointer to parameter $\dot{m}$.

Argument Type	Description
`int calc_tau`	Integer that indicates whether `tau` will be calculated (`tau` will be computed if `calc_tau` is set to a non-zero value).
`real *tau`	Pointer to τ^* value.

Function returns

`void`

There are six arguments to `DEFINE_EDC_MDOT`: `name`, `c`, `t`, `mdot`, `calc_tau`, and `tau`. You supply `name`, the name of the UDF. The variables `c`, `t`, `mdot`, `calc_tau`, and `tau` are passed by the Ansys Fluent solver to your UDF. The values of `mdot` and `tau` passed by the Ansys Fluent solver are those used by the solver unless they are reset by this routine. The routine may reset any or all of these values.

The `calc_tau` indicator is provided for efficiency purposes as `DEFINE_EDC_MDOT` will be called several times by the solver but will not always use the `tau` value. Enclosing the `tau` calculation with `if (calc_tau) {}`, as in Example (p. 70), will thus maximize the efficiency of this routine.

2.3.12.3. Example

The following example is a source code template where the `DEFINE_EDC_MDOT` function is used to set the term $\dot{m}$ for the EDC model.

```
/********************************************************************
 Example UDF that demonstrates the use of DEFINE_EDC_MDOT to set the
term mdot in the EDC model.
 ********************************************************************/
 #include <udf.h>

 DEFINE_EDC_MDOT(edc_mdot, c, t, mdot, calc_tau, tau)
 {
  real ted1 = MAX(C_D(c,t), 1.0e-03);
  real ted2 = C_D(c, t);
  real c2 = 0.4083;
  real c1 = 2.1377;
  real gamma2,gamma = c1 * pow( C_MU_L(c, t) * ted2 / (C_R(c, t) * SQR(C_K(c, t))), 0.25);
  gamma = MIN( gamma, 0.754877666248);
  gamma2 = gamma * gamma;
  *mdot = gamma2 / ((1. - gamma2 * gamma));
  if (calc_tau)
     {*tau=c2*sqrt(C_MU_L(c,t)/(ted1*C_R(c,t)));
      *tau = MAX(*tau, 1.e-8);}


 }
```

2.3.12.4. Hooking a `DEFINE_EDC_MDOT` UDF to Ansys Fluent

After you have compiled and loaded (Compiling UDFs (p. 399)) the `DEFINE_EDC_MDOT` UDF and enabled the **Eddy Dissipation Concept** model, the name of the argument that you supplied as the first `DEFINE` macro argument will become visible and selectable in the **Options** group box of the **Species Model** dialog box in Ansys Fluent. See Hooking `DEFINE_DOM_DIFFUSE_REFLECTIV-ITY` UDFs (p. 444) for details.

2.3.13. `DEFINE_EDC_SCALES`

2.3.13.1. Description

In Ansys Fluent, the Eddy Dissipation Concept (EDC) model closes the species transport equations according to the terms given in The Eddy-Dissipation-Concept (EDC) Model in the *Fluent Theory Guide*. This UDF allows you to modify the coefficients C_ξ (Equation 7.28) and C_τ (Equation 7.29) and the term τ^* (Equation 7.29).

2.3.13.2. Usage

`DEFINE_EDC_SCALES (name, c, t, c1, c2, calc_tau, tau)`

> **Important:**
>
> - Note that all the arguments of a `DEFINE` macro need to be placed on the same line in your source code. Splitting the `DEFINE` statement into several lines will result in a compilation error.
>
> - `DEFINE_EDC_SCALES` functions can be executed only as compiled UDFs.

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell or face index.
`Thread *t`	Pointer to the cell or face thread.
`real *c1`	Pointer to coefficient C_ξ.
`real *c2`	Pointer to coefficient C_τ.
`int calc_tau`	Integer that indicates whether `tau` will be calculated (`tau` will be computed if `calc_tau` is set to a non-zero value).
`real *tau`	Pointer to τ^* value.

Function returns

`void`

There are seven arguments to `DEFINE_EDC_SCALES`: `name`, `c`, `t`, `c1`, `c2`, `calc_tau`, and `tau`. You supply `name`, the name of the UDF. The variables `c`, `t`, `c1`, `c2`, `calc_tau`, and `tau` are passed by the Ansys Fluent solver to your UDF. The values of `c1`, `c2` and `tau` passed by the Ansys Fluent solver are those used by the solver unless they are reset by this routine. The routine may reset any or all of these values.

The `calc_tau` indicator is provided for efficiency purposes as `DEFINE_EDC_SCALES` will be called several times by the solver but will not always use the `tau` value. Enclosing the `tau` calculation with `if (calc_tau) {}`, as in Example (p. 72), will thus maximize the efficiency of this routine.

2.3.13.3. Example

The following example is a source code template where the `DEFINE_EDC_SCALES` function is used to set the EDC model scales.

```
/**********************************************************************
 Example UDF that demonstrates the use of DEFINE_EDC_SCALES to set the
EDC model scales.
 *********************************************************************/
 #include <udf.h>

 DEFINE_EDC_SCALES(edc_scales, c, t, c1, c2, calc_tau, tau)
 {
   real ted = MAX(C_D(c,t), 1.0e-03);

   *c1 = 2.1337;
   *c2 = 0.4082;

   if (calc_tau)
     *tau = (*c2)*sqrt(C_MU_L(c,t)/(ted*C_R(c,t)));
 }
```

2.3.13.4. Hooking a `DEFINE_EDC_SCALES` UDF to Ansys Fluent

After you have compiled and loaded (Compiling UDFs (p. 399)) the `DEFINE_EDC_SCALES` UDF and enabled the **Eddy Dissipation Concept** model, the name of the argument that you supplied as the first `DEFINE` macro argument will become visible and selectable in the **Options** group box of the **Species Model** dialog box in Ansys Fluent. See Hooking `DEFINE_EDC_SCALES` UDFs (p. 454) for details.

2.3.14. `DEFINE_EMISSIVITY_WEIGHTING_FACTOR`

2.3.14.1. Description

When employing the non-gray P-1 radiation model or the non-gray discrete ordinates (DO) radiation model, you can use `DEFINE_EMISSIVITY_WEIGHTING_FACTOR` to modify the emissivity weighting factor $F(0 \rightarrow n\lambda_2 T) - F(0 \rightarrow n\lambda_1 T)$. By default, the emissivity weighting factor is calculated internally by Ansys Fluent so it can be used in the emission term of the radiative transfer equation, as shown in Equation 5.40 and Equation 5.76 of the Theory Guide. This macro allows you to revise the calculated value.

2.3.14.2. Usage

`DEFINE_EMISSIVITY_WEIGHTING_FACTOR` (name, c, t, T, nb, `emissivity_weighting_factor`)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread.
real T	Temperature.
int nb	Band number.

| `real *emissivity_weight-ing_factor` | The emissivity weighting factor in the emission term of the radiative transfer equation for the non-gray P-1 model (Equation 5.40 in the Theory Guide) or the non-gray DO model (Equation 5.76 in the Theory Guide). |

Function returns

`void`

There are six arguments to `DEFINE_EMISSIVITY_WEIGHTING_FACTOR`: `name`, `c`, `t`, `T`, `nb`, and `emissivity_weighting_factor`. You supply `name`, the name of the UDF. `c`, `t`, `T`, `nb`, and `emissivity_weighting_factor` are variables that are passed by the Ansys Fluent solver to your UDF. `DEFINE_EMISSIVITY_WEIGHTING_FACTOR` is called by Ansys Fluent for each cell.

2.3.14.3. Example

In the following UDF (named `em_wt`), the emissivity weighting factor present in the emission term of the radiative transfer equation for the non-gray DO model is modified. The UDF is called for all of the cells. It modifies the emissivity weighting factor so that it is no longer the value calculated internally by Ansys Fluent, but is instead changed to 1.

```
/* UDF to alter the emissivity weighting factor for the non-gray DO model */

#include "udf.h"

DEFINE_EMISSIVITY_WEIGHTING_FACTOR(em_wt,c,t,T,nb,emissivity_weighting_factor)
{

  /* revise the calculated emissivity_weighting_factor to be a value of 1 */

  *emissivity_weighting_factor = 1.0;

}
```

2.3.14.4. Hooking an Emissivity Weighting Factor UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_EMISSIVITY_WEIGHTING_FACTOR` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `em_wt`) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking `DEFINE_EMISSIVITY_WEIGHTING_FACTOR` UDFs (p. 456) for details.

2.3.15. `DEFINE_FLAMELET_PARAMETERS`

2.3.15.1. Description

You can use `DEFINE_FLAMELET_PARAMETERS` to specify a user-defined variation of scalar dissipation, mean mixture fraction grid, and mean progress variable grid for flamelet generation with non-premixed or partially premixed combustion models. The UDF is only available if either the **Non-Premixed Combustion** or **Partially-Premixed Combustion** model is enabled.

2.3.15.2. Usage

```
DEFINE_FLAMELET_PARAMETERS (name, Nf, Nc, Ns, xf, xc, xs)
```

Argument Type	Description
symbol name	UDF name.
Nf	Number of mean mixture fraction grid points in flamelet.
Nc	Number of mean progress variable grid points.
Ns	Maximum number of flamelets.
real *xf	Array to provide mean mixture fraction points.
real *xc	Array to provide mean progress variable points.
real *xs	Array to provide scalar dissipation values.

Function returns

```
void
```

There are seven arguments to DEFINE_FLAMELET_PARAMETERS: name, Nf, Nc, Ns, xf, xc, and xs. You supply name, the name of the UDF. Nf, Nc, Ns, xf, xc, and xs are variables that are passed by the Ansys Fluent solver to your UDF. DEFINE_FLAMELET_PARAMETERS does not output a value. The specified grid discretizations or scalar dissipation variations are stored in the arrays xf, xc, and xs, respectively.

> **Note:**
>
> If **Automated Grid Refinement** is enabled for steady diffusion flamelet generation (see Steady Diffusion Flamelet in the *Fluent User's Guide*), then you can only use DEFINE_FLAMELET_PARAMETERS UDFs to specify user-defined scalar dissipation variation. Ansys Fluent will ignore the user-defined grid for the mean mixture fraction and/or reaction progress.

2.3.15.3. Example

The following compiled UDF, named user_scad_fmean_grid, is used to provide the user-defined scalar dissipation variation and user-defined mean mixture fraction grid points for flamelet solution.

```
#include "udf.h"
#define fsto 0.056

DEFINE_FLAMELET_PARAMETERS(user_scad_fmean_grid,nf,nc,ns,xf,xc,xs)
{
    int i,np,np_rich ;

 /* The arrays xf,xc,xs passed by the solver contains default
    discretization and default scalar dissipation variation*/

  /* These arrays can be over-written with user-defined grid or scalar
```

```
       dissipation variation*/

  /* Flamelets with Scalar dissipation of
     0.01, 0.05, 0.1, 0.15, 0.2, 1, 4, 7, 10,... */


    for(i=0;i<ns;i++)
      {
       if (i < 5)
         xs[i] = 0.01*5*i;
       else
         xs[i] = 1.+(i-5)*3;
      }

  /* Sample UDF to provide user defined discretization for fmean grid
     for flamelet generation*/

  /* Divide fmean grid into 3 intervals: (1) f=0 to fsto, (2) fsto to 2.*fsto
     and (3) 2.*fsto to f=1.*/

  /* Place equal number of grid points in these three intervals*/

    if(nf > 0)
      {
       np = (int) nf/3;
       np_rich = 2*np;
      }
    for(i=0;i<nf;i++)
      {
       if(i <= np)
         {
          xf[i] = i*fsto/np;
         }
       else if(i <= np_rich)
         {
          xf[i] = fsto+fsto*(i-np)/(np_rich-np);
         }
       else
         {
          xf[i]= 2.*fsto + (1.-2*fsto)*(i-np_rich)/(nf-1-np_rich);
         }
      }
  }
```

2.3.15.4. Hooking a Flamelet Parameters UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_FLAMELET_PARAMETERS is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first DEFINE macro argument (for example, user_scad_fmean_grid) will become visible and selectable in the **User-Defined Function** drop-down list under the **Flamelet** tab of the **Species Model** dialog box in Ansys Fluent. See Hooking DEFINE_FLAMELET_PARAMETERS UDFs (p. 458) for details.

2.3.16. DEFINE_ZONE_MOTION

2.3.16.1. Description

You can use DEFINE_ZONE_MOTION to specify the cell zone motion components in a moving reference frame, moving mesh, or solid motion simulation. All motion components are passed as arguments to the UDF, so that access to them is provided in a single UDF. The arguments contain the default values already, so that if a specific component (for example, the origin) is not modified by the UDF, it will retain its default value.

2.3.16.2. Usage

`DEFINE_ZONE_MOTION (name, omega, axis, origin, velocity, time, dtime)`

Argument Type	Description
`symbol name`	UDF name.
`real *omega`	Pointer to the rotational velocity magnitude, default 0.
`real axis[3]`	Rotation axis direction vector, default (0 0 1) and (1 0 0) for 2D axisymmetric cases.
`real origin[3]`	Rotation axis origin vector, default (0 0 0).
`real velocity[3]`	Translational velocity vector, default (0 0 0).
`real current_time`	Current time.
`real dtime`	Current time step.

Function returns

`void`

There are seven arguments to `DEFINE_ZONE_MOTION`: `name`, `omega`, `axis`, `origin`, `velocity`, `time`, and `dtime`. You supply `name`, the name of the UDF. The variables `omega`, `axis`, `origin`, `velocity`, `time`, and `dtime` are passed by the Ansys Fluent solver to your UDF and have SI units. Your UDF will need to compute the motion components that you want to modify. If a certain component is not modified by the UDF, it will retain its default value. The vector specified as `axis` does not need to be a unit vector; note that it will be normalized before it is used further inside the solver, that is, definitions such as (1 1 1) and (10 10 10) are equivalent.

All vectors are specified as 3 dimensional, even for 2D simulations. The third component of the origin and the translational velocity vectors will be ignored in 2D cases. For regular 2D cases, rotation is assumed to be around the Z axis. In a 2D axisymmetric case, the rotation is around the X axis. Hence, for 2D cases any modification to the axis vector inside the UDF will be ignored.

2.3.16.3. Example

The following UDF, named `fmotion`, computes the rotation rate of a cell zone, simulating start-up behavior. The angular velocity is increased linearly in time up to a flow time of 0.1 s, after which it remains constant at 250 rad/s. A constant translational velocity of 1 m/s in the X direction is assigned. The lines assigning the origin and axis vectors only repeat the default behavior, and could be omitted. The source can be interpreted or compiled.

```
#include "udf.h"
 DEFINE_ZONE_MOTION(fmotion,omega,axis,origin,velocity,time,dtime)
{
  if (time < 0.1)
    {
     *omega = 2500.0 * time;
    }
  else
    {
     *omega = 250.0;
    }
  N3V_D (velocity,=,1.0,0.0,0.0);
  N3V_S(origin,=,0.0);          /* default values, line could be omitted */
```

```
   N3V_D(axis,=,0.0,0.0,1.0);      /* default values, line could be omitted */

   return;
}
```

2.3.16.4. Hooking a Zone Motion UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_ZONE_MOTION is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first DEFINE macro argument will become visible and selectable in the cell zone condition dialog boxes.

See Hooking DEFINE_ZONE_MOTION UDFs (p. 460) for details.

2.3.17. DEFINE_GRAY_BAND_ABS_COEFF

2.3.17.1. Description

You can use DEFINE_GRAY_BAND_ABS_COEFF to specify a UDF for the gray band absorption coefficient as a function of temperature, that can be used with a non-gray discrete ordinates model.

2.3.17.2. Usage

DEFINE_GRAY_BAND_ABS_COEFF (name, c, t, nb)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to cell thread.
int nb	Band number associated with non-gray model.

Function returns

real

There are four arguments to DEFINE_GRAY_BAND_ABS_COEFF: name, c, t, and nb. You supply name, the name of the UDF. The variables c, t, and nb are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the real value of the gray band coefficient to the solver.

2.3.17.3. Example

The following UDF, named user_gray_band_abs, specifies the gray-band absorption coefficient as a function of temperature that can be used for a non-gray discrete ordinates model.

```
#include "udf.h"

 DEFINE_GRAY_BAND_ABS_COEFF(user_gray_band_abs,c,t,nb)
 {
   real abs_coeff = 0;
```

```
   real T = C_T(c,t);

   switch (nb)
   {
     case 0 : abs_coeff = 1.3+0.001*T; break;
     case 1 : abs_coeff = 2.7 + 0.005*T; break;
   }

 return abs_coeff;

 }
```

2.3.17.4. Hooking a Gray Band Coefficient UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_GRAY_BAND_ABS_COEFF` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `user_gray_band_abs`) will become visible and selectable in the **Create/Edit Materials** dialog box for the **Absorption Coefficient**.

See Hooking `DEFINE_GRAY_BAND_ABS_COEFF` UDFs (p. 462) for details.

2.3.18. `DEFINE_HEAT_FLUX`

2.3.18.1. Description

You can use `DEFINE_HEAT_FLUX` to modify the heat flux at a wall. Despite the name, a `DEFINE_HEAT_FLUX` UDF is *not* the means to specify the actual heat flux entering a domain from the outside. To specify this type of heat flux, you would simply use a `DEFINE_PROFILE` function in conjunction with a heat flux thermal boundary condition. In contrast, a `DEFINE_HEAT_FLUX` UDF allows you to modify the way in which the dependence between the flux entering the domain and the wall and cell temperatures is modeled.

Important:

This function allows you to modify the heat flux at walls adjacent to a solid. Note, however, that for solids since only heat conduction is occurring, any extra heat flux that you add in a heat flux UDF can have a detrimental effect on the solution of the energy equation. These effects will likely show up in conjugate heat transfer problems. To avoid this, you will need to make sure that your heat flux UDF excludes the walls adjacent to solids, or includes only the necessary walls adjacent to fluid zones.

2.3.18.2. Usage

`DEFINE_HEAT_FLUX (name,f,t,c0,t0,cid,cir)`

Argument Type	Description
symbol name	UDF name.
face_t f	Index that identifies a wall face.
Thread *t	Pointer to wall face thread on which heat flux function is to be applied.

Argument Type	Description
`cell_t c0`	Cell index that identifies the cell next to the wall.
`Thread *t0`	Pointer to the adjacent cell's thread.
`real cid[]`	Array of fluid-side diffusive heat transfer coefficients.
`real cir[]`	Array of radiative heat transfer coefficients.

Function returns

`void`

There are seven arguments to `DEFINE_HEAT_FLUX`: `name`, `f`, `t`, `c0`, `t0`, `cid`, and `cir`. You supply `name`, the name of the UDF. `f`, `t`, `c0`, and `t0` are variables that are passed by the Ansys Fluent solver to your UDF. Arrays `cir[]` and `cid[]` contain the linearizations of the radiative and diffusive heat fluxes, respectively, computed by Ansys Fluent based on the activated models. These arrays allow you to modify the heat flux in any way that you choose. Ansys Fluent computes the heat flux at the wall using these arrays *after* the call to `DEFINE_HEAT_FLUX`, so the total heat flux at the wall will be the currently computed heat flux (based on the activated models) with any modifications as defined by your UDF.

The diffusive heat flux (`qid`) and radiative heat flux (`qir`) are computed by Ansys Fluent according to the following equations:

```
qid = cid[0] + caf_fac*(cid[1]*C_T(c0,t0) - cid[2]*F_T(f,t)) - cid[3]*pow(F_T(f,t),4)
qir = cir[0] + cir[1]*C_T(c0,t0) - cir[2]*F_T(f,t) - cir[3]*pow(F_T(f,t),4)
```

where `caf_fac` is the convective augmentation factor defined using the `define/boundary-conditions/wall` text command (for further details, see Augmented Heat Transfer in the User's Guide).

The sum of `qid` and `qir` defines the total heat flux from the fluid to the wall (this direction being positive flux), and, from an energy balance at the wall, equals the heat flux of the surroundings (exterior to the domain). Note that heat flux UDFs (defined using `DEFINE_HEAT_FLUX`) are called by Ansys Fluent from within a loop over wall faces.

> **Important:**
>
> In order for the solver to compute `C_T` and `F_T`, the values you supply to `cid[1]` and `cid[2]` should never be zero.

2.3.18.3. Example

Implementing Ansys Fluent's P-1 Radiation Model Using User-Defined Scalars (p. 644) provides an example of the P-1 radiation model implementation through a user-defined scalar. An example of the usage of the `DEFINE_HEAT_FLUX` macro is included in that implementation.

2.3.18.4. Hooking a Heat Flux UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_HEAT_FLUX` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `heat_flux`) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking `DEFINE_HEAT_FLUX` UDFs (p. 463) for details.

2.3.19. `DEFINE_IGNITE_SOURCE`

2.3.19.1. Description

You can use `DEFINE_IGNITE_SOURCE` to customize the ignition time source term in the autoignition model.

2.3.19.2. Usage

`DEFINE_IGNITE_SOURCE` (name, c, t, source)

Argument Type	Description
`symbol name`	UDF name
`cell_t c`	Cell index
`Thread *t`	Pointer to cell thread on which the ignition source term is to be applied
`real *source`	Pointer to the ignition source term

Function returns

`void`

There are four arguments to `DEFINE_IGNITE_SOURCE`: `name`, `c`, `t`, and `source`. You supply `name`, the name of the UDF. `c`, `t` and `source` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to set the value referenced by the `source` pointer as shown in the example below.

2.3.19.3. Example

The following UDF, named `ign_udf_src`, specifies a custom source term in the ignition model. The source code must be executed as a compiled UDF in Ansys Fluent.

In the standard ignition model in Ansys Fluent, the source term for the ignition progress variable is given by a Livengood-Wu integral [7] (p. 713):

$$S_{ig} = \int_{t=t_0}^{t} \frac{dt}{\tau_{ig}} \tag{2.6}$$

where dt is the flow time step and τ_{ig} is the correlation between spark time and knock, by Douaud and Eyzat [3] (p. 713):

$$\tau = 0.01768 \left(\frac{ON}{100} \right)^{3.402} p^{-1.7} \exp\left(\frac{3800}{T} \right) \tag{2.7}$$

Here, ON is the octane number of the fuel, p is the absolute pressure in atmospheres and T is the temperature in Kelvin.

In the following UDF example, the Douaud and Eyzat correlation is used to calculate an induction time. See Additional Macros for Writing UDFs (p. 305) for details on the `NNULLP`, `C_STORAGE_R`, `C_PREMIXC_T`, `C_P`, `C_R`, `CURRENT_TIMESTEP` and `C_IGNITE` macros used below.

```
/*-----------------------------------------------------------------
 This UDF produces an ignition model source in Ansys Fluent v12.0 or
 later that uses the default parameters for the correlation of Douaud
 and Eyzat for knock.
 /*-----------------------------------------------------------------*/

#include "udf.h"

real A = 0.01768;    /* Preexponential      */
real Ea = 3800;      /* Activation temperature    */
real O_N = 90.0;     /* Octane number       */
real O_E = 3.402;    /* Octane number exponent    */
real P_E = -1.7;     /* Pressure exponent     */

static real A1 = 0.0;    /* Cached value of A*ON^OE   */
static real dt = 0.0;    /* Cached time step      */
static real p_op = 0.0;    /* Cached value of operating pressure */
static cxboolean lit = FALSE; /* Cached burning flag    */

DEFINE_IGNITE_SOURCE(ign_udf_src, c, t, source)
{
   real rho = C_R(c,t);
   real time = 0.0;
   real prog = NNULLP(THREAD_STORAGE(t,SV_PREMIXC_M1)) ?
     C_STORAGE_R(c,t,SV_PREMIXC_M1) :
     C_STORAGE_R(c,t,SV_PREMIXC) ;
   real fuel = 1.0 - prog;
   real T = C_PREMIXC_T(c,t);
   real P = C_P(c,t);
   real ipv = C_IGNITE(c,t);

   if (c == 0)
     {
        dt = CURRENT_TIMESTEP;
        p_op = RP_Get_Real("operating-pressure");
        A1 = A * pow(O_N/100,O_E);
  }

   if (ipv > 1.0)
     lit = TRUE;
   P += p_op;
   P /= 101325.;  /* in atm */
   P = MAX(P,0.01);  /* minimum pressure for ignition */

   if (fuel > 0.99 || lit)
     time = A1 * pow(P,P_E) * exp(Ea/T);

   if (time > 0.0)
     {
        real max_source = rho*(5.0-ipv)/dt;
        real user_source = rho/time;
        *source = MIN(user_source,max_source);
     }
   else
     *source = 0.0;

   return;
}
```

2.3.19.4. Hooking an Ignition Source UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_IGNITE_SOURCE` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `ign_udf_src`) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking `DEFINE_IGNITE_SOURCE` UDFs (p. 464) for details.

2.3.20. DEFINE_KW_GEKO Coefficients and Blending Function

2.3.20.1. Description

You can use `DEFINE_KW_GEKO_CSEP`, `DEFINE_KW_GEKO_CNW`, and `DEFINE_KW_GEKO_CMIX` to specify your own recipes for the calculation of the coefficients C_{SEP}, C_{NW}, and C_{MIX}, respectively. `DEFINE_KW_GEKO_BF` enables you to specify a user defined blending function.

2.3.20.2. Usage

DEFINE_KW_GEKO_CSEP (name, c, t)

DEFINE_KW_GEKO_CNW (name, c, t)

DEFINE_KW_GEKO_CMIX (name, c, t)

DEFINE_KW_GEKO_BF (name, c, t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Index that identifies the cell on which the GEKO coefficient or blending function is to be applied.
Thread *t	Pointer to cell thread.

Function returns

real

There are three arguments to these functions: `name`, `c`, and `t`. You specify the name of the UDF. `c` and `t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF returns the real value of the GEKO coefficients or blending function to the solver.

2.3.20.3. Example

In the following example, a zonal approach is demonstrated. Depending on the x-coordinate, the GEKO blending function is `1.0` or the built-in version is used.

```
#include "udf.h"
DEFINE_KW_GEKO_BF(user_geko_bf, c, t)
  {
     real bf_value;
     real xc[ND_ND];
     C_CENTROID(xc,c,t);
```

```
    if (xc[0] > 2.0)
      bf_value = Get_Geko_Blending_Function(c,t,
                 C_R(c,t),C_MU_L(c,t), C_WALL_DIST(c,t),
                 C_K(c,t),C_O(c,t));
    else
      bf_value = 1.0;
    return bf_value;
 }
```

2.3.20.4. Hooking a UDF for GEKO Coefficients or Blending Function to Ansys Fluent

After the UDF that you have defined using `DEFINE_KW_GEKO_CSEP`, `DEFINE_KW_GEKO_CNW`, `DEFINE_KW_GEKO_CMIX`, or `DEFINE_KW_GEKO_BF` is interpreted or compiled, the name of the argument that you supplied as the `DEFINE` macro argument (for example, user_geko_bf) will become visible and selectable in the Viscous Model dialog box in the drop-down menu of the corresponding model coefficient or blending function.

2.3.21. `DEFINE_MASS_TR_PROPERTY`

2.3.21.1. Description

You can use `DEFINE_MASS_TR_PROPERTY` to change inputs (such as heat transfer coefficients, heat flux augmentation/suppression factors, reference velocity, and temperature) for the semi-mechanistic boiling model.

2.3.21.2. Usage

`DEFINE_MASS_TR_PROPERTY` (name, c, mixture_thread, from_phase_index, from_species_index, to_phase_index, to_species_index, mass_transfer_index, tabular_data)

Important:

Note that all of the arguments to a `DEFINE` macro need to be placed on the same line in your source code. Splitting the `DEFINE` statement onto several lines will result in a compilation error.

Argument Type	Description
symbol name	UDF name.
cell_t c	Index of cell on the thread pointed to by `mixture_thread`.
Thread *mixture_thread	Pointer to mixture-level thread.
int from_phase_index	Index of phase from which mass is transferred.
int from_species_index	ID of species from which mass is transferred (ID= -1 if phase does not have mixture material).

Argument Type	Description
`int to_phase_index`	Index of phase to which mass is transferred.
`int to_species_index`	ID of species to which mass is transferred (ID= -1 if phase does not have mixture material).
`int mass_transfer_index`	Index of mass transfer mechanism.
`MT_Tabular_Data *tabular_data`	Pointer to the tabular data structure such as `tabular-pt` or `tabular-ptl`.

Function returns

`real`

There are nine arguments to `DEFINE_MASS_TR_PROPERTY`: `name`, `c`, `mixture_thread`, `from_phase_index`, `from_species_index`, `to_phase_index`, `to_species_index`, `mass_transfer_index`, and `tabular_data`. You supply `name`, the name of the UDF. The variables `c`, `mixture_thread`, `from_phase_index`, `from_species_index`, `to_phase_index`, `to_species_index`, `mass_transfer_index`, and `tabular_data` are passed by the Ansys Fluent solver to your UDF. Your UDF will return the real input to the solver for a quantity where the UDF is hooked.

2.3.21.3. Example

The following UDF, named `nuc_ht_coeff`, defines a nucleate boiling heat transfer coefficient in a two-phase mixture problem.

> **Note:**
>
> - The tabular-pt method supports the following functions for retrieving either temperature based on pressure or pressure based on temperature:
>   - `Get_Tabular_P_Given_T(tabular_data, T)`
>   - `Get_Tabular_T_Given_P(tabular_data, p_abs)`
>
> - The tabular_ptl method supports two additional functions for retrieving latent heat at given temperature or pressure:
>   - `Get_Tabular_L_Given_T(tabular_data, T)`
>   - `Get_Tabular_L_Given_P(tabular_data, p_abs)`

> **Important:**
>
> Note that in the example that follows, the `DEFINE_MASS_TR_PROPERTY` statement is broken up into two lines for the sake of readability. In your source file, you must make sure that the `DEFINE` statement is on one line only.

```
/* UDF to define a nucleate boiling heat transfer coefficient
   The "from" phase is the liquid phase and the "to" phase is the gas phase */
```

```
#include "udf.h"
#include "sg_mphase.h"
#define Tbulk 325
#define sigma 0.07

DEFINE_MASS_TR_PROPERTY(nuc_ht_coeff,f,t,c0,t0,from_index,from_species_index,to_index,to_species_index,mt_index
 {
   int liq_phase = from_index;
   int gas_phase = to_index;
   Thread *ptl    = THREAD_SUB_THREAD(t0, liq_phase);
   Thread *ptg    = THREAD_SUB_THREAD(t0, gas_phase);
   real T_sat = C_MT_SAT_TEMPERATURE(c0,t0, mt_index);
   real T0 = C_T(c0, t0);
   real Tw = F_T(f, t);
   real dT_sup = Tw - T_sat;
   real p_abs = C_P(c0,t0) + op_pres;

   real Psat_wall = Get_Tabular_P_Given_T(tabular_data, Tw);
   real Psat = Get_Tabular_P_Given_T(tabular_data, T_sat);
   real dP = MAX(Psat_wall - Psat, 0.);
   real h_lg = Get_Tabular_L_Given_P(tabular_data, p_abs);

   real rhol = C_R(c0, ptl);
   real rhog = C_R(c0, ptg);
   real mul = C_MU_L(c0, ptl);
   real cpl  = C_CP(c0, ptl);
   real kl   = C_K_L(c0, ptl);

   real tmp1 = 0.00122 * pow(kl, 0.79) * pow(cpl, 0.45) * pow(rhol, 0.49) * pow(dT_sup, 0.24) * pow(dP, 0.75);
   real tmp2 = pow(sigma, 0.5) * pow(mul, 0.29) * pow(h_lg, 0.24) * pow(rhog, 0.24);

   return tmp1 / tmp2;
 }
```

2.3.21.4. Hooking a `DEFINE_MASS_TR_PROPERTY` UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_MASS_TR_PROPERTY` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `nuc_ht_coeff`) will become visible and selectable in the **Evaporation-Condensation Model** dialog box in Ansys Fluent. See Hooking `DEFINE_MASS_TR_PROPERTY` UDFs (p. 465) for details.

2.3.22. `DEFINE_NET_REACTION_RATE`

2.3.22.1. Description

You can use `DEFINE_NET_REACTION_RATE` to compute the homogeneous net molar reaction rates of all species. The net reaction rate of a species is the sum over all reactions of the volumetric reaction rates:

$$R_i = \sum_{r=1}^{N_R} \hat{R}_{i,r} \tag{2.8}$$

where R_i is the net reaction rate of species i and $\hat{R}_{i,r}$ is the Arrhenius molar rate of creation/destruction of species i in reaction r.

A `DEFINE_NET_REACTION_RATE` UDF may be used for the laminar finite-rate (with the stiff chemistry solver), EDC, and PDF Transport models, as well as for the surface chemistry model. In

contrast, the surface UDF function `DEFINE_SR_RATE` is used for the laminar finite-rate model when stiff chemistry is not used.

2.3.22.2. Usage

`DEFINE_NET_REACTION_RATE (name, c, t, particle, pressure, temp, yi, rr, jac)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index of current particle.
`Thread *t`	Pointer to cell thread for particle.
`Particle *particle`	Pointer to `Particle` data structure that represents a particle used by the PDF transport model.
`double *pressure`	Pointer to pressure variable.
`double *temp`	Pointer to temperature variable.
`double *yi`	Pointer to array containing species mass fractions.
`double *rr`	Pointer to array containing net molar reaction rates.
`double *jac`	Pointer to array of Jacobians.

Function returns

`void`

There are nine arguments to `DEFINE_NET_REACTION_RATE`: `name`, `c`, `t`, `particle`, `pressure`, `temp`, `yi`, `rr`, and `jac`. You supply `name`, the name of the UDF. The variables `c`, `t`, `particle`, `pressure`, `temp`, `yi`, `rr`, and `jac` are passed by the Ansys Fluent solver to your UDF and have SI units. The outputs of the function are the array of net molar reaction rates, `rr` (with units kmol/m^3-s for volumetric reactions and kmol/m^2-s for the surface reactions), and the Jacobian array `jac`. The Jacobian is only required for surface chemistry, and is the derivative of the surface net reaction rate with respect to the species concentration.

`DEFINE_NET_REACTION_RATE` is called for all fluid zones (volumetric reactions as well as surface reactions in porous media) and for all wall thread zones whenever the **Reaction** option is enabled in the boundary conditions dialog box and the UDF is hooked to Ansys Fluent in the **User-Defined Function Hooks** dialog box.

Important:

`DEFINE_NET_REACTION_RATE` functions can be executed only as compiled UDFs.

2.3.22.3. Example

The following UDF, named `net_rxn`, assumes that the net volumetric reaction rate is the expression,

$$R_{net} = 1 / N_{spe} - Y_i \qquad (2.9)$$

where N_{spe} is the number of species.

```
/************************************************************
    Net Reaction Rate Example UDF
 ************************************************************/
#include "udf.h"

DEFINE_NET_REACTION_RATE(net_rxn,c,t,particle,pressure,temp,yi,rr,jac)
{
  int i;
  for(i=0;i<n_spe;i++)
     rr[i] = 1./(real)n_spe - yi[i];
}
```

Note that during the course of the ODE solution, the species mass fractions can exceed realizable bounds. For optimal ODE performance, the species mass fractions should not be clipped, but derived quantities, such as concentrations which are raised to non-integer powers, must be bounded. Also, if density is required, for instance to calculate concentrations, it should be calculated from the temperature and species passed into the UDF. Finally, double precision should be used for all local variables.

2.3.22.4. Hooking a Net Reaction Rate UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_NET_REACTION_RATE` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `net_rxn`) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking `DEFINE_NET_REACTION_RATE` UDFs (p. 467) for details.

2.3.23. `DEFINE_NOX_RATE`

2.3.23.1. Description

You can use the `DEFINE_NOX_RATE` to specify a custom NOx rate for thermal NOx, prompt NOx, fuel NOx, and N_2O intermediate pathways that can *either* replace the internally-calculated NOx rate in the source term equation, or be added to the Ansys Fluent rate. Example 1 demonstrates this use of `DEFINE_NOX_RATE`. By default, the **Add to Ansys Fluent Rate** option is enabled **UDF Rate** group box in each of the tabs under **Formation Model Parameters**, so that user-defined rates are added to the Ansys Fluent-calculated rates. You can change this default by selecting **Replace Ansys Fluent Rate**, so that the Ansys Fluent-calculated rate for that NOx pathway will not be used and it will instead be replaced by the NOx rate you have defined in your UDF.

> **Important:**
>
> Note that a single UDF is used to define the different rates for the four NOx pathways: thermal NOx, prompt NOx, fuel NOx, and N_2O intermediate pathway. That is, a NOx rate UDF can contain up to four separate rate functions that are concatenated in a single source file which you hook to Ansys Fluent.

`DEFINE_NOX_RATE` may also be used to calculate the upper limit for the integration of the temperature PDF (when temperature is accounted for in the turbulence interaction modeling). You can calculate a custom maximum limit (T_{max}) for each cell and then assign it to the `POLLUT_CTMAX(Pol-`

`lut_Par`) macro (see NOx Macros (p. 341) for further details about data access macros). Example 2 demonstrates this use of `DEFINE_NOX_RATE`.

Important:

If you want to use `DEFINE_NOX_RATE` only for the purpose of specifying T_{max}, then be sure that the user-defined NOx rate does not alter the internally-calculated rate for the source term calculation.

2.3.23.2. Usage

`DEFINE_NOX_RATE (name, c, t, Pollut, Pollut_Par, NOx)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *t`	Pointer to cell thread on which the NOx rate is to be applied.
`Pollut_Cell *Pollut`	Pointer to the data structure that contains the common data at each cell
`Pollut_Parameter *Pollut_Par`	Pointer to the data structure that contains auxiliary data.
`NOx_Parameter *NOx`	Pointer to the data structure that contains data specific to the NOx model.

Function returns

`void`

There are six arguments to `DEFINE_NOX_RATE`: `name`, `c`, `t`, `Pollut`, `Pollut_Par`, and `NOx`. You will supply `name`, the name of the UDF. `c`, `t`, `Pollut`, `Pollut_Par`, and `NOx` are variables that are passed by the Ansys Fluent solver to your function. A `DEFINE_NOX_RATE` function does not output a value. The calculated NOx rates (or other pollutant species rates) are returned through the `Pollut` structure as the forward rate `POLLUT_FRATE(Pollut)` and reverse rate `POL-LUT_RRATE(Pollut)`, respectively.

Important:

The data contained within the NOx structure is specific *only* to the NOx model. Alternatively, the `Pollut` structure contains data at each cell that are useful for all pollutant species (for example, forward and reverse rates, gas phase temperature, density). The `Pollut_Par` structure contains auxiliary data common to all pollutant species (for example, equation solved, universal gas constant, species molecular weights). Note that molecular weights extracted from the `Pollut_Par` structure (that is, `Pollut_Par->sp[IDX(i)].mw` for pollutant species—NO, HCN, and so on—and `Pollut_Par->sp[i].mw` for other species, such as O_2) has units of kg/kmol.

2.3.23.3. Example 1

The following compiled UDF, named `user_nox`, exactly reproduces the default Ansys Fluent NOx rates for the prompt NOx pathway. Note that this UDF will replace the Ansys Fluent rate *only* if you select `Replace Ansys Fluent Rate` in the **UDF Rate** group box in the **Prompt** tab. Otherwise, the rate computed in the UDF will be added to Ansys Fluent's default rate. See Hooking `DEFINE_NOX_RATE` UDFs (p. 469) for details.

See NOx Macros (p. 341) for details about the NOx macros (for example, POLLUT_EQN, MOLECON, ARRH) that are used in pollutant rate calculations in this UDF.

```
/*******************************************************************
 UDF example of User-Defined NOx Rate for Ansys Fluent 12 or later
 If used with the "Replace with UDF" radio buttons activated,
this UDF will exactly reproduce the default Ansys Fluent NOx
 rates for prompt NOx pathway.
The flag "Pollut_Par->pollut_io_pdf == IN_PDF" should always
 be used for rates other than that from char N, so that if
 requested, the contributions will be PDF integrated. Any
 contribution from char must be included within a switch
 statement of the form "Pollut_Par->pollut_io_pdf == OUT_PDF".
 *
 * Arguments:
 *  char nox_func_name          - UDF name
 *  cell_t c                    - Cell index
 *  Thread *t                   - Pointer to cell thread on
 *                        which the NOx rate is to be applied
 *  Pollut_Cell *Pollut         - Pointer to Pollut structure
 *  Pollut_Parameter *Pollut_Par  - Pointer to Pollut_Par structure
 *  NOx_Parameter *NOx          - Pointer to NOx structure

 *******************************************************************/

#include "udf.h"

DEFINE_NOX_RATE(user_nox, c, t, Pollut, Pollut_Par, NOx)
{
/* NOx->prompt_nox = Flag to indicate Prompt NOx is enabled
 * NOx->prompt_udf_replace = Flag to indicate UDF replace
 * Pollut_Par->nfstreams = Number of fuel streams
 * Pollut_Par->nfspe[i] = Number of fuel species in stream "i"
 * NOx->equiv_ratio[i] = Equivalence ratio for stream "i"
 * NOx->c_number[i] = Carbon number for stream "i"
 * Pollut_Par->fuel_idx[j][i] = Index of jth species in stream "i"
 * Pollut_Par->fuel_dup[j][i] = Fuel species duplication check
 * Pollut_Par->uni_R = Universal gas constant in SI units
 * Pollut->temp_m = Mean gas temperature (K)
 * Pollut->press = Pressure in SI units
 * Pollut->oxy_order = Oxygen order (please refer to user manual)
 */
 POLLUT_FRATE(Pollut) = 0.0;
 POLLUT_RRATE(Pollut) = 0.0;

 switch (Pollut_Par->pollut_io_pdf) {
 case IN_PDF:
   /* Included source terms other than those from char */

   if (POLLUT_EQN(Pollut_Par) == EQ_NO) {

       /* Prompt NOx */
       if (NOx->prompt_nox && NOx->prompt_udf_replace) {
         int ifstream;
         real f=0., rf;

         Rate_Const K_PM = {6.4e6, 0.0, 36483.49436};

         for(ifstream=0; ifstream<Pollut_Par->nfstreams; ifstream++) {
```

```
               int i;
               real xc_fuel=0., eqr=NOx->equiv_ratio[ifstream];
               for (i=0; i<Pollut_Par->nfspe[ifstream]; i++) {
                   if(!Pollut_Par->fuel_dup[i][ifstream])
                       xc_fuel += MOLECON(Pollut, Pollut_Par->fuel_idx[i][ifstream]);
               }
               f += (4.75 + 0.0819*NOx->c_number[ifstream]
                   - 23.2*eqr + 32.0*pow(eqr, 2.) - 12.2*pow(eqr, 3.))*xc_fuel;
           }
           rf = ARRH(Pollut, K_PM);
           rf *= pow((Pollut_Par->uni_R*Pollut->temp_m/Pollut->press),
               (1.+Pollut->oxy_order));
           rf *= pow(MOLECON(Pollut, O2), Pollut->oxy_order);
           rf *= MOLECON(Pollut, N2);

           POLLUT_FRATE(Pollut) += f*rf;
       }
     }
     break;

     case OUT_PDF:
     /* Char Contributions, must be included here */
     break;

   default:
   /* Not used */
   break;
   }
}
```

2.3.23.4. Example 2

The following compiled UDF, named `nox_func_name`, specifies a custom maximum limit (T_{max}) for the integration of the temperature PDF for each cell. Note that this UDF does not alter the internally-calculated NOx rate.

See NOx Macros (p. 341) for details about the NOx macro (`POLLUT_CTMAX`) used in this UDF.

```
/***********************************************************
 UDF example of User-Defined Tmax value
 *
 * Arguments:
 *  char nox_func_name        - UDF name
 *  cell_t c            - Cell index
 *  Thread *t           - Pointer to cell thread
 *              on which the NOx rate
 *              is to be applied
 *  Pollut_Cell *Pollut       - Pointer to Pollut_Cell
 *              structure
 *  Pollut_Parameter *Pollut_Par   - Pointer to Pollut_Parameter
 *              structure
 *  NOx_Parameter *NOx       - Pointer to NOx_Parameter
 *              structure
 Ansys Fluent Version: 12.0 or later
 ***********************************************************/

#include "udf.h"

int ud_nox_do_once=1;

enum
{
    CELL_TMAX=0,
    N_REQUIRED_UDM
};

/*Compute/assign Tmax at each cell*/
```

```
real ud_eval_cell_tmax(cell_t c,Thread *t)
{
   real tmax = 0.;

   /* Compute cell-based Tmax value */
   tmax = 1.1*C_T(c,t); /* This is only an example */

   return tmax;
}

DEFINE_NOX_RATE(user_nox, c, t, Pollut, Pollut_Par, NOx)
{
   /* Assign cell-based Tmax value */
   POLLUT_CTMAX(Pollut_Par) = ud_eval_cell_tmax(c,t);
   /*POLLUT_CTMAX(Pollut_Par) = C_UDMI(c,t,CELL_TMAX);*/
}

DEFINE_ON_DEMAND(init_tmax)
{
   Domain *domain;
   register Thread *t;
   register cell_t c;

   Message("Computing/Storing cell Tmax values\n");
   domain = Get_Domain(1);

   /* Store User-Defined Tmax at each cell */
   if(ud_nox_do_once == 1) {
     if(n_udm < N_REQUIRED_UDM)
        Error("Not enough udm allocated\n");

     thread_loop_c (t,domain)
        begin_c_loop (c,t)
          C_UDMI(c,t,CELL_TMAX) = ud_eval_cell_tmax(c,t);
        end_c_loop (c,t)
     ud_nox_do_once = 0;
   }
   Message("Computing cell Tmax values completed..\n");
}
```

2.3.23.5. Hooking a NOx Rate UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_NOX_RATE is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first DEFINE macro argument (for example, user_nox) will become visible and selectable in the **NOx Model** dialog box in Ansys Fluent. See Hooking DEFINE_NOX_RATE UDFs (p. 469) for details.

2.3.24. DEFINE_PERFORATED_CD

2.3.24.1. Description

You can use the DEFINE_PERFORATED_CD to customize the formulation of the discharge coefficient C_D, which is used in Equation 7.143 and Equation 7.144 in the *Fluent User's Guide* to estimate the injection mass flow rate in the perforated walls. The DEFINE_PERFORATED_CD UDF is only available for cases that involve discrete injections with the dynamic injection method.

2.3.24.2. Usage

`DEFINE_PERFORATED_CD (name, t, cd, local_variables)`

Important:

- Note that all the arguments of a `DEFINE` macro need to be placed on the same line in your source code. Splitting the `DEFINE` statement into several lines will result in a compilation error.

- `DEFINE_PERFORATED_CD` functions can be executed only as compiled UDFs.

Argument Type	**Description**
`symbol name`	UDF name.
`Thread *t`	Pointer to face thread.
`real *cd`	Pointer to parameter Cd
`real *local_variables`	Pointer to the variables that can be used in the formulation of Cd

Function returns

`void`

There are four arguments to `DEFINE_PERFORATED_CD`: `name`, `t`, `cd`, and `local_variables`. You supply `name`, the name of the UDF. The variables `t` and `local_variables` are passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the variable `cd`, which will be used to estimate the injection mass flow rate in the perforated wall when the dynamic injection method is used.

The array `local_variables` contains values of several pre-defined variables calculated by Ansys Fluent and listed in Table 2.7: Variables Stored in the Array `local_variables` (p. 92). They can be used to define your C_D as a function of these variables. You can directly refer to the variable you want to use in your UDF by using either the integer index or the enumerator.

Table 2.7: Variables Stored in the Array `local_variables`

Index	Enumerator	Variable
0	`DIA_DC`	Hole diameter
1	`LEN_DC`	Hole length
2	`AREA_DC`	Injection hole area
3	`PRESS_INJ_DC`	Injection hole pressure
4	`PRESS_EXT_DC`	Extraction hole pressure
5	`TEMP_DC`	Injection hole temperature
6	`DEN_DC`	Injection hole density
7	`XVEL_DC`	X component of the injection hole velocity
8	`YVEL_DC`	Y component of the injection hole velocity

Index	Enumerator	Variable
9	ZVEL_DC	Z component of the injection hole velocity

2.3.24.3. Example

The below examples shows the usage of `DEFINE_PERFORATED_CD`, named `my_cdprofile`, to compute the discharge coefficient C_D as the following function:

$$Cd = \begin{cases} 0.6 + 0.1 L/D & if\ L/D < 2 \\ 0.8 & if\ L/D \geq 2 \end{cases}$$

The first example illustrates how to use the index to refer to the variables in the `local_variables` array.

```
/*****************************************************************
#include "udf.h"
DEFINE_PERFORATED_CD(my_cdprofile, t, cd, local_variables)
{
  if(local_variables[1] / local_variables[0] <2.0 )
    *cd = 0.6 + 0.1*local_variables[1] / local_variables[0];
  else
    *cd = 0.8;
  return;
}
```

The second example illustrates how to use the enumerator to refer to the variables in the `local_variables` array.

```
/*****************************************************************
#include "udf.h"
#include "perforated.h"
DEFINE_PERFORATED_CD(my_cdprofile, t, cd, local_variables)
{
  if(local_variables[LEN_DC] / local_variables[DIA_DC] <2.0 )
    *cd = 0.6 + 0.1*local_variables[LEN_DC] / local_variables[DIA_DC];
  else
    *cd = 0.8;
  return;
}
```

2.3.24.4. Hooking a `DEFINE_PERFORATED_CD` UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_PERFORATED_CD` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `my_cdprofile`) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking `DEFINE_PERFORATED_CD` UDFs (p. 471) for details.

2.3.25. `DEFINE_PDF_TABLE`

2.3.25.1. Description

The Non-Premixed and Partially-Premixed models in Ansys Fluent employ look-up tables that store the convolution of state-relations with assumed-shape PDFs as described by Equation 8.17 to Equation 8.20, Equation 8.25 and Equation 8.26 in the Theory Guide. Ansys Fluent solves transport

equations for lower moments, such as the mean mixture fraction and the mean enthalpy, and interpolates the PDF table for the required variables:

- Mean temperature

- Mean density

- Mean specific heat

- Mean mixture molecular weight

- Mean species mole fractions

- Mean forward and reverse reaction rates of transported scalars with FGM tables

- Mean heat release rate

- Mean progress variable finite rate source term

You can use `DEFINE_PDF_TABLE` to customize the above variables.

Important:

- When using `DEFINE_PDF_TABLE`, you must use Ansys Fluent's transport equations for mixture fraction; for non-adiabatic cases, you must use mean enthalpy.

- The model settings and options that apply to your user-defined PDF Table must be set through the standard Ansys Fluent interface and a valid PDF table with the same settings and options must be generated or read into Ansys Fluent before you can use your UDF table. For example, if your PDF Table is a two-mixture fraction non-adiabatic table, you first generate/read a valid two-mixture fraction non-adiabatic PDF file. The reason for doing this is that Ansys Fluent will need to access some information about your system, such as the species order and the boundary compositions and temperatures through this default PDF table.

- When generating your default Ansys Fluent PDF file, you must use the same thermodynamic database file as you used to create your PDF table.

- You must ensure that the species order in your default Ansys Fluent PDF file is identical to the order in your PDF table. The default Ansys Fluent species order is in the material structure pdf-mixture, which is passed to the UDF.

- `DEFINE_PDF_TABLE` must use the identical fuel and oxidizer boundary compositions and temperatures as in the corresponding default Ansys Fluent PDF file. If you are using the two mixture fraction model, the same applies for the secondary stream.

- When you are using the Partially-Premixed or the Inert models, the `DEFINE_PDF_TABLE` UDF must return the properties of the **burnt** mixture.

- The UDF cannot currently be used for calculation of pollutant rates. Instead, Ansys Fluent will perform these calculations.

Your UDF can access and use the variables and functions in the default PDF table, such as boundary composition values and the adiabatic enthalpy function, listed in the header files pdf_props.h and pdf_table.h, which you would need to include in your UDF.

The UDF is called for all fluid cells and boundary faces, every iteration, and care should be taken to efficiently interpolate your table.

2.3.25.2. Usage

`DEFINE_PDF_TABLE (name, m, c, t, fmean, fvar, fmean2, fvar2, cmean, cvar h, what, prop, x, s_pollut)`

Important:

- Note that all the arguments of a `DEFINE` macro need to be placed on the same line in your source code. Splitting the `DEFINE` statement into several lines will result in a compilation error.

- `DEFINE_PDF_TABLE` functions can be executed only as compiled UDFs.

Argument Type	Description
`symbol name`	UDF name.
`Material *m`	Pointer to the mixture material *pdf-mixture*.
`cell_t c`	Cell or face index.
`Thread *t`	Pointer to the cell or face thread.
`real fmean`	Mean mixture fraction.
`real fvar`	Mixture fraction variance.
`real fmean2`	Secondary mean mixture fraction.
`real fvar2`	Secondary mixture fraction variance.
`real cmean`	Mean unnormalized progress variable (used with the partially premixed flamelet model).
`real cvar`	Progress variable variance (used with the partially premixed flamelet model).
`real h`	Mean enthalpy.
`int what`	Integer indicating the variables that the Ansys Fluent solver is expecting to be computed by the UDF as follows:
	0 calculate the thermodynamic properties in array prop
	1 calculate the thermodynamic properties and the species mole fractions x

Argument Type	Description
	2 calculate the thermodynamic properties, the species mole fractions and the pollutant rates in array s_pollut
	3 calculate FGM scalar equations forward and reverse rates from Equation 8.120 in the *Fluent Theory Guide*.
`real prop[MAX_PROP_UDF]`	Thermodynamic variables as follows:
	`prop[TEMP_UDF]` temperature
	`prop[DEN_UDF]` density
	`prop[CP_UDF]` specific heat
	`prop[MOL_WT_MIX_UDF]` mean molecular weight
	`prop[TSS_SCALAR_START_UDF + 2 * i]` forward rate of scalar i
	`prop[TSS_SCALAR_START_UDF + 2 * i + 1]` reverse rate of scalar i
	`prop[SCALAR_PRMX_SOURCE_UDF]` progress variable source term in 1/s
	`prop[SCALAR_HRR_UDF]` heat release rate
`real *x`	Species mole fractions.
`s_pollut[MAX_POLLUT_EQNS]`	Array of pointers to structure of mean pollutant rates for `MAX_POLLUT_EQNS` pollutants as follows:
	`s_pollut[ ]->fwdrate` forward rate gmol/m3/s
	`s_pollut[ ]->revrate` reverse rate gmol/m3/s
	`s_pollut[ ]->quasirate` quasi-steady concentration gmol/m3
	`s_pollut[ ]->rate[ ]` array of overall rates of individual pollutant models (used in postprocessing)

Function returns

`void`

There are thirteen arguments to `DEFINE_PDF_TABLE`. You supply `name`, the name of the UDF. The variables `m`, `c`, `t`, `fmean`, `fvar`, `fmean2`, `fvar2`, `cmean`, `cvar`, `h`, `what` are passed by the Ansys Fluent solver to your UDF. The output of the function are the array of thermodynamic variables `prop` and the array of mole fractions `x`.

Important:

For the `DEFINE_PDF_TABLE` UDF to work correctly, you must provide temperature, density, specific heat, molecular weight and species mole fractions through the array `prop`.

You can use macros listed in to extract the local PDF table cell information. The macros use the following arguments:

- `cell_t c`: Cell index
- `Thread *t`: Pointer to thread
- `int n`: Species index
- `real fmean`: Mean mixture fraction

Table 2.8: Generic Macros `Get_PDF`

Macro	Returns
`Get_Pdf_Xi(c,t,n)`	`real` PDF mole fraction for species `n`
`Get_Pdf_Yi(c,t,n)`	`real` PDF mass fraction for species `n`
`Get_Pdf_Temperature(c,t)`	`real` PDF temperature [K]
`Get_Pdf_Density(c,t)`	`real` PDF density [kg/m^3]
`Get_Pdf_MolWeight(c,t)`	`real` PDF mixture molecular weight
`Get_Pdf_Tss_Revrates(c,t,n)`	`real` PDF net chemical reverse rate of the FGM scalar `n` [kg/m^3/s] as in Equation 8.119 in the *Fluent Theory Guide*
`Get_Pdf_Tss_Fwdrates(c,t,n)`	`real` PDF net chemical forward rate of the FGM scalar `n` [kg/m^3/s] as in Equation 8.119 in the *Fluent Theory Guide*
`Get_Pdf_Min_Enthalpy(c,t)`	`real` local minimum PDF table static enthalpy [J/kg]
`Get_Pdf_Max_Enthalpy(c,t)`	`real` local maximum PDF table static enthalpy [J/kg]
`Get_Pdf_HeatReleaserate(c,t)`	`real` FGM heat release rate [J/m^3/s]
`Pdf_Adiabatic_Enthalpy (fmean)`	`real` PDF adiabatic enthalpy for a given mixture fraction [J/kg]

An example of the usage of these macros is shown in

2.3.25.3. Example

The following example is a source code template where the `DEFINE_PDF_TABLE` function is used to calculate properties and mole fractions from the PDF file loaded by the user.

```
#include <udf.h>
#include "pdf_props.h"
#include "pdf_table.h"
#define DEN_MIN 1E-4
DEFINE_PDF_TABLE(pdf_table, m, c, t, fmean, fvar, fmean2, fvar2, cmean, cvar, h, what, prop, x, s_pollut)
{
  int i;

  if NULLP(pf)
  Error("Please generate or read a Fluent PDF file first\n");

  #if 1
  if (what < 0) /*special call to return the FGM flame speed source*/
  {
    prop[SCALAR_PRMX_SOURCE_UDF] = Get_Premix_Frate_Source_Term(c, t)/C_R(c,t); /*This is the value
                                                      of progress variable source term in 1/s */
  }
  else if (what == 1) /* get all pdf values : T, rho, MW, mile fractions, cp*/
  {
    prop[TEMP_UDF] = Get_Pdf_Temperature(c, t);
    prop[CP_UDF] = Get_Pdf_Cp(c,t);
    prop[DEN_UDF]  = Get_Pdf_Density(c,t);
    prop[MOL_WT_MIX_UDF] = Get_Pdf_MolWeight(c,t);

    for (i = 0; i < n_spe_pdf; i++)
    x[i] = Get_Pdf_Xi(c,t,i);  /*get pdf mole fraction of species i */

  }
  else if (what == 3) /*TSS SCALARS RATES*/
  {
    tss_num_scalars = RP_Get_Integer("pdftss/tss-num-scalars");
    if (tss_num_scalars > 0)
    {
      real fwdrate = 0.0;
      real revrate = 0.0;
      for (i = 0; i < tss_num_scalars; i++)
      {
        if ( PdfModel.premix_flamelet && NNULLP(t) && GENERIC_CELL_THREAD_P(t) )
        fwdrate = Get_Pdf_Tss_FwdRates(c, t, i);

        if ( PdfModel.premix_flamelet && NNULLP(t) && GENERIC_CELL_THREAD_P(t))
        revrate = Get_Pdf_Tss_RevRates(c, t, i) ;

        prop[TSS_SCALAR_START_UDF +  2 * i] = fwdrate;
        prop[TSS_SCALAR_START_UDF +  2 * i + 1] = revrate;
      }
    }
  }
  else /*(what == 2 || 0)*/
  {
    prop[TEMP_UDF] = Get_Pdf_Temperature(c, t);
    prop[CP_UDF] = Get_Pdf_Cp(c,t);
    prop[DEN_UDF]  = Get_Pdf_Density(c,t);
    prop[MOL_WT_MIX_UDF] = Get_Pdf_MolWeight(c,t);
  }

  #endif

}
```

2.3.25.4. Hooking a DEFINE_PDF_TABLE UDF to Ansys Fluent

After you have enabled the **Non-Premixed** or **Partially-Premixed** models, generated or read a valid PDF table, and compiled (Compiling UDFs (p. 399)) the DEFINE_PDF_TABLE UDF, the name of the argument that you supplied as the first DEFINE macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking DEFINE_PDF_TABLE UDFs (p. 474) for details.

2.3.26. DEFINE_PR_RATE

2.3.26.1. Description

You can use DEFINE_PR_RATE to specify a custom particle surface reaction for the multiple surface reactions particle model. During Ansys Fluent execution, the same UDF is called sequentially for all particle surface reactions, so DEFINE_PR_RATE can be used to define custom reaction rates for a single reaction, or for multiple reactions. The volumetric and wall surface reactions are not affected by the definition of this macro and will follow the designated rates. Note that a DEFINE_PR_RATE UDF is *not* called with the coupled solution option, so you will need to disable the **Coupled Heat Mass Solution** option in the **Discrete Phase Model** dialog box when using it. The auxiliary function, zbrent_pr_rate, which is provided below, can be used when there is no analytical solution for the overall particle reaction rate.

2.3.26.2. Usage

DEFINE_PR_RATE (name, c, t, r, mw, ci, tp, sf, dif_index, cat_index, rr)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index of current particle.
Thread *t	Pointer to cell thread for particle.
Reaction *r	Pointer to data structure that represents the current reaction.
real *mw	Pointer to array containing gaseous and surface species molecular weights
real *ci	Pointer to array containing gas partial pressures.
Tracked_Particle *tp	Pointer to Tracked_Particle data structure that contains data related to the particle being tracked.
real *sf	Pointer to array containing mass fractions of the solid species in the particle char mass at the current time step.
int dif_index	Diffusion controlled species as defined in the **Reactions** dialog box for the current reaction.
int cat_index	Catalyst species as defined in the **Reactions** dialog box for the current reaction.

Argument Type	Description
`real *rr`	Pointer to array containing particle reaction rate (kg/s).

Function returns

`void`

There are eleven arguments to `DEFINE_PR_RATE`: `name`, `c`, `t`, `r`, `mw`, `ci`, `tp`, `sf`, `dif_index`, `cat_index`, and `rr`. You supply `name`, the name of the UDF. `c`, `t`, `r`, `mw`, `ci`, `tp`, `sf`, `dif_index`, `cat_index`, and `rr` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to set the value referenced by the `real` pointer `rr` to the particle reaction rate in kg/s.

Note that `tp` is an argument to many particle-specific macros defined in DPM Macros (p. 335). `sf` is the same as the order in which the species are defined in the **Selected Solid Species** list in the **Create/Edit Materials** dialog box, which is opened from the **Edit Species** names option for the **Mixture Material**.

`DEFINE_PR_RATE` is called by Ansys Fluent every time step during the particle tracking calculation. The auxiliary function `zbrent_pr_rate` is used when there is no analytical solution for the overall particle reaction rate. It uses Brent's method to find the root of a function known to lie between $x1$ and $x2$. The root will be refined until its accuracy has reached tolerance `tol`. This is demonstrated in Example 2.

2.3.26.3. Auxiliary function

```
zbrent_pr_rate(real (*func),(real,real[],int [],cxboolean [],char *,)
real ruser[],int iuser[], cxboolean buser[],char *cuser,real x1 real
x2,real tol,cxboolean *ifail)
```

Auxiliary function returns: `real`

2.3.26.4. Example 1

The following UDF, named `user_pr_rate`, specifies a particle reaction rate given by Equation 7.79 in the Theory Guide, where the effectiveness factor η_r is defined as

$$\eta_r = 1 - x$$

where x is the fractional conversion of the particle char mass. In this case, the UDF will be applied to all surface particle reactions defined in the Ansys Fluent model.

```
/* UDF of specifying the surface reaction rate of a particle */

 #include "udf.h"

 #define A1 0.002
 #define E1 7.9e7

 DEFINE_PR_RATE(user_pr_rate,c,t,r,mw,pp,tp,sf,dif_i,cat_i,rr)
 {
/* Argument types
    cell_t c
    Thread *t
```

```
    Reaction *r (reaction structure)
    real *mw (species molecular weight)
    real *pp (gas partial pressures)
    Tracked_Particle *tp (particle structure)
    real *sf  (current mass fractions of solid species in
        particle char mass)
    int dif_i (index of diffusion controlled species)
    int cat_i (index of catalyst species)
    real *rr  (rate of reaction kg/s)
*/

real ash_mass =
TP_INIT_MASS(tp)*(1.-DPM_CHAR_FRACTION(tp)-DPM_VOLATILE_FRACTION(tp));

real one_minus_conv =
MAX(0.,(TP_MASS(tp) -ash_mass) / TP_INIT_MASS(tp)/ DPM_CHAR_FRACTION(tp));

real rate = A1*exp(-E1/UNIVERSAL_GAS_CONSTANT/TP_T(tp));

*rr=-rate*TP_DIAM(tp)*TP_DIAM(tp)*M_PI*sf[0]*one_minus_conv;
}
```

2.3.26.5. Example 2

The following compiled UDF, named `user_rate`, specifies a particle reaction rate given by Equation 7.74 and Equation 7.77 in the Theory Guide. The reaction order on the kinetic rate is 0.9 and the effectiveness factor η_r is defined as

$$\eta_r = 1 - x$$

where x is the fractional conversion of the particle char mass. In this case it is necessary to obtain a numerical solution for the overall surface reaction rate.

This UDF is called only for reaction 2, which means that the default Ansys Fluent solution will be used for the rest of the particle surface reactions defined.

```
/* UDF of specifying the surface reaction rate of a particle,
using a numerical solution */

 #include "udf.h"

 #define c1 5e-12
 #define A1 0.002
 #define E1 7.9e7
 #define tolerance 1e-4
 #define order 0.9

 real reaction_rate(real rate, real ruser[], int iuser[], cxboolean buser[],
char *cuser)

 /* Note that all arguments in the reaction_rate function call
 in your .c source file MUST be on the same line or a
 compilation error will occur */

 {
    return (ruser[2]*pow(MAX(0.,(ruser[0]-rate/ruser[1])),order) -rate);
}

 DEFINE_PR_RATE(user_rate,c,t,r,mw,pp,tp,sf,dif_i,cat_i,rr)
 {
if (!strcmp(r->name, "reaction-2"))
    {
    cxboolean ifail=FALSE;

    real ash_mass =
```

```
        TP_INIT_MASS(tp)*(1.-DPM_CHAR_FRACTION(tp)-DPM_VOLATILE_FRACTION(tp));

        real one_minus_conv =
        MAX(0.,(TP_MASS(tp) -ash_mass) / TP_INIT_MASS(tp)/ DPM_CHAR_FRACTION(tp));

        real ruser[3];
        int iuser[1];  cxboolean buser[1];
        char cuser[30];
        real ratemin, ratemax, root;

        ruser[0] = pp[dif_i];
        ruser[1] = MAX(1.E-15, (c1*pow(0.5*(TP_T(tp)+C_T(c,t)),0.75)/TP_DIAM(tp)));
        ruser[2] = A1*exp(-E1/UNIVERSAL_GAS_CONSTANT/TP_T(tp));
        strcpy(cuser, "reaction-2");
        ratemin=0;
        ratemax=ruser[1]*pp[dif_i];

        /* arguments for auxiliary function zbrent_pr_rate */

        root = zbrent_pr_rate(reaction_rate, ruser, iuser, buser, cuser,
              ratemin, ratemax, tolerance, &ifail);

        if (ifail) root=MAX(1.E-15,ruser[1]);

        *rr=-root*TP_DIAM(tp)*TP_DIAM(tp)*M_PI*sf[0]*one_minus_conv;

        Message("Fail status %d\n", ifail);
        Message("Reaction rate for reaction %s : %g\n", cuser, *rr);

    }
  }
```

In this example, a real function named `reaction_rate` is defined at the top of the UDF. The arguments of `reaction_rate` are `real rate`, and the pointer arrays `real ruser[]`, `integer iuser[]`, `cxboolean buser[]`, and `char *cuser`, which must be declared and defined in the main body of the `DEFINE_PR_RATE` function.

Typically, if the particle surface reaction rate is described by

```
rate = f(ruser[],iuser[],rate)
```

then the real function (in this example `reaction_rate`) should return

```
f(ruser[],iuser[],rate) - rate
```

The variables `cxboolean buser[]` and `char *cuser` can be used to control the flow of the program in cases of complicated rate definitions.

`ratemin` and `ratemax`, hold the minimum and maximum possible values of the variable rate, respectively. They define the search interval where the numerical algorithm will search for the root of the equation, as defined in the function `reaction_rate`. The value of reaction rate `rr` will be refined until an accuracy specified by the value of tolerance `tol` is reached.

The variable `ifail` will take the value `TRUE` if the root of the function has not been found.

2.3.26.6. Hooking a Particle Reaction Rate UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_PR_RATE` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `user_pr_rate`) will become visible and selectable

in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking `DEFINE_PR_RATE` UDFs (p. 475) for details.

2.3.27. `DEFINE_PRANDTL` UDFs

The following `DEFINE` macros can be used to specify Prandtl numbers in Ansys Fluent, for single-phase flows.

2.3.27.1. `DEFINE_PRANDTL_D`

2.3.27.2. Description

You can use `DEFINE_PRANDTL_D` to specify Prandtl numbers for turbulent dissipation (ε).

2.3.27.3. Usage

`DEFINE_PRANDTL_D (name, c, t)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index of cell on which the Prandtl number function is to be applied.
`Thread *t`	Pointer to cell thread.

Function returns

`real`

There are three arguments to `DEFINE_PRANDTL_D`: `name`, `c`, and `t`. You supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the `real` value for the turbulent dissipation Prandtl number to the solver.

2.3.27.4. Example

An example of a `DEFINE_Prandtl_D` UDF is provided below in the source listing for `DEFINE_PRANDTL_K`.

2.3.27.5. Hooking a Prandtl Number UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_PRANDTL_D` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `user_pr_d`) will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. See Hooking `DEFINE_PRANDTL` UDFs (p. 477) for details.

2.3.27.6. `DEFINE_PRANDTL_K`

2.3.27.7. Description

You can use `DEFINE_PRANDTL_K` to specify Prandtl numbers for turbulence kinetic energy (k).

2.3.27.8. Usage

`DEFINE_PRANDTL_K (name, c, t)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index that identifies the cell on which the Prandtl number function is to be applied.
`Thread *t`	Pointer to cell thread.

Function returns

`real`

There are three arguments to `DEFINE_PRANDTL_K`: `name`, `c`, and `t`. You supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the `real` value for the kinetic energy Prandtl number to the solver.

2.3.27.9. Example

The following UDF implements a high-Re version of the RNG model, using the k-ε option that is activated in Ansys Fluent.

Three steps are required:

1. Set Cmu, C1eps, and C2eps as in the RNG model.

2. Calculate Prandtl numbers for k and ε using the UDF.

3. Add the -r source term in the ε equation.

In the RNG model, diffusion in k and ε equations appears as

$$\left(\mu + \mu_t\right) * \alpha$$

while in the standard k-ε model, it is given by

$$\mu + \frac{\mu_t}{Pr}$$

For the new implementation, a UDF is needed to define a Prandtl number Pr as

$$Pr = \frac{\mu_t}{\left[\left(\mu + \mu_t\right) * \alpha - \mu\right]}$$

in order to achieve the same implementation as the original RNG Model.

The following functions (which are concatenated into a single C source code file) demonstrate this usage. Note that the source code must be executed as a compiled UDF.

```
#include "udf.h"

DEFINE_PRANDTL_K(user_pr_k,c,t)
{
   real pr_k, alpha;
   real mu = C_MU_L(c,t);
   real mu_t = C_MU_T(c,t);

   alpha = rng_alpha(1., mu + mu_t, mu);

   pr_k = mu_t/((mu+mu_t)*alpha-mu);

   return pr_k;
}

DEFINE_PRANDTL_D(user_pr_d,c,t)
{
   real pr_d, alpha;
   real mu = C_MU_L(c,t);
   real mu_t = C_MU_T(c,t);

   alpha = rng_alpha(1., mu + mu_t, mu);

   pr_d = mu_t/((mu+mu_t)*alpha-mu);

   return pr_d;
}

DEFINE_SOURCE(eps_r_source,c,t,dS,eqn)
{
   real con, source;
   real mu = C_MU_L(c,t);
   real mu_t = C_MU_T(c,t);
   real k  = C_K(c,t);   real d  = C_D(c,t);
   real prod = C_PRODUCTION(c,t);

   real s = sqrt(prod/(mu+ mu_t)) ;
   real eta  = s*k/d;
   real eta_0 = 4.38;
   real term = mu_t*s*s*s/(1.0 + 0.012*eta*eta*eta);

   source = - term * (1. - eta/eta_0);
   dS[eqn] = - term/d;

   return source;
}
```

2.3.27.10. Hooking a Prandtl Number UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_PRANDTL_K` is interpreted (Interpreting UD-Fs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `user_pr_k`) will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. See Hooking `DEFINE_PRANDTL` UDFs (p. 477) for details.

2.3.27.11. `DEFINE_PRANDTL_O`

2.3.27.12. Description

You can use `DEFINE_PRANDTL_O` to specify Prandtl numbers for specific dissipation (ω in the k-ω model).

2.3.27.13. Usage

`DEFINE_PRANDTL_O` (name, c, t)

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index that identifies the cell on which the Prandtl number function is to be applied.
`Thread *t`	Pointer to cell thread.

Function returns

`real`

There are three arguments to `DEFINE_PRANDTL_O`: `name`, `c`, and `t`. You supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the `real` value for the specific dissipation Prandtl number to the solver.

2.3.27.14. Example

```
/* Specifying a Constant Specific Dissipation Prandtl Number */
#include "udf.h"

DEFINE_PRANDTL_O(user_pr_o,c,t)
{
   real pr_o;
   pr_o = 2.;
   return pr_o;
}
```

2.3.27.15. Hooking a Prandtl Number UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_PRANDTL_O` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `user_pr_o`) will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. See Hooking `DEFINE_PRANDTL` UDFs (p. 477) for details.

2.3.27.16. `DEFINE_PRANDTL_T`

2.3.27.17. Description

You can use `DEFINE_PRANDTL_T` to specify Prandtl numbers that appear in the temperature equation diffusion term.

2.3.27.18. Usage

```
DEFINE_PRANDTL_T (name, c, t)
```

Argument Type	Description
symbol name	UDF name.
cell_t c	Index that identifies the cell on which the Prandtl number function is to be applied.
Thread *t	Pointer to cell thread.

Function returns

```
real
```

There are three arguments to DEFINE_PRANDTL_T: name, c, and t. You supply name, the name of the UDF. c and t are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the real value for the temperature Prandtl number to the solver.

2.3.27.19. Example

```
/* Specifying a Constant Temperature Prandtl Number */
#include "udf.h"

DEFINE_PRANDTL_T(user_pr_t,c,t)
{
    real pr_t;
    pr_t = 0.85;
    return pr_t;
}
```

2.3.27.20. Hooking a Prandtl Number UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_PRANDTL_T is interpreted (Interpreting UD-Fs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first DEFINE macro argument (for example, user_pr_t) will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. See Hooking DEFINE_PRANDTL UDFs (p. 477) for details.

2.3.27.21. DEFINE_PRANDTL_T_WALL

2.3.27.22. Description

You can use DEFINE_PRANDTL_T_WALL to specify Prandtl numbers for thermal wall functions.

2.3.27.23. Usage

```
DEFINE_PRANDTL_T_WALL (name, c, t)
```

Argument Type	Description
symbol name	UDF name.

Argument Type	Description
`cell_t c`	Index that identifies the cell on which the Prandtl number function is to be applied.
`Thread *t`	Pointer to cell thread.

Function returns

`real`

There are three arguments to DEFINE_PRANDTL_T_WALL: `name`, `c`, and `t`. You supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the `real` value for the thermal wall function Prandtl number to the solver.

2.3.27.24. Example

```
/*************************************************************
  Specifying a constant thermal wall function Prandtl number
  ********************************************************* **/
#include "udf.h"

DEFINE_PRANDTL_T_WALL(user_pr_t_wall,c,t)
{
    real pr_t_wall;
    pr_t_wall = 0.85;
    return pr_t_wall;
}
```

2.3.27.25. Hooking a Prandtl Number UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_PRANDTL_T_WALL is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first DEFINE macro argument (for example, `user_pr_t_wall`) will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. See Hooking DEFINE_PRANDTL UDFs (p. 477) for details.

2.3.28. DEFINE_PROFILE

2.3.28.1. Description

You can use DEFINE_PROFILE to define a custom boundary profile or cell zone condition that varies as a function of spatial coordinates or time. Some of the variables you can customize are:

• velocity, pressure, temperature, turbulence kinetic energy, turbulence dissipation rate

• mass flux

• target mass flow rate as a function of physical flow time

• species mass fraction (species transport)

• volume fraction (multiphase models)

- wall thermal conditions (temperature, heat flux, heat generation rate, heat transfer coefficients, and external emissivity, and so on)

- shell layer heat generation rate

- wall roughness conditions

- wall shear and stress conditions

- porosity

- porous resistance direction vector

- wall adhesion contact angle (VOF multiphase model)

- source terms

- fixed variables

Note that `DEFINE_PROFILE` allows you to modify only a single value for wall heat flux. Single values are used in the explicit source term which Ansys Fluent does not linearize. If you want to linearize your source term for wall heat flux and account for conductive and radiative heat transfer separately, you will need to use `DEFINE_HEAT_FLUX` to specify your UDF.

Some examples of boundary profile UDFs are provided below. For an overview of the Ansys Fluent solution process which shows when a `DEFINE_PROFILE` UDF is called, refer to Figure 1.2: Solution Procedure for the Pressure-Based Segregated Solver (p. 14), Figure 1.3: Solution Procedure for the Pressure-Based Coupled Solver (p. 15), and Figure 1.4: Solution Procedure for the Density-Based Solver (p. 16).

2.3.28.2. Usage

`DEFINE_PROFILE` (name, t, i)

Argument Type	Description
symbol name	UDF name.
Thread *t	Pointer to thread on which boundary condition is to be applied.
int i	Index that identifies the variable that is to be defined. i is set when you hook the UDF with a variable in a boundary conditions dialog box through the graphical user interface. This index is subsequently passed to your UDF by the Ansys Fluent solver so that your function knows which variable to operate on.

Function returns

void

There are three arguments to `DEFINE_PROFILE`: `name`, `t`, and `i`. You supply `name`, the name of the UDF. `t` and `i` are variables that are passed by the Ansys Fluent solver to your UDF.

While `DEFINE_PROFILE` is usually used to specify a profile condition on a boundary face zone, it can also be used to specify, or fix, flow variables that are held constant during computation in a cell zone. See Fixing the Values of Variables in the User's Guide for more information on fixing values in a cell zone boundary condition. For these cases, the arguments of the macro will change accordingly.

Note that unlike source term and property UDFs, profile UDFs (defined using `DEFINE_PROFILE`) are *not* called by Ansys Fluent from within a loop on threads in the boundary zone. The solver passes only the pointer to the thread associated with the boundary zone to the `DEFINE_PROFILE` macro. Your UDF will need to do the work of looping over all of the faces in the thread, computing the face value for the boundary variable, and then storing the value in memory. Ansys Fluent has provided you with a face looping macro to loop over all faces in a thread (`begin_f_loop...`). See Additional Macros for Writing UDFs (p. 305) for details.

`F_PROFILE` is typically used along with `DEFINE_PROFILE` and is a predefined macro supplied by Ansys Fluent. `F_PROFILE` stores a boundary condition in memory for a given face and thread and is nested within the face loop as shown in the examples below. It is important to note that the index `i` that is an argument to `DEFINE_PROFILE` is the same argument to `F_PROFILE`. `F_PROFILE` uses the thread pointer `t`, face identifier `f`, and index `i` to set the appropriate boundary face value in memory. See Set Boundary Condition Value (`F_PROFILE`) (p. 331) for a description of `F_PROFILE`. Note that in the case of porosity profiles, you can also use `C_PROFILE` to define those types of functions. See the example UDFs provided below.

In multiphase cases a `DEFINE_PROFILE` UDF may be called more than once (particularly if the profile is used in a mixture domain thread). If this must be avoided, then add the prefix `MP_` to the UDF name. The function will then be called only once even if it is used for more than one profile.

If your UDF uses solver storage, it is recommended that you include a NULL storage check; this can prevent an abnormal termination of Fluent if your UDF is called before the flow is initialized (that is, when the storages have not yet been allocated by the solver). For example, you could define the density in the following manner:

```
if(NNULLP(THREAD_STORAGE(thread, SV_DENSITY)))
  density = F_R(f,thread);
else
  density = 1.225; /*A reasonable value to prevent abnormal termination if flow is not initialized*/
```

2.3.28.3. Example 1 - Pressure Profile

The following UDF, named `pressure_profile`, generates a parabolic pressure profile according to the equation

$$p(y) = 1.1 \times 10^5 - 0.1 \times 10^5 \left(\frac{y}{0.0745} \right)^2$$

Note that this UDF assumes that the mesh is generated such that the origin is at the geometric center of the boundary zone to which the UDF is to be applied. y is 0.0 at the center of the inlet and extends to ± 0.0745 at the top and bottom of the inlet. The source code can be interpreted or compiled in Ansys Fluent.

```
/*********************************************************************
 UDF for specifying steady-state parabolic pressure profile boundary
 profile for a turbine vane
 *********************************************************************/

#include "udf.h"

DEFINE_PROFILE(pressure_profile,t,i)
{
   real x[ND_ND];     /* this will hold the position vector */
   real y;
   face_t f;
   begin_f_loop(f,t)
     {
      F_CENTROID(x,f,t);
      y = x[1];
      F_PROFILE(f,t,i) = 1.1e5 - y*y/(.0745*.0745)*0.1e5;
      }
   end_f_loop(f,t)
}
```

The function named `pressure_profile` has two arguments: `t` and `i`. `t` is a pointer to the face's thread, and `i` is an integer that is a numerical label for the variable being set within each loop.

Within the function body variable `f` is declared as a face. A one-dimensional array `x` and variable `y` are declared as `real` data types. Following the variable declarations, a looping macro is used to loop over each face in the zone to create a profile, or an array of data. Within each loop, `F_CENTROID` returns the value of the face centroid (array `x`) for the face with index `f` that is on the thread pointed to by `t`. The y coordinate stored in `x[1]` is assigned to variable `y`, and is then used to calculate the pressure. This value is then assigned to `F_PROFILE` which uses the integer `i` (passed to it by the solver, based on your selection of the UDF as the boundary condition for pressure in the **Pressure Inlet** dialog box) to set the pressure face value in memory.

2.3.28.4. Example 2 - Velocity, Turbulent Kinetic Energy, and Turbulent Dissipation Rate Profiles

In the following example, `DEFINE_PROFILE` is used to generate profiles for the x velocity, turbulent kinetic energy, and dissipation rate, respectively, for a 2D fully-developed duct flow. Three separate UDFs named `x_velocity`, `k_profile`, and `dissip_profile` are defined. These functions are concatenated in a single C source file and can be interpreted or compiled in Ansys Fluent.

The 1/7th power law is used to specify the x velocity component:

$$v_x = v_{x,free} \left(\frac{y}{\delta} \right)^{1/7}$$

A fully-developed profile occurs when δ is one-half the duct height. In this example, the mean x velocity is prescribed and the peak (free-stream) velocity is determined by averaging across the channel.

The turbulent kinetic energy is assumed to vary linearly from a near-wall value of

$$k_{nw} = \frac{u_\tau^2}{\sqrt{C_\mu}}$$

to a free-stream value of

$$k_{inf} = 0.002 u_{free}^2$$

The dissipation rate is given by

$$\varepsilon = \frac{C_\mu^{3/4}\left(k^{3/2}\right)}{\ell}$$

where the mixing length ℓ is the minimum of κy and 0.085 δ. (κ is the von Karman constant = 0.41.)

The friction velocity and wall shear take the forms:

$$u_\tau = \sqrt{\tau_w / \rho}$$

$$\tau_w = \frac{f \rho u_{free}^2}{2}$$

The friction factor is estimated from the Blasius equation:

$$f = 0.045\left(\frac{u_{free}\delta}{\nu}\right)^{-1/4}$$

```
/**********************************************************************
  Concatenated UDFs for fully-developed turbulent inlet profiles
 **********************************************************************/

#include "udf.h"

#define YMIN 0.0        /* constants */
#define YMAX 0.4064
#define UMEAN 1.0
#define B 1./7.
#define DELOVRH 0.5
#define VISC 1.7894e-05
#define CMU 0.09
#define VKC 0.41


/* profile for x-velocity */


DEFINE_PROFILE(x_velocity,t,i)
{
   real y, del, h, x[ND_ND], ufree;  /* variable declarations */
   face_t f;

   h = YMAX - YMIN;
   del = DELOVRH*h;
   ufree = UMEAN*(B+1.);

   begin_f_loop(f,t)
     {
        F_CENTROID(x,f,t);
        y = x[1];
        if (y <= del)
          F_PROFILE(f,t,i) = ufree*pow(y/del,B);
        else
          F_PROFILE(f,t,i) = ufree*pow((h-y)/del,B);
     }
   end_f_loop(f,t)
}

/* profile for kinetic energy */
```

```
DEFINE_PROFILE(k_profile,t,i)
{
    real y, del, h, ufree, x[ND_ND];
    real ff, utau, knw, kinf;
    face_t f;

    h = YMAX - YMIN;
    del = DELOVRH*h;
    ufree = UMEAN*(B+1.);
    ff = 0.045/pow(ufree*del/VISC,0.25);
    utau=sqrt(ff*pow(ufree,2.)/2.0);
    knw=pow(utau,2.)/sqrt(CMU);
    kinf=0.002*pow(ufree,2.);

    begin_f_loop(f,t)
      {
          F_CENTROID(x,f,t);
          y=x[1];

            if (y <= del)
              F_PROFILE(f,t,i)=knw+y/del*(kinf-knw);
            else
              F_PROFILE(f,t,i)=knw+(h-y)/del*(kinf-knw);
      }
      end_f_loop(f,t)
}

/* profile for dissipation rate */

DEFINE_PROFILE(dissip_profile,t,i)
{
    real y, x[ND_ND], del, h, ufree;
    real ff, utau, knw, kinf;
    real mix, kay;
    face_t f;

    h = YMAX - YMIN;
    del = DELOVRH*h;
    ufree = UMEAN*(B+1.);
    ff = 0.045/pow(ufree*del/VISC,0.25);
    utau=sqrt(ff*pow(ufree,2.)/2.0);
    knw=pow(utau,2.)/sqrt(CMU);
    kinf=0.002*pow(ufree,2.);

    begin_f_loop(f,t)
      {
          F_CENTROID(x,f,t);
          y=x[1];

          if (y <= del)
            kay=knw+y/del*(kinf-knw);
          else
            kay=knw+(h-y)/del*(kinf-knw);

          if (VKC*y < 0.085*del)
            mix = VKC*y;
          else
            mix = 0.085*del;

          F_PROFILE(f,t,i)=pow(CMU,0.75)*pow(kay,1.5)/mix;
      }
      end_f_loop(f,t)
}
```

2.3.28.5. Example 3 - Fixed Velocity UDF

In the following example DEFINE_PROFILE is used to fix flow variables that are held constant during computation in a cell zone. Three separate UDFs named fixed_u, fixed_v, and fixed_ke

are defined in a single C source file. They specify fixed velocities that simulate the transient startup of an impeller in an impeller-driven mixing tank. The physical impeller is simulated by fixing the velocities and turbulence quantities using the fix option in Ansys Fluent. See Fixing the Values of Variables in the User's Guide for more information on fixing variables.

```
/**********************************************************************
   Concatenated UDFs for simulating an impeller using fixed velocity
 **********************************************************************/

#include "udf.h"

#define FLUID_ID 1
#define ua1 -7.1357e-2
#define ua2 54.304
#define ua3 -3.1345e3
#define ua4 4.5578e4
#define ua5 -1.9664e5

#define va1 3.1131e-2
#define va2 -10.313
#define va3 9.5558e2
#define va4 -2.0051e4
#define va5 1.1856e5

#define ka1 2.2723e-2
#define ka2 6.7989
#define ka3 -424.18
#define ka4 9.4615e3
#define ka5 -7.7251e4
#define ka6 1.8410e5

#define da1 -6.5819e-2
#define da2 88.845
#define da3 -5.3731e3
#define da4 1.1643e5
#define da5 -9.1202e5
#define da6 1.9567e6


DEFINE_PROFILE(fixed_u,t,i)
{
    cell_t c;
    real x[ND_ND];
    real r;

    begin_c_loop(c,t)
       {
/* centroid is defined to specify position dependent profiles */

        C_CENTROID(x,c,t);
        r =x[1];
        F_PROFILE(c,t,i) =
            ua1+(ua2*r)+(ua3*r*r)+(ua4*r*r*r)+(ua5*r*r*r*r);
       }
    end_c_loop(c,t)
}


DEFINE_PROFILE(fixed_v,t,i)
{
    cell_t c;
    real x[ND_ND];
    real r;
    begin_c_loop(c,t)
       {
/* centroid is defined to specify position dependent profiles*/

        C_CENTROID(x,c,t);
        r =x[1];
```

```
            F_PROFILE(c,t,i) =
            va1+(va2*r)+(va3*r*r)+(va4*r*r*r)+(va5*r*r*r*r);
        }
      end_c_loop(c,t)
}
DEFINE_PROFILE(fixed_ke,t,i)
{
    cell_t c;
    real x[ND_ND];
    real r;
    begin_c_loop(c,t)
        {
/* centroid is defined to specify position dependent profiles*/

        C_CENTROID(x,c,t);
        r =x[1];
        F_PROFILE(c,t,i) =
        ka1+(ka2*r)+(ka3*r*r)+(ka4*r*r*r)+(ka5*r*r*r*r)+(ka6*r*r*r*r*r);
        }
    end_c_loop(c,t)
}
```

2.3.28.6. Example 4 - Wall Heat Generation Rate Profile

The following UDF, named `wallheatgenerate`, generates a heat generation rate profile for a planar conduction wall. After it has been interpreted or compiled, you can activate this UDF in the **Wall** boundary conditions dialog box in Ansys Fluent.

```
/* Wall Heat Generation Rate Profile UDF */

#include "udf.h"

DEFINE_PROFILE(wallheatgenerate,thread,i)
{
    real source = 0.001;
    face_t f;
    begin_f_loop(f,thread)
        F_PROFILE(f,thread,i) = source;
    end_f_loop(f,thread)
}
```

2.3.28.7. Example 5 - Beam Direction Profile at Semi-Transparent Walls

The following UDF, named `q_nx`, where x is the direction vector i, j, k, specifies the beam direction normal to every face on the cylinder. After it has been interpreted or compiled, you can activate this UDF in the **Wall** boundary conditions dialog box in Ansys Fluent.

```
/* Beam Direction Profile UDF at Semi-Transparent Walls */

#include "udf.h"

DEFINE_PROFILE(q_ni, t, position)
{
    real A[3], e_n[3];
    face_t f;
    real At;
    begin_f_loop(f, t)
        {
        F_AREA(A, f, t);
        At = NV_MAG(A);
        NV_VS(e_n,=,A,/,At);
        F_PROFILE(f, t, position) = -e_n[0];
        }
    end_f_loop(f, t)
```

```
   }

   DEFINE_PROFILE(q_nj, t, position)
   {
      real A[3], e_n[3];
      face_t f;
      real At;
      begin_f_loop(f, t)
         {
           F_AREA(A, f, t);
           At = NV_MAG(A);
           NV_VS(e_n,=,A,/,At);
           F_PROFILE(f, t, position) = -e_n[1];
         }
      end_f_loop(f, t)
   }


   DEFINE_PROFILE(q_nk, t, position)
   {
      real A[3], e_n[3];
      face_t f;
      real At;
      begin_f_loop(f, t)
         {
           F_AREA(A, f, t);
           At = NV_MAG(A);
           NV_VS(e_n,=,A,/,At);
           F_PROFILE(f, t, position) = -e_n[2];
         }
      end_f_loop(f, t)

   }
```

2.3.28.8. Example 6 - Viscous Resistance Profile in a Porous Zone

You can either use `F_PROFILE` or `C_PROFILE` to define a viscous resistance profile in a porous zone. Below are two sample UDFs that demonstrate the usage of `F_PROFILE` and `C_PROFILE`, respectively. Note that porosity functions are hooked to Ansys Fluent in the **Porous Zone** tab in the appropriate **Fluid** cell zone conditions dialog box.

The following UDF, named `vis_res`, generates a viscous resistance profile in a porous zone. After it has been interpreted or compiled and loaded, you can activate this UDF in the **Fluid** cell zone condition dialog box in Ansys Fluent.

```
/* Viscous Resistance Profile UDF in a Porous Zone that utilizes F_PROFILE*/

#include "udf.h"

DEFINE_PROFILE(vis_res,t,i)
{
   real x[ND_ND];
   real a;
   cell_t c;
   begin_c_loop(c,t)
      {
        C_CENTROID(x,c,t);
        if(x[1] < (x[0]-0.01))
            a = 1e9;
        else
            a = 1.0;
        F_PROFILE(c,t,i) = a;
      }
   end_c_loop(c,t)
```

```
}

/* Viscous Resistance Profile UDF in a Porous Zone that utilizes C_PROFILE*/

#include "udf.h"

DEFINE_PROFILE(porosity_function, t, nv)
{
  cell_t c;
  begin_c_loop(c,t)
      C_PROFILE(c,t,nv) = USER INPUT;
  end_c_loop(c,t)
}
```

2.3.28.9. Example 7 - Porous Resistance Direction Vector

The following UDF contains profile functions for two porous resistance direction vectors that use C_PROFILE. These profiles can be hooked to corresponding direction vectors under **Porous Zone** in the **Fluid** cell zone condition dialog box.

```
/* Porous Resistance Direction Vector Profile that utilizes C_PROFILE*/

 #include "udf.h"

DEFINE_PROFILE{dir1, t, nv}
{
  cell_t c;
  begin_c_loop(c,t)
      C_PROFILE(c,t,nv) = USER INPUT1;
  end_c_loop(c,t)
}

DEFINE_PROFILE{dir2, t, nv}
{
  cell_t c;
  begin_c_loop(c,t)
      C_PROFILE(c,t,nv) = USER INPUT2;
  end_c_loop(c,t)
}
```

2.3.28.10. Example 8 - Target Mass Flow Rate UDF as a Function of Physical Flow Time

For some unsteady problems, it is desirable that the target mass flow rate be a function of the physical flow time. This boundary condition can be applied using a DEFINE_PROFILE UDF. The following UDF, named tm_pout2, adjusts the mass flow rate from $1.00 kg/s$ to $1.35 kg/s$ when the physical time step is greater than 0.2 seconds. After it has been interpreted or compiled, you can activate this UDF in the **Pressure Outlet** boundary condition dialog box in Ansys Fluent by selecting the **Specify target mass-flow rate** option, and then choosing the UDF name from the corresponding drop-down list.

Important:

Note that the mass flow rate profile is a function of time and only one constant value should be applied to all zone faces at a given time.

```
/* UDF for setting target mass flow rate in pressure-outlet    */
 /* at t0.2 sec the target mass flow rate set to 1.00 kg/s     */
 /* when t0.2 sec the target mass flow rate will change to 1.35 kg/s */
```

```
#include "udf.h"
DEFINE_PROFILE(tm_pout2, t, nv)
{
   face_t f ;
   real flow_time = RP_Get_Real("flow-time");
   if (flow_time < 0.2)
      {
         printf("Time   = %f sec. \n",flow_time);
         printf("Targeted mass-flow rate set at 1.0 kg/s \n");
         begin_f_loop(f,t)
           {
              F_PROFILE(f,t,nv) = 1.0 ;
           }
         end_f_loop(f,t)
      }
   else
      {
         printf("Time    = %f sec. \n",flow_time);
         printf("Targeted mass-flow rate set at 1.35 kg/s \n") ;

         begin_f_loop(f,t)
           {
            F_PROFILE(f,t,nv) = 1.35 ;
           }
         end_f_loop(f,t)
  } }
```

2.3.28.11. Example 9 - Mass Flow Rate UDF for a Mass-Flow Inlet or Mass-Flow Outlet

This UDF is used to provide a time-varying specification of the mass flow rate. This boundary condition can be applied using a `DEFINE_PROFILE` UDF. The following UDF, named `mass_flow`, initially specifies a mass flow rate of 3.0 kg/s for the first 10 milliseconds, then increases it to 4.0 kg/s for the next 10 milliseconds, and after that specifies 5.0 kg/s. The macro CURRENT_TIME looks up the current flow time. Since this is a built-in RP variable, it is available to all nodes. It is more efficient to access it once and store the value in a local variable, rather than accessing it for every face in the face loop.

After it has been interpreted or compiled, you can activate this UDF in the **Mass-Flow Inlet** or **Mass-Flow Outlet** boundary condition dialog box in Ansys Fluent by selecting the UDF from the **Mass Flow Rate** drop-down list.

```
#include "udf.h"

  DEFINE_PROFILE(mass_flow,th,i)
 {
   face_t f;
   real flow_time = CURRENT_TIME;
   begin_f_loop(f,th)
     {
       if(flow_time <= 0.01)
          F_PROFILE(f,th,i) = 3.0;
       else if(flow_time <=0.02)
          F_PROFILE(f,th,i) = 4.0;
       else
          F_PROFILE(f,th,i) = 5.0;
     }
   end_f_loop(f,th);
 }
```

2.3.28.12. Hooking a Boundary Profile UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_PROFILE` is interpreted (Interpreting UD-Fs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `vis_res`) will become visible and selectable in the appropriate boundary condition dialog box (for example, the **Velocity Inlet** dialog box), cell zone condition dialog box, and **Shell Conduction Layers** dialog box in Ansys Fluent. See Hooking `DEFINE_PROFILE` UDFs (p. 478) for details.

2.3.29. `DEFINE_PROPERTY` UDFs

2.3.29.1. Description

You can use `DEFINE_PROPERTY` to specify a custom material property in Ansys Fluent for single-phase and multiphase flows. When you are writing a user-defined mixing law UDF for a mixture material, you will need to use special utilities to access species material properties. These are described below. If you want to define a custom mass diffusivity property when modeling species transport, you must use `DEFINE_DIFFUSIVITY` instead of `DEFINE_PROPERTY`. See `DEFINE_DIFFUSIVITY` (p. 60) for details on `DEFINE_DIFFUSIVITY` UDFs. For an overview of the Ansys Fluent solution process which shows when a `DEFINE_PROPERTY` UDF is called, refer to Figure 1.2: Solution Procedure for the Pressure-Based Segregated Solver (p. 14), Figure 1.3: Solution Procedure for the Pressure-Based Coupled Solver (p. 15), and Figure 1.4: Solution Procedure for the Density-Based Solver (p. 16).

Some of the properties you can customize using `DEFINE_PROPERTY` are:

- density (as a function of temperature)

- viscosity

- thermal conductivity

- absorption and scattering coefficients

- laminar flame speed

- rate of strain

- user-defined mixing laws for density, viscosity, and thermal conductivity of mixture materials

- partially-premixed unburnt properties for unburnt density, unburnt temperature, unburnt specific heat, and unburnt thermal diffusivity

---

Important:

If you would like to use a UDF to define specific heat properties, you must use the DEFINE_SPECIFIC_HEAT, as described in DEFINE_SPECIFIC_HEAT (p. 154).

Important:

Note that when you specify a user-defined density function for a compressible liquid flow application, you must also include a speed of sound function in your model. Compressible liquid density UDFs can be used in the pressure-based solver and for single phase, multiphase mixture and cavitation models, only. See the example below for details.

For Multiphase Flows

- surface tension coefficient (VOF model)

- cavitation parameters including surface tension coefficient and vaporization pressure (Mixture, cavitation models)

- heat transfer coefficient (Mixture model)

- particle or droplet diameter (Mixture model)

- speed of sound function (Mixture, cavitation models)

- density (as a function of pressure) for compressible liquid flows only (Mixture, cavitation models)

- granular temperature and viscosity (Mixture, Eulerian models)

- granular bulk viscosity (Eulerian model)

- granular conductivity (Eulerian model)

- frictional pressure and viscosity (Eulerian model)

- frictional modulus (Eulerian model)

- elasticity modulus (Eulerian model)

- radial distribution (Eulerian model)

- solids pressure (Eulerian, Mixture models)

- diameter (Eulerian, Mixture models)

- saturation temperature

2.3.29.2. Usage

`DEFINE_PROPERTY (name, c, t)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *t`	Pointer to cell thread on which the property function is to be applied.

Function returns

`real`

There are three arguments to `DEFINE_PROPERTY`: `name`, `c`, and `t`. You supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the `real` property *only* for a single cell and return it to the solver.

Note that like source term UDFs, property UDFs (defined using `DEFINE_PROPERTY`) are called by Ansys Fluent from within a loop on cell threads. The solver passes all of the variables needed to allow a `DEFINE_PROPERTY` UDF to define a custom material, since properties are assigned on a cell basis. Consequently, your UDF will *not* need to loop over cells in a zone since Ansys Fluent is already doing it.

2.3.29.3. Auxiliary Utilities

Some commonly-used auxiliary utilities for custom property UDFs are described below. They are `generic_property`, `MATERIAL_PROPERTY`, `THREAD_MATERIAL`, and `mixture_species_loop`.

`generic_property` is a general purpose function that returns the `real` value for the given property id for the given thread material. It is defined in `prop.h` and is used only for species properties.

The following `Property_ID` variables are available:

- `PROP_rho`, density

- `PROP_mu`, viscosity

- `PROP_ktc`, thermal conductivity

`generic_property (name, c, t, prop, id, T)`

Argument Type	Description
`symbol name`	Function name.
`cell_t c`	Cell index.
`Thread *t`	Pointer to cell thread on which property function is to be applied.

Argument Type	Description
Property *prop	Pointer to property array for the thread material that can be obtained through the macro MATER-IAL_PROPERTY(m) See below.
Property_ID id	Property ID of the required property you want to define a custom mixing law for (for example, PROP_ktc for thermal conductivity). See below for list of variables.
real T	Temperature at which the property is to be evaluated (used only if a polynomial method is specified).

Function returns

real

MATERIAL_PROPERTY is defined in materials.h and returns a real pointer to the Property array prop for the given material pointer m.

MATERIAL_PROPERTY (m)

Argument Type	Description
Material *m	Material pointer.

Function returns

real

THREAD_MATERIAL is defined in threads.h and returns real pointer m to the Material that is associated with the given cell thread t.

Important:

Note that in previous versions of Ansys Fluent, THREAD_MATERIAL took two arguments (t, i), but now only takes one (t).

THREAD_MATERIAL (t)

Argument Type	Description
Thread *t	Pointer to cell thread.

Function returns

real

mixture_species_loop is defined in materials.h and loops over all of the species for the given mixture material.

mixture_species_loop (m,sp,i)

Argument Type	Description
`Material *m`	Material pointer.
`Material *sp`	Species pointer.
`int i`	Species index.

Function returns

`real`

2.3.29.4. Example 1 - Temperature-dependent Viscosity Property

The following UDF, named `cell_viscosity`, generates a variable viscosity profile to simulate solidification. The function is called for every cell in the zone. The viscosity in the warm ($T > 288$ K) fluid has a molecular value for the liquid (5.5×10^{-3} kg/m-s), while the viscosity for the cooler region ($T < 286$ K) has a much larger value (1.0 kg/m-s). In the intermediate temperature range (286 K $\leq T \leq$ 288 K), the viscosity follows a linear profile that extends between the two values given above:

$$\mu = 143.2135 - 0.49725T \tag{2.10}$$

This model is based on the assumption that as the liquid cools and rapidly becomes more viscous, its velocity will decrease, thereby simulating solidification. Here, no correction is made for the energy field to include the latent heat of freezing. The source code can be interpreted or compiled in Ansys Fluent.

```
/**********************************************************************
   UDF that simulates solidification by specifying a temperature-
   dependent viscosity property
 **********************************************************************/
#include "udf.h"

DEFINE_PROPERTY(cell_viscosity,c,t)
{
  real mu_lam;
  real temp = C_T(c,t);
  if (temp > 288.)
    mu_lam = 5.5e-3;
  else if (temp > 286.)
    mu_lam = 143.2135 - 0.49725 * temp;
  else
    mu_lam = 1.;
  return mu_lam;
}
```

The function `cell_viscosity` is defined on a cell. Two real variables are introduced: `temp`, the value of `C_T(c,t)`, and `mu_lam`, the laminar viscosity computed by the function. The value of the temperature is checked, and based upon the range into which it falls, the appropriate value of `mu_lam` is computed. At the end of the function the computed value for the viscosity (`mu_lam`) is returned to the solver.

2.3.29.5. Example 2 - User-defined Mixing Law for Thermal Conductivity

You can use `DEFINE_PROPERTY` to define custom user-defined mixing laws for density, viscosity, and conductivity of mixture materials. In order to access species material properties your UDF will need to utilize auxiliary utilities that are described above.

The following UDF, named `mass_wtd_k`, is an example of a mass-fraction weighted conductivity function. The UDF utilizes the `generic_property` function to obtain properties of individual species. It also makes use of `MATERIAL_PROPERTY` and `THREAD_MATERIAL`.

```
/************************************************************************
    UDF that specifies a custom mass-fraction weighted conductivity
*************************************************************************/
#include "udf.h"

DEFINE_PROPERTY(mass_wtd_k,c,t)
{
   real sum = 0.; int i;
   Material *sp;
   real ktc;
   Property *prop;
   mixture_species_loop(THREAD_MATERIAL(t),sp,i)
      {
        prop = (MATERIAL_PROPERTY(sp));
        ktc = generic_property(c,t,prop,PROP_ktc,C_T(c,t));
        sum += C_YI(c,t,i)*ktc;
      }
   return sum;
}
```

2.3.29.6. Example 3 - Surface Tension Coefficient UDF

`DEFINE_PROPERTY` can also be used to define a surface tension coefficient UDF for the multiphase VOF model. The following UDF specifies a surface tension coefficient as a quadratic function of temperature. The source code can be interpreted or compiled in Ansys Fluent.

```
/************************************************************
 Surface Tension Coefficient UDF for the multiphase VOF Model
*************************************************************/

#include "udf.h"
DEFINE_PROPERTY(sfc,c,t)
{
   real T = C_T(c,t);
   return 1.35 – 0.004*T + 5.0e-6*T*T;
}
```

2.3.29.7. Example 4 - Density Function for Compressible Liquids

Liquid density is not a constant but is instead a function of the pressure field. In order to stabilize the pressure solution for compressible flows in Ansys Fluent, an extra term related to the speed of sound is needed in the pressure correction equation. Consequently, when you want to define a custom density function for a compressible flow, your model must also include a speed of sound function. Although you can direct Ansys Fluent to calculate a speed of sound function by choosing one of the available methods (for example, piecewise-linear, polynomial) in the **Create/Edit Materials** dialog box, as a general guideline you should define a speed of sound function along with your density UDF using the formulation:

$$\sqrt{\left(\frac{\partial p}{\partial \rho}\right)}$$

For simplicity, it is recommended that you concatenate the density and speed of sound functions into a single UDF source file.

The following UDF source code example contains two concatenated functions: a density function named `superfluid_density` that is defined in terms of pressure and a custom speed of sound function named `sound_speed`.

```
/********************************************************************
   Density and speed of sound UDFs.
 ********************************************************************/
#include "udf.h"

#define BMODULUS 2.2e9
#define rho_ref 1000.0
#define p_ref 101325

DEFINE_PROPERTY(superfluid_density, c, t)
{
    real rho;
    real p, dp;
    p = C_P(c,t) + op_pres;
    dp = p-p_ref;
    rho = rho_ref/(1.0-dp/BMODULUS);
    return rho;
}

DEFINE_PROPERTY(sound_speed, c,t)
{
    real a;
    real p, dp;
    p = C_P(c,t) + op_pres;
    dp = p-p_ref;  a = sqrt(BMODULUS*(1.-dp/BMODULUS)/rho_ref);
    return a;
}
```

2.3.29.8. Hooking a Property UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_PROPERTY` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `sound_speed`) will become visible and selectable in dialog boxes in Ansys Fluent. See Hooking `DEFINE_PROPERTY` UDFs (p. 483) for details.

2.3.30. `DEFINE_REACTING_CHANNEL_BC`

2.3.30.1. Description

You can use `DEFINE_REACTING_CHANNEL_BC` to specify user-defined inlet conditions at any reacting channel inlet. Typically this UDF is used for manifolds where the flow in two (or more) reacting channels mix in a manifold into one (or more) reacting channels. During execution, the `DEFINE_REACTING_CHANNEL_BC` is called for every channel iteration. The UDF is available only if the reacting channel model is enabled.

2.3.30.2. Usage

`DEFINE_REACTING_CHANNEL_BC` (name, i, group)

Argument Type	Description
`symbol name`	UDF name.
`int i`	ID of the reacting channel group.

Argument Type	Description
`Channel_Group *group`	Pointer to structure of reacting channel group.

Function returns

`void`

There are three arguments to `DEFINE_REACTING_CHANNEL_BC`: `name`, `i`, `group`. You supply `name`, the name of the UDF, while `i` and `group` are variables that are passed by the Ansys Fluent solver to your UDF. The variable `i` is the integer ID of the reacting channel group. The last argument is `group`, which is a pointer to the reacting channel group structure. The members of the structure `group` are described in the header file `sg_reacting_channel.h`.

2.3.30.3. Example

The following UDF, named `tube3_bc_from_1_and_2`, specifies the inlet conditions for a reacting channel. In this UDF example, the inlet conditions (mass-flow rate, pressure, temperature, and species mass fractions) of the reacting channel, tube 3, are defined using the mass-flow weighted outlet states from the reacting channels, tubes 1 and 2.

```
/**********************************************************************
 * User-defined function to specify the inlet boundary conditions at a reacting channel
 * using mass weighted average outlet variables from two other reacting channels.
 * In this UDF, the user defined inlet conditions for temperature, mass flow rate,
 * species mass fraction, and pressure are applied
 **********************************************************************/

#include "udf.h"
#include "sg_reacting_channel.h"
#define MAX_GROUPS 3   /* maximum number of groups defined in the interface*/
#define total_tubes 3   /* total number of wall zones in all groups used as reacting channel*/

static void outlet_average_variables(int *group_ids, int num_groups,Channel_Group *group)
{
    if(num_groups > N_channel_groups-1 || NULLP(group_ids))
      {
        Message("incorrect number of groups\n");
        return;
      }
    else
      {
        Material *m =group->m;   /* Material of any group, specified in the interface*/
        int i,id,j,ns,nspe=MIXTURE_NSPECIES(m);
        real mf,sum_temp,sum_mf,sum_press;
        cxboolean group_found = FALSE;
        Channel_Tube tube;
        real *sum_yi = (real *) CX_Malloc(sizeof(real)*nspe);
        spe_loop(ns,nspe)
          sum_yi[ns] =0.;
        sum_temp =0.;
        sum_press =0.;
        sum_mf=0.;

        for(j=0;j<total_tubes;j++)
          /* Total number of wall zones selected in all groups in the interface in reacting channel model*/
          {
            group_found = FALSE;
            tube = channel_tubes[j];  /* structure for any wall zone,
                                        which is specified as reacting channel wall*/
            id = tube.group_id;   /* id of the group to which this wall zone belongs*/
            for(i=0;i<num_groups;i++)   /* loop over all groups*/
              if (id == group_ids[i])
```

```
                    {
                       group_found = TRUE;    /* the current wall zone is valid reacting channel zone
                                                and belongs to a group*/
                       break;
                    }
              if(!group_found) continue ;
              i = tube.n_bands-1;    /* last grid point of the 1D grid of reacting channel wall zone*/
              mf = tube.group->mf;    /* mass flow rate of the group*/
              sum_mf += mf;
              sum_temp += mf*tube.bands[i].T;    /* temperature at the last grid point,
                                                  i.e. at the exit of the current channel*/
              sum_press += mf*tube.bands[i].pressure;    /* similar to temperature, and other variables
                                                          at exit of current channel*/
              spe_loop(ns,nspe)
                sum_yi[ns] += mf*tube.bands[i].yi[ns];
            }
        if(sum_mf > SMALL)
          {
              group->mf = sum_mf;    /* here group is the current group, where udf is applied,
                                      the mass flow rate is sum of  mf from all other groups*/
              group->temp = sum_temp/sum_mf;    /* the temperature, pressure and mass fraction at this
                                                 group are mass weighted average from exit of other groups*/
              group->pressure = sum_press/sum_mf;
              spe_loop(ns,nspe)
                group->yi[ns] = sum_yi[ns]/sum_mf;
          }                CX_Free(sum_yi);
        }
 }
/* By default, the group will use conditions provided in the interface of the
 * group in reacting channel model set up.
 * The UDF can modify few or all inlet conditions.
 * Any inlet conditions provided by the udf will overwrite the default conditions
 * provided in the interface for any given group */


DEFINE_REACTING_CHANNEL_BC(tube3_bc_from_1_and_2,i,group)
 {
    int group_ids[MAX_GROUPS];
    int num_groups =0;
    group_ids[0] = 0;    /* id of the group 0 specified in gui or tui */
    group_ids[1] = 1;
    num_groups =2;    /* total number of groups from where the averaging is to be done,
                       in this case, averaged variables from 2 groups are used
                       to specify the inlet conditions for the third group*/
    outlet_average_variables(group_ids,num_groups,group);
 }
```

2.3.30.4. Hooking a Reacting Channel Solver UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_REACTING_CHANNEL_BC is compiled (Compiling UDFs (p. 399)), the name that you specified in the DEFINE macro argument (for example, tube3_bc_from_1_and_2) will become visible and selectable in the **Reacting Channel Model** dialog box in Ansys Fluent. See Hooking DEFINE_REACTING_CHANNEL_SOLVER UDFs (p. 484) for details.

2.3.31. DEFINE_REACTING_CHANNEL_SOLVER

2.3.31.1. Description

You can use DEFINE_REACTING_CHANNEL_SOLVER to specify a user-defined heat transfer coefficient or heat flux at the reacting channel walls. During execution, the DEFINE_REACT-

ING_CHANNEL_SOLVER is called for every channel iteration. The UDF is available only if the reacting channel model is enabled.

2.3.31.2. Usage

DEFINE_REACTING_CHANNEL_SOLVER (name, params, inlet_state, dist, dt, wall_temp, wall_hf, compute_htc)

Argument Type	Description
symbol name	UDF name.
int *params	Pointer to array of current channel parameters.
real *inlet_state	Pointer to array of inlet species composition, temperature and pressure.
real *dist	Pointer to array of axial distance of discrete points from channel inlet.
real *dt	Pointer to array of time step of integration at discrete points.
real *wall_temp	Pointer to array of channel wall temperature at discrete points.
real *wall_hf	Pointer to array of channel wall heat flux at discrete points
cxboolean *compute_htc	Pointer to a flag. Set TRUE only if heat transfer coefficient is computed with the UDF.

Function returns

void

There are eight arguments to DEFINE_REACTING_CHANNEL_SOLVER: name, params, inlet_state, dist, dt, wall_temp, wall_hf, compute_htc. You supply name, the name of the UDF, while params, inlet_state, dist, dt, wall_temp, wall_hf, and compute_htc are variables that are passed by the Ansys Fluent solver to your UDF. The variable params contains four integer values specifying the number of species, the number of discrete points along the channel axis, the index of the current channel, and the index of the current channel group. The array inlet_state has N +2 elements, where N is the number of species in the channel material. The first N element of the array contains the mass fraction of the species at the channel inlet. The next two elements are temperature and pressure at the channel inlet. The variables dist, dt, wall_temp, and wall_hf have M elements, where M is the number of discrete points specified along the axis of the channel from the channel inlet to the exit. Your UDF will need to set the Boolean variable compute_htc to TRUE if you want to set the heat transfer coefficient to the reacting channel wall using this UDF or set it to FALSE if you want to set the reacting channel wall heat flux. Your UDF will need to set the value of wall_hf with the heat transfer coefficient if the flag is true or the heat flux from the reacting channel if the flag is false.

2.3.31.3. Example

The following UDF, set_channel_htc, specifies the heat transfer coefficient at the reacting channel wall. The function must be executed as a compiled UDF in Ansys Fluent.

```
#include "udf.h"
#include "sg_reacting_channel.h"

/*

   There are following two main structures "defined in sg_reacting_channel.h"

   Channel_Group *channel_groups; //channel groups, containing information about inlet conditions,
   material, model setting etc.
   Channel_Tubes *channel_tubes ' // wall zones, selected as reacting channels, information regarding
   grid, heat flux, temperature and all other variables stored on this


   params is a pointer to integer array containing channel parameters. The array has following
   four values stored

   no of species = params[0];
   no of grid points in channel = params[1]
   id of channel = params[2];
   id of the group to which this channel belong = params[3];

   By just having id of channel, we can get all the information regarding a channel as following:


   id_channel = params[2];
   Channel_Tube tube =  channel_tubes[id_chanel];
   Channel_Group *group = tube.group;  //tube belongs to the group


   1. Getting the group variables like pressure, temperature, mass flow rate etc from the group
      obtained above. All these are inlet conditions.

   real pressure = group->pressure.
   real temp = group->temp;
   real mf = group->mf;
   real yi[ns] = group->yi[ns];

   2. The variables like velocity, diameter, rho etc are either available at each grid point
      of the tube or calculated at each grid point as follows

->Each tube has many grid points along the axial direction. Each grid point is called as bands.
  All solution variables are stored in bands.

        int nbands = params[1];

        For nth grid point, starting from inlet of the tube.

        Material *m = group->m ; //mixture material of the reacting channel group
        real temp = tube.bands[n].T
        yk[ns] = tube.bands[n].yi[ns];
        pressure = tube.bands[n].pressure;



        //band is any grid point of the channel

        rho = tube.bands[n].rho;
        band_ktc = Thermal_Conductivity(0, NULL, m, temp, 0., yk,xk,0.);
                  //xk is the mole fraction, yk is the mass fraction
        band_dia = 2.*sqrt(MAX(0., channel_tubes[i].bands[j].area/M_PI));
                  //area of the bands is available and stored, diamete is calculated.
        band_mu =  Viscosity(0,NULL,m,temp,rho,yk,xk,0.);
        band_vel = tube.bands[n].vel;
```

```
     */

DEFINE_REACTING_CHANNEL_SOLVER(set_channel_htc,params,inlet_state,dist,dt,wall_temp,wall_hf,compute_htc)
{
    int g_pts, id_c = params[2],j, ns,nspecies= params[0];
    real Nu,Re,vel,channel_dia,visc,Prt,rho,ktc;
    real xk[MAX_SPE_EQNS],yk[MAX_SPE_EQNS];
    Channel_Tube tube =  channel_tubes[id_c];
    Channel_Group *group = tube.group;
    Material *m = group->m ;
    real p,temp,pgauge,mfrate;
    real op_pres_save;

    *compute_htc = TRUE ;
    g_pts = params[1];
    Prt=0.85;

    for (j=0;j++)
      {
        if (j == 0 || N_ITER < 1) /*first band or first iteration*/
          {
            spe_loop(ns,nspecies) /*initial conditions from the inlet bc of the group*/
              yk[ns] = group->yi[ns];
            temp = group->temp ;
          }
        else
          {
            spe_loop(ns,nspecies)
              yk[ns] = tube.bands[j-1].yi[ns];
            temp = tube.bands[j-1].T ;
          }

        mfrate = group->mf;
        p = group->pressure;
        Mole_Fraction(m,yk,xk);

        pgauge =0.;
        op_pres_save = op_pres;
        op_pres = group->pressure; /*channel inlet pressure can be different than the operating pressure*/

        rho =  Density(0,NULL,m,temp,pgauge,yk,0,0,0,0,0,0,0,0);
        channel_dia = 2.*sqrt(MAX(0., tube.bands[j].area/M_PI));
        vel = mfrate/(SMALL_S+rho*tube.bands[j].area);
        visc= Viscosity(0,NULL,m,temp,rho,yk,xk,0.);
        ktc = Thermal_Conductivity(0, NULL, m, temp, 0., yk,xk,0.);
        Re = rho*channel_dia*vel/visc;
        op_pres = op_pres_save;

        Nu=3.66;/*You can specify own correlation for Nusselt Number: For example, one formulation is given bel
#if 1
        if(Re < 3000)
          Nu=3.66;
        else
          {/*use Gnielinksi correlation for turbulent flows*/
            real fac = 0.125/ (SMALL + pow((0.79*log(Re)-1.64), 2)); /*f/8*/
            Nu = 1.2*fac*(Re - 1000)*GVAR_TURB(coeff, ke_prt)/(SMALL + (1. + 12.7 *sqrt(SMALL+fac) * (pow(GVAR_
            Nu *= pow((temp/tube.bands[j].wall_temp),0.36); /*temperature correction*/
          }
#endif
        wall_hf[j] = Nu*ktc/channel_dia;
      }

}
```

2.3.31.4. Hooking a Reacting Channel Solver UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_REACTING_CHANNEL_SOLVER` is compiled (Compiling UDFs (p. 399)), the name that you specified in the `DEFINE` macro argument (for example, `set_channel_htc`) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking `DEFINE_REACTING_CHANNEL_SOLVER` UDFs (p. 484) for details.

2.3.32. `DEFINE_RELAX_TO_EQUILIBRIUM`

2.3.32.1. Description

When the **Relax to Chemical Equilibrium** species chemistry solver option is enabled (as described in Enabling Species Transport and Reactions and Choosing the Mixture Material in the *Fluent User's Guide* for details), you can use `DEFINE_RELAX_TO_EQUILIBRIUM` to specify the characteristic time τ_{char} at each cell, over which the species react toward their chemical equilibrium state. This UDF will replace Fluent internal calculations of τ_{char} described in The Relaxation to Chemical Equilibrium Model in the *Fluent Theory Guide*.

2.3.32.2. Usage

`DEFINE_RELAX_TO_EQUILIBRIUM(name,c,t,pressure,temp,yi,tau)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *t`	Pointer to cell thread.
`real pressure`	Pressure.
`real temp`	Temperature.
`real *yi`	Pointer to array containing species mass fractions.
`real *tau`	Pointer to characteristic time τ_{char} (scalar-valued).

Function returns

`void`

There are seven arguments to `DEFINE_RELAX_TO_EQUILIBRIUM`: `name`, `c`, `t`, `pressure`, `temp`, `yi`, and `tau`. You supply `name`, the name of the UDF. The variables `c`, `t`, `pressure`, `temp`, `yi`, and `tau` are passed by the Ansys Fluent solver to your UDF and have SI units. The output of the function is the characteristic time τ_{char} for current species with units of [s] (seconds).

`DEFINE_RELAX_TO_EQUILIBRIUM` is called for all reacting fluid zones whenever the UDF is hooked to Ansys Fluent in the **User-Defined Function Hooks** dialog box.

2.3.32.3. Example

The following UDF, named `relax_tau`, sets the value of `tau` based on the mass fraction of CH$_4$.

```
/************************************************************
 Relax to Equilibrium Example UDF
************************************************************/
#include "udf.h"

DEFINE_RELAX_TO_EQUILIBRIUM(relax_tau,c,t,pres,temp,yi,tau)
{
  /*Simple function to set tau based on mass fraction of CH4*/
  if(yi[13] > 1.e-1)
    *tau = 1.e3;
  else
    *tau = 1.e-3;

}
```

2.3.32.4. Hooking a `DEFINE_RELAX_TO_EQUILIBRIUM` UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_RELAX_TO_EQUILIBRIUM` is interpreted or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `relax_tau`) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking `DEFINE_RELAX_TO_EQUILIBRIUM` UDFs (p. 486) for details.

2.3.33. `DEFINE_SBES_BF`

2.3.33.1. Description

You can use `DEFINE_SBES_BF` to specify a custom blending function for the Stress-Blended Eddy Simulation (SBES) model, which is a hybrid RANS-LES turbulence model available for the following models: baseline (BSL) k-ω, shear-stress transport (SST) k-ω, and transition SST. Using a UDF allows you to define the blending function zonally for cases where the division between RANS and LES is clear from the geometry and the flow physics. See Stress-Blended Eddy Simulation (SBES) in the Theory Guide for details.

2.3.33.2. Usage

`DEFINE_SBES_BF` (name, c, t)

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index that identifies the cell on which the SBES blending function is to be applied.
`Thread *t`	Pointer to cell thread.

Function returns

`real`

There are three arguments to `DEFINE_SBES_BF`: `name`, `c`, and `t`. You supply `name`, which is the calling name of the UDF. The arguments `c` and `t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the real value of f_{SDES} for the SBES blending function to the solver.

2.3.33.3. Example

A zonal approach is demonstrated in the following example. Depending on the x-coordinate, the blending function is 0.0 (LES mode) or 1.0 (RANS mode).

```
#include "udf.h"
DEFINE_SBES_BF(user_SBES_bf, c, t)
  {
    real bf_value;
    real xc[ND_ND];
    C_CENTROID(xc,c,t);
    if (xc[0] > 2.0)
      bf_value = 0.0;
    else
      bf_value = 1.0;
    return bf_value;
  }
```

2.3.33.4. Hooking an SBES Blending Function UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_SBES_BF` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the `DEFINE` macro argument (for example, `user_SBES_bf`) will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. See Hooking `DEFINE_SBES_BF` UDFs (p. 487) for details.

2.3.34. `DEFINE_SCAT_PHASE_FUNC`

2.3.34.1. Description

You can use `DEFINE_SCAT_PHASE_FUNC` to specify the radiation scattering phase function for the discrete ordinates (DO) model. The function computes two values: the fraction of radiation energy scattered from direction i to direction j, and the forward scattering factor.

2.3.34.2. Usage

`DEFINE_SCAT_PHASE_FUNC` (name, cosine, f)

Argument Type	Description
`symbol name`	UDF name.
`real cosine`	Cosine of the angle between directions i and j.
`real *f`	Pointer to the location in memory where the real forward scattering factor is stored.

Function returns

```
real
```

There are three arguments to `DEFINE_SCAT_PHASE_FUNC`: `name`, `cosine`, and `f`. You supply `name`, the name of the UDF. `cosine` and `f` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the `real` fraction of radiation energy scattered from direction i to direction j and return it to the solver. Note that the solver computes and stores a scattering matrix for each material by calling this function for each unique pair of discrete ordinates.

2.3.34.3. Example

In the following example, a number of UDFs are concatenated in a single C source file. These UDFs implement backward and forward scattering phase functions that are cited by Jendoubi et al. [5] (p. 713). The source code can be interpreted or compiled in Ansys Fluent.

```c
/******************************************************************
   UDFs that implement backward and forward scattering
   phase functions as cited by Jendoubi et al.
 ******************************************************************/

#include "udf.h"

DEFINE_SCAT_PHASE_FUNC(ScatPhiB2,c,fsf)
{
   real phi=0;
   *fsf = 0;
   phi = 1.0 - 1.2*c + 0.25*(3*c*c-1);
   return (phi);
}

DEFINE_SCAT_PHASE_FUNC(ScatPhiB1,c,fsf)
{
   real phi=0;
   *fsf = 0;
   phi = 1.0 - 0.56524*c + 0.29783*0.5*(3*c*c-1) +
      0.08571*0.5*(5*c*c*c-3*c) + 0.01003/8*(35*c*c*c*c-30*c*c+3) +
      0.00063/8*(63*c*c*c*c*c-70*c*c*c+15*c);
   return (phi);
}

DEFINE_SCAT_PHASE_FUNC(ScatPhiF3,c,fsf)
{
   real phi=0;
   *fsf = 0;
   phi = 1.0 + 1.2*c + 0.25*(3*c*c-1);
   return (phi);
}

DEFINE_SCAT_PHASE_FUNC(ScatPhiF2,c,fsf)
{
   real phi=0;
   real coeffs[9]={1,2.00917,1.56339,0.67407,0.22215,0.04725,
         0.00671,0.00068,0.00005};
   real P[9];
   int i;
   *fsf = 0;
   P[0] = 1;
   P[1] = c;
   phi = P[0]*coeffs[0] + P[1]*coeffs[1];
   for(i=1;i<7;i++)
   {
      P[i+1] = 1/(i+1.0)*((2*i+1)*c*P[i] - i*P[i-1]);
      phi += coeffs[i+1]*P[i+1];
   }
   return (phi);
}
```

```
DEFINE_SCAT_PHASE_FUNC(ScatIso,c,fsf)
{
    *fsf=0;
    return (1.0);
}
```

2.3.34.4. Hooking a Scattering Phase UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_SCAT_PHASE_FUNCTION is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name that you specified in the DEFINE macro argument (for example, ScatPhiB) will become visible and selectable in the **Create/Edit Materials** dialog box in Ansys Fluent. See Hooking DEFINE_SCAT_PHASE_FUNC UDFs (p. 491) for details.

2.3.35. DEFINE_SOLAR_INTENSITY

2.3.35.1. Description

You can use the DEFINE_SOLAR_INTENSITY macro to define direct solar intensity or diffuse solar intensity UDFs for the solar load model. See Modeling Thermal Energy in the User's Guide for more information on the solar load model.

Important:

Note that solar intensity UDFs are used with the Solar Model, which is available only for the 3D geometries in Ansys Fluent.

2.3.35.2. Usage

DEFINE_SOLAR_INTENSITY (name, sum_x, sun_y, sun_z, S_hour, S_minute)

Argument Type	Description
symbol name	UDF name.
real sun_x	x component of the sun direction vector.
real sun_y	y component of the sun direction vector.
real sun_z	z component of the sun direction vector.
int S_hour	Time in hours.
int S_minute	Time in minutes.

Function returns

real

There are six arguments to DEFINE_SOLAR_INTENSITY: name, sun_x, sun_y, sun_z, S_hour, and S_minute. You provide the name of your user-defined function. The variables sun_x, sun_y, sun_z, S_hour, and S_minute are passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the direct or diffuse solar irradiation and return the real value (in w/m^2) to the solver.

2.3.35.3. Example

The following source code contains two UDFs: `sol_direct_intensity` computes the direct solar irradiation and returns it to the Ansys Fluent solver, and `sol_diffuse_intensity` computes the diffuse solar irradiation.

```
#include "udf.h"

 DEFINE_SOLAR_INTENSITY(sol_direct_intensity,sun_x,sun_y,sun_z,hour,minute)
 {
    real intensity;
    intensity = 1019;
    printf("solar-time=%d intensity=%e\n", minute, intensity);
    return intensity;
 }

 DEFINE_SOLAR_INTENSITY(sol_diffuse_intensity,sun_x,sun_y,sun_z,hour,minute)
 {
    real intensity;
    intensity = 275;
    printf("solar-time=%d intensity-diff=%e\n", minute, intensity);
    return intensity;
 }
```

2.3.35.4. Hooking a Solar Intensity UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_SOLAR_INTENSITY` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name that you specified (for example, `sol_direct_intensity`) in the `DEFINE` macro argument will become visible and selectable for **Direct Solar Irradiation** and **Diffuse Solar Irradiation** in the **Radiation Model** dialog box in Ansys Fluent. Note that the solar load model must be enabled. See Hooking `DEFINE_SOLAR_IN-TENSITY` UDFs (p. 492) for details.

2.3.36. `DEFINE_SOLIDIFICATION_PARAMS`

2.3.36.1. Description

You can use `DEFINE_SOLIDIFICATION_PARAMS` to specify user-defined mushy zone parameters and back diffusion parameters. See Modeling Solidification and Melting in the User's Guide for more information on the solidification and melting model.

> **Important:**
>
> The back diffusion parameter is only used if the **Back Diffusion** option is enabled in the **Solidification and Melting** dialog box.

2.3.36.2. Usage

`DEFINE_SOLIDIFICATION_PARAMS (name, c, t, Amush, Gamma)`

Argument Type	Description
symbol name	UDF name.
Int c	Cell Index.

Argument Type	Description
`Thread*`	Cell thread pointer.
`real*Amush`	Pointer to real having mushy zone constant.
`Real*Gamma`	Pointer to real for back diffusion parameter.

Function returns

`void`

2.3.36.3. Example

The following UDF, named `user_soild_params`, specifies the mushy zone and uses the Clyne Kurz model for the back diffusion parameter.

```
#include "udf.h"
 DEFINE_SOLIDIFICATION_PARAMS(user_solid_params, c,t,Amush,Gamma)
 {
   real alpha ;
   real local_solidification_time = 1. ;
   real sec_arm_spacing = 5e-5 ;
   real solid_diff = 1e-10 ; /*solid diffusivity m2/s */
   *Amush = 1e+6 ;
   alpha = 4. * solid_diff*local_solidification_time/SQR(sec_arm_spacing);
   *Gamma = 2.*alpha/(1.+2.*alpha);
 }
```

2.3.36.4. Hooking a Solidification Parameter UDF in Ansys Fluent

After the UDF has been interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (eg., `user_solid_params`) will become visible in the **Solidification and Melting** dialog box in Ansys Fluent. See Hooking `DEFINE_SOLIDIFICATION_PARAMS` UDFs (p. 494) for details.

2.3.37. `DEFINE_SOOT_MASS_RATES`

2.3.37.1. Description

You can use `DEFINE_SOOT_MASS_RATES` to specify a user-defined soot nucleation, surface growth, and oxidation rates for soot mass fraction equation. The UDF is only available if the **Moss-Brookes** or **Moss-Brookes-Hall** soot model is enabled.

2.3.37.2. Usage

`DEFINE_SOOT_MASS_RATES (name, c, t, Pollut, Pollut_Par, Soot, rates_linear, rates_nonlinear, index)`

Argument Type	Description
`symbol name`	UDF name
`cell_t c`	Cell index
`Thread *t`	Pointer to cell thread on which the Soot mass rates is to be applied

Argument Type	Description
`Pollut_Cell *Pollut`	Pointer to the data structure that contains the common data at each cell
`Pollut_Parameter *Pollut_Par`	Pointer to the data structure that contains auxiliary data
`Soot *soot`	Pointer to the data structure that contains data specific to the Soot mode.
`real *rates_lin-ear`	Array to provide the linear components of soot mass rates
`real *rates_non-linear`	Array to provide the nonlinear components of soot mass rates
`int index`	Integer value which tells whether the udf is called for nucleation, surface growth or oxidation rates

Function returns

`void`

There are nine arguments to `DEFINE_SOOT_MASS_RATES`: name, c, t, Pollut, Pollut_Par, Soot, rates_linear, rates_nonlinear, and index. You supply name, the name of the UDF. Variables c, t, Pollut, Pollut_par, Soot, rates_linear, rates_nonlinear, and index are passed by the Ansys Fluent solver to your function. A `DEFINE_SOOT_MASS_RATES` function does not output a value. The calculated soot nuclei rates will be stored in the arrays `rates_linear` and `rates_nonlinear`.

The soot mass rates consist of explicit and implicit parts and can be expressed as follows:

$$Mass\ Rates = A * B$$

where A is an explicit part of any mass rate (such as oxidation and surface growth) that can be a function of any variable(s) other than soot mass fraction and soot nuclei concentration, and B is an implicit part of the nuclei rates that represents terms involving soot mass fraction and soot nuclei concentration. The term A is stored in the `rates_linear` array, while the term B is stored in the `rates_nonlinear` array.

The units of soot mass rate are $kg/m^3/s$.

2.3.37.3. Example: Soot Mass Rate

The following compiled UDF, named `user_soot_mass_rates`, is used to provide the user-defined soot mass rate.

```
/********************************************************************************
Following UDF is implementation of Fenimore-Jones soot oxidation model using UDF
********************************************************************************/

#include "udf.h"
#include "sg_pollut.h"
#define NNORM 1.e+15
#define soot_dens 1800
#define MAXSOOT 0.01
```

```
/* user defined soot mass rates to be filled in arrays "rates_linear" and "rates_nonlinear" */
/* User defined soot mass rates have to be in following form:
  Rate = A*B,
  Where B is function of soot mass fraction and/or soot nuclie
  and A is function of any variables other than soot mass fraction and nuclei.
This break is done to ensure that the soot rates can be stored in PDF tables apriori */


/* If any soot rates returned can be written in terms of only linear part (A)
and/or only non-inear part (B), then this can be done in the following manner: */

/*
  case 1 : Only A is required
  rates_linear[index] = A
  rates_nonlinear[index] = 1.
  case 2: Only B is required
  rates_linear[index] = 1.
  rates_nonlinear[index] = B
*/

/*
  Return from the UDF are:
 rates_linear[index] = A; Linear part
    rates_nonlinear[index] = B; Nonlinear part
*/

DEFINE_SOOT_MASS_RATES(user_soot_mass_rates,c,t,Pollut,Pollut_Par,Soot,rates_linear,rates_nonlinear,
index)
{
 switch (index)
 {
 case UDF_MASS_NUC: /*index=0 for user defined nucleation rate in soot mass fraction equation */
  break;
 case UDF_MASS_SGS: /*index=1 for user defined surface growth rate in soot mass fraction equation */
  break;
 case UDF_MASS_OXID: /*index=2 for user defined oxidation rate in soot mass fraction equation */
  {
   real term1;
   real oh_conc;
   real temp = Pollut->fluct.temp;
   real rho = POLLUT_GASDEN(Pollut);
   real onethird = 1. / 3.;
   real twothird = 2. / 3.;
   real soot_yi, nuc;
   real o_eq = 0;
   real c_omega= 105.8125, coeff= 0.04, oxid_const=1;
   oh_conc = get_oheq(Pollut, Soot->oh_mode, o_eq)*1.e-03; /*kmol/m3*/
   term1 = sqrt(temp)*pow(M_PI*NNORM, onethird)*pow(6. / soot_dens, twothird);
   rates_linear[UDF_MASS_OXID] = -c_omega * coeff*oxid_const*oh_conc*rho*term1;
   if (NNULLP(t)) /*This check is must*/
   {
    soot_yi = C_POLLUT(c, t, EQ_SOOT);
    nuc = C_POLLUT(c, t, EQ_NUCLEI);
    rates_nonlinear[UDF_MASS_OXID] = pow(nuc, onethird)*pow(soot_yi, twothird);
   }
  }
 break;
 default:
  Error("Wrong index for user defined soot nucleation terms\n");
  break;
 }
}
```

2.3.37.4. Hooking a Soot Mass Rate UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_SOOT_MASS_RATES` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument

(for example, `user_soot_mass_rates`) will become visible and selectable in the **User Defined Soot Mass Rates** drop-down list of the **Soot Model** dialog box in Ansys Fluent. See Hooking DEFINE_SOOT_MASS_RATES UDFs (p. 497) for details.

2.3.38. DEFINE_SOOT_MOM_RATES

2.3.38.1. Description

You can use `DEFINE_SOOT_MOM_RATES` to specify user-defined soot nucleation, surface growth, and oxidation rates for soot moment equations. The UDF is available only for the **Method of Moments** soot model.

2.3.38.2. Usage

DEFINE_SOOT_MOM_RATES (name, c, t, Nnuc, Rnuc, NCnuc, Nsurf, Rsurf, Dsurf)

Argument Type	Description
symbol name	UDF name
cell_t c	Cell index
Thread *t	Pointer to cell thread on which the Method of Moments (MOM) rates is to be applied
int *Nnuc	Pointer to the number of nucleation reactions
real *Rnuc	Pointer to the nucleation rates for each nucleation reaction
real *NCnuc	Pointer to the number of carbon atoms in each nucleating species
int *Nsurf	Pointer to the number of surface reactions, including growth and oxidation
real *Rsurf	Pointer to the rates for surface growth and oxidation
real *Dsurf	Pointer to the number of carbon atoms added or removed due to surface growth or oxidation of the soot particles

Function returns

void

There are nine arguments to `DEFINE_SOOT_MOM_RATES`: name, c, t, Nnuc, Rnuc, NCnuc, Nsurf, Rsurf, and Dsurf. You supply name, the name of the UDF. c, t are passed by the Ansys Fluent solver to your function. Your UDF will need to specify variables Nnuc, Rnuc, NCnuc, Nsurf, Rsurf, and Dsurf. Note that for surface growth and oxidation, you need to specify the number of carbon atoms added (for surface growth) or removed (a negative value for oxidation) for each participating species in growth and oxidation. The oxidation is negative surface growth.

A `DEFINE_SOOT_MOM_RATES` function does not output a value. All calculated rates are stored in the arrays Rnuc and Rsurf.

The default values passed by the Ansys Fluent solver to your UDF are 0 for all the arrays. If you want to specify only user-defined nucleation rates, then you need to adjust only nucleation rates arguments, and surface growths will be modeled using the Ansys Fluent default formulation. Similarly,

if you need to specify only surface growth and oxidation rates, then you need to set only the last three arguments. In this case, the nucleation will be modeled using the Ansys Fluent default formulation.

The units of nucleation rates (`Rnuc`) and surface growth or oxidation rates (`Rsurf`) are number of particles/m^3/s. The source terms for the different moment transport equations will be automatically calculated by Ansys Fluent using the user-specified nucleation and growth rates.

2.3.38.3. Example: Soot MOM Rates

The following compiled UDF, named `user_soot_mom_rates`, is used to provide the user-defined soot MOM rates. In this example, a single nucleating species C_2H_2 is used to provide the user specified nucleation rates. For surface growth, HACA-based surface growth is modeled using C_2H_2 as growth species, and oxidation is modeled using O_2 and OH.

```
#include "udf.h"
#define soot_eps 1e-15

DEFINE_SOOT_MOM_RATES(user_soot_mom_rates,c,t,Nnuc,Rnuc,NCnuc,Nsurf,Rsurf,Dsurf)
{
    Material *sp,*m=mixture_material(t->domain);
    int  ns,ic2h2,io2,ioh,ih,ih2,ih2o;
    double mw[MAX_PDF_SPECIES], rf[MAX_PDF_SPECIES],rr[MAX_PDF_SPECIES];
    double aa,bb,alpha,fac0,fac1;
    double xmolc2h2,xmolo2,xmolh2o,xmolh,xmolh2,xmoloh,act_soot_rad;
    double cell_temp = (double) C_T(c,t);
    double rgas_t = UNIVERSAL_GAS_CONSTANT * cell_temp;
    double mu1 = MAX(soot_eps, C_SOOT_MOM_CLIP(c, t, 1)/MAX(soot_eps, C_SOOT_MOM_CLIP(c, t, 0)));
    double m0  = MAX(soot_eps, C_SOOT_MOM_CLIP(c, t, 0));

    double rho = (double) C_R(c,t);

     mixture_species_loop (m, sp, ns)
         mw[ns] = MATERIAL_PROP(sp, PROP_mwi);
     ic2h2 = mixture_specie_index(m,"c2h2");
     io2   = mixture_specie_index(m,"o2");
     ih2   = mixture_specie_index(m,"h2");
     ih    = mixture_specie_index(m,"h");
     ioh   = mixture_specie_index(m,"oh");
     ih2o  = mixture_specie_index(m,"h2o");

     /*molar concentration of species. kmol/m3*/
     xmolc2h2 = Pdf_Yi(c,t,ic2h2)  * rho/mw[ic2h2];
     xmolo2   = Pdf_Yi(c,t,io2)    * rho/mw[io2]  ;
     xmolh2o  = Pdf_Yi(c,t,ih2o)   * rho/mw[ih2o] ;
     xmolh    = Pdf_Yi(c,t,ih)     * rho/mw[ih]   ;
     xmoloh   = Pdf_Yi(c,t,ioh)    * rho/mw[ioh]  ;
     xmolh2   = Pdf_Yi(c,t,ih2)    * rho/mw[ih2]  ;

    /*nucleation sources using a single step nucleation reaction of C2H2*/
     Rnuc[0]  = 54. * exp(-1.746e+8/rgas_t) * xmolc2h2 * AVOGADRO_NUMBER;   /* number/m3-s*/
    *Nnuc     = 1;  /*only 1 nucleation reaction. Integer quantity. No unit*/
     NCnuc[0] = 2;  /*number of carbon atoms in PAH. No unit*/

    /*surface growth and oxidation using HACA */

    /* Parameters for Equation 14-163 of Fluent theory guide */
    aa = 12.65 - 0.00563 * cell_temp;
    bb = -1.38 + 0.00068 * cell_temp;
    alpha = MAX(0.2, MIN(1., tanh(aa / (1.e-6 + log(mu1)) + bb))); /*is unit less*/

    /*Reaction rates of HACA mechanism. Table 14.6 of Fluent Theory Guide*/

    rf[0] = 4.2e+10 * exp(-5.4392e+7 / rgas_t); /*unit is m3/kmol-s*/
    rf[1] = 1.0e+07 * pow(cell_temp, 0.734) * exp(-5.98312e+6 / rgas_t);
```

```
    rf[2] = 2.0e+10 ;
    rf[3] = 8.0e+04 * pow(cell_temp, 1.56) * exp(-1.5899e+7 / rgas_t);
    rf[4] = 2.e+09  * exp(-3.138e+7 / rgas_t);
    rf[5] = 1.3e-03 * (1.0 / (alpha * Soot_Mom->soot_site_dens/AVOGADRO_NUMBER)) *
            sqrt(rgas_t /(2. * M_PI * 17.0));

    rr[0] = 3.9e+09  * exp(-4.6027e+07 / rgas_t);
    rr[1] = 3.68e+05 * pow(cell_temp, 1.139) * exp(-7.15464e+07 / rgas_t);
    rr[2] = 0.;
    rr[3] = 0.;

    /*Equation 14-162 of Fluent Theory Guide */

    fac0 =  MAX(SMALL_S,  rf[0] * xmolh  + rf[1] * xmoloh + rf[5] * xmoloh);
    fac1 =  MAX(SMALL_S, (rr[0] * xmolh2 + rr[1] * xmolh2o + rf[2] * xmolh + rf[3] * xmolc2h2 +
            rf[4] * xmolo2));

    /*Number of active soot particles. Equation 14-163 of Fluent Theory Guide*/
    act_soot_rad = M_PI * Soot_Mom->diam_catom * Soot_Mom->diam_catom * alpha * Soot_Mom->soot_site_dens;
    act_soot_rad *=  fac0 / (fac0 + fac1);

    /*sources due to C2H2, O2 and OH respectively*/
    *Nsurf =3 ;    /*Number of species in surface growth and oxidation  */
    Dsurf[0] =  2; /*Number of carbon atom addition/removal due to c2h2 =  2  */
    Dsurf[1] = -2; /*Number of carbon atom addition/removal due to o2   = -2. negative means removal */
    Dsurf[2] = -1; /*Number of carbon atom addition/removal due to oh   = -1  */


    /*Surface growth rate due to c2h2*/
    Rsurf[0]  =   rf[3] * act_soot_rad * xmolc2h2 *m0 ; /*unit is number/m3-s*/

    /*Oxidation rate due to o2*/
    Rsurf[1]   =  rf[4] * act_soot_rad * xmolo2 *m0 ;   /*unit is number/m3-s*/

    /*Oxidation rate due to oh*/
    Rsurf[2]   =  rf[5] * (act_soot_rad * fac1 / fac0) * xmoloh *m0 ;   /*unit is number/m3-s*/


}
```

2.3.38.4. Hooking a Soot MOM Rates UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_SOOT_MOM_RATES is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first DEFINE macro argument (for example, user_soot_mom_rates) will become visible and selectable in the **User Defined Soot MOM Rates** drop-down list of the **Soot Model** dialog box in Ansys Fluent. See Hooking DEFINE_SOOT_MOM_RATES UDFs (p. 498) for details.

2.3.39. DEFINE_SOOT_NUCLEATION_RATES

2.3.39.1. Description

You can use DEFINE_SOOT_NUCLEATION_RATES to specify a user-defined soot nucleation and coagulation rates. The UDF is available only if the **Moss-Brookes** or the **Moss-Brookes-Hall** soot model is enabled.

2.3.39.2. Usage

```
DEFINE_SOOT_NUCLEATION_RATES (name, c, t, Pollut, Pollut_Par, Soot,
rates_linear, rates_nonlinear, index)
```

Argument Type	Description
symbol name	UDF name
cell_t c	Cell index
Thread *t	Pointer to cell thread on which the Soot nucleation rates is to be applied
Pollut_Cell *Pollut	Pointer to the data structure that contains the common data at each cell
Pollut_Parameter *Pollut_Par	Pointer to the data structure that contains auxiliary data
Soot *soot	Pointer to the data structure that contains data specific to the Soot mode.
real *rates_linear	Array to provide the linear components of soot nucleation rates
real *rates_nonlinear	Array to provide the nonlinear components of soot nucleation rates
int index	Integer value which tells whether the udf is called for nucleation or coagulation

Function returns

void

There are nine arguments to DEFINE_SOOT_NUCLEATION_RATES: name, c, t, Pollut, Pollut_Par, Soot, rates_linear, rates_nonlinear, and index. You supply name, the name of the UDF. Variables c, t, Pollut, Pollut_par, Soot, rates_linear, rates_nonlinear, and index are passed by the Ansys Fluent solver to your function. A DEFINE_SOOT_NUCLEATION_RATES function does not output a value. The calculated soot nuclei rates will be stored in the arrays rates_linear and rates_nonlinear.

The soot nucleation rates consist of explicit and implicit parts and can be expressed as follows:

$$Nucleation\,Rates = A * B$$

where A is an explicit part of the nuclei (nucleation or coagulation) that can be a function of any variable(s) other than soot mass fraction and soot nuclei concentration, and B is an implicit part of the nuclei rates that represents terms involving soot mass fraction and soot nuclei concentration. The term A is stored in the rates_linear array, while the term B is stored in the rates_nonlinear array.

The units of nuclei rate are nuclei/m^3/s.

2.3.39.3. Example: Soot Nucleation and Coagulation Rates

The following compiled UDF, named user_soot_nuc_rates, is used to provide the user-defined soot nucleation rate.

```
/**********************************************************************
Following UDF is implementation of nucleation and coagulation terms of soot
nuclei transport equation for Moss Brookes soot Model using UDF
**********************************************************************/
```

```
#include "udf.h"
#include "sg_pollut.h"
#define NNORM 1.e+15
#define soot_dens 1800
#define MAXSOOT 0.01

/*User defined Soot nucleation rates have to be in following form:
 Rate = A*B, Where B is function of soot mass fraction and/or soot nuclie
 and A is function of any variables other than soot mass fraction and nuclei.
This break is done to ensure that the soot rates can be stored in PDF tables apriori.*/

/*
  Return from the UDF are:
 rates_linear[index] = A; Linear part
 rates_nonlinear[index] = B; Nonlinear part
*/

/*If any soot rates returned can be written in terms of only linear part (A)
and/or only nonlinear part (B), then this can be done in the following manner:*/

/*
   case 1 : Only A is required
   rates_linear[index] = A
   rates_nonlinear[index] = 1.
   case 2: Only B is required
   rates_linear[index] = 1.
   rates_nonlinear[index] = B
*/

DEFINE_SOOT_NUCLEATION_RATES(user_soot_nuc_rates,c,t,Pollut,Pollut_Par,Soot,rates_linear,
rates_nonlinear,index)
{
 real T = Pollut->fluct.temp;
 int ns;
 real xmolprec;
 double avgn = 6.022137e+26, nnorm = 1.e+15;

 switch (index)
 {
 case UDF_NUC: /* index = 0 for user defined nucleation term in nuclei equation*/
 {
  xmolprec = 0.;
  for (ns = PRECSP; ns < PRECSP + Soot->nprec; ns++)
  {
   xmolprec += MOLECON(Pollut, ns)*1.e-03;
  }
    rates_linear[UDF_NUC]=(avgn/ nnorm)*Soot->calpha*xmolprec*exp(MAX(-70, -Soot->talpha / T));
    rates_nonlinear[UDF_NUC] = 1.;
 }
 break;
 case UDF_COAG: /* index = 1 for user defined coagulation term in nuclei equation*/
 {
  real pwr4 = 1. / 6., pwr3 = 5. / 6.;
  real term1, term2;
  real rho = POLLUT_GASDEN(Pollut);
  real spsoot = MIN(MAX(Pollut->fluct.yi[IDX(SOOT)], 0.), MAXSOOT);
  real spnuclei = MAX(Pollut->fluct.yi[IDX(NUCLEI)], 0.);
  term1 = pow(24.*Pollut_Par->uni_R*1000.*T / (Soot->soot_mdens*avgn), 0.5);
  term2 = pow(6. / (M_PI*Soot->soot_mdens), pwr4);
  rates_linear[UDF_COAG] = -rho * rho*Soot->cbeta*term1*term2*pow(nnorm, pwr3);
  rates_nonlinear[UDF_COAG] = pow(spsoot, pwr4)*pow(spnuclei, (pwr3 + 1.));
 }
 break;
 default:
  Error("Wrong index for user defined soot nucleation terms\n");
  break;
 }
}
```

2.3.39.4. Hooking a Nucleation and Coagulation Rates UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_SOOT_NUCLEATION_RATES` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `user_soot_nuc_rates`) will become visible and selectable in the **User Defined Soot Nuclei Rates** drop-down list of the **Soot Model** dialog box in Ansys Fluent. See Hooking `DEFINE_SOOT_NUCLEATION_RATES` UDFs (p. 499) for details.

2.3.40. `DEFINE_SOOT_OXIDATION_RATE`

2.3.40.1. Description

You can use `DEFINE_SOOT_OXIDATION_RATE` to specify a user-defined soot oxidation rate. The UDF is available only if the **Moss-Brookes** or the **Moss-Brookes-Hall** soot model is enabled.

2.3.40.2. Usage

`DEFINE_SOOT_OXIDATION_RATE (name, c, t, Pollut, Pollut_Par, Soot, soot_oxi_rate)`

Argument Type	Description
`symbol name`	UDF name
`cell_t c`	Cell index
`Thread *t`	Pointer to cell thread on which the Soot oxidation rate is to be applied
`Pollut_Cell *Pollut`	Pointer to the data structure that contains the common data at each cell
`Pollut_Parameter *Pollut_Par`	Pointer to the data structure that contains auxiliary data
`Soot *soot`	Pointer to the data structure that contains data specific to the Soot model
`Real soot_oxi_rate`	Array to return the soot oxidation rates

Function returns

`void`

There are seven arguments to `DEFINE_SOOT_OXIDATION_RATE`: `name`, `c`, `t`, `Pollut`, `Pollut_Par`, `Soot`, and `soot_oxi_rate`. You supply `name`, the name of the UDF. Variables `c`, `t`, `Pollut`, `Pollut_par`, `Soot`, and `soot_oxi_rate` are passed by the Ansys Fluent solver to your function. A `DEFINE_SOOT_OXIDATION_RATE` function does not output a value. The calculated soot oxidation rates will be stored in the array `soot_oxi_rate`.

The soot oxidation rate consists of explicit and implicit parts and can be expressed as follows:

$$oxidation\_rate = A * B$$

where A is an explicit part of the oxidation rate that can be a function of any variable(s) other than soot mass fraction and soot nuclei concentration, and B is an implicit part of the oxidation rate that represents terms involving soot mass fraction and soot nuclei concentration. The term A is stored as the first element of the `soot_oxi_rate` array, while the term B is stored as the second element of the array.

The units of oxidation rate are kg/m^3/s

2.3.40.3. Example: Soot Oxidation Rate

The following compiled UDF, named `user_soot_oxid_rate`, is used to provide the user-defined oxidation rate.

```
/********************************************************************
User defined Soot oxidation Rate: Fenimore-Jones soot oxidation model
********************************************************************/

#include "udf.h"
#include "sg_pollut.h"
#define NNORM 1.e+15
#define soot_dens 1800
#define c_omega 105.8125 /*kg-m-kmol-1 K^-1./2. S^-1, oxidation model constant*/
#define coleff 0.04 /*collisional*/
#define oxid_const 0.015 /*oxidation rate scaling parameter*/

/*user defined soot oxidation rates to be filled in array "soot_oxi_rate"*/
/*User defined Soot oxidation rate has to be in following form:
   Rate = A*B, Where B is function of soot mass fraction and/or soot nuclie and A is function of any
   variables other than soot mass fraction and nuclei. This break is done to apriori store the soot
   formation rates in PDF tables*/

/* Store the computed soot oxidation rate inside the UDF as: */
/*   soot_oxi_rate[0] = A; */ /*Linear part*/
/*   soot_oxi_rate[1] = B; */ /*Nonlinear part*/


/*Following UDF is implementation of Fenimore-Jones soot oxidation model using UDF*/

DEFINE_SOOT_OXIDATION_RATE(user_soot_oxid_rate,c,t,Pollut,Pollut_Par,Soot,soot_oxi_rate)
{
    real term1;
    real oh_conc;
    real temp = Pollut->fluct.temp;
    real rho =  POLLUT_GASDEN(Pollut);
    real onethird = 1./3.;
    real twothird = 2./3. ;
    real soot_yi,nuc;
    real o_eq=0.;

    oh_conc = get_oheq(Pollut, Soot->oh_mode, o_eq)*1.e-03; /*kmol/m3*/
    term1 = sqrt(temp)*pow(M_PI*NNORM, onethird)*pow(6./soot_dens, twothird);

    soot_oxi_rate[0] = c_omega*coleff*oxid_const*oh_conc*rho*term1;

    if(NNULLP(t)) /*This check is must*/
      {
        soot_yi = C_POLLUT(c,t,EQ_SOOT);
        nuc = C_POLLUT(c,t,EQ_NUCLEI);
        soot_oxi_rate[1] = pow(nuc, onethird)*pow(soot_yi,twothird);
      }

}
```

2.3.40.4. Hooking a Soot Oxidation Rate UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_SOOT_OXIDATION_RATE` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `user_soot_oxid_rate`) will become visible and selectable in the **User-Defined Oxidation Rate** drop-down list of the **Soot Model** dialog box in Ansys Fluent. See Hooking `DEFINE_SOOT_OXIDATION_RATE` UDFs (p. 500) for details.

2.3.41. `DEFINE_SOOT_PRECURSOR`

2.3.41.1. Description

You can use `DEFINE_SOOT_PRECURSOR` to specify a user-defined concentration of soot precursor. During execution, the `DEFINE_SOOT_PRECURSOR` is called at the time of calculating the average soot source terms. The UDF is only available if the **Moss Brookes** soot model is enabled.

2.3.41.1.1. Usage

`DEFINE_SOOT_PRECURSOR (name, c, t, Pollut, Pollut_Par, Soot)`

Argument Type	Description
`symbol name`	UDF name
`cell_t c`	Cell index
`Thread *t`	Pointer to cell thread on which the Soot precursor is to be applied
`Pollut_Cell *Pollut`	Pointer to the data structure that contains the common data at each cell
`Pollut_Parameter *Pollut_Par`	Pointer to the data structure that contains auxiliary data
`Soot *soot`	Pointer to the data structure that contains data specific to the Soot model

Function returns

`real`

There are six arguments to `DEFINE_SOOT_PRECURSOR`: `name`, `c`, `t`, `Pollut`, `Pollut_Par`, and `Soot`. You supply `name`, the name of the UDF. Variables `c`, `t`, `Pollut`, `Pollut_par`, and `Soot` are passed by the Ansys Fluent solver to your function. A `DEFINE_SOOT_PRECURSOR` function returns the soot precursor concentration in kg-mol/m^3.

2.3.41.1.2. Example: Soot Precursor

The following compiled UDF, named `user_soot_prec` is used to provide the user-defined soot precursor concentration in kg-mol/m^3.

```
/********************************************************
User Defined Soot precursor concentration:
Return molar concentration of soot precursor in kg-mol/m3
C2H2 is used as precursor
```

```
*****************************************************/

DEFINE_SOOT_PRECURSOR(user_soot_prec,Pollut,Pollut_Par,Soot)
{
  real xmol_prec;
  real rho =  POLLUT_GASDEN(Pollut);
  real c2h2_mol_wt =26;

  xmol_prec = rho*MAX(0., Pollut->fluct.yi[C2H2])/c2h2_mol_wt;
  return xmol_prec;
}
```

2.3.41.1.3. Hooking a SOOT_PRECURSOR UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_SOOT_PRECURSOR` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `user_soot_precursor`) will become visible and selectable in the **Species Definition** group box of the **Soot Model** dialog box in Ansys Fluent. See Hooking `DEFINE_SOOT_PRECURSOR` UDFs (p. 501) for details.

2.3.42. `DEFINE_SOURCE`

2.3.42.1. Description

You can use `DEFINE_SOURCE` to specify custom source terms for the different types of solved transport equations in Ansys Fluent (except the discrete ordinates radiation model) including:

- mass

- momentum

- k, ε

- energy (also for solid zones)

- species mass fractions

- P1 radiation model

- user-defined scalar (UDS) transport

- granular temperature (Eulerian, Mixture multiphase models)

2.3.42.2. Usage

`DEFINE_SOURCE` (name, c, t, dS, eqn)

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index that identifies cell on which the source term is to be applied.
`Thread *t`	Pointer to cell thread.

Argument Type	Description
`real dS[]`	Array that contains the derivative of the source term with respect to the dependent variable of the transport equation.
`int eqn`	Equation number.

Function returns

`real`

There are five arguments to `DEFINE_SOURCE`: `name`, `c`, `t`, `dS`, and `eqn`. You supply `name`, the name of the UDF. `c`, `t`, `dS`, and `eqn` are variables that are passed by the Ansys Fluent solver to your UDF. Note that the source term derivatives may be used to linearize the source term if they enhance the stability of the solver. To illustrate this, note that the source term can be expressed, in general, as Equation 2.11 (p. 149), where ϕ is the dependent variable, A is the explicit part of the source term, and $B\phi$ is the implicit part.

$$S_\phi = A + B\phi \tag{2.11}$$

Specifying a value for B in Equation 2.11 (p. 149) can enhance the stability of the solution and help convergence rates due to the increase in diagonal terms on the solution matrix. Ansys Fluent automatically determines if the value of B that is given by you will aid stability. If it does, then Ansys Fluent will define A as $S^* - \left(\partial S / \partial \phi\right)^* \phi^*$, and B as $\left(\partial S / \partial \phi\right)^*$. If not, the source term is handled explicitly.

Your UDF will need to compute the `real` source term *only* for a single cell and return the value to the solver, but you have the choice of setting the implicit term `dS[eqn]` to $dS / d\phi$, or forcing the explicit solution of the source term by setting it equal to 0.0.

Note that like property UDFs, source term UDFs (defined using `DEFINE_SOURCE`) are called by Ansys Fluent from within a loop on cell threads. The solver passes to the `DEFINE_SOURCE` term UDF all the necessary variables to define a custom source term, since source terms are solved on a cell basis. Consequently, your UDF will *not* need to loop over cells in the thread since Ansys Fluent is already doing it.

The units on all source terms are of the form generation-rate/volume. For example, a source term for the continuity equation would have units of kg/m^3-s.

2.3.42.3. Example 1 - Source Term Addition

The following UDF, named `xmom_source`, is used to add source terms in Ansys Fluent. The source code can be interpreted or compiled. The function generates an x-momentum source term that varies with y position as

$$source = -0.5 C_2 \rho y |v_x| v_x$$

Suppose

$$source = S = -A |v_x| v_x$$

where

$$A = 0.5 C_2 \rho y$$

Then

$$\frac{dS}{dv_x} = -A|v_x| - A v_x \frac{d}{dv_x}(|v_x|)$$

The source term returned is

$$source = -A|v_x|v_x$$

and the derivative of the source term with respect to v_x (true for both positive and negative values of v_x) is

$$\frac{dS}{dv_x} = -2A|v_x|$$

```
/**********************************************************************
    UDF for specifying an x-momentum source term in a spatially
   dependent porous media
 **********************************************************************/

#include "udf.h"

#define C2 100.0

DEFINE_SOURCE(xmom_source,c,t,dS,eqn)
{
   real x[ND_ND];
   real con, source;
   C_CENTROID(x,c,t);
   con = C2*0.5*C_R(c,t)*x[1];
   source = -con*fabs(C_U(c, t))*C_U(c,t);
   dS[eqn] = -2.*con*fabs(C_U(c,t));
   return source;
}
```

2.3.42.4. Example 2 - Degassing Boundary Condition

The following source code contains UDFs that define a degassing boundary condition for flow in a bubble column reactor (BCR) or a similar device. Typically, in a BCR, gas is sparged at the bottom of a vertical cylinder filled with water. The top surface can be modeled as a free-surface where the water level rises in proportion to the volume fraction of the incoming gas. However, to avoid modeling the extra head space at the top, sometimes the free surface is replaced by a degassing boundary where the top surface of the liquid behaves as a symmetry plane, and the gas is removed as a sink. The compressibility of gas and compressibility of liquid are ignored in this case.

Using this approach, the UDFs define the degassing source as the mass sink for the gas (secondary) phase and the momentum sources associated with the mass sources in X, Y, and Z directions for both the liquid (primary) and gas (secondary) phases. The bottom surface is defined as a standard velocity inlet for the gas phase. The UDF is applied to the top surface, where bubbles leave the liquid. The sink removes all gas phase mass in the cells next to the degassing boundary during one-time step.

```
/**********************************************************************
   This UDF is an implementation of the degassing boundary condition
 **********************************************************************/

#include "udf.h"
#include "sg.h"
```

```
#include "sg_mphase.h"
#include "flow.h"
#include "mem.h"
#include "metric.h"

DEFINE_SOURCE(degassing_source, cell, thread, dS, eqn)
{
   real source;
   Thread *tm = THREAD_SUPER_THREAD(thread);
   source = -C_R(cell,thread)*C_VOF(cell,thread)/CURRENT_TIMESTEP ;
   C_UDMI(cell,tm,0) = source;
   dS[eqn] = -C_R(cell,thread)/CURRENT_TIMESTEP;
   return source;
}

DEFINE_SOURCE(x_prim_recoil, cell, tp, dS, eqn)
{
   real source;
   Thread *tm = THREAD_SUPER_THREAD(tp);
   Thread *ts;
   ts = THREAD_SUB_THREAD(tm,1);
   source = -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP*C_U(cell,tp);
  dS[eqn] =-C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP;
   return source;
}

DEFINE_SOURCE(x_sec_recoil, cell, ts, dS, eqn)
{
   real source;
   Thread *tm = THREAD_SUPER_THREAD(ts);
   source = -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP*C_U(cell,ts);
   dS[eqn] = -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP;
   return source;
}

DEFINE_SOURCE(y_prim_recoil, cell, tp, dS, eqn)
{
   real source;
   Thread *tm = THREAD_SUPER_THREAD(tp);
   Thread *ts;
   ts = THREAD_SUB_THREAD(tm,1);
   source = -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP*C_V(cell,tp);
   dS[eqn] =-C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP;
   return source;
}

 DEFINE_SOURCE(y_sec_recoil, cell, ts, dS, eqn)
{
   real source;  Thread *tm = THREAD_SUPER_THREAD(ts);
   source = -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP*C_V(cell,ts);
  dS[eqn] = -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP;
   return source;
}

DEFINE_SOURCE(z_prim_recoil, cell, tp, dS, eqn)
{
   real source;
   Thread *tm = THREAD_SUPER_THREAD(tp);
   Thread *ts;
   ts = THREAD_SUB_THREAD(tm,1);
   source = -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP*C_W(cell,tp);
   dS[eqn] =-C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP;
   return source;
}

 DEFINE_SOURCE(z_sec_recoil, cell, ts, dS, eqn)
{
   real source;
   Thread *tm = THREAD_SUPER_THREAD(ts);
   source = -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP*C_W(cell,ts);
  dS[eqn] = -C_R(cell,ts)*C_VOF(cell,ts)/CURRENT_TIMESTEP;
```

```
      return source;
   }
```

2.3.42.5. Hooking a Source UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_SOURCE` is interpreted (Interpreting UD-Fs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `xmom_source`) will become visible and selectable in the **Fluid** or **Solid** cell zone condition dialog box in Ansys Fluent. See Hooking `DEFINE_SOURCE` UDFs (p. 495) for details.

2.3.43. `DEFINE_SPARK_GEOM (R14.5 spark model)`

The `DEFINE_SPARK_GEOM` is available with the R14.5 spark model only. See the R14.5 Ansys Fluent User's Guide for information about using the R14.5 spark model in Ansys Fluent.

2.3.43.1. Description

You can use `DEFINE_SPARK_GEOM` to define custom spark kernel volume shapes.

2.3.43.2. Usage

`DEFINE_SPARK_GEOM (name,c,t)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *t`	Pointer to cell thread on which the source term is to be applied.

Function returns

`integer  inside`

There are three arguments to `DEFINE_SPARK_GEOM`: `name`, `c`, and `t`. You will supply `name`, the name of the UDF, and the variables `c` and `t` are passed by the Ansys Fluent solver into the UDF. The UDF will need to return an integer value that indicates whether or not the cell defined by the arguments `c` and `t` is within the spark kernel volume. A returned value of zero indicates that the cell is not within the spark kernel. All other values indicate that the cell is within the kernel.

2.3.43.3. Example

This example UDF is used to define three different custom shape types: a sphere, a cylinder, or a frustum.

```
#include "udf.h"
#include "sg_spark.h"

 typedef enum {
    SPHERE,
    CYLINDER,
```

```
    FRUSTUM
} Spark_Geom;

DEFINE_SPARK_GEOM(spark_geom,c,t)
{
    int inside=0;
    Spark_Geom spark_geom=FRUSTUM; /* set to chosen shape */
    Spark_Par *spark_par = getSparkPar();
    int snum=0; /* spark index */
    switch(spark_geom) {
      case SPHERE:
        {
            real rad,rad2;
            real NV_VEC(xc);
            real NV_VEC(xdiff);
            real time = CURRENT_TIME;
            real start_time = spark_par[snum].start_time;
            real duration = spark_par[snum].duration;

            /* user sphere data */
            real r0 = 0.001;    /* initial radius */
            real rf = 0.003;    /* final radius */
            real xcen[3]={0.0,0.0,0.0}; /* sphere centre */
            real dr = ABS(rf-r0);
            C_CENTROID(xc,c,t);
            NV_VV(xdiff, =, xc,-,xcen);   /* user growth rate */
            rad = r0 + (time-start_time)*dr/duration;
            rad2 = rad*rad;

            /* flag cell if inside sphere */
            if (NV_DOT(xdiff,xdiff) < rad2)
              inside = 1;
              break;
        }
      case CYLINDER:
        {
            real rad, rad2;
            real am, NV_VEC(xa);
            real cm, NV_VEC(xc);
            real time = CURRENT_TIME;
            real start_time = spark_par[snum].start_time;
            real duration = spark_par[snum].duration;

            /* user cylinder data */
            real r0 = 0.001;    /* initial radius */
            real rf = 0.003;    /* final radius */
            real x0[3]={0.0,0.0,0.0}; /* axis start */
            real x1[3]={-0.003,0.0,0.0}; /* axis end */
            real dr = ABS(rf-r0);

            /* user growth rate */
            rad = r0 + (time-start_time)*dr/duration;
            rad2 = rad*rad;

            /* compute normalized axis vector */
            NV_VV(xa,=,x1,-,x0);
            am = NV_MAG(xa);
            NV_S(xa,/=,am);
            C_CENTROID(xc,c,t);
            NV_V (xc, -=, x0);
            cm = NV_DOT(xc,xa);

            /* flag cell if inside cylinder */
            if (cm >= 0 && cm <= am)
              {
                NV_VS(xc,-=,xa,*,cm);
                if (NV_MAG2(xc) <= rad2)
                  inside = 1;
              }
          break;
        }
```

```
    case FRUSTUM:
      {
        real rad, rad0, rad1, rad2;
        real am, NV_VEC(xa);
        real cm, NV_VEC(xc);
        real time = CURRENT_TIME;
        real start_time = spark_par[snum].start_time;
        real duration = spark_par[snum].duration;

        /* user frustum data */
        real r00 = 0.001, r01 = 0.002;  /* initial radii */
        real rf0 = 0.003, rf1 = 0.004;  /* final radii */
        real x0[3]={0.0,0.0,0.0};     /* axis start */
        real x1[3]={-0.003,-0.003,0.0};  /* axis end */
        real dr0 = ABS(rf0-r00);
        real dr1 = ABS(rf1-r01);

        /* user growth rate */
        rad0 = r00 + (time-start_time)*dr0/duration;
        rad1 = r01 + (time-start_time)*dr1/duration;

       /* compute normalized axis vector */
        NV_VV(xa,=,x1,-,x0);
        am = NV_MAG(xa);
        NV_S(xa,/=,am);
        C_CENTROID(xc,c,t);
        NV_V (xc, -=, x0);
        cm = NV_DOT(xc,xa);
        rad = rad0 + cm/am * (rad1-rad0);
        rad2 = rad*rad;

        /* flag cell if inside frustum */
        if (cm >= 0 && cm <= am)
          {
            NV_VS(xc,-=,xa,*,cm);
            if (NV_MAG2(xc) <= rad2)
               inside = 1;
          }
        break;
      }
     default:
      break;
      }
    return inside;
  }
```

2.3.43.4. Hooking a Spark Geometry UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_SPARK_GEOM` is compiled (Compiling UD-Fs (p. 399)), the name that you specified in the `DEFINE` macro argument will become visible and selectable in the **Set Spark Ignition** dialog box in Ansys Fluent. See Hooking `DEFINE_SPARK_GEOM` UDFs (p. 502) for details.

2.3.44. `DEFINE_SPECIFIC_HEAT`

2.3.44.1. Description

The `DEFINE_SPECIFIC_HEAT` macro can be used to define temperature-dependent functions for specific heat and sensible enthalpy for fluid, solid and mixture materials (this does not include

DPM particles). These functions must be defined in a consistent manner, that is, the enthalpy function should be the temperature integral of the specific heat function.

Important:

This option is not available with the premixed, non-premixed and partially premixed models, and should be used as a compiled UDF only.

2.3.44.2. Usage

`DEFINE_SPECIFIC_HEAT`(name,T,Tref,h,yi)

Argument Type	Description
`symbol name`	UDF name
`real T`	Temperature for the calculation of the specific heat and enthalpy
`real Tref`	Reference temperature for the enthalpy calculation
`real *h`	Pointer to `real`
`real *yi`	Pointer to array of mass fractions of gas phase species

Function returns

`real`

There are five arguments to `DEFINE_SPECIFIC_HEAT`: name, T, Tref, h, and yi. You supply name, the name of the UDF. T and Tref are `real` variables that are passed by the Ansys Fluent solver to the UDF, and h is a pointer to `real`.

The UDF must return the `real` value of the specific heat, and set the sensible enthalpy to the value referenced by the `real` pointer h. Note that the entropy is not computed in the UDF, instead Ansys Fluent sets the entropy as $S = cp(Tmean)log(T/Tref)$, where cp is computed by the UDF at $Tmean$, and $Tmean$ is the mean logarithmic average of T and Tref.

2.3.44.3. Example

```
/***********************************************************************
   UDF that computes specific heat and sets the sensible enthalpy
   to the referenced value
 ***********************************************************************/

#include "udf.h"
DEFINE_SPECIFIC_HEAT(my_user_cp, T, Tref, h, yi)
{
  real cp=2000.;
  *h = cp*(T-Tref);
  return cp;
}
```

2.3.44.4. Hooking a Specific Heat UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_SPECIFIC_HEAT` is compiled (Compiling UDFs (p. 399)), the name that you specified in the `DEFINE` macro argument (for example, `my_user_cp`) will become visible and selectable in the **Create/Edit Materials** dialog box in Ansys Fluent. See Hooking `DEFINE_SPECIFIC_HEAT` UDFs (p. 503) for details.

2.3.45. `DEFINE_SR_RATE`

2.3.45.1. Description

You can use `DEFINE_SR_RATE` to specify a custom surface reaction rate. A custom surface reaction rate function defined using this macro will overwrite the default reaction rate (for example, `finite-rate`) that is specified in the **Create/Edit Materials** dialog box. A `DEFINE_SR_RATE` UDF is compatible with the laminar finite-rate model. It is not available with the stiff chemistry solver.

An example of a reaction rate that depends upon gas species mass fractions is provided below. Also provided is a reaction rate UDF that takes into account site species.

> **Important:**
>
> Note that the three types of surface reaction species are internally numbered with an (integer) index `i` in order

2.3.45.2. Usage

`DEFINE_SR_RATE (name,f,t,r,my,yi,rr)`

Argument Type	Description
`symbol name`	UDF name.
`face_t f`	Index that identifies a face within the given thread (or cell in the case of surface reaction in a porous zone).
`Thread *t`	Pointer to face thread on which the surface rate reaction is to be applied.
`Reaction *r`	Pointer to data structure for the reaction.
`double *mw`	Pointer to array of species molecular weights.
`double *yi`	Pointer to array of mass fractions of gas species at the surface and the coverage of site species (or site fractions).
`double *rr`	Pointer to reaction rate.

Function returns

`void`

There are seven arguments to `DEFINE_SR_RATE`: `name`, `f`, `t`, `r`, `my`, `yi`, and `rr`. You supply `name`, the name of the UDF. After your UDF is compiled and linked, the name that you have chosen for

your function will become visible and selectable in the graphical user interface in Ansys Fluent. `f`, `t`, `r`, `my`, and `yi` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to set the reaction rate to the value referenced by the `real` pointer `rr` as shown in the examples below.

2.3.45.3. Example 1 - Surface Reaction Rate Using Species Mass Fractions

The following compiled UDF, named `arrhenius`, defines a custom surface reaction rate using species mass fractions in Ansys Fluent.

```
/********************************************************************
 Custom surface reaction rate UDF
 ********************************************************************/
#include "udf.h"
/* ARRHENIUS CONSTANTS */
#define PRE_EXP 1e+15
#define ACTIVE 1e+08
#define BETA 0.0

real arrhenius_rate(real temp)
{
    return
    PRE_EXP*pow(temp,BETA)*exp(-ACTIVE/(UNIVERSAL_GAS_CONSTANT*temp));
}

/* Species numbers. Must match order in Ansys Fluent dialog box */
#define HF 0
#define WF6 1
#define H2O 2
#define NUM_SPECS 3

/* Reaction Exponents */
#define HF_EXP 2.0
#define WF6_EXP 0.0
#define H2O_EXP 0.0

#define MW_H2 2.0
#define STOIC_H2 3.0

/* Reaction Rate Routine */

real reaction_rate(cell_t c, Thread *cthread,real mw[],real yi[])

/* Note that all arguments in the reaction_rate function call in your .c
source file MUST be on the same line or a compilation error will occur */
{
    real concenHF = C_R(c,cthread)*yi[HF]/mw[HF];
    return arrhenius_rate(C_T(c,cthread))*pow(concenHF,HF_EXP);
}

DEFINE_SR_RATE(arrhenius,f,fthread,r,mw,yi,rr)
{
  *rr =
  reaction_rate(F_C0(f,fthread),THREAD_T0(fthread),mw,yi);
}
```

2.3.45.4. Example 2 - Surface Reaction Rate Using Site Species

The following compiled UDF, named `my_rate`, defines a custom surface reaction rate that takes into account site species.

```
/********************************************************************
 Custom surface reaction rate UDF
 ********************************************************************/
```

```
#include "udf.h"
DEFINE_SR_RATE(my_rate,f,t,r,mw,yi,rr)
{
  Thread *t0=t->t0;
  cell_t c0=F_C0(f,t);
  double sih4 = yi[0];   /* mass fraction of sih4 at the wall */
  double si2h6 = yi[1];
  double sih2 = yi[2];
  double h2 = yi[3];
  double ar = yi[4];   /* mass fraction of ar at the wall */
  double rho_w = 1.0, site_rho = 1.0e-6, T_w = 300.0;
  double si_s = yi[6];   /* site fraction of si_s*/
  double sih_s = yi[7];   /* site fraction of sih_s*/
  T_w = F_T(f,t);
  rho_w = C_R(c0,t0)*C_T(c0,t0)/T_w;
  sih4 *= rho_w/mw[0]; /* converting of mass fractions to molar concentrations */
  si2h6 *= rho_w/mw[1];
  sih2 *= rho_w/mw[2];
  h2 *= rho_w/mw[3];
  ar *= rho_w/mw[4];
  si_s  *= site_rho;   /* converting of site fractions to site concentrations */
  sih_s *= site_rho;
  if (STREQ(r->name, "reaction-1"))
    *rr = 100.0*sih4;
  else if (STREQ(r->name, "reaction-2"))
    *rr = 0.1*sih_s;
  else if (STREQ(r->name, "reaction-3"))
    *rr = 100*si2h6*si_s;
  else if (STREQ(r->name, "reaction-4"))
    *rr = 1.0e10*sih2;
}
```

2.3.45.5. Hooking a Surface Reaction Rate UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_SR_RATE` is interpreted (Interpreting UD-Fs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `my_rate`) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking `DEFINE_SR_RATE` UDFs (p. 505) for details.

2.3.46. `DEFINE_THICKENED_FLAME_MODEL`

2.3.46.1. Description

You can use `DEFINE_THICKENED_FLAME_MODEL` to specify the thickening factor F, efficiency factor, E, and dynamic thickening parameter Ω for the thickened flame model (TFM). This UDF can be hooked only if the thickened flame model is enabled in the **Species Model** dialog box.

2.3.46.2. Usage

`DEFINE_THICKENED_FLAME_MODEL (name,c,t,F,E,Omega)`

Argument Type	Description
`symbol name`	UDF name.
`Thread *c`	Index that identifies a cell within the given thread.

Argument Type	Description
Thread *t	Pointer to cell thread on which the TFM parameters (F, E, Omega) are to be applied.
F	Pointer to array of the thickening factor.
E	Pointer to array of the efficiency factor.
Omega	Pointer to array of the dynamic thickening factor.

Function returns

void

There are six arguments to DEFINE_THICKENED_FLAME_MODEL: name, c, t, F, E, and Omega. You supply name, the name of the UDF. After your UDF is compiled, the name that you have chosen for your function will become visible and selectable in the graphical user interface in Ansys Fluent. c, t, F, E, and Omega are variables that are passed by the Ansys Fluent solver to your UDF.

Note:

The default values of F, E, and Omega are calculated before the UDF is called, therefore none of these values necessarily need to be set in the UDF.

2.3.46.3. Example - Thickened Flame Model

In the simple example below, the DEFINE_THICKENED_FLAME_MODEL returns a fixed thickening factor $F=10$, with a unity efficiency factor (so that the effect of thickening on the turbulent flame speed is neglected), and a unity dynamic thickening parameter (so that thickening is applied everywhere in the domain, including far from the flame front).

```
 #include "udf.h"

DEFINE_THICKENED_FLAME_MODEL(user_TFM, c, t, F, E, Omega)
{
    *F = 10.;

    *E = 1.;

    *Omega = 1.;
}
```

2.3.46.4. Hooking a Thickened Flame Model UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_THICKENED_FLAME_MODEL is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the DEFINE macro argument (for example, user_TFM) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking DEFINE_THICKENED_FLAME_MODEL UDFs (p. 506) for details.

2.3.47. `DEFINE_TRANS` UDFs

The following `DEFINE` macros can be used to specify transition correlations for the Transition SST model in Ansys Fluent.

2.3.47.1. `DEFINE_TRANS_ALG_REC_COR`

2.3.47.1.1. Description

You can use **`DEFINE_TRANS_ALG_REC_COR`** to specify your own formulation of the critical Reynolds number correlation for the Algebraic Transition Model.

2.3.47.1.2. Usage

DEFINE_TRANS_ALG_REC_COR (name, c, t)

Argument Type	Description
symbol name	UDF name.
cell_t c	Index that identifies the cell on which the critical Reynolds number correlation is to be applied.
Thread *t	Pointer to cell thread.

Function returns

real

There are three arguments to these functions: **`name`**, **`c`**, and **`t`**. You specify the name of the UDF. **`c`** and **`t`** are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the real value for the critical Reynolds number correlation to the solver.

2.3.47.1.3. Example

In the following example, a zonal approach is demonstrated. Depending on the x-coordinate, the critical Reynolds number correlation is set to 10.0 or the built-in version is used.

```
#include "udf.h"
DEFINE_TRANS_ALG_REC_COR(user_trans_alg_rec_cor, c, t)
 {
    real rec_value;
    real xc[ND_ND];
    C_CENTROID(xc,c,t);
    if (xc[0]>2.0)
      rec_value = Get_Trans_Alg_Rec_Cor (c, t,
        C_R(c,t), C_MU_L(c,t), C_WALL_DIST(c,t),
        C_O(c,t), Strainrate_Mag(c,t), Rotationrate_Mag(c,t));
    else
      rec_value = 10.0;
    return rec_value;
 }
```

2.3.47.1.4. Hooking a UDF for Critical Reynolds Number to Ansys Fluent

After the UDF that you have defined using DEFINE_TRANS_ALG_REC_COR is interpreted or compiled, the name of the argument that you supplied as the DEFINE macro argument (for example, user_trans_alg_rec_cor) will become visible and selectable in the Viscous Model dialog box in the drop-down menu of the model coefficient.

2.3.47.2. `DEFINE_TRANS_FLENGTH`

2.3.47.3. Description

You can use `DEFINE_TRANS_FLENGTH` to specify the transition length for the Transition SST turbulence model.

2.3.47.4. Usage

`DEFINE_TRANS_FLENGTH (name, c, t)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index of cell on which the transition length function is to be applied.
`Thread *t`	Pointer to cell thread.

Function returns

`real`

There are three arguments to `DEFINE_TRANS_FLENGTH`: `name`, `c`, and `t`. You supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the `real` value for the transition length function to the solver.

2.3.47.5. Example

An example of a `TRANS_FLENGTH` UDF is provided at the end of this section.

2.3.47.6. Hooking a Transition Correlation UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_TRANS_FLENGTH` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the `DEFINE` macro argument (for example, `user_Flength`) will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. See Hooking `DEFINE_TRANS` UDFs (p. 508) for details.

2.3.47.7. `DEFINE_TRANS_GEOMRGH`

2.3.47.8. Description

You can use `DEFINE_TRANS_GEOMRGH` to specify the geometric roughness height for the Transition SST turbulence model. For more details on the model, see Transition SST and Rough Walls.

2.3.47.9. Usage

`DEFINE_TRANS_GEOMRGH (name, c, t)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index of cell on which the geometric roughness height is to be applied.
`Thread *t`	Pointer to cell thread.

Function returns

`real`

There are three arguments to `DEFINE_TRANS_GEOMRGH`: `name`, `c`, and `t`. You supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the `real` value for the geometric roughness height to the solver.

2.3.47.10. Example

An example of a `DEFINE_TRANS_GEOMRGH` UDF is provided at the end of this section.

2.3.47.11. Hooking a Transition Correlation UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_TRANS_GEOMRGH` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the `DEFINE` macro argument (for example, `user_Geomrgh`) will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. See Hooking `DEFINE_TRANS` UDFs (p. 508) for details.

2.3.47.12. `DEFINE_TRANS_RETHETA_C`

2.3.47.13. Description

You can use `DEFINE_TRANS_RETHETA_C` to specify the critical momentum thickness Reynolds number for the Transition SST turbulence model.

2.3.47.14. Usage

`DEFINE_TRANS_RETHETA_C (name, c, t)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index that identifies the cell on which the critical momentum thickness Reynolds number is to be applied.
`Thread *t`	Pointer to cell thread.

Function returns

`real`

There are three arguments to `DEFINE_TRANS_RETHETA_C`: `name`, `c`, and `t`. You supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the `real` value for the critical momentum thickness Reynolds number to the solver.

2.3.47.15. Example

An example of a `TRANS_RETHETA_C` UDF is provided at the end of this section.

2.3.47.16. Hooking a Transition Correlation UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_TRANS_RETHETA_C` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the `DEFINE` macro argument (for example, `user_Re_thetac`) will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. See Hooking `DEFINE_TRANS` UDFs (p. 508) for details.

2.3.47.17. `DEFINE_TRANS_RETHETA_T`

2.3.47.18. Description

You can use `DEFINE_TRANS_RETHETA_T` to specify the transition onset momentum thickness Reynolds number for the Transition SST turbulence model.

2.3.47.19. Usage

`DEFINE_TRANS_RETHETA_T` (name, c, t)

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index that identifies the cell on which the transition onset momentum thickness Reynolds number is to be applied.
`Thread *t`	Pointer to cell thread.

Function returns

```
real
```

There are three arguments to `DEFINE_TRANS_RETHETA_T`: `name`, `c`, and `t`. You supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the `real` value for the transition onset momentum thickness Reynolds number to the solver.

2.3.47.20. Example

The following functions (which are concatenated into a single C source code file) demonstrate this usage.

```
#include "udf.h"

DEFINE_TRANS_FLENGTH(user_Flength, c, t)
{
    real Flength = 31.468;
    return Flength;
}

DEFINE_TRANS_GEOMRGH(user_Geomrgh, c, t)
{
/* Dimensional value of geometric roughness height */
    real Geomrgh = 0.001;
    return Geomrgh;
}

DEFINE_TRANS_RETHETA_C(user_Re_thetac, c, t)
{
    real Re_thetac = 176.396;
    return Re_thetac;
}

DEFINE_TRANS_RETHETA_T(user_Re_thetat, c, t)
{
    real Re_thetat = 210;
    return Re_thetat;
}
```

2.3.47.21. Hooking a Transition Correlation UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_TRANS` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the `DEFINE` macro argument (for example, `user_Re_thetat`) will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. See Hooking `DEFINE_TRANS` UDFs (p. 508) for details.

2.3.48. `DEFINE_TRANSIENT_PROFILE`

2.3.48.1. Description

You can use the `DEFINE_TRANSIENT_PROFILE` macro to specify cell zone conditions that vary over time (for example, the rotation rate of a fan rotor zone at machine startup). Using this macro, you can replace the default transient profile interpolation method in Ansys Fluent or provide an analytic expression for the corresponding variable.

2.3.48.2. Usage

DEFINE_TRANSIENT_PROFILE (name, current_time)

Argument Type	Description
symbol name	UDF name.
real current_time	Current time.

Function returns

real

There are two arguments to DEFINE_TRANSIENT_PROFILE: name and current_time. You supply name, and then current_time is passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the real value of the corresponding variable, to which the function is assigned.

2.3.48.3. Example

The following UDF, named rotation_rate_ramp, computes the rotation rate of a cell zone, simulating startup behavior. The angular velocity is increased linearly in time up to a flow time of 0.1 s, after which it remains constant at a rate of 250 rad/s. The source can be interpreted or compiled.

```
#include "udf.h"

DEFINE_TRANSIENT_PROFILE(rotation_rate_ramp,time)
{
 real rotation_rate = 0.0;
if (time < 0.1)
  {
    rotation_rate = 2500.0 * time;
  }
else
  {
   rotation_rate = 250.0;
  }
 return rotation_rate;
}
```

2.3.48.4. Hooking a Transient Profile UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_TRANSIENT_PROFILE is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name that you specified in the DEFINE macro argument will become visible and selectable in the cell zone condition dialog boxes. See Hooking DEFINE_TRANSIENT_PROFILE UDFs (p. 510) for details.

2.3.49. `DEFINE_TURB_PREMIX_SOURCE`

2.3.49.1. Description

You can use `DEFINE_TURB_PREMIX_SOURCE` to customize the turbulent flame speed and source term (S_c in Equation 8.70 and $\rho \bar{S}_{Y_c}$ in Equation 8.113 in the *Fluent Theory Guide*) in the premixed combustion model and the partially premixed combustion model.

2.3.49.2. Usage

`DEFINE_TURB_PREMIX_SOURCE (name, c, t, turb_flame_speed, source)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *t`	Pointer to cell thread on which the turbulent premixed source term is to be applied.
`real *turb_flame_speed`	Pointer to the turbulent flame speed.
`real *source`	Pointer to the reaction progress source term.

Function returns

`void`

There are five arguments to `DEFINE_TURB_PREMIX_SOURCE`: `name`, `c`, `t`, `turb_flame_speed`, and `source`. You supply `name`, the name of the UDF. `c`, `t`, `turb_flame_speed`, and `source` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to set the turbulent flame speed to the value referenced by the `turb_flame_speed` pointer. It will also need to set the source term to the value referenced by the `source` pointer.

2.3.49.3. Example

The following UDF, named `turb_flame_src`, specifies a custom turbulent flame speed and source term in the premixed combustion model. The source code must be executed as a compiled UDF in Ansys Fluent.

In the standard premixed combustion model in Ansys Fluent, the mean reaction rate of the progress variable (that is, the source term) is modeled as

$$\rho S_c = \rho_u U_t \left| \nabla c \right| \tag{2.12}$$

where c is the mean reaction progress variable, ρ is the density, and U_t is the turbulent flame speed.

In the UDF example, the turbulent flame speed is modeled as

$$U_t = U_l \sqrt{1 + \left(u' / U_l \right)^2} \tag{2.13}$$

where U_l is the laminar flame speed and u' is the turbulent fluctuation. Note that the partially premixed combustion model is assumed to be enabled (see Modeling Partially Premixed Combustion in the User's Guide), so that the unburned density and laminar flame speed are available as polynomials. See Additional Macros for Writing UDFs (p. 305) for details on the NULLP, THREAD_STORAGE, and SV_VARS macros.

```
/********************************************************************
  UDF that specifies a custom turbulent flame speed and source
  for the premixed combustion model
*********************************************************************/

#include "udf.h"
#include "sg_pdf.h" /* not included in udf.h so must include here */

DEFINE_TURB_PREMIX_SOURCE(turb_flame_src,c,t,turb_flame_speed,source)
{
   real up = TRB_VEL_SCAL(c,t);
   real ut, ul, grad_c, rho_u, Xl, DV[ND_ND];
   ul = C_LAM_FLAME_SPEED(c,t);
   Xl = THREAD_VAR(t).fluid.premix_var.tdiff_u;
   rho_u = THREAD_VAR(t).fluid.premix_var.rho_u;

   if(NNULLP(THREAD_STORAGE(t,SV_PREMIXC_G)))
     {
         NV_V(DV, =, C_STORAGE_R_NV(c,t,SV_PREMIXC_G));
         grad_c = sqrt(NV_DOT(DV,DV));
     }
   ut = ul*sqrt(1. + SQR(up/ul));
   *turb_flame_speed = ut;
   *source = rho_u*ut*grad_c;
}
```

2.3.49.4. Hooking a Turbulent Premixed Source UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_TURB_PREMIX_SOURCE is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first DEFINE macro argument (for example, turb_flame_src) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking DEFINE_TURB_PREMIX_SOURCE UDFs (p. 510) for details.

2.3.50. DEFINE_TURB_SCHMIDT UDF

The DEFINE_TURB_SCHMIDT macro can be used to specify the turbulent Schmidt numbers of all transported species in Ansys Fluent, for single-phase flows.

2.3.50.1. Description

The turbulent Schmidt number, denoted Sc_t, controls the turbulent diffusion of species transported in Ansys Fluent. You can use DEFINE_TURB_SCHMIDT to specify Sc_t for each species solved.

2.3.50.2. Usage

DEFINE_TURB_SCHMIDT (name, c, t, i)

Argument Type	Description
symbol name	UDF name.

Argument Type	Description
`cell_t c`	Index of cell on which the Turbulent Schmidt number function is to be applied.
`Thread *t`	Pointer to cell thread.
`int i`	Species index.

Function returns

`real`

There are four arguments to `DEFINE_TURB_SCHMIDT`: `name`, `c`, `t` and `i`. You supply `name`, the name of the UDF. `c`, `t` and `i` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the `real` value for the turbulent Schmidt number to the solver.

2.3.50.3. Example

The following example sets Sc_t to be inversely proportional to the species index. Hence, the first species in the materials list will have the smallest turbulent diffusion, and the last species will have the largest turbulent diffusion.

```
#include "udf.h"

 DEFINE_TURB_SCHMIDT(udf_sct, c, t, i)
 {
    return 1./((real)i+1.);
 }
```

2.3.50.4. Hooking a Turbulent Schmidt Number UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_TURB_SCHMIDT` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `udf_sct` in the above example) will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. See Hooking `DEFINE_TURB_SCHMIDT` UDFs (p. 512) for details.

2.3.51. `DEFINE_TURBULENT_VISCOSITY`

2.3.51.1. Description

You can use `DEFINE_TURBULENT_VISCOSITY` to specify a custom turbulent viscosity function for the Spalart-Allmaras, k-ε, and k-ω turbulence models for single-phase applications. In addition, for 3D versions of Ansys Fluent you can specify a subgrid-scale turbulent viscosity UDF for the large eddy simulation model. For Eulerian multiphase flows, turbulent viscosity UDFs can be assigned on a per-phase basis, and/or to the mixture, depending on the turbulence model. See Table 2.9: Eulerian Multiphase Model and `DEFINE_TURBULENT_VISCOSITY` UDF Usage (p. 168) for details.

Table 2.9: Eulerian Multiphase Model and `DEFINE_TURBULENT_VISCOSITY` UDF Usage

Turbulence Model	Phase that Turbulent Viscosity UDF Is Specified On
k-ε Mixture	mixture, primary and secondary phases

Turbulence Model	Phase that Turbulent Viscosity UDF Is Specified On
k-ε Dispersed	primary and secondary phases
k-ε Per-Phase	primary and secondary phases

2.3.51.2. Usage

```
DEFINE_TURBULENT_VISCOSITY (name, c, t)
```

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *t`	Pointer to cell thread on which the turbulent viscosity is to be applied.

Function returns

`real`

There are three arguments to `DEFINE_TURBULENT_VISCOSITY`: `name`, `c`, and `t`. You supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the `real` value of the turbulent viscosity to the solver.

2.3.51.3. Example 1 - Single Phase Turbulent Viscosity UDF

The following UDF, named `user_mu_t`, defines a custom turbulent viscosity for the standard k-ε turbulence model. Note that the value of `GVAR_TURB(coeff, ke_Cmu)` in the example is defined through the graphical user interface, but made accessible to all UDFs. The source code can be interpreted or compiled in Ansys Fluent.

```
/********************************************************************
  UDF that specifies a custom turbulent viscosity for standard
  k-epsilon formulation
 ********************************************************************/

#include "udf.h"

DEFINE_TURBULENT_VISCOSITY(user_mu_t,c,t)
{
   real mu_t;
   real rho = C_R(c,t);
   real k  = C_K(c,t);
   real d  = C_D(c,t);
   mu_t = GVAR_TURB(coeff, ke_Cmu)*rho*SQR(k)/d;
   return mu_t;
}
```

2.3.51.4. Example 2 - Multiphase Turbulent Viscosity UDF

```
/********************************************************************
   Custom turbulent viscosity functions for each phase and the
   mixture in a two-phase multiphase flow
 ********************************************************************/
#include "udf.h"

DEFINE_TURBULENT_VISCOSITY(mu_t_ke_mixture, c, t)
```

```
{
    real mu_t;
    real rho = C_R(c,t);
    real k   = C_K(c,t);
    real d   = C_D(c,t);
    real cmu = GVAR_TURB(coeff, ke_Cmu);
    mu_t = rho*cmu*k*k/d;
    return mu_t;
}

DEFINE_TURBULENT_VISCOSITY(mu_t_ke_1, c, t)
{
    Thread *tm = lookup_thread_by_id(DOMAIN_SUPER_DOMAIN(THREAD_DOMAIN(t)),t->id);
    CACHE_T_SV_R (density, t, SV_DENSITY);
    CACHE_T_SV_R (mu_t, t, SV_MU_T);
    CACHE_T_SV_R (density_m, tm, SV_DENSITY);
    CACHE_T_SV_R (mu_t_m, tm, SV_MU_T);
    return density[c]/density_m[c]*mu_t_m[c];
}

DEFINE_TURBULENT_VISCOSITY(mu_t_ke_2, c, t)
{
    Thread *tm = lookup_thread_by_id(DOMAIN_SUPER_DOMAIN(THREAD_DOMAIN(t)),t->id);
    CACHE_T_SV_R (density, t, SV_DENSITY);
    CACHE_T_SV_R (mu_t, t, SV_MU_T);
    CACHE_T_SV_R (density_m, tm, SV_DENSITY);
    CACHE_T_SV_R (mu_t_m, tm, SV_MU_T);
    return density[c]/density_m[c]*mu_t_m[c];
}
```

2.3.51.5. Hooking a Turbulent Viscosity UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_TURBULENT_VISCOSITY` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the function name(s) that you specified in the `DEFINE` macro argument(s) (for example `user_mu_t` for single phase, or `mu_t_ke_mixture`, `mu_t_ke_1`, and `mu_t_ke_2` for multiphase) will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. See Hooking `DEFINE_TURBULENT_VISCOSITY` UDFs (p. 514) for details.

2.3.52. `DEFINE_VR_RATE`

2.3.52.1. Description

You can use `DEFINE_VR_RATE` to specify a custom volumetric reaction rate for a single reaction or for multiple reactions. During Ansys Fluent execution, `DEFINE_VR_RATE` is called for every reaction in every single cell. A `DEFINE_VR_RATE` UDF is compatible with the laminar finite-rate model. It is not available with the Chemkin-CFD solver.

2.3.52.2. Usage

`DEFINE_VR_RATE (name,c,t,r,mw,yi,rr,rr_t)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.

Argument Type	Description
`Thread *t`	Pointer to cell thread on which the volumetric reaction rate is to be applied.
`Reaction *r`	Pointer to data structure that represents the current reaction.
`real *mw`	Pointer to array of species molecular weights.
`real *yi`	Pointer to array of the species mass fractions.
`real *rr`	Pointer to laminar reaction rate.
`real *rr_t`	Pointer to turbulent reaction rate.

Function returns

`void`

There are eight arguments to `DEFINE_VR_RATE`: `name`, `c`, `t`, `r`, `mw`, `yi`, `rr`, and `rr_t`. You supply `name`, the name of the UDF. `c`, `t`, `r`, `mw`, `yi`, `rr`, and `rr_t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to set the values referenced by the `real` pointers `rr` and `rr_t` to the laminar and turbulent reaction rates, respectively. Note that when using the stiff chemistry solver, Ansys Fluent does not use `rr_t`.

`rr` and `rr_t` (defined by the UDF) are computed, and the lower of the two values is used when the finite-rate/eddy-dissipation chemical reaction mechanism used. Note that `rr` and `rr_t` are conversion rates in kmol/m^3-s. These rates, when multiplied by the respective stoichiometric coefficients, yield the production/consumption rates of the individual chemical components.

2.3.52.3. Example 1

The following UDF, named `vol_reac_rate`, specifies a volume reaction rate. The function must be executed as a compiled UDF in Ansys Fluent.

```
/************************************************************************
   UDF for specifying a volume reaction rate
   The basics of Ansys Fluent's calculation of reaction rates: only an
   Arrhenius ("finite rate") reaction rate is calculated
   from the inputs given by the user in the graphical user interface
 ************************************************************************/

#include "udf.h"

DEFINE_VR_RATE(vol_reac_rate,c,t,r,wk,yk,rate,rr_t)
{
   real ci, prod;
   int i;

  /* Calculate Arrhenius reaction rate */
   prod = 1.;
   for(i = 0; i < r->n_reactants; i++)
     {
       ci  = C_R(c,t) * yk[r->reactant[i]] / wk[r->reactant[i]];
       prod *= pow(ci, r->exp_reactant[i]);
     }
     *rate = r->A * exp(- r->E / (UNIVERSAL_GAS_CONSTANT * C_T(c,t))) *
             pow(C_T(c,t), r->b) * prod;
     *rr_t = *rate;
     /* No "return..;" value. */
}
```

2.3.52.4. Example 2

When multiple reactions are specified, a volume reaction rate UDF is called several times in each cell. Different values are assigned to the pointer `r`, depending on which reaction the UDF is being called for. Therefore, you will need to determine which reaction is being called, and return the correct rates for that reaction. Reactions can be identified by their name through the `r->name` statement. To test whether a given reaction has the name `reaction-1`, for example, you can use the following C construct:

```
if (!strcmp(r->name, "reaction-1"))
  {
    .... /* r->name is identical to "reaction-1" ... */
  }
```

Important:

Note that `strcmp(r->name, "reaction-1")` returns 0 which is equal to FALSE when the two strings are identical.

It should be noted that `DEFINE_VR_RATE` defines only the reaction rate for a predefined stoichiometric equation (set in the **Reactions** dialog box) therefore providing an alternative to the Arrhenius rate model. `DEFINE_VR_RATE` does not directly address the particular rate of species creation or depletion; this is done by the Ansys Fluent solver using the reaction rate supplied by your UDF.

The following is a source code template that shows how to use `DEFINE_VR_RATE` in connection with more than one user-specified reaction. Note that Ansys Fluent always calculates the `rr` and `rr_t` reaction rates before the UDF is called. Consequently, the values that are calculated are available only in the given variables when the UDF is called.

```
/**********************************************************************
   Multiple reaction UDF that specifies different reaction rates
   for different volumetric chemical reactions
 **********************************************************************/
#include "udf.h"

DEFINE_VR_RATE(myrate,c,t,r,mw,yi,rr,rr_t)
{
  /*If more than one reaction is defined, it is necessary to distinguish
  between these using the names of the reactions.      */
    if (!strcmp(r->name, "reaction-1"))
      {
        /* Reaction 1 */
      }
    else if (!strcmp(r->name, "reaction-2"))
      {
        /* Reaction 2 */
      }
    else
      {
        /*  Message("Unknown Reaction\n"); */
      }
  /* Message("Actual Reaction: %s\n",r->name); */

}
```

2.3.52.5. Hooking a Volumetric Reaction Rate UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_VR_RATE` is interpreted (Interpreting UDFs (p. 393))or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `myrate`) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking `DEFINE_VR_RATE` UDFs (p. 515) for details.

2.3.53. `DEFINE_WALL_FUNCTIONS`

2.3.53.1. Description

You can use `DEFINE_WALL_FUNCTIONS` to provide custom wall functions for applications when you want to replace the standard wall functions in Ansys Fluent. Note that this is available only for use with the k-ε turbulence models, and cannot be combined with the Eulerian multiphase model.

2.3.53.2. Usage

`DEFINE_WALL_FUNCTIONS (name, f, t, c0, t0, wf_ret, yPlus, Emod)`

Argument Type	Description
symbol name	UDF name.
face_t f	face index.
Thread *t	pointer to face thread
cell_t c0	cell index.
Thread *t0	pointer to cell thread.
int wf_ret	wall function index
real yPlus	y+ value
real Emod	wall function E constant

Function returns

`real`

There are eight arguments to `DEFINE_WALL_FUNCTIONS`: `name`, `f`, `t`, `c0`, `t0`, `wf_ret`, `yPlus`, and `Emod`. You supply `name`, the name of the UDF. `f`, `t`, `c0`, `t0`, `wf_ret`, `yPlus`, and `Emod` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the `real` value of the wall functions `U+`, `dU+/dy+`, and `d2U+/dY+2` for laminar and turbulent regions and return them to the solver.

2.3.53.3. Example

The following UDF, named `user_log_law`, computes `U+` and `dU+/dy+`, and `d2U+/dY+2` for laminar and turbulent regions using `DEFINE_WALL_FUNCTIONS`. The source code can be interpreted or compiled in Ansys Fluent.

```
/******************************************************************
   User-defined wall functions: separated into turbulent and laminar regimes
 /******************************************************************/
```

```
#include "udf.h"

DEFINE_WALL_FUNCTIONS(user_log_law, f, t, c0, t0, wf_ret, yPlus, Emod)
{
   real wf_value;
   switch (wf_ret)
      {
      case UPLUS_LAM:
         wf_value = yPlus;
         break;
      case UPLUS_TRB:
         wf_value = log(Emod*yPlus)/KAPPA;
         break;
      case DUPLUS_LAM:
         wf_value = 1.0;
         break;
      case DUPLUS_TRB:
         wf_value = 1./(KAPPA*yPlus);
         break;
      case D2UPLUS_TRB:
         wf_value = -1./(KAPPA*yPlus*yPlus);
         break;
      default:
         printf("Wall function return value unavailable\n");
      }
   return wf_value;
}
```

2.3.53.4. Hooking a Wall Function UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_WALL_FUNCTIONS` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `user_log_law`) will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. See Hooking `DEFINE_WALL_FUNCTIONS` UDFs (p. 517) for details.

2.3.54. `DEFINE_WALL_NODAL_DISP`

2.3.54.1. Description

You can use `DEFINE_WALL_NODAL_DISP` to define the displacement of nodes in a wall that is adjacent to a solid zone, as part of an intrinsic fluid-structure interaction (FSI) simulation. For details about such simulations, see Modeling Fluid-Structure Interaction (FSI) Within Fluent in the User's Guide.

2.3.54.2. Usage

`DEFINE_WALL_NODAL_DISP` (name, f, t, v, m)

Argument Type	Description
`symbol name`	UDF name.
`face_t f`	Face index.
`Thread *t`	Pointer to face thread.
`Node *v`	Node pointer.

Argument Type	Description
int m	Index that defines the displacement component.

Function returns

real

There are five arguments to DEFINE_WALL_NODAL_DISP: name, f, t, v, and m. You supply name, the name of the UDF. f, t, v, and m are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the real value of the displacement Phib for each component and return them to the solver.

2.3.54.3. Example

The following UDF, named wall_nodal_disp, sets displacement components using DEFINE_WALL_NODAL_DISP. It defines each component of nodal displacement to be equal to the corresponding wall coordinate. The source code can be interpreted or compiled in Ansys Fluent.

```
#include "udf.h"

DEFINE_WALL_NODAL_DISP(wall_nodal_disp, f, t, v, m)
{
  int n;
  real Phib;

  if (m==0)
    Phib = NODE_X(v);
  else if (m==1)
    Phib = NODE_Y(v);
  else
    Phib = NODE_Z(v);

  return Phib;
}
```

2.3.55. DEFINE_WALL_NODAL_FORCE

2.3.55.1. Description

You can use DEFINE_WALL_NODAL_FORCE to define the force applied to nodes in a wall that is adjacent to a solid zone as part of an intrinsic fluid-structure interaction (FSI) simulation. For details about such simulations, see Modeling Fluid-Structure Interaction (FSI) Within Fluent in the User's Guide.

2.3.55.2. Usage

DEFINE_WALL_NODAL_FORCE (name, f, t, v, m)

Argument Type	Description
symbol name	UDF name.
face_t f	Face index.

Argument Type	Description
`Thread *t`	Pointer to face thread.
`Node *v`	Node pointer.
`int m`	Index that defines the force component.

Function returns

`real`

There are five arguments to `DEFINE_WALL_NODAL_FORCE`: name, `f`, `t`, `v`, and `m`. You supply name, the name of the UDF. `f`, `t`, `v`, and `m` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the `real` value of the force `Phib` for each component and return them to the solver.

2.3.55.3. Example

The following UDF, named `wall_nodal_force`, sets force components using `DEFINE_WALL_NODAL_DISP`. It specifies the force for each node based on the node location. The source code can be interpreted or compiled in Ansys Fluent.

```
#include "udf.h"

DEFINE_WALL_NODAL_FORCE(wall_nodal_force, f, t, v, m)
{
  int n;
  real Phib;
  real y = NODE_Y(v);

  if (y > 10.1)
    Phib = 125;
  else if (y < 9.9)
    Phib = 125;
  else
    Phib = 250;

  return Phib;
}
```

2.3.56. `DEFINE_SOURCE_FE`

2.3.56.1. Description

You can use `DEFINE_SOURCE_FE` to define the volumetric structural body force, as part of an intrinsic fluid-structure interaction (FSI) simulation. For details about such simulations, see Modeling Fluid-Structure Interaction (FSI) Within Fluent in the User's Guide.

2.3.56.2. Usage

`DEFINE_SOURCE_FE` (name, `c`, `t`)

Argument Type	Description
`symbol name`	UDF name.

Argument Type	Description
cell_t c	Cell index.
Thread *t	Cell thread pointer.

Function returns

real

There are three arguments to DEFINE_SOURCE_FE: name, c, and t. You supply name, the name of the UDF. c and t are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the real value of the force source and return it to the solver.

2.3.56.3. Example

The following UDF, named dz_force, is used to simulate the volumetric body force in the z-direction due to the stretching of a bar under its own weight using DEFINE_SOURCE_FE. It determines the volumetric force by multiplying the density C_R(c,t) by a gravitational factor gravity_z. The source code can be interpreted or compiled in Ansys Fluent.

```
#include "udf.h"

DEFINE_SOURCE_FE(dz_force,c,t)
{
    real gravity_z = -10.;
    real rho = C_R(c,t);
    real source;

    source = rho * gravity_z;

    return source;
}
```

2.3.56.4. Hooking a Volumetric Structural Body Force UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_SOURCE_FE is interpreted (Interpreting UD-Fs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first DEFINE macro argument (for example, dz_force) will become visible and selectable in the appropriate **Displacement Source** dialog for **Solid** cell zones in Ansys Fluent. See Hooking DEFINE_SOURCE_FE UDFs (p. 519) for details.

2.3.57. DEFINE_WSGGM_ABS_COEFF

2.3.57.1. Description

You can use DEFINE_WSGGM_ABS_COEFF to customize the absorption coefficient computed using the domain-based weighted-sum-of-gray-gases model (WSGGM) model, by either replacing the internally calculated value or by modifying the value computed by Ansys Fluent. During the execution, a DEFINE_WSGGM_ABS_COEFF function is called by Ansys Fluent for each fluid zone and also for each band (in the case of a non-gray model). If the soot model is enabled, DEFINE_WSGGM_ABS_COEFF can also be used to modify the soot absorption coefficient computed

by Ansys Fluent. See Radiation in Combusting Flows in the Theory Guide for further information about how composition-dependent absorption coefficients are calculated.

Important:

The WSGGM is implemented in a gray approach. If the WSGGM is used with a non-gray model, the absorption coefficient will be the same in all bands. Use `DEFINE_GRAY_BAND_ABS_COEFF` to change the absorption coefficient per band or per gray gas.

2.3.57.2. Usage

`DEFINE_WSGGM_ABS_COEFF (name, c, t, xi, p_t, s, soot_conc, Tcell, nb, ab_wsggm, ab_soot)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *t`	Pointer to cell thread on which the WSGGM absorption coefficient function is to be applied.
`real xi[]`	Array containing species mole fractions.
`real p_t`	Total pressure.
`real s`	Beam length based on domain size.
`real soot_conc`	Concentration of the soot (if the soot model is enabled).
`real Tcell`	Temperature of the cell.
`int nb`	Band number (nb=0 for gray model).
`real *ab_wsggm`	Absorption coefficient computed by the WSGGM in Ansys Fluent.
`real *ab_soot`	Absorption coefficient computed by the soot model in Ansys Fluent.

Function returns

`void`

There are eleven arguments to `DEFINE_WSGGM_ABS_COEFF`: `name`, `c`, `t`, `xi`, `p_t`, `s`, `soot_conc`, `Tcell`, `nb`, `ab_wsggm`, and `ab_soot`. You supply `name`, the name of the UDF. `c`, `t`, `xi`, `p_t`, `s`, `soot_conc`, `Tcell`, `nb`, `ab_wsggm`, and `ab_soot` are variables that are passed by the Ansys Fluent solver to your UDF.

2.3.57.3. Example

The following UDF, named `user_wsggm_abs_coeff`, replaces the WSGGM and soot absorption coefficients so that they are no longer the value calculated internally by Ansys Fluent. While

`DEFINE_WSGGM_ABS_COEFF` UDFs can be interpreted or compiled in Ansys Fluent, the following example can only be compiled.

```
include "udf.h"
#include "materials.h"

DEFINE_WSGGM_ABS_COEFF(user_wsggm_abs_coeff, c, t, xi, p_t, s, soot_conc, Tcell, nb, ab_wsggm, ab_soot)
{
  Material *m = THREAD_MATERIAL(t);
  int ico2 = mixture_specie_index(m, "co2");
  int ih2o = mixture_specie_index(m, "h2o");
  real CO2_molf, H2O_molf;
  real k2, k3, k4;


  CO2_molf= xi[ico2];
  H2O_molf= xi[ih2o];


      switch (nb)
        {
        case 0 :  /*  First gray gas*/
          {
             *ab_wsggm = 0;
          }
          break;

        case 1 :  /*  Second gray gas*/
          {
            k2   = 0.1;
            *ab_wsggm = (k2 * (H2O_molf + CO2_molf)) * p_t;
          }
          break;

        case 2 :  /*  Third gray gas*/
          {
            k3   =  7.1;
            *ab_wsggm = (k3 * (H2O_molf + CO2_molf)) * p_t;
          }
          break;

        case 3 :  /*  Fourth gray gas*/
          {
            k4   =  60.0;
            *ab_wsggm = (k4 * (H2O_molf + CO2_molf)) * p_t;
          }
        }

      *ab_soot =  0.1;
}
```

2.3.57.4. Hooking a Wall Function UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_WSGGM_ABS_COEFF` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `user_wsggm_abs_coeff`) will become visible and selectable in the **Create/Edit Materials** dialog box in Ansys Fluent. See Hooking `DEFINE_WSGGM_ABS_COEFF` UDFs (p. 520) for details.

2.4. Multiphase `DEFINE` Macros

The `DEFINE` macros presented in this section are used for multiphase applications, *only*.

Table 2.10: Quick Reference Guide for Multiphase DEFINE Macros (p. 180) provides a quick reference guide to the multiphase-specific `DEFINE` macros, the functions they are used to define, and the dialog boxes in which they are activated in Ansys Fluent. Definitions of each `DEFINE` macro are listed in the `udf.h` header file (see Appendix C: Quick Reference Guide for Multiphase `DEFINE` Macros (p. 699)).

Appendix B: `DEFINE` Macro Definitions (p. 693) contains a list of general purpose `DEFINE` macros that can also be used to define UDFs for multiphase cases. For example, the general purpose `DEFINE_PROPERTY` macro is used to define a surface tension coefficient UDF for the multiphase VOF model. See `DEFINE_PROPERTY` UDFs (p. 119) for details.

Table 2.10: Quick Reference Guide for Multiphase DEFINE Macros

Model	Function	**DEFINE Macro**	Dialog Box Activated
VOF	linearized mass transfer	`DEFINE_LINEARIZED_MASS_TRANSFER`	**Phase Interaction**
	mass transfer	`DEFINE_MASS_TRANSFER`	**Phase Interaction**
	heterogeneous reaction rate	`DEFINE_HET_RXN_RATE`	**Phase Interaction**
Mixture	linearized mass transfer	`DEFINE_LINEARIZED_MASS_TRANSFER`	**Phase Interaction**
	mass transfer	`DEFINE_MASS_TRANSFER`	**Phase Interaction**
	drag exchange coefficient	`DEFINE_EXCHANGE_PROPERTY`	**Phase Interaction**
	drag modification factor	`DEFINE_EXCHANGE_PROPERTY`	**Phase Interaction**
	slip velocity	`DEFINE_VECTOR_EXCHANGE_PROPERTY`	**Phase Interaction**
	cavitation rate	`DEFINE_CAVITATION_RATE`	**User-Defined Function Hooks**
	heterogeneous reaction rate	`DEFINE_HET_RXN_RATE`	**Phase Interaction**
Eulerian	linearized mass transfer	`DEFINE_LINEARIZED_MASS_TRANSFER`	**Phase Interaction**
	mass transfer	`DEFINE_MASS_TRANSFER`	**Phase Interaction**
	heat transfer	`DEFINE_EXCHANGE_PROPERTY`	**Phase Interaction**

Model	Function	DEFINE Macro	Dialog Box Activated
	drag exchange coefficient	`DEFINE_EXCHANGE_PROPERTY`	**Phase Interaction**
	drag modification factor	`DEFINE_EXCHANGE_PROPERTY`	**Phase Interaction**
	lift coefficient	`DEFINE_EXCHANGE_PROPERTY`	**Phase Interaction**
	wall lubrication coefficient	`DEFINE_EXCHANGE_PROPERTY`	**Phase Interaction**
	wall lubrication model parameters	`DEFINE_EXCHANGE_PROPERTY`	**Phase Interaction**
	turbulent dispersion force	`DEFINE_VECTOR_EXCHANGE_PROPERTY`	**Phase Interaction**
	turbulent dispersion model parameters	`DEFINE_EXCHANGE_PROPERTY`	**Phase Interaction**
	turbulence interaction model parameters	`DEFINE_EXCHANGE_PROPERTY`	**Phase Interaction**
	virtual mass coefficient	`DEFINE_EXCHANGE_PROPERTY`	**Phase Interaction**
	interfacial area	`DEFINE_EXCHANGE_PROPERTY`	**Phase Interaction**
	boiling model and quenching	`DEFINE_BOILING_PROPERTY`	**Boiling Model**
	heterogeneous reaction rate	`DEFINE_HET_RXN_RATE`	**Phase Interaction**

2.4.1. `DEFINE_BOILING_PROPERTY`

2.4.1.1. Description

You can use `DEFINE_BOILING_PROPERTY` to model the boiling model parameters and the quenching model correction. The parameters include the **Bubble Departure Diameter**, **Frequency of Bubble Departure**, **Nucleation Site Density**, **Area Influence Coeff.**, and **Liquid Reference Temperature** for quenching correction.

2.4.1.2. Usage

`DEFINE_BOILING_PROPERTY (name, f, t, c0, t0, from_phase_index, from_species_index, to_phase_index, to_species_index)`

> **Important:**
>
> As with all the other user-defined functions, all of the arguments to this `DEFINE` macro must be placed on the same line in your source code. Splitting the `DEFINE` statement onto several lines will result in a compilation error.

Argument Type	Description
`symbol name`	UDF name.
`face_t f`	Index that identifies a wall face.
`Thread *t`	Pointer to the wall face thread.
`cell_t c0`	Cell index that identifies the cell next to the wall.
`Thread *t0`	Pointer to mixture-level cell thread.
`int from_phase_index`	Liquid phase in boiling models.
`int from_species_index`	ID of liquid species (ID=-1 if no mixture material).
`int to_phase_index`	Vapor phase in boiling models.
`int to_species_index`	ID of vapor species (ID=-1 if no mixture material).

Function returns

`real`

There are nine arguments to `DEFINE_BOILING_PROPERTY`: `name`, `f`, `t`, `c0`, `t0`, `from_phase_index`, `from_species_index`, `to_phase_index`, and `to_species_index`. You supply `name`, the name of the UDF. The remaining eight variables are passed by the Ansys Fluent solver to your UDF. The defined UDF will return the desired `real` value for a specific model parameter.

> **Important:**
>
> Note that the arguments `from_species_index` and `to_species_index` are relevant for multiphase species transport problems only, and only if the respective phase has a mixture material associated with it.

2.4.1.3. Example

The following UDF named `bubble_depart_dia`, demonstrates how the bubble diameter is computed. All other boiling parameters can use this example and can be modified accordingly.

```
/************************************************************************
 UDF that demonstrates how to compute the bubble diameter based on tolubinski-kostanchuk.
 Can be interpreted or compiled.
 ***********************************************************************/

#include "udf.h"

#define d_bw_max 0.0014
#define d_bw_coef 0.0006
#define subcool_ref 45.0

DEFINE_BOILING_PROPERTY(bubble_depart_dia,f,t,c0,t0,from_index,from_species_index,to_index,to_species_index)
  {
    real diam_b, subcool;

    int liq_phase = from_index;
    Thread **pt0    = THREAD_SUB_THREADS(t0);
    real T_SAT = C_STORAGE_R(c0,t0,SV_SAT_TEMPERATURE);
```

```
    real T_l = C_T(c0, pt0[liq_phase]);

    subcool = T_SAT - T_l;
    diam_b = MIN(d_bw_max,d_bw_coef*exp(-subcool/subcool_ref));

    return diam_b;
}
```

2.4.1.4. Hooking a Boiling Property UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_BOILING_PROPERTY` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `bubble_depart_dia`) will become visible and selectable in the **Boiling Models** dialog box in Ansys Fluent. See Hooking `DEFINE_BOIL-ING_PROPERTY` UDFs (p. 522) for details.

2.4.2. `DEFINE_CAVITATION_RATE`

2.4.2.1. Description

You can use `DEFINE_CAVITATION_RATE` to model the cavitation source terms R_e and R_c in the vapor mass fraction transport equation used in the Singhal et al model (see Equation 14.565 in the Theory Guide). Assuming m_{dot} denotes the mass-transfer rate between liquid and vapor phases, we have

$$R_e = MAX\left[m_{dot}, 0\right] f_1$$

$$R_c = MAX\left[-m_{dot}, 0\right] f_v$$

where f_1 and f_v are the mass-fraction of the liquid and vapor phase, respectively.

`DEFINE_CAVITATION_RATE` is used to calculate m_{dot} only. The values of R_e and R_c are computed by the solver, accordingly.

2.4.2.2. Usage

`DEFINE_CAVITATION_RATE` (name, c, t, p, rhoV, rhoL, mafV, p_v, cigma, f_gas, m_dot)

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *t`	Pointer to the mixture-level thread.
`real *p[c]`	Pointer to shared pressure.
`real *rhoV[c]`	Pointer to vapor density.
`real *rhoL[c]`	Pointer to liquid density.
`real *mafV[c]`	Pointer to vapor mass fraction.
`real *p_v`	Pointer to vaporization pressure.

Argument Type	Description
`real *cigma`	Pointer to liquid surface tension coefficient.
`real *f_gas`	Pointer to the prescribed mass fraction of non condensable gases.
`real *m_dot`	Pointer to cavitation mass transfer rate.

Function returns

`void`

There are eleven arguments to `DEFINE_CAVITATION_RATE`: `name`, `c`, `t`, `p`, `rhoV`, `rhoL`, `mafV`, `p_v`, `cigma`, `f_gas`, and `m_dot`. You supply `name`, the name of the UDF. `c`, `t`, `p`, `rhoV`, `rhoL`, `mafV`, `p_v`, `cigma`, `f_gas`, and `m_dot` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to set the value referenced by the `real` pointer `m_dot` to the cavitation rate.

2.4.2.3. Example

The following UDF named `c_rate`, is an example of a cavitation model for a multiphase mixture that is different from the default model in Ansys Fluent. This cavitation model calculates the cavitation mass transfer rates between the liquid and vapor phase depending on fluid pressure (`*p`), turbulence kinetic energy (`C_K(c,t)`), and the liquid vaporization pressure (`*p_v`).

In general, the existence of turbulence enhances cavitation. In this example, the turbulence effect is taken into account by increasing the cavitation pressure by `0.195* C_R(c,t) * C_K(c,t)`. The pressure `p_vapor` that determines whether cavitation occurs increases from `p_v` to

```
p_v + 0.195 * C_R(c,t) * C_K(c,t)
```

When the absolute fluid pressure (`ABS_P`) is lower than `p_vapor`, then liquid evaporates to vapor (R_e). When it is greater than `p_vapor`, vapor condenses to liquid (R_c).

The evaporation rate is calculated by

```
If ABS_P < p_vapor, then
  c_evap * rhoV[c] * sqrt(2.0/3.0*rhoL[c]) * ABS(p_vapor - ABS_P(p[c]))
```

The condensation rate is

```
If ABS_P > p_vapor, then
  -c_con*rhoL[c] * sqrt(2.0/3.0*rhoL[c]) * ABS(p_vapor - ABS_P(p[c]))
```

where `c_evap` and `c_con` are model coefficients.

```
/***********************************************************************
 UDF that is an example of a cavitation model different from default.
 Can be interpreted or compiled.
 **********************************************************************/

#include "udf.h"

#define c_evap 1.0
#define c_con 0.1

DEFINE_CAVITATION_RATE(c_rate,c,t,p,rhoV,rhoL,mafV,p_v,cigma,f_gas, m_dot)
```

```
  {
      real p_vapor = *p_v;
      real dp, dp0, source;
      p_vapor += MIN(0.195*C_R(c,t)*C_K(c,t), 5.0*p_vapor);
      dp = p_vapor - ABS_P(p[c], op_pres);
      dp0 = MAX(0.1, ABS(dp));
      source = sqrt(2.0/3.0*rhoL[c])*dp0;
      if(dp > 0.0)
          *m_dot = c_evap*rhoV[c]*source;
      else
          *m_dot = -c_con*rhoL[c]*source;
  }
```

2.4.2.4. Hooking a Cavitation Rate UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_CAVITATION_RATE` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `c_rate`) will become visible and selectable in the **User-Defined Function Hooks** dialog box in Ansys Fluent. See Hooking `DEFINE_CAVITATION_RATE` UDFs (p. 523) for details.

2.4.3. `DEFINE_EXCHANGE_PROPERTY`

2.4.3.1. Description

You can use `DEFINE_EXCHANGE_PROPERTY` to specify UDFs for some phase interaction variables in multiphase models. These include net heat transfer rate between phases, virtual mass coefficient, drag exchange coefficient, lift coefficient, wall lubrication coefficient, and interfacial area (for the Eulerian multiphase boiling model). Drag exchange coefficient may also be specified for the Mixture model. Below is a list of user-defined functions that can be specified using `DEFINE_EX-CHANGE_PROPERTY` for the multiphase models in Ansys Fluent. Note that there are some phase interaction variables such as vaporization pressure and surface tension coefficient (cavitation parameters) that are defined using `DEFINE_PROPERTY`. See `DEFINE_PROPERTY` UDFs (p. 119) for details.

Table 2.11: `DEFINE_EXCHANGE_PROPERTY` Variables

Mixture Model	Eulerian Model
drag exchange coefficient drag modification factor, η in Equation 14.270 of the Theory Guide	volumetric heat transfer coefficient, h_{pq} in Equation 14.359 of the Theory Guide virtual mass coefficient, C_{vm} in Equation 14.320 in the *Fluent Theory Guide* drag exchange coefficient, K_{pq} in Equation 14.189 of the Theory Guide drag modification factor, η in Equation 14.270 of the Theory Guide lift coefficient, C_l in Equation 14.276 of the Theory Guide

Mixture Model	Eulerian Model
	wall lubrication coefficient, C_{wl} in Equation 14.297 of the Theory Guide
	interfacial area, A_i in Interfacial Area Concentration in the Theory Guide
	model coefficients for wall lubrication, turbulent dispersion, and turbulence interaction models.

2.4.3.2. Usage

`DEFINE_EXCHANGE_PROPERTY (name, c, mixture_thread, phase_index_1, phase_index_2)`

Important:

Note that all of the arguments to a `DEFINE` macro must be placed on the same line in your source code. Splitting the `DEFINE` statement onto several lines will result in a compilation error.

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *mixture_thread`	Pointer to the mixture-level thread.
`int phase_index_1`	Identifier of the first phase from the pair of phases for which you want to customize a phase interaction property. An index of **0** corresponds to the primary phase and is incremented by one for each secondary phase.
`int phase_index_2`	Identifier of the second phase from the pair of phases for which you want to customize a phase interaction property. See `int phase_index_1` for more information.

Function returns

`real`

There are five arguments to `DEFINE_EXCHANGE_PROPERTY`: `name`, `c`, `mixture_thread`, `phase_index_1`, and `phase_index_2`. You supply `name`, the name of the UDF. `c`, `mixture_thread`, `phase_index_1`, and `phase_index_2` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the `real` value of the volumetric

heat transfer coefficient, virtual mass coefficient, lift coefficient, drag exchange coefficient, wall lubrication coefficient, or interfacial area to the solver.

Note:

- **Ansys Fluent** passes values of phase identifiers in the same order as they appear in the **Phase Pairs** selection list in the **Multiphase Model** dialog box (**Phase Interaction** > **Forces** tab). In the example shown in Defining Forces in the Forces Tab in the *Fluent User's Guide*, if you want to customize a phase interaction property for the phase pair (**liquid vapor**), where **liquid** is the primary phase and **vapor** is the only secondary phase, **Ansys Fluent** will pass 0 and 1 for phase_index_1 and phase_index_2, respectively.

- Your UDF should return the effective coefficient. So, if the quantity being modeled by your UDF depends on the interfacial area, your UDF should return the coefficient multiplied by the interfacial area.

2.4.3.3. Example 1 - Custom Drag Law

The following UDF, named custom_drag, can be used to customize the default Syamlal drag law in Ansys Fluent. The default drag law uses 0.8 (for void < =0.85) and 2.65 (void>0.85) for bfac. This results in a minimum fluid velocity of 25 cm/s. The UDF modifies the drag law to result in a minimum fluid velocity of 8 cm/s, using 0.28 and 9.07 for the bfac parameters.

```
/********************************************************************
 UDF for customizing the default Syamlal drag law in ANSYS Fluent
 *******************************************************************/
#include "udf.h"
#define pi 4.*atan(1.)
#define diam2 3.e-4
DEFINE_EXCHANGE_PROPERTY(custom_drag,cell,mix_thread,s_col,f_col)
{
  Thread *thread_g, *thread_s;
  real x_vel_g, x_vel_s, y_vel_g, y_vel_s, abs_v, slip_x, slip_y,
      rho_g, rho_s, mu_g, reyp, afac,
      bfac, void_g, vfac, fdrgs, taup, k_g_s;

  /* find the threads for the gas (primary) */
  /* and solids (secondary phases)   */
  thread_g = THREAD_SUB_THREAD(mix_thread, s_col);/* gas phase */
  thread_s = THREAD_SUB_THREAD(mix_thread, f_col);/* solid phase*/

  /* find phase velocities and properties*/
  x_vel_g = C_U(cell, thread_g);
  y_vel_g = C_V(cell, thread_g);
  x_vel_s = C_U(cell, thread_s);
  y_vel_s = C_V(cell, thread_s);
  slip_x = x_vel_g - x_vel_s;
  slip_y = y_vel_g - y_vel_s;
  rho_g = C_R(cell, thread_g); rho_s = C_R(cell, thread_s);
  mu_g = C_MU_L(cell, thread_g);

  /*compute slip*/
  abs_v = sqrt(slip_x*slip_x + slip_y*slip_y);

  /*compute Reynolds number*/
  reyp = rho_g*abs_v*diam2/mu_g;

  /* compute particle relaxation time */
  taup = rho_s*diam2*diam2/18./mu_g;
```

```
      void_g = C_VOF(cell, thread_g);/* gas vol frac*/

      /*compute drag and return drag coeff, k_g_s*/
      afac = pow(void_g,4.14);
      if(void_g <= 0.85)
        bfac = 0.281632*pow(void_g, 1.28);
      else
        bfac = pow(void_g, 9.076960);
       vfac = 0.5*(afac-0.06*reyp+sqrt(0.0036*reyp*reyp+0.12*reyp*(2.*bfac-
             afac)+afac*afac));
       fdrgs = void_g*(pow((0.63*sqrt(reyp)/
              vfac+4.8*sqrt(vfac)/vfac),2))/24.0;
      k_g_s = (1.-void_g)*rho_s*fdrgs/taup;
     return k_g_s;
  }
```

2.4.3.4. Example 2 - Custom Lift Law

The following UDF, named custom_lift, computes the coefficient for the lift force using a formulation developed by Tomiyama in 2002:

```
/* this example uses "user-defined" to implement a lift model by Tomiyama et al in 2002  */
#include "udf.h"
#include "flow.h"
#define Eo_l1 4.0
#define Eo_l2 10.0

DEFINE_EXCHANGE_PROPERTY(custom_lift,c,t,i,j)
{
/* i -- liquid-phase; j -- vapor-phase */
  Thread **pt = THREAD_SUB_THREADS(t);

  real v_x=0., v_y=0., v_z=0.;
  real vel, Rev, Eo, d2, T_sfc, sigma;
  real lift_coeff, lift_co, wk_co;;

  real diam = C_PHASE_DIAMETER(c,pt[j]);
  real rho_v = C_R(c,pt[j]);
  real rho_l = C_R(c,pt[i]);
  real mu_l  = C_MU_L(c,pt[i]);
  real gravity = NV_MAG(M_gravity);

  Property *prop =
  DOMAIN_COMPLEX_PROP_PROPERTY(DOMAIN_INTERACTION(root_domain),
                                              COMPLEX_PROP_sfc_tension_coeff,
                                              i,j);
  T_sfc = (sg_temperature && NNULLP(THREAD_STORAGE(pt[i],SV_T)))? C_T(c,pt[i]) :  T_REF;
  if(prop == NULL || PROPERTY_METHOD(prop,0) == PROP_METHOD_NONE)
     Error("Lift-Tomiyama: Please set value for surface tension !");
  sigma = generic_property(c,t,prop,(Property_ID)0,T_sfc);
  if(sigma <= 0.) Error("Lift-Tomiyama: Please set nonzero value for surface tension !");

/*   calculate bubble Reynolds Number */
  v_x = C_U(c,pt[j]) - C_U(c,pt[i]);
  v_y = C_V(c,pt[j]) - C_V(c,pt[i]);
#if RP_3D
  v_z = C_W(c,pt[j]) - C_W(c,pt[i]);
#endif
  vel = sqrt(v_x*v_x + v_y*v_y + v_z*v_z);
  Rev =  RE_NUMBER(rho_l,vel,diam,mu_l);

  d2 = diam*diam;
  Eo = gravity*(rho_l-rho_v)*d2/sigma;

  if (Eo <= Eo_l1)
     wk_co = 0.0;
  else if (Eo < Eo_l2)
```

```
      wk_co = -0.096*Eo + 0.384;
  else
      wk_co = -0.576;

  lift_co = 0.288*tanh(0.121*Rev);

  lift_coeff = lift_co + wk_co;

  return lift_coeff;
}
```

2.4.3.5. Example 3- Heat Transfer

The following UDF, named `heat_udf`, specifies a coefficient that when multiplied by the temperature difference between the dispersed and continuous phases, is equal to the net rate of heat transfer per unit volume.

```
#include "udf.h"

 #define PR_NUMBER(cp,mu,k) ((cp)*(mu)/(k))
 #define IP_HEAT_COEFF(vof,k,nu,d) ((vof)*6.*(k)*(Nu)/(d)/(d))

 static real heat_ranz_marshall(cell_t c, Thread *ti, Thread *tj)
 {
   real h;
   real d = C_PHASE_DIAMETER(c,tj);
   real k = C_K_L(c,ti);
   real NV_VEC(v), vel, Re, Pr, Nu;
   NV_DD(v,=,C_U(c,tj),C_V(c,tj),C_W(c,tj),-,C_U(c,ti),C_V(c,ti),C_W(c,ti));
   vel = NV_MAG(v);
   Re = RE_NUMBER(C_R(c,ti),vel,d,C_MU_L(c,ti));
   Pr = PR_NUMBER (C_CP(c,ti),C_MU_L(c,ti),k);
   Nu = 2. + 0.6*sqrt(Re)*pow(Pr,1./3.);
   h = IP_HEAT_COEFF(C_VOF(c,tj),k,Nu,d);
   return h;
 }

 DEFINE_EXCHANGE_PROPERTY(heat_udf, c, t, i, j)
 {
     Thread *ti = THREAD_SUB_THREAD(t,i);
     Thread *tj = THREAD_SUB_THREAD(t,j);
     real val;
     val = heat_ranz_marshall(c,ti, tj);
     return val;
 }
```

2.4.3.6. Example 4- Custom Interfacial Area

The following UDF named `custom_ia`, computes the interfacial area, while including the symmetric model.

```
#include "udf.h"

DEFINE_EXCHANGE_PROPERTY(custom_ia,c,t,i,j)
{
/* i -- liquid-phase; j -- vapor-phase */
  Thread **pt = THREAD_SUB_THREADS(t);
  real diam = C_PHASE_DIAMETER(c, pt[j]);
  real vof_i = C_VOF(c,pt[i]);
  real vof_j = C_VOF(c,pt[j]);

  real area_intf;

  area_intf = 6.*vof_i*vof_j/diam;
```

```
    return area_intf;
}
```

2.4.3.7. Hooking an Exchange Property UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_EXCHANGE_PROPERTY` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `heat_udf`) will become visible and selectable in the **Phase Interaction** tab of the **Multiphase Model** dialog box. See Hooking `DEFINE_EXCHANGE_PROPERTY` UDFs (p. 525) for details.

2.4.4. `DEFINE_HET_RXN_RATE`

2.4.4.1. Description

You need to use `DEFINE_HET_RXN_RATE` to specify reaction rates for heterogeneous reactions. A heterogeneous reaction is one that involves reactants and products from more than one phase. Unlike `DEFINE_VR_RATE`, a `DEFINE_HET_RXN_RATE` UDF can be specified differently for different heterogeneous reactions.

During Ansys Fluent execution, the `DEFINE_HET_RXN_RATE` UDF for each heterogeneous reaction that is defined is called in every fluid cell. Ansys Fluent will use the reaction rate specified by the UDF to compute production/destruction of the species participating in the reaction, as well as heat and momentum transfer across phases due to the reaction.

A heterogeneous reaction is typically used to define reactions involving species of different phases. Heterogeneous reactions are defined in the **Phase Interaction** > **Heat, Mass, Reactions** > **Reactions** tab of the **Multiphase Model** dialog box.

2.4.4.2. Usage

`DEFINE_HET_RXN_RATE (name,c,t,r,mw,yi,rr,rr_t)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *t`	Cell thread (mixture level) on which heterogeneous reaction rate is to be applied.
`Hetero_Reaction *r`	Pointer to data structure that represents the current heterogeneous reaction (see `sg_mphase.h`).
`real mw[MAX_PHASES][MAX_SPE_EQNS]`	Matrix of species molecular weights. `mw[i][j]` will give molecular weight of species with ID `j` in phase with index `i`. For phase which has fluid material, the molecular weight can be accessed as mw[i][0]
`real yi[MAX_PHASES][MAX_SPE_EQNS]`	Matrix of species mass fractions. `yi[i][j]` will give mass fraction of species with ID `j`

Argument Type	Description
	in phase with index `i`. For phase which has fluid material, `yi[i][0]` will be 1.
`real *rr`	Pointer to laminar reaction rate.
`real *rr_t`	Currently not used. Provided for future use.

Function returns

`void`

There are eight arguments to `DEFINE_HET_RXN_RATE`: `name`, `c`, `t`, `r`, `mw`, `yi`, `rr`, and `rr_t`. You supply `name`, the name of the UDF. `c`, `t`, `r`, `mw`, `yi`, `rr`, and `rr_t` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to set the values referenced by the `real` pointer `rr`. The values must be specified in $kmol/m^3s$ (where the volume is the cell volume).

2.4.4.3. Example

The following compiled UDF named `user_evap_condens_react` defines the reaction rate required to simulate evaporation or condensation on the surface of droplets. Such a reaction can be formally described by the following:

$$H_2O_{(liq)} \leftarrow \rightarrow H_2O_{(gas)} \tag{2.14}$$

Here, gas is a primary phase mixture of two species: $H_2O_{(gas)}$ and air. Droplets constitute the secondary phase and represent a mixture of one species - $H_2O_{(liq)}$. Single-species mixtures are allowed in multiphase models.

The formulation for the reaction rate follows the model for particle evaporation that is defined in Droplet Vaporization (Law 2) in the Theory Guide.

```
#include "udf.h"

 /*Constants used in psat_h2o to calculate saturation pressure*/

 #define PSAT_A 0.01
 #define PSAT_TP 338.15
 #define C_LOOP 8
 #define H2O_PC 22.089E6
 #define H2O_TC 647.286

 /*user inputs*/

 #define MAX_SPE_EQNS_PRIM 2 /*total number of species in primary phase*/
 #define index_evap_primary 0 /*evaporating species index in primary phase*/
 #define prim_index 0 /*index of primary phase*/
 #define P_OPER 101325 /*operating pressure equal to GUI value*/

 /*end of user inputs*/

 /************************************************************/
 /* UDF for specifying an interfacial area density   */
 /************************************************************/
 double psat_h2o(double tsat)
 /*               */
 /* Computes saturation pressure of water vapor     */
 /* as function of temperature              */
 /* Equation is taken from THERMODYNAMIC PROPERTIES IN SI, */
 /* by Reynolds, 1979                  */
```

```
                     /* Returns pressure in PASCALS, given temperature in KELVIN */
                     {
                       int i;
                       double var1,sum1,ans1,psat;
                       double constants[8]={-7.4192420, 2.97221E-1, -1.155286E-1,
                           8.68563E-3, 1.094098E-3, -4.39993E-3, 2.520658E-3, -5.218684E-4};

                       /* var1 is an expression that is used in the summation loop */
                       var1 = PSAT_A*(tsat-PSAT_TP);

                       /* Compute summation loop */
                       i = 0;
                       sum1 = 0.0;
                       while (i < C_LOOP){
                           sum1+=constants[i]*pow(var1,i);
                           ++i;
                       }
                       ans1 = sum1*(H2O_TC/tsat-1.0);

                       /* compute exponential to determine result */
                       /* psat has units of Pascals      */

                       psat = H2O_PC*exp(ans1);
                       return psat;
                     }

                     DEFINE_HET_RXN_RATE(user_evap_condens_react, c, t, hr, mw, yi, rr, rr_t)
                     {
                        Thread **pt = THREAD_SUB_THREADS(t);
                        Thread *tp = pt[0];
                        Thread *ts = pt[1];
                        int i;
                        real concentration_evap_primary, accum = 0., mole_frac_evap_prim,
                         concentration_sat ;
                        real T_prim = C_T(c,tp); /*primary phase (gas) temperature*/
                        real T_sec = C_T(c,ts); /*secondary phase (droplet) temperature*/
                        real diam = C_PHASE_DIAMETER(c,ts); /*secondary phase diameter*/
                        real D_evap_prim = C_DIFF_EFF(c,tp,index_evap_primary)
                          - 0.7*C_MU_T(c,tp)/C_R(c,tp);

                         /*primary phase species turbulent diffusivity*/
                        real Re, Sc, Nu, urel, urelx,urely,urelz=0., mass_coeff, area_density,
                          flux_evap ;

                        if(Data_Valid_P())
                          {
                             urelx = C_U(c,tp) - C_U(c,ts);
                             urely = C_V(c,tp) - C_V(c,ts);

                             #if RP_3D
                               urelz = C_W(c,tp) - C_W(c,ts);
                             #endif

                             urel = sqrt(urelx*urelx + urely*urely + urelz*urelz);
                             /*relative velocity*/

                            Re = urel * diam * C_R(c,tp) / C_MU_L(c,tp);
                            Sc = C_MU_L(c,tp) / C_R(c,tp) / D_evap_prim ;
                            Nu = 2. + 0.6 * pow(Re, 0.5)* pow(Sc, 0.333);
                            mass_coeff = Nu * D_evap_prim / diam ;
                            for (i=0; i < MAX_SPE_EQNS_PRIM ; i++)
                              {
                                accum = accum + C_YI(c,tp,i)/mw[i][prim_index];
                              }
                            mole_frac_evap_prim = C_YI(c,tp,index_evap_primary)
                                / mw[index_evap_primary][prim_index] / accum;
                            concentration_evap_primary = mole_frac_evap_prim * P_OPER
                                / UNIVERSAL_GAS_CONSTANT / T_prim ;
                            concentration_sat = psat_h2o(T_sec)/UNIVERSAL_GAS_CONSTANT/T_sec ;
                            area_density = 6. * C_VOF(c,ts) / diam ;
                            flux_evap = mass_coeff *
```

```
            (concentration_sat - concentration_evap_primary) ;
        *rr = area_density * flux_evap ;
    }
}
```

2.4.4.4. Hooking a Heterogeneous Reaction Rate UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_HET_RXN_RATE` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `user_evap_condens_react`) will become visible and selectable under **Reaction Rate Function** in the **Phase Interaction** > **Heat, Mass, Reactions** > **Reactions** tab of the **Multiphase Model** dialog box. (Note you will first need to specify the **Total Number of Reactions** greater than **0**.) See Hooking `DEFINE_HET_RXN_RATE` UDFs (p. 528) for details.

2.4.5. `DEFINE_LINEARIZED_MASS_TRANSFER`

2.4.5.1. Description

You can use `DEFINE_LINEARIZED_MASS_TRANSFER` when you want to model mass transfer in a multiphase problem. This macro allows you to linearize the mass transfer source terms as well as couple the interfacial mass transfer with flows. This is the recommend UDF method for modeling mass transfer in multiphase flows.

You can linearize the mass transfer term for the calculation of the volume fraction equation in Ansys Fluent, such that

$$\dot{m} = \dot{m}_0 + k_1 \alpha_{from} - k_2 \alpha_{to} \tag{2.15}$$

where

$\dot{m}$ = mass transfer rate

α_{from} = volume fraction of the phase from which mass is transferred

α_{to} = volume fraction of the phase to which mass is transferred

$\dot{m}_0$ = mass transfer source which cannot be linearized to α_{from} and α_{to}

k_1 linearization coefficient related to α_{from}

k_2 linearization coefficient related to α_{to}

To couple the mass transfer terms with flow transport equations, the derivative of the mass transfer rate to pressure is required to be computed and stored in a macro:

$$C\_STORAGE\_R(c, mixture\_thread, SV\_MT\_DS\_DP)$$
$$= abs\left\{ \left(\frac{1}{\rho_{ref,to}} - \frac{1}{\rho_{ref,from}} \right) \frac{\partial \dot{m}}{\partial P} \right\} \tag{2.16}$$

Where $\rho_{ref,to}$ and $\rho_{ref,from}$ are the reference densities of the phases. Typically, they are the cell phase densities.

2.4.5.2. Usage

`DEFINE_LINEARIZED_MASS_TRANSFER (name, c, mixture_thread, from_phase_index, from_species_index, to_phase_index, to_species_index, lin_from, lin_to)`

Important:

Note that all of the arguments to a `DEFINE` macro need to be placed on the same line in your source code. Splitting the `DEFINE` statement onto several lines will result in a compilation error.

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index of cell on the thread pointed to by `mixture_thread`.
`Thread *mixture_thread`	Pointer to mixture-level thread.
`int from_phase_index`	Index of phase from which mass is transferred.
`int from_species_index`	ID of species from which mass is transferred (ID= -1 if phase does not have mixture material).
`int to_phase_index`	Index of phase to which mass is transferred.
`int to_species_index`	ID of species to which mass is transferred (ID= -1 if phase does not have mixture material).
`real *lin_from`	Linearization term for the origin phase
`real *lin_to`	Linearization term for the destination phase

Function returns

`real`

There are nine arguments to `DEFINE_LINEARIZED_MASS_TRANSFER`: `name`, `c`, `mixture_thread`, `from_phase_index`, `from_species_index`, `to_phase_index`, `to_species_index`, `lin_from`, `lin_to`. You supply `name`, the name of the UDF. The variables `c`, `mixture_thread`, `from_phase_index`, `from_species_index`, `to_phase_index`, `to_species_index`, `lin_from`, and `lin_to` are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the real value of the mass transfer rate to the solver and the two linearized terms `lin_from` and `lin_to`.

Important:

The linearization terms `*lin_from` and `*lin_to` are calculated based on the linearization coefficients k_1 and k_2 in [Equation 2.15 (p. 193):](#)

$$*lin\_from = k_1 + k_2 \tag{2.17}$$

$$*lin_{to} = -k_1 - k_2 \tag{2.18}$$

The derivative of mass transfer rate to pressure is stored in the above-mentioned macro in Equation 2.16 (p. 193).

The arguments `from_species_index` and `to_species_index` are relevant for multiphase species transport problems only, and only if the respective phase has a mixture material associated with it.

2.4.5.3. Example

The following UDF, named `cav_source`, specifies mass transfer source terms as a function of liquid vaporization pressure and flow pressure.

Important:

Note that in the example that follows, the `DEFINE_LINEARIZED_MASS_TRANSFER` statement is broken up into three lines for the sake of readability. In your source file, you must make sure that the `DEFINE` statement is on one line only.

```
 #include "udf.h"
DEFINE_LINEARIZED_MASS_TRANSFER(cav_source,cell,thread,from_index,from_species_index, to_index, to_species_inde
{
   real  vof_nuc =  RP_Get_Real("mp/cvt/cfx/vof-nuc");
   real  r_b     =  RP_Get_Real("mp/cvt/cfx/r-bubbles");
   real  F_evap  =  RP_Get_Real("mp/cvt/cfx/f-evap");
   real  F_cond  =  RP_Get_Real("mp/cvt/cfx/f-cond");
   real  c_evap = 3.0*F_evap*vof_nuc/r_b;
   real  c_cond = 3.0*F_cond/r_b;
   real  P_SAT = RP_Get_Real("mp/cvt/vapor-p");

   Thread *liq = THREAD_SUB_THREAD(thread, from_index);
   Thread *vap = THREAD_SUB_THREAD(thread, to_index);


   real m_dot, dp, m_source;
   real p_op = RP_Get_Real ("operating-pressure");
   real press = C_P(cell, thread) + p_op;
   real rho_l = C_R(cell,liq);
   real rho_v = C_R(cell,vap);
   real vof_l = C_VOF(cell,liq);
   real vof_v = C_VOF(cell,vap);
   real r_rho_lv = 1./rho_v - 1./rho_l;
m_dot = 0.;
   m_source = 0.0;

   if (press <=  P_SAT)
     {
        dp = P_SAT - press;
        dp = MAX(dp, 1e-4);

        m_dot = c_evap*rho_v*sqrt(2/3.0*dp/rho_l);
        m_source = m_dot*vof_l;

        *d_mdot_d_vof_from  = m_dot;
        *d_mdot_d_vof_to    = -m_dot;
     }
   else
     {
       dp = press - P_SAT;
       dp = MAX(dp, 1e-4);

       m_dot = -c_cond*rho_v*sqrt(2/3.0*dp/rho_l);
       m_source = m_dot*vof_v;
```

```
        *d_mdot_d_vof_from  = m_dot;
        *d_mdot_d_vof_to    = -m_dot;
      }

  /*  ++++++++++ ds/dp term ++++++++++++++   */
    if(NNULLP(THREAD_STORAGE(thread, SV_MT_DS_DP)))
        C_STORAGE_R(cell,thread,SV_MT_DS_DP) = ABS(r_rho_lv*m_source/(2*dp));

  return m_source;
}
```

2.4.5.4. Hooking a Linearized Mass Transfer UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_LINEARIZED_MASS_TRANSFER` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible and selectable under **Mass Transfer** in the **Phase Interaction** > **Heat, Mass, Reactions** > **Mass** tab of the **Multiphase Model** dialog box after you specify the **Number of Mass Transfer Mechanisms**. See Hooking `DEFINE_LINEARIZED_MASS_TRANSFER` UDFs (p. 529) for details.

2.4.6. `DEFINE_MASS_TRANSFER`

2.4.6.1. Description

You can use `DEFINE_MASS_TRANSFER` when you want to model mass transfer in a multiphase problem. The mass transfer rate specified using a `DEFINE_MASS_TRANSFER` UDF is used to compute mass, momentum, energy, and species sources for the phases involved in the mass transfer. For problems in which species transport is enabled, the mass transfer will be from one species in one phase, to another species in another phase. If one of the phases does not have a mixture material associated with it, then the mass transfer will be with the bulk fluid of that phase.

> **Note:**
>
> By default, Fluent will attempt to hook mass transfer UDFs defined using `DEFINE_LINEARIZED_MASS_TRANSFER`. In order to hook a `DEFINE_MASS_TRANSFER` UDF to Fluent, you must first disable the following text command: `solve/set/advanced/linearized-mass-transfer-udf`.
>
> You may want to consider using the `DEFINE_LINEARIZED_MASS_TRANSFER` macro (`DEFINE_LINEARIZED_MASS_TRANSFER` (p. 193)) as it may provide a more robust solution, even though the results may be the same when converged.

2.4.6.2. Usage

`DEFINE_MASS_TRANSFER (name, c, mixture_thread, from_phase_index, from_spe-cies_index, to_phase_index, to_species_index)`

Important:

Note that all of the arguments to a `DEFINE` macro need to be placed on the same line in your source code. Splitting the `DEFINE` statement onto several lines will result in a compilation error.

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index of cell on the thread pointed to by `mixture_thread`.
`Thread *mixture_thread`	Pointer to mixture-level thread.
`int from_phase_index`	Index of phase from which mass is transferred.
`int from_species_index`	ID of species from which mass is transferred (ID= -1 if phase does not have mixture material).
`int to_phase_index`	Index of phase to which mass is transferred.
`int to_species_index`	ID of species to which mass is transferred (ID= -1 if phase does not have mixture material).

Function returns

`real`

There are seven arguments to `DEFINE_MASS_TRANSFER`: `name`, `c`, `mixture_thread`, `from_phase_index`, `from_species_index`, `to_phase_index`, `to_species_index`. You supply `name`, the name of the UDF. The variables `c`, `mixture_thread`, `from_phase_index`, `from_species_index`, `to_phase_index`, and `to_species_index` are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the real value of the mass transfer to the solver in the units of kg/m^3/s.

Important:

The arguments `from_species_index` and `to_species_index` are relevant for multiphase species transport problems only, and only if the respective phase has a mixture material associated with it.

2.4.6.3. Example

The following UDF, named `liq_gas_source`, specifies a simple mass transfer coefficient based on saturation temperature:

Important:

Note that in the example that follows, the `DEFINE_MASS_TRANSFER` statement is broken up into two lines for the sake of readability. In your source file, you must make sure that the `DEFINE` statement is on one line only.

```
/* UDF to define a simple mass transfer based on Saturation Temperature.
   The "from" phase is the gas phase and the "to" phase is the liquid phase */

  #include "udf.h"

  DEFINE_MASS_TRANSFER(liq_gas_source,cell,thread,from_index,from_species_index,to_index,to_species_index)
  {
    real m_lg;
    real T_SAT = 373.15;
    Thread *gas, *liq;
    gas = THREAD_SUB_THREAD(thread, from_index);
    liq = THREAD_SUB_THREAD(thread, to_index);
    m_lg = 0.0;

    if (C_T(cell, liq) > T_SAT)
      {                                              /* Evaporating */
        m_lg = -0.1*C_VOF(cell,liq)*C_R(cell,liq)*
               (C_T(cell,liq)-T_SAT)/T_SAT;
      }
    else if (C_T(cell, gas) < T_SAT)
      {                                              /* Condensing */
        m_lg = 0.1*C_VOF(cell,gas)*C_R(cell,gas)*
               (T_SAT-C_T(cell,gas))/T_SAT;
      }

    return (m_lg);
  }
```

2.4.6.4. Hooking a Mass Transfer UDF to Ansys Fluent

In order to hook a `DEFINE_MASS_TRANSFER` UDF to Fluent, you must first disable the following text command: `solve/set/advanced/linearized-mass-transfer-udf`.

After the UDF that you have defined using `DEFINE_MASS_TRANSFER` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `liq_gas_source`) will become visible and selectable under **Mass Transfer** in the **Phase Interaction** > **Heat, Mass, Reactions** > **Mass** tab of the **Multiphase Model** dialog box after you specify the **Number of Mass Transfer Mechanisms**. See Hooking `DEFINE_MASS_TRANSFER` UDFs (p. 530) for details.

2.4.7. `DEFINE_VECTOR_EXCHANGE_PROPERTY`

2.4.7.1. Description

You can use `DEFINE_VECTOR_EXCHANGE_PROPERTY` to specify custom slip velocities for the multiphase Mixture model or custom turbulent dispersion forces for the multiphase Eulerian model.

2.4.7.2. Usage

`DEFINE_VECTOR_EXCHANGE_PROPERTY (name, c, mixture_thread, phase_index_1, phase_index_2, vector_result)`

> **Important:**
>
> Note that all of the arguments to a `DEFINE` macro need to be placed on the same line in your source code. Splitting the `DEFINE` statement onto several lines will result in a compilation error.

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *mixture_thread`	Pointer to cell thread of mixture domain.
`int phase_index_1`	Identifier of the first phase from the pair of phases for which you want to customize a phase interaction property. An index of **0** corresponds to the primary phase and is incremented by one for each secondary phase.
`int phase_index_2`	Identifier of the second phase from the pair of phases for which you want to customize a phase interaction property. See `int phase_index_1` for more information.
`real *vector_result`	Pointer to slip velocity vector.

Function returns

`real`

There are six arguments to `DEFINE_VECTOR_EXCHANGE_PROPERTY`: `name`, `c`, `mixture_thread`, `phase_index_1`, `phase_index_2`, and `vector_result`. You supply `name`, the name of the UDF. `c`, `mixture_thread`, `phase_index_1`, `phase_index_2`, and `vector_result` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to set the values referenced by the `real` pointer to the slip velocity vector (`vector_result`) to the com-

ponents of the slip velocity vector (for example, `vector_result[0]`, `vector_result[1]` for a 2D problem).

> **Note:**
>
> **Ansys Fluent** passes values of phase identifiers in the same order as they appear in the **Phase Pairs** selection list in the **Multiphase Model** dialog box (**Phase Interaction** > **Forces** tab).

2.4.7.3. Example 1 — Custom Slip Velocity

The following UDF, named `custom_slip`, specifies a custom slip velocity in a two-phase mixture problem.

> **Important:**
>
> Note that in the example that follows, the `DEFINE_VECTOR_EXCHANGE_PROPERTY` statement is broken up into two lines for the sake of readability. In your source file, you must make sure that the `DEFINE` statement is on one line only.

```
/****************************************************************
   UDF for a defining a custom slip velocity in a 2-phase
   mixture problem
 ****************************************************************/

#include "udf.h"

DEFINE_VECTOR_EXCHANGE_PROPERTY(custom_slip,c,mixture_thread,phase_index_1,
    phase_index_2,vector_result)
{
  real grav[2] = {0., -9.81};
  real K = 5.e4;

  real pgrad_x, pgrad_y;

  Thread *pt, *st;/* thread pointers for primary and secondary phases*/

  pt = THREAD_SUB_THREAD(mixture_thread, phase_index_1);
  st = THREAD_SUB_THREAD(mixture_thread, phase_index_2);

  /* at this point the phase threads are known for primary (0) and
  secondary(1) phases */

  pgrad_x = C_DP(c,mixture_thread)[0];
  pgrad_y = C_DP(c,mixture_thread)[1];

  vector_result[0] =
    -(pgrad_x/K)
    +((((C_R(c, st)-
    C_R(c, pt))/K)*
    grav[0]);

  vector_result[1] =
    -(pgrad_y/K) +((((C_R(c, st)-
    C_R(c, pt))/K)*
```

```
        grav[1]);
}
```

Important:

Note that the pressure gradient macro `C_DP` is now obsolete. A more current pressure gradient macro can be found in Table 3.4: Macro for Cell Volume Defined in `mem.h` (p. 309).

2.4.7.4. Example 2 — Custom Turbulent Dispersion

The following UDF, named `custom_td`, specifies a custom turbulent dispersion force in a two-phase mixture problem.

```
/* this example uses DEFINE_VECTOR_EXCHANGE_PROPERTY to implement the lopez-de-bertodano model  */
#include "udf.h"

DEFINE_VECTOR_EXCHANGE_PROPERTY(custom_td, c, t, i, j, val)
{
  /* i -- liquid-phase; j -- vapor-phase */

  Thread **pt = THREAD_SUB_THREADS(t);
  Thread *ti = pt[i];
  Thread *tj = pt[j];
  real term;
  real rho_P = C_R(c,ti);
  real turb_k = (mp_ke_type==TURB_MP_KE_MIXTURE ||
              mp_rsm_type==TURB_MP_RSM_MIXTURE)?
              C_K(c,t) : C_K(c,pt[P_PHASE]);

  term = -rho_P*turb_k;
  term *= C_VOLUME(c,t);

  NV_VS(val, =, C_VOF_G(c,tj),*,term);

  if(NNULLP(THREAD_STORAGE(tj,SV_MP_DRIFT_S_P_COEFF)))
    P_DRIFT_COEFF(c,tj) = 0.0;

  if(NNULLP(THREAD_STORAGE(tj,SV_MP_DRIFT_COEFF)))
    S_DRIFT_COEFF(c,tj) = term;
}
```

2.4.7.5. Hooking a Vector Exchange Property UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_VECTOR_EXCHANGE_PROPERTY` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `custom_slip`) will become visible and selectable in the **Phase Interaction** tab of the **Multiphase Model** dialog box in Ansys Fluent. See Hooking `DEFINE_VECTOR_EXCHANGE_PROPERTY` UDFs (p. 532) for details.

2.5. Discrete Phase Model (DPM) `DEFINE` Macros

This section contains descriptions of `DEFINE` macros for the discrete phase model (DPM). Table 2.12: Quick Reference Guide for DPM-Specific DEFINE Macros (p. 202) provides a quick reference guide to the DPM `DEFINE` macros, the functions they define, and the dialog boxes where they are ac-

tivated in Ansys Fluent. Definitions of each `DEFINE` macro are contained in the `udf.h` header file. For your convenience, they are listed in Appendix B:`DEFINE` Macro Definitions (p. 693).

Table 2.12: Quick Reference Guide for DPM-Specific DEFINE Macros

Function	**DEFINE Macro**	**Dialog Box Activated In**
particle state at boundaries	DEFINE_DPM_BC	boundary condition (for example, **Velocity Inlet**)
body forces on particles	DEFINE_DPM_BODY_FORCE	**Discrete Phase Model**
drag coefficients between particles and fluid	DEFINE_DPM_DRAG	**Discrete Phase Model**
erosion and accretion rates	DEFINE_DPM_EROSION	**Discrete Phase Model**
heat and mass transfer of multicomponent particles to the gas phase	DEFINE_DPM_HEAT_MASS	**Set Injection Properties**
initializes injections	DEFINE_DPM_INJEC-TION_INIT	**Set Injection Properties**
custom laws for particles	DEFINE_DPM_LAW	**Custom Laws**
modifies what is written to the sampling plane output	DEFINE_DPM_OUTPUT	**Sample Trajectories**
material properties	DEFINE_DPM_PROPERTY	**Create/Edit Materials**
updates scalar every time a particle position is updated	DEFINE_DPM_SCALAR_UP-DATE	**Discrete Phase Model**
particle source terms	DEFINE_DPM_SOURCE	**Discrete Phase Model**
particle collisions algorithm	DEFINE_DPM_SPRAY_COL-LIDE	**Discrete Phase Model**
changes the criteria for switching between laws	DEFINE_DPM_SWITCH	**Custom Laws**
time step control for DPM simulation	DEFINE_DPM_TIMESTEP	**Discrete Phase Model**
equilibrium vapor pressure of vaporizing components of multicomponent particles	DEFINE_DPM_VP_EQUI-LIB	**Create/Edit Materials**
particle impingement regime selection criteria	DEFINE_IMPINGEMENT	**Discrete Phase Model**
particle impingement regimes	DEFINE_FILM_REGIME	**Discrete Phase Model**
distribution of splashed particles	DEFINE_SPLASHING_DISTRI-BUTION	**Discrete Phase Model**

Note:

If you plan to use the **Hybrid** parallel DPM tracking mode, the following special requirement must be considered for UDFs. Since multiple concurrent threads may be used in a single compute node (each being executed by its own CPU core, but all together forming a single process), access to most variables is shared among these concurrent threads. Therefore, every

UDF that has access to a `Tracked_Particle` pointer variable must never change a value that is neither part of the `Tracked_Particle` data structure nor of any other structure or array that is entirely specific to that particular `Tracked_Particle`. Values in memory that must not be changed by such UDFs to which Ansys Fluent passes a `Tracked_Particle` pointer include (but are not limited to):

- cell, face, and node values
- injection properties
- material properties
- any global variables

This section describes the following macros:

2.5.1. `DEFINE_DPM_BC`

2.5.1.1. Description

You can use `DEFINE_DPM_BC` to specify your own boundary conditions for particles. The function is executed every time a particle touches a boundary of the domain, except for symmetric or periodic boundaries. You can define a separate UDF (using `DEFINE_DPM_BC`) for each boundary.

2.5.1.2. Usage

`DEFINE_DPM_BC (name, tp, t, f, f_normal, dim)`

Argument Type	Description
`symbol name`	UDF name.
`Tracked_Particle *tp`	Pointer to the `Tracked_Particle` data structure which contains data related to the particle being tracked.
`Thread *t`	Pointer to the face thread the particle is currently hitting.
`face_t f`	Index of the face that the particle is hitting.
`real f_normal[]`	Array that contains the unit vector which is normal to the face.
`int dim`	Dimension of the flow problem. The value is `2` in 2D, for 2D-axisymmetric and 2D-axisymmetric-swirling flow, while it is `3` in 3D flows.

Function returns

`int`

There are six arguments to `DEFINE_DPM_BC`: `name`, `tp`, `t`, `f`, `f_normal`, and `dim`. You supply `name`, the name of the UDF. `tp`, `t`, `f`, `f_normal`, and `dim` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the new velocity of a particle after hitting the wall, and then return the status of the particle track (as an `int`), after it has hit the wall.

For the return status `PATH_ACTIVE`, the particle continues to track. For the return status `PATH_ABORT`, the particle will be stopped and considered to be aborted. For the return status `PATH_END`, the particle will be stopped as well, but considered to have escaped from the domain.

> **Important:**
>
> Pointer `tp` can be used as an argument to the particle-specific macros (defined in DPM Macros (p. 335)) to obtain information about particle properties.

2.5.1.3. Example 1

This example shows the usage of `DEFINE_DPM_BC` for a simple reflection at walls. It is similar to the reflection method executed by Ansys Fluent except that Ansys Fluent accommodates moving walls. The function must be executed as a compiled UDF.

The function assumes an ideal reflection for the normal velocity component (`nor_coeff = 1`) while the tangential component is damped (`tan_coeff = 0.3`). First, the angle of incidence is computed. Next, the normal particle velocity, with respect to the wall, is computed and subtracted from the particles velocity. The reflection is complete after the reflected normal velocity is added. The new particle velocity has to be stored in `TP_VEL0` to account for the change of particle velocity in the momentum balance for coupled flows. The function returns `PATH_ACTIVE` for inert particles while it stops particles of all other types.

```
/* reflect boundary condition for inert particles */
 #include "udf.h"
 DEFINE_DPM_BC(bc_reflect,tp,t,f,f_normal,dim)
```

```
{
   real alpha; /* angle of particle path with face normal */
   real vn=0.;
   real nor_coeff = 1.;
   real tan_coeff = 0.3;
   real normal[3];
   int i, idim = dim;
   real NV_VEC(x);

#if RP_2D
   /* dim is always 2 in 2D compilation. Need special treatment for 2d
     axisymmetric and swirl flows */
   if (rp_axi_swirl)
     {
        real R = sqrt(TP_POS(tp)[1]*TP_POS(tp)[1] +
             TP_POS(tp)[2]*TP_POS(tp)[2]);
        if (R > 1.e-20)
          {
             idim = 3;
             normal[0] = f_normal[0];
             normal[1] = (f_normal[1]*TP_POS(tp)[1])/R;
             normal[2] = (f_normal[1]*TP_POS(tp)[2])/R;
          }
        else
          {
             for (i=0; i<idim; i++)
               normal[i] = f_normal[i];
          }
     }
   else
#endif
   for (i=0; i<idim; i++)
      normal[i] = f_normal[i];

   if(TP_TYPE(tp) == DPM_TYPE_INERT)
     {
        alpha = M_PI/2. - acos(MAX(-1.,MIN(1.,NV_DOT(normal,TP_VEL(tp))/
             MAX(NV_MAG(TP_VEL(tp)),DPM_SMALL))));
        if ((NNULLP(t)) && (THREAD_TYPE(t) == THREAD_F_WALL))
             F_CENTROID(x,f,t);

        /* calculate the normal component, rescale its magnitude by
          the coefficient of restitution and subtract the change */

        /* Compute normal velocity. */
        for(i=0; i<idim; i++)
          vn += TP_VEL(tp)[i]*normal[i];

        /* Subtract off normal velocity. */
          for(i=0; i<idim; i++)
            TP_VEL(tp)[i] -= vn*normal[i];

        /* Apply tangential coefficient of restitution. */
          for(i=0; i<idim; i++)
            TP_VEL(tp)[i] *= tan_coeff;

        /* Add reflected normal velocity. */
          for(i=0; i<idim; i++)
            TP_VEL(tp)[i] -= nor_coeff*vn*normal[i];

        /* Store new velocity in TP_VEL0 of particle */
        for(i=0; i<idim; i++)
          TP_VEL0(tp)[i] = TP_VEL(tp)[i];

        return PATH_ACTIVE;
     }
 return PATH_ABORT;
}
```

2.5.1.4. Example 2

This example shows how to use `DEFINE_DPM_BC` for a wall impingement model. The function must be executed as a compiled UDF.

```
#include "udf.h"
#include "dpm.h"
#include "surf.h"
#include "random.h"

/* define a user-defined dpm boundary condition routine
 * bc_reflect: name
 * tp:  the tracked particle
 * t:   the touched face thread
 * f:   the touched face
 * f_normal:  normal vector of touched face
 * dim:  dimension of the problem (2 in 2d and 2d-axi-swirl, 3 in 3d)
 *
 * return is the status of the particle, see enumeration of Path_Status
 * in dpm.h
 */

#define V_CROSS(a,b,r)\
    ((r)[0] = (a)[1]*(b)[2] - (b)[1]*(a)[2],\
    (r)[1] = (a)[2]*(b)[0] - (b)[2]*(a)[0],\
    (r)[2] = (a)[0]*(b)[1] - (b)[0]*(a)[1])

DEFINE_DPM_BC(bc_wall_jet, tp, thread, f, f_normal, dim)
{
    /*
      Routine implementing the Naber and Reitz Wall
      impingement model (SAE 880107)
    */
    real normal[3];
    real tan_1[3];
    real tan_2[3];
    real rel_vel[3];
    real face_vel[3];
    real alpha, beta, phi, cp, sp;
    real rel_dot_n, vmag, vnew, dum;
    real weber_in, weber_out;
    int i, idim = dim;
    cxboolean moving = (SV_ALLOCATED_P (thread,SV_BND_GRID_V) &&
    SV_ALLOCATED_P (thread,SV_WALL_V));

#if RP_2D
    if (rp_axi_swirl)
      {
        real R = sqrt(TP_POS(tp)[1]*TP_POS(tp)[1] +
          TP_POS(tp)[2]*TP_POS(tp)[2]);
        if (R > 1.e-20)
          {
            idim = 3;
            normal[0] = f_normal[0];
            normal[1] = (f_normal[1]*TP_POS(tp)[1])/R;
            normal[2] = (f_normal[1]*TP_POS(tp)[2])/R;
          }
        else
          {
            for (i=0; i<idim; i++)
              normal[i] = f_normal[i];
          }
      }
    else
#endif

    for (i=0; i<idim; i++)
      normal[i] = f_normal[i];
```

```
   /*
     Set up velocity vectors and calculate the Weber number
     to determine the regime.
   */

for (i=0; i<idim; i++)
   {
      if (moving)
        face_vel[i] = WALL_F_VV(f,thread)[i] + BOUNDARY_F_GRID_VV(f,thread)[i];
      else
        face_vel[i] = 0.0;
        rel_vel[i] = TP_VEL(tp)[i] - face_vel[i];
   }

vmag = MAX(NV_MAG(rel_vel),DPM_SMALL);
rel_dot_n = MAX(NV_DOT(rel_vel,normal),DPM_SMALL);
weber_in = TP_RHO(tp) * SQR(rel_dot_n) * TP_DIAM(tp) /
  MAX(DPM_SURFTEN(tp), DPM_SMALL);

/*
Regime where bouncing occurs (We_in < 80).
(Data from Mundo, Sommerfeld and Tropea
 Int. J. of Multiphase Flow, v21, #2, pp151-173, 1995)  */

if (weber_in <= 80.)
  {
      weber_out = 0.6785*weber_in*exp(-0.04415*weber_in);
      vnew = rel_dot_n * (1.0 + sqrt(weber_out /
      MAX(weber_in, DPM_SMALL)));

 /*
 The normal component of the velocity is changed based
 on the experimental paper above (that is the Weber number
 is based on the relative velocity).
   */

      for (i=0; i < idim; i++)
        TP_VEL(tp)[i] = rel_vel[i] - vnew*normal[i] + face_vel[i];
  }

if (weber_in > 80.)
  {
      alpha = acos(-rel_dot_n/vmag);

 /*
 Get one tangent vector by subtracting off the normal
 component from the impingement vector, then cross the
 normal with the tangent to get an out-of-plane vector.
 */

 for (i=0; i < idim; i++)
   tan_1[i] = rel_vel[i] - rel_dot_n*normal[i];

 UNIT_VECT(tan_1,tan_1);
 V_CROSS(tan_1,normal,tan_2);

 /*
 beta is calculated by neglecting the coth(alpha)
 term in the paper (it is approximately right).
 */

 beta = MAX(M_PI*sqrt(sin(alpha)/(1.0-sin(alpha))),DPM_SMALL);
 phi= -M_PI/beta*log(1.0-cheap_uniform_random()*(1.0-exp(-beta)));
 if (cheap_uniform_random() > 0.5)
   phi = -phi;

 vnew = vmag;

 cp = cos(phi);
 sp = sin(phi);
```

```
   for (i=0; i < idim; i++)
     TP_VEL(tp)[i] = vnew*(tan_1[i]*cp + tan_2[i]*sp) + face_vel[i];


    }

 /*
   Subtract off from the original state.
 */
   for (i=0; i < idim; i++)
       TP_VEL0(tp)[i] = TP_VEL(tp)[i];

   if (DPM_STOCHASTIC_P(TP_INJECTION(tp)))
     {
        /* Reflect turbulent fluctuations also */
        /* Compute normal velocity. */

      dum = 0;
      for(i=0; i<idim; i++)
        dum += tp->V_prime[i]*normal[i];

          /* Subtract off normal velocity. */

       for(i=0; i<idim; i++)
          tp->V_prime[i] -= 2.*dum*normal[i];
     }
    return PATH_ACTIVE;
 }
```

2.5.1.5. Example 3

This example shows how to use `DEFINE_DPM_BC` for a simple filter model. It illustrates the proper way to handle particles that cross the boundary zone and enter a new cell. If the particle leaves the current cell, this information is stored in TP_CCELL(tp).

```
#include "udf.h"
#include "cxsurf.h"


/**
 *  filter particles that are above a specified cut-off size; allow others to pass
 **/

#define CUT_OFF_SIZE 10.0e-6   /* define the cut-off size in meters */

DEFINE_DPM_BC(my_filter, tp, t, f, f_normal, dim)
{
  Thread *t0 = THREAD_T0(t);
  Thread *t1 = THREAD_T1(t);
  cxboolean in_t0;
  CX_Cell_Id neigh;

  /* determine what cell/thread the particle is currently in */
  if ( (P_CELL_THREAD(tp)->id == t0->id) && (P_CELL(tp) == F_C0(f, t)) )
    in_t0 = TRUE;
  else
    in_t0 = FALSE;

  /* filter particles above cut-off size */
  if (P_DIAM(tp) > CUT_OFF_SIZE)
    return PATH_END;

  /* particle diameter is below cut-off size - set the new cell and thread */
  if (in_t0)
    STORE_CX_CELL(&neigh, F_C1(f, t), t1);
  else
    STORE_CX_CELL(&neigh, F_C0(f, t), t0);
```

```
  COPY_CELL_ID(TP_CCELL(tp), &neigh);
  calcFaceEquations(tp);

  return PATH_ACTIVE;
}
```

2.5.1.6. Example 4

This example shows how to use `DEFINE_DPM_BC` to reflect a Lagrangian wall film particle off of a boundary for cases that use the subtet particle tracking algorithm. If your UDF modifies one of the particle macros (mainly, `TP_POS(tp)`, `TP_CCELL(tp)`, `TP_FILM_FACE(tp)`, or `TP_FILM_THREAD(tp)`), you must ensure that the values of these particle variables are consistent with each other. For example, the particle position, `TP_POS(tp)`, must be within the cell specified by `TP_CCELL(tp)`, and the cell face specified by `TP_FILM_FACE(tp)` must be one of the faces of that cell. If the UDF does not modify any of these macros, then the current values will still be valid after the UDF is executed.

```
#include "udf.h"


/**
 *  reflect wall film particles off of a boundary when using subtet particle tracking
 **/


DEFINE_DPM_BC(my_reflect_bc, tp, tf, f, wall_normal, dim)
{
  int i;

  double dot_product;
  double normal[3];          /* wall film face normal */
  real edge_normal[3];       /* unit edge normal */

#if RP_2D
  normal[2] = 0.0;
#endif

  /* this UDF is only valid when using subtet tracking */
  if ( !dpm_par.use_subtet_tracking )
    return PATH_ABORT;

  /* get the wall film face normal */
  memcpy(normal, tp->gvtp.subtet->normal, 3 * sizeof(double));

  /* get the dot product of the film face normal and the reflected wall normal */
  dot_product = NV_DOT(normal, wall_normal);

  /* get the projection of the wall normal onto the wall film face normal*/
  for (i = 0; i < 3; i++)
    normal[i] *= dot_product;

  /* subtract the normal component from the original wall normal */
  for (i = 0; i < 3; i++)
    edge_normal[i] = wall_normal[i] - (real) normal[i];  /* in-plane (edge) normal */

  /* reflect the particle in the plane of the wall film face */
  UNIT_VECT(edge_normal, edge_normal);
  Reflect_Particle(tp, edge_normal, NULL, f, tf, NULL, -1);

  return PATH_ACTIVE;
}
```

2.5.1.7. Hooking a DPM Boundary Condition UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DPM_BC` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible in the appropriate boundary condition dialog box (for example, the **Velocity Inlet** dialog box) in Ansys Fluent. See Hooking `DEFINE_DPM_BC` UDFs (p. 535) for details on how to hook your `DEFINE_DPM_BC` UDF to Ansys Fluent.

2.5.2. `DEFINE_DPM_BODY_FORCE`

2.5.2.1. Description

You can use `DEFINE_DPM_BODY_FORCE` to specify a body force other than a gravitational or drag force on the particles.

2.5.2.2. Usage

`DEFINE_DPM_BODY_FORCE (name, tp, i)`

Argument Type	Description
`symbol name`	UDF name.
`Tracked_Particle *tp`	Pointer to the `Tracked_Particle` data structure which contains data related to the particle being tracked.
`int i`	An index (`0`, `1`, or `2`) that identifies the Cartesian component of the body force that is to be returned by the function.

Function returns

`real`

There are three arguments to `DEFINE_DPM_BODY_FORCE`: `name`, `tp`, and `i`. You supply `name`, the name of the UDF. `tp` and `i` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the `real` value of the acceleration due to the body force (in m/s^2) to the Ansys Fluent solver.

> **Important:**
>
> Pointer `tp` can be used as an argument to the macros defined in DPM Macros (p. 335) to specify a body force other than a gravitational or drag force on the particles. Note that the high-resolution particle tracking option requires that forces on the particle are continuous across cell boundaries. See Example (p. 211) for an example of `DEFINE_DPM_BODY_FORCE` usage for cases with the high high-resolution particle tracking option enabled.

2.5.2.3. Example

The following UDF computes the electrostatic force on a charged particle. It consists of two functions: one that enables the continuous interpolation of the electrostatic field to the particle position, and another that calculates the acceleration of the particle due to the electric field. This example assumes that the electric field components at the cell centroid are stored in user-defined memory.

`DEFINE_DPM_BODY_FORCE` is called at every particle time step in Ansys Fluent and requires a significant amount of CPU time to execute. For this reason, the UDF should be executed as a compiled UDF.

```
#include "udf.h"

/* setup interpolation of the electrostatic field */
DEFINE_DPM_INTERPOLATION(dpm_setup_interp, tp)
{
  /* interpolate the electrostatic field to the particle position */
  if ( HIGH_RESOLUTION_TRACKING_ENABLED )
  {

    if (N_UDM < 7)
      return;

    /* specify which UDM should be interpolated */
    int udm_components[] = { 4, 5, 6 };

    /* interpolate UDM 4, 5, and 6 to the particle position */
    DPM_Interpolate_UDM_from_Nodes(tp, 3, udm_components);
  }
}

/* calculate the electrostatic force (acceleration) on charged particles */
DEFINE_DPM_BODY_FORCE(particle_E_force, tp, i)
{
  /* From Bell and Hockberg correlation at 30k RPM and 70kV */
  const real chargedensity = 3.884e-3;    /* C/kg */
  real electric_field;

  if (N_UDM < 7)
    return 0.0;

  if ( HIGH_RESOLUTION_TRACKING_ENABLED )
    electric_field = TP_UDMI(tp, 4 + i);  /* E-field at the particle position */
  else
  {
    Thread *t = TP_CELL_THREAD(tp);
    cell_t c  = TP_CELL(tp);

    electric_field = C_UDMI(c, t, 4 + i); /* E-field at the cell centroid */
  }

  /* return the acceleration due to the electric force */
  return electric_field * chargedensity;
}
```

2.5.2.4. Hooking a DPM Body Force UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DPM_BODY_FORCE` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible in the **Discrete Phase Model** dialog box in Ansys Fluent. See Hooking `DEFINE_DPM_BODY_FORCE` UDFs (p. 536) for details on how to hook your `DEFINE_DPM_BODY_FORCE` UDF to Ansys Fluent.

2.5.3. `DEFINE_DPM_DRAG`

2.5.3.1. Description

You can use `DEFINE_DPM_DRAG` to specify the drag between particles and fluid as a dimensionless group ($\frac{18C_DRe}{24}$) as it appears in the drag force per unit particle mass:

$$F_D\left(u-u_p\right)=\frac{\mu}{\rho_p d_p^2}\underbrace{\frac{18C_DRe}{24}}_{DEFINE\_DPM\_DRAG}\left(u-u_p\right) \tag{2.19}$$

where

μ = viscosity.

ρ_p = particle density

D_p = particle diameter

C_D = drag coefficient

Re = Reynolds number

2.5.3.2. Usage

`DEFINE_DPM_DRAG` (name, Re, tp)

Argument Type	Description
`symbol name`	UDF name.
`real Re`	Particle Reynolds number based on the particle diameter and relative gas velocity.
`Tracked_Particle *tp`	Pointer to the `Tracked_Particle` data structure which contains data related to the particle being tracked.

Function returns

`real`

There are three arguments to `DEFINE_DPM_DRAG`: `name`, `Re`, and `tp`. You supply `name`, the name of the UDF. `Re` and `tp` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the `real` value of the drag force on a particle. The value returned to the solver must be dimensionless and represent `18 * Cd * Re / 24`.

> **Important:**
>
> Pointer `tp` can be used as an argument to the macros defined in DPM Macros (p. 335) to obtain information about particle properties (for example, injection properties). Note that the high-resolution particle tracking requires forces on the particle, including the drag force, to be continuous across cell boundaries. Discontinuities in the force can result in particles being lost or stuck at interior cell faces.

2.5.3.3. Example

The following UDF, named `particle_drag_force`, computes the drag force on a particle and is a variation of the body force UDF presented in `DEFINE_DPM_BODY_FORCE` (p. 210). The flow is the same, but a different curve is used to describe the particle drag. `DEFINE_DPM_DRAG` is called at every particle time step in Ansys Fluent, and requires a significant amount of CPU time to execute. For this reason, the UDF should be executed as a compiled UDF.

```
/**************************************************************************
   UDF for computing particle drag coefficient (18 Cd Re/24)
   curve as suggested by R. Clift, J. R. Grace and M. E. Weber
   "Bubbles, Drops, and Particles" (1978)
 **************************************************************************/

#include "udf.h"

DEFINE_DPM_DRAG(particle_drag_force,Re,tp)
{
    real w, drag_force;
    if (Re < 0.01)
    {
      drag_force=18.0;
      return (drag_force);
    }
    else if (Re < 20.0)
    {
      w = log10(Re);
      drag_force = 18.0 + 2.367*pow(Re,0.82-0.05*w) ;
      return (drag_force);
    }
    else
    /* Note: suggested valid range 20 < Re < 260 */
    {
      drag_force = 18.0 + 3.483*pow(Re,0.6305) ;
      return (drag_force);
    }
}
```

2.5.3.4. Hooking a DPM Drag Coefficient UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DPM_DRAG` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible in the **Set Injection Properties** dialog box in Ansys Fluent. See Hooking `DEFINE_DPM_DRAG` UDFs (p. 537) for details on how to hook your `DEFINE_DPM_DRAG` UDF to Ansys Fluent.

2.5.4. `DEFINE_DPM_EROSION`

2.5.4.1. Description

You can use `DEFINE_DPM_EROSION` to specify the erosion and accretion rates calculated as the particle stream strikes a wall surface or hits a porous jump surface. The function is called when the particle encounters a reflecting surface.

2.5.4.2. Usage

`DEFINE_DPM_EROSION (name, tp, t, f, normal, alpha, Vmag, mdot)`

Argument Type	Description
`symbol name`	UDF name.
`Tracked_Particle *tp`	Pointer to the `Tracked_Particle` data structure which contains data related to the particle being tracked.
`Thread *t`	Pointer to the face thread the particle is currently hitting.
`face_t f`	Index of the face that the particle is hitting.
`real normal[]`	Array that contains the unit vector that is normal to the face.
`real alpha`	Variable that represents the impact angle between the particle path and the face (in radians).
`real Vmag`	Variable that represents the magnitude of the particle velocity (in m/s).
`real mdot`	Steady flow simulations: Current flow rate of the particle stream per number of stochastic tries as it hits the face (kg/s).
	Transient flow simulations: Mass of the parcel hitting the face (kg).
	Axial symmetric configurations: The value is per $2/\pi$.

Function returns

`void`

There are eight arguments to `DEFINE_DPM_EROSION`: `name`, `tp`, `t`, `f`, `normal`, `alpha`, `Vmag`, and `mdot`. You supply `name`, the name of the UDF. `tp`, `t`, `f`, `normal`, `alpha`, `Vmag`, and `mdot` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the values for the erosion rate and/or accretion rate and store the values at the faces in `F_STOR-AGE_R_XV(f, t, SV_DPMS_EROSION, EROSION_UDF)` and `F_STORAGE_R (f,t,SV_DPMS_ACCRETION)`, respectively.

Important:

Pointer `tp` can be used as an argument to the macros defined in DPM Macros (p. 335) to obtain information about particle properties (for example, injection properties).

2.5.4.3. Example

The following is an example of a compiled UDF that uses `DEFINE_DPM_EROSION` to extend postprocessing of wall impacts in a 2D axisymmetric flow. It provides additional information on how the local particle deposition rate depends on the diameter and normal velocity of the particles. It is based on the assumption that every wall impact leads to more accretion, and, therefore, every trajectory is removed at its first wall impact. (This is done by setting the flag `P_FL_REMOVED` within `DEFINE_DPM_EROSION`.) User-defined memory locations (UDMLs) are used to store and visualize the following:

- number of wall impacts since UDMLs were reset. (Resetting is typically done at the beginning of an Ansys Fluent session by the use of `DEFINE_ON_DEMAND` in order to avoid the use of uninitialized data fields. Resetting prevents the addition of sampled data being read from a file).

- average diameter of particles hitting the wall.

- average radial velocity of particles.

> **Note:**
>
> Before tracing the particles, you will have to reset the UDMLs and assign the global domain pointer by executing the DEFINE_ON_DEMAND UDF function, reset_UDM(), which appears at the end of this Example.

```
/*************************************************************************
   UDF for extending postprocessing of wall impacts
 *************************************************************************/
 #include "udf.h"

#define MIN_IMPACT_VELO -1000.
/* Minimum particle velocity normal to wall (m/s) to allow Accretion.*/

Domain *domain; /* Get the domain pointer and assign it later to domain*/

enum   /* Enumeration of used User-Defined Memory Locations. */
  {
    NUM_OF_HITS, /* Number of particle hits into wall face considered.*/
    AVG_DIAMETER, /* Average diameter of particles that hit the wall. */
    AVG_RADI_VELO, /* Average radial velocity of "" "" ------------ */
    NUM_OF_USED_UDM
  };

int UDM_checked = 0;  /* Availability of UDMLs checked? */

void reset_UDM_s(void); /* Function to follow below. */

int check_for_UDM(void)  /* Check for UDMLs availability... */
{
  Thread *t;
  if (UDM_checked)
    return UDM_checked;

  thread_loop_c(t,domain) /* We require all cell threads to */
  {        /* provide space in memory for UDML */
    if (FLUID_THREAD_P(t))   if (NULLP(THREAD_STORAGE(t,SV_UDM_I)))
       return 0;
  }
  UDM_checked = 1;  /* To make the following work properly... */
  reset_UDM_s();   /* This line will be executed only once, */
  return UDM_checked; /* because check_for_UDM checks for */
}      /* UDM_checked first. */

void reset_UDM_s(void)
{
  Thread *t;
  cell_t c;
  face_t f;
  int  i;
  if (!check_for_UDM()) /* Dont do it, if memory is not available. */
    return;
  Message("Resetting User Defined Memory...\n");
  thread_loop_f(t, domain)
    {
      if (NNULLP(THREAD_STORAGE(t,SV_UDM_I)))
        {
          begin_f_loop(f,t)
          {
            for (i = 0; i < NUM_OF_USED_UDM; i++)
             F_UDMI(f,t,i) = 0.;
          }
```

```
                end_f_loop(f, t)
             }
        else
           {
              Message("Skipping FACE thread no. %d..\n", THREAD_ID(t));
           }
      }
  thread_loop_c(t,domain)
     {
        if (NNULLP(THREAD_STORAGE(t,SV_UDM_I)))
           {
              begin_c_loop(c,t)
              {
                for (i = 0; i < NUM_OF_USED_UDM; i++)
                  C_UDMI(c,t,i) = 0.;
              }
              end_c_loop(c,t)
           }
        else
           {
              Message(" Skipping CELL thread no. %d..\n", THREAD_ID(t));
           }
     }      /* Skipping Cell Threads can happen if the user */
  /* uses reset_UDM prior to initializing. */
  Message(" --- Done.\n");
}

DEFINE_DPM_EROSION(dpm_accr, tp, t, f, normal, alpha, Vmag, Mdot)
{
  real A[ND_ND], area;
  int num_in_data;
  Thread *t0;
  cell_t c0;
  real imp_vel[3], vel_ortho;

#if RP_2D
  if (rp_axi)
     {
        real radi_pos[3], radius;
        /* The following is ONLY valid for 2d-axisymmetric calculations!!! */
        /* Additional effort is necessary because DPM tracking is done in */
        /* THREE dimensions for TWO-dimensional axisymmetric calculations. */

        radi_pos[0] = TP_POS(tp)[1];  /* Radial location vector. */
        radi_pos[1] = TP_POS(tp)[2];  /* (Y and Z in 0 and 1...) */
        radius = NV_MAG(radi_pos);
        NV_VS(radi_pos, =, radi_pos, /, radius);
        /* Normalized radius direction vector.*/
        imp_vel[0] = TP_VEL(tp)[0]; /* Axial particle velocity component. */
        imp_vel[1] = NVD_DOT(radi_pos, TP_VEL(tp)[1], TP_VEL(tp)[2], 0.);
     }
  else
#endif
    NV_V(imp_vel, =, TP_VEL(tp));

  /* Dot product of normalized radius vector and y & z components */
  /* of particle velocity vector gives _radial_ particle velocity */
  /* component */
  vel_ortho = NV_DOT(imp_vel, normal); /*velocity orthogonal to wall */

  if (vel_ortho < MIN_IMPACT_VELO) /* See above, MIN_IMPACT_VELO */
    return;

  if (!UDM_checked)   /* We will need some UDMs, */
    if (!check_for_UDM()) /* so check for their availability.. */
      return;     /* (Using int variable for speed, could */
  /* even just call check_for UDFM().) */

  c0 = F_C0(f,t);
  t0 = THREAD_T0(t);
```

```
    F_AREA(A,f,t);
    area = NV_MAG(A);
    F_STORAGE_R(f,t,SV_DPMS_ACCRETION) += Mdot / area;

    MARK_TP(tp, P_FL_REMOVED); /* Remove particle after accretion */

    /* F_UDMI not allocated for porous jumps */
    if (THREAD_TYPE(t) == THREAD_F_JUMP)
      return;

    num_in_data  = F_UDMI(f,t,NUM_OF_HITS);

    /* Average diameter of particles that hit the particular wall face:*/
    F_UDMI(f,t,AVG_DIAMETER) = (TP_DIAM(tp)
          + num_in_data * F_UDMI(f,t,AVG_DIAMETER))
                              / (num_in_data + 1);
    C_UDMI(c0,t0,AVG_DIAMETER) = F_UDMI(f,t,AVG_DIAMETER);

    /* Average velocity normal to wall of particles hitting the wall:*/
    F_UDMI(f,t,AVG_RADI_VELO) = (vel_ortho
          + num_in_data * F_UDMI(f,t,AVG_RADI_VELO))
                              / (num_in_data + 1);
    C_UDMI(c0,t0,AVG_RADI_VELO) = F_UDMI(f,t,AVG_RADI_VELO);

    F_UDMI(f, t, NUM_OF_HITS) = num_in_data + 1;
    C_UDMI(c0,t0,NUM_OF_HITS) = num_in_data + 1;


}

DEFINE_ON_DEMAND(reset_UDM)
{
  /* assign domain pointer with global domain */
  domain = Get_Domain(1);
  reset_UDM_s();
}
```

2.5.4.4. Hooking an Erosion/Accretion UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DPM_EROSION` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible in the **Discrete Phase Model** dialog box in Ansys Fluent. See Hooking `DEFINE_DPM_EROSION` UDFs (p. 538) for details on how to hook your `DEFINE_DPM_EROSION` UDF to Ansys Fluent.

2.5.5. `DEFINE_DPM_HEAT_MASS`

2.5.5.1. Description

You can use `DEFINE_DPM_HEAT_MASS` to specify the heat and mass transfer of multicomponent particles to the gas phase.

When a `DEFINE_DPM_HEAT_MASS` UDF is activated, then the number of species that can be referenced and interact with the particles via the UDF is limited to those with a species index less than the maximum UDF species number, defined using the TUI command `define/models/dpm/options/maximum-udf-species`. The default number for `maximum-udf-species` is 50.

2.5.5.2. Usage

`DEFINE_DPM_HEAT_MASS (name, tp, C_p, hgas, hvap, cvap_surf, Z, dydt, dzdt)`

Argument Type	Description
`symbol name`	UDF name.
`Tracked_Particle *tp`	Pointer to the `Tracked_Particle` data structure which contains data related to the particle being tracked.
`real C_p`	Particle heat capacity.
`real *hgas`	Enthalpies of vaporizing gas phase species.
`real *hvap`	Vaporization enthalpies of vaporizing components.
`real *cvap_surf`	Vapor equilibrium concentrations of vaporizing components.
`real Z`	Compressibility, Z^V
`real *dydt`	Source terms of the particle temperature and component masses.
`dpms_t *dzdt`	Source terms of the gas phase enthalpy and species masses.

Function returns

`void`

There are eight arguments to `DEFINE_DPM_HEAT_MASS`: `name`, `tp`, `C_p`, `hgas`, `hvap`, `cvap_surf`, `Z`, `dydt`, and `dzdt`. You supply `name`, the name of the UDF. `tp`, `C_p`, `hgas`, `hvap`, `cvap_surf`, and `Z` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the particle and gas phase source terms and store the values in `dydt` and `dzdt`, respectively.

2.5.5.3. Example

The following is an example of a compiled UDF that uses `DEFINE_DPM_HEAT_MASS`. It implements the source terms for the following:

Source Term	Variable	Unit
particle temperature	`dydt[0]`	K/s
particle component mass	`dydt[1..]`	kg/s
gas phase enthalpy	`dzdt->energy`	J/s
gas phase species mass	`dzdt->species[0..]`	kg/s

```
/************************************************************************
   UDF for defining the heat and mass transport for
   multicomponent particle vaporization
 ************************************************************************/
#include "udf.h"
#include "dpm_mem.h"
```

```
DEFINE_DPM_HEAT_MASS(multivap,tp,Cp,hgas,hvap,cvap_surf,Z,dydt,dzdt)
{
  int ns;
  Material *sp;
  real dens_total = 0.0;      /* total vapor density*/
  real P_total = 0.0;       /* vapor pressure */
  int nc = TP_N_COMPONENTS(tp);    /* number of particle components */
  Thread *t0 = TP_CELL_THREAD(tp);    /* thread where the particle is in*/
  Material *gas_mix = THREAD_MATERIAL(DPM_THREAD(t0, tp)); /* gas mixture
  material */
  Material *cond_mix = TP_MATERIAL(tp); /* particle mixture material*/
  cphase_state_t *c = &(tp->cphase[0]); /* cell information of particle location*/
  real molwt[MAX_SPE_EQNS]; /* molecular weight of gas species */
  real Tp = TP_T(tp);    /* particle temperature */
  real mp = TP_MASS(tp);    /* particle mass */
  real molwt_bulk = 0.;   /* average molecular weight in bulk gas */
  real Dp = DPM_DIAM_FROM_VOL(mp / TP_RHO(tp)); /* particle diameter */
  real Ap = DPM_AREA(Dp);       /* particle surface */
  real Pr = c->sHeat * c->mu / c->tCond;   /* Prandtl number */
  real Nu = 2.0 + 0.6 * sqrt(tp->Re) * pow(Pr, 1./3.); /* Nusselt number */
  real h = Nu * c->tCond / Dp;     /* Heat transfer coefficient*/
  real dh_dt = h * (c->temp - Tp) * Ap;  /* heat source term*/
  dydt[0] += dh_dt / (mp * Cp);
  dzdt->energy -= dh_dt;
  mixture_species_loop(gas_mix,sp,ns)
  {
     molwt[ns] = MATERIAL_PROP(sp,PROP_mwi); /* molecular weight of gas
        species */
   molwt_bulk += c->yi[ns] / molwt[ns]; /* average molecular weight */
  }

/* prevent division by zero */
molwt_bulk = MAX(molwt_bulk,DPM_SMALL);

for (ns = 0; ns < nc; ns++)
  {
     int gas_index = TP_COMPONENT_INDEX_I(tp,ns);  /* gas species index of
     vaporization */
 if(gas_index >= 0)
   {
      /* condensed material */
      Material * cond_c = MIXTURE_COMPONENT(cond_mix, ns);
      /* vaporization temperature */
      real vap_temp = MATERIAL_PROP(cond_c,PROP_vap_temp);
      /* diffusion coefficient */
      real D = DPM_BINARY_DIFFUSIVITY(tp,cond_c,TP_T(tp));
      /* Schmidt number */
      real Sc = c->mu / (c->rho * D);
      /* mass transfer coefficient */
      real k = (2. + 0.6 * sqrt(tp->Re) * pow(Sc, 1./3.)) * D / Dp;
      /* bulk gas concentration (ideal gas) */
      real cvap_bulk = c->pressure / UNIVERSAL_GAS_CONSTANT / c->temp
      * c->yi[gas_index] / molwt_bulk / solver_par.molWeight[gas_index];
      /* vaporization rate */
      real vap_rate = k * molwt[gas_index] * Ap
      * (cvap_surf[ns] - cvap_bulk);
      /* no vaporization below vaporization temperature, no condensation */
      if (Tp < vap_temp || vap_rate < 0.0)
        vap_rate = 0.;

      dydt[1+ns] -= vap_rate;
      dzdt->species[gas_index] += vap_rate;
      /* dT/dt = dh/dt / (m Cp)*/
      dydt[0] -= hvap[gas_index] * vap_rate / (mp * Cp);
      /* gas enthalpy source term */
      dzdt->energy += hgas[gas_index] * vap_rate;

      P_total += cvap_surf[ns];
      dens_total += cvap_surf[ns] * molwt[gas_index];
    }
  }
```

```
    /* multicomponent boiling */
    P_total *= Z * UNIVERSAL_GAS_CONSTANT * Tp;
    if (P_total > c->pressure && dydt[0] > 0.)
      {
        real h_boil = dydt[0] * mp * Cp;
        /* keep particle temperature constant */
        dydt[0] = 0.;
        for (ns = 0; ns < nc; ns++)
          {
            int gas_index = TP_COMPONENT_INDEX_I(tp,ns);
            if (gas_index >= 0)
              {
                real boil_rate = h_boil / hvap[gas_index] * cvap_surf[ns] *
                  molwt[gas_index] / dens_total;
                /* particle component mass source term */
                dydt[1+ns] -= boil_rate;
                /* fluid species source */
                dzdt->species[gas_index] += boil_rate;
                /* fluid energy source */
                dzdt->energy += hgas[gas_index] * boil_rate;
              }
          }
      }
  }
```

2.5.5.4. Hooking a DPM Particle Heat and Mass Transfer UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DPM_HEAT_MASS` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `multivap`) will become visible in the **Set Injection Properties** dialog box in Ansys Fluent. See Hooking `DEFINE_DPM_HEAT_MASS` UDFs (p. 540) for details on how to hook your `DEFINE_DPM_HEAT_MASS` UDF to Ansys Fluent.

2.5.6. `DEFINE_DPM_INJECTION_INIT`

2.5.6.1. Description

You can use `DEFINE_DPM_INJECTION_INIT` to initialize a particle's injection properties such as location, diameter, and velocity.

2.5.6.2. Usage

`DEFINE_DPM_INJECTION_INIT` (name, I)

Argument Type	Description
`symbol name`	UDF name.
`Injection *I`	Pointer to the `Injection` structure which is a container for the particles being created. This function is called twice for each `Injection` before the first DPM iteration, and then called once for each `Injection` before the particles are injected into the domain at each subsequent DPM iteration.

Function returns

```
void
```

There are two arguments to `DEFINE_DPM_INJECTION_INIT`: `name` and `I`. You supply `name`, the name of the UDF. `I` is a variable that is passed by the Ansys Fluent solver to your UDF.

> **Note:**
>
> When the `DEFINE_DPM_INJECTION_INIT` UDF modifies the diameter (`PP_DIAM(p)`, `PP_INIT_DIAM(p)`) or the mass (`PP_MASS(p)`, `PP_INIT_MASS(p)`) of the particle, then the other one of these two must be adjusted accordingly in order for the two to be consistent with the particle density (`PP_RHO(p)`). The latter should not be changed because it must remain consistent with the corresponding material property definition.

2.5.6.3. Example

The following UDF, named `init_bubbles`, initializes particles on a surface injection due to a surface reaction. This function must be executed as a compiled UDF. Note that if you are going to use this UDF in a transient simulation to compute transient particles, you will need to replace `loop(p, I->p)` with `loop(p, I->p_init)`. Transient particle initialization cannot be performed with a loop over `I->p`.

```
/***********************************************************************
   UDF that initializes particles on a surface injection due
   to a surface reaction
 ***********************************************************************/

#include "udf.h"
#include "surf.h" /* RP_CELL and RP_THREAD are defined in surf.h */

#define REACTING_SURFACE_ID 2
#define MW_H2 2
#define STOIC_H2 1

/* ARRHENIUS CONSTANTS */
#define PRE_EXP 1e+15
#define ACTIVE 1e+08
#define BETA 0.0

real arrhenius_rate(real temp)
{
  return PRE_EXP*pow(temp,BETA)*exp(-ACTIVE/(UNIVERSAL_GAS_CONSTANT*temp));
}

/* Species numbers. Must match order in Ansys Fluent dialog box */
#define HF 0

/* Reaction Exponents */
#define HF_EXP 2.0


/* Reaction Rate Routine used in UDF */

real reaction_rate(cell_t c, Thread *cthread,real mw[],real yi[])

/* Note that all arguments in the reaction_rate function call in your .c source file
   MUST be on the same line or a compilation error will occur */
{
  real concenHF = C_R(c,cthread)*yi[HF]/mw[HF];
  return arrhenius_rate(C_T(c,cthread))*pow(concenHF,HF_EXP);
}

real contact_area(cell_t c,Thread *t,int s_id,int *n);
```

```
    DEFINE_DPM_INJECTION_INIT(init_bubbles,I)
   {
     int count,i;
     real area, mw[MAX_SPE_EQNS], yi[MAX_SPE_EQNS];
     /* MAX_SPE_EQNS is an Ansys Fluent constant in materials.h */
     Particle *p;
     cell_t cell;
     Thread *cthread;
     Material *mix, *sp;
     Message("Initializing Injection: %s\n",I->name);
     loop(p,I->p) /* Standard Ansys Fluent Looping Macro to get particle
            streams in an Injection */
       {
          cell = PP_CELL(p);  /* Get the cell and thread that
                               * the particle is currently in */
          cthread = PP_CELL_THREAD(p);
          /* Set up molecular weight & mass fraction arrays */
          mix = THREAD_MATERIAL(cthread);
          mixture_species_loop(mix,sp,i)
              {
               mw[i] = MATERIAL_PROP(sp,PROP_mwi);
               yi[i] = C_YI(cell,cthread,i);
              }
        area = contact_area(cell, cthread, REACTING_SURFACE_ID,&count);
        /* Function that gets total area of REACTING_SURFACE faces in
          contact with cell */
        /* count is the number of contacting faces, and is needed
          to share the total bubble emission between the faces  */
        if (count > 0) /* if cell is in contact with REACTING_SURFACE */
          {
           PP_FLOW_RATE(p) = (area *MW_H2* STOIC_H2 *
              reaction_rate(cell, cthread, mw, yi))/
              (real)count; /* to get correct total flow
               rate when multiple faces contact the same cell */
           PP_DIAM(p) = 1e-3;
           PP_RHO(p) = 1.0;
           PP_MASS(p) = PP_RHO(p)*M_PI*pow(PP_DIAM(p),3.0)/6.0;
          }
        else
           PP_FLOW_RATE(p) = 0.0;
          }
      }

real contact_area(cell_t c, Thread *t, int s_id, int *n)
{
  int i = 0;
  real area = 0.0, A[ND_ND];
  *n = 0;
  c_face_loop(c,t,i)
    {
     if(THREAD_ID(C_FACE_THREAD(c,t,i)) == s_id)
         {
         (*n)++;
         F_AREA(A,C_FACE(c,t,i), C_FACE_THREAD(c,t,i));
         area += NV_MAG(A);
         }
    } return area;
}
```

2.5.6.4. Using `DEFINE_DPM_INJECTION_INIT` with an unsteady injection file

When you are using a DPM particle sampling file or a VOF-to-DPM lump conversion transcript file as an unsteady injection file in a separate simulation, you can add more injection property data to such a file in extra columns. To do this, you can, for example, use a `DEFINE_DPM_OUTPUT` UDF during either sampling or VOF-to-DPM lump conversion (see Sampling of Trajectories in the *Fluent*

You can then use a
`DEFINE_DPM_INJECTION_INIT` UDF to read that additional information from the file's extra
columns and initialize the particles that Ansys Fluent generates from the same unsteady injection
file.

In the context of parallel processing, different particles generated from the unsteady injection file
may be located in different partitions. Therefore, the `DEFINE_DPM_INJECTION_INIT` UDF must
first obtain the contents of the file on all compute-node processes and then, for each new particle
found in any partition, identify the corresponding line in the unsteady injection file. Only then can
the additional information from the file be used, for example, to initialize user-defined particle
properties.

The easiest way to obtain the contents of the file on every compute-node process is to read the
file independently on every process, as shown in the example below. Note that Ansys Fluent itself
reads the file only on `compute-node 0` and then uses inter-process communication to automat-
ically distribute the standard column data across all compute-node processes. You could implement
a similar approach in your UDF for reading and sending information from the extra columns to
other compute-node processes, but it involves extra programming that will not be further discussed
here.

The following example can be used as a template for many purposes. The `injudf` UDF first reads
the particle id (after the colon near the end of each line in the injection file) and then uses this id
to initialize the first DPM particle user real in the new particle. For a typical purpose, there would
be additional columns in the file, and you would use some extra `fscanf(...)` statement in this
UDF to read them from the file.

```
#define USE_MY_OWN_FILE _NT

#if USE_MY_OWN_FILE
# define USE_FLUENT_IO_API  0
# define USE_UDF_PAR_IO  0
#endif

#include "udf.h"

/* Following definition divides by EPSILON to cancel
 * that from the definition of REAL_EQUAL_SCALED:
 */
#define  REL_RESOL_IN_INJ_FILE 0.0001
/* If, for example, the file format is...:
 *  (( 4.9821e+00  -1.0061e+00  4.2000e+00  -1.6938e-05 ...
 * then there are five significant digits
 * the "relative resolution" is 0.0001 = 1.e-4
 */

#define MY_REAL_EQUAL(a, b)  \
   REAL_EQUAL_SCALED((a), (b), (REL_RESOL_IN_INJ_FILE / EPSILON))

DEFINE_DPM_INJECTION_INIT(injudf, I)
{
  Particle *p;
  char rest[1024];
  FILE *my_inj_file_ptr = I->inj_fil_ptr;

  if (dpm_par.n_user_reals < 1)
  {
    Message0("\n\nWARNING: UDF 'injudf' cannot operate, "
              "needs at least one DPM user-real!!\n\n");
    return;
  }

#if USE_MY_OWN_FILE
```

```
    my_inj_file_ptr = fopen(I->injection_file, "r");
    /* Cannot use the open file pointer provided
     * by the Fluent executable.
     */
#endif

    if (NULLP(my_inj_file_ptr))
    {
      Message0("\n\nWARNING: UDF 'injudf' cannot operate, "
                   "file '%s' is not open or could not be "
                   "opened for reading!!\n\n",
               I->injection_file);
      return;
    }

#if USE_MY_OWN_FILE
  /* "read away" the file header: */
  fgets(rest, 1023, my_inj_file_ptr);
  fgets(rest, 1023, my_inj_file_ptr);
#elif 00
  rewind(my_inj_file_ptr);
#endif

loop(p, I->p_init)
  {
    while ( ! feof(my_inj_file_ptr))
    {
      real file_x, file_y, file_z,
           file_u, file_v, file_w,
           file_d, file_T, file_M,
           file_m, file_n, file_t, file_f = 0.;

      int nmatch = fscanf(my_inj_file_ptr,
                      "((%lg%lg%lg%lg%lg%lg%lg%lg%lg%lg%lg%lg%lg",
                      &file_x, &file_y, &file_z,
                      &file_u, &file_v, &file_w,
                      &file_d, &file_T, &file_M,
                      &file_m, &file_n, &file_t, &file_f);

      if (13 != nmatch)
      {
        CX_Message("\nnode-%d: WARNING: UDF 'injudf': Could not "
          "read from the file '%s' as expected -- bailing out!\n",
                   myid, I->injection_file);
      }

      /* Read additional numerical columns here…! */

      /* Read the rest of the line into a string buffer: */
      fgets(rest, 1023, my_inj_file_ptr);

      file_f -= I->start_flow_time_in_unst_file;

      if (0. < I->repeat_intrvl_from_unst_file)
        if (file_f > I->repeat_intrvl_from_unst_file)
        {
          rewind(my_inj_file_ptr);
          /* "read away" the file header: */
          fgets(rest, 1023, my_inj_file_ptr);
          fgets(rest, 1023, my_inj_file_ptr);
          file_f = 0.;
          continue;
        }

      file_f += I->unsteady_start;

      if (0. < I->repeat_intrvl_from_unst_file)
        while (file_f < dpm_par.time)
          file_f += I->repeat_intrvl_from_unst_file;

      if (MY_REAL_EQUAL(PP_POS(p)[0],          file_x) &&
```

```
            MY_REAL_EQUAL(PP_POS(p)[1],        file_y) &&
            MY_REAL_EQUAL(PP_POS(p)[2],        file_z) &&
            MY_REAL_EQUAL(PP_VEL(p)[0],        file_u) &&
            MY_REAL_EQUAL(PP_VEL(p)[1],        file_v) &&
            MY_REAL_EQUAL(PP_VEL(p)[2],        file_w) &&
            MY_REAL_EQUAL(PP_DIAM(p),          file_d) &&
            MY_REAL_EQUAL(PP_T(p),             file_T) &&
            MY_REAL_EQUAL(PP_FLOW_RATE(p),     file_M) &&
        /* The following PP_...(p) values aren't yet known in
         * parallel and will be calculated from P_FLOW_RATE(p)
         * later:
         * MY_REAL_EQUAL(PP_MASS(p) * PP_N(p), file_M) &&
         * MY_REAL_EQUAL(PP_MASS(p),          file_m) &&
         * MY_REAL_EQUAL(PP_N(p),             file_n) &&
         */
            MY_REAL_EQUAL(PP_INJ_DURATIME(p),  file_t) &&
            MY_REAL_EQUAL(PP_INJ_FLOWTIME(p),  file_f))
      {
        /* From the rest of the line, determine some old part_id,
         * add 7000 and assign to the new particle's user real:
         */
        sscanf(strchr(rest, ':') + 1, "%lg", &(PP_USER_REAL(p, 0)));
        PP_USER_REAL(p, 0) += 7000.;
        break;
      }
    }   /* while(TRUE) */

    /* End of file, but yet another particle to do? */
    if (feof(my_inj_file_ptr) &&  ! NULLP(p->next))
    {
      CX_Message("\n\n==========================================="
               "===========================================\n"
                 "====  WARNING:  injudf could not process all"
                 " new particles on compute-node: %d..!  ====\n"
                  "==========================================="
               "===========================================\n\n",
               myid);
      break;
    }
  }
}

#if USE_MY_OWN_FILE
  fclose(my_inj_file_ptr);
#endif
}
```

As mentioned above, in a parallel environment, the UDF must identify for every particle on each compute-node process which line from the injection file describes the particle. Therefore, the UDF reads all information from the file into every compute-node process and then compares the values from the standard columns against the properties of each particle in order to find the matching file entry for every particle. Once the correct line has been found, the UDF processes additional particle property information and assigns it to the particle.

The preprocessor macro _NT, which is used in the definition of the macro USE_MY_OWN_FILE, is defined to TRUE (1) by udf.h on Windows only and not on Linux. This is needed because on Windows, the UDF must read the file independently of the file handle already established by the Fluent executable. The reason for this is that, as long as you compile your UDF with the recommended compiler, it cannot use the file handle data structure provided by the Fluent executable.

2.5.6.5. Hooking a DPM Initialization UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_DPM_INJECTION_INIT is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you

supplied as the first `DEFINE` macro argument will become visible in the **Set Injection Properties** dialog box in Ansys Fluent.

See Hooking `DEFINE_DPM_INJECTION_INIT` UDFs (p. 541) for details on how to hook your `DEFINE_DPM_INJECTION_INIT` UDF to Ansys Fluent.

2.5.7. `DEFINE_DPM_LAW`

2.5.7.1. Description

You can use `DEFINE_DPM_LAW` to customize laws for particles. For example your UDF can specify custom laws for heat and mass transfer rates for droplets and combusting particles. Additionally, you can specify custom laws for mass, diameter, and temperature properties as the droplet or particle exchanges mass and energy with its surroundings.

When a `DEFINE_DPM_LAW UDF` is activated, then the number of species that can be referenced and interact with the particles via the UDF is limited to those with a species index less than the maximum UDF species number, defined using the TUI command `define/models/dpm/options/maximum-udf-species`. The default number for `maximum-udf-species` is 50.

2.5.7.2. Usage

`DEFINE_DPM_LAW` (name, tp, ci)

Argument Type	Description
symbol name	UDF name.
Tracked_Particle *tp	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
int ci	Variable that indicates whether the continuous and discrete phases are coupled (equal to 1 if coupled with continuous phase, 0 if not coupled).

Function returns

void

There are three arguments to `DEFINE_DPM_LAW`: `name`, `tp`, and `ci`. You supply `name`, the name of the UDF. `tp` and `ci` are variables that are passed by the Ansys Fluent solver to your UDF.

> **Important:**
>
> Pointer `tp` can be used as an argument to the macros defined in DPM Macros (p. 335) to obtain information about particle properties (for example, injection properties).

2.5.7.3. Example

The following UDF, named `Evapor_Swelling_Law`, models a custom law for the evaporation swelling of particles. The source code can be interpreted or compiled in Ansys Fluent. See Example (p. 247) `DEFINE_DPM_LAW` usage.

```
/************************************************************************
   UDF that models a custom law for evaporation swelling of particles
 ************************************************************************/

#include "udf.h"

DEFINE_DPM_LAW(Evapor_Swelling_Law,tp,ci)
{
   real swelling_coeff = 1.1;

   /* first, call standard evaporation routine to calculate
   the mass and heat transfer       */
   VaporizationLaw(tp);

   /* compute new particle diameter and density */
   TP_DIAM(tp) = TP_INIT_DIAM(tp)*(1. + (swelling_coeff - 1.)*
   (TP_INIT_MASS(tp)-TP_MASS(tp))/(DPM_VOLATILE_FRACTION(tp)*TP_INIT_MASS(tp)));
   TP_RHO(tp) = TP_MASS(tp) / (3.14159*TP_DIAM(tp)*TP_DIAM(tp)*TP_DIAM(tp)/6);
   TP_RHO(tp) = MAX(0.1, MIN(1e5, TP_RHO(tp)));
}
```

2.5.7.4. Hooking a Custom DPM Law to Ansys Fluent

After the UDF that you have defined using `DEFINE_DPM_LAW` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible in the **Custom Laws** dialog box in Ansys Fluent. See Hooking `DEFINE_DPM_LAW` UDFs (p. 542) for details on how to hook your `DEFINE_DPM_LAW` UDF to Ansys Fluent.

2.5.8. `DEFINE_DPM_OUTPUT`

2.5.8.1. Description

You can use `DEFINE_DPM_OUTPUT` to modify what is written to the sampling device output. This function allows access to the variables that are written to a file as a particle passes through a sampler (see Modeling Discrete Phase in the User's Guide for details).

2.5.8.2. Usage

`DEFINE_DPM_OUTPUT` (name, header, fp, tp, t, plane)

Argument Type	Description
`symbol name`	UDF name.
`int header`	Variable that is equal to `1` at the first call of the function before particles are tracked and set to `0` for subsequent calls.
`FILE *fp`	Pointer to the file to which you are writing.

Argument Type	Description
`Tracked_Particle *tp`	Pointer to the `Tracked_Particle` data structure which contains data related to the particle being tracked.
`Thread *t`	Pointer to the thread that the particle is passing through if the sampler is represented by a mesh zone. If the sampler is not defined as a mesh zone, then the value of `t` is `NULL`.
`Plane *plane`	Pointer to the `Plane` structure (see `dpm_types.h`) if the sampling device is defined as a plane surface (line in 2D). If a mesh zone is used by the sampler, then `plane` is `NULL`.

Function returns

`void`

There are six arguments to `DEFINE_DPM_OUTPUT`: `name`, `header`, `fp`, `tp`, `t`, and `plane`. You supply `name`, the name of the UDF. `header`, `fp`, `tp`, `t`, and `plane` are variables that are passed by the Ansys Fluent solver to your UDF. The output of your UDF will be written to the file indicated by `fp`.

Important:

Pointer `tp` can be used as an argument to the macros defined in DPM Macros (p. 335) to obtain information about particle properties (for example, injection properties).

When using `DEFINE_DPM_OUTPUT` to write sample files, certain special file operations must be performed by Ansys Fluent. Therefore, the usual C output function `fprintf` cannot be used. Instead, you must use the functions `par_fprintf` and `par_fprintf_head`. For details on the use of these functions, refer to The `par_fprintf_head` and `par_fprintf` Functions (p. 386) and the following example. In particular, note that the first two arguments passed to `par_fprintf` must always be given as indicated, because they are used to sort the contents of the file but they will not appear in the final output.

2.5.8.3. Example 1 - Sampling and Removing Particles

The following UDF named `discrete_phase_sample` samples the size and velocity of discrete phase particles at selected planes downstream of an injection. For 2D axisymmetric simulations, it is assumed that droplets/particles are being sampled at planes (lines) corresponding to constant x. For 3D simulations, the sampling planes correspond to constant z.

To remove particles from the domain after they have been sampled, change the value of RE-MOVE_PARTICLES to `TRUE`. In this case, particles will be deleted following the time step in which

they cross the plane. This is useful when you want to sample a spray immediately in front of an injector and you do not want to track the particles further downstream.

Important:

This UDF works with unsteady and steady simulations that include droplet break-up or collisions. Note that the discrete phase must be traced in an unsteady manner.

```
#include "udf.h"

/********************************************************************/
/* UDF that samples discrete phase size and velocity distributions */
/* within the domain.                                               */
/********************************************************************/

#define REMOVE_PARTICLES FALSE

DEFINE_DPM_OUTPUT(discrete_phase_sample,header,fp,tp,t,plane)
{
#if RP_2D
  real y;

  if(header)
  {
    par_fprintf_head(fp," #Time[s]  R [m]  X-velocity[m/s]");
    par_fprintf_head(fp," W-velocity[m/s] R-velocity[m/s] ");
    par_fprintf_head(fp,"Drop Diameter[m] Number of Drops  ");
    par_fprintf_head(fp,"Temperature [K] Initial Diam [m] ");
    par_fprintf_head(fp,"Injection Time [s] \n");
  }

  if(NULLP(tp))
    return;

  if (rp_axi && (sg_swirl || GVAR_TURB(rp, ke)))
    y = MAX(sqrt(SQR(TP_POS(tp)[1]) + SQR(TP_POS(tp)[2])),DPM_SMALL);
  else
    y = TP_POS(tp)[1];

  /* Note: The first two arguments to par_fprintf are used internally and */
  /* must not be changed, even though they do not appear in the final output.*/
  par_fprintf(fp, "%d %" int64_fmt " %e %f %f %f %f %e %e %f %e %f \n",
              P_INJ_ID(TP_INJECTION(tp)),
              TP_ID(tp),
              TP_TIME(tp),
              y,
              TP_VEL(tp)[0],
              TP_VEL(tp)[1],
              TP_VEL(tp)[2],
              TP_DIAM(tp),
              TP_N(tp),
              TP_T(tp),
              TP_INIT_DIAM(tp),
              TP_TIME_OF_BIRTH(tp));
#else
  real r, x, y;

  if(header)
  {
    par_fprintf_head(fp," #Time[s] R [m] x-velocity[m/s] ");
    par_fprintf_head(fp,"y-velocity[m/s] z-velocity[m/s]  ");
    par_fprintf_head(fp,"Drop Diameter[m]  Number of Drops ");
    par_fprintf_head(fp,"Temperature [K]  Initial Diam [m] ");
    par_fprintf_head(fp,"Injection Time [s] \n");
  }

  if(NULLP(tp))
```

```
    return;

  x = TP_POS(tp)[0];
  y = TP_POS(tp)[1];
  r = sqrt(SQR(x) + SQR(y));

  /* Note: The first two arguments to par_fprintf are used internally and */
  /* must not be changed, even though they do not appear in the final output.*/
  par_fprintf(fp,"%d %" int64_fmt " %e %f %f %f %f %e %e %f %e %f \n",
                P_INJ_ID(TP_INJECTION(tp)),
                TP_ID(tp),
                TP_TIME(tp),
                r,
                TP_VEL(tp)[0],
                TP_VEL(tp)[1],
                TP_VEL(tp)[2],
                TP_DIAM(tp),
                TP_N(tp),
                TP_T(tp),
                TP_INIT_DIAM(tp),
                TP_TIME_OF_BIRTH(tp));
#endif

#if REMOVE_PARTICLES
  MARK_TP(tp, P_FL_REMOVED);
#endif
}
```

2.5.8.4. Example 2 - Source Code Template

The following example provides the source code used in Ansys Fluent simulations when you do not use a DEFINE_DPM_OUTPUT UDF. You can modify this template to adapt it to your needs.

```
#include "udf.h"

/****************************************************************/
/* DPM sampling output UDF that does what Fluent does by default */
/****************************************************************/

#define REMOVE_PARTICLES FALSE

DEFINE_DPM_OUTPUT(my_dpm_out, header, fp, tp, thread, plane)
{
  if (header)
  {
    char *sort_name;
    char sort_fn[4096];

    if (NNULLP(thread))
      sort_name = THREAD_HEAD(thread)->dpm_summary.sort_file_name;
    else if ( ! NULLP(plane))
      sort_name = plane->sort_file_name;
    else   /* This is not expected to happen for regular particle sampling.. */
    {
      if (dpm_par.unsteady_tracking)
        sort_name = "parcels";
      else
        sort_name = "tracks";
    }

    /* sort_name may contain "/" (Linux)
     * or ":" and "\" (Windows) --
     * replace them all by "_":
     */
    strcpy(sort_fn, sort_name);
    replace_path_chars_in_string(sort_fn);
    if (dpm_par.unsteady_tracking)
      par_fprintf_head(fp, "(%s %d)\n", sort_fn, 13);
```

```
      else
        par_fprintf_head(fp, "(%s %d)\n", sort_fn, 12);

#if RP_2D
      if (rp_axi_swirl)
        par_fprintf_head(fp, "(x    r    theta   u    v    w");
      else
#endif
        par_fprintf_head(fp, "(x    y    z    u    v    w");

      if (dpm_par.unsteady_tracking)
        par_fprintf_head(fp, " diameter   t    parcel-mass  "
                              " mass   n-in-parcel   time   flow-time   name)\n");
      else
        par_fprintf_head(fp, " diameter   t    mass-flow  "
                              " mass    frequency    time    name)\n");
  }
  else if ( ! NULLP(tp))
  {
    /* Do some preparatory calculations for later use:
     */
    real flow_rate = 0.;
    real V_vel = TP_VEL(tp)[1];
    real W_vel = TP_VEL(tp)[2];
    real Y = TP_POS(tp)[1];
    real Z = TP_POS(tp)[2];
    real strength = 0.;
    real mass = 0.;

    if (TP_INJECTION(tp)->type != DPM_TYPE_MASSLESS)
    {
      mass = TP_MASS(tp);

      if (dpm_par.unsteady_tracking)
        strength = TP_N(tp);
      else
      {
        strength = TP_FLOW_RATE(tp) / TP_INIT_MASS(tp);
        if (TP_STOCHASTIC(tp))
          strength /= (real)TP_STOCHASTIC_NTRIES(tp);
      }

      flow_rate = strength * mass;
    }

#if RP_2D
    if (rp_axi_swirl)
    {
      Y = MAX(sqrt(TP_POS(tp)[1] * TP_POS(tp)[1] + TP_POS(tp)[2] * TP_POS(tp)[2]), DPM_SMALL);
      V_vel = (TP_VEL(tp)[1] * TP_POS(tp)[1] + TP_VEL(tp)[2] * TP_POS(tp)[2]) / Y;
      W_vel = (TP_VEL(tp)[2] * TP_POS(tp)[1] - TP_VEL(tp)[1] * TP_POS(tp)[2]) / Y;
      if (Y > 1.e-20) Z = LIMIT_ACOS(TP_POS(tp)[1] / Y);
    }
#endif

    if ( ! dpm_par.unsteady_tracking)
      par_fprintf(fp, /* Note: The first two arguments to par_fprintf are */
                      /* used internally and must not be changed, even */
                      /* though they do not appear in the final output.*/
                  "%d %" int64_fmt " ((%e  %e  %e  %e  %e  %e "
                  " %e  %e  %e  %e  %e  %e) %s:%" int64_fmt ")\n",
                  P_INJ_ID(TP_INJECTION(tp)), tp->part_id,
                  TP_POS(tp)[0],
                  Y,
                  Z,
                  TP_VEL(tp)[0],
                  V_vel,
                  W_vel,
                  TP_DIAM(tp),
                  TP_T(tp),
                  flow_rate,
```

```
                   mass,
                   strength,
                   TP_TIME(tp) - tp->time_of_birth,
                   TP_INJECTION(tp)->name,
                   tp->part_id);
    else
      par_fprintf(fp, /* Note: The first two arguments to par_fprintf are */
                      /* used internally and must not be changed, even */
                      /* though they do not appear in the final output.*/
                  "%d %" int64_fmt " ((%e  %e  %e  %e  %e  %e "
                  " %e  %e  %e  %e  %e  %e  %e) %s:%" int64_fmt ")\n",
                  (int) TP_TIME(tp),
                  (int64_t) ((TP_TIME(tp) -
                                (real) ((int) TP_TIME(tp))) *
                                (real) 1000000000000.),
                  TP_POS(tp)[0],
                  Y,
                  Z,
                  TP_VEL(tp)[0],
                  V_vel,
                  W_vel,
                  TP_DIAM(tp),
                  TP_T(tp),
                  flow_rate,
                  mass,
                  strength,
                  TP_TIME(tp) - tp->time_of_birth,
                  TP_TIME(tp),
                  TP_INJECTION(tp)->name,
                  tp->part_id);

#if REMOVE_PARTICLES
    MARK_TP(tp, P_FL_REMOVED);
#endif
  }
}
```

2.5.8.5. Using `DEFINE_DPM_OUTPUT` in VOF-to-DPM Simulations

For a VOF-to-DPM model transition simulation, if you use the option to write information about every liquid lump converted into DPM particle parcels to a file (as described in Setting up the VOF-to-DPM Model Transition in the *Fluent User's Guide*), you can use a `DEFINE_DPM_OUTPUT` function to define the file content. After you hook the `DEFINE_DPM_OUTPUT` function as described in Hooking a DPM Output UDF to Ansys Fluent (p. 236), that function will be also called multiple times every time the VOF-to-DPM model transition mechanism is triggered.

> **Note:**
>
> Once you hook `DEFINE_DPM_OUTPUT` into Ansys Fluent, it will be used for both particle sampling and VOF-to-DPM lump conversion transcript.

1. For the first call, the four last arguments `fp`, `tp`, `t`, and `plane` will be `NULL`. The UDF can be used to perform preparatory steps, such as looping over cells, findings lumps by the `C_LUMP_ID(c, t)` value, and collecting per-lump information that is not collected by default.

2. Starting with the second call, `fp` will contain a valid value to give an access to the file. In the second call, the header argument will be set to 1 to indicate that the UDF is supposed to write the file header.

3. The third call is similar to the first call, but with the correctly assigned `fp` argument. You can use it to finalize the collection of per-lump information. Note that only the first and third calls will be done synchronously on all compute-node processes so that global reductions can be used to correctly characterize liquid lumps that are spread across multiple partitions. To exclude exterior cells from the looping, use the `begin_c_loop_int()` statement.

4. The fourth call will provide a pointer to a fully allocated and initialized `Tracked_Particle` data structure in the `tp` argument. This call happens once for every lump elected for conversion by the criteria specified and in parallel processing, it is done on node-0 only. You can use this call for multiple purposes:

   • Write information to the file.

     Note that, if the `DEFINE_DPM_OUTPUT` UDF has been hooked, no information will be written to the lump conversion transcript file, other than by the UDF.

   • Modify the initial conditions of the particle parcel.

     Make sure to keep particle properties such as `TP_MASS(tp)`, `TP_MASS0(tp)`, `TP_INIT_MASS(tp)`, etc. consistent for all particle or parcel properties that you choose to modify.

   • Suppress the lump conversion by setting `TP_MASS(tp)` to a negative value.

   • Have the lump converted and eliminated from the VOF solution and then immediately discard the resulting DPM particle parcels.

     For that purpose, use the following statement:

     ```
     MARK_TP(tp, P_FL_REMOVED);
     ```

   At the time of these UDF calls, the solver has not determined yet whether the lump will be converted into single or multiple particle parcels. Any changes that the UDF applies to the `Tracked_Particle` data structure in the fourth call will apply automatically to the set of particle parcels into which the liquid lump will be split.

5. For more control over the DPM particle parcels, you can use a fifth call to the `DEFINE_DPM_OUTPUT` UDF for every particle parcel in the aforementioned set of parcels. The fourth and fifth calls can be distinguished by the expression `REAL_EQUAL(1.0, TP_N(tp))`, which will be `TRUE` in the fourth call and `FALSE` in the fifth.

You can also use the `DEFINE_DPM_OUTPUT` function to control whether a liquid droplet is represented by a single particle parcel or split into multiple particle parcels as follows:

1. In the **VOF-to-DPM Transition Parameters** dialog box, set **Split any DPM Parcel that Exceeds the Cell Volume by Factor** to a very large number.

2. For each droplet that you want to split into multiple particle parcels, each representing a fraction of the droplet that is not larger than the volume of the hosting cell, use the following statements in your UDF:

   • In the fourth call:

     ```
     MARK_TP(tp, P_FL_REMOVED);
     ```

- In the fifth call:

```
UNMARK_TP(tp, P_FL_REMOVED);
```

2.5.8.5.1. Example

The following sample UDF is similar to the one shown in Example 2 - Source Code Template (p. 230),
but it contains a few extra clauses demonstrating some of the options outlined above. If you
need more advice on using the DPM Output UDF for VOF-to-DPM model transition simulations,
contact your technical support engineer for assistance.

```c
#include "udf.h"
#include "lump_detect.h"

/****************************************************************/
/* DPM sampling output UDF that does what Fluent does by default */
/****************************************************************/

DEFINE_DPM_OUTPUT(my_dpm_out, header, fp, tp, thread, plane)
{
  if (header)
  {
    char *sort_name;
    char sort_fn[4096];

    if (NNULLP(thread))
      sort_name = THREAD_HEAD(thread)->dpm_summary.sort_file_name;
    else if ( ! NULLP(plane))
      sort_name = plane->sort_file_name;
    else    /* This is not expected to happen for regular particle sampling.. */
    {
      if ( ! NULLP(convert_lump_args.myldps.injection))
       sort_name = convert_lump_args.myldps.injection->name;
      else
      if (dpm_par.unsteady_tracking)
        sort_name = "parcels";
      else
        sort_name = "tracks";
    }

    /* sort_name may contain "/" (Linux)
     * or ":" and "\" (Windows) --
     * replace them all by "_":
     */
    strcpy(sort_fn, sort_name);
    replace_path_chars_in_string(sort_fn);
    if (dpm_par.unsteady_tracking)
      par_fprintf_head(fp, "(%s %d)\n", sort_fn, 13);
    else
      par_fprintf_head(fp, "(%s %d)\n", sort_fn, 12);

#if RP_2D
    if (rp_axi_swirl)
      par_fprintf_head(fp, "(x     r     theta    u    v     w");
    else
#endif
      par_fprintf_head(fp, "(x     y    z     u    v     w");

    if (dpm_par.unsteady_tracking)
      par_fprintf_head(fp, " diameter    t     parcel-mass  "
                           " mass    n-in-parcel   time    flow-time  name)\n");
    else
      par_fprintf_head(fp, " diameter    t    mass-flow  "
                           " mass    frequency    time    name)\n");
  }
  else if ( ! NULLP(tp))
  {
    /* Do some preparatory calculations for later use:
```

```
      */
     real flow_rate = 0.;
     real V_vel = TP_VEL(tp)[1];
     real W_vel = TP_VEL(tp)[2];
     real Y = TP_POS(tp)[1];
     real Z = TP_POS(tp)[2];
     real strength = 0.;
     real mass = 0.;

     if (TP_INJECTION(tp)->type != DPM_TYPE_MASSLESS)
     {
       mass = TP_MASS(tp);

       if (dpm_par.unsteady_tracking)
         strength = TP_N(tp);
       else
       {
         strength = TP_FLOW_RATE(tp) / TP_INIT_MASS(tp);
         if (TP_STOCHASTIC(tp))
           strength /= (real)TP_STOCHASTIC_NTRIES(tp);
       }

       flow_rate = strength * mass;
     }

#if RP_2D
     if (rp_axi_swirl)
     {
       Y = MAX(sqrt(TP_POS(tp)[1] * TP_POS(tp)[1] + TP_POS(tp)[2] * TP_POS(tp)[2]), DPM_SMALL);
       V_vel = (TP_VEL(tp)[1] * TP_POS(tp)[1] + TP_VEL(tp)[2] * TP_POS(tp)[2]) / Y;
       W_vel = (TP_VEL(tp)[2] * TP_POS(tp)[1] - TP_VEL(tp)[1] * TP_POS(tp)[2]) / Y;
       if (Y > 1.e-20) Z = LIMIT_ACOS(TP_POS(tp)[1] / Y);
     }
#endif

     if ( ! NULLP(convert_lump_args.domain))
     {
       /* This is definitely VOF-to-DPM calling,
        * so demonstrate some things we can do:
        */
       if (TP_POS(tp)[0] > 0.07)   /* Silly condition for demonstration purposes. */
       {
         TP_N(tp) = -1.;           /* Tell the code not even to convert the lump. */
         return;                   /* Do not go on to write anything to the file. */
       }
       else
       if (TP_POS(tp)[0] > 0.04)   /* Silly condition for demonstration purposes. */
       {
         MARK_TP(tp, P_FL_REMOVED);    /* Drop the particle, never track it. */
         return;                   /* Do not go on to write anything to the file. */
       }
       else
       {
         /* Just for demonstration that it works,
          * mirror the particle at the XY plane,
          * or set its temperature to 777:
          */
         /* TP_POS(tp)[2] *= -1.; */
         TP_T(tp) = 777.;
       }
     }

     if (NULLP(convert_lump_args.domain) ||  /* For future use: Not VOF-to-DPM, but...: */
         ( ! NULLP(thread)) ||                /* ...thread AND plane may still be NULL... */
         ( ! NULLP(plane))  ||   /* When VOF-to-DPM calls this, "thread" and "plane" are NULL */
#if RP_NODE
         I_AM_NODE_ZERO_P    ||   /* and only ONE compute-node process must write to the file: */
#endif
         I_AM_NODE_HOST_P    ||   /* (This line for shared-memory parallel DPM tracking only) */
         I_AM_NODE_SERIAL_P)      /* (This line for SERIAL ["-t0"] execution [unsupported]) */
       if ((1. == strength) ||   /* VOF-to-DPM: want to record one single entry per lump! */
```

```
                     (NULLP(convert_lump_args.domain)))   /* Otherwise, write every particle.. */
        {
         if ( ! dpm_par.unsteady_tracking)
           par_fprintf(fp, /* The first two arguments to par_fprintf are used internally and */
                           /* must not be changed, although they do not appear in the output.*/
                       "%d %" int64_fmt " ((%.6e  %.6e  %.6e "
                       " %.6e  %.6e  %.6e  %.6e  %.6e  %.6e "
                       " %.6e  %.6e  %.6e) %s:%" int64_fmt ")\n",
                       P_INJ_ID(TP_INJECTION(tp)), tp->part_id,
                       TP_POS(tp)[0],
                       Y,
                       Z,
                       TP_VEL(tp)[0],
                       V_vel,
                       W_vel,
                       TP_DIAM(tp),
                       TP_T(tp),
                       flow_rate,
                       mass,
                       strength,
                       TP_TIME(tp) - tp->time_of_birth,
                       TP_INJECTION(tp)->name,
                       tp->part_id);
         else
           par_fprintf(fp, /* The first two arguments to par_fprintf are used internally and */
                           /* must not be changed, although they do not appear in the output.*/
                       "%d %" int64_fmt " ((%.6e  %.6e  %.6e "
                       " %.6e  %.6e  %.6e  %.6e  %.6e  %.6e "
                       " %.6e  %.6e  %.6e  %.6e) %s:%" int64_fmt ")\n",
                       (int) TP_TIME(tp),
                       (int64_t) ((TP_TIME(tp) -
                                   (real) ((int) TP_TIME(tp))) *
                                  (real) 1000000000000.),
                       TP_POS(tp)[0],
                       Y,
                       Z,
                       TP_VEL(tp)[0],
                       V_vel,
                       W_vel,
                       TP_DIAM(tp),
                       TP_T(tp),
                       flow_rate,
                       mass,
                       strength,
                       TP_TIME(tp) - tp->time_of_birth,
                       TP_TIME(tp),
                       TP_INJECTION(tp)->name,
                       tp->part_id);
       }
    }
    else
    {
      /* tp is NULL: VOF-to-DPM calls us TWICE synchronously on all compute-node proc.
       * so that we have an opportunity e.g. to do our own lump characterisation (i.e.
       * calculate some lump properties that have not been calculated by Fluent yet),
       * or can interfere with the lump properties Fluent has determined so far:
       */
      Message0("\nmy_dpm_out: last_lump_id:  %d  =="
   "==  file yet opened? -- %s  ====\n",
            convert_lump_args.last_lump_id, NULLP(fp) ? "NO" : "yes!");
    }
}
```

2.5.8.6. Hooking a DPM Output UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_DPM_OUTPUT is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied

as the first `DEFINE` macro argument will become visible in the **Sample Trajectories** dialog box in Ansys Fluent. See Hooking `DEFINE_DPM_OUTPUT` UDFs (p. 543) for details on how to hook your `DEFINE_DPM_OUTPUT` UDF to Ansys Fluent.

2.5.9. `DEFINE_DPM_PROPERTY`

2.5.9.1. Description

You can use `DEFINE_DPM_PROPERTY` to specify properties of discrete phase materials. You can model the following dispersed phase properties with this type of UDF:

- particle emissivity
- vapor pressure
- vaporization temperature
- thermophoretic coefficient
- particle scattering factor
- boiling point
- particle viscosity
- particle density
- particle surface tension
- binary diffusivity
- swelling coefficient
- latent heat
- specific heat

> **Important:**
>
> - When you are using the `DEFINE_DPM_PROPERTY` macro to specify the density property for a combusting particle material, all other model-specific density calculations, such as the swelling calculation during particle devolatilization, or the composition dependent char density will be ignored and the density calculated by the UDF will always be used. Similarly when you are using the `DEFINE_DPM_PROPERTY` macro to specify the specific heat property for a combusting particle material, the composition dependent char specific heat option will be ignored.
>
> - When you are using either the non-premixed or the partially-premixed combustion model in the continuous phase calculation together with `DEFINE_DPM_PROPERTY` for particle specific heat, the `DEFINE_DPM_PROPERTY` UDF will be used for the specific heat and enthalpy calculations of the non-volatile/non-reacting particle mass.

2.5.9.2. Usage

DEFINE_DPM_PROPERTY (name, c, t, tp, T)

Argument Type	Description
symbol name	UDF name.
cell_t c	Index that identifies the cell where the particle is located in the given thread.
Thread *t	Pointer to the thread where the particle is located.
Tracked_Particle *tp	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
real T	Temperature. The appropriate temperature will be passed to your UDF by the solver. Depending on the DPM model you are using and the physical property type, it may be the temperature of the particle, the film, or the continuous phase at the current, previous, or initial location of the particle being tracked.

Function returns

real

There are five arguments to DEFINE_DPM_PROPERTY: name, c, t, tp, and T. You supply name, the name of the UDF. c, t, tp, and T are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the real value of the discrete phase property and return it to the solver.

If you are using DEFINE_DPM_PROPERTY to specify the specific heat for particle materials, your UDF will also need to set the value of the particle enthalpy in the Tracked_Particle *tp, tp->enthalpy, to the particle sensible enthalpy, which should be calculated as the temperature integral of the specific heat function from the reference temperature, T_REF, to the temperature, T.

Important:

- Pointer tp can be used as an argument to the macros defined in DPM Macros (p. 335) to obtain information about particle properties (for example, injection properties).

- In some situations, when Ansys Fluent calls DEFINE_DPM_PROPERTY, tp may point to a dummy Tracked_Particle structure. If that is the case, your UDF code must not use any data from that structure, except TP_INJECTION(tp), which is always available. You can add the following condition into your UDF code to determine whether Tracked_Particle that tp points to is a dummy structure:

```
if (NULLP(tp->pp) ||
    NULLP(TP_CELL_THREAD(tp)))
  {
    ... /* Do something withOUT using tp,
```

```
                         * except for accessing the Injection
                         * data structure through TP_INJECTION(tp).
                         */
                  tp->enthalpy = ...;   /* needed for specific heat only */
                  return ...;
                }
              else
                {
                  ... /* Do the regular computation --
                      * can use tp in all possible ways.
                      */
                  tp->enthalpy = ...;   /* needed for specific heat only */
                  return ...;
                }
```

2.5.9.3. Example

In the following example, three discrete phase material property UDFs (named coal_emissivity
, coal_scattering, and coal_cp, respectively) are concatenated into a single C source file.
These UDFs must be executed as compiled UDFs in Ansys Fluent.

```c
/***********************************************************************
 *     UDF that specifies discrete phase material properties
 ***********************************************************************
 */

#include "udf.h"

DEFINE_DPM_PROPERTY(coal_emissivity, c, t, tp, T)
{
  real mp0;
  real mp;
  real vf, cf;

  if (NULLP(tp->pp) ||
      NULLP(TP_CELL_THREAD(tp)))
    return 1.0;                         /* initial value */

  mp0 = TP_INIT_MASS(tp);
  mp  = TP_MASS(tp);

  /* get the material char and volatile fractions and
   * store them in vf and cf:
   */
  vf = DPM_VOLATILE_FRACTION(tp);
  cf = DPM_CHAR_FRACTION(tp);

  if (!(((mp / mp0) >= 1) ||
        ((mp / mp0) <= 0)))
    {
      if ((mp / mp0) < (1 - vf - cf))
        {
          /* only ash left  */
          /* vf = cf = 0    */

          return .001;
        }
      else if ((mp / mp0) < (1 - vf))
        {
          /* only ash and char left             */
          /* cf = 1 - (1 - vf - cf) / (mp / mp0)  */
          /* vf = 0                               */

          return 1.0;
```

```
        }
      else
        {
          /* volatiles, char, and ash left    */
          /* cf =             cf   / (mp / mp0)   */
          /* vf = 1 - (1 - vf) / (mp / mp0)   */

          return 1.0;
        }
    }

  return 1.0;
}


DEFINE_DPM_PROPERTY(coal_scattering, c, t, tp, T)
{
  real mp0;
  real mp;
  real cf, vf;

  if (NULLP(tp->pp) ||
      NULLP(TP_CELL_THREAD(tp)))
    return 1.0;                       /* initial value */


  mp0 = TP_INIT_MASS(tp);
  mp  = TP_MASS(tp);

  /* get the original char and volatile fractions
  /* and store them in vf and cf:
   */
  vf = DPM_VOLATILE_FRACTION(tp);
  cf = DPM_CHAR_FRACTION(tp);

  if (!(((mp / mp0) >= 1) ||
        ((mp / mp0) <= 0)))
    {
      if ((mp / mp0) < (1 - vf - cf))
        {
          /* only ash left   */
          /* vf = cf = 0      */

          return 1.1;
        }
      else if ((mp / mp0) < (1 - vf))
        {
          /* only ash and char left              */
          /* cf = 1 - (1 - vf - cf) / (mp / mp0)  */
          /* vf = 0                               */

          return 0.9;
        }
      else
        {
          /* volatiles, char, and ash left    */
          /* cf =             cf   / (mp / mp0)   */
          /* vf = 1 - (1 - vf) / (mp / mp0)   */

          return 1.0;
        }
    }

  return 1.0;
}


DEFINE_DPM_PROPERTY(coal_cp, c, t, tp, T)
{
  real mp0;
  real mp;
```

```
   real cf;
   real vf;
   real af;
   real Cp;

   if (NULLP(tp->pp) ||
      NULLP(TP_CELL_THREAD(tp)))
     {
       Cp = 1600.;                       /* initial value */
     }
   else
     {
       mp0 = TP_INIT_MASS(tp);
       mp  = TP_MASS(tp);

       cf  = TP_CF(tp);                  /* char fraction */
       vf  = TP_VF(tp);                  /* volatiles fraction */
       af  = 1. - TP_VF(tp) - TP_CF(tp); /* ash fraction */

       Cp  = 2000. * af +
             1100. * vf +
             1300. * cf;
     }

   tp->enthalpy = Cp * (T - T_REF);

   return Cp;
}
```

2.5.9.4. Hooking a DPM Material Property UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DPM_PROPERTY` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible in the **Create/Edit Materials** dialog box in Ansys Fluent. See Hooking `DEFINE_DPM_PROPERTY` UDFs (p. 544) for details on how to hook your `DEFINE_DPM_PROPERTY` UDF to Ansys Fluent.

2.5.10. `DEFINE_DPM_SCALAR_UPDATE`

2.5.10.1. Description

You can use `DEFINE_DPM_SCALAR_UPDATE` to update scalar quantities every time a particle position is updated. The function allows particle-related variables to be updated or integrated over the life of the particle. Particle values can be stored in an array associated with the `Tracked_Particle` (accessed with the macro `TP_USER_REAL(tp,i)`). Values calculated and stored in the array can be used to color the particle trajectory.

During Ansys Fluent execution, the `DEFINE_DPM_SCALAR_UPDATE` function is called at the start of particle integration and then after each time step for the particle trajectory integration. A value of 1 for `initialize` will be passed to the UDF when the particle is first injected or each time the particle tracker is called for unsteady particle tracking.

2.5.10.2. Usage

`DEFINE_DPM_SCALAR_UPDATE` (name, c, t, initialize, tp)

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index that identifies the cell that the particle is currently in.
`Thread *t`	Pointer to the thread the particle is currently in.
`int initialize`	Variable that has a value of 1 when the function is called at the start of the particle integration, and 0 thereafter.
`Tracked_Particle *tp`	Pointer to the `Tracked_Particle` data structure which contains data related to the particle being tracked.

Function returns

`void`

There are five arguments to `DEFINE_DPM_SCALAR_UPDATE`: `name`, `c`, `t`, `initialize`, and `tp`. You supply `name`, the name of the UDF. `c`, `t`, `initialize`, and `tp` are variables that are passed by the Ansys Fluent solver to your UDF.

Important:

Pointer `tp` can be used as an argument to the macros defined in DPM Macros (p. 335) to obtain information about particle properties (for example, injection properties). Also, the `real` array `user` is available for storage. The size of this array should be set in the **Discrete Phase Model** dialog box in the **Number of Scalars** field.

2.5.10.3. Example

The following compiled UDF computes the melting index along a particle trajectory. The `DEFINE_DPM_SCALAR_UPDATE` function is called at every particle time step in Ansys Fluent and requires a significant amount of CPU time to execute.

The melting index is computed from

$$\text{melting index} = \int_0^t \frac{1}{\mu} dt \qquad (2.20)$$

Also included in this UDF is an initialization function `DEFINE_INIT` that is used to initialize the scalar variables. `DPM_OUTPUT` is used to write the melting index at sample planes and surfaces. The macro `NULLP(p)`, which expands to `((p) == NULL)`, checks if its argument `p` is a null pointer.

```
/***********************************************************************
   UDF for computing the melting index along a particle trajectory
 ***********************************************************************/
#include "udf.h"

DEFINE_INIT(melt_setup,domain)
{
    /* if memory for the particle variable titles has not been
```

```
   * allocated yet, do it now */
    if (NULLP(user_particle_vars)) Init_User_Particle_Vars();
      /* now set the name and label */
      strcpy(user_particle_vars[0].name,"melting-index");
      strcpy(user_particle_vars[0].label,"Melting Index");
      strcpy(user_particle_vars[1].name,"melting-index-0");
      strcpy(user_particle_vars[1].label,"Melting Index 0");
}

/* update the user scalar variables */
DEFINE_DPM_SCALAR_UPDATE(melting_index,cell,thread,initialize,tp)
{
   cphase_state_t *c = &(tp->cphase[0]);
   if (initialize)
      {
      /* this is the initialization call, set:
      * TP_USER_REAL(tp,0) contains the melting index, initialize to 0
      * TP_USER_REAL(tp,1) contains the viscosity at the start of a time step*/
      TP_USER_REAL(tp,0) = 0.;
      TP_USER_REAL(tp,1) = c->mu;
      }
   else
      {
       /* use a trapezoidal rule to integrate the melting index */
       TP_USER_REAL(tp,0) += TP_DT(tp) * .5 * (1/TP_USER_REAL(tp,1) + 1/c->mu);
       /* save current fluid viscosity for start of next step */
       TP_USER_REAL(tp,1) = c->mu;
      }
   }

/* write melting index when sorting particles at surfaces */
DEFINE_DPM_OUTPUT(melting_output,header,fp,tp,thread,plane)
{
   char name[100];
   if (header)
   {
    if (NNULLP(thread))
      par_fprintf_head(fp,"(%s %d)\n",THREAD_HEAD(thread)->
        dpm_summary.sort_file_name,11);
    else
      par_fprintf_head(fp,"(%s %d)\n",plane->sort_file_name,11);
      par_fprintf_head(fp,"(%10s %10s %10s %10s %10s %10s %10s"
        " %10s %10s %10s %10s %s)\n",
        "X","Y","Z","U","V","W","diameter","T","mass-flow",
        "time","melt-index","name");
      }
   else
      {
      sprintf(name,"%s:%d",TP_INJECTION(tp)->name,TP_ID(tp));
      /* Note: The first two arguments to par_fprintf are used internally and     */
      /* must not be changed, even though they do not appear in the final output. */
      par_fprintf(fp,
          "%d %d ((%10.6g %10.6g %10.6g %10.6g %10.6g %10.6g "
          "%10.6g %10.6g %10.6g %10.6g %10.6g) %s)\n",
        P_INJ_ID(TP_INJECTION(tp)), TP_ID(tp),
        TP_POS(tp)[0], TP_POS(tp)[1], TP_POS(tp)[2],
        TP_VEL(tp)[0], TP_VEL(tp)[1], TP_VEL(tp)[2],
        TP_DIAM(tp), TP_T(tp), TP_FLOW_RATE(tp), TP_TIME(tp),
        TP_USER_REAL(tp,0), name);
      }
}
```

2.5.10.4. Hooking a DPM Scalar Update UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_DPM_SCALAR_UPDATE is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you

supplied as the first `DEFINE` macro argument will become visible in the **Discrete Phase Model** dialog box in Ansys Fluent.

See Hooking `DEFINE_DPM_SCALAR_UPDATE` UDFs (p. 546) for details on how to hook your `DEFINE_DPM_SCALAR_UPDATE` UDF to Ansys Fluent.

2.5.11. `DEFINE_DPM_SOURCE`

2.5.11.1. Description

You can use `DEFINE_DPM_SOURCE` to specify particle source terms. The function allows access to the accumulated source terms for a particle in a given cell before they are added to the mass, momentum, and energy exchange terms for coupled DPM calculations.

When a `DEFINE_DPM_SOURCE` UDF is activated, then the number of species that can be referenced and interact with the particles via the UDF is limited to those with a species index less than the maximum UDF species number, defined using the TUI command define/models/dpm/options/maximum-udf-species. The default number for `maximum-udf-species` is 50.

2.5.11.2. Usage

`DEFINE_DPM_SOURCE (name, c, t, S, strength, tp)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Index that identifies the cell that the particle is currently in.
`Thread *t`	Pointer to the thread the particle is currently in.
`dpms_t *S`	Pointer to the source structure `dpms_t`, which contains the source terms for the cell.
`real strength`	Particle number flow rate in particles/second (divided by the number of tries if stochastic tracking is used).
`Tracked_Particle *tp`	Pointer to the `Tracked_Particle` data structure which contains data related to the particle being tracked.

Function returns

`void`

There are six arguments to `DEFINE_DPM_SOURCE`: `name`, `c`, `t`, `S`, `strength`, and `tp`. You supply `name`, the name of the UDF. `c`, `t`, `S`, `strength`, and `tp` are variables that are passed by the Ansys

Fluent solver to your UDF. The modified source terms, after they have been computed by the function, will be stored in S.

> **Important:**
>
> Pointer tp can be used as an argument to the macros defined in DPM Macros (p. 335) to obtain information about particle properties (for example, injection properties).

2.5.11.3. Example

See Example (p. 247) for an example of DEFINE_DPM_SOURCE usage.

2.5.11.4. Hooking a DPM Source Term UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_DPM_SOURCE is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the **Discrete Phase Model** dialog box in Ansys Fluent. See Hooking DEFINE_DPM_SOURCE UDFs (p. 547) for details on how to hook your DEFINE_DPM_SOURCE UDF to Ansys Fluent.

2.5.12. DEFINE_DPM_SPRAY_COLLIDE

2.5.12.1. Description

You can use DEFINE_DPM_SPRAY_COLLIDE to side-step the default Ansys Fluent spray collision algorithm. When droplets collide they may bounce (in which case their velocity changes) or they may coalesce (in which case their velocity is changed, as well as their diameter and number in the DPM parcel). A spray collide UDF is called during droplet tracking after every droplet time step and requires that **Droplet Collision** is enabled in the **Discrete Phase Model** dialog box.

2.5.12.2. Usage

DEFINE_DPM_SPRAY_COLLIDE (name, tp, p)

Argument Type	Description
symbol name	UDF name.
Tracked_Particle *tp	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
Particle *p	Pointer to the Particle data structure where particles p are stored in a linked list.

Function returns

void

There are three arguments to DEFINE_DPM_SPRAY_COLLIDE: name, tp, and p. You supply name, the name of the UDF. tp and p are variables that are passed by the Ansys Fluent solver to

your UDF. When collision is enabled, this linked list is ordered by the cell that the particle is currently in. As particles from this linked list are tracked, they are copied from the particle list into a `Tracked_Particle` structure.

2.5.12.3. Example

The following UDF, named `mean_spray_collide`, is a simple (and non-physical) example that demonstrates the usage of `DEFINE_SPRAY_COLLIDE`. The droplet diameters are assumed to relax to their initial diameter over a specified time `t_relax`. The droplet velocity is also assumed to relax to the mean velocity of all droplets in the cell over the same time scale.

```
/**********************************************************
   DPM Spray Collide Example UDF
 **********************************************************/
#include "udf.h"
#include "dpm.h"
#include "surf.h"
DEFINE_DPM_SPRAY_COLLIDE(mean_spray_collide,tp,p)
{
    /* non-physical collision UDF that relaxes the particle */
    /* velocity and diameter in a cell to the mean over the */
    /* specified time scale t_relax */
    const real t_relax = 0.001; /* seconds */

    /* get the cell and Thread that the particle is currently in */
    cell_t c = TP_CELL(tp);
    Thread *t = TP_CELL_THREAD(tp);
    /* Particle index for looping over all particles in the cell */
    Particle *pi;

    /* loop over all particles in the cell to find their mass */
    /* weighted mean velocity and diameter */
    int i;
    real u_mean[3]={0.}, mass_mean=0.;
    real d_orig = TP_DIAM(tp);
    real decay = 1. - exp(-t_relax);
    begin_particle_cell_loop(pi,c,t)
      {
         mass_mean += PP_MASS(pi);
         for(i=0;i<3;i++)
           u_mean[i] += PP_VEL(pi)[i]*PP_MASS(pi);
      }
    end_particle_cell_loop(pi,c,t)  /* relax particle velocity to the mean and diameter to the */
    /* initial diameter over the relaxation time scale t_relax */
    if(mass_mean > 0.)
      {
       for(i=0;i<3;i++)
         u_mean[i] /= mass_mean;
       for(i=0;i<3;i++)
         TP_VEL(tp)[i] += decay*(u_mean[i] - TP_VEL(tp)[i]);
         TP_DIAM(tp) += decay*(TP_INIT_DIAM(tp) - TP_DIAM(tp));
         /* adjust the number in the droplet parcel to conserve mass */
         TP_N(tp) *= CUB(d_orig/TP_DIAM(tp));
      }
  }
```

2.5.12.4. Hooking a DPM Spray Collide UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DPM_SPRAY_COLLIDE` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible in the **Discrete Phase Model** dialog box in Ansys Fluent.

See for details on how to hook your
`DEFINE_DPM_SPRAY_COLLIDE` UDF to Ansys Fluent.

2.5.13. `DEFINE_DPM_SWITCH`

2.5.13.1. Description

You can use `DEFINE_DPM_SWITCH` to modify the criteria for switching between laws. The function can be used to control the switching between the user-defined particle laws and the default particle laws, or between different user-defined or default particle laws.

2.5.13.2. Usage

`DEFINE_DPM_SWITCH` (name, tp, ci)

Argument Type	Description
`symbol name`	UDF name.
`Tracked_Particle *tp`	Pointer to the `Tracked_Particle` data structure which contains data related to the particle being tracked.
`int ci`	Variable that indicates if the continuous and discrete phases are coupled (equal to 1 if coupled with continuous phase, 0 if not coupled).

Function returns

`void`

There are three arguments to `DEFINE_DPM_SWITCH`: `name`, `tp`, and `ci`. You supply `name`, the name of the UDF. `tp` and `ci` are variables that are passed by the Ansys Fluent solver to your UDF.

Important:

Pointer `tp` can be used as an argument to the macros defined in DPM Macros (p. 335) to obtain information about particle properties (for example, injection properties).

2.5.13.3. Example

The following is an example of a compiled UDF that uses `DEFINE_DPM_SWITCH` to switch between DPM laws using a criterion. The UDF switches to `DPM_LAW_USER_1` which refers to `condenshumid-law` since only one user law has been defined. The switching criterion is the local humidity which is computed in the domain using a `DEFINE_ON_DEMAND` function, which again calls the function `myHumidity` for every cell. In the case where the humidity is greater than 1, condensation is computed by applying a simple mass transfer calculation. Otherwise, Ansys Fluent's standard law for Inert Heating is applied. The UDF requires one UDML and needs a species called `h2o` to compute the local humidity.

```
/**********************************************************************
   Concatenated UDFs for the Discrete Phase Model including
   an implementation of a condensation model
   an example for the use of DPM_SWITCH
 **********************************************************************/

#include "udf.h"
#include "dpm.h"

#define UDM_RH 0          /* no. of UDM holding relative humidity */
#define N_REQ_UDM 1       /* 1 more than UDM_RH */
#define CONDENS 1.0e-4    /* a condensation rate constant */

int h2o_index=0;          /* index of water vapor species in mixture material */
real mw_h2o=18.;          /* molecular weight of water */

real H2O_Saturation_Pressure(real T)
{
  real ratio, aTmTp;
  T = MAX(T, 273);
  T = MIN(T, 647.286);
  aTmTp = .01 * (T - 338.15);
  ratio = (647.286 / T - 1.) *
          (-7.419242 + aTmTp * (.29721 +
                       aTmTp * (-.1155286 +
                       aTmTp * (8.685635e-3 +
                       aTmTp * (1.094098e-3 +
                       aTmTp * (-4.39993e-3 +
                       aTmTp * (2.520658e-3 -
                       aTmTp * 5.218684e-4)))))));
 return (22.089e6 * exp(MIN(ratio, 35.)));
}

real myHumidity(cell_t c, Thread *t)
{
  int i;
  Material *m = THREAD_MATERIAL(t), *sp;
  real yi_h2o = 0;       /* water mass fraction */
  real r_mix = 0.0;      /* sum of [mass fraction / mol. weight] over all species */
  real humidity;

  if ((MATERIAL_TYPE(m) == MATERIAL_MIXTURE) && (FLUID_THREAD_P(t)))
    {
      yi_h2o = C_YI(c, t, h2o_index);      /* water vapor mass fraction */
      mixture_species_loop(m, sp, i)
        {
          r_mix += C_YI(c,t,i) / MATERIAL_PROP(sp, PROP_mwi);
        }

      humidity = op_pres * yi_h2o / (mw_h2o * r_mix) /
                 H2O_Saturation_Pressure(C_T(c,t));
      return humidity;
    }
  else
    return 0.;
}

DEFINE_DPM_LAW(condenshumidlaw, tp, coupled)
{
  real area;
  real mp_dot;

  /* Get Cell and Thread from Particle Structure */
  cell_t c = TP_CELL(tp);
  Thread *t = TP_CELL_THREAD(tp);

  area = 4.0 * M_PI * (TP_DIAM(tp) * TP_DIAM(tp));

  /* Note This law only used if Humidity > 1.0 so mp_dot always positive*/
  mp_dot = CONDENS * sqrt(area) * (myHumidity(c, t) - 1.0);
  if (mp_dot > 0.0)
```

```
      {
        TP_MASS(tp) += mp_dot * TP_DT(tp);
        TP_DIAM(tp) = pow(6.0 * TP_MASS(tp) / (TP_RHO(tp) * M_PI), 1./3.);
      }
  /* Assume condensing particle is in thermal equilibrium with fluid in cell */
  TP_T(tp) = C_T(c,t);
}

DEFINE_DPM_SOURCE(dpm_source, c, t, S, strength, tp)
{
  real mp_dot;

  /* mp_dot is the mass source to the continuous phase
   * (Difference in mass between entry and exit from cell)
   * multiplied by strength (Number of particles/s in stream)
   */
  mp_dot = (TP_MASS0(tp) - TP_MASS(tp)) * strength;

  if (TP_CURRENT_LAW(tp) == DPM_LAW_USER_1)
    {
      /* Sources relevant to the user law 1:
       * add the source to the condensing species
       * equation and adjust the energy source by
       * adding the latent heat at reference temperature
       */
      S->species[h2o_index] += mp_dot;
      S->energy -= mp_dot * TP_INJECTION(tp)->latent_heat_ref;
    }
}

DEFINE_DPM_SWITCH(dpm_switch, tp, coupled)
{
  cell_t c = TP_CELL(tp);
  Thread *t = TP_CELL_THREAD(tp);
  Material *m = TP_MATERIAL(tp);

  /* If the relative humidity is higher than 1
   * and the particle temperature below the boiling temperature
   * switch to condensation law
   */
  if ((C_UDMI(c,t,UDM_RH) > 1.0) && (TP_T(tp) < DPM_BOILING_TEMPERATURE(tp, m)))
    TP_CURRENT_LAW(tp) = DPM_LAW_USER_1;
  else
    TP_CURRENT_LAW(tp) = DPM_LAW_INITIAL_INERT_HEATING;
}

DEFINE_ADJUST(adj_relhum, domain)
{
  cell_t cell;
  Thread *thread;

  if(sg_udm < N_REQ_UDM)
    Message("\nNot enough user defined memory allocated. %d required.\n",
    N_REQ_UDM);
  else
    {
      real humidity, min, max;
      min = 1e10;
      max = 0.0;
      thread_loop_c(thread, domain)
        {
          /* Check if thread is a Fluid thread and has UDMs set up on it */
          if (FLUID_THREAD_P(thread) && NNULLP(THREAD_STORAGE(thread, SV_UDM_I)))
            {
              Material *m = THREAD_MATERIAL(thread), *sp;
              int i;
              /* Set the species index and molecular weight of water */
              if (MATERIAL_TYPE(m) == MATERIAL_MIXTURE)
                mixture_species_loop (m,sp,i)
                  {
                    if (0 == strcmp(MIXTURE_SPECIE_NAME(m,i),"h2o") ||
```

```
                        (0 == strcmp(MIXTURE_SPECIE_NAME(m,i),"H2O")))
                    {
                      h2o_index = i;
                      mw_h2o = MATERIAL_PROP(sp,PROP_mwi);
                    }
                }

            begin_c_loop(cell,thread)
              {
                humidity = myHumidity(cell, thread);
                min = MIN(min, humidity);
                max = MAX(max, humidity);
                C_UDMI(cell, thread, UDM_RH) = humidity;
              }
            end_c_loop(cell, thread)
          }
        }
      Message("\nRelative Humidity set in udm-%d", UDM_RH);
      Message(" range:(%f,%f)\n", min, max);
    }/* end if for enough UDSs and UDMs */
}

DEFINE_ON_DEMAND(set_relhum)
{
  adj_relhum(Get_Domain(1));
}
```

2.5.13.4. Hooking a DPM Switching UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DPM_SWITCH` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible in the **Custom Laws** dialog box in Ansys Fluent. See Hooking `DEFINE_DPM_SWITCH` UDFs (p. 550) for details on how to hook your `DEFINE_DPM_SWITCH` UDF to Ansys Fluent.

2.5.14. `DEFINE_DPM_TIMESTEP`

2.5.14.1. Description

You can use `DEFINE_DPM_TIMESTEP` to change the time step for DPM particle tracking based on user-specified inputs. The time step can be prescribed for special applications where a certain time step is needed. It can also be limited to values that are required to validate physical models.

2.5.14.2. Usage

`DEFINE_DPM_TIMESTEP` (name, tp, ts)

Argument Type	Description
symbol name	UDF name.
Tracked_Particle *tp	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
real ts	Time step.

Function returns

```
real
```

There are three arguments to `DEFINE_DPM_TIMESTEP`: `name`, `tp`, and `ts`. You supply the `name` of your user-defined function. `tp` and `ts` are variables that are passed by the Ansys Fluent solver to your UDF. Your function will return the `real` value of the DPM particle timestep to the solver.

2.5.14.3. Example 1

The following compiled UDF named `limit_to_e_minus_four` sets the time step to a maximum value of 1e-4. If the time step computed by Ansys Fluent (and passed as an argument) is smaller than 1e-4, then Ansys Fluent's time step is returned.

```
/* Time step control UDF for DPM */

#include "udf.h"
#include "dpm.h"

DEFINE_DPM_TIMESTEP(limit_to_e_minus_four,tp,dt)
{
   if (dt > 1.e-4)
     {
       /* TP_NEXT_TIME_STEP(tp) = 1.e-4; */
       return 1.e-4;
     }
   return dt;
}
```

2.5.14.4. Example 2

The following compiled UDF named `limit_to_fifth_of_prt` computes the particle relaxation time based on the formula:

$$\tau_p = \frac{\rho_p d_p^2}{18\mu} \frac{24}{C_d \mathrm{Re}_p} \tag{2.21}$$

where

$$Re_p = \frac{\rho d_p \|u - u_p\|}{\mu} \tag{2.22}$$

The particle time step is limited to a fifth of the particle relaxation time. If the particle time step computed by Ansys Fluent (and passed as an argument) is smaller than this value, then Ansys Fluent's time step is returned.

> **Important:**
>
> The high-resolution particle tracking option requires forces on the particle and the particle time step to be continuous across cell boundaries. Discontinuities in the forces or the time step can result in particles being lost or stuck at interior cell faces.

When executing the below UDF with high-resolution particle tracking, it is recommended that the continuous phase viscosity be interpolated to the particle position if it varies with position. This can be enabled with the following text command:

```
define/models/dpm/numerics/high-resolution-tracking/barycentric-inter-
polation/interpolate-flow-viscosity? yes
```

```
/* Particle time step control UDF for DPM */

#include "udf.h"
#include "dpm.h"

DEFINE_DPM_TIMESTEP(limit_to_fifth_of_prt,tp,dt)
{
  real drag_factor = 0.;
  real p_relax_time;
  cphase_state_t *c = &(tp->cphase[0]);
  /* compute particle relaxation time */
  if (TP_DIAM(tp) != 0.0)
     drag_factor = DragCoeff(tp) * c->mu / (TP_RHO(tp) * TP_DIAM(tp) * TP_DIAM(tp));
  else
     drag_factor = 1.;
  p_relax_time = 1./drag_factor;
  /* check the condition and return the time step */
  if (dt > p_relax_time/5.)
   {
     return p_relax_time/5.;
   }
  return dt;
}
```

2.5.14.5. Hooking a DPM Timestep UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DPM_TIMESTEP` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible and selectable for **DPM Timestep** in the **Discrete Phase Model** dialog box in Ansys Fluent. See Hooking `DEFINE_DPM_TIMESTEP` UDFs (p. 551) for details on how to hook your `DEFINE_DPM_TIMESTEP` UDF to Ansys Fluent.

2.5.15. `DEFINE_DPM_VP_EQUILIB`

2.5.15.1. Description

You can use `DEFINE_DPM_VP_EQUILIB` to specify the equilibrium vapor pressure of vaporizing components of multicomponent particles.

2.5.15.2. Usage

`DEFINE_DPM_VP_EQUILIB` (name, tp, T, cvap_surf, Z)

Argument Type	Description
`symbol name`	UDF name.
`Tracked_Particle *tp`	Pointer to the `Tracked_Particle` data structure which contains data related to the particle being tracked.
`real T`	Temperature. The appropriate temperature will be passed to your UDF by the solver and may be equal to the particle, the film, or the multi-component droplet saturation temperature.

Argument Type	Description
`real *cvap_surf`	Array that contains the equilibrium vapor concentration over the particle surface
`real *Z`	Pointer to the compressibility factor, Z

Function returns

`void`

There are five arguments to `DEFINE_DPM_VP_EQUILIB`: `name`, `tp`, `T`, `cvap_surf`, and `Z`. You supply the `name` of your user-defined function. `tp` and `T` are passed by the Ansys Fluent solver to your UDF. Your UDF will need to compute the equilibrium vapor concentration and the compressibility factor and store their values in `cvap_surf` and `Z`, respectively.

2.5.15.3. Example

The following UDF named `raoult_vpe` computes the equilibrium vapor concentration of a multicomponent particle using the Raoult law. The vapor pressure in the law is proportional to the molar fraction of the condenses material. `DEFINE_VP_EQUILIB` is called several times every particle time step in Ansys Fluent and requires a significant amount of CPU time to execute. For this reason, the UDF should be executed as a compiled UDF.

```
/**************************************************************************
   UDF for defining the vapor particle equilibrium
   for multicomponent particles
 **************************************************************************/
#include <udf.h>
DEFINE_DPM_VP_EQUILIB(raoult_vpe,tp,Tp,cvap_surf,Z)
{
   int is;
   real molwt[MAX_SPE_EQNS];
   Thread *t0 = TP_CELL_THREAD(tp);  /* cell thread of particle location */
   Material *gas_mix = THREAD_MATERIAL(t0); /* gas mixture material */
   Material *cond_mix = TP_MATERIAL(tp); /* particle mixture material */
   int nc = TP_N_COMPONENTS(tp);     /* number of particle components */
   real molwt_cond = 0.;  /* reciprocal molecular weight of the particle */
   for (is = 0; is < nc; is++)
     {
        int gas_index = TP_COMPONENT_INDEX_I(tp,is); /* index of vaporizing
           component in the gas phase */
        if (gas_index >= 0)
          {
           /* the molecular weight of particle material */
           molwt[gas_index] =
           MATERIAL_PROP(MIXTURE_COMPONENT(gas_mix,gas_index),PROP_mwi);
            molwt_cond += TP_COMPONENT_I(tp,is) / molwt[gas_index];
           }
        }
   /* prevent division by zero */
   molwt_cond = MAX(molwt_cond,DPM_SMALL);
   for (is = 0; is < nc; is++)  {
    /* gas species index of vaporization */
    int gas_index = TP_COMPONENT_INDEX_I(tp,is);
    if(gas_index >= 0)
      {
       /* condensed material */
       Material * cond_c = MIXTURE_COMPONENT(cond_mix, is);
       /* condensed component molefraction */
       real xi_cond = TP_COMPONENT_I(tp,is)/(molwt[gas_index]*molwt_cond);
       /* particle saturation pressure */
       real p_saturation = DPM_vapor_pressure(tp, cond_c, Tp);
       if (p_saturation < 0.0)
```

```
        p_saturation = 0.0;
      /* vapor pressure over the surface, this is the actual Raoult law */
      cvap_surf[is] = xi_cond * p_saturation / UNIVERSAL_GAS_CONSTANT / Tp;
      }
    }
  }
  /* compressibility for ideal gas */
  *Z = 1.0;
}
```

2.5.15.4. Hooking a DPM Vapor Equilibrium UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DPM_VP_EQUILIB` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible and selectable in the **Create/Edit Materials** dialog box in Ansys Fluent. Note that before you hook the UDF, you'll need to create particle injections in the **Injections** dialog box with the type **Multicomponent** chosen. See Hooking `DEFINE_DPM_VP_EQUILIB` UDFs (p. 552) for details on how to hook your `DEFINE_DPM_VP_EQUILIB` UDF to Ansys Fluent.

2.5.16. `DEFINE_IMPINGEMENT`

2.5.16.1. Description

You can use `DEFINE_IMPINGEMENT` to customize the impingement regime selection criteria.

2.5.16.2. Usage

`DEFINE_IMPINGEMENT (name, tp, rel_dot_n, f, t, y_s, E_imp)`

Argument Type	Description
`symbol name`	UDF name.
`Tracked_Particle *tp`	Pointer to the `Tracked_Particle` data structure which contains data related to the particle being tracked.
`real real_dot_n`	The particle impingement velocity magnitude
`face_t f`	Index of the face that the particle is currently hitting
`Thread *t`	Pointer to the face thread the particle is currently hitting
`real *y_s`	Pointer to the fraction of particle mass that remains in the free stream after impact
`real *E_imp`	Pointer to the particle impingement parameter that is used as a regime selection criterion.

Function returns

`int`

There are seven arguments to `DEFINE_IMPINGEMENT`: `name`, `tp`, `real_dot_n`, `f`, `t`, `y_s`, and `E_imp`. You supply the `name` of your user-defined function. `tp`, `real_dot_n`, `f`, and `t` are passed by the Ansys Fluent solver to your UDF. Your function will:

- Compute the fraction of the particle mass that remains in the free stream after impact and store it in `y_s`

- Compute the impingement parameter and store it in `E_imp`

  The variable `E_imp` calculated in your UDF replaces the impingement energy calculated by Equation 12.230 and will be used also in Equation 12.238 for the peak diameter of the splashed droplets.

- Return the impingement regime

The impingement regime that you return can either be one of the four predefined impingement regimes in Ansys Fluent, described in Interaction During Impact with a Boundary in the *Fluent Theory Guide*:

```
FILM_STICK

FILM_REBOUND

FILM_SPREAD

FILM_SPLASH
```
or one of the following user-defined regimes defined in a `DEFINE_FILM_REGIME` (p. 257) UDF:

```
FILM_USER_0

FILM_USER_1

FILM_USER_2

FILM_USER_3

FILM_USER_4

FILM_USER_5

FILM_USER_6

FILM_USER_7

FILM_USER_8

FILM_USER_9
```

2.5.16.3. Example

The following UDF, named `dry_impingement`, returns one of the pre-defined regimes, or the user-defined regime `FILM_USER_0` for high wall temperature and high impact energy conditions. This function must be run as a compiled UDF.

```
#include "udf.h"
#define E_crit_0 16.
#define E_crit_1 3329.
#define E_crit_2 7500.
#define T_crit 1.5
DEFINE_IMPINGEMENT(dry_impingement, tp, rel_dot_n, f, t, y_s, E_imp)
{
  real one = 1.0, zero = 0.0;
```

```
      real fh = F_WALL_FILM_HEIGHT(f,t);
      real Re, denom;
      real abs_visc  = MAX(DPM_MU(tp),DPM_SMALL);
      real sigma = MAX(DPM_SURFTEN(tp),DPM_SMALL);
      real p_rho = MAX(TP_RHO(tp),DPM_SMALL);
      real d_0   = TP_DIAM(tp);
      real T_w = rf_energy ? F_T(f,t) : 300.0;
      real T_b = dpm_par.Tmax;
      int regime = FILM_STICK;

      *y_s = zero;
      Re = p_rho * d_0 * rel_dot_n / abs_visc;
      denom = sigma * (MIN(fh/d_0,1.0) + pow(Re,-0.5));
      *E_imp = SQR(rel_dot_n) * d_0 * p_rho / MAX(DPM_SMALL, denom);

      /* rf_energy is defined if Energy model is enabled. It is possible to use inert particle
      with isothermal conditions (Energy off). In that case the UDF returns FILM_STICK. */
      if (rf_energy && TP_LIQUID(tp))
        {
          Material *mb;
          if (TP_WET_COMBUSTION(tp))
            mb = TP_WET_COMB_MATERIAL(tp);
          else
            mb = TP_MATERIAL(tp);
          T_b = DPM_Boiling_Temperature(tp,mb);
        }
      else
        return FILM_STICK;

      if (T_w <= T_crit*T_b)
        {
          if (*E_imp <= E_crit_0)
            regime = FILM_STICK;
          else if (*E_imp > E_crit_0 && *E_imp <=  E_crit_1)
            regime = FILM_SPREAD;
          else if (*E_imp > E_crit_1)
            regime = FILM_SPLASH;
        }
      else
        {
          if (*E_imp < E_crit_1)
            regime = FILM_REBOUND;
          else
            regime = FILM_USER_0;
        }

      if ((regime == FILM_SPREAD) || (regime == FILM_STICK))
        *y_s = zero;
      else if ((regime == FILM_REBOUND)|| (regime == FILM_USER_0))
        *y_s = one;
      else if (regime == FILM_SPLASH)
        {
          if (*E_imp < E_crit_2)
            *y_s = MAX(0., 1.8e-4*(*E_imp - E_crit_1));
          else
            *y_s = 0.7;
        }
      return regime;
}
```

2.5.16.4. Hooking an Impingement UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_IMPINGEMENT` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name that you supplied as the first `DEFINE` macro argument will become visible and selectable in the **Discrete Phase Model** dialog box in

Ansys Fluent. See for details on how to hook your `DEFINE_IMPINGEMENT` UDF to Ansys Fluent.

2.5.17. `DEFINE_FILM_REGIME`

2.5.17.1. Description

You can use `DEFINE_FILM_REGIME` to set the particle variables for user-defined regimes of particle impingement. Fluent supports up to 10 user-defined regimes:

```
FILM_USER_0
FILM_USER_1
FILM_USER_2
FILM_USER_3
FILM_USER_4
FILM_USER_5
FILM_USER_6
FILM_USER_7
FILM_USER_8
FILM_USER_9
```

2.5.17.2. Usage

`DEFINE_FILM_REGIME (name, regime, tp, pp, f, t, f_normal, update)`

Argument Type	Description
`symbol name`	UDF name.
`int regime`	The particle impingement regime.
`Tracked_Particle *tp`	Pointer to the `Tracked_Particle` data structure which contains data related to the particle being tracked.
`Particle *pp`	Pointer to the `Particle` data structure where particles `pp` are stored in a linked list.
`face_t f`	Index of the face that the particle is currently hitting
`Thread *t`	Pointer to the face thread the particle is currently hitting
`real f_normal[]`	Array that contains the unit vector which is normal to the face
`int update`	Variable indicating whether the particle performs a sub-iteration, or a full tracking step. The value is `0` for sub-iteration and `1` for a full tracking step.

Function returns

```
void
```

There are eight arguments to `DEFINE_FILM_REGIME`: `name`, `regime`, `tp`, `pp`, `f`, `t`, `f_normal`, and `update`. You supply the `name` of your user-defined function. `regime`, `tp`, `pp`, `f`, `t`, `f_normal`, and `update` are passed by the Ansys Fluent solver to your UDF. Your function will need to compute the particle parameters and variables corresponding to the user-defined regime and modify the `Tracked_Particle *tp` and `Particle *pp` structures accordingly.

Note:

If you are defining multiple user-defined regimes, they must all be defined within a single `DEFINE_FILM_REGIME` UDF.

2.5.17.3. Example

The following UDF, named dry_breakup, describes a user-defined impingement regime where the impinging droplet particle bounces off the wall surface and breaks up into smaller droplets without any film formation. The ratio of the initial droplet diameter to the diameter after reflection is defined by the parameter `D_RATIO`. The UDF uses the Ansys Fluent function `Reflect_Particle` to compute the particle velocities after impingement. The reflected droplet's new diameter and mass, as well as the number in parcel and flowrate are updated in the `Tracked_Particle *tp` structure. Note that the mass and diameter of the current and previous time steps, as well as the initial values must be updated. The initial mass and diameter must also be adjusted in `Particle structure *pp` at the end of the tracking step.

```
#include "udf.h"
#define D_RATIO 5
#define DIAM_FROM_VOL(V) ((real)pow((6.0 * (V) / M_PI), 1./3.))

DEFINE_FILM_REGIME(dry_breakup, regime, tp, pp, f, t, f_normal, update)
{
  real d1 = TP_DIAM(tp);
  real d2 = MAX(TP_DIAM(tp)/D_RATIO,dpm_par.lowest_diam);
  real drat_cubed;

  if (regime == FILM_USER_0)
    {
      TP_ON_WALL(tp) = FALSE;
      tp->tracking_scheme = tp->free_stream_tracking_scheme;
      TP_FILM_FACE(tp) = NULL_FACE;
      TP_FILM_THREAD(tp) = NULL;
      tp->gvtp.n_rebound ++ ;
      Reflect_Particle(tp,f_normal,ND_ND,f,t, NULL, NULL_FACE);
      drat_cubed = CUB(d2/d1);
      TP_DIAM(tp) = d2;
      TP_N(tp) /= drat_cubed;
      TP_MASS(tp) *= drat_cubed;
      TP_FLOW_RATE(tp) /= drat_cubed;
      /* after diameter change all relevant variables in previous and initial states must be consistent */
      /* initialize the previous state mass and diameter to the current */
      TP_MASS0(tp) = TP_MASS(tp);
      TP_DIAM0(tp) = TP_DIAM(tp);
      /* initialize the initial state mass and diameter to the current */
      TP_INIT_MASS(tp) *= drat_cubed;
      TP_INIT_DIAM(tp) = DIAM_FROM_VOL(  TP_INIT_MASS(tp) /  TP_INIT_RHO(tp) );
      /* at the end of the iteration step update the initial mass
         and diameter in the stored particle list */
      if (update)
        {
          PP_INIT_MASS(pp) *= drat_cubed;
          PP_INIT_DIAM(pp) = DIAM_FROM_VOL(  PP_INIT_MASS(pp) /  PP_INIT_RHO(pp) );
```

```
            }
      }
   else
      {
        Error("User defined impingement regime not implemented: %d.\n",regime);
      }
}
```

2.5.17.4. Hooking a Film Regime UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_FILM_REGIME is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name that you supplied as the first DEFINE macro argument will become visible and selectable in the **Discrete Phase Model** dialog box in Ansys Fluent. See Hooking DEFINE_FILM_REGIME UDFs (p. 556) for details on how to hook your DEFINE_FILM_REGIME UDF to Ansys Fluent.

2.5.18. DEFINE_SPLASHING_DISTRIBUTION

2.5.18.1. Description

You can use DEFINE_SPLASHING_DISTRIBUTION to customize the diameter, number in parcel, and velocity distribution for the splashed particles.

2.5.18.2. Usage

DEFINE_SPLASHING_DISTRIBUTION (name, tp, rel_dot_n, f_normal, n_splash, splashing_distribution_t *s)

Argument Type	Description
symbol name	UDF name.
Tracked_Particle *tp	Pointer to the Tracked_Particle data structure which contains data related to the particle being tracked.
real real_dot_n	The particle impingement velocity magnitude
real f_normal[]	Array that contains the unit vector which is normal to the face
int n_splash	Number of splashed particle parcels.
splashing_distribution_t *s	Array of pointers to structure s of size n_splash, containing the splashed particle variables distribution defined by the following macros:

> SPLASH_DIAM(s,i) diameter of splashed particle i
>
> SPLASH_N(s,i) number in parcel of splashed particle i
>
> SPLASH_VEL(s,i)[3] velocity components of splashed particle i

Function returns

```
void
```

There are six arguments to DEFINE_SPLASHING_DISTRIBUTION: name, tp, real_dot_n, f_normal, n_splash, and s. You supply the name of your user-defined function. tp, real_dot_n, f_normal, and n_splash are passed by the Ansys Fluent solver to your UDF. Note that n_splash is the value entered for **Number of Splashed Drops** in the **DPM** tab of the boundary condition dialog box when **Boundary Cond. Type** is set to **wall-film**. The output of your function is the array of pointers to the structure *s containing the distribution of the n_splash particle parcels.

2.5.18.3. Example

The following UDF applies the O'Rourke & Amsden splashing model [10] (p. 713). This function must be run as a compiled UDF.

```
#include "udf.h"
#define NUKIYAMA_TANASAWA_TABLE_SIZE 15

static double nukiyama_tanasawa_table[NUKIYAMA_TANASAWA_TABLE_SIZE][2] =
  {
    {0.000000000000, 0.000000000000},
    {0.081108621634, 0.500000000000},
    {0.193908811137, 0.700000000000},
    {0.266112345561, 0.800000000000},
    {0.345136495805, 0.900000000000},
    {0.427593287415, 1.000000000000},
    {0.510077497807, 1.100000000000},
    {0.589500744183, 1.200000000000},
    {0.663337543595, 1.300000000000},
    {0.729766711799, 1.400000000000},
    {0.787709698251, 1.500000000000},
    {0.877181682854, 1.700000000000},
    {0.934793421924, 1.900000000000},
    {0.953988295926, 2.000000000000},
    {1.000000000000, 4.000000000000},
  };

double get_interpolated_value(double table[][2], int table_size, double inval)
{
  /* table is table[tableSize][1] with table[i][0] being the lookup values
     and table[i][1] the return values */

  int lowerindex, upperindex, midindex, jump;
  double lowerval, upperval, midval, ratio;

  lowerindex = 0;
  upperindex = table_size-1;

  lowerval = table[lowerindex][0];
  upperval = table[upperindex][0];

  jump = (upperindex-lowerindex)/2;

  /* Find values for Interpolation */

  while (jump)
    {
      midindex = lowerindex + jump;
      midval = table[midindex][0];

      if (inval > midval)
  {
    lowerval = midval;
```

```
    lowerindex = midindex;
 }
       else
 {
   upperval = midval;
   upperindex = midindex;
 }
       jump /= 2;
     }

 ratio = (inval-lowerval)/(upperval - lowerval);
 lowerval = table[lowerindex][1];
 upperval = table[upperindex][1];

 return lowerval + ratio*(upperval - lowerval);
}

double nukiyama_tanasawa_random()
{
 return get_interpolated_value(nukiyama_tanasawa_table, NUKIYAMA_TANASAWA_TABLE_SIZE, uniform_random());
}

double nabor_reitz_random(double beta)
{
 double exp_beta;
 double rand;
 double angle_range;

 angle_range = M_PI;
 rand = 1.0 - 2.0*uniform_random(); /* -1.0 < rand <= 1.0 */

 if (beta == 0.0)
   return angle_range * rand;

 if (rand < 0.0)
   {
     angle_range = -M_PI;
     rand = -rand;    /* 0.0 <= rand <= 1.0 */
   }
 exp_beta = exp(beta);

 return angle_range*(beta - log(exp_beta + rand*(1.0 - exp_beta)))/beta;
}

/*  O'Rourke & Amsden splashing model BC definition */

#define E_2_CRIT 3329.29 /* O'Rourke & Amsden critical energy squared (57.7^2) */

DEFINE_SPLASHING_DISTRIBUTION(splash, tp, rel_dot_n, norm, n_splashes, s)
{
 int i;
 real p_diam, p_v_mag; /* Particle properties */
 real w0, v0, v_t;     /* Particle condition  */
 real E_2;
 real sin_alpha;

 real e_t[ND_ND], e_p[ND_ND];
 /* Splashed particle properties */
 real weber;
 real d_max, w_dash, v_dash;
 real beta, psi, component;
 real total_volume = 0.;

 p_diam = TP_DIAM(tp);

 /* norm into wall and TP_VEL should be so w0 >= 0.0 */
 w0 = NV_DOT(TP_VEL(tp),norm);

 /* Particle impingement energy */
 E_2 = TP_IMPINGEMENT_PARAMETER(tp);
```

```
    /* Calculate new splash stream droplet diameters */
    weber = TP_WEBER_IMP(tp);

    d_max = MAX(MAX((E_2_CRIT/E_2),(6.4/weber)),0.06)*p_diam;

    for (i=0;i<n_splashes;i++)
      {
        SPLASH_DIAM(s,i) = d_max * nukiyama_tanasawa_random();
      }

    /* Calculate droplet velocities */
    NV_CROSS(e_p,TP_VEL(tp),norm);
    p_v_mag = NV_MAG(TP_VEL(tp));

    NV_S(e_p,/=,p_v_mag);
    sin_alpha = NV_MAG(e_p);

    /* If particle hits exactly normal to surface tilt slightly */
    sin_alpha = MAX(1.E-6, sin_alpha);
    NV_S(e_p,/=,sin_alpha);

    beta = M_PI*sqrt(sin_alpha/MAX(DPM_SMALL,(1.0-sin_alpha)));

    NV_CROSS(e_t,norm,e_p);

    v0 = NV_DOT(TP_VEL(tp),e_t); /* Tangential to wall */
    v_t = 0.8*v0;

    for (i=0;i<n_splashes;i++)
      {
        w_dash = -0.2 * w0 * nukiyama_tanasawa_random();
        NV_VS(SPLASH_VEL(s,i),=,norm,*,w_dash);

        v_dash = 0.1 * w0 * fabs(gauss_random()); /* O'Rourke and Amsden: delta = 0.1 w0 */
        v_dash += 0.12*w0;
        psi = nabor_reitz_random(beta);
        component = v_dash * cos(psi);
        NV_VS(SPLASH_VEL(s,i),+=,e_t,*,component);
        component = v_dash * sin(psi);
        NV_VS(SPLASH_VEL(s,i),+=,e_p,*,component);

        NV_VS(SPLASH_VEL(s,i),+=,e_t,*,v_t);
      }

    /*  The flow rate for each diameter will be proportional to its mass (=volume as density equal) */
    for (i=0;i<n_splashes;i++)
      {
        total_volume += CUB( SPLASH_DIAM(s,i));
      }
    for (i=0;i<n_splashes;i++)
      {
        SPLASH_N(s,i) = TP_SPLASHED_FRACTION(tp)*CUB(SPLASH_DIAM(s,i))/total_volume*TP_N(tp);
      }
}
```

2.5.18.4. Hooking a Splashing Distribution UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_SPLASHING_DISTRIBUTION` is compiled (Compiling UDFs (p. 399)) the name that you supplied as the first `DEFINE` macro argument will become visible and selectable in the **Discrete Phase Model** dialog box in Ansys Fluent. See Hooking `DEFINE_SPLASHING_DISTRIBUTION` UDFs (p. 558) for details on how to hook your `DEFINE_SPLASHING_DISTRIBUTION` UDF to Ansys Fluent.

2.6. Dynamic Mesh `DEFINE` Macros

This section contains descriptions of `DEFINE` macros that you can use to define UDFs that control the behavior of a dynamic mesh. Note that dynamic mesh UDFs that are defined using `DEFINE_CG_MOTION`, `DEFINE_DYNAMIC_ZONE_PROPERTY`, `DEFINE_GEOM`, and `DEFINE_GRID_MOTION` can *only* be executed as compiled UDFs.

Table 2.13: Quick Reference Guide for Dynamic Mesh-Specific DEFINE Macros (p. 263) provides a quick reference guide to the dynamic mesh `DEFINE` macros, the functions they define, and the dialog boxes where they are activated in Ansys Fluent. Definitions of each `DEFINE` macro are contained in the `udf.h` header file. For your convenience, they are listed in Appendix B: `DEFINE` Macro Definitions (p. 693).

Table 2.13: Quick Reference Guide for Dynamic Mesh-Specific DEFINE Macros

Function	**DEFINE Macro**	**Dialog Box Activated In**
center of gravity motion	DEFINE_CG_MOTION	**Dynamic Mesh Zones**
swirl center	DEFINE_DYNAMIC_ZONE_PROPERTY	**In-Cylinder Output Controls**
varying cell layering height	DEFINE_DYNAMIC_ZONE_PROPERTY	**Dynamic Mesh Zones**
mesh motion	DEFINE_GRID_MOTION	**Dynamic Mesh Zones**
periodic displacement	DEFINE_PERDISP_MOTION	**Periodic Displacement**
geometry deformation	DEFINE_GEOM	**Dynamic Mesh Zones**
properties for Six Degrees of Freedom (six DOF) Solver	DEFINE_SDOF_PROPERTIES	**Dynamic Mesh Zones**
contact detection	DEFINE_CONTACT	**Options**

2.6.1. `DEFINE_CG_MOTION`

2.6.1.1. Description

You can use `DEFINE_CG_MOTION` to specify the motion of a particular dynamic zone in Ansys Fluent by providing Ansys Fluent with the linear and angular velocities at every time step. Ansys Fluent uses these velocities to update the node positions on the dynamic zone based on solid-body motion.

Note that UDFs that are defined using `DEFINE_CG_MOTION` can ONLY be executed as compiled UDFs.

Important:

Depending on the mesh smoothing models used and the dynamic mesh setup, Ansys Fluent can call mesh motion UDFs multiple times during a single time step. Therefore, UDFs must be written so that the specified motion is an absolute function of time or so that the motion is not incremented incorrectly if the UDF is called multiple times. See the Example (p. 264) for a sample UDF.

2.6.1.2. Usage

`DEFINE_CG_MOTION (name, dt, vel, omega, time, dtime)`

Argument Type	Description
`symbol name`	UDF name.
`Dynamic_Thread *dt`	Pointer to structure that stores the dynamic mesh attributes that you have specified (or that are calculated by Ansys Fluent).
`real vel[]`	Linear velocity.
`real omega[]`	Angular velocity.
`real time`	Current time.
`real dtime`	Time step.

Function returns

`void`

There are six arguments to `DEFINE_CG_MOTION`: `name`, `dt`, `vel`, `omega`, `time`, and `dtime`. You supply `name`, the name of the UDF. `dt`, `vel`, `omega`, `time`, and `dtime` are variables that are passed by the Ansys Fluent solver to your UDF and have SI units. The linear and angular velocities are returned to Ansys Fluent by overwriting the arrays `vel` and `omega`, respectively.

When the **Relative Motion** option is enabled for the rigid body dynamic mesh zone, by default the motion of the dynamic zone `dt` is defined in the reference frame of the selected **Relative Zone**. If you want to specify its rotation and translation in the absolute reference frame, set `DT_NEST_LOC_ROT_P(dt)` and `DT_NEST_LOC_TRAN_P(dt)` to `FALSE`. The `DT_NEST_LOC_ROT_P(dt)` and `DT_NEST_LOC_TRAN_P(dt)` functions store a Boolean value (`TRUE` or `FALSE`) that determines whether the motion of the dynamic zone `dt` is specified relative to the **Relative Zone** reference frame or absolute reference frame for rotation and translation, respectively.

For additional information, see Rigid Body Motion in the *Fluent User's Guide*.

2.6.1.3. Example

Consider the following example where the linear velocity is computed from a simple force balance on the body in the X direction such that

$$\int_{t_o}^{t} dv = \int_{t_o}^{t} \left(F / m \right) dt \qquad (2.23)$$

where v is velocity, F is the force and m is the mass of the body. The velocity at time t is calculated using an explicit Euler formula as

$$v_t = v_{t-\Delta t} + \left(F / m \right) \Delta t \qquad (2.24)$$

```
/***********************************************************
* 1-degree of freedom equation of motion (x-direction)
* compiled UDF
***********************************************************/
#include "udf.h"

static real v_prev = 0.0;
static real time_prev = 0.0;

DEFINE_CG_MOTION(piston,dt,vel,omega,time,dtime)
{
  Thread *t;
  face_t f;
  real NV_VEC(A);
  real force_x, dv;

  /* reset velocities */
  NV_S(vel, =, 0.0);
  NV_S(omega, =, 0.0);
  if (!Data_Valid_P())
    return;
  /* get the thread pointer for which this motion is defined */
  t = DT_THREAD(dt);
  /* compute pressure force on body by looping through all faces */
  force_x = 0.0;
  begin_f_loop(f,t)
    {
      F_AREA(A,f,t);
      force_x += F_P(f,t) * A[0];
    }
  end_f_loop(f,t)
  /* compute change in velocity, dv = F*dt/mass */
  dv = dtime * force_x / 50.0;
  /* motion UDFs can be called multiple times and should not cause
     false velocity updates */
  if (time > (time_prev + EPSILON))
    {
      v_prev += dv;
      time_prev = time;
    }
  Message("time = %f, x_vel = %f, x_force = %f\n", time, v_prev, force_x);
  /* set x-component of velocity */
  vel[0] = v_prev;
}
```

2.6.1.4. Hooking a Center of Gravity Motion UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_CG_MOTION` is compiled (Compiling UD-Fs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible in the **Dynamic Mesh Zones** dialog box in Ansys Fluent. See Hooking `DEFINE_CG_MOTION` UDFs (p. 560) for details on how to hook your `DEFINE_CG_MOTION` UDF to Ansys Fluent.

2.6.2. `DEFINE_DYNAMIC_ZONE_PROPERTY`

2.6.2.1. Description

The `DEFINE_DYNAMIC_ZONE_PROPERTY` UDF can be used in the following applications:

- swirl center definition for in-cylinder applications

- variable cell layering height

2.6.2.2. Swirl Center Definition for In-Cylinder Applications

You can use `DEFINE_DYNAMIC_ZONE_PROPERTY` to calculate swirl center while computing in-cylinder specific output.

Important:

Note that UDFs that are defined using `DEFINE_DYNAMIC_ZONE_PROPERTY` can *only* be executed as compiled UDFs.

For information on setting in-cylinder parameters, see In-Cylinder Settings in the User's Guide.

2.6.2.2.1. Usage

`DEFINE_DYNAMIC_ZONE_PROPERTY (name, dt, swirl_center)`

Argument Type	Description
`symbol name`	UDF name.
`Dynamic_Thread *dt`	Pointer to a structure that stores the dynamic mesh attributes. This is set to NULL internally as there are no dynamic zones in the current calculation of swirl center.
`real *swirl_center`	Pointer to a real array of 3 dimension. You will assign this value in the UDF. The x, y and z values of the `swirl_center` can be assigned in the UDF through `swirl_center[0]`, `swirl_center[1]` and `swirl_center[2]` respectively.

Function returns

`void`

There are three arguments to `DEFINE_DYNAMIC_ZONE_PROPERTY`: `name`, `dt`, and `swirl_center`. You supply `name`, the name of the UDF, and pointer to a real array, `swirl_center`. `dt` is a variable that is passed by the Ansys Fluent solver to your UDF.

2.6.2.2.2. Example

```
/* UDF hook for calculating Swirl Center while computing
  In-Cylinder specific output. Arguments for the UDF
  hook are name of the UDF, dt (dynamic thread) which is
  set to NULL and it is not supposed to be manipulated
  in the UDF, as there are no dynamic zones in the current
  context and swirl center which is to be calculated in the
  UDF. Works in parallel as well.
*/

#include "udf.h"
#define RPM RP_Get_Real("dynamesh/in-cyn/crank-rpm")

static real Zmin_at_TDC = -0.0014; /* Piston location at TDC */
static real Zmax = 0.0145;   /* Zmax, a fixed point */

static void my_swirl_center(real * swirl_center)
{
   real piston_displacement, lambda, CA, l, r;
  #if !RP_NODE
     l = RP_Get_List_Ref_Float("dynamesh/in-cyn/piston-data", 0);
     r= 0.5 * RP_Get_List_Ref_Float("dynamesh/in-cyn/piston-data",1);
  #endif
   host_to_node_real_2(l,r);
   lambda = r/l;
   CA = (CURRENT_TIME*RPM*6.0 +
       RP_Get_Real("dynamesh/in-cyn/crank-start-angle"))*M_PI/180;
   piston_displacement = r*((1+1/lambda) - cos(CA) -
       pow(1-lambda*lambda*sin(CA)*sin(CA),0.5)/lambda);
   swirl_center[0]=0;
   swirl_center[1]=0;
   if (Zmin_at_TDC<Zmax)
     swirl_center[2]=0.5*(Zmin_at_TDC+Zmax-piston_displacement);
   else
     swirl_center[2]=0.5*(Zmin_at_TDC+Zmax+piston_displacement);
   return;
}

DEFINE_DYNAMIC_ZONE_PROPERTY(swirl_udf, dt, sc)
{
   my_swirl_center(sc);
}
```

2.6.2.2.3. Hooking a Swirl Center UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_DYNAMIC_ZONE_PROPERTY` is compiled (as described in Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible in the **In-Cylinder Output Controls** dialog box in Ansys Fluent.

See Hooking `DEFINE_DYNAMIC_ZONE_PROPERTY` UDFs (p. 561) for details on how to hook your `DEFINE_DYNAMIC_ZONE_PROPERTY` UDF to Ansys Fluent.

2.6.2.3. Variable Cell Layering Height

You can use `DEFINE_DYNAMIC_ZONE_PROPERTY` to specify a varying cell layering height when using the dynamic layering method to split or merge cells adjacent to a moving boundary. The cell

layering height can be specified as a function of time for general applications, or as a function of crank angle for in-cylinder applications.

Important:

Note that UDFs that are defined using `DEFINE_DYNAMIC_ZONE_PROPERTY` can *only* be executed as compiled UDFs.

For information on the dynamic layering method, see Dynamic Layering in the User's Guide.

2.6.2.3.1. Usage

`DEFINE_DYNAMIC_ZONE_PROPERTY (name, dt, height)`

Argument Type	Description
`symbol name`	UDF name.
`Dynamic_Thread *dt`	Pointer to a structure that stores the dynamic mesh attributes.
`real *height`	Pointer to a real value layering height whose value will be varied in the UDF as a function of time or crank angle.

Function returns

`void`

There are three arguments to `DEFINE_DYNAMIC_ZONE_PROPERTY`: `name`, `dt`, and `height`. You supply `name`, the name of the UDF, and `height`, the cell layering height to be assigned in the UDF as a function of time / crank angle. `dt` is a variable that is passed by the Ansys Fluent solver to your UDF.

In addition to the arguments listed previously, you can use the variable `in_cyl_ca_period` and the macros `DYNAMESH_CURRENT_TIME` and `TIME_TO_ABSOLUTE_CRANK_ANGLE (time)`, which are described as follows:

Variable/Macro	Description
`in_cyl_ca_period`	Crank angle period.
`DYNAMESH_CURRENT_TIME`	Current dynamic mesh time.
`TIME_TO_ABSOLUTE_CRANK_ANGLE (time)`	Macro which takes the current time as input and returns the absolute value of the crank angle that is displayed on the mesh preview screen.

Note that `in_cyl_ca_period` is the value entered for **Crank Period** in the **In-Cylinder** tab of the **Options** dialog box (which can be opened via the **Dynamic Mesh** task page). The usage of this variable or the macros specified previously necessitates that the `DEFINE_DYNAMIC_ZONE_PROPERTY` UDF be a compiled UDF. Their usage is illustrated in the example that follows.

Note that the header file dynamesh_tools.h should be included in the UDF, as shown in the example that follows.

2.6.2.3.2. Example

```
/* UDF hook for implementing varying cell layering height.
  Arguments are the Name of the UDF,
  variable for dynamic thread, and variable
  which holds the layering height value.
  Works only as a compiled UDF, because the usage of
  in_cyn_ca_period and the macros are not
  allowed in interpreted UDFs.
  Header file dynamesh_tools.h should be
  included in order to access the macros
  DYNAMESH_CURRENT_TIME and TIME_TO_ABSOLUTE_CRANK_ANGLE
*/

#include "udf.h"
#include "dynamesh_tools.h"

DEFINE_DYNAMIC_ZONE_PROPERTY(nonconst_height, dt, lh)
{
  int temp;

  /* Local variable for storing the value of
  Absolute Crank Angle */ real abs_ca;

  /* Local variables for saving time and
  Crank Angle, etc. */ real half,quart,time,ca;

  half = in_cyn_ca_period / 2.0;
  quart = in_cyn_ca_period /4.0;

  time = DYNAMESH_CURRENT_TIME;

  ca = TIME_TO_ABSOLUTE_CRANK_ANGLE(time);
  temp = (int) (ca / half);
  abs_ca = ca – temp * half ;
  /* *lh controls the layering height */
  if(abs_ca <= quart)
    *lh = (0.5 + (abs_ca)/ quart * 0.8);
  else
    *lh = (0.5 + ((half - abs_ca) / quart) * 0.8);

}
```

2.6.2.3.3. Hooking a Variable Cell Layering Height UDF to Ansys Fluent

After the UDF that you have defined using DEFINE_DYNAMIC_ZONE_PROPERTY is compiled (as described in Compiling UDFs (p. 399)), the name of the argument that you supplied as the first DEFINE macro argument will become visible in the **Dynamic Mesh Zones** dialog box in Ansys Fluent.

See Hooking DEFINE_DYNAMIC_ZONE_PROPERTY UDFs (p. 561) for details on how to hook your DEFINE_DYNAMIC_ZONE_PROPERTY UDF to Ansys Fluent.

2.6.3. `DEFINE_GEOM`

2.6.3.1. Description

You can use `DEFINE_GEOM` to specify the geometry of a deforming zone. By default, Ansys Fluent provides a mechanism for defining node motion along a planar or cylindrical surface. When Ansys Fluent updates a node on a deforming zone (for example, through spring-based smoothing or after local face re-meshing) the node is "repositioned" by calling the `DEFINE_GEOM` UDF. Note that UDFs that are defined using `DEFINE_GEOM` can *only* be executed as compiled UDFs.

2.6.3.2. Usage

`DEFINE_GEOM (name, d, dt, position)`

Argument Type	Description
`symbol name`	UDF name.
`Domain *d`	Pointer to domain.
`Dynamic_Thread *dt`	Pointer to structure that stores the dynamic mesh attributes that you have specified (or that are calculated by Ansys Fluent).
`real *position`	Pointer to array that stores the position.

Function returns

`void`

There are four arguments to `DEFINE_GEOM`: `name`, `d`, `dt`, and `position`. You supply `name`, the name of the UDF. `d`, `dt`, and `position` are variables that are passed by the Ansys Fluent solver to your UDF. The new position (after projection to the geometry defining the zone) is returned to Ansys Fluent by overwriting the `position` array.

2.6.3.3. Example

The following UDF, named `parabola`, is executed as a compiled UDF.

```
/***********************************************************
 * defining parabola through points (0, 1), (1/2, 5/4), (1, 1)
 ***********************************************************/
#include "udf.h"

DEFINE_GEOM(parabola,domain,dt,position)
{
   /* set y = -x^2 + x + 1 */
   position[1] = - position[0]*position[0] + position[0] + 1;
}
```

2.6.3.4. Hooking a Dynamic Mesh Geometry UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_GEOM` is compiled (see Compiling UDFs (p. 399) for details), the name of the argument that you supplied as the first `DEFINE` macro argument will

become visible in the **Dynamic Mesh Zones** dialog box in Ansys Fluent. See Hooking `DEFINE_GEOM` UDFs (p. 564) for details on how to hook your `DEFINE_GEOM` UDF to Ansys Fluent.

2.6.4. `DEFINE_GRID_MOTION`

2.6.4.1. Description

By default, Ansys Fluent updates the node positions on a dynamic zone by applying the solid-body motion equation. This implies that there is no relative motion between the nodes on the dynamic zone. However, if you need to control the motion of each node independently, then you can use `DEFINE_GRID_MOTION` UDF. A mesh motion UDF can, for example, update the position of each node based on the deflection due to fluid-structure interaction. Note that if your `DEFINE_GRID_MOTION` UDF defines boundary node motion that has components that should not contribute to the boundary condition, you must take additional steps in the setup, as described in User-Defined Motion in the *Fluent User's Guide*; an example of this is a boundary with nodes that undergo both rigid body motion (which should contribute to the boundary condition) and smoothing (which should not).

Note that UDFs that are defined using `DEFINE_GRID_MOTION` can be executed ONLY as compiled UDFs.

2.6.4.2. Usage

`DEFINE_GRID_MOTION` (name, d, dt, time, dtime)

Argument Type	Description
`symbol name`	UDF name.
`Domain *d`	Pointer to domain.
`Dynamic_Thread *dt`	Pointer to structure that stores the dynamic mesh attributes that you have specified (or that are calculated by Ansys Fluent).
`real time`	Current time.
`real dtime`	Time step.

Function returns

`void`

There are five arguments to `DEFINE_GRID_MOTION`: `name`, `d`, `dt`, `time`, and `dtime`. You supply `name`, the name of the UDF. `d`, `dt`, `time`, and `dtime` are variables that are passed by the Ansys Fluent solver to your UDF.

To specify a deformation as relative to a rigid body, you must use `NODE_COORD_NEST` to specify the total deformation, instead of specifying the absolute position of the nodes. If you want the deformation to be relative to the absolute reference frame, you must set `DT_NESTED_LOCAL_P`(dt) to `FALSE`. For additional information about defining a deformation on top of rigid body motion, see User-Defined Motion in the *Fluent User's Guide*.

2.6.4.3. Example

Consider the following example where you want to specify the deflection on a cantilever beam based on the x position such that

$$\omega_y(x) = -10.4\sqrt{x}\,\sin 26.178t \qquad x > 0.02$$
$$\omega_y(x) = 0 \qquad x <= 0.02$$

(2.25)

where $\omega_y(x)$ is the y-component of the angular velocity at a position x. The node position is updated based on

$$(\vec{r})^{t+\Delta t} = (\vec{r})^t + \vec{\Omega} \times (\vec{r})^t \Delta t$$

(2.26)

where $\vec{\Omega}$ is the angular velocity and $\vec{r}$ is the position vector of a node on the dynamic zone.

```
/**********************************************************
   node motion based on simple beam deflection equation
   compiled UDF
 **********************************************************/
#include "udf.h"

DEFINE_GRID_MOTION(beam,domain,dt,time,dtime)
{
   Thread *tf = DT_THREAD(dt);
   face_t f;
   Node *v;
   real NV_VEC(omega), NV_VEC(axis), NV_VEC(dx);
   real NV_VEC(origin), NV_VEC(rvec);
   real sign;
   int n;
   /* set deforming flag on adjacent cell zone */
   SET_DEFORMING_THREAD_FLAG(THREAD_T0(tf));
   sign = -5.0 * sin (26.178 * time);
   Message ("time = %f, omega = %f\n", time, sign);
   NV_S(omega, =, 0.0);
   NV_D(axis, =, 0.0, 1.0, 0.0);
   NV_D(origin, =, 0.0, 0.0, 0.152);
   begin_f_loop(f,tf)
   {
      f_node_loop(f,tf,n)
      {
       v = F_NODE(f,tf,n);
       /* update node if x position is greater than 0.02
       and that the current node has not been previously
       visited when looping through previous faces */
       if (NODE_X(v) > 0.020 && NODE_POS_NEED_UPDATE (v))
       {
        /* indicate that node position has been update
        so that it's not updated more than once */
        NODE_POS_UPDATED(v);
        omega[1] = sign * pow (NODE_X(v)/0.230, 0.5);
        NV_VV(rvec, =, NODE_COORD(v), -, origin);
        NV_CROSS(dx, omega, rvec);
        NV_S(dx, *=, dtime);
        NV_V(NODE_COORD(v), +=, dx);
       }
     }
  }
  end_f_loop(f,tf);
 }
```

2.6.4.4. Hooking a `DEFINE_GRID_MOTION` to Ansys Fluent

After the UDF that you have defined using `DEFINE_GRID_MOTION` is compiled (Compiling UD-Fs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible in the **Dynamic Mesh Zones** dialog box in Ansys Fluent. See Hooking `DEFINE_GRID_MOTION` UDFs (p. 566) for details on how to hook your `DEFINE_GRID_MOTION` UDF to Ansys Fluent.

2.6.5. `DEFINE_PERDISP_MOTION`

2.6.5.1. Description

You can use `DEFINE_PERDISP_MOTION` to specify any of the nodal based variables that compose the periodic displacement. This includes the real and/or imaginary mode shape displacement vector components, the mode frequency, amplitude/scaling, and blade passage number. Note that the `DEFINE_PERDISP_MOTION` UDF is executed only once before the simulation starts to fill the periodic displacement node variables.

2.6.5.2. Usage

`DEFINE_PERDISP_MOTION (name, dt, perdisp_array)`

Argument Type	Description
`symbol name`	UDF name.
`Dynamic_Thread *dt`	Pointer to structure that stores the dynamic mesh attributes that you have specified (or that are calculated by Ansys Fluent).
`double *perdisp_array`	Pointer to periodic displacement component node based array to be filled by the user.

Function returns

`void`

There are 3 arguments to `DEFINE_PERDISP_MOTION`: `name`, `dt`, and `perdisp_array`. You supply `name`, the name of the UDF. `dt` and `perdisp_array` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to fill the `perdisp_array` that you want to modify

2.6.5.3. Example

In the following example, you want to specify the Z component of a real mode shape to be used by the Periodic Displacement feature. This component is a function of the radial coordinate calculated at the initial (undeformed) coordinates of the mesh nodes.

```
/************************************************************
 node motion based on the bending motion of STCF-11 case
 compiled UDF
 ************************************************************/
#include "udf.h"

DEFINE_PERDISP_MOTION(z_displacement,tf,displacement)
```

```
{
  face_t f;
  Node *v;
  int n,node_count;

  Message ("filling Z real comp. displacement\n");

  real radial;

  begin_f_loop(f,tf)
  {
    f_node_loop(f,tf,n)SET_V_FLAGS(F_NODE(f,tf,n),MARK_FLAG);
  }
  end_f_loop(f,tf)

  node_count = 0;
  begin_f_loop(f,tf)
    {
      f_node_loop(f,tf,n)
        {
          v = F_NODE(f,tf,n);
          if (!NODE_FLAG(v,MARK_FLAG)) continue;
          radial = sqrt(NODE_COORD_REF(v)[0]*NODE_COORD_REF(v)[0]
          + NODE_COORD_REF(v)[1]*NODE_COORD_REF(v)[1]);
          displacement[node_count] = -cos(49.15*M_PI/180.0)*(0.063*radial-0.00594)*0.001;
          CLEAR_V_FLAGS(v, MARK_FLAG);
          node_count++;
        }
    }

  end_f_loop(f,tf);
}
```

2.6.5.4. Hooking a `DEFINE_PERDISP_MOTION` to Ansys Fluent

After the UDF that you have defined using `DEFINE_PERDISP_MOTION` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible in the **Periodic Displacement** dialog box.

2.6.6. `DEFINE_SDOF_PROPERTIES`

2.6.6.1. Description

You can use `DEFINE_SDOF_PROPERTIES` to specify custom properties of moving objects for the six-degrees of freedom (six DOF) solver in Ansys Fluent. These include mass, moment and products of inertia, constraints, and external forces and moment properties. The properties of an object which can consist of multiple zones can change in time, if desired. External load forces and moments can either be specified as global coordinates or body coordinates. In addition, you can specify custom transformation matrices using `DEFINE_SDOF_PROPERTIES`.

Note that if the moving object is modeled as a half model and includes a plane of symmetry, then you have to specify this by providing the normal vector to the symmetry plane. Further, all six DOF properties such as mass and moments of inertia have to be specified for the full body if a symmetry plane is specified.

2.6.6.2. Usage

`DEFINE_SDOF_PROPERTIES (name, properties, dt, time, dtime)`

Argument Type	Description
`symbol name`	UDF name.
`real *properties`	Pointer to the array that stores the six DOF properties.
`Dynamic_Thread *dt`	Pointer to structure that stores the dynamic mesh attributes that you have specified (or that are calculated by Ansys Fluent).
`real time`	Current time.
`real dtime`	Time step.

Function returns

`void`

There are five arguments to `DEFINE_SDOF_PROPERTIES`: `name`, `properties`, `dt`, `time`, and `dtime`. You provide the `name` of the UDF. `properties`, `dt`, `time`, and `dtime` are variables that are passed by the Ansys Fluent solver to your UDF. The property array pointer that is passed to your function allows you to specify values for any of the following six DOF properties:

```
SDOF_MASS    /* mass */
SDOF_IXX,    /* moment of inertia */
SDOF_IYY,    /* moment of inertia */
SDOF_IZZ,    /* moment of inertia */
SDOF_IXY,    /* product of inertia */
SDOF_IXZ,    /* product of inertia */
SDOF_IYZ,    /* product of inertia */
SDOF_LOAD_LOCAL, /* boolean */
SDOF_LOAD_F_X, /* external force */
SDOF_LOAD_F_Y, /* external force */
SDOF_LOAD_F_Z, /* external force */
SDOF_LOAD_M_X, /* external moment */
SDOF_LOAD_M_Y, /* external moment */
SDOF_LOAD_M_Z, /* external moment */
SDOF_ZERO_TRANS_X, /* boolean, suppress translation in x-direction */
SDOF_ZERO_TRANS_Y, /* boolean, suppress translation in y-direction */
SDOF_ZERO_TRANS_Z, /* boolean, suppress translation in z-direction */
SDOF_ZERO_ROT_X, /* boolean, suppress rotation around x-axis */
SDOF_ZERO_ROT_Y, /* boolean, suppress rotation around y-axis */
SDOF_ZERO_ROT_Z, /* boolean, suppress rotation around z-axis */
SDOF_SYMMETRY_X, /* normal vector of symmetry plane for half model */
SDOF_SYMMETRY_Y, /* normal vector of symmetry plane for half model */
SDOF_SYMMETRY_Z, /* normal vector of symmetry plane for half model */
```

In addition to these properties, you can also use macros to define one DOF translation / rotation settings. For details, see .

The Boolean `properties[SDOF_LOAD_LOCAL]` can be used to determine whether the forces and moments are expressed in terms of global coordinates (`FALSE`) or body coordinates (`TRUE`). The default value for `properties[SDOF_LOAD_LOCAL]` is `FALSE`.

The booleans `properties[SDOF_ZERO_TRANS_X]`, `properties[SDOF_ZERO_TRANS_Y]`, and so on can be used to freeze selected translational and rotational components of the six DOF motion. A value of `TRUE` will freeze the corresponding component. The default value is `FALSE`.

If your moving object consists of a half model, then the symmetry plane has to be specified by providing the components of the normal vector, `properties[SDOF_SYMMETRY_X]`, `proper-ties[SDOF_SYMMETRY_Y]`, and `properties[SDOF_SYMMETRY_Z]`.

2.6.6.3. Custom Transformation Variables

The default transformations used by Ansys Fluent are typical for most aerospace and other types of applications. However, if your model requires custom transformations, you can specify these matrices in your six DOF UDF. First set the `SDOF_CUSTOM_TRANS` Boolean to `TRUE`. Then use the macros listed below to define custom coordination rotation and derivative rotation matrices. `CTRANS` is the body-global coordinate rotation matrix and `DTRANS` is the body-global derivative rotation matrix.

```
SDOF_CUSTOM_TRANS,   /* boolean */
SDOF_CTRANS_11,   /* coordinate rotation matrices */
SDOF_CTRANS_12,
SDOF_CTRANS_13,
SDOF_CTRANS_21,
SDOF_CTRANS_22,
SDOF_CTRANS_23,
SDOF_CTRANS_31,
SDOF_CTRANS_32,
SDOF_CTRANS_33,
SDOF_DTRANS_11,   /* derivative rotation matrices */
SDOF_DTRANS_12,
SDOF_DTRANS_13,
SDOF_DTRANS_21,
SDOF_DTRANS_22,
SDOF_DTRANS_23,
SDOF_DTRANS_31,
SDOF_DTRANS_32,
SDOF_DTRANS_33,
```

2.6.6.4. Example 1

The following UDF, named `stage`, is a simple example of setting mass and moments of inertia properties for a moving object. This UDF is typical for applications in which a body is dropped and the six DOF solver computes the body's motion in the flow field.

```
/************************************************************
   Simple example of a six DOF property UDF for a moving body
 ************************************************************/
#include "udf.h"

DEFINE_SDOF_PROPERTIES(stage, prop, dt, time, dtime)
{
  prop[SDOF_MASS]  = 800.0;
  prop[SDOF_IXX]  = 200.0;
  prop[SDOF_IYY]  = 100.0;
  prop[SDOF_IZZ]  = 100.0;
  printf ("\nstage: updated 6DOF properties");
}
```

2.6.6.5. Example 2

The following UDF named `delta_missile` specifies case injector forces and moments that are time-dependent. Specifically, the external forces and moments depend on the current angular orientation of the moving object. Note that this UDF must be executed as a compiled UDF.

```
/*****************************************************
   Six DOF property compiled UDF with external forces/moments
 *****************************************************/
#include "udf.h"

DEFINE_SDOF_PROPERTIES(delta_missile, prop, dt, time, dtime)
```

```
  {
    prop[SDOF_MASS]  = 907.185;
    prop[SDOF_IXX]  = 27.116;
    prop[SDOF_IYY]  = 488.094;
    prop[SDOF_IZZ]  = 488.094;
    /* add injector forces, moments */
    {
     register real dfront = fabs (DT_CG (dt)[2] -
          (0.179832*DT_THETA (dt)[1]));
     register real dback = fabs (DT_CG (dt)[2] +
          (0.329184*DT_THETA (dt)[1]));
     if (dfront <= 0.100584)
       {
        prop[SDOF_LOAD_F_Z] = 10676.0;
        prop[SDOF_LOAD_M_Y] = -1920.0;
       }
     if (dback <= 0.100584)
       {
        prop[SDOF_LOAD_F_Z] += 42703.0;
        prop[SDOF_LOAD_M_Y] += 14057.0;
       }
    }
    printf ("\ndelta_missile: updated 6DOF properties");
  }
```

2.6.6.6. Example 3

The following UDF named `blade` is for a moving object that is constrained to one DOF rotation. The rotation axis is the vector (0,0,1) and the center of rotation is the point (0.1, 0, 0). The rotation is constrained between -0.5 radians and 0.5 radians. Note that this UDF must be executed as a compiled UDF.

```
/*******************************************************
   Six DOF property compiled UDF with one DOF rotation
 *******************************************************/
#include "udf.h"

DEFINE_SDOF_PROPERTIES(blade, sdof_prop, dt, time, dtime)
{
  Six_DOF_Object *sdof_obj = NULL;

  sdof_prop[SDOF_MASS] = 1.0;
  sdof_prop[SDOF_IXX] = 1.0;
  sdof_prop[SDOF_IYY] = 1.0;
  sdof_prop[SDOF_IZZ] = 1.0;
  sdof_prop[SDOF_LOAD_M_X] = 0.0;
  sdof_prop[SDOF_LOAD_M_Y] = 0.0;
  sdof_prop[SDOF_LOAD_M_Z] = 0.0;


  sdof_obj = Get_SDOF_Object(DT_PU_NAME(dt));

  if (NULLP(sdof_obj))
  {
    /* Allocate_SDOF_Object must be called with the same name as the udf */
    sdof_obj = Allocate_SDOF_Object(DT_PU_NAME(dt));

    SDOFO_1DOF_T_P(sdof_obj) = FALSE;       /* one DOF translation */
    SDOFO_1DOF_R_P(sdof_obj) = TRUE;        /* one DOF rotation */
    SDOFO_DIR(sdof_obj)[0] = 0.0;
    SDOFO_DIR(sdof_obj)[1] = 0.0;
    SDOFO_DIR(sdof_obj)[2] = 1.0;
    SDOFO_CENTER_ROT(sdof_obj)[0] = 0.1;    /* only needed for one DOF rotation */
    SDOFO_CENTER_ROT(sdof_obj)[1] = 0.0;    /* only needed for one DOF rotation */
    SDOFO_CENTER_ROT(sdof_obj)[2] = 0.0;    /* only needed for one DOF rotation */
    SDOFO_CONS_P(sdof_obj) = TRUE;          /* constrained */
```

```
    if (SDOFO_CONS_P(sdof_obj))
    {
      SDOFO_LOC(sdof_obj) = 0.0;
      SDOFO_MIN(sdof_obj) = -0.5;
      SDOFO_MAX(sdof_obj) = 0.5;
      SDOFO_F(sdof_obj) = 0.0;              /* spring preload */
      SDOFO_K(sdof_obj) = 0.0;              /* spring constant */

      SDOFO_INIT(sdof_obj) = SDOFO_LOC(sdof_obj);
      SDOFO_LOC_N(sdof_obj) = SDOFO_LOC(sdof_obj);
    }
  }
}
```

2.6.6.7. Hooking a `DEFINE_SDOF_PROPERTIES` UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_SDOF_PROPERTIES` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible in the **Six DOF UDF/Properties** drop-down list in the **Dynamic Mesh Zones** dialog box in Ansys Fluent. See Hooking `DEFINE_SDOF_PROPERTIES` UDFs (p. 567) for details on how to hook your `DEFINE_SDOF_PROPERTIES` UDF to Ansys Fluent.

2.6.7. `DEFINE_CONTACT`

2.6.7.1. Description

You can use `DEFINE_CONTACT` to specify the response to a contact detection event in Ansys Fluent. Note that UDFs that are defined using `DEFINE_CONTACT` can be executed only as compiled UDFs.

2.6.7.2. Usage

`DEFINE_CONTACT (name, dt, contacts)`

Argument Type	Description
`symbol name`	UDF name.
`Dynamic_Thread *dt`	Pointer to structure that stores the dynamic mesh attributes that you have specified (or that are calculated by Ansys Fluent).
`Objp *contacts`	Pointer to a NULL-terminated linked list of elements involved in the contact detection event.

Function returns

`void`

There are three arguments to `DEFINE_CONTACT`: `name`, `dt`, and `contacts`. You supply `name`, the name of the UDF. The `dt` and `contacts` structure pointers are passed by the Ansys Fluent solver to your UDF.

2.6.7.3. Example 1

```
/*************************************************************\
 * 2-degree of freedom equation of motion compiled UDF      *
\*************************************************************/

DEFINE_CONTACT(contact_props, dt, contacts)
{
  Objp *o;
  face_t face;
  Thread *thread;
  Domain *domain = NULL;
  Dynamic_Thread *ndt = NULL;

  int tid, nid, n_faces;

  real v0dotn1, v1dotn0;
  real nc_mag, norm0_mag, norm1_mag;

  real N3V_VEC (vel_rel);
  real N3V_VEC (nc), N3V_VEC (nctmp);
  real N3V_VEC (xc), N3V_VEC (xctmp);
  real N3V_VEC (vel0), N3V_VEC (omega0), N3V_VEC (theta0), N3V_VEC (norm0);
  real N3V_VEC (vel1), N3V_VEC (omega1), N3V_VEC (theta1), N3V_VEC (norm1);

  if (!Data_Valid_P())
    {
      return;
    }

  /* Define a common contact point / plane */
  N3V_S (nc, =, 0.0);
  N3V_S (xc, =, 0.0);

  /* Fetch current thread ID */
  tid = THREAD_ID (DT_THREAD (dt));

  nid = -1;
  n_faces = 0;

  loop (o, contacts)
    {
      face = O_F (o);
      thread = O_F_THREAD (o);

      /* Skip faces on current thread */
      if (THREAD_ID (thread) == tid)
        {
          continue;
        }

      /* Note ID of the other thread for posterity */
      if (nid == -1)
        {
          nid = THREAD_ID (thread);
        }

      /* Initialize to zero */
      N3V_S (nctmp, =, 0.0);
      N3V_S (xctmp, =, 0.0);

      F_AREA (nctmp, face, thread);
      F_CENTROID (xctmp, face, thread);

      /**
       * Negative sum because wall normals
       * point out of the fluid domain
       */
      N3V_V (nc, -=, nctmp);
      N3V_V (xc, +=, xctmp);
```

```
         n_faces++;
     }

# if RP_NODE
     {
       /* Reduce in parallel */
       nid = PRF_GIHIGH1 (nid);
       n_faces = PRF_GISUM1 (n_faces);

       PRF_GRSUM3 (nc[0], nc[1], nc[2]);
       PRF_GRSUM3 (xc[0], xc[1], xc[2]);
     }
# endif

  /* Propagate to host */
  node_to_host_int_2 (nid, n_faces);
  node_to_host_real_3 (nc[0], nc[1], nc[2]);
  node_to_host_real_3 (xc[0], xc[1], xc[2]);

  if (n_faces > 0)
    {
       nc_mag = N3V_MAG (nc) + REAL_MIN;

       N3V_S (nc, /=, nc_mag);
       N3V_S (xc, /=, n_faces);
    }
  else
    {
       return;
    }

  Message
  (
    "\nContact:: tid: %d nid: %d n_faces: %d "
    "Point: (%f %f %f) Normal: (%f %f %f)",
    tid, nid, n_faces,
    xc[0], xc[1], xc[2],
    nc[0], nc[1], nc[2]
  );

  /* Fetch thread for opposite body */
  domain = THREAD_DOMAIN (DT_THREAD (dt));
  thread = Lookup_Thread (domain, nid);

  if (NULLP (thread))
    {
       Message ("\nWarning: No thread for nid: %d ", nid);

       return;
    }
  else
    {
       ndt = THREAD_DT (thread);
    }

  /* Fetch body parameters */
  SDOF_Get_Motion (dt, vel0, omega0, theta0);

  /* Compute difference vectors and normalize */
  N3V_VV (norm0, =, xc, -, DT_CG (dt));
  norm0_mag = N3V_MAG (norm0) + REAL_MIN;
  N3V_S (norm0, /=, norm0_mag);

  if (NULLP (ndt))
    {
       /* Stationary body / wall. Use contact normal */
       N3V_V (norm1, =, nc);

       /* Compute relative velocity */
       N3V_S (vel1, =, 0.0);
```

```
              N3V_V (vel_rel, =, vel0);
        }
    else
        {
           /* Fetch body parameters */
           SDOF_Get_Motion (ndt, vel1, omega1, theta1);

           /* Compute relative velocity */
           N3V_VV (vel_rel, =, vel0, -, vel1);

           /* Compute difference vectors and normalize */
           N3V_VV (norm1, =, xc, -, DT_CG (ndt));
           norm1_mag = N3V_MAG (norm1) + REAL_MIN;
           N3V_S (norm1, /=, norm1_mag);

           /* Check if velocity needs to be reversed */
           if (N3V_DOT (vel_rel, nc) < 0.0)
             {
                /* Reflect velocity across the normal */
                v1dotn0 = 2.0 * N3V_DOT (vel1, norm0);

                N3V_S (norm0, *=, v1dotn0);
                N3V_V (vel1, -=, norm0);

                /* Override body velocity */
                SDOF_Overwrite_Motion (ndt, vel1, omega1, theta1);
             }
        }

  /* Check if velocity needs to be reversed */
  if (N3V_DOT (vel_rel, nc) < 0.0)
     {
        /* Reflect velocity across the normal */
        v0dotn1 = 2.0 * N3V_DOT (vel0, norm1);

        N3V_S (norm1, *=, v0dotn1);
        N3V_V (vel0, -=, norm1);

        /* Override body velocity */
        SDOF_Overwrite_Motion (dt, vel0, omega0, theta0);
     }

  Message
  (
     "\ncontact_props: Updated :: vel0 = (%f %f %f) vel1 = (%f %f %f)",
     vel0[0], vel0[1], vel0[2], vel1[0], vel1[1], vel1[2]
  );
}
```

2.6.7.4. Example 2

You can also use nodal contact information in a DEFINE_GRID_MOTION UDF that can be used to constrain grid motion in certain situations.

```
/*********************************************************************\
 * This UDF provides the moving-deforming mesh model with the        *
 * locations of the nodes on a deforming boundary. The UDF uses      *
 * nodal contact information to constrain motion.                    *
 * Compiled UDF, all metric units                                    *
\*********************************************************************/

#include "udf.h"

static real L = 2.5;
static real xoffset = 2.8;
static real amplitude = 1.0;
static real total_time = 15.0;
```

```
DEFINE_GRID_MOTION(wall_deform_top, domain, dt, time, dtime)
{
  int n;
  Node *node;
  face_t face;

  real A, factor, fraction;

  Thread *thread = DT_THREAD (dt);

  /**
   * Set/activate the deforming flag on adjacent cell zone, which
   * means that the cells adjacent to the deforming wall will also be
   * deformed, in order to avoid skewness.
   */
  SET_DEFORMING_THREAD_FLAG (THREAD_T0 (thread));

  /**
   * Loop over the deforming boundary zone faces;
   * inner loop loops over all nodes of a given face;
   * Thus, since one node can belong to several faces, one must guard
   * against operating on a given node more than once:
   */
  begin_f_loop (face, thread)
    {
      f_node_loop (face, thread, n)
        {
          node = F_NODE (face, thread, n);

          /* Compute the amplitude as a function of time */
          fraction = (time / total_time);

          factor = -1.0 * fabs (2.0 * (fraction - floor (fraction + 0.5)));

          A = factor * amplitude * sin (M_PI * (NODE_X (node) - xoffset) / L);

          /*
           * If node is in contact and motion is downward,
           * prevent further motion.
           */
          if (NODE_POS_CONTACT_P (node) && ((A + 1.0) < NODE_Y (node)))
            {
              NODE_POS_UPDATED (node);
              continue;
            }

          /*
           * Update the current node only if it has not been
           * previously visited:
           */
          if (NODE_POS_NEED_UPDATE (node))
            {
              /**
               * Set flag to indicate that the current node's
               * position has been updated, so that it will not be
               * updated during a future pass through the loop:
               */
              NODE_POS_UPDATED (node);

              NODE_Y (node) = A + 1.0;
            }
        }
    }
  end_f_loop (face, thread);
}
```

2.6.7.5. Hooking a `DEFINE_CONTACT` UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_CONTACT` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible in the **UDF** drop-down list in the **Contact Detection** tab of the **Options** dialog box in Ansys Fluent. See Hooking `DEFINE_CONTACT` UDFs (p. 568) for details on how to hook your `DEFINE_CONTACT` UDF to Ansys Fluent.

2.7. Battery Model `DEFINE` Macros

This section contains descriptions of `DEFINE` macros for the MSMD-based battery model. The functions can be hooked up from the **Battery Model** dialog box (**UDF** tab). Table 2.14: Quick Reference Guide for Battery Model-Specific `DEFINE` Macros (p. 283) provides a quick reference guide to the battery model `DEFINE` macros, the functions they define, and the dialog boxes where they are activated in Ansys Fluent. For your convenience, they are listed in Battery Model `DEFINE` Macros (p. 697). Definitions of each DEFINE macro are contained in the `udf.h` header file.

Note:

All the battery model UDFs can only be compiled and cannot be run in the interpreted mode.

Table 2.14: Quick Reference Guide for Battery Model-Specific `DEFINE` Macros

Function	`DEFINE` Macro	Dialog Box Activated In
cell clustering target variable	`DEFINE_BATTERY_CLUSTER`	**Battery Model**
voltage-current relationship for the user-defined electrochemical model	`DEFINE_BATTERY_ECHEM_MODEL`	**Battery Model**
entropic heat term	`DEFINE_BATTERY_ENTROPIC_HEAT`	**Battery Model**
the Butler-Volmer rate for the Newman's P2D model	`DEFINE_BATTERY_NEWMAN_BV_RATE`	**Battery Model**
export the Newman's P2D model results	`DEFINE_BATTERY_NEWMAN_POSTPRO-CESSING`	**Battery Model**

Function	**`DEFINE` Macro**	**Dialog Box Activated In**
parameters for the ECM model	`DEFINE_BATTERY_PARAMETER_ECM`	**Battery Model**
`U` and `Y` functions for the NTGK model	`DEFINE_BATTERY_PARAMETER_NTGK`	**Battery Model**
material properties for the Newman's P2D model	`DEFINE_BATTERY_PROPERTY`	**Battery Model**

2.7.1. `DEFINE_BATTERY_CLUSTER`

2.7.1.1. Description

You can use `DEFINE_BATTERY_CLUSTER` to specify the target variable in cell clustering.

2.7.1.2. Usage

`DEFINE_BATTERY_CLUSTER` (name, c, t)

Argument Type   **Description**

`symbol name`   UDF name.

`cell_t c`   Cell index.

`Thread *t`   Pointer to the cell thread.

Function returns

`real`

There are three arguments to `DEFINE_BATTERY_CLUSTER`: `name`, `c`, and `t`. You supply `name`, the name of the UDF. `c` and `t` are variables that are passed by the Ansys Fluent solver to your UDF. Your function will return a value of the cell clustering variable.

2.7.1.3. Example

The example shows the usage of `DEFINE_BATTERY_CLUSTER`. In a jelly roll battery simulation, radius could be used as the target variable. The following code shows an example of such a UDF.

```
#include "udf.h"

DEFINE_BATTERY_CLUSTER(battery_clustering, c, t)
{
    real value, x[3];

  C_CENTROID(x, c, t);
  value = sqrt(x[0]*x[0] + x[1]*x[1]);  /* use radius as the clustering criterion */

    return value;
}
```

2.7.1.4. Hooking a `DEFINE_BATTERY_CLUSTER` UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_BATTERY_CLUSTER` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible and selectable in the **Battery Model** dialog box (**Model Option** tab). See Hooking Battery Model UDFs (p. 573) for details.

2.7.2. `DEFINE_BATTERY_ECHEM_MODEL`

2.7.2.1. Description

You can use `DEFINE_BATTERY_ECHEM_MODEL` to specify the voltage-current relationship used in the user-defined electrochemical model. You need to define both scenarios:

- computing voltage when current is specified
- computing current when voltage is specified

2.7.2.2. Usage

`DEFINE_BATTERY_ECHEM_MODEL` (name, zero_start, mode, temperature, voltage_in, current_in, dt, current_out, Qe, voltage_out)

Argument Type	Description
`symbol name`	UDF name.
`int zero_start`	Flag to indicate whether or not the submodel is to be initialized. A value of 0 indicates the initial UDF call requesting submodel initialization. A value of 1 indicates the subsequent call.
`int mode`	Flag indicating which electric load boundary type is used: • `mode` = 0 indicates that the voltage load is used. `voltage_in` is the input, and `current_out` needs to be computed • `mode` = 1 indicates that the current load is used. `current_in` is the input, and `voltage_out` needs to be computed.
`real temperature`	Temperature at the time the submodel is called.
`real voltage_in`	A voltage value. This value is used as an input when the voltage load boundary condition is used.
`real current_in`	A current value. This value is used as an input when the current load boundary condition is used.

Argument Type	Description
`real dt`	Time step.
`real *current_out`	Pointer to the computed current value. This value needs to be computed when the voltage input condition is used.
`real *voltage_out`	Pointer to the computed voltage value. This value needs to be computed when the current input condition is used.
`eal Qe`	Heat generation rate.

Function returns

`void`

There are ten arguments to `DEFINE_BATTERY_ECHEM_MODEL`: `name`, `zero_start`, `mode`, `temperature`, `voltage_in`, `current_in`, `dt`, `current_out`, `Qe`, and `voltage_out`. You supply `name`, the name of the UDF. `zero_start`, `mode`, `temperature`, `voltage_in`, `current_in`, and `dt` are variables that are passed by the Ansys Fluent solver to your UDF. Your function will need to compute the values of `current_out`, `Qe`, and `voltage_out`.

2.7.2.3. Example

This example shows the usage of `DEFINE_BATTERY_ECHEM_MODEL`. The NTGK model is reproduced by using the user-defined electrochemical model.

```
#include "udf.h"
DEFINE_BATTERY_ECHEM_MODEL(echem_model, zero_start, mode, temperature, voltage_in, current_in, \
                           dtime, current_out, Qe, voltage_out)
{
  real DOD, U_fun, Y_fun;
  real a[]     = {4.12, -0.804, 1.075, -1.177, 0.0, 0.0};
  real b[]     = {1168.59, -8928, 52504.6, -136231, 158531.7, -67578.5};
  real T_coef[] = {1800.0, -0.00095};
  real T_ref    =  298.0;

  if (zero_start == 0) USER_DEFINED_ECHEM[0] = 0.0;
  DOD = USER_DEFINED_ECHEM[0];

  U_fun = a[0] + a[1] * DOD + a[2] * DOD * DOD + a[3] * DOD * DOD * DOD \
          + a[4] * DOD * DOD * DOD * DOD + a[5] * DOD * DOD * DOD * DOD * DOD;
  U_fun -= T_coef[1] * (temperature - T_ref);


  Y_fun = b[0] + b[1] * DOD + b[2] * DOD * DOD + b[3] * DOD * DOD * DOD \
          + b[4] * DOD * DOD * DOD * DOD + b[5] * DOD * DOD * DOD * DOD * DOD;
  Y_fun *= exp(T_coef[0] * (1. / T_ref - 1. / temperature));

  if (mode == 0)
  {
    *current_out  = Y_fun  * (U_fun - voltage_in);
    *Qe = *current_out * (U_fun - voltage_in);
    USER_DEFINED_ECHEM[0] += *current_out * dtime / (3600.0 * 14.6);
  }
  else
  {
    *voltage_out = U_fun - current_in / Y_fun;
  }
}
```

2.7.2.4. Hooking a `DEFINE_BATTERY_ECHEM_MODEL` UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_BATTERY_ECHEM_MODEL` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible and selectable in the **Battery Model** dialog box (**UDF** tab). See Hooking Battery Model UDFs (p. 573) for details.

2.7.3. `DEFINE_BATTERY_ENTROPIC_HEAT`

2.7.3.1. Description

You can use `DEFINE_BATTERY_ENTROPIC_HEAT` to compute the entropic heat term in the battery thermal analysis if you want to include the entropic heat effect in your simulation. The entropic heat can be included in all the battery submodels including the NTGK, ECM and Newman's P2D models.

2.7.3.2. Usage

`DEFINE_BATTERY_ENTROPIC_HEAT (name, soc)`

Argument Type	Description
`symbol name`	UDF name.
`real soc`	State of the charge (SOC).

Function returns

`real`

There are two arguments to `DEFINE_BATTERY_ENTROPIC_HEAT`: `name`, and `soc`. You supply `name`, the name of the UDF. `soc` is a variable that is passed by the Ansys Fluent solver to your UDF. Your function will return a value of the $\partial U/\partial T$ term in the entropic heat calculation (see Equation 20.8, Equation 20.13, and Equation 20.30 in the *Fluent Theory Guide*).

2.7.3.3. Example

The entropic heat is ignored in the default Ansys Fluent battery model, which corresponds to $\partial U/\partial T$ =0. This example shows how to use `DEFINE_BATTERY_ENTROPIC_HEAT` to compute the $\partial U/\partial T$ term in the entropic heat calculation.

```
#include "udf.h"
DEFINE_BATTERY_ENTROPIC_HEAT(entropic_heat, soc)
{
    real dUdT = 0.0;
    return dUdT;
}
```

2.7.3.4. Hooking a `DEFINE_BATTERY_ENTROPIC_HEAT` UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_BATTERY_ENTROPIC_HEAT` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro

argument will become visible and selectable in the **Battery Model** dialog box (**UDF** tab). See
Hooking Battery Model UDFs (p. 573) for details.

2.7.4. `DEFINE_BATTERY_NEWMAN_BV_RATE`

2.7.4.1. Description

You can use `DEFINE_BATTERY_NEWMAN_BV_RATE` to customize the Butler-Volmer rate used in
the Newman's P2D model. See Newman's P2D Model in the *Fluent Theory Guide* for more information.

2.7.4.2. Usage

`DEFINE_BATTERY_NEWMAN_BV_RATE` (name, Ce, Cs, Cs_max, T, eta, k_m, al-
pha_a, alpha_c, mode)

Argument Type	Description
`symbol name`	UDF name.
`real Ce`	Lithium ion concentration in electrolyte.
`real Cs`	Lithium concentration in electrode.
`real Cs_max`	Maximum lithium concentration in electrode.
`real T`	Temperature.
`real eta`	Overpotential ($\phi_s - \phi_e - U$).
`real k_m`	Rate pre-exponential factor.
`real alpha_a`	Anodic charge transfer coefficient.
`real alpha_c`	Cathodic charge transfer coefficient.
`int mode`	Flag to indicate whether the reaction is in anode or cathode. A value of 0 indicates anode, and a value of 1 indicates cathode.

Function returns

`real`

There are ten arguments to `DEFINE_BATTERY_NEWMAN_BV_RATE`: `name`, `Ce`, `Cs`, `Cs_max`, `T`,
`eta`, `k_m`, `alpha_a`, `alpha_c`, and `mode`. You supply `name`, the name of the UDF. `Ce`, `Cs`, `Cs_max`,
`T`, `eta`, `k_m`, `alpha_a`, `alpha_c`, and `mode` are variables that are passed by the Ansys Fluent
solver to your UDF. Your function will return the Butler-Volmer rate.

2.7.4.3. Example

This example shows the usage of `DEFINE_BATTERY_NEWMAN_BV_RATE` to compute the electro-
chemical reaction rate for both cathode and anode using the default Butler-Volmer equations with
the kinetics parameters.

```
#include "udf.h"
DEFINE_BATTERY_NEWMAN_BV_RATE(p2d_bv_rate, Ce, Cs, Cs_max, T, eta, k_m, alpha_a, alpha_c, mode)
{
   real rate, i_0;
   i_0 = k_m * pow(Ce, alpha_a)  * pow(Cs, alpha_c) * pow(Cs_max - Cs, alpha_a);
```

```
    rate = i_0 * (exp(alpha_a * 9.6485e4 / (8.314 * T) * eta) - exp(-alpha_c * 9.6485e4 / (8.314 * T) * eta));
    return rate;
}
```

2.7.4.4. Hooking a `DEFINE_BATTERY_NEWMAN_BV_RATE` UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_BATTERY_NEWMAN_BV_RATE` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible and selectable in the **Battery Model** dialog box (**UDF** tab). See Hooking Battery Model UDFs (p. 573) for details.

2.7.5. `DEFINE_BATTERY_NEWMAN_POSTPROCESSING`

2.7.5.1. Description

You can use `DEFINE_BATTERY_NEWMAN_POSTPROCESSING` to export the detailed information of the Newman's P2D model results. See Newman's P2D Model in the *Fluent Theory Guide* for more information.

2.7.5.2. Usage

`DEFINE_BATTERY_NEWMAN_POSTPROCESSING (name, c, t, T, Vp, Vn)`

Argument Type	Description
`symbol name`	UDF name.
`real c`	Index of the cell calling the function.
`Thread *t`	Pointer to the thread of the function calling cell.
`real T`	Temperature.
`real Vp`	Cathode potential of the electrode.
`real Vn`	Anode potential of the electrode.

Function returns

`void`

There are six arguments to `DEFINE_BATTERY_NEWMAN_POSTPROCESSING`: `name`, `c`, `t`, `T`, `Vp`, and `Vn`. You supply `name`, the name of the UDF. `c`, `t`, `T`, `Vp`, and `Vn` are variables that are passed by the Ansys Fluent solver to your UDF. You can use the UDF to export information such as potential and lithium concentration to a file or print it in the console window.

2.7.5.3. Example

This example shows how to use `DEFINE_BATTERY_NEWMAN_POSTPROCESSING` to print some fundamental variables of the Newman's P2D model to the console window.

The function is called every time the Newman's P2D solver is called. The UDF prints a large amount of information. You may want to add some controls to print or save only selected variables.

```
#include "udf.h"
DEFINE_BATTERY_NEWMAN_POSTPROCESSING(p2d_output, c, t, T, Vp, Vn)
{
    int i, j;

    Message(" Newman model results for cell %d of Thread_id %d at Time=%f\n", c, \
              NNULLP (t) ? THREAD_ID(t) : -1, CURRENT_TIME);
    Message(" T=%f Vp=%f Vn=%f \n", T, Vp, Vn);

    Message(" Grid center location\n");
    Message("NE\n");
    for (i = 0; i < battery_n_XN; i++) Message(" %g\n", battery_X_cv_NE[i]);
    Message("SP\n");
    for (i = 0; i < battery_n_XS; i++) Message(" %g\n", battery_X_cv_SP[i]);
    Message("PE\n");
    for (i = 0; i < battery_n_XP; i++) Message(" %g\n", battery_X_cv_PE[i]);

    {
      Message("RP\n");
      for (i = 0; i < battery_n_RP; i++) Message(" %g\n", battery_R_cv_PE[i]);
      Message("RN\n");
      for (i = 0; i < battery_n_RN; i++) Message(" %g\n", battery_R_cv_NE[i]);
    }

    Message("Positive Electrode: battery_C_CS_P and battery_C_CE_P \n");
    for (i = 0; i < battery_n_XP; i++)
    {
      for (j = 0; j < battery_n_RP; j++)
      {
        Message("%12.5e  ", battery_CsPE[i][j]);
      }
      Message("\n");
      Message("%12.5e\n",  battery_CePE[i]);
    }

    Message("Separator  battery_C_CE \n");
    for (i = 0; i < battery_n_XS; i++)
    {
      Message(" %12.5e\n", battery_CeSP[i]);
    }

    Message("Negative Electrode: battery_C_CS_N and battery_C_CE_N \n");
    for (i = 0; i < battery_n_XN; i++)
    {
      for (j = 0; j < battery_n_RN; j++)
      {
        Message("%12.5e  ", battery_CsNE[i][j]);
      }
      Message("\n");
      Message("%12.5e\n",  battery_CeNE[i]);
    }

    Message("potential profile in electrolyte\n");
    for (i = 0; i < battery_n_XN; i++)
      Message(" %le \n", battery_PhieNE[i]);
    for (i = 0; i < battery_n_XS; i++)
      Message(" %le \n", battery_PhieSP[i]);
    for (i = 0; i < battery_n_XP; i++)
      Message(" %le \n", battery_PhiePE[i]);

    Message("potential profile in electrode\n");
    for (i = 0; i < battery_n_XN; i++)
      Message(" %le \n", battery_PhisNE[i]);
    for (i = 0; i < battery_n_XP; i++)
      Message(" %le \n", battery_PhisPE[i]);


    for (i = 0; i < battery_n_XN; i++)
      Message(" %12.5e %12.5e %12.5e %12.5e %12.5e %12.5e\n", battery_kappa_NE[i], battery_diff_NE[i], \
              battery_t_plus_NE[i], battery_kappa2_NE[i], battery_eta_NE[i], battery_i_NE[i]);
    for (i = 0; i < battery_n_XS; i++)
```

```
      Message(" %12.5e %12.5e %12.5e %12.5e\n", battery_kappa_SP[i], battery_diff_SP[i], \
                battery_t_plus_SP[i], battery_kappa2_SP[i]);
   for (i = 0; i < battery_n_XP; i++)
      Message(" %12.5e %12.5e %12.5e %12.5e %12.5e %12.5e\n", battery_kappa_PE[i], battery_diff_PE[i], \
             battery_t_plus_PE[i], battery_kappa2_PE[i], battery_eta_PE[i], battery_i_PE[i]);

   return ;
}
```

2.7.5.4. Hooking a `DEFINE_BATTERY_NEWMAN_POSTPROCESSING` UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_BATTERY_NEWMAN_POSTPROCESSING` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible and selectable in the **Battery Model** dialog box (**UDF** tab). See Hooking Battery Model UDFs (p. 573) for details.

2.7.6. `DEFINE_BATTERY_PARAMETER_ECM`

2.7.6.1. Description

You can use `DEFINE_BATTERY_PARAMETER_ECM` to specify your own parameters for the ECM model. See ECM Model in the *Fluent Theory Guide* for details.

2.7.6.2. Usage

`DEFINE_BATTERY_PARAMETER_ECM` (name, c, t, T, soc, mode, Voc, Rs, R1, C1, R2, C2, R3, C3)

Argument Type	Description
symbol name	UDF name.
cell_t c	Cell index.
Thread *t	Pointer to the thread of the active zone.
int mode	Integer flag that indicates charging/discharging mode. A value of 1 indicates discharging, while a value of 0 indicates charging.
real T	Temperature.
real soc	Value of the state of the charge (SOC).
real *Voc	Pointer to the V_{oc} value.
real *RS	Pointer to the R_s value.
real *R1	Pointer to the R_1 value.
real *C1	Pointer to the C_1 value.
real *R2	Pointer to the R_2 value.
real *C2	Pointer to the C_2 value.
real *R3	Pointer to the R_3 value.

Argument Type	Description
`real *C3`	Pointer to the C_3 value.

Function returns

`void`

There are fourteen arguments to `DEFINE_BATTERY_PARAMETER_ECM`: `name`, `c`, `t`, `T`, `soc`, `mode`, `Voc`, `Rs`, `R1`, `C1`, `R2`, `C2`, `R3`, and `C3`. You will supply `name`, the name of the UDF. `c`, `t`, `T`, `soc`, and `mode` are variables that are passed by the Ansys Fluent solver to your UDF. Your function will need to compute the values of `Voc`, `Rs`, `R1`, `C1`, `R2`, `C2`, `R3`, and `C3`.

2.7.6.3. Example

This example shows how to use `DEFINE_BATTERY_PARAMETER_ECM` to reproduce the default polynomial functions in Chen's form.

The UDF uses a 3 RC loop circuit. The default Chen's circuit uses only 2 RC loops. Small values of R_3 and C_3 are used to skip the third RC loop.

```
#include "udf.h"
DEFINE_BATTERY_PARAMETER_ECM(ecm_model_parameter, c, t, T, soc, mode, Voc, RRs, RR1, CC1, RR2, CC2, RR3, CC3)
{
  *Voc = 3.685 - 1.031*exp(-35.0*soc) + 0.2156*soc - 0.1178*soc*soc + 0.3201*soc*soc*soc;
  *RRs = 0.07446 + 0.1562*exp(-24.37*soc);
  *RR1   = 0.04669 + 0.3208*exp(-29.14*soc);
  *RR2   = 0.04984 + 6.603*exp(-155.2*soc);
  *CC1   = 703.6   - 752.9*exp(-13.51*soc);
  *CC2   = 4475.0  - 6056.*exp(-27.12*soc);
  *RR3   = 1.0e-5;
  *CC3   = 1.0e-5;
}
```

2.7.6.4. Hooking a `DEFINE_BATTERY_PARAMETER_ECM` UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_BATTERY_PARAMETER_ECM` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible and selectable in the **Battery Model** dialog box (**UDF** tab). See Hooking Battery Model UDFs (p. 573) for details.

2.7.7. `DEFINE_BATTERY_PARAMETER_NTGK`

2.7.7.1. Description

You can use `DEFINE_BATTERY_PARAMETER_NTGK` to specify your own `U` and `Y` functions used in the NTGK model. See NTGK Model in the *Fluent Theory Guide* for details.

2.7.7.2. Usage

`DEFINE_BATTERY_PARAMETER_NTGK (name, c, t, T, soc, U, Y)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *t`	Pointer to the thread of the active zone.
`real *T`	Temperature.
`real soc`	Value of the state of charge (SOC).
`real *U`	Pointer to the value of `U`.
`real *Y`	Pointer to the value of `Y`.

Function returns

`void`

There are seven arguments to `DEFINE_BATTERY_PARAMETER_NTGK`: `name`, `c`, `t`, `T`, `soc`, `U`, and `Y`. You will supply `name`, the name of the UDF. `c`, `t`, `T`, and `soc` are variables that are passed by the Ansys Fluent solver to your UDF. Your function will need to compute the values of `U` and `Y`.

2.7.7.3. Example

This example shows how to use `DEFINE_BATTERY_PARAMETER_NTGK` to reproduce the default polynomial `U` and `Y` functions.

```
#include "udf.h"
DEFINE_BATTERY_PARAMETER_NTGK(ntgk_model_parameter, c, t, T, soc, U, Y)
{
  real a[] = {4.12, -0.804, 1.075, -1.177, 0.0, 0.0};
  real b[] = {1168.59, -8928, 52504.6, -136231, 158531.7, -67578.5};
  real C[] = {1800.0, -0.00095};
  real DOD = 1.0 - soc;
  real T_ref = 298.0;

  *U = a[0]+a[1]*DOD+a[2]*DOD*DOD+a[3]*DOD*DOD*DOD \
      + a[4]*DOD*DOD*DOD*DOD + a[5]*DOD*DOD*DOD*DOD*DOD;
  *U -= C[1]*(T - T_ref);

  *Y = b[0]+b[1]*DOD+b[2]*DOD*DOD+b[3]*DOD*DOD*DOD \
      + b[4]*DOD*DOD*DOD*DOD+b[5]*DOD*DOD*DOD*DOD*DOD;

  *Y *= exp(C[0]*(1./T_ref - 1./T)));
}
```

2.7.7.4. Hooking a `DEFINE_BATTERY_PARAMETER_NTGK` UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_BATTERY_PARAMETER_NTGK` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible and selectable in the **Battery Model** dialog box (**UDF** tab). See Hooking Battery Model UDFs (p. 573) for details.

2.7.8. `DEFINE_BATTERY_PROPERTY`

2.7.8.1. Description

You can use `DEFINE_BATTERY_PROPERTY` to specify material properties used in the Newman's P2D model. See Newman's P2D Model in the *Fluent Theory Guide* for more information.

2.7.8.2. Usage

`DEFINE_BATTERY_PROPERTY(name, x, T)`

Argument Type	Description
`symbol name`	UDF name.
`real x`	Dependent variable of material property.
`real T`	Temperature.

Function returns

`real`

There are three arguments to `DEFINE_BATTERY_PROPERTY`: `name`, `x`, and `T`. You supply `name`, the name of the UDF. `x` and `T` are variables that are passed by the Ansys Fluent solver to your UDF. These are the variables the material property depends on. For example, if the UDF is used to define open circuit potential, then `x` will be the soc of the electrode, that is, the ratio of the lithium concentration at the electrode surface and the maximum electrode concentration. If the UDF is used to define any electrolyte property, then `x` will be the lithium concentration in the electrolyte. If the UDF is used for any solid electrode property, then `x` will be the lithium concentration in the electrode. Your function will return a value of the material property.

2.7.8.3. Example

This example shows the usage of `DEFINE_BATTERY_PROPERTY`. The first two UDFs are used to compute the open circuit potential used for the cathode and anode electrode. And the last one is used to compute the ionic conductivity of the electrolyte material. The UDFs are used to reproduce the default settings used in the Newman's P2D model.

```
#include "udf.h"

DEFINE_BATTERY_PROPERTY(ocp_p, x, T)
{
  real UPE, xx;
  xx = -14.555 * x + 8.609;
  xx *= 2.0;
  UPE = 4.199 + 0.0566 * (exp(xx) - 1.0) / (exp(xx) + 1.0)
        - 0.0275 * (pow(0.998 - x, -0.492) - 1.901)
        - 0.1570 * exp(-0.0474 * pow(x, 8.0))
        + 0.8100 * exp(-40.0 * (x - 0.134));
   return UPE;
}

DEFINE_BATTERY_PROPERTY(ocp_n, x, T)
{
  real UNE = -0.16 + 1.32 * exp(-3.0 * x) + 10.0 * exp(-2000.0 * x);
  return UNE;
}
```

```
DEFINE_BATTERY_PROPERTY(kappa, ce, T)
{
  return 1.0793e-2 + (6.7461e-4) * ce - (5.2245e-7) * ce * ce
          + (1.3605e-10) * ce * ce * ce - (1.1724e-14) * ce * ce * ce * ce;
}
```

2.7.8.4. Hooking a `DEFINE_BATTERY_PROPERTY` UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_BATTERY_PROPERTY` is compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument will become visible and selectable in the **Battery Model** dialog box (**Model Parameter** tab). See Hooking Battery Model UDFs (p. 573) for details.

2.8. User-Defined Scalar (UDS) Transport Equation `DEFINE` Macros

This section provides information on how you can define UDFs that can be used in UDS transport equations in Ansys Fluent. See User-Defined Scalar (UDS) Transport Equations in the User's Guide for UDS equation theory and details on how to set up scalar equations. Descriptions of `DEFINE` macros for UDS applications are provided below. Definitions of `DEFINE` macros are contained in the `udf.h` header file. For your convenience, they are also listed in Appendix B: `DEFINE` Macro Definitions (p. 693). Detailed examples of user-defined scalar transport UDFs can be found in User-Defined Scalars (p. 642).

2.8.1. Introduction

2.8.2. DEFINE_ANISOTROPIC_DIFFUSIVITY

2.8.3. DEFINE_UDS_FLUX

2.8.4. DEFINE_UDS_UNSTEADY

2.8.1. Introduction

For each of the N scalar equations you specified in your Ansys Fluent model you can supply a unique UDF for the diffusion coefficients, flux, and unsteady terms in the scalar transport equation. For multiphase you have the added benefit of specifying UDFs on a per-phase basis in both fluid and solid zones. Additionally, you can specify a UDF for each source term you define for a given scalar equation as well as boundary conditions on wall, inflow, and outflow boundaries.

2.8.1.1. Diffusion Coefficient UDFs

For each of the N scalar equations you have specified in your Ansys Fluent model using the **User-Defined Scalars** dialog box you can supply a unique user-defined function (UDF) for isotropic and anisotropic diffusivity for both fluid and solid materials. Recall that Ansys Fluent computes the diffusion coefficient in the UDS equation.

Isotropic diffusivity UDFs are defined using the `DEFINE_DIFFUSIVITY` macro (`DEFINE_DIFFUSIVITY` (p. 60)) and anisotropic coefficients UDFs are defined using `DEFINE_ANISOTROPIC_DIFFUSIVITY` (`DEFINE_ANISOTROPIC_DIFFUSIVITY` (p. 297)). Additional pre-defined macros that you can use when coding UDS functions are provided in User-Defined Scalar (UDS) Transport Equation Macros (p. 354).

2.8.1.2. Flux UDFs

For each of the N scalar equations you have specified in your Ansys Fluent model using the **User-Defined Scalars** dialog box you can supply a unique user-defined function (or UDF) for the advective flux term. Recall that Ansys Fluent computes the flux in the UDS equation.

UDS Flux UDFs are defined using the `DEFINE_UDS_FLUX` macro (`DEFINE_UDS_FLUX` (p. 299)). Additional pre-defined macros that you can use when coding scalar flux UDFs are provided in User-Defined Scalar (UDS) Transport Equation Macros (p. 354).

2.8.1.3. Unsteady UDFs

For each of the N scalar equations you have specified in your Ansys Fluent model using the **User-Defined Scalars** dialog box you can supply a unique UDF for the unsteady function. Recall that Ansys Fluent computes the unsteady term in the UDS equation.

Scalar Unsteady UDFs are defined using the `DEFINE_UDS_UNSTEADY` macro (`DEFINE_UDS_UNSTEADY` (p. 302)). Additional pre-defined macros that you can use when coding scalar unsteady UDFs are provided in User-Defined Scalar (UDS) Transport Equation Macros (p. 354).

2.8.1.4. Source Term UDFs

For each of the N scalar equations you have specified in your Ansys Fluent model using the **User-Defined Scalars** dialog box you can supply a unique UDF for *each* source. Recall that Ansys Fluent computes the source term in the UDS equation.

Scalar source UDFs are defined using the `DEFINE_SOURCE` macro and must compute the source term, S_{ϕ_k}, and its derivative $\frac{\partial S_{\phi_k}}{\partial \phi_k}$ (`DEFINE_SOURCE` (p. 148)). Additional pre-defined macros that you can use when coding scalar source term UDFs are provided in User-Defined Scalar (UDS) Transport Equation Macros (p. 354).

2.8.1.5. Fixed Value Boundary Condition UDFs

For each of the N scalar equations you have specified in your Ansys Fluent model using the **User-Defined Scalars** dialog box you can supply a fixed value profile UDF for fluid boundaries.

Fixed value UDFs are defined using the `DEFINE_PROFILE` macro. See `DEFINE_PROFILE` (p. 108) for details. Additional pre-defined macros that you can use for coding scalar transport equation UDFs are provided in User-Defined Scalar (UDS) Transport Equation Macros (p. 354).

2.8.1.6. Wall, Inflow, and Outflow Boundary Condition UDFs

For each of the N scalar equations you have specified in your Ansys Fluent model using the **User-Defined Scalars** dialog box you can supply a specified value or flux UDF for all wall, inflow, and outflow boundaries.

Wall, inflow, and outflow boundary UDFs are defined using the `DEFINE_PROFILE` macro (`DEFINE_PROFILE` (p. 108)). Additional pre-defined macros that you can use for coding scalar transport equation UDFs are provided in User-Defined Scalar (UDS) Transport Equation Macros (p. 354).

2.8.2. `DEFINE_ANISOTROPIC_DIFFUSIVITY`

2.8.2.1. Description

You can use `DEFINE_ANISOTROPIC_DIFFUSIVITY` to specify an anisotropic diffusivity for a user-defined scalar (UDS) transport equation. See Anisotropic Diffusion in the User's Guide for details about anisotropic diffusivity material properties in Ansys Fluent.

2.8.2.2. Usage

`DEFINE_ANISOTROPIC_DIFFUSIVITY (name, c, t, i, dmatrix)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *t`	Pointer to cell thread on which the anisotropic diffusivity function is to be applied.
`int i`	Index that identifies the user-defined scalar.
`real dmatrix[ND_ND][ND_ND]`	Anisotropic diffusivity matrix to be filled in by user.

Function returns

`void`

There are five arguments to `DEFINE_ANISOTROPIC_DIFFUSIVITY`: `name`, `c`, `t`, `i`, and `dmatrix`. You will supply name, the name of the UDF. `c`, `t`, `i`, and `dmatrix` are variables that are passed by the Ansys Fluent solver to your UDF. Your function will compute the diffusivity tensor for a single cell and fill `dmatrix` with it. Note that anisotropic diffusivity UDFs are called by Ansys Fluent from within a loop on cell threads. Consequently, your UDF will not need to loop over cells in a thread since Ansys Fluent is doing it outside of the function call.

2.8.2.3. Example

The following UDF, named `cyl_ortho_diff` computes the anisotropic diffusivity matrix for a cylindrical shell which has different diffusivities in radial, tangential, and axial directions. This function can be executed as a compiled UDF.

```
/***************************************************************
   Example UDF that demonstrates DEFINE_ANISOTROPIC_DIFFUSIVITY
 ***************************************************************/
#include "udf.h"

/* Computation of anisotropic diffusivity matrix for
 * cylindrical orthotropic diffusivity */

/* axis definition for cylindrical diffusivity */
static const real origin[3] = {0.0, 0.0, 0.0};
static const real axis[3]   = {0.0, 0.0, 1.0};

/* diffusivities in radial, tangential and axial directions */
static const real diff[3] = {1.0, 0.01, 0.01};
```

```
DEFINE_ANISOTROPIC_DIFFUSIVITY(cyl_ortho_diff,c,t,i,dmatrix)
{
   real x[3][3]; /* principal direction matrix for cell in cartesian coords. */
   real xcent[ND_ND];
   real R;
   C_CENTROID(xcent,c,t);
   NV_VV(x[0],=,xcent,-,origin);
#if RP_3D
     NV_V(x[2],=,axis);
#endif
#if RP_3D
     R = NV_DOT(x[0],x[2]);
     NV_VS(x[0],-=,x[2],*,R);
#endif
   R = NV_MAG(x[0]);
   if (R > 0.0)
     NV_S(x[0],/=,R);
#if RP_3D
 N3V_CROSS(x[1],x[2],x[0]);
#else
     x[1][0] = -x[0][1];
     x[1][1] = x[0][0];
#endif
    /* dmatrix is computed as xT*diff*x */
   dmatrix[0][0] = diff[0]*x[0][0]*x[0][0]
      + diff[1]*x[1][0]*x[1][0]
#if RP_3D
    + diff[2]*x[2][0]*x[2][0]
#endif
;
   dmatrix[1][1] = diff[0]*x[0][1]*x[0][1]
    + diff[1]*x[1][1]*x[1][1]
#if RP_3D
    + diff[2]*x[2][1]*x[2][1]
#endif
    ;
   dmatrix[1][0] = diff[0]*x[0][1]*x[0][0]
    + diff[1]*x[1][1]*x[1][0]
#if RP_3D
    + diff[2]*x[2][1]*x[2][0]
#endif
;
   dmatrix[0][1] = dmatrix[1][0];

#if RP_3D
   dmatrix[2][2] = diff[0]*x[0][2]*x[0][2]
    + diff[1]*x[1][2]*x[1][2]
    + diff[2]*x[2][2]*x[2][2]
    ;
   dmatrix[0][2] = diff[0]*x[0][0]*x[0][2]
    + diff[1]*x[1][0]*x[1][2]
    + diff[2]*x[2][0]*x[2][2]
    ;
   dmatrix[2][0] = dmatrix[0][2];
  dmatrix[1][2] = diff[0]*x[0][1]*x[0][2]
    + diff[1]*x[1][1]*x[1][2]
    + diff[2]*x[2][1]*x[2][2]
    ;
   dmatrix[2][1] = dmatrix[1][2];
  #endif
}
```

2.8.2.4. Hooking an Anisotropic Diffusivity UDF to Ansys Fluent

After the UDF that you have defined using `DEFINE_ANISOTROPIC_DIFFUSIVITY` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that

you supplied as the first `DEFINE` macro argument (for example, `cyl_ortho_diff`) will become selectable via the **UDS Diffusion Coefficients** dialog box. You'll first need to select **defined-per-uds** for **UDS Diffusivity** in the **Create/Edit Materials** dialog box, then select the `user-defined-anisotropic` option for **Coefficient** from the **UDS Diffusion Coefficients** dialog box for a particular user-defined scalar diffusion equation (for example, `uds-0`). See Hooking `DEFINE_ANISO-TROPIC_DIFFUSIVITY` UDFs (p. 570) for details.

2.8.3. `DEFINE_UDS_FLUX`

2.8.3.1. Description

You can use `DEFINE_UDS_FLUX` to customize how the advective flux term is computed in your user-defined scalar (UDS) transport equations. Note that this macro is for fluid cell zones only, as it cannot be used to calculate convective flux in a solid zone. See User-Defined Scalar (UDS) Transport Equations in the User's Guide for details on setting up and solving UDS transport equations.

2.8.3.2. Usage

`DEFINE_UDS_FLUX(name,f,t,i)`

Argument Type	Description
`symbol name`	UDF name.
`face_t f`	Face index.
`Thread *t`	Pointer to face thread on which the user-defined scalar flux is to be applied.
`int i`	Index that identifies the user-defined scalar for which the flux term is to be set.

Function returns

`real`

There are four arguments to `DEFINE_UDS_FLUX`: `name`, `f`, `t`, and `i`. You supply `name`, the name of the UDF. `f`, `t`, and `i` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to return the `real` value of the mass flow rate through the given face to the solver.

The advection term in the differential transport equation has the following most general form:

$$\nabla \cdot \vec{\psi}\phi \tag{2.27}$$

where ϕ is the user-defined scalar conservation quantity and $\vec{\psi}$ is a vector field. In the default advection term, $\vec{\psi}$ is, by default, the product of the scalar density and the velocity vector:

$$\vec{\psi}_{default} = \rho\vec{v} \tag{2.28}$$

To define the advection term in Equation 2.27 (p. 299) using `DEFINE_UDS_FLUX`, your UDF must return the scalar value $\vec{\psi} \cdot \vec{A}$ to Ansys Fluent, where $\vec{\psi}$ is the same as defined in Equation 2.27 (p. 299) and $\vec{A}$ is the face normal vector of the face.

Important:

Note that the advective flux field that is supplied by your UDF should be divergence-free (that is, it satisfies the continuity equation). In discrete terms this means that the sum of fluxes over all the faces of each cell should be zero. If the advective field is not divergence-free, then ϕ is not "conserved" and will result in overshoots/undershoots in the cell value of ϕ.

You will need to compute $\vec{\psi}$ in your UDF using, for example, predefined macros for velocity vector and scalar density that Ansys Fluent has provided (see Additional Macros for Writing UDFs (p. 305)) or using your own prescription. The first case is illustrated in the sample C source code, shown below.

Important:

Note that if more than one scalar is being solved, you can use a conditional `if` statement in your UDF to define a different flux function for each `i`. `i = 0` is associated with scalar-0 (the first scalar equation being solved).

Important:

Note also that $\vec{\psi} \cdot \vec{A}$ must have units of mass flow rate in SI (that is, kg/s).

```
/************************************************************************
    sample C source code that computes dot product of psi and A
    Note that this is not a complete C function
 ************************************************************************/

real NV_VEC(psi), NV_VEC(A);    /* declaring vectors psi and A */
        /* defining psi in terms of velocity field */
NV_D(psi, =, F_U(f,t), F_V(f,t), F_W(f,t));

NV_S(psi, *=, F_R(f,t))   /* multiplying density to get psi vector  */

F_AREA(A,f,t)     /* face normal vector returned from F_AREA */

return NV_DOT(psi,A);   /* dot product of the two returned   */
```

Additionally, since most quantities in Ansys Fluent are not allocated in memory for interior faces, only for boundary faces (for example, wall zones), your UDF will also need to calculate interior face values from the cell values of adjacent cells. This is most easily done using the arithmetic mean method. Vector arithmetic can be coded in C using the `NV_` and `ND_` macros (see Additional Macros for Writing UDFs (p. 305)).

Note that if you had to implement the default advection term in a UDF without the fluid density in the definition of ψ (see above), you could simply put the following line in your `DEFINE_UDS_FLUX` UDF:

```
return F_FLUX(f,t) / rho;
```

where the denominator ρ can be determined by averaging the adjacent cell's density values `C_R(F_C0(f,t),THREAD_T0(t))` and `C_R(F_C1(f,t),THREAD_T1(t))`.

2.8.3.3. Example

The following UDF, named `my_uds_flux`, returns the mass flow rate through a given face. The flux is usually available through the Ansys Fluent-supplied macro `F_FLUX(f,t)` (Face Macros (p. 322)). The sign of flux that is computed by the Ansys Fluent solver is positive if the flow direction is the same as the face area normal direction (as determined by `F_AREA` - see Face Area Vector (`F_AREA`) (p. 322)), and is negative if the flow direction and the face area normal directions are opposite. By convention, face area normals always point out of the domain for boundary faces, and they point in the direction from cell `c0` to cell `c1` for interior faces.

The UDF must be executed as a compiled UDF.

```
/**********************************************************************/
/* UDF that implements a simplified advective term in the   */
/* scalar transport equation          */
/**********************************************************************/

#include "udf.h"

DEFINE_UDS_FLUX(my_uds_flux,f,t,i)
{
   cell_t c0, c1 = -1;
   Thread *t0, *t1 = NULL;
   real NV_VEC(psi_vec), NV_VEC(A), flux = 0.0;
   c0 = F_C0(f,t);
   t0 = F_C0_THREAD(f,t);
   F_AREA(A, f, t);
   /* If face lies at domain boundary, use face values; */
   /* If face lies IN the domain, use average of adjacent cells. */
  if (BOUNDARY_FACE_THREAD_P(t)) /*Most face values will be available*/
   {
      real dens;
      /* Depending on its BC, density may not be set on face thread*/
      if (NNULLP(THREAD_STORAGE(t,SV_DENSITY)))
        dens = F_R(f,t);  /* Set dens to face value if available */
      else
        dens = C_R(c0,t0); /* else, set dens to cell value */
      NV_DS(psi_vec, =, F_U(f,t), F_V(f,t), F_W(f,t), *, dens);
      flux = NV_DOT(psi_vec, A); /* flux through Face */  }
   else
   {
    c1 = F_C1(f,t);    /* Get cell on other side of face */
    t1 = F_C1_THREAD(f,t);
    NV_DS(psi_vec, =, C_U(c0,t0),C_V(c0,t0),C_W(c0,t0),*,C_R(c0,t0));
    NV_DS(psi_vec, +=, C_U(c1,t1),C_V(c1,t1),C_W(c1,t1),*,C_R(c1,t1));
    flux = NV_DOT(psi_vec, A)/2.0; /* Average flux through face */
   }
 /* Ansys Fluent will multiply the returned value by phi_f (the scalar's
 value at the face) to get the ''complete'' advective term. */
 return flux;
}
```

2.8.3.4. Hooking a UDS Flux Function to Ansys Fluent

After the UDF that you have defined using `DEFINE_UDS_FLUX` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as

the first `DEFINE` macro argument (for example, `my_uds_flux`) will become visible and selectable in the **User-Defined Scalars** dialog box in Ansys Fluent. See Hooking `DEFINE_UDS_FLUX` UDFs (p. 571) for details.

2.8.4. `DEFINE_UDS_UNSTEADY`

2.8.4.1. Description

You can use `DEFINE_UDS_UNSTEADY` to customize unsteady terms in your user-defined scalar (UDS) transport equations. See User-Defined Scalar (UDS) Transport Equations in the User's Guide for details on setting up and solving UDS transport equations.

2.8.4.2. Usage

`DEFINE_UDS_UNSTEADY(name,c,t,i,apu,su)`

Argument Type	Description
`symbol name`	UDF name.
`cell_t c`	Cell index.
`Thread *t`	Pointer to cell thread on which the unsteady term for the user-defined scalar transport equation is to be applied.
`int i`	Index that identifies the user-defined scalar for which the unsteady term is to be set.
`real *apu`	Pointer to central coefficient.
`real *su`	Pointer to source term.

Function returns

`void`

There are six arguments to `DEFINE_UDS_UNSTEADY`: `name`, `c`, `t`, `i`, `apu`, and `su`. You supply `name`, the name of the UDF. `c`, `t`, and `i` are variables that are passed by the Ansys Fluent solver to your UDF. Your UDF will need to set the values of the unsteady terms referenced by the `real` pointers `apu` and `su` to the central coefficient and source term, respectively.

The Ansys Fluent solver expects that the transient term will be decomposed into a source term, `su`, and a central coefficient term, `apu`. These terms are included in the equation set in a similar manner to the way the explicit and implicit components of a source term might be handled. Hence, the unsteady term is moved to the right-hand side and discretized as follows:

$$
\begin{aligned}
\text{unsteadyterm} \quad &= \quad -\int \frac{\partial}{\partial t}(\rho\phi)\, dV \\
&\approx \quad -\left[\frac{(\rho\phi)^{n}-(\rho\phi)^{n-1}}{\Delta t}\right]\cdot \Delta V \\
&= \quad \underbrace{-\frac{\rho\Delta V}{\Delta t}}_{\text{apu}}\phi^{n}+\underbrace{\frac{\rho\Delta V}{\Delta t}}_{\text{su}}\phi^{n-1}
\end{aligned}
\tag{2.29}
$$

Equation 2.29 (p. 302) shows how `su` and `apu` are defined. Note that if more than one scalar is being solved, a conditional `if` statement can be used in your UDF to define a different unsteady term for each `i`. `i`= 0 is associated with scalar-0 (the first scalar equation being solved).

2.8.4.3. Example

The following UDF, named `my_uds_unsteady`, modifies user-defined scalar time derivatives using `DEFINE_UDS_UNSTEADY`. The source code can be interpreted or compiled in Ansys Fluent.

```
/************************************************************************
   UDF for specifying user-defined scalar time derivatives
 ************************************************************************/

#include "udf.h"

DEFINE_UDS_UNSTEADY(my_uds_unsteady,c,t,i,apu,su)
{
   real physical_dt, vol, rho, phi_old;
   physical_dt = RP_Get_Real("physical-time-step");
   vol = C_VOLUME(c,t);
   rho = C_R_M1(c,t);
   *apu = -rho*vol / physical_dt;/*implicit part*/
   phi_old = C_STORAGE_R(c,t,SV_UDSI_M1(i));
   *su = rho*vol*phi_old/physical_dt;/*explicit part*/
}
```

2.8.4.4. Hooking a UDS Unsteady Function to Ansys Fluent

After the UDF that you have defined using `DEFINE_UDS_UNSTEADY` is interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)), the name of the argument that you supplied as the first `DEFINE` macro argument (for example, `my_uds_unsteady`) will become visible and selectable in the **User-Defined Scalars** dialog box in Ansys Fluent. See Hooking `DEFINE_UDS_UNSTEADY` UDFs (p. 572) for details.

Chapter 3: Additional Macros for Writing UDFs

This chapter provides predefined macros that you can use when defining your user-defined function (UDF).

3.1. Introduction

Ansys Fluent provides numerous C types, functions, and preprocessor macros to facilitate the programming of UDFs and the use of CFD objects as defined inside Ansys Fluent. The previous chapter presented `DEFINE` macros with which you must define your UDF. This chapter presents predefined functions (implemented as macros in the code) that are supplied by Ansys Fluent that you will use to code your UDF. These macros allow you to access Ansys Fluent solver data such as cell variables (for example, cell temperature, centroid), face variables (for example, face temperature, area), or connectivity variables (for example, adjacent cell thread and index) that your UDF can use in a computation. Ansys Fluent provides:

- Macros commonly used in UDFs that return such values as the thread ID pointer (an internal Ansys Fluent structure) when they are passed the Zone ID (the number assigned to a zone in a boundary conditions dialog box).

- The `F_PROFILE` macro, which enables your UDF to set a boundary condition value in the solver.

- Other macros that enable your function to loop over nodes, cells, and faces in a thread or domain in order to retrieve and/or set values.

- Data access macros that are specific to a particular model (for example, DPM, NOx).

- Macros that perform vector, time-dependent, Scheme, and I/O operations.

Function definitions for the macros provided in this chapter are contained in header files. Header files are identified by the `.h` suffix as in `mem.h`, `metric.h`, `dpm_types.h`, and `dpm_laws.h` and are stored in multiple directories under the following:

`path\ANSYS Inc\v212\fluent\fluent21.2.0\src`

where *path* is the folder in which you have installed Ansys Fluent (by default, the path is `C:\Program Files`).

The header files, unless explicitly noted, are included in the `udf.h` file, so your UDF does not need to contain a special `#include` compiler directive. You must, however, remember to include the `#include "udf.h"` directive in any UDF that you write.

Access to data from an Ansys Fluent solver is accomplished by hooking your UDF C function (after it is compiled or interpreted) to the code through the graphical user interface (GUI). After the UDF is correctly hooked, the solver's data is passed to the function and is available to use whenever it is called. These data are automatically passed by the solver to your UDF as function arguments. Note that all solver data, regardless of whether they are passed to your UDF by the solver or returned to the solver by the UDF, are specified in SI units. Macros in this chapter are listed with their arguments, argument types, returned values (if applicable), and header file.

Each function behind a macro either outputs a value to the solver as an argument, or returns a value that is then available for assignment in your UDF. Input arguments belong to the following Ansys Fluent data types:

`Node *node`	pointer to a node
`cell_t c`	cell identifier
`face_t f`	face identifier
`Thread *t`	pointer to a thread
`Thread **pt`	pointer to an array of phase threads

Below is an example of a UDF that utilizes two data access macros (`C_T` and `C_CENTROID`) and two looping macros (`begin..end_c_loop_all` and `thread_loop_c`) to assign initial temperature. Two looping macros are used to set the cell temperature of *each* cell in *every* thread in the computational domain. `begin..end_c_loop_all` is used to loop over all the cells in a cell thread to get the cell centroid and set the cell temperature, and `thread_loop_c` allows this loop to be repeated over all cell threads in the domain.

`C_CENTROID` has three arguments: `xc`, `c`, and `t`. Cell identifier `c` and cell thread pointer `t` are input arguments, and the argument array `xc` (the cell centroid) is output (as an argument) to the solver and used in the UDF in a conditional test.

`C_T` is used to set the cell temperature to the value of 400 or 300, depending on the outcome of the conditional test. It is passed the cell's ID `c` and thread pointer `t` and returns the `real` value of the cell temperature to the Ansys Fluent solver.

Example

```
/*********************************************************************
   UDF for initializing flow field variables
   Example of C_T and C_CENTROID usage.
 *********************************************************************/

#include "udf.h"

DEFINE_INIT(my_init_func,d)
{
 cell_t c;
 Thread *t;
 real xc[ND_ND];
 /* loop over all cell threads in the domain */
```

```
 thread_loop_c(t,d)
 {
  /* loop over all cells */
  begin_c_loop_all(c,t)
  {
   C_CENTROID(xc,c,t);
   if (sqrt(ND_SUM(pow(xc[0] - 0.5,2.),
       pow(xc[1] - 0.5,2.),
       pow(xc[2] - 0.5,2.))) < 0.25)
   C_T(c,t) = 400.;
   else
   C_T(c,t) = 300.;
  }
  end_c_loop_all(c,t)
 }
}
```

3.2. Data Access Macros

The macros presented in this section access Ansys Fluent data that you can use in your UDF. Unless indicated, these macros can be used in UDFs for single-phase and multiphase applications.

3.2.1. Axisymmetric Considerations for Data Access Macros

C-side calculations for axisymmetric models in Ansys Fluent are made on a 1 radian basis. Therefore, when you are utilizing certain data access macros (for example, `F_AREA` or `F_FLUX`) for axisymmetric flows, your UDF will need to multiply the result by 2*PI (utilizing the macro M_PI) to get the desired value.

3.2.2. Node Macros

A mesh in Ansys Fluent is defined by the position of its nodes and how the nodes are connected. The macros listed in Table 3.1: Macros for Node Coordinates Defined in `metric.h` (p. 308) and Table 3.2: Macro for Number of Nodes Defined in `mem.h` (p. 308) can be used to return the `real` Cartesian coordinates of the cell node (at the cell corner) in SI units. The variables are available in both the pressure-based and the density-based solver. Definitions for these macros can be found in `metric.h`. The argument `Node *node` for each of the variables defines a node.

3.2.2.1. Node Position

Table 3.1: Macros for Node Coordinates Defined in `metric.h`

Macro	Argument Types	Returns
`NODE_X(node)`	`Node *node`	`real` x coordinate of node
`NODE_Y(node)`	`Node *node`	`real` y coordinate of node
`NODE_Z(node)`	`Node *node`	`real` z coordinate of node

3.2.2.2. Number of Nodes in a Face (`F_NNODES`)

The macro `F_NNODES` shown in Table 3.2: Macro for Number of Nodes Defined in `mem.h` (p. 308) returns the integer number of nodes associated with a face.

Table 3.2: Macro for Number of Nodes Defined in `mem.h`

Macro	Argument Types	Returns
`F_NNODES(f,t)`	`face_t f,Thread *t`	`int` number of nodes in a face

3.2.3. Cell Macros

The macros listed in Table 3.3: Macro for Cell Centroids Defined in `metric.h` (p. 308) – Table 3.20: Macros for Multiphase Variables Defined in `sg_mphase.h` (p. 322) can be used to return `real` cell variables in SI units. They are identified by the `C_` prefix. These variables are available in the pressure-based and the density-based solver. The quantities that are returned are available only if the corresponding physical model is active. For example, species mass fraction is available only if species transport has been enabled in the **Species Model** dialog box in Ansys Fluent. Definitions for these macros can be found in the referenced header file (for example, `mem.h`).

3.2.3.1. Cell Centroid (`C_CENTROID`)

The macro listed in Table 3.3: Macro for Cell Centroids Defined in `metric.h` (p. 308) can be used to obtain the `real` centroid of a cell. `C_CENTROID` finds the coordinate position of the centroid of the cell `c` and stores the coordinates in the `x` array. Note that the `x` array is always one-dimensional, but it can be `x[2]` or `x[3]` depending on whether you are using the **2D** or **3D** solver.

Table 3.3: Macro for Cell Centroids Defined in `metric.h`

Macro	Argument Types	Outputs
`C_CENTROID(x,c,t)`	`real x[ND_ND],` `cell_t c,Thread * t`	x (cell centroid)

See `DEFINE_INIT` (p. 31) for an example UDF that utilizes `C_CENTROID`.

3.2.3.2. Cell Volume (`C_VOLUME`)

The macro listed in Table 3.4: Macro for Cell Volume Defined in `mem.h` (p. 309) can be used to obtain the `real` cell volume for 2D, 3D, and axisymmetric simulations.

Table 3.4: Macro for Cell Volume Defined in `mem.h`

Macro	Argument Types	Returns
C_VOLUME(c,t)	cell_t c, Thread *t	real cell volume for 2D or 3D, real cell volume/2π for axisymmetric

See `DEFINE_UDS_UNSTEADY` (p. 302) `C_VOLUME`.

3.2.3.3. Number of Faces (`C_NFACES`) and Nodes (`C_NNODES`) in a Cell

The macro `C_NFACES` shown in Table 3.5: Macros for Number of Node and Faces Defined in `mem.h` (p. 309) returns the integer number of faces for a given cell. `C_NNODES`, also shown in Table 3.2: Macro for Number of Nodes Defined in `mem.h` (p. 308), returns the integer number of nodes for a given cell.

Table 3.5: Macros for Number of Node and Faces Defined in `mem.h`

Macro	Argument Types	Returns
C_NNODES(c,t)	cell_t c, Thread *t	int number of nodes in a cell
C_NFACES(c,t)	cell_t c, Thread *t	int number of faces in a cell

3.2.3.4. Cell Face Index (`C_FACE`)

`C_FACE` expands to return the global face index `face_t f` for the given `cell_t c`, `Thread *t`, and local face index number `i`. Specific faces can be accessed via the integer index `i` and all faces can be looped over with `c_face_loop`. The macro is defined in `mem.h`.

> **Note:**
>
> If you are running in parallel, `C_FACE` expands to return the local face index for a compute node.

Table 3.6: Macro for Cell Face Index Defined in `mem.h`

Macro	Argument Types	Returns
C_FACE(c,t,i)	cell_t c, Thread *t, int i	global face index face_t f

3.2.3.5. Cell Face Thread (`C_FACE_THREAD`)

`C_FACE_THREAD` expands to return the `Thread *t` of the `face_t f` that is returned by `C_FACE` (see above). Specific faces can be accessed via the integer index `i` and all faces can be looped over with `c_face_loop`. The macro is defined in `mem.h`.

Table 3.7: Macro for Cell Face Index Defined in `mem.h`

Macro	Argument Types	Returns
`C_FACE_THREAD`	`cell_t c, Thread *t, int i`	`Thread *t` of `face_t f` returned by `C_FACE`.

3.2.3.6. Flow Variable Macros for Cells

You can access flow variables using macros listed in Table 3.8: Macros for Cell Flow Variables Defined in `mem.h` or `sg_mem.h` (p. 310).

Table 3.8: Macros for Cell Flow Variables Defined in `mem.h` or `sg_mem.h`

Macro	Argument Types	Returns
`C_R(c,t)`	`cell_t c, Thread *t`	density
`C_P(c,t)`	`cell_t c, Thread *t`	pressure
`C_U(c,t)`	`cell_t c, Thread *t`	*u* velocity
`C_V(c,t)`	`cell_t c, Thread *t`	*v* velocity
`C_W(c,t)`	`cell_t c, Thread *t`	*w* velocity
`C_T(c,t)`	`cell_t c, Thread *t`	temperature
`C_H(c,t)`	`cell_t c, Thread *t`	enthalpy
`C_K(c,t)`	`cell_t c, Thread *t`	turb. kinetic energy
`C_NUT(c,t)`	`cell_t c, Thread *t`	turbulent viscosity for Spalart-Allmaras
`C_D(c,t)`	`cell_t c, Thread *t`	turb. kinetic energy dissipation rate
`C_O(c,t)`	`cell_t c, Thread *t`	specific dissipation rate
`C_YI(c,t,i)`	`cell_t c, Thread *t, int i` Note: `int i` is species index	species mass fraction
`C_IGNITE (c,t)`	`cell_t c, Thread *t`	ignition mass fraction
`C_PREMIXC_T(c,t)`	`cell_t c, Thread *t`	premixed combustion temperature
`C_STORAGE_R(c,t,nv)`	`cell_t c, Thread *t, real nv`	value of variable `nv`, where `nv` is a generic flow variable. For instance, for temperature, `nv = SV_T`, for pressure, `nv = SV_P`,

Macro	Argument Types	Returns
		for density, `nv = SV_DENSITY`, and so on.

Note:

The `C_YI(c,t,i)` macro is not available with the non/partially premixed models. See Species Fractions Calculations with the Non- and Partially- Premixed Models (p. 311) for Information on calculating the species fractions with the non-premixed and partially premixed models.

Table 3.9: Macro for Cell Porosity in `mem.h`

Macro	Argument Types	Returns
`C_POR(c,t)`	`cell_t c, Thread *t`	porosity of fluid cell
`C_DUAL_ZN_POROS-ITY(c,t)`	`cell_t c, Thread *t`	porosity of fluid at the dual cell zone region in the non-equilibrium thermal model[a]

[a] see γ in Equation 7.15 in the *Fluent User's Guide*.

3.2.3.6.1. Species Fractions Calculations with the Non- and Partially- Premixed Models

When the non-premixed or partially premixed model is enabled, Ansys Fluent uses lookup tables to calculate temperature, density, and species fractions. If you need to access these variables in your UDF, then note that while density and temperature can be obtained through the macros `C_R(c,t)` and `C_T(c,t)`, if you need to access the species fractions, you will need to first retrieve them by calling the species lookup functions `Pdf_Yi(c, t, n)` or `Pdf_XY(c,t,x,y)`. The functions are defined in the header file `pdf_props.h`, which you will need to include in your UDF:

`Pdf_XY` returns the species mole and mass fraction arrays x and y.

Function: `Pdf_XY(cell_t c, Thread *t, real *x, real *y)`

Argument Type	**Description**
`cell_t c`	Cell index.
`Thread *t`	Pointer to thread.
`real *x`	Array of species mole fractions.
`real *y`	Array of species mass fractions.

Function returns

`void`

`Pdf_Yi` returns the mass fraction of species n.

Function: `Pdf_Yi(cell_t c, Thread *t, int n)`

Argument Type	Description
`cell_t c`	Cell index.
`Thread *t`	Pointer to thread.
`int n`	Species index.

Function returns

`real`

The species number in the lookup tables is stored in the integer variable `n_spe_pdf`, which is also included in the header file `pdf_props.h`.

3.2.3.7. Gradient (G) and Reconstruction Gradient (RG) Vector Macros

You can access gradient and reconstruction gradient vectors (and components) for many of the cell variables listed in Table 3.8: Macros for Cell Flow Variables Defined in `mem.h` or `sg_mem.h` (p. 310). Ansys Fluent calculates the gradient of flow in a cell (based on the divergence theory) and stores this value in the variable identified by the suffix `_G`. For example, cell temperature is stored in the variable `C_T`, and the temperature gradient of the cell is stored in `C_T_G`. The gradients stored in variables with the `_G` suffix are non-limited values and if used to reconstruct values within the cell (at faces, for example), may potentially result in values that are higher (or lower) than values in the surrounding cells. Therefore, if your UDF needs to compute face values from cell gradients, you should use the reconstruction gradient (RG) values instead of non-limited gradient (G) values. Reconstruction gradient variables are identified by the suffix `_RG`, and use the limiting method that you have activated in your Ansys Fluent model to limit the cell gradient values.

Gradient (G) Vector Macros

Table 3.10: Macros for Cell Gradients Defined in `mem.h` (p. 313) shows a list of cell gradient vector macros. Note that gradient variables are available *only* when the equation for that variable is being solved. For example, if you are defining a source term for energy, your UDF can access the cell temperature gradient (using `C_T_G`), but it cannot get access to the x-velocity gradient (using `C_U_G`). The reason for this is that the solver continually removes data from memory that it does not need. In order to retain the gradient data (when you want to set up user-defined scalar transport equations, for example), you can prevent the solver from freeing up memory by enabling the following text command: `solve/set/advanced/retain-temporary-solver-mem`. Note that when you do this, all of the gradient data is retained, but the calculation requires more memory to run.

You can access a component of a gradient vector by specifying it as an argument in the gradient vector call (`0` for the *x* component; `1` for *y*; and `2` for *z*). For example,

```
C_T_G(c,t)[0]; /* returns the x-component of the cell temperature gradient vector */
```

Table 3.10: Macros for Cell Gradients Defined in `mem.h`

Macro	Argument Types	Returns
`C_P_G(c,t)`	`cell_t c, Thread *t`	pressure gradient vector
`C_U_G(c,t)`	`cell_t c, Thread *t`	velocity gradient vector
`C_V_G(c,t)`	`cell_t c, Thread *t`	velocity gradient vector
`C_W_G(c,t)`	`cell_t c, Thread *t`	velocity gradient vector
`C_T_G(c,t)`	`cell_t c, Thread *t`	temperature gradient vector
`C_H_G(c,t)`	`cell_t c, Thread *t`	enthalpy gradient vector
`C_NUT_G(c,t)`	`cell_t c, Thread *t`	turbulent viscosity for Spalart- Allmaras gradient vector
`C_K_G(c,t)`	`cell_t c, Thread *t`	turbulent kinetic energy gradient vector
`C_D_G(c,t)`	`cell_t c, Thread *t`	turbulent kinetic energy dissipation rate gradient vector
`C_O_G(c,t)`	`cell_t c, Thread *t`	specific dissipation rate gradient vector
`C_YI_G(c,t,i)`	`cell_t c, Thread *t, int i` Note: `int i` is species index	species mass fraction gradient vector

Important:

Note that you can access vector components of each of the variables listed in Table 3.10: Macros for Cell Gradients Defined in `mem.h` (p. 313) by using the integer index `[i]` for each macro listed in Table 3.10: Macros for Cell Gradients Defined in `mem.h` (p. 313). For example, `C_T_G(c,t)[i]` will access a component of the temperature gradient vector.

Important:

`C_P_G` can be used only in the pressure-based solver.

Important:

`C_YI_G` can be used only in the density-based solver. To use this in the pressure-based solver, you will need to set the rpvar `'species/save-gradients?` to `#t`.

As stated previously, the availability of gradient variables is affected by your solver selection, which models are turned on, the setting for the spatial discretization, and whether the temporary solver memory is retained. To make it easy for you to verify what gradient variables are available for your particular case and data files, the following UDF (named showgrad.c) is provided. Simply compile this UDF, run your solution, and then hook the UDF using the **Execute on Demand** dialog box (as described in Hooking DEFINE_ON_DEMAND UDFs (p. 431)). The available gradient variables will be displayed in the console.

Important:

Note that the showgrad.c UDF is useful only for single-phase models.

```
/*
* ON Demand User-Defined Functions to check                     *
* on the availability of Reconstruction Gradient and Gradients  *
* for a given Solver and Solver settings:                       *
*                                                               *
* Availability of Gradients & Reconstruction Gradients depends on: *
* 1) the selected Solver (density based or pressure based)      *
* 2) the selected Model                                         *
* 3) the order of discretizations                               *
* 4) whether the temporary solver memory is being retained (to keep *
*    temporary memory, enable the following text command:       *
*    solve/set/advanced/retain-temporary-solver-mem.            *
*                                                               *
*                                                               *
* How to use showgrad:                                          *
*                                                               *
* - Read in your case & data file.                              *
* - Compile showgrad.c UDF.                                     *
* - Load library libudf.                                        *
* - Attach the showgrad UDF in the Execute on Demand dialog box. *
* - Run your solution.                                          *
* - Click the Execute button in the Execute on Demand dialog box. *
*                                                               *
* A list of available Grads and Recon Grads will be displayed in the *
* console.                                                      *
*                                                               *
* 2004 Laith Zori                                               *
*/ #include "udf.h"

DEFINE_ON_DEMAND(showgrad)
{
   Domain *domain;
   Thread *t;  domain=Get_Domain(1);
   if (! Data_Valid_P()) return;
   Message0(" >>> entering show-grad: \n ");
   thread_loop_c(t, domain)
     {
      Material *m = THREAD_MATERIAL(t);
      int nspe = MIXTURE_NSPECIES(m);
      int nspm = nspe-1;
      Message0(":::::\n ");
      Message0(":::::  Reconstruction Gradients :::: \n ");
      Message0(":::::\n ");
      if (NNULLP(THREAD_STORAGE(t, SV_P_RG)))
        {
         Message0("....show-grad:Reconstruction Gradient of P is available \n ");
        }
      if (NNULLP(THREAD_STORAGE(t, SV_U_RG)))
        {
         Message0("....show-grad:Reconstruction Gradient of U is available \n ");
        }
      if (NNULLP(THREAD_STORAGE(t, SV_V_RG)))
        {
```

```
            Message0("....show-grad:Reconstruction Gradient of V is available \n ");
          }
      if (NNULLP(THREAD_STORAGE(t, SV_W_RG)))
        {
         Message0("....show-grad:Reconstruction Gradient of W is available \n ");
        }
      if (NNULLP(THREAD_STORAGE(t, SV_T_RG)))
        {
         Message0("....show-grad:Reconstruction Gradient of T is available \n ");
        }
      if (NNULLP(THREAD_STORAGE(t, SV_H_RG)))
        {
         Message0("....show-grad:Reconstruction Gradient of H is available \n ");
        }
      if (NNULLP(THREAD_STORAGE(t, SV_K_RG)))
        {
         Message0("....show-grad:Reconstruction Gradient of K is available \n ");
        }
      if (NNULLP(THREAD_STORAGE(t, SV_D_RG)))
        {
         Message0("....show-grad:Reconstruction Gradient of D is available \n ");
        }
      if (NNULLP(THREAD_STORAGE(t, SV_O_RG)))
        {
         Message0("....show-grad:Reconstruction Gradient of O is available \n ");
        }
      if (NNULLP(THREAD_STORAGE(t, SV_NUT_RG)))
        {
         Message0("....show-grad:Reconstruction Gradient of NUT is available \n ");
        }

      if (nspe && NNULLP(THREAD_STORAGE(t, SV_Y_RG)))
        {
          Message0("....show-grad:Reconstruction Gradient of Species is available \n ");
        }

   /****************************************************************/
   /****************************************************************/
   /****************************************************************/
   /****************************************************************/
   Message0(":::::\n ");
   Message0(":::::   Gradients :::: \n ");
   Message0(":::::\n ");
   if (NNULLP(THREAD_STORAGE(t, SV_P_G)))
     {
      Message0("....show-grad:Gradient of P is available \n ");
     }
   if (NNULLP(THREAD_STORAGE(t, SV_U_G)))
     {
      Message0("....show-grad:Gradient of U is available \n ");
     }
   if (NNULLP(THREAD_STORAGE(t, SV_V_G)))
     {
      Message0("....show-grad:Gradient of V is available \n ");
     }
 if (NNULLP(THREAD_STORAGE(t, SV_W_G)))
     {
      Message0("....show-grad:Gradient of W is available \n ");
     }
   if (NNULLP(THREAD_STORAGE(t, SV_T_G)))
     {
      Message0("....show-grad:Gradient of T is available \n ");
     }
   if (NNULLP(THREAD_STORAGE(t, SV_H_G)))
     {
      Message0("....show-grad:Gradient of H is available \n ");
     }
   if (NNULLP(THREAD_STORAGE(t, SV_K_G)))
     {
      Message0("....show-grad:Gradient of K is available \n ");
     }
```

```
    if (NNULLP(THREAD_STORAGE(t, SV_D_G)))
      {
       Message0("....show-grad:Gradient of D is available \n ");
      }
    if (NNULLP(THREAD_STORAGE(t, SV_O_G)))
      {
       Message0("....show-grad:Gradient of O is available \n ");
      }
    if (NNULLP(THREAD_STORAGE(t, SV_NUT_G)))
      {
       Message0("....show-grad:Gradient of NUT is available \n ");
      }

    if (nspe && NNULLP(THREAD_STORAGE(t, SV_Y_G)))
      {
        Message0("....show-grad:Gradient of Species is available \n ");
      }
   }
 }
```

Reconstruction Gradient (RG) Vector Macros

Table 3.11: Macros for Cell Reconstruction Gradients (RG) Defined in `mem.h` (p. 316) shows a list of cell reconstruction gradient vector macros. Like gradient variables, RG variables are available only when the equation for that variable is being solved. As in the case of gradient variables, you can retain all of the reconstruction gradient data by enabling the following text command: `solve/set/advanced/retain-temporary-solver-mem`. Note that when you do this, the reconstruction gradient data is retained, but the calculation requires more memory to run.

You can access a component of a reconstruction gradient vector by specifying it as an argument in the reconstruction gradient vector call (`0` for the *x* component; `1` for *y*; and `2` for *z*). For example,

```
C_T_RG(c,t)[0]; /* returns the x-component of the cell temperature
                   reconstruction gradient vector */
```

Table 3.11: Macros for Cell Reconstruction Gradients (RG) Defined in `mem.h`

Macro	Argument Types	Returns
C_R_RG(c,t)	cell_t c, Thread *t	density RG vector
C_P_RG(c,t)	cell_t c, Thread *t	pressure RG vector
C_U_RG(c,t)	cell_t c, Thread *t	velocity RG vector
C_V_RG(c,t)	cell_t c, Thread *t	velocity RG vector
C_W_RG(c,t)	cell_t c, Thread *t	velocity RG vector
C_T_RG(c,t)	cell_t c, Thread *t	temperature RG vector
C_H_RG(c,t)	cell_t c, Thread *t	enthalpy RG vector
C_NUT_RG(c,t)	cell_t c, Thread *t	turbulent viscosity for Spalart-Allmaras RG vector
C_K_RG(c,t)	cell_t c, Thread *t	turbulent kinetic energy RG vector
C_D_RG(c,t)	cell_t c, Thread *t	turbulent kinetic energy dissipation rate RG vector

Macro	Argument Types	Returns
`C_YI_RG(c,t,i)`	`cell_t c,Thread *t, int i` Note: `int i` is species index	species mass fraction RG vector

Important:

Note that you can access vector components by using the integer index `[i]` for each macro listed in Table 3.11: Macros for Cell Reconstruction Gradients (RG) Defined in `mem.h` (p. 316). For example, `C_T_RG(c,t)[i]` will access a component of the temperature reconstruction gradient vector.

Important:

`C_P_RG` can be used in the pressure-based solver only when the second order discretization Scheme for pressure is specified.

Important:

`C_YI_RG` can be used only in the density-based solver.

As stated previously, the availability of reconstruction gradient variables is affected by your solver selection, which models are turned on, the setting for the spatial discretization, and whether the temporary solver memory is freed. To make it easy for you to verify which reconstruction gradient variables are available for your particular case and data files, a UDF (named `showgrad.c`) has been provided that will display the available gradients in the console. See the previous section for details.

3.2.3.8. Previous Time Step Macros

The `_M1` suffix can be applied to some of the cell variable macros in Table 3.8: Macros for Cell Flow Variables Defined in `mem.h` or `sg_mem.h` (p. 310) to allow access to the value of the variable at the previous time step (that is, $t - \Delta t$). These data may be useful in unsteady simulations. For example,

```
C_T_M1(c,t);
```

returns the value of the cell temperature at the previous time step. Previous time step macros are shown in Table 3.12: Macros for Cell Time Level 1 Defined in `mem.h` (p. 318).

Important:

Note that data from `C_T_M1` is available *only* if user-defined scalars are defined. It can also be used with adaptive time stepping.

Table 3.12: Macros for Cell Time Level 1 Defined in `mem.h`

Macro	Argument Types	Returns
`C_R_M1(c,t)`	`cell_t c, Thread *t`	density, previous time step
`C_U_M1(c,t)`	`cell_t c, Thread *t`	velocity, previous time step
`C_V_M1(c,t)`	`cell_t c, Thread *t`	velocity, previous time step
`C_W_M1(c,t)`	`cell_t c, Thread *t`	velocity, previous time step
`C_T_M1(c,t)`	`cell_t c, Thread *t`	temperature, previous time step
`C_YI_M1(c,t,i)`	`cell_t c, Thread *t, int i` Note: `int i` is species index	species mass fraction, previous time step

See `DEFINE_UDS_UNSTEADY` (p. 302) for an example UDF that utilizes `C_R_M1`.

The `M2` suffix can be applied to some of the cell variable macros in Table 3.12: Macros for Cell Time Level 1 Defined in `mem.h` (p. 318) to allow access to the value of the variable at the time step before the previous one (that is, $t-2\Delta t$). These data may be useful in unsteady simulations. For example,

```
C_T_M2(c,t);
```

returns the value of the cell temperature at the time step before the previous one (referred to as second previous time step). Two previous time step macros are shown in Table 3.13: Macros for Cell Time Level 2 Defined in `mem.h` (p. 318).

Important:

Note that data from `C_T_M2` is available *only* if user-defined scalars are defined. It can also be used with adaptive time stepping.

Table 3.13: Macros for Cell Time Level 2 Defined in `mem.h`

Macro	Argument Types	Returns
`C_R_M2(c,t)`	`cell_t c, Thread *t`	density, second previous time step
`C_U_M2(c,t)`	`cell_t c, Thread *t`	velocity, second previous time step

Macro	Argument Types	Returns
C_V_M2(c,t)	cell_t c,Thread *t	velocity, second previous time step
C_W_M2(c,t)	cell_t c,Thread *t	velocity, second previous time step
C_T_M2(c,t)	cell_t c,Thread *t	temperature, second previous time step
C_YI_M2(c,t,i)	cell_t c,Thread *t, int i	species mass fraction, second previous time step

3.2.3.9. Derivative Macros

The macros listed in Table 3.14: Macros for Cell Velocity Derivatives Defined in mem.h (p. 319) can be used to return real velocity derivative variables in SI units. The variables are available in both the pressure-based and the density-based solver. Definitions for these macros can be found in the mem.h header file.

Table 3.14: Macros for Cell Velocity Derivatives Defined in mem.h

Macro	Argument Types	Returns
C_STRAIN_RATE_MAG(c,t)	cell_t c,Thread *t	strain rate magnitude
C_DUDX(c,t)	cell_t c,Thread *t	velocity derivative
C_DUDY(c,t)	cell_t c,Thread *t	velocity derivative
C_DUDZ(c,t)	cell_t c,Thread *t	velocity derivative
C_DVDX(c,t)	cell_t c,Thread *t	velocity derivative
C_DVDY(c,t)	cell_t c,Thread *t	velocity derivative
C_DVDZ(c,t)	cell_t c,Thread *t	velocity derivative
C_DWDX(c,t)	cell_t c,Thread *t	velocity derivative
C_DWDY(c,t)	cell_t c,Thread *t	velocity derivative
C_DWDZ(c,t)	cell_t c,Thread *t	velocity derivative

3.2.3.10. Material Property Macros

The macros listed in Table 3.15: Macros for Diffusion Coefficients Defined in mem.h (p. 319) – Table 3.17: Additional Material Property Macros Defined in sg_mem.h (p. 320) can be used to return real material property variables in SI units. The variables are available in both the pressure-based and the density-based solver. Argument real prt is the turbulent Prandtl number. Definitions for material property macros can be found in the referenced header file (for example, mem.h).

Table 3.15: Macros for Diffusion Coefficients Defined in mem.h

Macro	Argument Types	Returns
C_MU_L(c,t)	cell_t c,Thread *t	laminar viscosity
C_MU_T(c,t)	cell_t c,Thread *t	turbulent viscosity[a]
C_MU_EFF(c,t)	cell_t c,Thread *t	effective viscosity

Macro	Argument Types	Returns
C_K_L(c,t)	cell_t c, Thread *t	thermal conductivity
C_K_T(c,t,prt)	cell_t c, Thread *t, real prt	turbulent thermal conductivity
C_K_EFF(c,t,prt)	cell_t c, Thread *t, real prt	effective thermal conductivity
C_DIFF_L(c,t,i,j)	cell_t c, Thread *t, int i, int j	laminar species diffusivity
C_DIFF_EFF(c,t,i)	cell_t c, Thread *t, int i	effective species diffusivity

[a] In an Embedded LES case with SAS or DES for the global turbulence model, the global turbulence model is solved even inside the LES zone, although it does not affect the velocity equations or any other model there. (This allows the global turbulence model in a downstream RANS zone to have proper inflow turbulence conditions.) Inside the LES zone, the turbulent eddy viscosity of the "muted" global SAS or DES model can be accessed through the C_MU_T_LES_ZONE(c,t) macro. (All other global turbulence models are completely frozen in all LES zones; in such cases, only the LES sub-grid scale model's eddy viscosity is available through C_MU_T(c,t) in the LES zones, as is always true for all LES zones and all pure LES cases.)

Table 3.16: Macros for Thermodynamic Properties Defined in `mem.h`

Name (Arguments)	Argument Types	Returns
C_CP(c,t)	cell_t c, Thread *t	specific heat
C_RGAS(c,t)	cell_t c, Thread *t	universal gas constant/molecular weight
C_NUT(c,t)	cell_t c, Thread *t	turbulent viscosity for Spalart-Allmaras

Table 3.17: Additional Material Property Macros Defined in `sg_mem.h`

Macro	Argument Types	Returns
C_FMEAN(c,t)	cell_t c, Thread *t	primary mean mixture fraction
C_FMEAN2(c,t)	cell_t c, Thread *t	secondary mean mixture fraction
C_FVAR(c,t)	cell_t c, Thread *t	primary mixture fraction variance
C_FVAR2(c,t)	cell_t c, Thread *t	secondary mixture fraction variance
C_PREMIXC(c,t)	cell_t c, Thread *t	reaction progress variable
C_LAM_FLAME_SPEED(c,t)	cell_t c, Thread *t	laminar flame speed
C_SCAT_COEFF(c,t)	cell_t c, Thread *t	scattering coefficient
C_ABS_COEFF(c,t)	cell_t c, Thread *t	absorption coefficient
C_CRITICAL_STRAIN_RATE(c,t)	cell_t c, Thread *t	critical strain rate
C_LIQF(c,t)	cell_t c, Thread *t	liquid fraction in a cell

Macro	Argument Types	Returns
C_POLLUT(c,t,i)	cell_t c, Thread *t, int i	ith pollutant species mass fraction (see table below)

Important:

C_LIQF is available only in fluid cells and only if solidification is turned ON.

Table 3.18: Table of Definitions for Argument i of the Pollutant Species Mass Fraction Function C_POLLUT

i	Definitions
0	Mass Fraction of NO
1	Mass Fraction of HCN
2	Mass Fraction of NH3
3	Mass Fraction of N2O
4	Soot Mass Fraction
5	Normalized Radical Nuclei

Note:

Concentration in particles $\times 10^{-15}$/kg. For mass fraction concentrations in the table above, see Equation 9.121 in the *Fluent Theory Guide* for the defining equation.

3.2.3.11. Reynolds Stress Model Macros

The macros listed in Table 3.19: Macros for Reynolds Stress Model Variables Defined in sg_mem.h (p. 321) can be used to return real variables for the Reynolds stress turbulence model in SI units. The variables are available in both the pressure-based and the density-based solver. Definitions for these macros can be found in the metric.h header file.

Table 3.19: Macros for Reynolds Stress Model Variables Defined in sg_mem.h

Macro	Argument Types	Returns
C_RUU(c,t)	cell_t c, Thread *t	*uu* Reynolds stress
C_RVV(c,t)	cell_t c, Thread *t	*vv* Reynolds stress
C_RWW(c,t)	cell_t c, Thread *t	*ww* Reynolds stress
C_RUV(c,t)	cell_t c, Thread *t	*uv* Reynolds stress
C_RVW(c,t)	cell_t c, Thread *t	*vw* Reynolds stress
C_RUW(c,t)	cell_t c, Thread *t	*uw* Reynolds stress

3.2.3.12. VOF Multiphase Model Macro

The macro C_VOF can be used to return real variables associated with the VOF multiphase model in SI units. The variables are available in both the pressure-based and the density-based

solver, with the exception of the VOF variable, which is available only for the pressure-based solver. Definitions for these macros can be found in `sg_mphase.h`, which is included in `udf.h`.

Table 3.20: Macros for Multiphase Variables Defined in `sg_mphase.h`

Macro	Argument Types	Returns
C_VOF(c,t)	cell_t c, Thread *t (has to be a phase thread)	volume fraction for the phase corresponding to phase thread t.

3.2.4. Face Macros

The macros listed in Table 3.21: Macro for Face Centroids Defined in `metric.h` (p. 322) – Table 3.24: Macros for Interior and Boundary Face Flow Variables Defined in `mem.h` (p. 324) can be used to return `real` face variables in SI units. They are identified by the `F_` prefix. Note that these variables are available *only* in the pressure-based solver. In addition, quantities that are returned are available only if the corresponding physical model is active. For example, species mass fraction is available only if species transport has been enabled in the **Species Model** dialog box in Ansys Fluent. Definitions for these macros can be found in the referenced header files (for example, `mem.h`).

3.2.4.1. Face Centroid (`F_CENTROID`)

The macro listed in Table 3.21: Macro for Face Centroids Defined in `metric.h` (p. 322) can be used to obtain the `real` centroid of a face. `F_CENTROID` finds the coordinate position of the centroid of the face `f` and stores the coordinates in the `x` array. Note that the `x` array is always one-dimensional, but it can be `x[2]` or `x[3]` depending on whether you are using the **2D** or **3D** solver.

Table 3.21: Macro for Face Centroids Defined in `metric.h`

Macro	Argument Types	Outputs
F_CENTROID(x,f,t)	real x[ND_ND], face_t f, Thread *t	x (face centroid)

The `ND_ND` macro returns 2 or 3 in 2D and 3D cases, respectively, as defined in The `ND` Macros (p. 377). `DEFINE_PROFILE` (p. 108) contains an example of `F_CENTROID` usage.

3.2.4.2. Face Area Vector (`F_AREA`)

`F_AREA` can be used to return the `real` face area vector (or 'face area normal') of a given face `f` in a face thread `t`. See `DEFINE_UDS_FLUX` (p. 299) for an example UDF that utilizes `F_AREA`.
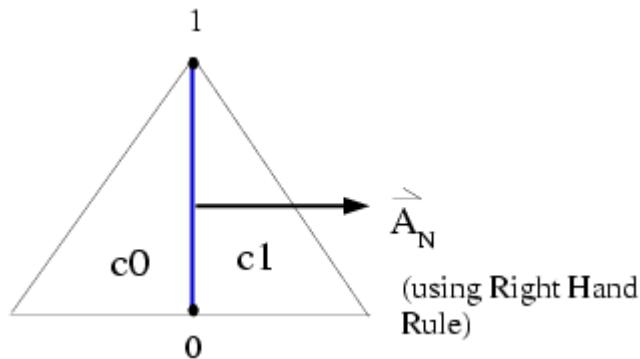
Table 3.22: Macro for Face Area Vector Defined in `metric.h`

Macro	Argument Types	Outputs
F_AREA(A,f,t)	A[ND_ND], face_t f, Thread *t	A (area vector)

By convention in Ansys Fluent, boundary face area normals always point out of the domain. Ansys Fluent determines the direction of the face area normals for interior faces by applying the right

hand rule to the nodes on a face, in order of increasing node number. This is shown in Figure 3.1: Ansys Fluent Determination of Face Area Normal Direction: 2D Face (p. 323).

Figure 3.1: Ansys Fluent Determination of Face Area Normal Direction: 2D Face

Ansys Fluent assigns adjacent cells to an interior face (`c0` and `c1`) according to the following convention: the cell *out* of which a face area normal is pointing is designated as cell `C0`, while the cell *in* to which a face area normal is pointing is cell `c1` (Figure 3.1: Ansys Fluent Determination of Face Area Normal Direction: 2D Face (p. 323)). In other words, face area normals always point from cell `c0` to cell `c1`.

3.2.4.3. Flow Variable Macros for Boundary Faces

The macros listed in Table 3.23: Macros for Boundary Face Flow Variables Defined in `mem.h` (p. 323) access flow variables at a boundary face.

Table 3.23: Macros for Boundary Face Flow Variables Defined in `mem.h`

Macro	Argument Types	Returns
`F_U(f,t)`	`face_t f, Thread *t,`	*u* velocity
`F_V(f,t)`	`face_t f, Thread *t,`	*v* velocity
`F_W(f,t)`	`face_t f, Thread *t,`	*w* velocity
`F_T(f,t)`	`face_t f, Thread *t,`	temperature
`F_H(f,t)`	`face_t f, Thread *t,`	enthalpy
`F_K(f t)`	`face_t f, Thread *t,`	turbulent kinetic energy
`F_D(f,t)`	`face_t f, Thread *t,`	turbulent kinetic energy dissipation rate
`F_YI(f,t,i)`	`face_t f, Thread *t, int i`	species mass fraction

See `DEFINE_UDS_FLUX` (p. 299) for an example UDF that utilizes some of these macros.

3.2.4.4. Flow Variable Macros at Interior and Boundary Faces

The macros listed in Table 3.24: Macros for Interior and Boundary Face Flow Variables Defined in `mem.h` (p. 324) access flow variables at interior faces and boundary faces.

Table 3.24: Macros for Interior and Boundary Face Flow Variables Defined in `mem.h`

Macro	Argument Types	Returns
F_P(f,t)	face_t f, Thread *t,	pressure
F_FLUX(f,t)	face_t f, Thread *t	mass flow rate through a face

`F_FLUX` can be used to return the `real` scalar mass flow rate through a given face `f` in a face thread `t`. The sign of `F_FLUX` that is computed by the Ansys Fluent solver is positive if the flow direction is the same as the face area normal direction (as determined by `F_AREA` - see Face Area Vector (`F_AREA`) (p. 322)), and is negative if the flow direction and the face area normal directions are opposite. In other words, the flux is positive if the flow is *out* of the domain, and is negative if the flow is *in* to the domain.

Note that the sign of the flux that is computed by the solver is opposite to that which is reported in the Ansys Fluent GUI (for example, the **Flux Reports** dialog box).

> **Important:**
>
> `F_P(f,t)` is not available in the density-based solver.
>
> In the density-based solver, `F_FLUX(f,t)` will only return a value if one or more scalar equations (for example, turbulence quantities) are being solved that require the mass flux of a face to be stored by the solver.

3.2.5. Connectivity Macros

Ansys Fluent provides macros that allow the vectors connecting cell centroids and the vectors connecting cell and face centroids to be readily defined. These macros return information that is helpful in evaluating face values of scalars which are generally not stored, as well as the diffusive flux of scalars across cell boundaries. The geometry and gradients involved with these macros are summarized in Figure 3.2: Adjacent Cells c0 and c1 with Vector and Gradient Definitions (p. 325).

To better understand the parameters that are returned by these macros, it is best to consider how the aforementioned calculations are evaluated. Assuming that the gradient of a scalar is available, the face value of a scalar, ϕ, can be approximated by

$$\phi_f = \phi_0 + \nabla\phi \cdot \vec{dr} \tag{3.1}$$

where $\vec{dr}$ is the vector that connects the cell centroid with the face centroid. The gradient in this case is evaluated at the cell centroid where ϕ_0 is also stored.

The diffusive flux, D_f, across a face, f, of a scalar ϕ is given by,
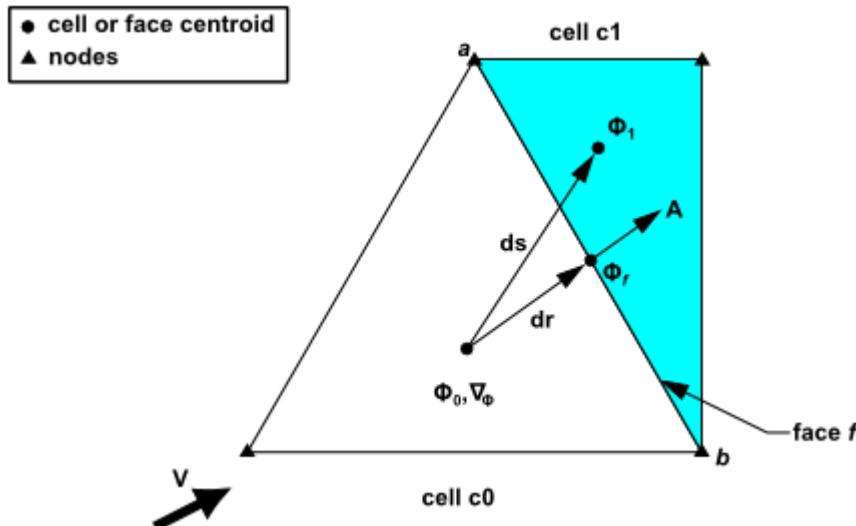
$$D_f = \Gamma_f \nabla\phi \cdot \vec{A} \tag{3.2}$$

where Γ_f is the diffusion coefficient at the face. In Ansys Fluent's unstructured solver, the gradient along the face normal direction may be approximated by evaluating gradients along the directions that connect cell centroids and along a direction confined within the plane of the face. Given this, D_f maybe approximated as,

$$D_f = \Gamma_f \frac{\left(\phi_1 - \phi_0\right)}{ds} \frac{\vec{A} \cdot \vec{A}}{\vec{A} \cdot \vec{e}_s} + \Gamma_f \left(\overline{\nabla \phi} \cdot \vec{A} - \overline{\nabla \phi} \cdot \vec{e}_s \frac{\vec{A} \cdot \vec{A}}{\vec{A} \cdot \vec{e}_s} \right) \tag{3.3}$$

where the first term on the right hand side represents the primary gradient directed along the vector $\vec{e}_s$ and the second term represents the 'cross' diffusion term. In this equation, A is the area normal vector of face f directed from cell c0 to c1, ds is the distance between the cell centroids, and $\vec{e}_s$ is the unit normal vector in this direction. $\overline{\nabla \phi}$ is the average of the gradients at the two adjacent cells. (For boundary faces, the variable is the gradient of the c0 cell.) This is shown in Figure 3.2: Adjacent Cells c0 and c1 with Vector and Gradient Definitions (p. 325).

Figure 3.2: Adjacent Cells c0 and c1 with Vector and Gradient Definitions

3.2.5.1. Adjacent Cell Index (`F_C0`, `F_C1`)

The cells on either side of a face may or may not belong to the same cell thread. Referring to Figure 3.2: Adjacent Cells c0 and c1 with Vector and Gradient Definitions (p. 325), if a face is on the boundary of a domain, then only c0 exists. (c1 is undefined for an external face). Alternatively, if the face is in the interior of the domain, then both c0 and c1 exist.

There are two macros, F_C0(f,t) and F_C1(f,t), that can be used to identify cells that are adjacent to a given face thread t. F_C0 expands to a function that returns the index of a face's neighboring c0 cell (Figure 3.2: Adjacent Cells c0 and c1 with Vector and Gradient Definitions (p. 325)), while F_C1 returns the cell index for c1 (Figure 3.2: Adjacent Cells c0 and c1 with Vector and Gradient Definitions (p. 325)), if it exists.

Table 3.25: Adjacent Cell Index Macros Defined in `mem.h`

Macro	Argument Types	Returns
F_C0(f,t)	face_t f, Thread *t	cell_t c for cell c0

Macro	Argument Types	Returns
`F_C1(f,t)`	`face_t f, Thread *t`	`cell_t c` for cell c1

See `DEFINE_UDS_FLUX` (p. 299) for an example UDF that utilizes `F_C0`.

3.2.5.2. Adjacent Cell Thread (`THREAD_T0`, `THREAD_T1`)

The cells on either side of a face may or may not belong to the same cell thread. Referring to Figure 3.2: Adjacent Cells c0 and c1 with Vector and Gradient Definitions (p. 325), if a face is on the boundary of a domain, then only `c0` exists. (`c1` is undefined for an external face). Alternatively, if the face is in the interior of the domain, then both `c0` and `c1` exist.

There are two macros, `THREAD_T0(t)` and `THREAD_T1(t)`, that can be used to identify cell threads that are adjacent to a given face `f` in a face thread `t`. `THREAD_T0` expands to a function that returns the cell thread of a given face's adjacent cell `c0`, and `THREAD_T1` returns the cell thread for `c1` (if it exists).

Table 3.26: Adjacent Cell Thread Macros Defined in `mem.h`

Macro	Argument Types	Returns
`THREAD_T0(t)`	`Thread *t`	cell thread pointer for cell c0
`THREAD_T1(t)`	`Thread *t`	cell thread pointer for cell c1

3.2.5.3. Interior Face Geometry (`INTERIOR_FACE_GEOMETRY`)

`INTERIOR_FACE_GEOMETRY(f,t,A,ds,es,A_by_es,dr0,dr1)` expands to a function that outputs the following variables to the solver, for a given face `f`, on face thread `t`. The macro is defined in the `sg.h` header file which is *not* included in `udf.h`. You will need to include this file in your UDF using the `#include` directive.

`real A[ND_ND]`	the area normal vector
`real ds`	distance between the cell centroids
`real es[ND_ND]`	the unit normal vector in the direction from cell c0 to c1
`real A_by_es`	the value $\frac{\vec{A}\cdot\vec{A}}{\vec{A}\cdot\vec{e}_s}$
`real dr0[ND_ND]`	vector that connects the centroid of cell `c0` to the face centroid
`real dr1[ND_ND]`	the vector that connects the centroid of cell `c1` to the face centroid

Note that `INTERIOR_FACE_GEOMETRY` can be called to retrieve some of the terms needed to evaluate Equation 3.1 (p. 324) and Equation 3.3 (p. 325).

3.2.5.4. Boundary Face Geometry (`BOUNDARY_FACE_GEOMETRY`)

`BOUNDARY_FACE_GEOMETRY(f,t,A,ds,es,A_by_es,dr0)` expands to a function that outputs the following variables to the solver, for a given face `f`, on face thread `t`. It is defined in the `sg.h` header file which is *not* included in `udf.h`. You will need to include this file in your UDF using the `#include` directive.

`BOUNDARY_FACE_GEOMETRY` can be called to retrieve some of the terms needed to evaluate Equation 3.1 (p. 324) and Equation 3.3 (p. 325).

`real A[ND_ND]`	area normal vector
`real ds`	distance between the cell centroid and the face centroid
`real es[ND_ND]`	unit normal vector in the direction from centroid of cell c0 to the face centroid
`real A_by_es`	value $\dfrac{\vec{A} \cdot \vec{A}}{\vec{A} \cdot \vec{e}_s}$
`real dr0[ND_ND]`	vector that connects the centroid of cell c0 to the face centroid

3.2.5.5. Boundary Face Thread (`BOUNDARY_FACE_THREAD`)

`BOUNDARY_FACE_THREAD_P(t)` expands to a function that returns `TRUE` if `Thread *t` is a boundary face thread. The macro is defined in `threads.h` which is included in `udf.h`. See DEFINE_UDS_FLUX (p. 299) for an example UDF that utilizes `BOUNDARY_FACE_THREAD_P`.

3.2.5.6. Boundary Secondary Gradient Source (`BOUNDARY_SECONDARY_GRADIENT_SOURCE`)

`BOUNDARY_SECONDARY_GRADIENT_SOURCE(source,n,dphi,dx,A_by_es,k)` expands to a function that outputs the following variables to the solver, for a given face and face thread. It is defined in the `sg.h` header file which is *not* included in `udf.h`. You will need to include this file in your UDF using the `#include` directive.

> **Important:**
>
> The use of `BOUNDARY_SECONDARY_GRADIENT_SOURCE` first requires that cell geometry information be defined, which can be readily obtained by the use of the `BOUNDARY_FACE_GEOMETRY` macro (described previously in this section). See Implementing Ansys Fluent's P-1 Radiation Model Using User-Defined Scalars (p. 644) for an example.

`BOUNDARY_SECONDARY_GRADIENT_SOURCE` can be called to retrieve some of the terms needed to evaluate Equation 3.3 (p. 325).

`real source`	the cross diffusion term of the diffusive flux (that is, the second term on the right side of Equation 3.3 (p. 325))

`Svar n`	the `Svar` enumeration value (for example, `SV_...._G` or `SV_UDSI_G(...)` for the (spatial) gradient of the solution variable `SV_...` or `SV_UDSI_(...)` being worked on
`real dphi[ND_ND]`	a dummy scratch variable array that stores the facial gradient value during the computation
`real dx[ND_ND]`	the unit normal vector in the direction from centroid of cell c0 to the face centroid
`real A_by_es`	the value $\frac{\vec{A}\cdot\vec{A}}{\vec{A}\cdot\vec{e_s}}$
`real k`	the diffusion coefficient at the face (Γ_f in [Equation 3.3 (p. 325)](#))

Important:

Note that the gradient field variable addressed by the `Svar` value n is not always allocated, and so your UDF must verify its status (using the `NULLP` or `NNULLP` function, as described in [NULLP & NNULLP (p. 391)](#)) and assign a value as necessary. See [Implementing Ansys Fluent's P-1 Radiation Model Using User-Defined Scalars (p. 644)](#) for an example.

3.2.6. Special Macros

The macros listed in this section are special macros that are used often in UDFs.

- `Lookup_Thread`

- `THREAD_ID`

- `Get_Domain`

- `F_PROFILE`

- `THREAD_SHADOW`

3.2.6.1. Thread Pointer for Zone ID (`Lookup_Thread`)

You can use `Lookup_Thread` when you want to retrieve the pointer t to the thread that is associated with a given integer zone ID number for a boundary zone. The `zone_ID` that is passed to the macro is the zone number that Ansys Fluent assigns to the boundary and displays in the boundary condition dialog box (for example, `Fluid`). Note that this macro does the inverse of `THREAD_ID` (see below).

There are two arguments to `Lookup_Thread`. `domain` is passed by Ansys Fluent and is the pointer to the domain structure. You supply the integer value of `zone_ID`.

For example, the code

```
int zone_ID = 2;
Thread *thread_name = Lookup_Thread(domain,zone_ID);
```

passes a zone ID of 2 to `Lookup_Thread`. A zone ID of 2 may, for example, correspond to a wall zone in your case.

Now suppose that your UDF needs to operate on a particular thread in a domain (instead of looping over all threads), and the `DEFINE` macro you are using to define your UDF does not have the `thread` pointer passed to it from the solver (for example, `DEFINE_ADJUST`). You can use `Lookup_Thread` in your UDF to get the desired thread pointer. This is a two-step process.

First, you will need to get the integer ID of the zone by visiting the boundary condition dialog box (for example, **Fluid**) and noting the zone ID. You can also obtain the value of the Zone ID from the solver using `RP_Get_Integer`. Note that in order to use `RP_Get_Integer`, you will have to define the zone ID variable first on the Scheme side using `rp-var-define` or `make-new-rpvar` (see for details.)

Next, you supply the `zone_ID` as an argument to `Lookup_Thread` either as a hard-coded integer (for example, 1, 2) or as the variable assigned from `RP_Get_Integer`. `Lookup_Thread` returns the pointer to the `thread` that is associated with the given zone ID. You can then assign the thread pointer to a `thread_name` and use it in your UDF.

Important:

Note that when `Lookup_Thread` is utilized in a multiphase flow problem, the domain pointer that is passed to the function depends on the UDF that it is contained within. For example, if `Lookup_Thread` is used in an adjust function (`DEFINE_ADJUST`), then the mixture domain is passed and the thread pointer returned is the mixture-level thread.

Example

Below is a UDF that uses `Lookup_Thread`. In this example, the pointer to the thread for a given `zone_ID` is retrieved by `Lookup_Thread` and is assigned to `thread`. The `thread` pointer is then used in `begin_f_loop` to loop over all faces in the given thread, and in `F_CENTROID` to get the face centroid value.

```
/*********************************************************************
    Example of an adjust UDF that uses Lookup_Thread.
    Note that if this UDF is applied to a multiphase flow problem,
    the thread that is returned is the mixture-level thread
*********************************************************************/

#include "udf.h"

/* domain passed to Adjust function is mixture domain for multiphase*/

DEFINE_ADJUST(print_f_centroids, domain)
{
    real FC[2];
    face_t f;
    int ID = 1;
    /* Zone ID for wall-1 zone from Boundary Conditions task page */
    Thread *thread = Lookup_Thread(domain, ID);
    begin_f_loop(f, thread)
    {
        F_CENTROID(FC,f,thread);
        printf("x-coord = %f y-coord = %f", FC[0], FC[1]);
    }
```

```
        end_f_loop(f,thread)
    }
```

3.2.6.2. Zone ID (`THREAD_ID`)

You can use `THREAD_ID` when you want to retrieve the integer zone ID number (displayed in a boundary conditions dialog box such as `Fluid`) that is associated with a given thread pointer `t`. Note that this macro does the inverse of `Lookup_Thread` (see above).

```
    int zone_ID = THREAD_ID(t);
```

3.2.6.3. Domain Pointer (`Get_Domain`)

You can use the `Get_Domain` macro to retrieve a domain pointer when it is not explicitly passed as an argument to your UDF. This is commonly used in `ON_DEMAND` functions since `DEFINE_ON_DEMAND` is not passed any arguments from the Ansys Fluent solver. It is also used in initialization and adjust functions for multiphase applications where a phase domain pointer is needed but only a mixture pointer is passed.

```
    Get_Domain(domain_id);
```

`domain_id` is an `integer` whose value is 1 for the mixture domain, but the values for the phase domains can be any integer greater than 1. The ID for a particular phase can be found be selecting it in the **Phases** dialog box in Ansys Fluent.

 Setup → **Models** → **Multiphase** → **Phases** **Edit...**

Single-Phase Flows

In the case of single-phase flows, `domain_id` is 1 and `Get_Domain(1)` will return the fluid domain pointer.

```
    DEFINE_ON_DEMAND(my_udf)
    {
       Domain *domain;          /* domain is declared as a variable  */
       domain = Get_Domain(1);  /* returns fluid domain pointer       */
       ...
    }
```

Multiphase Flows

In the case of multiphase flows, the value returned by `Get_Domain` is either the mixture-level, a phase-level, or an interaction phase-level domain pointer. The value of `domain_id` is always 1 for the mixture domain. You can obtain the `domain_id` using the Ansys Fluent graphical user interface much in the same way that you can determine the zone ID from the **Boundary Conditions** task page. Simply go to the **Phases** dialog box in Ansys Fluent and select the desired phase. The `domain_id` will then be displayed. You will need to hard code this integer ID as an argument to the macro as shown below.

```
    DEFINE_ON_DEMAND(my_udf)
    {
       Domain *mixture_domain;
       mixture_domain = Get_Domain(1); /* returns mixture domain pointer */
                                       /* and assigns to variable        */
```

```
    Domain *subdomain;
    subdomain = Get_Domain(2);      /* returns phase with ID=2 domain pointer*/
                                    /* and assigns to variable              */
    ...
}
```

Example

The following example is a UDF named `get_coords` that prints the thread face centroids for two specified thread IDs. The function implements the `Get_Domain` utility for a single-phase application. In this example, the function `Print_Thread_Face_Centroids` uses the `Lookup_Thread` function to determine the pointer to a thread, and then writes the face centroids of all the faces in a specified thread to a file. The `Get_Domain(1)` function call returns the pointer to the domain (or mixture domain, in the case of a multiphase application). This argument is not passed to `DEFINE_ON_DEMAND`.

```
/*****************************************************************
    Example of UDF for single phase that uses Get_Domain utility
 *****************************************************************/

#include "udf.h"

FILE *fout;

void Print_Thread_Face_Centroids(Domain *domain, int id)
{
    real FC[3];
    face_t f;
    Thread *t = Lookup_Thread(domain, id);
    fprintf(fout,"thread id %d\n", id);
    begin_f_loop(f,t)
      {
        F_CENTROID(FC,f,t);
        fprintf(fout, "f%d %g %g %g\n", f, FC[0], FC[1], FC[2]);
      }
    end_f_loop(f,t)
    fprintf(fout, "\n");
}

DEFINE_ON_DEMAND(get_coords)
{
    Domain *domain;
    domain = Get_Domain(1);
    fout = fopen("faces.out", "w");
    Print_Thread_Face_Centroids(domain, 2);
    Print_Thread_Face_Centroids(domain, 4);
    fclose(fout);
}
```

Note that `Get_Domain(1)` replaces the `extern Domain *domain` expression used in releases of **Ansys Fluent 6**.

3.2.6.4. Set Boundary Condition Value (`F_PROFILE`)

`F_PROFILE` is typically used in a `DEFINE_PROFILE` UDF to set a boundary condition value in memory for a given face and thread. The index `i` that is an argument to `F_PROFILE` is also an argument to `DEFINE_PROFILE` and identifies the particular boundary variable (for example, pressure, temperature, velocity) that is to be set. `F_PROFILE` is defined in `mem.h`.

Macro: F_PROFILE (f, t, i)

Argument types:	face_t f
	Thread *t
	int i
Function returns:	void

The arguments of F_PROFILE are f, the index of the face face_t; t, a pointer to the face's thread t; and i, an integer index to the particular face variable that is to be set. i is defined by Ansys Fluent when you hook a DEFINE_PROFILE UDF to a particular variable (for example, pressure, temperature, velocity) in a boundary condition dialog box. This index is passed to your UDF by the Ansys Fluent solver so that the function knows which variable to operate on.

Suppose you want to define a custom inlet boundary pressure profile for your Ansys Fluent case defined by the following equation:

$$p(y) = 1.1 \times 10^5 - 0.1 \times 10^5 \left(\frac{y}{0.0745} \right)^2$$

You can set the pressure profile using a DEFINE_PROFILE UDF. Since a profile is an array of data, your UDF will need to create the pressure array by looping over all faces in the boundary zone, and for each face, set the pressure value using F_PROFILE. In the sample UDF source code shown below, the y coordinate of the centroid is obtained using F_CENTROID, and this value is used in the pressure calculation that is stored for each face. The solver passes the UDF the right index to the pressure variable because the UDF is hooked to **Gauge Total Pressure** in the **Pressure Inlet** boundary condition dialog box. See DEFINE_PROFILE (p. 108) for more information on DEFINE_PROFILE UDFs.

```
/************************************************************************
   UDF for specifying a parabolic pressure profile boundary profile
 ************************************************************************/

#include "udf.h"

DEFINE_PROFILE(pressure_profile,t,i)
{
   real x[ND_ND];      /* this will hold the position vector */
   real y;
   face_t f;
   begin_f_loop(f,t)
   {
    F_CENTROID(x,f,t);
    y = x[1];
    F_PROFILE(f,t,i) = 1.1e5 - y*y/(.0745*.0745)*0.1e5;
   }
   end_f_loop(f,t)
}
```

3.2.6.5. THREAD_SHADOW(t)

THREAD_SHADOW returns the face thread that is the shadow of Thread *t if it is one of a face/face-shadow pair that makes up a thin wall. It returns NULL if the boundary is not part of a thin wall and is often used in an if statement such as:

```
if (!NULLP(ts = THREAD_SHADOW(t)))
{
   /* Do things here using the shadow wall thread (ts) */
}
```

3.2.7. Time-Sampled Data

In transient simulations, Ansys Fluent can collect time-sampled data for postprocessing of time-averaged mean and RMS values of many solution variables. In addition, resolved Reynolds stresses and some other correlation functions can be calculated.

To access the quantities that can be evaluated during postprocessing, the following macros can be used:

- **Mean Values**

  – Pressure/Velocity Components:

  ```
  P_mean = C_STORAGE_R(c,t,  SV_P_MEAN)/delta_time_sampled;
  u_mean = C_STORAGE_R(c,t,  SV_U_MEAN)/delta_time_sampled;
  v_mean = C_STORAGE_R(c,t,  SV_V_MEAN)/delta_time_sampled;
  w_mean = C_STORAGE_R(c,t,  SV_W_MEAN)/delta_time_sampled;
  ```

  – Temperature/Species Mass Fraction:

  ```
  T_mean = C_STORAGE_R(c,t,  SV_T_MEAN)/delta_time_sampled;
  YI_mean = C_STORAGE_R_XV(c,t,
  SV_Y_MEAN,n)/delta_time_sampled_species[n];
  ```

  – Mixture Fraction/Progress Variable:

  ```
  Mixture_mean = C_STORAGE_R(c,t,
  SV_F_MEAN)/delta_time_sampled_non_premix;
  progress_mean = C_STORAGE_R(c,t,
  SV_C_MEAN)/delta_time_sampled_premix;
  ```

These quantities, as well as many others, may or may not be available depending on what models have been activated. Access to all of them always follows the same structure.

> **Note:**
>
> The storage variable identifiers `SV_..._MEAN` do not refer directly to time-averaged quantities. Instead, these storage variables contain the time-integral of these variables. It is necessary to divide by the sampling time to obtain the time averaged values.

- **RMS Values**

  – Pressure/Velocity Components:

  ```
  P_rms = RMS(C_STORAGE_R(c,t,  SV_P_MEAN), C_STORAGE_R(c,t,
  SV_P_RMS), delta_time_sampled, SQR(delta_time_sampled));
  ```

```
u_rms = RMS(C_STORAGE_R(c,t, SV_U_MEAN), C_STORAGE_R(c,t,
SV_U_RMS), delta_time_sampled, SQR(delta_time_sampled));

v_rms = RMS(C_STORAGE_R(c,t, SV_V_MEAN), C_STORAGE_R(c,t,
SV_V_RMS), delta_time_sampled, SQR(delta_time_sampled));

w_rms = RMS(C_STORAGE_R(c,t, SV_W_MEAN), C_STORAGE_R(c,t,
SV_W_RMS), delta_time_sampled, SQR(delta_time_sampled));
```

– Temperature/Species Mass Fraction:

```
T_rms = RMS(C_STORAGE_R(c,t, SV_T_MEAN), C_STORAGE_R(c,t,
SV_T_RMS), delta_time_sampled, SQR(delta_time_sampled));

YI_rms = RMS(C_STORAGE_R(c,t, SV_YI_MEAN(n)), C_STOR-
AGE_R(c,t, SV_YI_RMS(n)), delta_time_sampled_species[n],
SQR(delta_time_sampled));
```

The `RMS` preprocessor macro must be defined before it is used:

```
#define RMS(mean_accum, rms_accum, n, nsq) \
sqrt(fabs(rms_accum/n - SQR(mean_accum)/nsq))
```

Again, these quantities, as well as many others, may or may not be available depending on what models have been activated. Access to all of them always follows the same structure.

> **Note:**
>
> The storage variable identifiers `SV_..._RMS` contain the time-integral of the square of the solution variables. It is necessary to divide by the sampling time to obtain the time averaged values.

- **Resolved Reynolds (Shear) Stresses**

  UV: `uiuj = CROSS_CORRELATION(C_STORAGE_R(c,t, SV_U_MEAN),`
  `C_STORAGE_R(c,t, SV_V_MEAN), C_STORAGE_R(c,t,`
  `SV_UV_MEAN), delta_time_sampled_shear,`
  `SQR(delta_time_sampled));`

  UW: `uiuj = CROSS_CORRELATION(C_STORAGE_R(c,t, SV_U_MEAN),`
  `C_STORAGE_R(c,t, SV_W_MEAN), C_STORAGE_R(c,t,`
  `SV_UW_MEAN), delta_time_sampled_shear,`
  `SQR(delta_time_sampled));`

  VW: `uiuj = CROSS_CORRELATION(C_STORAGE_R(c,t, SV_V_MEAN),`
  `C_STORAGE_R(c,t, SV_W_MEAN), C_STORAGE_R(c,t,`
  `SV_VW_MEAN), delta_time_sampled_shear,`
  `SQR(delta_time_sampled));`

  As before, the `CROSS_CORRELATION` preprocessor macro must be defined before it is used:

```
#define CROSS_CORRELATION(mean_accum_1, mean_accum_2, cross_accum, n_cross, nsq_mean) \
(cross_accum/n_cross - (mean_accum_1*mean_accum_2)/nsq_mean)
```

These quantities, as well as many others, may or may not be available depending on what models have been activated. Access to all of them always follows the same structure.

Note:

The `SV_UV/UW/VW_MEAN` storage variables contain time-integrals of the products of two different storage variables each. In addition to the Reynolds stresses, `SV_[U|V|W]T_MEAN` are available to calculate temperature-velocity component correlations as shown below:

```
ut = CROSS_CORRELATION(C_STORAGE_R(c,t, SV_U_MEAN),
C_STORAGE_R(c,t, SV_T_MEAN), C_STORAGE_R(c,t,
SV_UT_MEAN), delta_time_sampled_heat_flux,
SQR(delta_time_sampled));
```

3.2.8. Model-Specific Macros

3.2.8.1. DPM Macros

The macros listed in Table 3.27: Macros for Particles at Current Position Defined in `dpm_types.h` (p. 336) – Table 3.32: Macros for Particle Material Properties Defined in `dpm_laws.h` (p. 339) can be used to return `real` variables associated with the Discrete Phase Model (DPM), in SI units (where relevant). They are typically used in DPM UDFs that are described in Discrete Phase Model (DPM) `DEFINE` Macros (p. 201). The variables are available in both the pressure-based and the density-based solver. The macros are defined in the `dpm_types.h` and `dpm_laws.h` header files, which are included in `udf.h`.

The variable `tp` indicates a pointer to the `Tracked_Particle` structure (`Tracked_Particle *tp`) which gives you the value for the particle at the current position, the variable `m` indicates a pointer to the relevant Material structure (Material *m), and the real variable `t` is the temperature.

Refer to the following sections for examples of UDFs that use some of these macros: DEFINE_DPM_LAW (p. 226), DEFINE_DPM_BC (p. 203), DEFINE_DPM_INJECTION_INIT (p. 220), DEFINE_DPM_SWITCH (p. 247), and DEFINE_DPM_PROPERTY (p. 237).

The `TP_...` macros listed in Table 3.27: Macros for Particles at Current Position Defined in `dpm_types.h` (p. 336) through Table 3.31: Macros for Particle Species, Laws, Materials, and User Scalars Defined in `dpm_types.h` (p. 338) are to be used to access elements of variables of the type `Tracked_Particles` or elements within instances of the data structure pointed to by pointer variables of the type `Tracked_Particle *`.

In addition, the `dpm_types.h` header file also contains definitions for the `PP_...` macros that are analogous to the corresponding `TP_...` macros. The `PP_...` macros are similar in function,

but they are to be used for accessing elements of variables of the `Particle` or elements within the data structure that are pointed to by pointer variables of the type `Particle *`.

Important:

The `TP_....` and analogous `PP_...` macros should not be used interchangeably. Their definitions may be modified without notice in future releases.

Table 3.27: Macros for Particles at Current Position Defined in `dpm_types.h`

Macro	Argument Types	Returns
`TP_POS(tp)[i]`	`Tracked_Particle *tp, int i`	position i = 0, 1, 2
`TP_VEL(tp)[i]`	`Tracked_Particle *tp, int i`	velocity i = 0, 1, 2
`TP_DIAM(tp)`	`Tracked_Particle *tp`	diameter
`TP_T(tp)`	`Tracked_Particle *tp`	temperature
`TP_RHO(tp)`	`Tracked_Particle *tp`	density
`TP_MASS(tp)`	`Tracked_Particle *tp`	mass
`TP_TIME(tp)`	`Tracked_Particle *tp`	current particle time
`TP_DT(tp)`	`Tracked_Particle *tp`	time step
`TP_FLOW_RATE(tp)`	`Tracked_Particle *tp`	flow rate of particles in a stream in kg/s (see below for details)
`TP_LMF(tp)`	`Tracked_Particle *tp`	liquid mass fraction (wet combusting particles only)
`TP_LF(tp)`	`Tracked_Particle *tp`	liquid volume fraction (wet combusting particles only)
`TP_VF(tp)`	`Tracked_Particle *tp`	volatile fraction (combusting particles only)
`TP_CF(tp)`	`Tracked_Particle *tp`	char mass fraction (combusting particles only)
`TP_VFF(tp)`	`Tracked_Particle *tp`	volatile fraction remaining (two competing rates devolatilization model only)

`TP_FLOW_RATE(tp)`

Each particle in a steady flow calculation represents a "stream" of many particles that follow the same path. The number of particles in this stream that passes a particular point in a second is the

"strength" of the stream. `TP_FLOW_RATE(tp)` returns the strength multiplied by `TP_INIT_MASS(tp)` at the current particle position.

Table 3.28: Macros for Particles at Entry to Current Cell Defined in `dpm_types.h`

Macro	Argument Types	Returns
`TP_POS0(tp)[i]`	`Tracked_Particle *tp,int i`	position i= 0, 1, 2
`TP_VEL0(tp)[i]`	`Tracked_Particle *tp,int i`	velocity i= 0, 1, 2
`TP_DIAM0(tp)`	`Tracked_Particle *tp`	diameter
`TP_T0(tp)`	`Tracked_Particle *tp`	temperature
`TP_RHO0(tp)`	`Tracked_Particle *tp`	density
`TP_MASS0(tp)`	`Tracked_Particle *tp`	mass
`TP_TIME0(tp)`	`Tracked_Particle *tp`	particle time at entry
`TP_LMF0(tp)`	`Tracked_Particle *tp`	liquid mass fraction (wet combusting particles only)

Important:

Note that when you are using the macros listed in Table 3.28: Macros for Particles at Entry to Current Cell Defined in `dpm_types.h` (p. 337) to track transient particles, the particle state is the beginning of the fluid flow time step only if the particle does *not* cross a cell boundary.

Table 3.29: Macros for Particle Cell Index and Thread Pointer Defined in `dpm_types.h`

Name (Arguments)	Argument Types	Returns
`TP_CELL(tp)`	`Tracked_Particle *tp`	`cell_t c`, cell index of the cell in which the particle is currently located
`TP_CELL_THREAD(tp)`	`Tracked_Particle *tp`	Thread `*tc`, pointer to the thread of the cell in which the particle is currently located
`TP_CCELL(tp)`	`Tracked_Particle *tp`	The address of the variable `cCell` stored on the tracked particle structure, which includes both the cell index (`TP_CELL`) and the cell thread (`TP_CELL_THREAD`). This

Name (Arguments)	Argument Types	Returns
		corresponds to the cell in which the particle is currently located.
`TP_FILM_FACE(tp)`	`Tracked_Particle *tp`	`face_t f`, face index of the face to which the film particle is currently attached
`TP_FILM_THREAD(tp)`	`Tracked_Particle *tp`	`Thread *tf`, pointer to the thread of the face to which the film particle is currently attached

Table 3.30: Macros for Particles at Injection into Domain Defined in `dpm_types.h`

Macro	Argument Types	Returns
`TP_INIT_POS(tp)[i]`	`Tracked_Particle *tp, int i`	position i = 0, 1, 2
`TP_INIT_VEL(tp)[i]`	`Tracked_Particle *tp, int i`	velocity i = 0, 1, 2
`TP_INIT_DIAM(tp)`	`Tracked_Particle *tp`	diameter
`TP_INIT_TEMP(tp)`	`Tracked_Particle *tp`	temperature
`TP_INIT_RHO(tp)`	`Tracked_Particle *tp`	density
`TP_INIT_MASS(tp)`	`Tracked_Particle *tp`	mass
`TP_INIT_LMF(tp)`	`Tracked_Particle *tp`	liquid mass fraction (wet combusting particles only)

Table 3.31: Macros for Particle Species, Laws, Materials, and User Scalars Defined in `dpm_types.h`

Macro	Argument Types	Returns
`TP_EVAP_SPECIES_IN-DEX(tp)`	`Tracked_Particle *tp`	evaporating species index in mixture
`TP_DEVOL_SPECIES_IN-DEX(tp)`	`Tracked_Particle *tp`	devolatilizing species index in mixture
`TP_OXID_SPECIES_IN-DEX(tp)`	`Tracked_Particle *tp`	oxidizing species index in mixture
`TP_PROD_SPECIES_IN-DEX(tp)`	`Tracked_Particle *tp`	combustion products species index in mixture
`TP_CURRENT_LAW(tp)`	`Tracked_Particle *tp`	int law, current particle law index

Macro	Argument Types	Returns
`TP_NEXT_LAW(tp)`	`Tracked_Particle *tp`	int next_law, next particle law index
`TP_MATERIAL(tp)`	`Tracked_Particle *tp`	Material *m, material pointer
`TP_USER_REAL(tp,i)`	`Tracked_Particle *tp, int i`	storage array for user-defined values (indexed by `i`)

Table 3.32: Macros for Particle Material Properties Defined in `dpm_laws.h`

Macro	Argument Types	Returns
`DPM_BOILING_TEMPER-ATURE (tp,m)`	`Tracked_Particle *tp,Material *m`	boiling temperature

`DPM_DIFFU-SION_COEFF(tp,t)`
 	
`DPM_BINARY_DIFFUSIV-ITY(tp,m,t)`
	`Tracked_Particle *tp,Material *m, real t`	Binary diffusion coefficient to be used in the gaseous boundary layer around the particle

`DPM_EMISSIV-ITY(tp,m)`
 	
`DPM_SCATT_FACTOR(tp,m)`
	`Tracked_Particle *tp,Material *m`	emissivity and scattering factor for the radiation model

`DPM_KTC(tp)`	`Tracked_Particle *tp`	thermal conductivity
`DPM_HEAT_OF_PYROLYS-IS(tp)`	`Tracked_Particle *tp`	heat of pyrolysis
`DPM_HEAT_OF_REAC-TION(tp)`	`Tracked_Particle *tp`	heat of reaction
`DPM_LAT-ENT_HEAT(tp,m)`	`Tracked_Particle *tp, Material *m`	latent heat
`DPM_LIQUID_SPECIF-IC_HEAT(tp,t)`	`Tracked_Particle *tp,real t`	specific heat of material used for liquid associated with particle
`DPM_MU(tp)`	`Tracked_Particle *tp`	dynamic viscosity of liquid part of particle
`DPM_RHO(tp,m,t)`	`Tracked_Particle *tp,Material *m, real t`	particle density
`DPM_SPECIF-IC_HEAT(tp,t)`	`Tracked_Particle *tp,real t`	specific heat
`DPM_SWELL-ING_COEFF(tp)`	`Tracked_Particle *tp`	swelling coefficient for devolatilization
`DPM_SURFTEN(tp)`	`Tracked_Particle *tp`	surface tension of liquid part of particles

Macro	Argument Types	Returns

`DPM_VAPOR_PRES-`
`SURE(tp,m,t)`
	`Tracked_Particle`
	`*tp,Material *m,`
	`real t`
		vapor pressure of liquid part of particle

`DPM_VA-`
`POR_TEMP(tp,m)`
 	
`DPM_EVAPORATION_TEM-`
`PERATURE(tp,m)`
	`Tracked_Particle`
	`*tp,Material *m`
		vaporization temperature used to switch to vaporization law

`DPM_VOLATILE_FRAC-` `TION(tp)`	`Tracked_Particle` `*tp`	particle material volatile fraction
`DPM_CHAR_FRAC-` `TION(tp)`	`Tracked_Particle` `*tp`	particle material char fraction

Table 3.33: Macros to access source terms on CFD cells for the DPM model

Macro	Argument Types	Returns
`C_DPMS_MOM_S(c,t)`	`cell_t c, Thread` `*t`	momentum source (SI units: kg m/s^2), explicit
`C_DPMS_MOM_AP(c,t)`	`cell_t c, Thread` `*t`	momentum source (SI units: kg/s), implicit
`C_DPMS_WSWIRL_S(c,t)`	`cell_t c, Thread` `*t`	momentum source swirl component (SI units: kg m/s^2), explicit
`C_DPMS_WSWIRL_AP(c,t)`	`cell_t c, Thread` `*t`	momentum source swirl component (SI units: kg/s), implicit
`C_DPMS_ENERGY(c,t)`	`cell_t c, Thread` `*t`	energy source (SI units: J/s), explicit
`C_DPMS_ENERGY_AP(c,t)`	`cell_t c, Thread` `*t`	energy source (SI units: J/K/s), implicit

`C_DPMS_YI(c,t,i)`	`cell_t c, Thread`
	`*t,int i (species`
	`index)`
		species mass source (SI units: kg/s), explicit

`C_DPMS_YI_AP(c,t,i)`	`cell_t c, Thread`
	`*t,int i (species`
	`index)`
		species mass source (SI units: kg/s), implicit

`C_DPMS_REAC-`
`TION_RATE_POST(c,t,i)`
	`cell_t c, Thread`
	`*t,int i (particle`
	`reaction index)`
		reaction rates of particle REACTIONS (SI units: kg/s/m^3)

`C_DPMS_VAP_PER_MAT(c,t,i)`	`cell_t c, Thread`
	`*t,int i (material`
	`index)`
		vaporization mass source of material `i` (SI units: kg/s)

Macro	Argument Types	Returns
`C_DPMS_DE-VOL_PER_MAT(c,t,i)`	`cell_t c, Thread *t,int i` (material index)	devolatilization mass source of material `i` (SI units: kg/s)
`C_DPMS_BURN_PER_MAT(c,t,i)`	`cell_t c, Thread *t,int i` (material index)	burnout mass source of material `i` (SI units: kg/s)
`C_DPMS_PDF_1(c,t)`	`cell_t c, Thread *t`	mass source of pdf stream 1 (SI units: kg/s)
`C_DPMS_PDF_2(c,t)`	`cell_t c, Thread *t`	mass source of pdf stream 2 (SI units: kg/s)
`C_DPMS_EMISS(c,t)`	`cell_t c, Thread *t`	particles emissivity (SI units: m^2K^4)
`C_DPMS_ABS(c,t)`	`cell_t c, Thread *t`	particles absorption coefficient (SI units: m^3/m)
`C_DPMS_SCAT(c,t)`	`cell_t c, Thread *t`	particles scattering coefficient (SI units: m^3/m)
`C_DPMS_BURNOUT(c,t)`	`cell_t c, Thread *t`	burnout mass source (SI units: kg/s)
`C_DPMS_CONCENTRA-TION(c,t)`	`cell_t c, Thread *t`	concentration of particles (SI units: $kg/s/m^3$)
`C_DPMS_SURF_YI(c,t,i)`	`cell_t c, Thread *t,int i` (surface species index)	concentration of particle surface species (SI units: $kg/s/m^3$)

Note:

- The macros listed in Table 3.33: Macros to access source terms on CFD cells for the DPM model (p. 340) in general are not intended to be used to assign values inside `DEFINE_DPM_SOURCE`.

- By design, you cannot assign values to some macros, such as `C_DPMS_YI` and `C_DPMS_YI_AP`. These macros are used only for reporting purposes in `DEFINE_ADJUST` and `DEFINE_ON_DEMAND`.

3.2.8.2. NOx Macros

The following macros can be used in NOx model UDFs in the calculation of pollutant rates. These macros are defined in the header file `sg_nox.h`, which is included in `udf.h`. They can be used to return `real` NOx variables in SI units, and are available in both the pressure-based and the

density-based solver. See for examples of `DEFINE_NOX_RATE` UDFs that use these macros.

Table 3.34: Macros for NOx UDFs Defined in `sg_nox.h`

Macro	Returns
`POLLUT_EQN(Pollut_Par)`	index of pollutant equation being solved (see below)
`MOLECON(Pollut,SPE)`	molar concentration of species specified by `SPE` (see below)
`NULLIDX(Pollut_Par,SPE)`	TRUE if the species specified by `SPE` does not exist in Ansys Fluent case (that is, in the **Species** dialog box)
`ARRH(Pollut,K)`	Arrhenius rate calculated from the constants specified by `K` (see below)
`POLLUT_FRATE(Pollut)`	production rate of the pollutant species being solved
`POLLUT_RRATE(Pollut)`	reduction rate of the pollutant species being solved
`POLLUT_QRATE(Pollut)`	quasi-steady rate of N_2O formation (if the quasi-steady model is used)
`POLLUT_FLUCTDEN(Pollut)`	fluctuating density value (or, if no PDF model is used, mean density at a given cell
`POLLUT_FLUCTTEM(Pollut)`	fluctuating temperature value (or, if no PDF model is used, mean temperature at a given cell)
`POLLUT_FLUCTYI(Pollut,SPE)`	fluctuating mass fraction value (or, if no PDF model is used, mean mass fraction at a given cell) of the species given by index `SPE`
`POLLUT_CTMAX(Pollut_Par)`	upper limit for the temperature PDF integration (see below)

Important:

> `Pollut_Par` is a pointer to the `Pollut_Parameter` data structure that contains auxiliary data common to all pollutant species and `NOx` is a pointer to the `NOx_Parameter` data structure that contains data specific to the NOx model.

- `POLLUT_EQN(Pollut_Par)` returns the index of the pollutant equation currently being solved. The indices are `EQ_NO` for NO, `EQ_HCN` for HCN, `EQ_N2O` for N_2O, and `EQ_NH3` for NH_3.

- `MOLECON(Pollut,SPE)` returns the molar concentration of a species specified by `SPE`, which is either the name of the species or `IDX(i)` when the species is a pollutant (like NO). `SPE` must be replaced by one of the following identifiers: `FUEL`, `O2`, `O`, `OH`, `H2O`, `N2`, `N`, `CH`, `CH2`, `CH3`, `IDX(NO)`, `IDX(N2O)`, `IDX(HCN)`, `IDX(NH3)`. For example, for O_2 molar concentration you should call `MOLECON(Pollut, O2)`, whereas for NO molar concentration the call should be `MOLECON(Pollut, IDX(NO))`. The identifier `FUEL` represents the fuel

species as specified in the **Fuel Species** drop-down list under **Prompt NO Parameters** in the **NOx Model** dialog box.

- `ARRH(Pollut,K)` returns the Arrhenius rate calculated from the constants specified by `K`. `K` is defined using the `Rate_Const` data type and has three elements - *A*, *B*, and *C*. The Arrhenius rate is given in the form of

$$R = AT^B \exp\left(-C \,/\, T\right)$$

where *T* is the temperature.

Note that the units of `K` must be in m-mol-J-s.

- `POLLUT_CTMAX(Pollut_Par)` can be used to modify the $T_{\max}$ value used as the upper limit for the integration of the temperature PDF (when temperature is accounted for in the turbulence interaction modeling). You must make sure not to put this macro under any conditions within the UDF (for example, `IN_PDF` or `OUT_PDF`).

3.2.8.3. Dynamic Mesh Macros

The macros listed in Table 3.35: Macros for Dynamic Mesh Variables Defined in `dynamesh_tools.h` (p. 343) are useful in dynamic mesh UDFs. The argument `dt` is a pointer to the dynamic thread structure, and `time` is a real value. These macros are defined in the `dynamesh_tools.h`.

Table 3.35: Macros for Dynamic Mesh Variables Defined in `dynamesh_tools.h`

Name (Arguments)	Argument Types	Returns
`DT_THREAD(dt)`	`Dynamic_Thread *dt`	pointer to a thread
`DT_CG(dt)`	`Dynamic_Thread *dt`	center of gravity vector
`DT_VEL_CG(dt)`	`Dynamic_Thread *dt`	cg velocity vector
`DT_OMEGA_CG(t)`	`Dynamic_Thread *dt`	angular velocity vector
`DT_THETA(dt)`	`Dynamic_Thread *dt`	orientation of body-fixed axis vector
`DYNAMESH_CUR-RENT_TIME`	N/A	current dynamic mesh time
`TIME_TO_ABSO-LUTE_CRANK_ANGLE (time)`	`real time`	absolute value of the crank angle

See `DEFINE_GRID_MOTION` (p. 271) for an example UDF that utilizes `DT_THREAD`.

3.2.8.4. Battery Model Macros

The macros listed in Table 3.36: Macros for Battery UDFs Defined in `sg_mem.h` (p. 344) can be used in a UDF to access the MSMD battery model variables. These macros are defined in the header file

`sg_mem.h`, which is included in `udf.h`. They can be used to return real variables in SI units. Note that they are available only if the MSMD solution method is used in a simulation.

Table 3.36: Macros for Battery UDFs Defined in `sg_mem.h`

Macro	Returns
`C_BATTERY_DISCHARGE_DEPTH(c, t)`	degree of discharge (-)
`C_BATTERY_SHORT_R(c,t)`	volumetric contact resistance in the internal short circuit model (ohm m^3)
`C_BATTERY_OHMIC_SOURCE(c,t`	heat source due to Joule heating (W/m^3)
`C_BATTERY_ECHEM_SOURCE(c,t)`	heat source due to electrochemical reaction (W/m^3)
`C_BATTERY_SHORT_SOURCE(c,t)`	heat source due to internal short (W/m^3)
`C_BATTERY_ABUSE_SOURCE(c,t)`	heat source due to thermal abuse reactions (W/m^3)
`C_BATTERY_T_SOURCE(c,t)`	total heat source (W/m^3)
`C_BATTERY_SEI_REACTION(c,t)`	reaction progress variable for the decomposition reaction of SEI layer (-)
`C_BATTERY_PE_REACTION(c,t)`	reaction progress variable for the decomposition reaction between positive electrode and solvent (-)
`C_BATTERY_NE_REACTION(c,t)`	reaction progress variable for the decomposition reaction between negative electrode and solvent (-)
`C_BATTERY_E_REACTION(c,t)`	reaction progress variable for the decomposition reaction of electrolyte (-)
`C_BATTERY_ISC_REACTION(c,t)`	reaction progress variable for the decomposition reaction due to internal short circuit (-)
`C_BATTERY_CURR_X(c,t)`	electric current component in x-direction (A/m^2)
`C_BATTERY_CURR_Y(c,t)`	electric current component in y-direction (A/m^2)
`C_BATTERY_CURR_Z(c,t)`	electric current component in z-direction (A/m^2)

3.2.9. NIST Real Gas Saturation Properties

You can create saturation tables for pure fluids, binary, and multi-species mixtures. Fluent provides the following functions that you can use in UDFs to compute saturation properties for the bubble and dew points for a single species or multiple-species:

3.2.9.1. Saturation Curves for Single-Species

When using a single-species NIST real gas model, Fluent provides a function that you can use in your UDFs to obtain saturation properties for the bubble and dew points. You can specify either temperature or pressure and the function will return an array containing the pressures or temperatures, respectively, and the liquid and vapor phase densities.

```
void getsatvalues_NIST(int index, double x, double y[]);
```

 where:

    $index$ = 0 if x is a specified temperature

         = 1 if x is a specified pressure

    x = temperature or pressure (according to value of $index$) at which to obtain saturation properties

    $y[]$ = array of saturation properties

The values returned in $y[]$ depend on the choice of $index$.

	Array elements	**index=0**	**index=1**
Bubble Point	$y[0]$	pressure (Pa)	temperature (K)
	$y[1]$	density of liquid (kg/m^3)	density of liquid (kg/m^3)
Dew Point	$y[2]$	pressure (Pa)	temperature (K)
	$y[3]$	density of vapor (kg/m^3)	density of vapor (kg/m^3)

This function can be used regardless of whether you have enabled the lookup table.

3.2.9.2. Saturation Curves for Multi-Species (UDF 1)

```
int getsatvalues_NIST_msp (int index, double x, double y[], int
ibubbleordew)
```

This function is similar to the single-species function, $getsatvalues\_NIST$, but it applies to multi-species. For a given temperature or pressure you can obtain the corresponding pressure or temperature and the liquid and vapor phase densities.

In the function, $index$, x, and $ibubbleordew$ are inputs. If the $index=0$, the input for x is temperature and if the $index=1$, the input for x is pressure. If $ibubbleordew=0$, Fluent returns bubble point properties, otherwise it returns dew point properties. $y[]$ is an array of saturation properties.

The function returns an integer indicating the number of interpolation points for a given temperature and pressure:

- 0—no interpolation points, which could indicate that the given temperature and pressure is outside of the saturation curve range.

- 1—interpolation point with a bubble or dew curve.

- 2—interpolation points in the retrograde region of the saturation curves.

When `index = 0`:

	Bubble Point	Dew Point
Function Returns 1		
`y[0]`	Pressure	Pressure
`y[1]`	Density (liquid)	Density (vapor)
Function Returns 2		
`y[0]`	Pressure	Pressure
`y[1]`	Density (liquid)	Density (vapor)
`y[2]`	Pressure	Pressure
`y[3]`	Density (liquid)	Density (vapor)

When `index = 1`:

	Bubble Point	Dew Point
Function Returns 1		
`y[0]`	Temperature	Temperature
`y[1]`	Density (liquid)	Density (vapor)
Function Returns 2		
`y[0]`	Temperature	Temperature
`y[1]`	Density (liquid)	Density (vapor)
`y[2]`	Temperature	Temperature
`y[3]`	Density (liquid)	Density (vapor)

3.2.9.3. Saturation Curves for Multi-Species (UDF 2)

```
void get_satprop_NIST_msp (index, z, minrho, maxrho, nsat, npts, p_bub,
t_bub, p_dew, t_dew);
```

The 2nd function for multi-species can be used to generate a saturation table for variable compositions.

Inputs:

Argument	Description
`index`	0: composition in mass fractions >0: composition in mole fractions
`z`	real array—composition of fluid (ensure sum=1)
`minrho`	minimum density in moles per liter (0.1 is a reasonable default for most cases)

Argument	Description
`maxrho`	maximum density in moles per liter (25 is a reasonable default for most cases)
`nsat`	number of points in bubble/dew curves (minimum is 12)

Outputs:

Argument	Description
`npts[0]`	actual number of bubble curve points returned by the function
`npts[1]`	actual number of dew curve points returned by the function
`p_bub`	a pressure array of the bubble point curve returned by the function
`t_bub`	a temperature array of the bubble point curve returned by the function
`p_dew`	a pressure array of the dew point curve returned by the function
`t_dew`	a temperature array of the dew point curve returned by the function

This function computes saturation data for the given composition in z. This could be a binary mixture, three species, or more.

3.2.9.3.1. Using Multi-Species User-Defined Function (UDF2)

To use `get_satprop_NIST_msp` in a UDF:

1. Start a Fluent session as normal.

2. Enter the text command `define/user-defined/real-gas-models/nist-multispecies-real-gas-model` and specify the number and species, but DO NOT activate the NIST-Table. This is only for generating user-defined bubble and dew point curves using NIST models.

3. Ensure that the number and order of components selected in Fluent matches the number and order of species in `z` of the `get_satprop_NIST_msp` function.

> **Note:**
>
> The output temperature and pressure arrays appear in ascending order for dew points and descending order for bubble points. However, when the output is for single-species, that is, $z[i]=1.0$, all $z[k]=0.0$,(k=0,...,n; k!=i), they are in ascending order. The values for the critical condition are also presented in the output arrays. They are stored in the 1$^{\text{st}}$ element of the bubble point array and last element of the dew point array. For single-species, it is stored in the last element of the output arrays for both bubble and dew points.

3.2.9.3.2. Example Multi-Species User-Defined Function (UDF2)

This UDF, named `getsat_msp`, computes and writes saturation curves. It demonstrates how a binary phase envelope is calculated using the NIST real gas model.

```
#include "udf.h"
#include <string.h>

#define NUM_SAT 50
#define SMALL_NUM 1.e-6
DEFINE_ON_DEMAND(getsat_msp)
{
  double t, p, y[10];
  int np,index;
  int nsat,npts[2];
  double z[10]={0.}, minrho, maxrho;
  double p_bub[NUM_SAT],t_bub[NUM_SAT],p_dew[NUM_SAT],t_dew[NUM_SAT];
  FILE *fp1, *fp2;
  int i,j,k,ncomps=2;

  fp1 = fopen("NIST_Bub_Pnts_Curve.xy", "w");
  fp2 = fopen("NIST_Dew_Pnts_Curve.xy", "w");

  /* test on udf built NIST table - sat */
  minrho = 0.1;
  maxrho = 25.;
  nsat = NUM_SAT;
  z[0] = 0.;

  fprintf(fp1, "(title \"Sat Bubble P-T\")");
  fprintf(fp1, "\n(labels \"Temperature (K)\" \"Pressure (kPa)\")\n");
  fprintf(fp2, "(title \"Sat Dew P-T\")");
  fprintf(fp2, "\n(labels \"Temperature (K)\" \"Pressure (kPa)\")\n");

  for (j=0; j<21; j++)
  {
    if (z[0] > 1.) z[0] = 1.;
    z[1] = 1. - z[0];
    if (z[1] < 0.) z[1] = 0.;
    get_satprop_NIST_msp(0,z,minrho,maxrho,nsat,npts,p_bub,t_bub,p_dew,t_dew);

  /* bubble */
  for (i=0; i<1; i++)
    fprintf(fp1, "\n((xy/key/label \"z[0]=%g  numpts=%d\")\n",z[i],npts[0]);
    if (fabs(z[0])<SMALL_NUM || fabs(z[0]-1.)<SMALL_NUM || fabs(z[1])<SMALL_NUM || fabs(z[1]-1.)<SMALL_NUM)
{
  for (i=0;i<npts[0];i++)
 fprintf(fp1, "%g\t %g\t \n",t_bub[i],p_bub[i]/1000.);
}
  else
   {
  for (i=npts[0]-1;i>-1;i--)
  fprintf(fp1, "%g\t %g\t \n",t_bub[i],p_bub[i]/1000.);
}
 fprintf(fp1, ")\n");

  /* dew */
  for (i=0; i<1; i++)
    fprintf(fp2, "\n((xy/key/label \"z[0]=%g  numpts=%d\")\n",z[i],npts[1]);
  for (i=0;i<npts[1];i++)
    fprintf(fp2, "%g\t %g\t \n",t_dew[i],p_dew[i]/1000.);
  fprintf(fp2, ")\n");

  z[0] += 0.05;}
  }
  fclose(fp1);
  fclose(fp2);
}
```

3.2.10. NIST Real Gas UDF Access Macro for Multi-Species Mixtures

3.2.10.1. Description

For given temperature, pressure, and mixture composition, you can obtain thermodynamic properties of a mixture material in liquid, gas, and two-phase regimes using `get_prop_NIST_msp`. NIST implements the GERG model for thermodynamic properties of the mixtures. This model applies mixing rules to the Helmholtz energy for each mixture components. Mixture data for the binary pairs can be found in `hmx.bnc` provided with your Ansys Fluent installation.

```
Void get_prop_NIST_msp(double t, double p, double z[], double prop[]);
```

Argument Type	Description
`double t`	Temperature (K).
`double p`	Pressure (Pa).
`double z[]`	Real array, mass fractions of each component/species. The sum of the mass fractions for all the materials in a mixture must be 1. `z[0]` corresponds to the first component/species mass fraction, z[1] to the second, etc. The mixture is limited to a maximum of 20 components/species.

Function returns

`double prop[]`: Real array with thermodynamic properties as follows.

Thermodynamic Properties Output	
Vapor quality	
`prop[NIST_q_molar]=q`	vapor/gas moles/mixture moles
	$q<0$ indicates subcooled (compressed) liquid
	$q=0$ indicates saturated liquid
	$0<q<1$ indicates two-phase regime
	$q=1$ indicates saturated vapor/gas
	$q=998$ superheated vapor/gas, but quality not defined
	$q=999$ supercritical state ($t>T_c$) and ($p>P_c$)
`prop[NIST_q_mass]`	vapor/gas mass fraction
`prop[NIST_q_vol]`	vapor/gas volume fraction
Density (Kg/m^3)	
`prop[NIST_den_m]`	bulk/mixture density
`prop[NIST_den_l]`	liquid phase density
`prop[NIST_den_v]`	vapor/gas phase density
Specific-heat, cp (J/Kg-K)	

Thermodynamic Properties Output	
`prop[NIST_cp_m]`	bulk/mixture specific heat
`prop[NIST_cp_l]`	liquid phase specific heat
`prop[NIST_cp_v]`	vapor/gas phase specific heat
Thermal conductivity (W/m-K)	
`prop[NIST_ktc_m]`	bulk/mixture thermal conductivity
`prop[NIST_ktc_l]`	liquid phase thermal conductivity
`prop[NIST_ktc_v]`	vapor/gas phase thermal conductivity
Viscosity (Pa-s)	
`prop[NIST_mu_m]`	bulk/mixture viscosity
`prop[NIST_mu_l]`	liquid phase viscosity
`prop[NIST_mu_v]`	vapor/gas phase viscosity
Enthalpy (J/Kg)	
`prop[NIST_h_m]`	bulk/mixture enthalpy
`prop[NIST_h_l]`	liquid phase enthalpy
`prop[NIST_h_v]`	vapor/gas phase enthalpy
Speed of sound (m/s)	
`prop[NIST_a_m]`	bulk speed of sound (0 if two-phase regime)
`prop[NIST_a_l]`	in liquid phase
`prop[NIST_a_v]`	in vapor/gas phase
Component mass fractions in liquid phase	
`prop[NIST_massfrac0_l+i]`	component/species, i = [0] … [n-1]
Component mass fractions in vapor phase	
`prop[NIST_massfrac0_v+i]`	component/species, i = [0] … [n-1]

Important:

The ranges of temperatures and pressures must be carefully selected, so that the limits on the material property application are not breached.

NIST is currently unable to calculate two-phase or mixture regime properties since all properties are relative to a pure or single phase. Ansys Fluent implements simple mixing rules based on volume fractions:

$$\rho = \sum_i \rho_i \alpha_i$$

where ρ is the bulk property, i refers to a liquid or vapor phase, ρ_i are the phase specific properties in vapor and liquid, and α_i is the volume fraction of the i [th] phase.

3.2.10.2. Using `get_prop_NIST_msp`

You can embed the `get_prop_NIST_msp` UDF access macro in a number of Fluent UDF functions, such as `DEFINE_ON_DEMAND`. For example, you can use `get_prop_NIST_msp` to build a property database with variables of temperature, pressure and composition.

To use the `get_prop_NIST_msp` access function, you must first enable the NIST real gas model for multi-species by entering the following text command in the Fluent console:

define/user-defined/real-gas-models/nist-multispecies-real-gas-model

`use multispecies NIST real gas? [no]` **yes**

Once the model is enabled, the list of available pure-fluid materials that you can select will be displayed.

Then enter responses to the following prompts in the console:

1. `Number of species [0]`

   Enter the number of species that form the mixture.

2. `select real-gas data file 1 []`

   For each fluid material, enter the name enclosed in quotation marks. For example:

   `select real-gas data file 1 []` **"methane.fld"**

   `select real-gas data file 2 []` **"ethane.fld"**

   Once the fluids are selected, Fluent prints a list of properties for each fluid material and its application ranges of temperature and pressure.

3. `Create NIST LookUp Table for multi components?`

   `[no]`

   Enter **no** at the prompt.

4. Follow the normal UDF practices.

3.2.10.3. Error Handling

If the input parameters for `get_prop_NIST_msp` are outside the limits specified in NIST, the first element of the prop[] array, `prop[NIST_q_molar]=q`, will return the value of -2999.0. This indicates that NIST calculations of thermodynamic properties are questionable or simply impossible for given temperature, pressure and/or combination of materials. Since the returned value is a real double number, you can capture this error by using an appropriate conditional statement, such as

`if (q > -2999.1 && q < -2998.9)`

3.2.10.4. Example

The following UDF, named `getprop_msp`, is called from a `DEFINE_ON_DEMAND` UDF Macro (`DEFINE_ON_DEMAND` (p. 33)). The function obtains thermodynamic properties of a mixture consisting of seven species. The species names and mass fractions are specified in the arrays `spe_names[]` and `spe_mfractions[]`, respectively. (Note that the `spe_mfractions[]` array could also be variable.) The values of these thermodynamic properties are generated by the NIST Real Gas Model (current Version 9.1) in Ansys Fluent. A series of temperature and pressure conditions are specified in the arrays `temp[]` and `press[]`, respectively.

The obtained properties values are printed in the console and stored in the `NIST_output.txt` file. Any errors messages are recorded in the `NIST_errors.txt` file.

The UDF can be executed as an interpreted or compiled UDF in Ansys Fluent.

```
#include "udf.h"
#include <string.h>

#define NUM_MAX 150
#define SPE_MAX 20
#define N_SPE 7
#define N_DATA 6
DEFINE_ON_DEMAND(getprop_msp)
{
  double t,p,q,sum;
  double z[SPE_MAX]={0.};
  FILE *fp1,*fp2;
  int i,j,k,n,ncomps;
  double prop[NUM_MAX]={0.};
  double temp[N_DATA]={185,200,215,200,200,200};
  double press[N_DATA]={5.e6,5.e6,5.e6,4.5e6,5.5e6,6.e6};
  char *spe_names[N_SPE]={"methane","nitrogen","carbon-dioxide","ethane","propane","isobutane","butane"};
  double spe_mfractions[N_SPE]={0.70061,0.20227,0.023246,0.053209,0.014306,0.0031167,0.0032414};

  fp1 = fopen("NIST_output.txt","w");
  fp2 = fopen("NIST_errors.txt","w");

  /* testing nist property access function */
 ncomps = N_SPE; /* number of species. to be matched by setting in fluent */
  /* composition in mass fraction */
  sum = 0.;
  for (i=0; i<N_SPE; i++)
  {
    z[i] = spe_mfractions[i];
    sum += z[i];
  }
  for (i=0; i<N_SPE;  i++)
    z[i] /= sum;

  Message0("\nComposition (Mass Fractions): %d species\n",ncomps);
  for (i=0; i<ncomps; i++)
    Message0(" %s : %f ",spe_names[i],z[i]);

  fprintf(fp1, "quality_molar, quality_mass, quality_vol, density, den_liq, den_vap, h, h_liq, h_vap,
          cp, cp_liq, cp_vap, ktc, ktc_liq, ktc_vap, mu, mu_liq, mu_vap, ");
  for (i=0; i<ncomps; i++)
    fprintf(fp1, " %s_liq, ",spe_names[i]);
  for (i=0; i<ncomps; i++)
    fprintf(fp1, " %s_vap, ",spe_names[i]);
  fprintf(fp1, " speed-of-sound");

  fprintf(fp2, "NIST calculations questionable or not possible for the following given inputs\n");
  fprintf(fp2, "\nPlease make sure the inputs are within permitted limits set by NIST!");
  fprintf(fp2, "\nTemperature   Pressure   Composition: (Mass Fractions) ");
  fprintf(fp2, "\n(K)  (Pa) ");
  for (i=0; i<ncomps; i++)
```

```
      fprintf(fp2, " %s ",spe_names[i]);

   for (k=0; k<N_DATA; k++)
   {
     t = temp[k];
     p = press[k];

     get_prop_NIST_msp(t,p,z,prop); /* call NIST property calculations */

     q = prop[NIST_q_molar]; /* error handling */
     if (q > -2999.1 && q < -2998.9)
     {
        fprintf(fp2, "\n%5.4e %5.4e ",t,p);
        for (i=0; i<ncomps; i++)
   fprintf(fp2, " %5.4e ",z[i]);
        continue;
     }

   Message0("\n\ngetprop_msp: T=%f (K), P=%e (Bar) ",t,p/1.e5);
   Message0("\n\nquality (mol/mol)=%f  quality (kg/kg)=%f quality (vol/vol)=%f\n", prop[NIST_q_molar],
            prop[NIST_q_mass],prop[NIST_q_vol]);
   Message0("\n                         Bulk           Liquid-Phase           Vapor-Phase\n");
   Message0("\ndensity (kg/m^3):        %f        %f              %f\n",prop[NIST_den_m],
            prop[NIST_den_l],prop[NIST_den_v]);
   Message0("\nspecific-heat (kj/kg-k):  %f         %f               %f\n",prop[NIST_cp_m]/1.e3,
            prop[NIST_cp_l]/1.e3,prop[NIST_cp_v]/1.e3);
   Message0("\nthermal-ktc (mk/m-k):    %f         %f             %f\n",prop[NIST_ktc_m]*1.e3,
            prop[NIST_ktc_l]*1.e3,prop[NIST_ktc_v]*1.e3);
   Message0("\nviscosity (uPa-s):        %f         %f               %f\n",prop[NIST_mu_m]*1.e6,
            prop[NIST_mu_l]*1.e6,prop[NIST_mu_v]*1.e6);
   Message0("\nenthalpy (kj/kg):         %f        %f              %f\n",prop[NIST_h_m]/1.e3,
            prop[NIST_h_l]/1.e3,prop[NIST_h_v]/1.e3);
   Message0("\nspeed of sound (m/s):    %f        %f              %f\n",prop[NIST_a_m],
            prop[NIST_a_l],prop[NIST_a_v]);

   Message0("\nMass fractions in liquid phase: ");
   for (i=0; i<ncomps; i++)
     Message0("%s : %f  ",spe_names[i],prop[NIST_massfrac0_l+i]);
   Message0("\nMass fractions in vapor phase:  ");
   for (i=0; i<ncomps; i++)
     Message0("%s : %f  ",spe_names[i],prop[NIST_massfrac0_v+i]);

   /* quality */
   fprintf(fp1, "\n%5.4f %5.4f %5.4f ", prop[NIST_q_molar],prop[NIST_q_mass],prop[NIST_q_vol]);
   /* density (kg/m^3) */
   fprintf(fp1, "%5.4f %5.4f %5.4f ",prop[NIST_den_m],prop[NIST_den_l],prop[NIST_den_v]);
   /* enthalpy (KJ/Kg) */
   fprintf(fp1, "%5.4f %5.4f %5.4f ",prop[NIST_h_m]/1.e3,prop[NIST_h_l]/1.e3,prop[NIST_h_v]/1.e3);
   /* specific-heat cp (Kj/Kg-k) */
   fprintf(fp1, "%5.4f %5.4f %5.4f ",prop[NIST_cp_m]/1.e3,prop[NIST_cp_l]/1.e3,prop[NIST_cp_v]/1.e3);
   /* thermal conductivity (mw/m-k) */
   fprintf(fp1, "%5.4f %5.4f %5.4f ",prop[NIST_ktc_m]*1.e3,prop[NIST_ktc_l]*1.e3,prop[NIST_ktc_v]*1.e3);
   /* viscosity (uPa-s) */
   fprintf(fp1, "%5.4f %5.4f %5.4f ",prop[NIST_mu_m]*1.e6,prop[NIST_mu_l]*1.e6,prop[NIST_mu_v]*1.e6);

   /* Mass fractions in liquid phase */
   for (i=0; i<ncomps; i++)
     fprintf(fp1, "%5.4f ",prop[NIST_massfrac0_l+i]);
   /* Mass fractions in vapor phase */
   for (i=0; i<ncomps; i++)
     fprintf(fp1, "%5.4f ",prop[NIST_massfrac0_v+i]);

   /* speed of sound (m/s) */
   fprintf(fp1, "%5.4f %5.4f %5.4f ",prop[NIST_a_m],prop[NIST_a_l],prop[NIST_a_v]);
   }

   fclose(fp1);
   fclose(fp2);
}
```

3.2.11. User-Defined Scalar (UDS) Transport Equation Macros

This section contains macros that you can use when defining scalar transport UDFs in Ansys Fluent. Note that if you try to use the macros listed below (for example, F_UDSI, C_UDSI) before you have specified user-defined scalars in your Ansys Fluent model (in the **User-Defined Scalars** dialog box), then an error will result.

3.2.11.1. `Set_User_Scalar_Name`

Ansys Fluent assigns a default name for every user-defined scalar that you allocate in the graphical user-interface. For example, if you specify 2 as the `Number of User-Defined Scalars`, then two variables with default names `User Scalar 0` and `User Scalar 1` will be defined and the variables with these default names will appear in setup and postprocessing dialog boxes. You can change the default names if you want, using `Set_User_Scalar_Name` as described below.

The default name that appears in the graphical user interface and on plots in Ansys Fluent for user-defined scalars (for example, `User Scalar 0`) can now be changed using the function `Set_User_Scalar_Name`.

```
void Set_User_Scalar_Name(int i,char *name);
```

`i` is the index of the scalar and `name` is a string containing the name you want to assign. It is defined in `sg_udms.h`.

`Set_User_Scalar_Name` should be used only once and is best used in an `EXECUTE_ON_LOADING` UDF (see `DEFINE_EXECUTE_ON_LOADING` (p. 28)). Due to the mechanism used, UDS variables cannot be renamed after they have been set, so if the name is changed in a UDF, for example, and the UDF library is reloaded, then the old name could remain. In this case, restart Ansys Fluent and load the library again.

3.2.11.2. `F_UDSI`

You can use `F_UDSI` when you want to access face variables that are computed for user-defined scalar transport equations (Table 3.37: Accessing User-Defined Scalar Face Variables (mem.h) (p. 354)). See Example UDF that Utilizes UDM and UDS Variables (p. 360) for an example of `F_UDSI` usage.

Table 3.37: Accessing User-Defined Scalar Face Variables (`mem.h`)

Macro	Argument Types	Returns
F_UDSI(f,t,i)	face_t f,Thread *t, int i Note: i is index of scalar	UDS face variables

Important:

Note that `F_UDSI` is available for wall and flow boundary faces, only. If a UDS attempts to access any other face zone, then an error will result.

3.2.11.3. `C_UDSI`

You can use `C_UDSI` when you want to access cell variables that are computed for user-defined scalar transport equations. Macros for accessing UDS cell variables are listed in Table 3.38: `C_UDSI` for Accessing UDS Transport Cell Variables (`mem.h`) (p. 355). Some examples of usage for these macros include defining non-constant source terms for UDS transport equations and initializing equations. See Example UDF that Utilizes UDM and UDS Variables (p. 360) for an example of `C_UDSI` usage.

Table 3.38: `C_UDSI` for Accessing UDS Transport Cell Variables (`mem.h`)

Macro	Argument Types	Returns
`C_UDSI(c,t,i)`	`cell_t c, Thread *t, int i`	UDS cell variables
`C_UDSI_G(c,t,i)`	`cell_t c, Thread *t, int i`	UDS gradient
`C_UDSI_M1(c,t,i)`	`cell_t c, Thread *t, int i`	UDS previous time step
`C_UDSI_M2(c,t,i)`	`cell_t c, Thread *t, int i`	UDS second previous time step
`C_UDSI_DIFF(c,t,i)`	`cell_t c, Thread *t, int i` Note: `i` is index of scalar	UDS diffusivity

3.2.11.4. Reserving UDS Variables

Prior to use, you can reserve scalar variables to avoid data conflicts between multiple UDF libraries using the same user-defined scalars (see `Reserve_User_Scalar_Vars` (p. 355)).

3.2.11.5. `Reserve_User_Scalar_Vars`

The new capability of loading more than one UDF library into Ansys Fluent raises the possibility of user-defined scalar (UDS) clashes. To avoid data contention between multiple UDF libraries using the same user-defined scalars, Ansys Fluent has provided the macro `Reserve_User_Scalar_Vars` that allows you to reserve scalars prior to use.

```
int Reserve_User_Scalar_Vars(int num)
```

`int num` is the number of user-defined scalars that the library uses. The integer returned is the lowest UDS index that the library may use. After calling:

```
offset = Reserve_User_Scalar_Vars(int num);
```

the library may safely use `C_UDSI(c,t,offset)` to `C_UDSI(c,t,offset+num-1)`. See `DEFINE_EXECUTE_ON_LOADING` (p. 28) for an example of macro usage. Note that there are other methods you can use within UDFs to hardcode the offset to prevent data contention.

`Reserve_User_Scalar_Vars` (defined in `sg_udms.h`) is designed to be called from an EX-ECUTE_ON_LOADING UDF (`DEFINE_EXECUTE_ON_LOADING` (p. 28)). An on-loading UDF, as

its name implies, executes as soon as the shared library is loaded into Ansys Fluent. The macro can also be called from an `INIT` or `ON_DEMAND` UDF. After a user scalar has been reserved, it can be set to unique names for the particular library using `Set_User_Memory_Name` (see below for details on `Set_User_Memory_Name`). After the number of UDS that are needed by a particular library is set in the GUI and the variables are successfully reserved for the loaded library, the other functions in the library can safely use `C_UDMI(c,t,offset)` up to `C_UDMI(c,t,offset+num-1)` to store values in user scalars without interference.

3.2.11.6. Unreserving UDS Variables

Ansys Fluent does not currently provide the capability to unreserve UDS variables using a macro. `Unreserve` macros will be available in future versions of Ansys Fluent.
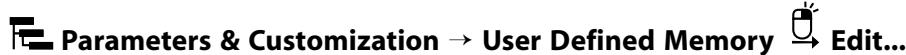
3.2.11.7. `N_UDS`

You can use `N_UDS` to access the number of user-defined scalar (UDS) transport equations that have been specified in Ansys Fluent. The macro takes no arguments and returns the integer number of equations. It is defined in `models.h`.

3.2.12. User-Defined Memory (UDM) Macros

This section contains macros that access user-defined memory (UDM) and user-defined node memory (UDNM) variables in Ansys Fluent.

Before you can store variables in memory using the macros provided below, you will first need to allocate the appropriate number of memory location(s) in the **User-Defined Memory** dialog box in Ansys Fluent.

Parameters & Customization → **User Defined Memory** 🖰 **Edit...**

Important:

Note that if you try to use `F_UDMI`, `C_UDMI`, or `N_UDMI` before you have allocated memory, then an error will result.

A variable will be created for every user-defined memory and user-defined node memory location that you allocate in the graphical user interface. For example, if you specify 2 as the `Number of User-Defined Memory Locations` or `Number of User-Defined Node Memory Locations`, then two variables with default names `User Memory 0` and `User Memory 1`, or `User Node Memory 0` and `User Node Memory 1` will be defined for your model and the default variable names will appear in postprocessing dialog boxes. You can change the default names if you want, using `Set_User_Memory_Name` or `Set_User_Node_Memory_Name` as described below.

Important:

The total number of memory locations is limited to 500. For large numbers of memory locations, system memory requirements will increase.

3.2.12.1. `Set_User_Memory_Name`

The default name that appears in the graphical user interface and on plots for user-defined memory (UDM) values in Ansys Fluent (for example, `User Memory 0`) can be changed using the function `Set_User_Memory_Name`.

```
void Set_User_Memory_Name(int i,char *name);
```

`i` is the index of the memory value and `name` is a string containing the name you want to assign. It is defined in `sg_udms.h`.

The `Set_User_Memory_Name` function should be used only once and it is best used in an `EX-ECUTE_ON_LOADING` UDF (see `DEFINE_EXECUTE_ON_LOADING` (p. 28)). Due to the mechanism used, `User Memory` values cannot be renamed after they have been set, so if the name is changed in a UDF, for example, and the UDF library is reloaded, then the old name could remain. In this case, restart Ansys Fluent and load the library again.

3.2.12.2. `Set_User_Node_Memory_Name`

The default name that appears in the graphical user interface and on plots for user-defined node memory values in Ansys Fluent (for example, `User Node Memory 0`) can be changed using the function `Set_User_Node_Memory_Name`.

```
void Set_User_Node_Memory_Name(int i,char *name);
```

`i` is the index of the memory value and `name` is a string containing the name you want to assign. It is defined in `sg_udms.h`.

The `Set_User_Node_Memory_Name` function should be used only once and is best used in an `EXECUTE_ON_LOADING` UDF. Due to the mechanism used, `User Memory` values cannot be renamed after they have been set, so if the name is changed in a UDF, for example, and the UDF library is reloaded, then the old name could remain. In this case, restart Ansys Fluent and load the library again.

3.2.12.3. `F_UDMI`

You can use `F_UDMI` (Table 3.39: Storage of User-Defined Memory on Faces (`mem.h`) (p. 358)) to access or store the value of the user-defined memory on a face. `F_UDMI` can be used to allocate

up to 500 memory locations in order to store and retrieve the values of face field variables computed by UDFs. These stored values can then be used by other UDFs.

Important:

Note that `F_UDMI` is available for wall and flow boundary faces, only.

Table 3.39: Storage of User-Defined Memory on Faces (`mem.h`)

Macro	Argument Types	Usage
`F_UDMI(f,t,i)`	`face_t f, Thread *t, int i`	stores the face value of a user- defined memory with index `i`

There are three arguments to `F_UDMI`: `f`, `t`, and `i`. `f` is the face identifier, `t` is a pointer to the face thread, and `i` is an integer index that identifies the memory location where data is to be stored. An index `i` of `0` corresponds to user-defined memory location `0` (or `User Memory 0`).

Example

```
/* Compute face temperature and store in user-defined memory */
begin_f_loop(f,t)
  {
    temp = F_T(f,t);
    F_UDMI(f,t,0) = (temp - tmin) / (tmax-tmin);
  }
end_f_loop(f,t)
  }
```

See DEFINE_DPM_EROSION (p. 213) for another example of `F_UDMI` usage.

Postprocessing `F_UDMI`

While you cannot plot contours of `F_UDMI` directly, you can store the values from `F_UDMI` in a `C_UDMI` and then create a contour plot.

A UDF to transfer values from `F_UDMI` to `C_UDMI` could look like:

```
begin_f_loop(f, f_thread)
{

c0 = F_C0(f,f_thread); /* Get the cell id of the cell adjacent to the face*/
t0 = F_C0_THREAD(f,f_thread); /* Get the Thread id of the cells adjacent to the face*/
C_UDMI(c0,t0,0)=F_UDMI(f,f_thread,0); /*Store the F_UDMI into cell UDMI for graphical visualization*/

}
end_f_loop(f,f_thread)
```

Note:

The above UDF assumes that `F_UDMI` is defined and already has values assigned to it.

3.2.12.4. `C_UDMI`

You can use `C_UDMI` to access or store the value of the user-defined memory in a cell. `C_UDMI` can be used to allocate up to 500 memory locations in order to store and retrieve the values of cell field variables computed by UDFs (Table 3.40: Storage of User-Defined Memory in Cells (`mem.h`) (p. 359)). These stored values can then be used for postprocessing, for example, or by other UDFs. See Example UDF that Utilizes UDM and UDS Variables (p. 360) for an example of `C_UDMI` usage.

Table 3.40: Storage of User-Defined Memory in Cells (`mem.h`)

Macro	Argument Types	Usage
`C_UDMI(c,t,i)`	`cell_t c, Thread *t, int i`	stores the cell value of a user- defined memory with index `i`

There are three arguments to `C_UDMI`: `c`, `thread`, and `i`. `c` is the cell identifier, `thread` is a pointer to the cell thread, and `i` is an `integer` index that identifies the memory location where data is to be stored. An index `i` of `0` corresponds to user-defined memory location `0` (or `User Memory 0`).

3.2.12.5. `N_UDMI`

You can use `N_UDMI` to access or store the value of the user-defined memory in a mesh node. `N_UDMI` can be used to allocate up to 500 memory locations (Table 3.41: Storage of User-Defined Memory at Mesh Nodes (`mem.h`) (p. 359)). These stored values can then be used for postprocessing, for example, or by other UDFs.

Table 3.41: Storage of User-Defined Memory at Mesh Nodes (`mem.h`)

Macro	Argument Types	Usage
`N_UDMI(v,i)`	`Node *v, int i`	stores the node value of a user- defined memory with index `i`

There are two arguments to `N_UDMI`: `v`, and `i`. `v` is a pointer to the mesh node, and `i` is an `integer` index that identifies the memory location where data is to be stored. An index `i` of `0` corresponds to user-defined memory location `0` (or `User Node Memory 0`).

Example

```
/* Store the mesh node coordinates in user-defined node memory */
thread_loop_c (t,domain)
 {
 begin_c_loop (c,t)
  {
  c_node_loop (c,t,n)
   {
   v = C_NODE(c,t,n);
   N_UDMI(v,0) = NODE_X(v);
   N_UDMI(v,1) = NODE_Y(v);
#if RP_3D
   N_UDMI(v,2) = NODE_Z(v);
#endif
   }
  }
```

```
    end_c_loop (c,t)
    }
```

3.2.12.6. Example UDF that Utilizes UDM and UDS Variables

UDMs are often used to store diagnostic values derived from calculated values of a UDS. Below is an example that shows a technique for plotting the gradient of any flow variable. In this case, the volume fraction of a phase is loaded into a user scalar. If an iteration is made such that the UDS is not calculated, the gradients of the scalar will nevertheless be updated without altering the values of the user scalar. The gradient is then available to be copied into a User Memory variable for displaying.

```
# include "udf.h"
# define domain_ID 2

DEFINE_ADJUST(adjust_gradient, domain)
{
    Thread *t;
    cell_t c;
    face_t f;
    domain = Get_Domain(domain_ID);
    /* Fill UDS with the variable. */
    thread_loop_c (t,domain)
        {
        begin_c_loop (c,t)
            {
            C_UDSI(c,t,0) = C_VOF(c,t);
            }
        end_c_loop (c,t)
        }
    thread_loop_f (t,domain)
        {
        if (THREAD_STORAGE(t,SV_UDS_I(0))!=NULL)
          begin_f_loop (f,t)
            {
            F_UDSI(f,t,0) = F_VOF(f,t);
            }
          end_f_loop (f,t)
        }
}

DEFINE_ON_DEMAND(store_gradient)
{   Domain *domain;
    cell_t c;
    Thread *t;
    domain=Get_Domain(1);
    /* Fill the UDM with magnitude of gradient. */
    thread_loop_c (t,domain)
        {
        begin_c_loop (c,t)
          {
            C_UDMI(c,t,0) = NV_MAG(C_UDSI_G(c,t,0));
          }
        end_c_loop (c,t)
        }
}
```

3.2.12.7. Reserving UDM Variables Using `Reserve_User_Memory_Vars`

The capability of loading more than one UDF library into Ansys Fluent raises the possibility of user-defined memory (UDM) clashes. If, for example, you want to use one UDF library that has a fixed 2D magnetic field stored in `User Memory 0` and `User Memory 1` and you want to use another UDF library that models the mass exchange between phases using `User Memory 0` for the ex-

change rates and these two libraries are loaded at the same time, then the two models are going to interfere with each other's data in `User Memory 0`. To avoid data contention problems, Ansys Fluent has a macro that will allow a UDF library to "reserve" UDM locations prior to usage. Note that there are other methods you can use within UDFs to hardcode the offset for UDMs to prevent contention that are not discussed here.

```
int Reserve_User_Memory_Vars(int num)
```

The integer given as an argument to the macro (`num`) specifies the number of UDMs needed by the library. The integer returned by the function is the starting point or "offset" from which the library may use the UDMs. It should be saved as a global integer such as `offset` in the UDF and it should be initialized to the special variable `UDM_UNRESERVED`.

```
offset = Reserve_User_Memory_Vars(int num);
```

`Reserve_User_Memory_Vars` (defined in `sg_udms.h`) is designed to be called from an EX-ECUTE_ON_LOADING UDF (DEFINE_EXECUTE_ON_LOADING (p. 28)). An on-loading UDF, as its name implies, executes as soon as the shared library is loaded into Ansys Fluent. The macro can also be called from an `INIT` or `ON_DEMAND` UDF, although this is discouraged except for testing purposes. After a UDM is reserved, it can be set to unique names for the particular library using `Set_User_Memory_Name` (see below for details.) After the number of UDMs that are needed by a particular library is set in the GUI and the UDMs are successfully reserved for the loaded library, the other functions in the library can safely use `C_UDMI(c,t,offset)` up to `C_UDMI(c,t,offset+num-1)` to store values in memory locations without interference. Two example source code files named `udm_res1.c` and `udm_res2.c` each containing two UDFs are listed below. The first UDF is an `EXECUTE_ON_LOADING` UDF that is used to reserve UDMs for the library and set unique names for the UDM locations so that they can be easily identified in postprocessing. The second UDF is an `ON_DEMAND` UDF that is used to set the values of the UDM locations after the solution has been initialized. The `ON_DEMAND` UDF sets the initial values of the UDM locations using `udf_offset`, which is defined in the `EXECUTE_ON_LOADING` UDF. Note that the on demand UDF must be executed *after* the solution is initialized to reset the initial values for the UDMs.

The following describes the process of reserving five UDMs for two libraries named `libudf` and `libudf2`.

1. In the **User-Defined Memory** dialog box, specify **5** for the **Number of User-Defined Memory Locations**.

2. In the **Compiled UDFs** dialog box, build the compiled library named `libudf` for `udm_res1.c` and load the library.

3. Build the compiled library for `udm_res2.c` named `libudf2` and load the library.

4. Initialize the solution.

5. Execute the `on-demand` UDFs for `libudf` and `libudf2` in the **Execute On Demand** dialog box.

6. Iterate the solution.

7. Postprocess the results.

3.2.12.8. Example 1

```
/************************************************************************
  udm_res1.c contains two UDFs: an execute on loading UDF that reserves
  three UDMs for libudf and renames the UDMs to enhance postprocessing,
  and an on-demand UDF that sets the initial value of the UDMs.
************************************************************************/
#include "udf.h"
#define NUM_UDM 3
static int udm_offset = UDM_UNRESERVED;

DEFINE_EXECUTE_ON_LOADING(on_loading, libname)
{
    if (udm_offset == UDM_UNRESERVED) udm_offset =
            Reserve_User_Memory_Vars(NUM_UDM);
    if (udm_offset == UDM_UNRESERVED)
        Message("\nYou need to define up to %d extra UDMs in GUI and "
                "then reload current library %s\n", NUM_UDM, libname);
    else
    {
        Message("%d UDMs have been reserved by the current "
                "library %s\n", NUM_UDM, libname);
        Set_User_Memory_Name(udm_offset,"lib1-UDM-0");
        Set_User_Memory_Name(udm_offset+1,"lib1-UDM-1");
        Set_User_Memory_Name(udm_offset+2,"lib1-UDM-2");
    }
    Message("\nUDM Offset for Current Loaded Library = %d",udm_offset);
}

DEFINE_ON_DEMAND(set_udms)
{
    Domain *d;
    Thread *ct;
    cell_t c;
    int i;
    d=Get_Domain(1);
    if(udm_offset != UDM_UNRESERVED)
    {
        Message("Setting UDMs\n");
        for (i=0; i<NUM_UDM; i++)
        {
            thread_loop_c(ct,d)
            {
                begin_c_loop(c,ct)
                {
                    C_UDMI(c,ct,udm_offset+i)=3.0+i/10.0;
                }
                end_c_loop(c,ct)
            }
        }
    }
    else
        Message("UDMs have not yet been reserved for library 1\n");
}
```

3.2.12.9. Example 2

```
/************************************************************************
udm_res2.c contains two UDFs:
an execute on loading UDF that reserves
two UDMs for libudf and renames the UDMs to enhance postprocessing,
and an on-demand UDF that sets the initial value of the UDMs.
************************************************************************/
#include "udf.h"
#define NUM_UDM 2
static int udm_offset = UDM_UNRESERVED;

DEFINE_EXECUTE_ON_LOADING(on_loading, libname)
```

```
{
    if (udm_offset == UDM_UNRESERVED) udm_offset =
            Reserve_User_Memory_Vars(NUM_UDM);
    if (udm_offset == UDM_UNRESERVED)
        Message("\nYou need to define up to %d extra UDMs in GUI and "
                "then reload current library %s\n", NUM_UDM, libname);
    else
    {
        Message("%d UDMs have been reserved by the current "
                "library %s\n", NUM_UDM, libname);
        Set_User_Memory_Name(udm_offset,"lib2-UDM-0");
        Set_User_Memory_Name(udm_offset+1,"lib2-UDM-1");
    }
    Message("\nUDM Offset for Current Loaded Library = %d",udm_offset);
}

DEFINE_ON_DEMAND(set_udms)
{
    Domain *d;
    Thread *ct;
    cell_t c;
    int i;
    d=Get_Domain(1);
    if(udm_offset != UDM_UNRESERVED)
    {
        Message("Setting UDMs\n");
        for (i=0; i<NUM_UDM; i++)
        {
            thread_loop_c(ct,d)
            {
                begin_c_loop(c,ct)
                {
                    C_UDMI(c,ct,udm_offset+i)=2.0+i/10.0;
                }
                end_c_loop(c,ct)
            }
        }
    }
    else
        Message("UDMs have not yet been reserved for library 1\n");
}
```

If your model uses a number of UDMs, it may be useful to define your variables in an easy-to-read format, either at the top of the source file or in a separate header file using the preprocessor #define directive:

```
#define C_MAG_X(c,t)C_UDMI(c,t,udm_offset)
#define C_MAG_Y(c,t)C_UDMI(c,t,udm_offset+1)
```

Following this definition, in the remainder of your UDF you can simply use C_MAG_X(c,t) and C_MAG_Y(c,t) to specify the fixed magnetic field components.

3.2.12.10. Unreserving UDM Variables

Ansys Fluent does not currently provide the capability to unreserve UDM variables using a macro. Unreserve macros will be available in future versions of Ansys Fluent. You will need to exit Ansys Fluent to ensure that all UDM variables are reset.

3.3. Looping Macros

Many UDF tasks require repeated operations to be performed on nodes, cells, and threads in a computational domain. For your convenience, Ansys Fluent has provided you with a set of predefined macros to accomplish looping tasks. For example, to define a custom boundary profile function you will need to loop over all the faces in a face thread using `begin..end_f_loop` looping macros. For operations where you want to loop over all the faces or cells in a domain, you will need to nest a `begin..end_f_loop` or `begin..end_c_loop` inside a `thread_loop_f` or `thread_loop_c`, respectively.

The following general looping macros can be used for UDFs in single-phase or multiphase models in Ansys Fluent. Definitions for these macros are contained in the `mem.h` header file.

Important:

You should not access a Scheme variable using any of the `RP_GET_...` functions from inside a cell or face looping macro (c_loop or f_loop). This type of communication between the solver and cortex is very time consuming and therefore should be done outside of loops.

For more information, see the following sections:

3.3.1. Looping Over Cell Threads in a Domain (`thread_loop_c`)

You can use `thread_loop_c` when you want to loop over all cell threads in a given domain. It consists of a single statement, followed by the operation(s) to be performed on all cell threads in the domain enclosed within braces   as shown below. Note that `thread_loop_c` is similar in implementation to the `thread_loop_f` macro described below.

```
Domain *domain;
Thread *c_thread;
thread_loop_c(c_thread, domain) /*loops over all cell threads in domain*/
  {
  }
```

3.3.2. Looping Over Face Threads in a Domain (`thread_loop_f`)

You can use `thread_loop_f` when you want to loop over all face threads in a given domain. It consists of a single statement, followed by the operation(s) to be performed on all face threads in

the domain enclosed within braces   as shown below. Note that `thread_loop_f` is similar in implementation to the `thread_loop_c` macro described above.

```
Thread *f_thread;
Domain *domain;
thread_loop_f(f_thread, domain)/* loops over all face threads in a domain*/
  {
  }
```

3.3.3. Looping Over Cells in a Cell Thread (`begin...end_c_loop`)

You can use `begin_c_loop` and `end_c_loop` when you want to loop over all cells in a given cell thread. It contains a `begin` and `end` loop statement, and performs operation(s) on each cell in the cell thread as defined between the braces   . This loop is usually nested within `thread_loop_c` when you want to loop over all cells in all cell threads in a domain.

```
cell_t c;
Thread *c_thread;
begin_c_loop(c, c_thread) /* loops over cells in a cell thread */
  {
  }
end_c_loop(c, c_thread)
```

Example

```
/* Loop over cells in a thread to get information stored in cells. */
 begin_c_loop(c, c_thread)
 {
 /* C_T gets cell temperature. The += will cause all of the cell
  temperatures to be added together. */
  temp += C_T(c, c_thread);
 }
 end_c_loop(c, c_thread)
}
```

3.3.4. Looping Over Faces in a Face Thread (`begin...end_f_loop`)

You can use `begin_f_loop` and `end_f_loop` when you want to loop over all faces in a given face thread. It contains a `begin` and `end` loop statement, and performs operation(s) on each face in the face thread as defined between the braces   . This loop is usually nested within `thread_loop_f` when you want to loop over all faces in all face threads in a domain.

```
face_t f;
Thread *f_thread;
begin_f_loop(f, f_thread) /* loops over faces in a face thread */
  {
  }
end_f_loop(f, f_thread)
```

Example

```
/* Loop over faces in a face thread to get the information stored on faces. */
 begin_f_loop(f, f_thread)
 {
  /* F_T gets face temperature. The += will cause all of the face
   temperatures to be added together. */
  temp += F_T(f, f_thread);
 }
 end_f_loop(f, f_thread)
```

3.3.5. Looping Over Faces of a Cell (`c_face_loop`)

The following looping function loops over all faces of a given cell. It consists of a single loop statement, followed by the action to be taken in braces.

```
cell_t c;
Thread *t;
face_t f;
Thread *tf;
int n;
c_face_loop(c, t, n)    /* loops over all faces of a cell */
 {
 .
 .
 .
 f = C_FACE(c,t,n);
 tf = C_FACE_THREAD(c,t,n);
 .
 .
 .
 }
```

The argument `n` is the local face index number. The local face index number is used in the `C_FACE` macro to obtain the global face number (for example, `f = C_FACE(c,t,n)`).

Another useful macro that is often used in `c_face_loop` is `C_FACE_THREAD`. This macro is used to reference the associated face thread (for example, `tf = C_FACE_THREAD(c,t,n)`).

Refer to Miscellaneous Macros (p. 387) for other macros that are associated with `c_face_loop`.

3.3.6. Looping Over Nodes of a Cell (`c_node_loop`)

`c_node_loop(c,t,n)` is a function that loops over all nodes of a given cell. It consists of a single loop statement, followed by the action to be taken in braces { }.

Example:

```
cell_t c;
Thread *t;
int n;
Node *node;
c_node_loop(c,t,n)
 {
 .
 .
 node = C_NODE(c,t,n);
 .
 .
 }
```

Here, `n` is the local node index number. The index number can be used with the `C_NODE` macro to obtain the global cell node number (for example, `node = C_NODE(c,t,n)`).

3.3.7. Looping Over Nodes of a Face (`f_node_loop`)

`f_node_loop(f,t,n)` is a function that loops over all nodes of a given face. It consists of a single loop statement, followed by the action to be taken in braces { }.

Example

```
face_t f;
Thread *t;
int n;
Node *node;
f_node_loop(f,t,n)
 {
 .
 .
 .
 node = F_NODE(f,t,n);
 .
 .

 .
 }
```

Here, n is the local node index number. The index number can be used with the F_NODE macro to obtain the global face node number (for example, node = F_NODE(f,t,n)).

See DEFINE_GRID_MOTION for an example of a UDF that uses f_node_loop.

3.3.8. Overset Mesh Looping Macros

This section contains descriptions of looping macros that can be used for cases that have overset meshes. For further information about overset meshes, see Overset Meshes in the User's Guide.

3.3.8.1. Looping Over Overset Interface Cell Threads (`thread_loop_overset_c`)

You can use thread_loop_overset_c when you want to loop over only the cell threads participating in the overset interfaces for a given domain (as opposed to the thread_loop_c function, which is for operations that need to be performed across all cell threads in the domain). It consists of a single statement, followed by the operations (enclosed within parentheses, as shown in the example that follows) to be performed on all the cell threads related to the overset interfaces in the domain.

```
 Domain *domain;
 Thread *c_thread;
 thread_loop_overset_c(c_thread, domain) /*loops over all cell threads participating in overset */
                                         /*interfaces in the domain                           */
  {
  }
```

3.3.8.2. Looping Over Active Overset Cells in a Cell Thread (`begin...end_c_loop_active`, `begin...end_c_loop_solve`)

The cells in a domain with overset meshes may be classified into two broad categories: active cells and dead (inactive) cells. This classification is important with regards to the solver discretization (gradient evaluations, sources, AMG system, and so on) and the solution variables. Active cells may further be divided into solve cells and receptor (interpolation) cells (see Overset Field Functions in the User's Guide).

Therefore, a generalization of begin_c_loop and end_c_loop for overset meshes is available. The two loops are:

- `begin_c_loop_active`

- `begin_c_loop_solve`

The active function loops over all cells except dead (inactive) cells. The solve loop is restricted to active cells where no interpolation is performed (that is, receptor cells are excluded). It is recommended that you use overset specific loops when global operations are performed, such as the evaluation of minimum or maximum values in the cell thread, or the summation over cell threads. The choice to include receptor cells in these evaluations will determine the choice of loops, and is left to your discretion.

> **Note:**
>
> Global summations over all active or solve cells in all cell threads may still introduce an error due to mesh overlap in an overset mesh. For details, see Overset Postprocessing Limitations.

You can use `begin_c_loop_active` and `end_c_loop_active` or `begin_c_loop_solve` and `end_c_loop_solve` when you want to loop over all active or solve cells, respectively, in a given cell thread. These functions contain a begin and end loop statement, and perform operations (enclosed within parentheses, as shown in the example that follows) on each cell in the cell thread. Such loops are usually nested within `thread_loop_overset_c`, so that they loop over cells in only the cell threads participating in the overset interface in a domain; you may also nest them under `thread_loop_c`.

```
cell_t c;
Thread *c_thread;
begin_c_loop_active(c, c_thread) /* loops over solve + receptor cells in a cell thread, no dead cells */
  {
  }
end_c_loop_active(c, c_thread)

cell_t c;
Thread *c_thread;
begin_c_loop_solve(c, c_thread) /* loops over solve cells in a cell thread, no receptor or dead cells */
  {
  }
end_c_loop_solve(c, c_thread)
```

3.3.8.2.1. Example 1

```
/* Loop over solve cells in a thread and perform arithmetic averaging of Temperature. */
  real no_solve_cells;
  real avg_temp_solve=0.0;
  begin_c_loop_solve(c, c_thread)
  {
  /* C_T gets cell temperature. The += causes all cell temperatures to be added together. */
   avg_temp_solve += C_T(c, c_thread);
   no_solve_cells=no_solve_cells+1.0;
  }
  end_c_loop_solve(c, c_thread)
  }
 avg_temp_solve = avg_temp_solve/no_solve_cells;
```

3.3.8.2.2. Example 2

```
/* Loop over active cells (solve + receptor) in a thread and perform arithmetic averaging of Temperature. */
  real no_active_cells;
  real avg_temp_active=0.0;
```

```
  begin_c_loop_active(c, c_thread)
  {
  /* C_T gets cell temperature. The += causes all cell temperatures to be added together. */
   avg_temp_active += C_T(c, c_thread);
   no_cells_active = no_cells_active+1.0;
  }
  end_c_loop_active(c, c_thread)
 }

avg_temp_active = avg_temp_active/no_active_cells;
```
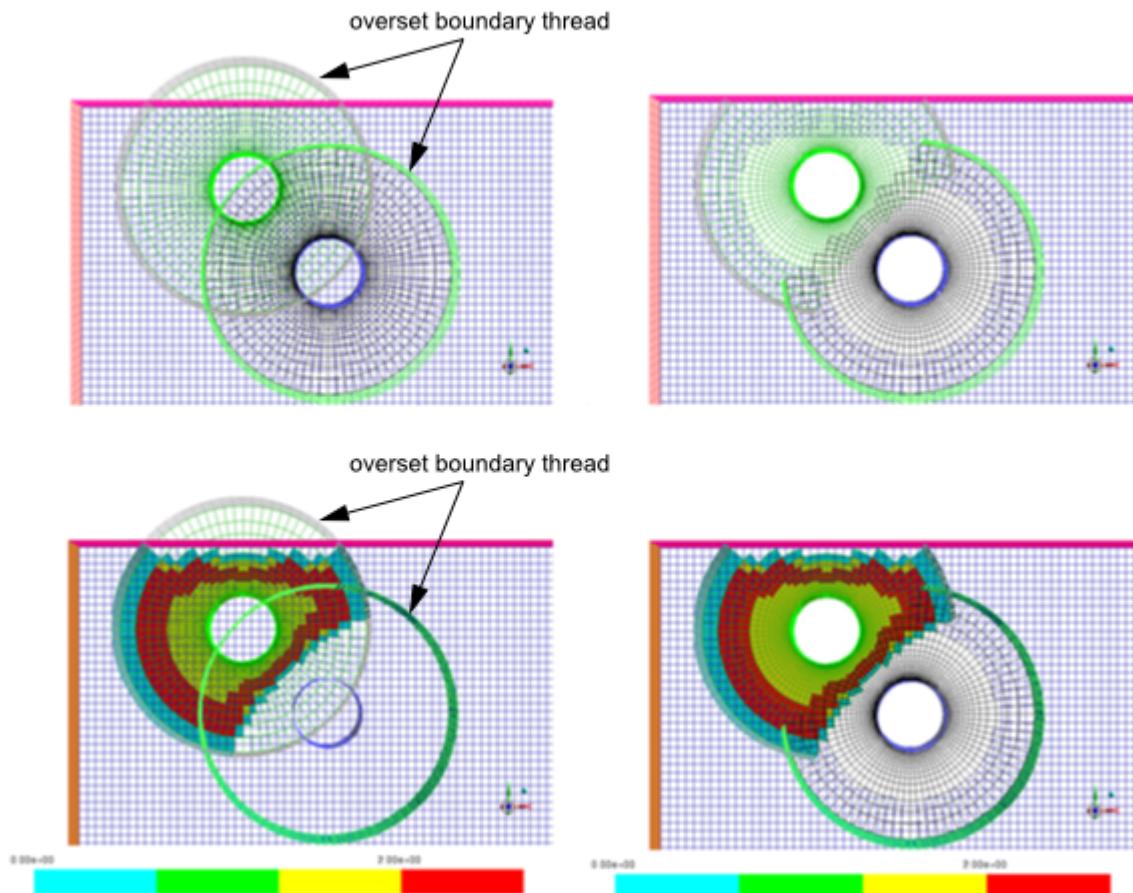
3.3.8.3. Looping Over Faces in a Face Thread with Overset Mesh (`begin...end_f_loop_active`)

Similar to the cell loops, you can use `begin_f_loop_active` and `end_f_loop_active` when you want to loop over all of the faces in a given face thread, excluding the overset interface and dead faces. The overset dead faces are defined as faces that lie adjacent to a dead cell for a boundary thread and between two dead cells when they are part of an interior face thread. The overset interface face is defined as a face that

- lies on an overset boundary thread adjacent to a receptor cell

- separates active cells from dead cells (that is, between receptor and dead cells)

Figure 3.3: Overset Interface Example

Figure 3.3: Overset Interface Example (p. 369) shows the overset interface before and after intersection (on the left and right side, respectively), so that the location of dead cells are apparent. The bottom row shows the overset cell type field contour plot for one of the component meshes (cyan = receptor, red = donor, yellow = solve). This figure illustrates that an overset interface face can lie either on the overset boundary thread or between receptor and dead cells.

This loop allows face-based operations from being adversely affected by dead or inactive cells. It contains a begin and end loop statement, and performs operations (enclosed within parentheses, as shown in the example that follows) on each face in the face thread. This loop is usually nested within `thread_loop_f`, so that it loops over all of the faces in all face threads in a domain.

```
face_t f;
Thread *f_thread;
begin_f_loop_active(f, f_thread) /* loops over all active faces in a face thread */
  {
  }
end_f_loop_active(f, f_thread)
```

3.3.8.3.1. Example

```
/* Loop over active faces in a face thread to get the information stored on faces. */
  real no_active_faces;
  real avg_temp_active=0.0;
  begin_f_loop_active(f, f_thread)
  {
   /* F_T gets face temperature. The += causes all face temperatures to be added together. */
   avg_temp_active += F_T(f, f_thread);
   no_active_faces = no_active_faces + 1.0;
  }
  end_f_loop_active(f, f_thread)

  avg_temp_active /= no_of_active_faces;
```

3.3.9. Multiphase Looping Macros

This section contains a description of looping macros that are to be used for multiphase UDFs only. They enable your function to loop over all cells and faces for given threads or domains. Refer to Multiphase-specific Data Types (p. 16) and, in particular, Figure 1.5: Domain and Thread Structure Hierarchy (p. 17) for a discussion on hierarchy of structures within Ansys Fluent.

3.3.9.1. Looping Over Phase Domains in Mixture (`sub_domain_loop`)

The `sub_domain_loop` macro loops over all phase domains (subdomains) within the mixture domain. The macro steps through and provides each phase domain pointer defined in the mixture domain as well as the corresponding `phase_domain_index`. As discussed in Multiphase-specific Data Types (p. 16), the domain pointer is needed, in part, to gain access to data within each phase. Note that `sub_domain_loop` is similar in implementation to the `sub_thread_loop` macro described below.

```
int phase_domain_index; /* index of subdomain pointers */
Domain *mixture_domain;
Domain *subdomain;
sub_domain_loop(subdomain, mixture_domain, phase_domain_index)
```

The variable arguments to `sub_domain_loop` are `subdomain`, `mixture_domain`, and `phase_domain_index`. `subdomain` is a pointer to the phase-level domain, and `mixture_do-`

main is a pointer to the mixture-level domain. The mixture_domain is automatically passed to your UDF by the Ansys Fluent solver when you use a DEFINE macro that contains a domain variable argument (for example, DEFINE_ADJUST) and your UDF is hooked to the mixture. If mixture_domain is not explicitly passed to your UDF, you will need to use another utility macro to retrieve it (for example, Get_Domain(1)) before calling sub_domain_loop (see Domain Pointer (Get_Domain) (p. 330)). phase_domain_index is an index of subdomain pointers. phase_domain_index is 0 for the primary phase, and is incremented by one for each secondary phase in the mixture. Note that subdomain and phase_domain_index are set within the sub_domain_loop macro.

Example

The following interpreted UDF patches an initial volume fraction for a particular phase in a solution. It is executed once at the beginning of the solution process. The function sets up a spherical volume centered at 0.5, 0.5, 0.5 with a radius of 0.25. A secondary-phase volume fraction of 1 is then patched to the cells within the spherical volume, while the volume fraction for the secondary phase in all other cells is set to 0.

```
/******************************************************************
   UDF for initializing phase volume fraction
 ******************************************************************/

#include "udf.h"

/* domain pointer that is passed by INIT function is mixture domain */
DEFINE_INIT(my_init_function, mixture_domain)
{
   int phase_domain_index;
   cell_t cell;
   Thread *cell_thread;
   Domain *subdomain;
   real xc[ND_ND];
   /* loop over all subdomains (phases) in the superdomain (mixture) */
   sub_domain_loop(subdomain, mixture_domain, phase_domain_index)
   {
   /* loop if secondary phase */
   if (DOMAIN_ID(subdomain) == 3)
   /* loop over all cell threads in the secondary phase domain */
   thread_loop_c (cell_thread,subdomain)
   {
    /* loop over all cells in secondary phase cell threads */
    begin_c_loop_all (cell,cell_thread)
    {
     C_CENTROID(xc,cell,cell_thread);
     if (sqrt(ND_SUM(pow(xc[0] - 0.5,2.),
         pow(xc[1] - 0.5,2.),
         pow(xc[2] - 0.5,2.))) < 0.25)
     /* set volume fraction to 1 for centroid */
     C_VOF(cell,cell_thread) = 1.;     else
     /* otherwise initialize to zero */
     C_VOF(cell,cell_thread) = 0.;
    }
    end_c_loop_all (cell,cell_thread)
   }
  }
 }
```

3.3.9.2. Looping Over Phase Threads in Mixture (`sub_thread_loop`)

The sub_thread_loop macro loops over all phase-level threads (subthreads) associated with a mixture-level thread. The macro steps through and returns the pointer to each subthread as well

as the corresponding `phase_domain_index`. As discussed in Multiphase-specific Data Types (p. 16), if the subthread pointer is associated with an inlet zone, then the macro will provide the pointers to the face threads associated with the inlet for each of the phases.

```
int phase_domain_index;
Thread *subthread;
Thread *mixture_thread;
sub_thread_loop(subthread, mixture_thread, phase_domain_index)
```

The variable arguments to `sub_thread_loop` are `subthread`, `mixture_thread`, and `phase_domain_index`. `subthread` is a pointer to the phase thread, and `mixture_thread` is a pointer to the mixture-level thread. The `mixture_thread` is automatically passed to your UDF by the Ansys Fluent solver when you use a `DEFINE` macro that contains a thread variable argument (for example, `DEFINE_PROFILE`) and your UDF is hooked to the mixture. If the `mixture_thread` is not explicitly passed to your UDF, you will need to use a utility macro to retrieve it before calling `sub_thread_loop`. `phase_domain_index` is an index of subdomain pointers that can be retrieved using the `PHASE_DOMAIN_INDEX` macro. (See Phase Domain Index (`PHASE_DOMAIN_INDEX`) (p. 376) 0 for the primary phase, and is incremented by one for each secondary phase in the mixture. Note that `subthread` and `phase_domain_index` are initialized within the `sub_thread_loop` macro definition.

3.3.9.3. Looping Over Phase Cell Threads in Mixture (`mp_thread_loop_c`)

The `mp_thread_loop_c` macro loops through all cell threads (at the mixture level) within the mixture domain and provides the pointers of the phase-level (cell) threads associated with each mixture-level thread. This is nearly identical to the `thread_loop_c` macro (Looping Over Cell Threads in a Domain (`thread_loop_c`) (p. 364)) when applied to the mixture domain. The difference is that, in addition to stepping through each cell thread, the macro also returns a pointer array (`pt`) that identifies the corresponding phase-level threads. The pointer to the cell thread for the ith phase is `pt[i]`, where `i` is the `phase_domain_index`. `pt[i]` can be used as an argument to macros requiring the phase-level thread pointer. `phase_domain_index` can be retrieved using the `PHASE_DOMAIN_INDEX` macro. (See Phase Domain Index (`PHASE_DOMAIN_INDEX`) (p. 376)

```
Thread **pt;
Thread *cell_threads;
Domain *mixture_domain;
mp_thread_loop_c(cell_threads, mixture_domain, pt)
```

The variable arguments to `mp_thread_loop_c` are `cell_threads`, `mixture_domain`, and `pt`. `cell_threads` is a pointer to the cell threads, and `mixture_domain` is a pointer to the mixture-level domain. `pt` is an array pointer whose elements contain pointers to phase-level threads.

`mixture_domain` is automatically passed to your UDF by the Ansys Fluent solver when you use a `DEFINE` macro that contains a domain variable argument (for example, `DEFINE_ADJUST`) and your UDF is hooked to the mixture. If `mixture_domain` is not explicitly passed to your UDF, you will need to use another utility macro to retrieve it (for example, `Get_Domain(1)`, described in Domain Pointer (`Get_Domain`) (p. 330)). Note that the values for `pt` and `cell_threads` are set within the looping function.

`mp_thread_loop_c` is typically used along with `begin_c_loop`. `begin_c_loop` loops over cells in a cell thread. When `begin_c_loop` is nested within `mp_thread_loop_c`, you can loop over all cells in all phase cell threads within a mixture.

3.3.9.4. Looping Over Phase Face Threads in Mixture (`mp_thread_loop_f`)

The `mp_thread_loop_f` macro loops through all face threads (at the mixture level) within the mixture domain and provides the pointers of the phase-level (face) threads associated with each mixture-level thread. This is nearly identical to the `thread_loop_f` macro when applied to the mixture domain. The difference is that, in addition to stepping through each face thread, the macro also returns a pointer array (`pt`) that identifies the corresponding phase-level threads. The pointer to the face thread for the ith phase is `pt[i]`, where i is the `phase_domain_index`. `pt[i]` can be used as an argument to macros requiring the phase-level thread pointer. The `phase_domain_index` can be retrieved using the `PHASE_DOMAIN_INDEX` macro. (See Phase Domain Index (`PHASE_DOMAIN_INDEX`) (p. 376) for details.)

```
Thread **pt;
Thread *face_threads;
Domain *mixture_domain;
mp_thread_loop_f(face_threads, mixture_domain, pt)
```

The variable arguments to `mp_thread_loop_f` are `face_threads`, `mixture_domain`, and `pt`. `face_threads` is a pointer to the face threads, and `mixture_domain` is a pointer to the mixture-level domain. `pt` is an array pointer whose elements contain pointers to phase-level threads.

`mixture_domain` is automatically passed to your UDF by the Ansys Fluent solver if you are using a `DEFINE` macro that contains a domain variable argument (for example, `DEFINE_ADJUST`) and your UDF is hooked to the mixture. If `mixture_domain` is not explicitly passed to your UDF, you may use another utility macro to retrieve it (for example, `Get_Domain(1)`, described in Domain Pointer (`Get_Domain`) (p. 330)). Note that the values for `pt` and `face_threads` are set within the looping function.

`mp_thread_loop_f` is typically used along with `begin_f_loop`. `begin_f_loop` loops over faces in a face thread. When `begin_f_loop` is nested within `mp_thread_loop_f`, you can loop over all faces in all phase face threads within a mixture.

3.3.10. Advanced Multiphase Macros

For most standard UDFs written for multiphase models (for example, source term, material property, profile functions), variables that your function needs (domain pointers, thread pointers, and so on) are passed directly to your UDF as arguments by the solver in the solution process. All you need to do is hook the UDF to your model and everything is taken care of. For example, if your multiphase UDF defines a custom profile for a particular boundary zone (using `DEFINE_PROFILE`) and is hooked to the appropriate phase or mixture in Ansys Fluent in the relevant boundary condition dialog box, then appropriate phase or mixture variables will be passed to your function by the solver at run time.

There may, however, be more complex functions you want to write that require a variable that is *not* directly passed through its arguments. `DEFINE_ADJUST` and `DEFINE_INIT` functions, for example, are passed mixture domain variables only. If a UDF requires a phase domain pointer, instead, then it will need to use macros presented in this section to retrieve it. `ON_DEMAND` UDFS are not directly passed any variables through their arguments. Consequently, any on demand function that requires access to phase or domain variables will also need to use macros presented in this section to retrieve them.

Recall that when you are writing UDFs for multiphase models, you will need to keep in mind the hierarchy of structures within Ansys Fluent (see Multiphase-specific Data Types (p. 16) for details).

The particular domain or thread structure that gets passed into your UDF from the solver depends on the DEFINE macro you are using, as well as the domain the function is hooked to (either through the graphical user interface, or hardwired in the code). As mentioned above, it also may depend on the multiphase model that you are using. Refer to Multiphase-specific Data Types (p. 16) and, in particular, Figure 1.5: Domain and Thread Structure Hierarchy (p. 17) for a discussion on hierarchy of structures within Ansys Fluent.

3.3.10.1. Phase Domain Pointer (`DOMAIN_SUB_DOMAIN`)

There are two ways you can get access to a specific phase (or subdomain) pointer within the mixture domain. You can use either the DOMAIN_SUB_DOMAIN macro (described below) or Get_Domain, which is described below.

DOMAIN_SUB_DOMAIN has two arguments: mixture_domain and phase_domain_index. The function returns the phase pointer subdomain for the given phase_domain_index. Note that DOMAIN_SUB_DOMAIN is similar in implementation to the THREAD_SUB_THREAD macro described in Phase-Level Thread Pointer (THREAD_SUB_THREAD) (p. 374).

```
int phase_domain_index = 0;    /* primary phase index is 0 */
Domain *mixture_domain;
Domain *subdomain = DOMAIN_SUB_DOMAIN(mixture_domain,phase_domain_index);
```

mixture_domain is a pointer to the mixture-level domain. It is automatically passed to your UDF by the Ansys Fluent solver when you use a DEFINE macro that contains a domain variable argument (for example, DEFINE_ADJUST) and your UDF is hooked to the mixture. Otherwise, if the mixture_domain is not explicitly passed to your UDF, you will need to use another utility macro to retrieve it (for example, Get_Domain(1)) before calling sub_domain_loop.

phase_domain_index is an index of subdomain pointers. It is an integer that starts with 0 for the primary phase and is incremented by one for each secondary phase. phase_domain_index is automatically passed to your UDF by the Ansys Fluent solver when you use a DEFINE macro that contains a phase domain index argument (DEFINE_EXCHANGE_PROPERTY, DEFINE_VEC-TOR_EXCHANGE_PROPERTY) and your UDF is hooked to a specific interaction phase. Otherwise, you will need to hard code the integer value of phase_domain_index to the DOMAIN_SUB_DO-MAIN macro. If your multiphase model has only two phases defined, then phase_domain_index is 0 for the primary phase, and 1 for the secondary phase. However, if you have more than one secondary phase defined for your multiphase model, you will need to use the PHASE_DOMAIN_IN-DEX utility to retrieve the corresponding phase_domain_index for the given domain. See Phase Domain Index (PHASE_DOMAIN_INDEX) (p. 376) for details.

3.3.10.2. Phase-Level Thread Pointer (`THREAD_SUB_THREAD`)

The THREAD_SUB_THREAD macro can be used to retrieve the phase-level thread (subthread) pointer, given the phase domain index. THREAD_SUB_THREAD has two arguments: mixture_thread and phase_domain_index.

The function returns the phase-level thread pointer for the given phase_domain_index. Note that THREAD_SUB_THREAD is similar in implementation to the DOMAIN_SUB_DOMAIN macro described in Phase Domain Pointer (DOMAIN_SUB_DOMAIN) (p. 374).

```
int phase_domain_index = 0;               /* primary phase index is 0    */
Thread *mixture_thread;                    /* mixture-level thread pointer */
Thread *subthread = THREAD_SUB_THREAD(mixture_thread,phase_domain_index);
```

`mixture_thread` is a pointer to a mixture-level thread. It is automatically passed to your UDF by the Ansys Fluent solver when you use a `DEFINE` macro that contains a variable thread argument (for example, `DEFINE_PROFILE`), and the function is hooked to the mixture. Otherwise, if the mixture thread pointer is not explicitly passed to your UDF, then you will need to use the `Lookup_Thread` utility macro to retrieve it (see Thread Pointer for Zone ID (`Lookup_Thread`) (p. 328)).

`phase_domain_index` is an index of subdomain pointers. It is an integer that starts with `0` for the primary phase and is incremented by one for each secondary phase. `phase_domain_index` is automatically passed to your UDF by the Ansys Fluent solver when you use a `DEFINE` macro that contains a phase domain index argument (`DEFINE_EXCHANGE_PROPERTY`, `DEFINE_VEC-TOR_EXCHANGE_PROPERTY`) and your UDF is hooked to a specific interaction phase. (See `DEFINE_EXCHANGE_PROPERTY` (p. 185) `phase_domain_index` to the `THREAD_SUB_THREAD` macro.) If your multiphase model has only two phases defined, then `phase_domain_index` is `0` for the primary phase, and `1` for the secondary phase. However, if you have more than one secondary phase defined for your multiphase model, you will need to use the `PHASE_DOMAIN_INDEX` utility to retrieve the corresponding `phase_domain_index` for the given domain. See Phase Domain Index (`PHASE_DOMAIN_INDEX`) (p. 376) for details.

3.3.10.3. Phase Thread Pointer Array (`THREAD_SUB_THREADS`)

The `THREAD_SUB_THREADS` macro can be used to retrieve the pointer array, `pt`, whose elements contain pointers to phase-level threads (subthreads). `THREAD_SUB_THREADS` has one argument, `mixture_thread`.

```
Thread *mixture_thread;
Thread **pt;  /* initialize pt  */
pt = THREAD_SUB_THREADS(mixture_thread);
```

`mixture_thread` is a pointer to a mixture-level thread which can represent a cell thread or a face thread. It is automatically passed to your UDF by the Ansys Fluent solver when you use a `DEFINE` macro that contains a variable thread argument (for example, `DEFINE_PROFILE`), and the function is hooked to the mixture. Otherwise, if the mixture thread pointer is not explicitly passed to your UDF, then you will need to use another method to retrieve it. For example you can use the `Lookup_Thread` utility macro (see Thread Pointer for Zone ID (`Lookup_Thread`) (p. 328)).

`pt[i]`, an element in the array, is a pointer to the corresponding phase-level thread for the *i*th phase, where `i` is the `phase_domain_index`. You can use `pt[i]` as an argument to some cell variable macros when you want to retrieve specific phase information at a cell. For example, `C_R(c,pt[i])` can be used to return the density of the ith phase fluid at cell `c`. The pointer `pt[i]` can also be retrieved using `THREAD_SUB_THREAD`, discussed in Phase-Level Thread Pointer (`THREAD_SUB_THREAD`) (p. 374), using `i` as an argument. The `phase_domain_index` can be retrieved using the `PHASE_DOMAIN_INDEX` macro. See Phase Domain Index (`PHASE_DO-MAIN_INDEX`) (p. 376) for details.

3.3.10.4. Mixture Domain Pointer (`DOMAIN_SUPER_DOMAIN`)

You can use `DOMAIN_SUPER_DOMAIN` when your UDF has access to a particular phase-level domain (subdomain) pointer, and you want to retrieve the mixture-level domain pointer. `DOMAIN_SU-PER_DOMAIN` has one argument, `subdomain`. Note that `DOMAIN_SUPER_DOMAIN` is similar in implementation to the `THREAD_SUPER_THREAD` macro described in Mixture Thread Pointer (`THREAD_SUPER_THREAD`) (p. 376).

```
Domain *subdomain;
Domain *mixture_domain = DOMAIN_SUPER_DOMAIN(subdomain);
```

subdomain is a pointer to a phase-level domain within the multiphase mixture. It is automatically passed to your UDF by the Ansys Fluent solver when you use a DEFINE macro that contains a domain variable argument (for example, DEFINE_ADJUST), and the function is hooked to a primary or secondary phase in the mixture. Note that in the current version of Ansys Fluent, DOMAIN_SU-PER_DOMAIN will return the same pointer as Get_Domain(1). Therefore, if a subdomain pointer is available in your UDF, it is recommended that the DOMAIN_SUPER_DOMAIN macro be used instead of the Get_Domain macro to avoid potential incompatibility issues with future releases of Ansys Fluent.

3.3.10.5. Mixture Thread Pointer (THREAD_SUPER_THREAD)

You can use the THREAD_SUPER_THREAD macro when your UDF has access to a particular phase-level thread (subthread) pointer, and you want to retrieve the mixture-level thread pointer. THREAD_SUPER_THREAD has one argument, subthread.

```
Thread *subthread;
Thread *mixture_thread = THREAD_SUPER_THREAD(subthread);
```

subthread is a pointer to a particular phase-level thread within the multiphase mixture. It is automatically passed to your UDF by the Ansys Fluent solver when you use a DEFINE macro that contains a thread variable argument (for example, DEFINE_PROFILE, and the function is hooked to a primary or secondary phase in the mixture. Note that THREAD_SUPER_THREAD is similar in implementation to the DOMAIN_SUPER_DOMAIN macro described in Mixture Domain Pointer (DOMAIN_SUPER_DOMAIN) (p. 375).

3.3.10.6. Domain ID (DOMAIN_ID)

You can use DOMAIN_ID when you want to access the domain_id that corresponds to a given phase-level domain pointer. DOMAIN_ID has one argument, subdomain, which is the pointer to a phase-level domain. The default domain_id value for the top-level domain (mixture) is 1. That is, if the domain pointer that is passed to DOMAIN_ID is the mixture-level domain pointer, then the function will return a value of 1. Note that the domain_id that is returned by the macro is the same integer ID that is displayed in the graphical user interface when you select the desired phase in the **Phases** dialog box in Ansys Fluent.

```
Domain *subdomain;
int domain_id = DOMAIN_ID(subdomain);
```

3.3.10.7. Phase Domain Index (PHASE_DOMAIN_INDEX)

The PHASE_DOMAIN_INDEX macro retrieves the phase_domain_index for a given phase-level domain (subdomain) pointer. PHASE_DOMAIN_INDEX has one argument, subdomain, which is the pointer to a phase-level domain. phase_domain_index is an index of subdomain pointers. It is an integer that starts with 0 for the primary phase and is incremented by one for each secondary phase.

```
Domain *subdomain;
int phase_domain_index = PHASE_DOMAIN_INDEX(subdomain);
```

3.4. Vector and Dimension Macros

Ansys Fluent provides some utilities that you can use in your UDFs to access or manipulate vector quantities and deal with two and three dimensions. These utilities are implemented as macros in the code.

There is a naming convention for vector utility macros. `V` denotes a vector, `S` denotes a scalar, and `D` denotes a sequence of three vector components of which the third is always ignored for a two-dimensional calculation. The standard order of operations convention of parentheses, exponents, multiplication, division, addition, and subtraction (PEMDAS) is not followed in vector functions. Instead, the underscore (_) sign is used to group operands into pairs, so that operations are performed on the elements of pairs before they are performed on groups.

> **Important:**
>
> Note that all of the vector utilities in this section have been designed to work correctly in 2D and 3D. Consequently, you do not need to do any testing to determine this in your UDF.

For more information, see the following sections:

3.4.1. Macros for Dealing with Two and Three Dimensions

3.4.2. The ND Macros

3.4.3. The NV Macros

3.4.4. Vector Operation Macros

3.4.1. Macros for Dealing with Two and Three Dimensions

There are two ways that you can deal with expressions involving two and three dimensions in your UDF. The first is to use an explicit method to direct the compiler to compile separate sections of the code for 2D and 3D, respectively. This is done using `RP_2D` and `RP_3D` in conditional-if statements. The second method allows you to include general 3D expressions in your UDF, and use `ND` and `NV` macros that will remove the *z*-components when compiling with `RP_2D`. `NV` macros operate on vectors while `ND` macros operate on separate components.

3.4.1.1. `RP_2D` and `RP_3D`

The use of a `RP_2D` and `RP_3D` macro in a conditional-if statement will direct the compiler to compile separate sections of the code for 2D and 3D, respectively. For example, if you want to direct the compiler to compute swirl terms for the 3D version of Ansys Fluent only, then you would use the following conditional compile statement in your UDF:

```
#if RP_3D
/* compute swirl terms */
#endif
```

3.4.2. The `ND` Macros

The use of ND macros in a UDF allows you to include general 3D expressions in your code, and the `ND` macros take care of removing the *z* components of a vector when you are compiling with `RP_2D`.

3.4.2.1. `ND_ND`

The constant `ND_ND` is defined as `2` for `RP_2D` (Ansys Fluent 2D) and `3` for `RP_3D` (Ansys Fluent 3D). It can be used when you want to build a 2×2 matrix in 2D and a 3×3 matrix in 3D. When you use `ND_ND`, your UDF will work for both 2D and 3D cases, without requiring any modifications.

```
real A[ND_ND][ND_ND]

for (i=0; i<ND_ND; ++i)
 for (j=0; j<ND_ND; ++j)
 A[i][j] = f(i, j);
```

3.4.2.2. `ND_SUM`

The utility `ND_SUM` computes the sum of `ND_ND` arguments.

```
ND_SUM(x, y, z)
 2D: x + y;
 3D: x + y + z;
```

3.4.2.3. `ND_SET`

The utility `ND_SET` generates `ND_ND` assignment statements.

```
ND_SET(u, v, w, C_U(c, t), C_V(c, t), C_W(c, t))
 u = C_U(c, t);
 v = C_V(c, t);

if 3D:
21
 w = C_W(c, t);
```

3.4.3. The `NV` Macros

The `NV` macros have the same purpose as `ND` macros, but they operate on vectors (that is, arrays of length `ND_ND`) instead of separate components.

3.4.3.1. `NV_V`

The utility `NV_V` performs an operation on two vectors.

```
NV_V(a, =, x);
 a[0] = x[0]; a[1] = x[1]; etc.
```

Note that if you use + = instead of = in the above equation, then you get

```
 a[0]+=x[0]; etc.
```

See DEFINE_GRID_MOTION (p. 271) for an example UDF that utilizes `NV_V`.

3.4.3.2. `NV_VV`

The utility `NV_VV` performs operations on vector elements. The operation that is performed on the elements depends upon what symbol (– , / , *) is used as an argument in place of the + signs in the following macro call.

```
NV_VV(a, =, x, +, y)
  2D: a[0] = x[0] + y[0], a[1] = x[1] + y[1];
```

See `DEFINE_GRID_MOTION` for an example UDF that utilizes `NV_VV`.

3.4.3.3. `NV_V_VS`

The utility `NV_V_VS` adds a vector to another vector which is multiplied by a scalar.

```
NV_V_VS(a, =, x, +, y, *, 0.5);
  2D: a[0] = x[0] + (y[0]*0.5), a[1] = x[1] +(y[1]*0.5);
```

Note that the + sign can be replaced by –, /, or *, and the * sign can be replaced by /.

3.4.3.4. `NV_VS_VS`

The utility `NV_VS_VS` adds a vector to another vector which are each multiplied by a scalar.

```
NV_VS_VS(a, =, x, *, 2.0, +, y, *, 0.5);
  2D: a[0] = (x[0]*2.0) + (y[0]*0.5), a[1] = (x[1]*2.0) + (y[1]*0.5);
```

Note that the + sign can be used in place of –, *, or /, and the * sign can be replaced by /.

3.4.4. Vector Operation Macros

There are macros that you can use in your UDFs that will allow you to perform operations such as computing the vector magnitude, dot product, and cross product. For example, you can use the `real` function `NV_MAG(V)` to compute the magnitude of vector `V`. Alternatively, you can use the `real` function `NV_MAG2(V)` to obtain the square of the magnitude of vector `V`.

3.4.4.1. Vector Magnitude Using `NV_MAG` and `NV_MAG2`

The utility `NV_MAG` computes the magnitude of a vector. This is taken as the square root of the sum of the squares of the vector components.

```
NV_MAG(x)
  2D: sqrt(x[0]*x[0] + x[1]*x[1]);
  3D: sqrt(x[0]*x[0] + x[1]*x[1] + x[2]*x[2]);
```

The utility `NV_MAG2` computes the sum of squares of vector components.

```
NV_MAG2(x)
  2D: (x[0]*x[0] + x[1]*x[1]);
  3D: (x[0]*x[0] + x[1]*x[1] + x[2]*x[2]);
```

See `DEFINE_DPM_BC` for an example UDF that utilizes `NV_MAG`.

3.4.4.2. Dot Product

The following utilities compute the dot product of two sets of vector components.

```
ND_DOT(x, y, z, u, v, w)
 2D: (x*u + y*v);
 3D: (x*u + y*v + z*w);

 NV_DOT(x, u)
 2D: (x[0]*u[0] + x[1]*u[1]);
 3D: (x[0]*u[0] + x[1]*u[1] + x[2]*u[2]);

 NVD_DOT(x, u, v, w)
 2D: (x[0]*u + x[1]*v);
 3D: (x[0]*u + x[1]*v + x[2]*w);
```

See `DEFINE_DOM_SPECULAR_REFLECTIVITY` (p. 64) for an example UDF that utilizes `NV_DOT`.

3.4.4.3. Cross Product

For 3D, the `CROSS` macros return the specified component of the vector cross product. For 2D, the macros return the cross product of the vectors with the *z*-component of each vector set to 0.

```
ND_CROSS_X(x0,x1,x2,y0,y1,y2)
2D: 0.0
3D: (((x1)*(y2))-(y1)*(x2)))

ND_CROSS_Y(x0,x1,x2,y0,y1,y2)
2D: 0.0
3D: (((x2)*(y0))-(y2)*(x0)))

ND_CROSS_Z(x0,x1,x2,y0,y1,y2)
2D and 3D: (((x0)*(y1))-(y0)*(x1)))

NV_CROSS_X(x,y)
 ND_CROSS_X(x[0],x[1],x[2],y[0],y[1],y[2])

NV_CROSS_Y(x,y)
 ND_CROSS_Y(x[0],x[1],x[2],y[0],y[1],y[2])

NV_CROSS_Z(x,y)
 ND_CROSS_Z(x[0],x[1],x[2],y[0],y[1],y[2])

NV_CROSS(a,x,y)
 a[0] = NV_CROSS_X(x,y);
 a[1] = NV_CROSS_Y(x,y);
 a[2] = NV_CROSS_Z(x,y);
```

See `DEFINE_GRID_MOTION` (p. 271) for an example UDF that utilizes `NV_CROSS`.

3.5. Time-Dependent Macros

You can access time-dependent variables in your UDF in two different ways: direct access using a solver macro, or indirect access using an `RP` variable macro. Table 3.42: Solver Macros for Time-Dependent Variables (p. 381) contains a list of solver macros that you can use to access time-dependent variables in Ansys Fluent. An example of a UDF that uses a solver macro to access a time-dependent variable is

provided below. See DEFINE_DELTAT (p. 23) for another example that utilizes a time-dependent macro.

Table 3.42: Solver Macros for Time-Dependent Variables

Macro Name	Returns
CURRENT_TIME	real current flow time (in seconds)
CURRENT_TIMESTEP	real current physical time step size (in seconds)
PREVIOUS_TIME	real previous flow time (in seconds)
PREVIOUS_2_TIME	real flow time two steps back in time (in seconds)
PREVIOUS_TIMESTEP	real previous physical time step size (in seconds)
N_TIME	integer number of time steps
N_ITER	integer number of iterations

> **Important:**
>
> You *must* include the unsteady.h header file in your UDF source code when using the PREVIOUS_TIME or PREVIOUS_2_TIME macros since it is not included in udf.h.

> **Important:**
>
> N_ITER can only be utilized in compiled UDFs.

Some time-dependent variables such as current physical flow time can be accessed directly using a solver macro (CURRENT_TIME) , or indirectly by means of the RP variable macro.

Solver Macro Usage

```
real current_time;
current_time = CURRENT_TIME;
```

"Equivalent" RP Macro Usage

```
real current_time;
current_time = RP_Get_Real("flow-time");
```

Table 3.43: Solver and RP Macros that Access the Same Time-Dependent Variable (p. 381) shows the correspondence between solver and RP macros that access the same time-dependent variables.

Table 3.43: Solver and RP Macros that Access the Same Time-Dependent Variable

Solver Macro	"Equivalent" RP Variable Macro
CURRENT_TIME	RP_Get_Real("flow-time")
CURRENT_TIMESTEP	RP_Get_Real("physical-time-step")

Solver Macro	"Equivalent" RP Variable Macro
`N_TIME`	`RP_Get_Integer("time-step")`

Important:

You should not access a Scheme variable using any of the `RP_GET_...` functions from inside a cell or face looping macro (c_loop or f_loop). This type of communication between the solver and cortex is very time consuming and therefore should be done outside of loops.

Example

The integer time step count (accessed using `N_TIME`) is useful in `DEFINE_ADJUST` functions for detecting whether the current iteration is the first in the time step.

```
/************************************************************************
    Example UDF that uses N_TIME
*************************************************************************/
static int last_ts = -1;  /* Global variable. Time step is never <0 */

DEFINE_ADJUST(first_iter_only, domain)
{
    int curr_ts;
    curr_ts = N_TIME;
    if (last_ts != curr_ts)
    {
    last_ts = curr_ts;
    /* things to be done only on first iteration of each time step
    can be put here */
    }
}
```

Important:

There is a new variable named `first_iteration` that can be used in the above `if` statement. `first_iteration` is true only at the first iteration of a timestep. Since the adjust UDF is also called before timestepping begins, the two methods vary slightly as to when they are true. You must decide which behavior is more appropriate for your case.

3.6. Scheme Macros

The text interface of Ansys Fluent executes a Scheme interpreter, which allows you to define your own variables that can be stored in Ansys Fluent and accessed via a UDF. This capability can be very useful, for example, if you want to alter certain parameters in your case, and you do not want to recompile your UDF each time. Suppose you want to apply a UDF to multiple zones in a mesh. You can do this manually by accessing a particular Zone ID in the graphical user interface, hard-coding the integer ID in your UDF, and then recompiling the UDF. This can be a tedious process if you want to apply the UDF to a number of zones. By defining your own Scheme variable, if you want to alter the variable later, then you can do it from the text interface using a Scheme command.

Macros that are used to define and access user-specified Scheme variables from the text interface are identified by the prefix `rp`, (for example, `rp-var-define`). Macros that are used to access user-defined Scheme variables in an Ansys Fluent solver, are identified by the prefix `RP` (for example, `RP_Get_Real`). These macros are executed within UDFs.

For more information, see the following sections:

3.6.1. Defining a Scheme Variable in the Text Interface

To define a Scheme variable named `pres_av/thread-id` in the text interface, you can use the following Scheme command:

```
(rp-var-define 'pres_av/thread-id 2 'integer #f)
```

This will create the Scheme variable as an integer type and assign it the initial value 2. Your Scheme variable and its value will be saved and restored with the case file. Other types besides `'integer` include `'real`, `'boolean`, and `'string`.

Typically, you should avoid redefining an existing Scheme variable. The following Scheme command will only define a new Scheme variable if there is not an existing Scheme variable with the same name, therefore it is preferred over the `(rp-var-define '…)` command:

```
(make-new-rpvar 'pres_av/thread-id 2 'integer)
```

This command first checks that the variable `pres_av/thread-id` is not already defined, and then sets it up as an integer with an initial value of 2. *If a Scheme variable with this name is already defined, its value will not be changed/overwritten by this command.*

Note that the string `'/'` is allowed in Scheme variable names (as in `pres_av/thread-id`), and is a useful way to organize variables so that they do not interfere with each other.

3.6.2. Accessing a Scheme Variable in the Text Interface

After you define a Scheme variable in the text interface, you can access the variable. For example, if you want to check the current value of the variable (for example, `pres_av/thread-id`) on the Scheme side, you can type the following command in the text window:

```
(%rpgetvar 'pres_av/thread-id)
```

> **Important:**
>
> It is recommended that you use `%rpgetvar` when you are retrieving an Ansys Fluent variable using a Scheme command. The `%` symbol forces Ansys Fluent to check for the stored value of the variable instead of the buffered value. Access to buffered values using `rpgetvar` (without the `%` symbol) is faster, but when the RP variable is changed from the C side (a UDF or Ansys Fluent), the buffered value might be outdated. Only in instances where the variable is not changed from the C side, can you neglect the `%` symbol and use the buffered value with confidence.

3.6.3. Changing a Scheme Variable to Another Value in the Text Interface

Alternatively, if you want to change the value of the variable you have defined (`pres_av/thread-id`) to say, 7, then you will need to use `rpsetvar` and issue the following command in the text window:

```
(rpsetvar 'pres_av/thread-id 7)
```

3.6.4. Accessing a Scheme Variable in a UDF

After a new variable is defined on the Scheme side (using a text command), you will need to bring it over to the solver side to be able to use it in your UDF. '`RP`' macros are used to access Scheme variables in UDFs, and are listed below.

`RP_Variable_Exists_P("variable-name")`	Returns true if the variable exists
`RP_Get_Real("variable-name")`	Returns the double value of `variable-name`
`RP_Get_Integer("variable-name")`	Returns the integer value of `variable-name`
`RP_Get_String("variable-name")`	Returns the const char* value of `variable-name`
`RP_Get_Boolean("variable-name")`	Returns the Boolean value of `variable-name`
`Get_Input_Parameter("variable-name")`	Returns the input parameter value of `variable-name`

For example, to access the user-defined Scheme variable `pres_av/thread-id` in your UDF C function, you will use `RP_Get_Integer`. You can then assign the variable returned to a local variable you have declared in your UDF (for example, `surface_thread_id`) as demonstrated below:

```
surface_thread_id = RP_Get_Integer("pres_av/thread-id");
```

You can also change a Scheme variable in your UDF using the following RP macros:

`RP_Set_Real("variable-name", ...)`	Sets the double value of `variable-name`
`RP_Set_Integer("variable-name", ...)`	Sets the integer value of `variable-name`
`RP_Set_String("variable-name", ...)`	Sets the const char* value of `variable-name`
`RP_Set_Boolean("variable-name", ...)`	Sets the Boolean value of `variable-name`

> **Important:**
>
> When you change the value of a Scheme variable in a UDF using `RP_Set_...`, it must be done on every process (host and nodes) that will access the value with the `RP_Get_...`

macro. Ensure that the value is synchronized between the node(s) and host processes before using `RP_Set_....`

To enable the consistent use of the changed value on the Scheme side, `RP_Set_...` must be called on the host process and the Scheme side must use (`%rpgetvar '…`) (refer to Accessing a Scheme Variable in the Text Interface (p. 383) for more information).

3.7. Input/Output Functions

Ansys Fluent provides some utilities in addition to the standard C I/O functions that you can use to perform input/output (I/O) tasks. These are listed below and are described in the following sections:

`Message(format, ...)`	prints a message to the console
`Error(format, ...)`	prints an error message to the console
`par_fprintf_head(fp, format, ...)`	prints a header line at the top of a sample file when using the `DEFINE_DPM_OUTPUT` macro
`par_fprintf(fp, format, ...)`	prints one line of particle information to a sample file when using the `DEFINE_DPM_OUTPUT` macro

For more information, see the following sections:

3.7.1. Message

3.7.2. Error

3.7.3. The par_fprintf_head and par_fprintf Functions

3.7.1. `Message`

The `Message` function is a utility that displays data to the console in a format that you specify.

```
int Message(const char *format,...);
```

The first argument in the `Message` function is the format string. It specifies how the remaining arguments are to be displayed in the console. The format string is defined within quotes. The value of the replacement variables that follow the format string will be substituted in the display for all instances of `%type`. The `%` character is used to designate the character type. Some common format characters are: `%d` for integers, `%f` for floating point numbers, `%g` for double data type, and `%e` for floating point numbers in exponential format (with `e` before the exponent). Consult a C programming language manual for more details. The format string for `Message` is similar to `printf`, the standard C I/O function (see Standard I/O Functions (p. 687) for details).

In the example below, the text `Volume integral of turbulent dissipation:` will be displayed in the console, and the value of the replacement variable, `sum_diss`, will be substituted in the message for all instances of `%g`.

Example:

```
Message("Volume integral of turbulent dissipation: %g\n", sum_diss);
 /* g represents floating point number in f or e format */
 /* \n denotes a new line */
```

> **Important:**
>
> It is recommended that you use `Message` instead of `printf` in compiled UDFs.

3.7.2. `Error`

You can use `Error` when you want to stop execution of a UDF and print an error message to the console.

Example:

```
if (table_file == NULL)
Error("error reading file");
```

> **Important:**
>
> `Error` is not supported by the interpreter and can be used only in compiled UDFs.

3.7.3. The `par_fprintf_head` and `par_fprintf` Functions

When using `DEFINE_DPM_OUTPUT` to write a DPM sample file, the special text output functions `par_fprintf` and `par_fprintf_head` must be used in place of the C I/O function `fprintf`. Typically, one or more header lines is written to the top of the file with column headings or other non-repeating information using `par_fprintf_head`. Following the header, a line is written for each particle sample using `par_fprintf`. Further details and examples are provided in the following sections:

3.7.3.1. `par_fprintf_head`

The `par_fprintf_head` function generates a header at the top of the DPM sample file.

```
int par_fprintf_head(FILE *fp, const char *format, ...);
```

The first argument is the file pointer provided by the calling routine defined with `DEFINE_DPM_OUTPUT`. The second argument is the format string, used as described for the `Message` function in Message (p. 385). An example of the usage of `par_fprintf_head` is given below:

Example:

```
par_fprintf_head(fp, "x-coordinate y-coordinate z-coordinate\n");
```

This prints the column names `x-coordinate`, `y-coordinate`, and `z-coordinate` followed by a line feed at the top of the file indicated by `FILE` pointer `fp`. Multiple calls can be made to `par_fprintf_head` as needed to write all desired information to the top of the sample file.

For an illustration of the use of `par_fprintf_head` within a DEFINE_DPM_OUTPUT UDF refer to the Example provided in `DEFINE_DPM_OUTPUT` (p. 227).

3.7.3.2. `par_fprintf`

The `par_fprintf` function writes a single particle sample line into a DPM sample file.

```
int par_fprintf(FILE *fp, const char *format, ...);
```

The first argument is the file pointer provided by the calling routine defined with `DEFINE_DPM_OUTPUT`. The second argument is the format string, used as described for the `Message` function in `Message` (p. 385). When used with `par_fprintf`, the first two replacement variables in the format string must be the particle injection ID and particle ID, respectively. The rest of the format string can be chosen to specify the output that will be written and its formatting.

For an illustration of the use of `par_fprintf` within a DEFINE_DPM_OUTPUT UDF refer to Example 1 - Sampling and Removing Particles (p. 228) in `DEFINE_DPM_OUTPUT` (p. 227).

3.8. Miscellaneous Macros

For more information, see the following sections:

3.8.1. `Data_Valid_P()`

You can check that the cell values of the variables that appear in your UDF are accessible before you use them in a computation by using the `Data_Valid_P` macro.

```
cxboolean Data_Valid_P()
```

`Data_Valid_P` is defined in the `id.h` header file, and is included in `udf.h`. The function returns `1` (true) if the data that is passed as an argument is valid, and `0` (false) if it is not.

Example:

```
if(!Data_Valid_P()) return;
```

Suppose you read a case file and, in the process, load a UDF. If the UDF performs a calculation using variables that have not yet been initialized, such as the velocity at interior cells, then an error will occur. To avoid this kind of error, an `if else` condition can be added to your code. If (`if`) the data

are available, the function can be computed in the normal way. If the data are not available (`else`), then no calculation, or a trivial calculation can be performed instead. After the flow field has been initialized, the function can be reinvoked so that the correct calculation can be performed.

3.8.2. `FLUID_THREAD_P()`

```
cxboolean FLUID_THREAD_P(t);
```

You can use `FLUID_THREAD_P` to check whether a cell thread is a fluid thread. The macro is passed a cell thread pointer `t`, and returns `1` (or `TRUE`) if the thread that is passed is a fluid thread, and `0` (or `FALSE`) if it is not.

Note that `FLUID_THREAD_P(t)` assumes that the thread is a cell thread.

For example,

```
FLUID_THREAD_P(t0);
```

returns `TRUE` if the thread pointer `t0` passed as an argument represents a fluid thread.

3.8.3. `Get_Report_Definition_Values`

You can access the last calculated report definition value for any report definition you created in Fluent using the `Get_Report_Definition_Values` API.

Before calling this API you must either specify the output arguments as `NULL` or allocate appropriate memory to them. The API will fill the output arguments that are not specified as `NULL`. If memory is not appropriately allocated, the calculation run may crash or result in unexpected behavior.

```
int Get_Report_Definition_Values(const char* name, int
timeStep/iteration, int* nrOfvalues, real* values, int* ids, int*
index)
```

A return value of `0` means the call was successful, `1` means the specified report definition does not exist.

Important:

For `Get_Report_Definition_Values` to work properly with its referenced report definition:

- The report definition must be included in at least one report file or report plot.

- The **Average Over** value of the report definition must match the **Get Data Every** frequency of the including report file or report plot.

Argument Type	Description
`const char*` [input]	name of the report definition
`int` [input]	flag with a value of 0 or 1. 0 provides the value calculated at an iteration, 1 provides the value

	at a time-step. (1 is only valid for unsteady calculations.)
int* [output]	number of values available for the report definition
real* [output]	values for the report definition
** int* [output]	surface/zone ids corresponding to the values
int* [output]	calculation index

** If the per-zone or per-surface option is disabled and the number of surfaces/zones is greater than 1, the surface/zone id will return -1

Steady-State Example

The following example shows how to use the API for a steady-state case:

```
int nrOfvalues=0;
real *values;
int *ids;
int index;
int counter;

/*First call to get the number of values. For number of values,
  the int pointer is passed, which is 0 for iterations.*/

int rv = Get_Report_Definition_Values("report-def-0", 0, &nrOfvalues, NULL, NULL,NULL);

if (rv==0 && nrOfvalues)
{

    Message("Report definition evaluated at iteration has %d values\n", nrOfvalues);

    /*Memory is allocated for values and ids.*/

    values = (real*) malloc(sizeof(real)* nrOfvalues);
    ids = (int*) malloc(sizeof(int)* nrOfvalues);

    /* Second call to get data. The number of values is null, but the last
     * three are not.*/
    rv = Get_Report_Definition_Values("report-def-0", 0, NULL, values, ids, &index);

    Message("Values correspond to iteration index:%d\n", index);

    for ( counter = 0; counter < nrOfvalues; counter++ )
    {
        Message("report definition values: %d, %f\n", ids[counter], values[counter]);
    }

    /*Memory is freed.*/
    free(values);
    free(ids);
}
else
{
    /*The command can be unsuccessful if the report definition does not exist
     or if it has not been evaluated yet.*/
    if (rv == 1)
    {
        Message("report definition: %s does not exist\n", "report-def-0");
    }
    else if ( nrOfvalues == 0 )
    {
```

```
            Message("report definition: %s not evaluated at iteration level\n", "report-def-0");
    }
}
```

Transient Example

The following example shows how to use the API for a transient case:

```
int nrOfvalues=0;
real *values;
int *ids;
int index;
int counter;

/*First call to get the number of values. For number of values,
  the int pointer is passed, which is 1 for timesteps.*/

int rv = Get_Report_Definition_Values("report-def-0", 1, &nrOfvalues, NULL, NULL,NULL);

if (rv==0 && nrOfvalues)
{

    Message("Report definition evaluated at time-step has %d values\n", nrOfvalues);

    /*Memory is allocated for values and ids.*/

    values = (real*) malloc(sizeof(real)* nrOfvalues);
    ids = (int*) malloc(sizeof(int)* nrOfvalues);

    /* Second call to get data. The number of values is null, but the last
     * three are not.*/
    rv = Get_Report_Definition_Values("report-def-0", 1, NULL, values, ids, &index);

    Message("Values correspond to time-step index:%d\n", index);

    for ( counter = 0; counter < nrOfvalues; counter++ )
    {
        Message("report definition values: %d, %f\n", ids[counter], values[counter]);
    }

    /*Memory is freed.*/
    free(values);
    free(ids);
}
else
{
    /*The command can be unsuccessful if the report definition does not exist
      or if it has not been evaluated yet.*/
    if (rv == 1)
    {
        Message("report definition: %s does not exist\n", "report-def-0");
    }
    else if ( nrOfvalues == 0 )
    {
        Message("report definition: %s not evaluated at time-step level\n", "report-def-0");
    }
}
```

3.8.4. `M_PI`

The macro `M_PI` returns the value of π.

3.8.5. `NULLP & NNULLP`

You can use the `NULLP` and `NNULLP` functions to check whether storage has been allocated for user-defined scalars. `NULLP` returns `TRUE` if storage is *not* allocated, and `NNULLP` returns `TRUE` if storage is allocated. Below are some examples of usage.

```
NULLP(T_STORAGE_R_NV(t0, SV_UDSI_G(p1)))

/* NULLP returns TRUE if storage is not allocated for
 user-defined storage variable       */



NNULLP(T_STORAGE_R_NV(t0, SV_UDSI_G(p1)))

/* NNULLP returns TRUE if storage is allocated for
 user-defined storage variable       */
```

3.8.6. `N_UDM`

You can use `N_UDM` to access the number of user-defined memory (UDM) locations that have been used in Ansys Fluent. The macro takes no arguments, and returns the integer number of memory locations used. It is defined in `models.h`.

3.8.7. `N_UDS`

You can use `N_UDS` to access the number of user-defined scalar (UDS) transport equations that have been specified in Ansys Fluent. The macro takes no arguments and returns the integer number of equations. It is defined in `models.h`.

3.8.8. `SQR(k)`

`SQR(k)` returns the square of the given variable `k`, or *k*k*.

3.8.9. `UNIVERSAL_GAS_CONSTANT`

`UNIVERSAL_GAS_CONSTANT` returns the value of the universal gas constant ($8314.34 J / Kmol-K$).

> **Important:**
>
> Note that this constant is *not* expressed in SI units.

See `DEFINE_VR_RATE` (p. 170) for an example UDF that utilizes `UNIVERSAL_GAS_CONSTANT`.

Chapter 4: Interpreting UDFs

After you have written your UDF and have saved the source code file with a `.c` extension in your working folder, you are ready to interpret the source file. Follow the instructions below in After it has been interpreted, the UDF function name(s) that you supplied in the `DEFINE` macro(s) appear in drop-down lists in Ansys Fluent, ready for you to hook to your CFD model.

Alternatively, if you want to compile your UDF source file, see

4.1. Introduction

An interpreted UDF is a function that is interpreted directly from a source file (for example, `udfexample.c`) at *run time*. You will use the **Interpreted UDFs** dialog box to interpret all of the functions in the source file in a single step.

After a source file is interpreted, you can write the case file, and the names and contents of the interpreted function(s) are stored in the case. In this way, the function(s) will be automatically interpreted whenever the case file is subsequently read. After it has been interpreted (either manually through the **Interpreted UDFs** dialog box or automatically upon reading a case file), all of the interpreted UDFs that are contained within a source file will become visible and selectable in dialog boxes in Ansys Fluent.

Inside Ansys Fluent, the source code is compiled into an intermediate, architecture-independent machine code using a C preprocessor. This machine code then executes on an internal emulator, or interpreter, when the UDF is invoked. This extra layer of code incurs a performance penalty, but allows an interpreted UDF to be shared effortlessly between different architectures, operating systems, and Ansys Fluent versions. If execution speed does become an issue, an interpreted UDF can always be run in compiled mode without modification.

For more information, see the following:

4.1.1. Location of the `udf.h` File

UDFs are defined using `DEFINE` macros (see) and the definitions for `DEFINE` macros are included in `udf.h` header file. Consequently, before you can interpret a UDF source file, `udf.h` will need to be accessible in your path, or saved locally within your working folder.

The location of the `udf.h` file is:

```
path\ANSYS Inc\v212\fluent\fluent21.2.0\src\udf\udf.h
```

where `path` is the folder in which you have installed Ansys Fluent (by default, the path is `C:\Program Files`).

Important:

- You should not, under any circumstances, alter the `udf.h` file.

- In general, you should not copy `udf.h` from the installation area. The compiler is designed to look for this file locally (in your current folder) first. If it is not found in your current folder, the compiler will look in the `\src\udf` folder automatically. In the event that you upgrade your release area, but do not delete an old copy of `udf.h` from your working folder, you will not be accessing the most recent version of this file.

4.1.2. Limitations

The interpreter that is used for interpreted UDFs does not have all of the capabilities of a standard C compiler (which is used for compiled UDFs). Specifically, interpreted UDFs *cannot* contain any of the following C programming language elements:

- `goto` statements

- Non-ANSI-C prototypes for syntax

- Direct data structure references

- Declarations of local structures

- Unions

- Pointers to functions

- Arrays of functions

- Multi-dimensional arrays

In addition, the following types of macros are not supported in interpreted UDFs:

-

-

-

Caution:

Unlike compiled UDFs, there is no filtering for interpreted UDFs in the Fluent user interface, meaning that care needs to be taken to assign the appropriate UDF type to the corresponding field. For example, an interpreted DEFINE_PROPERTY UDF can be assigned to a field that only accepts DEFINE_PROFILE UDFs, which may cause Fluent to fail at run time.

4.2. Interpreting a UDF Source File Using the Interpreted UDFs Dialog Box

This section presents the steps for interpreting a source file in Ansys Fluent. After it has been interpreted, the names of UDFs contained within the source file will appear in drop-down lists in Ansys Fluent.

The general procedure for interpreting a source file is as follows:

1. Make sure that the UDF source file is in the same folder that contains your case and data files.
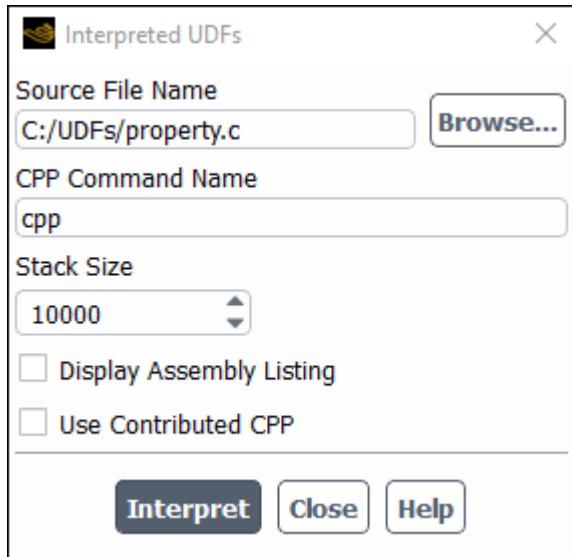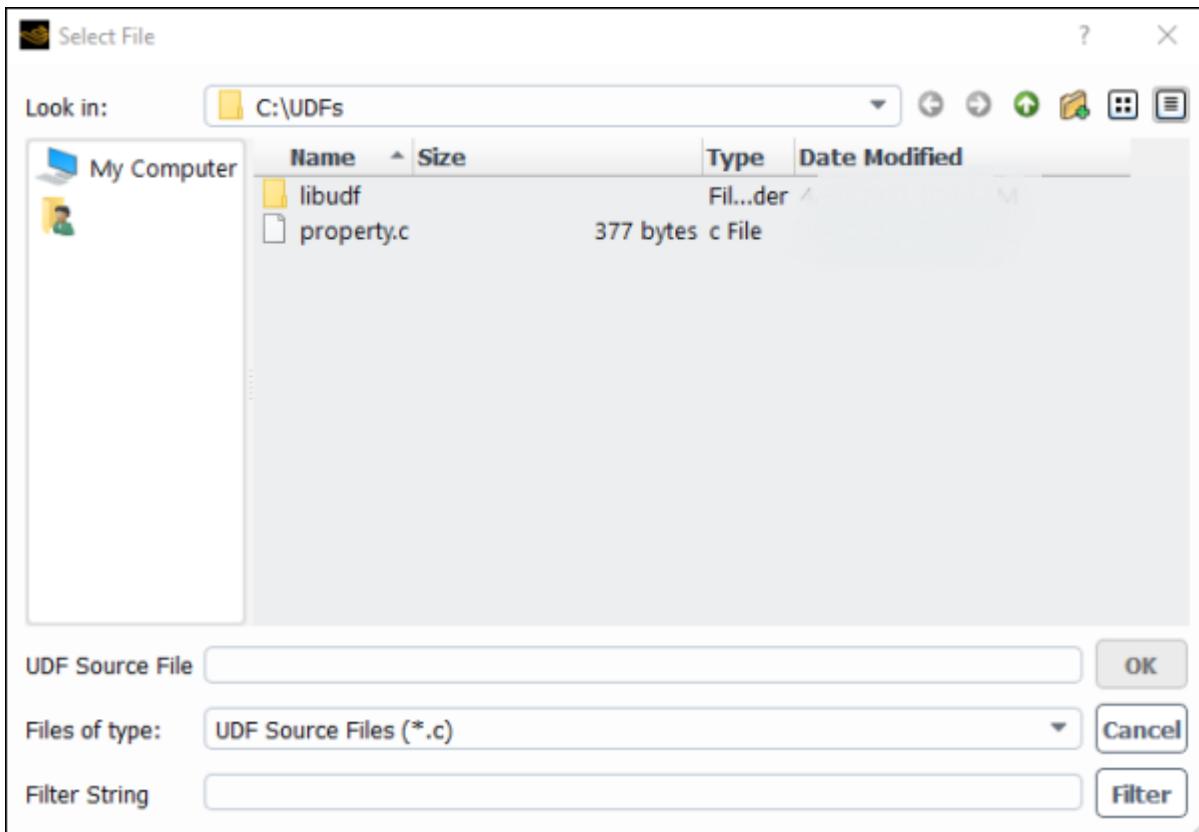
    > **Important:**
    >
    > If you are running the parallel version of Ansys Fluent on a network of Windows machines, you *must* "share" the working folder that contains your UDF source, case, and data files so that all of the compute nodes in the cluster can see it. To share the working folder:
    >
    > 1. Open Windows Explorer and browse to the folder.
    >
    > 2. Right-click the working folder and select **Sharing and Security** from the menu.
    >
    > 3. Click **Share this folder**.
    >
    > 4. Click **OK**.

2. The next step depends on your computer's operating system:

    *   **For Linux**, start Ansys Fluent from the directory that contains your case, data, and UDF source files.

    *   **For Windows**, start Ansys Fluent using Fluent Launcher, being sure to specify the folder that contains your case, data, and UDF source files in the **Working Directory** field in the **General Options** tab.

3. Read (or set up) your case file.

4. Interpret the UDF using the **Interpreted UDFs** dialog box ().

    🗔 **Parameters & Customization** → **User Defined Functions** 🖱 **Interpreted...**

Figure 4.1: The Interpreted UDFs Dialog Box

a.   Indicate the UDF source file you want to interpret by clicking **Browse....** This opens the **Select File** dialog box (Figure 4.2: The Select File Dialog Box (p. 396)).

Figure 4.2: The Select File Dialog Box

In the **Select File** dialog box, select the desired file (for example, `udfexample.c`) and click **OK**. The **Select File** dialog box closes and the complete path to the file you selected appears in the **Source File Name** text box in the **Interpreted UDFs** dialog box (Figure 4.1: The Interpreted UDFs Dialog Box (p. 396)).

b. In the **Interpreted UDFs** dialog box, specify the C preprocessor to be used in the **CPP Com-mand Name** field. You can keep the default `cpp` or you can enable the **Use Contributed CPP** option to use the preprocessor supplied by Ansys Fluent.

c. Keep the default **Stack Size** setting of `10000`, unless the number of local variables in your function will cause the stack to overflow. In this case, set the **Stack Size** to a number that is greater than the number of local variables used.

d. If you want a listing of assembly language code to appear in the console when the function interprets, enable the **Display Assembly Listing** option. This option will be saved in your case file, so that when you read the case in a subsequent Ansys Fluent session, the assembly code will be automatically displayed.

e. Click **Interpret** to interpret your UDF:

    • If the compilation is successful and you have enabled **Display Assembly Listing**, then the assembler code is displayed in the console.

    • If you chose not to display the listing and the compilation is successful, then the **CPP Command Name** that was executed is displayed the console.

    • If the compilation is unsuccessful, then Ansys Fluent reports an error and you will need to debug your program. See . You can also view the compilation history in the `log` file that is saved in your working folder.

f. **Close** the **Interpreted UDFs** dialog box when the interpreter has finished.

5. Write the case file. The interpreted function(s) are saved with the case file and are *automatically* interpreted when the case file is subsequently read.

4.3. Common Errors Made While Interpreting A Source File

If there are compilation errors when you interpret a UDF source file, they will appear in the console. However, you may not see all the error messages if they scroll off the screen too quickly. For this reason, you may want to disable the **Display Assembly Listing** option while debugging your UDF. You can view the compilation history in the `log` file that is saved in your working folder.

If you keep the **Interpreted UDFs** dialog box open while you are in the process of debugging your UDF, the **Interpret** button can be used repeatedly since you can make changes with an editor in a separate window. Then, you can continue to debug and interpret until no errors are reported. Remember to save changes to the source code file in the editor window before trying to interpret again.

One of the more common errors made when interpreting source files is trying to interpret code that contains elements of C that the interpreter does not accommodate. For example, if you have code that contains a structured reference call (which is not supported by the C preprocessor), the interpretation will fail and you will get an error message similar to the following:

```
Error: /nfs/clblnx/home/clb/fluent/udfexample.c:
 line 15: structure reference
```

Note:

If you have a source file that contains DOS-style line endings, before you can interpret the source file in Ansys Fluent on Linux, you must first run the dos2unix utility (for example, dos2unix *filename*.c) in the command line in order to make the source file compatible with the Ansys Fluent Linux compiler.

Chapter 5: Compiling UDFs

After you have written your UDF(s) and have saved the source file with a `.c` extension in your working folder, you are ready to compile the UDF source file, build a shared library from the resulting objects, and load the library into Ansys Fluent. After being loaded, the function(s) contained in the library will appear in drop-down lists in dialog boxes, ready for you to hook to your CFD model. Follow the instructions in Compiling a UDF Using the GUI (p. 403) to compile UDF source files using the graphical user interface (GUI). Compile a UDF Using the TUI (p. 407) explains how you can use the text user interface (TUI) to do the same. The text interface option provides the added capability of enabling you to link precompiled object files derived from non-Ansys Fluent sources (for example, Fortran sources) to your UDF (Link Precompiled Object Files From Non-Ansys Fluent Sources (p. 414)). This feature is not available in the GUI. Load and Unload Libraries Using the UDF Library Manager Dialog Box (p. 418) describes how you can load (and unload) multiple UDF libraries using the **UDF Library Manager** dialog box. The capability of loading more than one UDF library into Ansys Fluent raises the possibility of data contention if multiple libraries use the same user-defined scalar (UDS) and user-defined memory (UDM) locations. These clashes can be avoided if libraries reserve UDS or UDM prior to usage. See `Reserve_User_Scalar_Vars` (p. 355) and Reserving UDM Variables Using `Reserve_User_Memory_Vars` (p. 360), respectively, for details.

Note:

If the case file being read by Ansys Fluent uses a compiled UDF library, then Ansys Fluent looks for a corresponding library. If the library is missing, then Ansys Fluent will attempt to automatically compile the UDF library on the current platform. Also, if the UDF library is found to be outdated based on time stamps (that is, the UDF source or `user.udf` or `user_nt.udf` files are modified after the UDF library is compiled), then during the reading of the case, Ansys Fluent automatically recompiles the UDF library. Note the following:

- All required source and header files should be kept alongside the mesh/case file in the same directory.

- In order to save the compilation settings, you should load the mesh/case file first, then manually compile and load the UDF library, and save the case file. These settings can be removed by unloading the UDF library from the current session and then saving the case file.

5.1. Introduction

Compiled UDFs are built in the same way that the Ansys Fluent executable itself is built. Internally, a script called `Makefile` is used to invoke the system C compiler to build an object code library that contains the native machine language translation of your higher-level C source code. The object library is specific to the computer architecture being used during the Ansys Fluent session, as well as to the particular version of the Ansys Fluent executable being run. Therefore, UDF object libraries must be rebuilt any time Ansys Fluent is upgraded, when the computer's operating system level changes, or when the job is run on a different type of computer architecture. The generic process for compiling a UDF involves two steps: compile/build and load.

The compile/build step takes one or more source files (for example, `myudf.c`) containing at least one UDF and compiles them into object files (for example, `myudf.o` or `myudf.obj`) and then builds a "shared library" (for example, `libudf`) with the object files. If you compile your source file using the GUI, this compile/build process is executed when you click **Build** in the **Compiled UDFs** dialog box. The shared library that you name (for example, `libudf`) is automatically built for the architecture and version of Ansys Fluent you are running during that session (for example, `lnamd64/2d_host` and `lnamd64/2d_node`), and will store the UDF object file(s).

If you compile your source file using the TUI, you must first set up target folders for the shared libraries, modify a file named `Makefile` to specify source parameters, and then execute the `Makefile` which directs the compile/build process. Compiling a UDF using the TUI has the added advantage of allowing precompiled object files derived from non-Ansys Fluent sources to be linked to Ansys Fluent (Link Precompiled Object Files From Non-Ansys Fluent Sources (p. 414)). This option is not available using the GUI.

After the shared library is built (using the TUI or GUI), you load the UDF library into Ansys Fluent before you can use it. You can do this using the **Load** button in the **Compiled UDFs** dialog box. After being loaded, all of the compiled UDFs that are contained within the shared library will become visible and selectable in graphics dialog boxes in Ansys Fluent. Note that compiled UDFs are displayed in Ansys Fluent dialog boxes with the associated UDF library name separated by two colons (`::`). For example, a compiled UDF named `rrate` that is associated with a shared library named `libudf` would appear in Ansys Fluent dialog boxes as `rrate::libudf`. This distinguishes UDFs that are compiled from those that are interpreted.

If you write your case file when a UDF library is loaded, the library will be saved with the case and will be *automatically* loaded whenever that case file is subsequently read. This process of "dynamic loading" saves you having to reload the compiled library every time you want to run a simulation.

Before you compile your UDF source file(s) using one of the two methods provided in Compiling a UDF Using the GUI (p. 403) and Compile a UDF Using the TUI (p. 407), you must make sure that the `udf.h` header file is accessible in your path, or is saved locally within your working folder (Location of the `udf.h` File (p. 401)).

For more information, see the following sections:

5.1.1. Location of the `udf.h` File

UDFs are defined using `DEFINE` macros (see `DEFINE` Macros (p. 19)) and the definitions for `DEFINE` macros are included in `udf.h`. Consequently, before you compile your source file, the `udf.h` header file must be accessible in your path, or saved locally within your working folder.

The location of the `udf.h` file is:

`path\ANSYS Inc\v212\fluent\fluent21.2.0\src\udf\udf.h`

where `path` is the folder in which you have installed Ansys Fluent (by default, the path is `C:\Program Files`).

Important:

- You should not, under any circumstances, alter the `udf.h` file.

- In general, you should not copy `udf.h` from the installation area. The compiler is designed to look for this file locally (in your current folder) first. If it is not found in your current folder, the compiler will look in the `\src\udf` folder automatically. In the event that you upgrade your release area, but do not delete an old copy of `udf.h` from your working folder, you will not be accessing the most recent version of this file.

There may be instances when you will want to include additional header files in the compilation process. Make sure that all header files needed for UDFs are located in a folder under the `\src` folder.
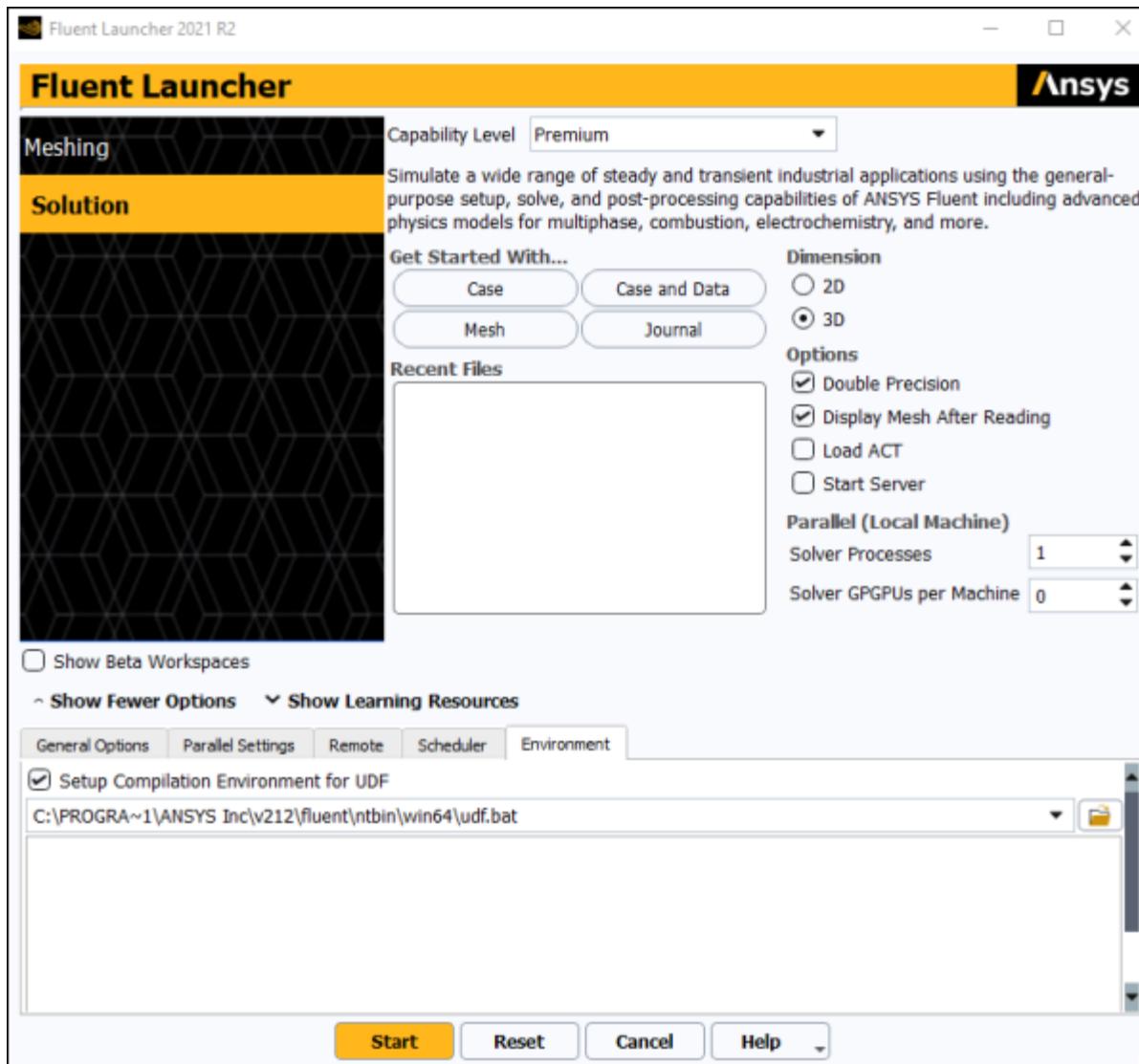
5.1.2. Compilers

The graphical and text interface processes for a compiled UDF require the use of a C compiler that is native to the operating system and machine you are running on. Most Linux operating systems provide a C compiler as a standard feature. For Linux machines, Ansys Fluent supports any ANSI-compliant compiler. If you are operating on a Windows system, it is recommended that you install a supported version of Microsoft Visual Studio or Clang on your machine before you proceed (for a list of supported versions, see Compiler Requirements for Windows Systems); note that if Fluent determines that you have not installed a compiler on your computer, a built-in compiler (Clang) that is provided as part of the Fluent installation is automatically used. If you are unsure about compiler requirements for your system, contact Ansys Fluent installation support.

Important:

Obsolete versions of any native compiler may not work properly with compiled UDFs.

When launching Ansys Fluent on Windows using Fluent Launcher, the **Environment** tab (Figure 5.1: The Environment Tab of the Fluent Launcher Dialog Box (p. 402)) allows you to specify compiler settings for compiling UDFs.

Figure 5.1: The Environment Tab of the Fluent Launcher Dialog Box

The **Set up Compilation Environment for UDF** option is enabled by default, and allows you to specify a batch file that contains UDF compilation environment settings. Enter a batch file name and path in the field, or click to browse for a batch file. By default, the Fluent Launcher dialog box is set to use the `udf.bat` file that is saved in your computer as part of the Ansys Fluent installation. It is recommended that you keep the default batch file, which is tested with versions of MS Visual Studio C++ and Clang compilers, as well as the built-in compiler.

5.2. Compiling a UDF Using the GUI

The general procedure for compiling a UDF source file, building a shared library for the resulting objects, and loading the compiled UDF library into Ansys Fluent using the graphical user interface (GUI) is as follows.

Important:

Note that when running serial or parallel Ansys Fluent on a Windows system, it is recommended that you install a supported version of Microsoft Visual Studio or Clang on your machine, preferably on the `C:` drive; if the compiler you installed is an older version that is not supported, then you should enable the use of a built-in compiler (Clang) that is provided as part of the Fluent installation, as described in a later step.

---
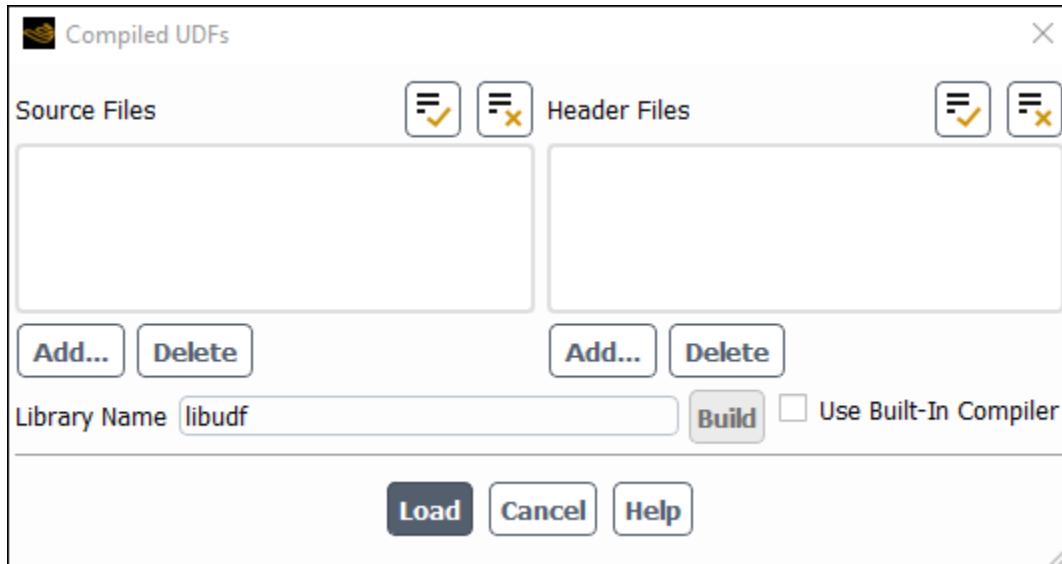
1. Make sure that the UDF source file you want to compile is in the same folder that contains your case and data files.
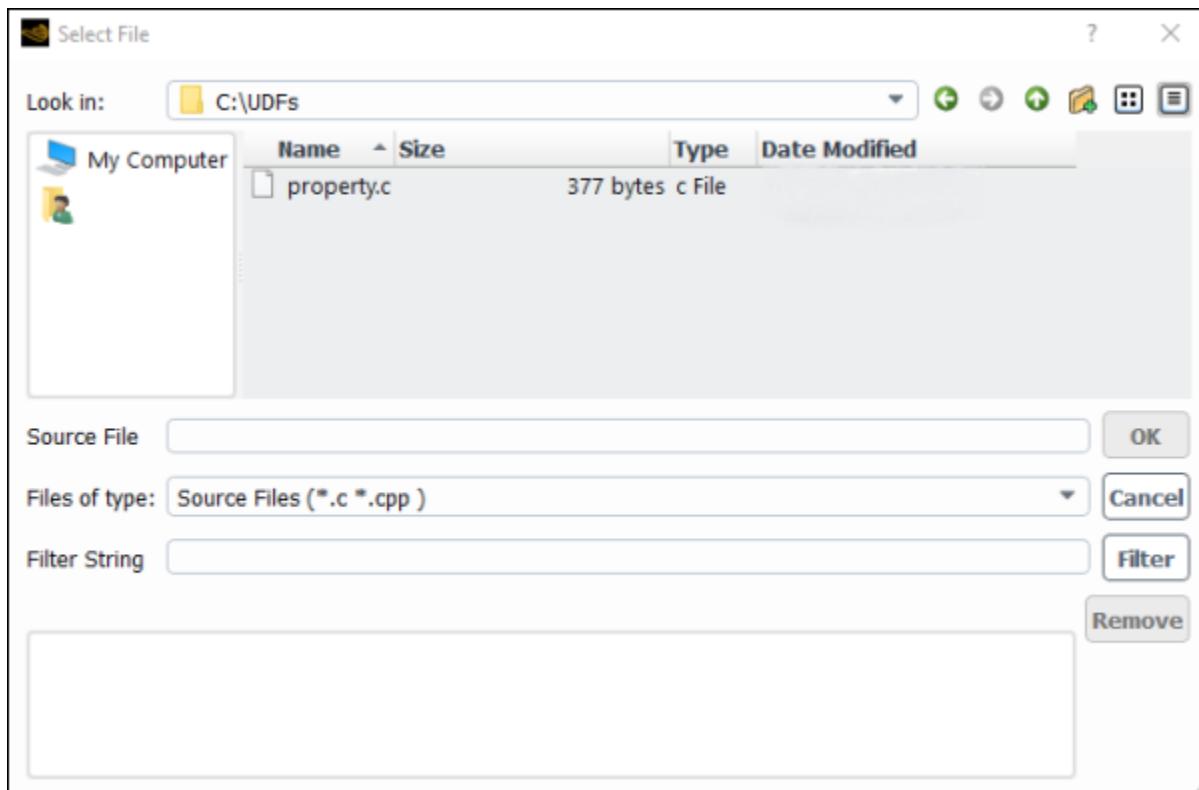
    Important:

    If you are running the parallel version of Ansys Fluent on a network of Windows machines, you *must* "share" the working folder that contains your UDF source, case, and data files so that all of the compute nodes in the cluster can see it. To share the working folder:

    a. Open Windows Explorer and browse to the folder.

    b. Right-click the working folder and select **Sharing and Security** from the menu.

    c. Click **Share this folder**.

    d. Click **OK**.

    ---

2. For Linux, start Ansys Fluent from the directory that contains your case, data, and UDF source files. For Windows, start Ansys Fluent using Fluent Launcher with the following settings:

    *   Specify the folder that contains your case, data, and UDF source files in the **Working Directory** field in the **General Options** tab.

    *   Make sure that the batch file for the UDF compilation environment settings is correctly specified in the **Environment** tab (see Compilers (p. 401) for further details).

3. Read (or set up) your case file.

4. Open the **Compiled UDFs** dialog box (Figure 5.2: The Compiled UDFs Dialog Box (p. 404)).

    ⊡ Parameters & Customization → User Defined Functions 🖱 Compiled...

Figure 5.2: The Compiled UDFs Dialog Box

5. In the **Compiled UDFs** dialog box click **Add...** under **Source Files** to select the UDF source file (or files) you want to compile. This will open the **Select File** dialog box (shown in Figure 5.3: The Select File Dialog Box (p. 404)).
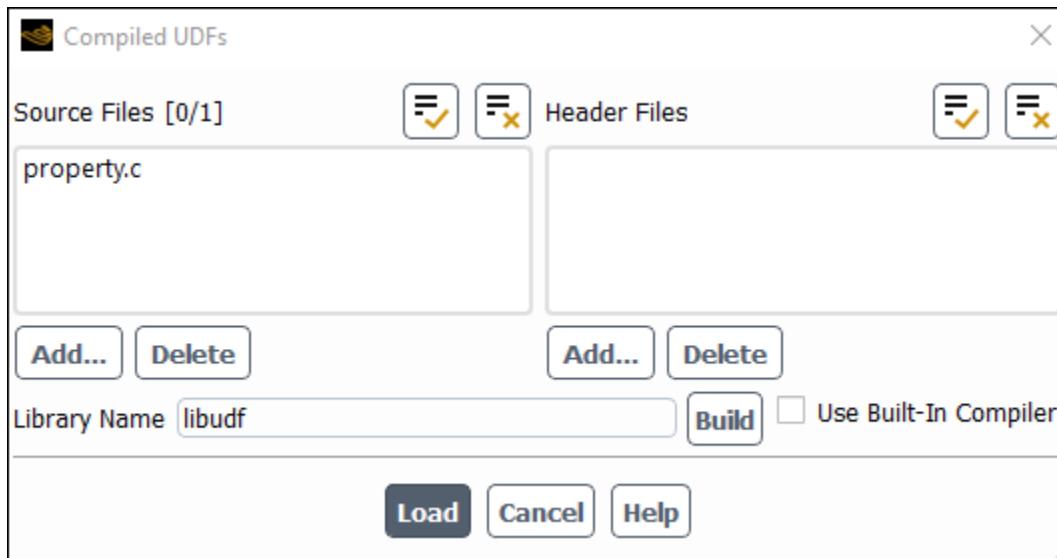
Figure 5.3: The Select File Dialog Box

6. In the **Select File** dialog box, click the names of all of the desired files (for example, `udfexample.c`), so that the complete paths to the source files are displayed under **Source File(s)**. You can remove

---

a selection by clicking the path in **Source File(s)** list and then clicking **Remove**. Click **OK** when your selections are complete.
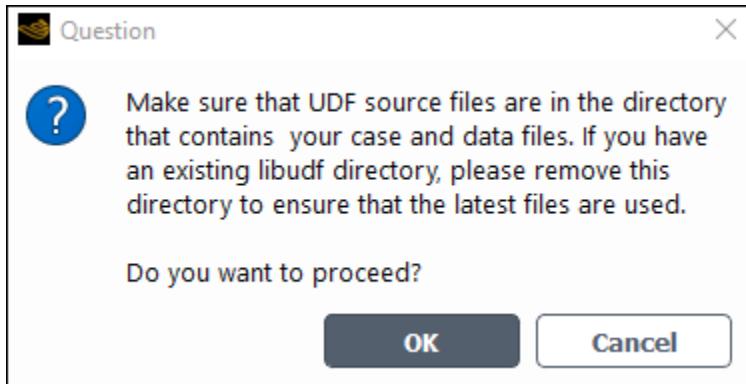
The **Select File** dialog box will close and the file you selected (for example, `udfexample.c`) will appear in the **Source Files** list in the **Compiled UDFs** dialog box (Figure 5.4: The Compiled UDFs Dialog Box (p. 405)). You can delete a file after adding it by selecting the source file and then clicking **Delete** in the **Compiled UDFs** dialog box.

Figure 5.4: The Compiled UDFs Dialog Box

7. In the **Compiled UDFs** dialog box, select additional header files that you want to include in the compilation by clicking **Add...** under **Header File(s)** and repeat the previous step.

8. In the **Compiled UDFs** dialog box (Figure 5.4: The Compiled UDFs Dialog Box (p. 405)), enter the name of the shared library you want to build in the **Library Name** field (or leave the default name **libudf**).

9. If you are running on Windows and have installed an older version of Microsoft Visual Studio or Clang on your machine that is not supported, then you should enable the **Use Built-In Compiler** option, in order to request the use of a built-in compiler (Clang) that is provided with the Fluent installation. If Fluent determines that you have not installed a compiler on your computer, the built-in compiler is used automatically, whether or not you have enabled the **Use Built-In Compiler** option.

10. Click **Build**. All of the UDFs that are contained within each C source file you selected will be compiled and the build files will be stored in the shared library you specified (for example, `libudf`).

    As the compile/build process begins, a **Question** dialog box (Figure 5.5: The Question Dialog Box (p. 406)) will appear reminding you that the source file(s) must be in the same folder as the case and data files. Click **OK** to close the dialog box and continue with the build.

Figure 5.5: The Question Dialog Box

As the build process progresses, the results of the build will be displayed in the console. You can also view the compilation history in the `log` file that is saved in your working folder.

Console messages for a successful compile/build for a source file named `udfexample.c` and a UDF library named `libudf` for a Windows architecture are shown below.

```
Copied E:\/E:\udfexample.c to libudf\src
Creating user_nt.udf file for 3d_host ...
(system "copy C:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\src\udf\makefile_nt.udf "libudf\
win64\3d_host\makefile" ")
        1 file(s) copied.
(chdir "libudf")(chdir "win64\3d_host")# Generating ud_io1.h
udfexample.c
# Generating udf_names.c because of makefile udfexample.obj
udf_names.c
# Linking libudf.dll because of makefile user_nt.udf udf_names.obj udfexample.obj
Microsoft (R) Incremental Linker Version 12.00.31101.0
Copyright (C) Microsoft Corporation.  All rights reserved.

   Creating library libudf.lib and object libudf.exp
Creating user_nt.udf file for 3d_node ...
(system "copy C:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\src\udf\makefile_nt.udf "libudf\
win64\3d_host\makefile" ")
        1 file(s) copied.
(chdir "libudf")(chdir "win64\3d_node")# Generating ud_io1.h
udfexample.c
# Generating udf_names.c because of makefile udfexample.obj
udf_names.c
# Linking libudf.dll because of makefile user_nt.udf udf_names.obj udfexample.obj
Microsoft (R) Incremental Linker Version 12.00.31101.0
Copyright (C) Microsoft Corporation.  All rights reserved.

   Creating library libudf.lib and object libudf.exp

Done.
```

11. In the **Compiled UDFs** dialog box (Figure 5.4: The Compiled UDFs Dialog Box (p. 405)), load the shared library that was just built into Ansys Fluent by clicking **Load**.

    A message will be displayed on the console providing a status of the load process. For example:

```
Opening library "E:\libudf"...
Library "E:\libudf\win64\3d_host\libudf.dll" opened
Opening library "E:\libudf"...
Library "E:\libudf\win64\3d_node\libudf.dll" opened
 stage
Done.
```

indicates that the shared library named **libudf** was successfully loaded (on a Windows machine).

> **Important:**
>
> Note that compiled UDFs are displayed in Ansys Fluent dialog boxes with the associated UDF library name using the `::` identifier. For example, a compiled UDF named `in-let_x_velocity` that is associated with a shared library named `libudf` will appear in Ansys Fluent dialog boxes as `inlet_x_velocity::libudf`. This visually distinguishes UDFs that are compiled from those that are interpreted.

After the compiled UDF(s) become visible and selectable in graphics dialog boxes in Ansys Fluent, they can be hooked to your model. See Hooking UDFs to Ansys Fluent (p. 423) for details. You can use the **UDF Library Manager** dialog box to unload the shared library, if desired. See Load and Unload Libraries Using the UDF Library Manager Dialog Box (p. 418) for details.

12. Write the case file if you want the compiled function(s) in the shared library to be saved with the case. The functions will be loaded *automatically* into Ansys Fluent whenever the case is subsequently read.

> **Important:**
>
> If you do not want the shared library saved with your case file, then you must remember to load it into Ansys Fluent using the **Compiled UDFs** dialog box or the **UDF Library Manager** dialog box in subsequent sessions.

5.3. Compile a UDF Using the TUI

The first step in compiling a UDF source file using the text user interface (TUI) involves setting up the folder structure where the shared (compiled) library will reside, for each of the versions of Ansys Fluent you want to run (that is, `2d_node` and `2d_host`). You will then modify the file named `Makefile` to set up source file parameters. Subsequently, you will execute the `Makefile`, which compiles the source file and builds the shared library from the resulting object files. Finally, you will load the UDF library into Ansys Fluent. Using the TUI option allows you the added advantage of building a shared library for precompiled object file(s) that are derived from non-Ansys Fluent sources (for example, `.o` objects from `.f` sources). See Link Precompiled Object Files From Non-Ansys Fluent Sources (p. 414) for details.

> **Important:**
>
> Note that when running serial or parallel Ansys Fluent on a Windows system, it is recommended that you install a supported version of Microsoft Visual Studio or Clang on your machine, preferably on the `C:` drive. If the compiler you installed is an older version that is not supported, then you should enable the use of a built-in compiler (Clang) that is provided as part of the Fluent installation by entering the following text command prior to compiling: `define/user-defined/use-built-in-compiler? yes`. If Fluent determines that you have not installed a compiler on your computer, the built-in compiler is used automatically, whether or not you have enabled this text command.

For more information, see the following sections:

5.3.1. Set Up the Directory Structure

The folder/directory structures for Windows systems and Linux systems are different, so the procedure for setting up the folder/directory structure is described separately for each.

5.3.1.1. Windows Systems

For compiled UDFs on Windows systems, two Ansys Fluent files are required to build your shared UDF library: `makefile_nt.udf` and `user_nt.udf`. The file `user_nt.udf` has a user-modifiable section that allows you to specify source file parameters.

The procedure below outlines steps that you must follow to set up the folder structure required for the shared library.

1. In your working folder, make a folder that will store your UDF library (for example, `libudf`).

2. Make a folder below this called `src`.

3. Put all your UDF source files into this folder (for example, `libudf\src`).

4. Make an architecture folder below the library folder called `win64` for Intel systems running Windows (for example, `libudf\win64`).

5. In the architecture folder (for example, `libudf\win64`), create folders for the Ansys Fluent versions you want to build for your architecture. (for example, `win64\2d_node` and `win64\2d_host`). Possible versions are:

`2d_node` and `2d_host`	single-precision 2D
`3d_node` and `3d_host`	single-precision 3D
`2ddp_node` and `2ddp_host`	double-precision 2D
`3ddp_node` and `3ddp_host`	double-precision 3D

   Important:

   Note that you must create *two* build folders for each version of the solver (two for the 3D version, two for the 2D double-precision version, and so on), regardless of the number of compute nodes.

6. Copy `user_nt.udf` from

   path`\ANSYS Inc\v212\fluent\fluent21.2.0\src\udf\`

   to all the version sub-folders you have made (for example, `libudf\win64\3d_node` and `libudf\win64\3d_host`).

Note that $path$ is the folder in which you have installed Ansys Fluent (by default, the path is `C:\Program Files`).

7. Copy `makefile_nt.udf` from

   $path$`\ANSYS Inc\v212\fluent\fluent21.2.0\src\udf\`

   to all the version sub-folders you have made (for example, `libudf\win64\3d_node` and `libudf\win64\3d_host`) and rename it `Makefile`.

   Note that $path$ is the folder in which you have installed Ansys Fluent (by default, the path is `C:\Program Files`).

   > **Note:**
   >
   > If you are compiling a UDF outside of the Fluent environment, you need to add the `FLUENT_INC=<your fluent path>` and `FLUENT_ARCH=<machine archi-tecture> (for example, win64)` environment variables to the `<user>_nt.udf` file.

   > **Important:**
   >
   > Be sure the path to your Ansys Fluent home directory is in your command search path environment variable by executing the `setenv.exe` program located in the Ansys Fluent directory (for example, `C:\Program Files\ANSYS Inc\v212\fluent\nt-bin\win64`).

5.3.1.2. Linux Systems

For compiled UDFs on Linux systems, three Ansys Fluent files are required to build your shared UDF library: `makefile.udf`, `makefile.udf2`, and `user.udf`.

The procedure below outlines steps that you must follow to set up the directory structure required for the shared library.

1. In your working directory, make a directory that will store your UDF library (for example, `libudf`).

2. Copy `makefile.udf2` from

   $path$`/ansys_inc/v212/fluent/fluent21.2.0/src/udf`

   to the library directory (for example, `libudf`), and name it `makefile`.

   Note that $path$ is the directory in which you have installed Ansys Fluent.

3. In the library directory you just created in Step 1, make a directory that will store your source file and name it `src`.

4. Copy your source file (for example, `myudf.c`) to the source directory (`src`).

5. Copy `makefile.udf` from

> `path/ansys_inc/v212/fluent/fluent21.2.0/src/udf`

to the `/src` directory, and name it `makefile`.

6. Identify the architecture name of the machine on which you are running (for example, `lnamd64`). This can be done by either typing the command `(fluent-arch)` in the Ansys Fluent TUI window, or running the Ansys Fluent utility program `fluent_arch` at the command line of a Linux shell.

7. In the library directory (for example, `libudf`), create an architecture directory that is named after the architecture identifier determined in the previous step (for example, `lnamd64`).

8. In the architecture directory, create directories named after the Ansys Fluent versions for which you want to build shared libraries (for example, `lnamd64/2d_node` and `lnamd64/2d_host`). Possible versions are:

`2d_node` and `2d_host`	single-precision 2D
`3d_node` and `3d_host`	single-precision 3D
`2ddp_node` and `2ddp_host`	double-precision 2D
`3ddp_node` and `3ddp_host`	double-precision 3D

> **Important:**
>
> Note that you must create *two* build directories for each version of the solver (two for the 3D version, two for the 2D double-precision version, and so on), regardless of the number of compute nodes.

9. Copy `user.udf` from `path/ansys_inc/v212/fluent/fluent21.2.0/src/udf/user.udf` to all the version sub-folders that you have made (for example, `libudf/lnamd64/3d_node` and `libudf/lnamd64/3d_host`).

5.3.2. Build the UDF Library

After you have set up the folder structure and put the files in the proper places, you can compile and build the shared library using the TUI.

5.3.2.1. Windows Systems

1. Edit every `user_nt.udf` file in each version folder to set the following parameters: `CSOURCES`, `HSOURCES`, `VERSION`, and `PARALLEL_NODE`.

   `CSOURCES  =`

   the user-defined source file(s) to be compiled.

   Use the prefix `$(SRC)` before each filename. For example, `$(SRC)udfexample.c` for one file, and `$(SRC)udfexample1.c $(SRC)udfexample2.c` for two files.

HSOURCES =

the user-defined header file(s) to be compiled.

Use the prefix `$(SRC)` before each filename. For example, `$(SRC)udfexample.h` for one file, and `$(SRC)udfexample1.h $(SRC)udfexample2.h` for two files.

VERSION =

the version of the solver you are running which will be the name of the build folder where `user_nt.udf` is located. (`2d_host`, `2d_node`, `3d_host`, `3d_node`, `2ddp_host`, `2ddp_node`, `3ddp_host`, or `3ddp_node`).

PARALLEL_NODE =

the parallel communications library.

Specify `none` for the host version, or one of the following for the node version:

`intel`: parallel using Intel MPI

`msmpi`: parallel using Microsoft MPI

Important:

Be sure to edit *both* copies of `user_nt.udf` (the one in the host folder and the one in the node folder), and specify the appropriate `CSOURCES`, `HSOURCES`, `VERSION`, and `PARALLEL_NODE` in each file. Set `PARALLEL_NODE = none` for the host version and one of the other options (that is, `intel` or `msmpi`) for the node version, depending on which message passing method you are going to use.

An excerpt from a sample `user_nt.udf` file is shown below:

```
# Replace text in " " (and remove quotes)
# | indicates a choice
# note: $(SRC) is defined in the Makefile

  CSOURCES = $(SRC)udfexample.c
  HSOURCES = $(SRC)udfexample.h
  VERSION = 2d_host
  PARALLEL_NODE = none
```

2. If you have installed a supported compiler, go to each version folder (for example, `libudf\win64\2d_node` and `libudf\win64\2d_host`) in the appropriate command prompt window and compile the UDF using the `nmake` command (for Microsoft Visual Studio) or the `scons` command (for Clang). The following is an example using the Visual Studio command prompt window:

```
  C:\users\user_name\work_dir\libudf\win64\2d_node>nmake
```

The following messages will be displayed:

```
  Microsoft (R) Program Maintenance Utility Version 7.10.3077
 Copyright (C) Microsoft Corporation. All rights reserved.
  cl /c /Za /DUDF_EXPORTING
```

```
    -Ic:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\win64\2d_node
    -Ic:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\src
    -Ic:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\cortex\src
    -Ic:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\client\src
    -Ic:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\tgrid\src
    -Ic:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\multiport\src
        ..\..\src\udfexample.c
Microsoft (R) 32-bit C/C++ Standard Compiler Version 13.10.3077 for 80x86
Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.

 udfexample.c
 # Generating udf_names.c because of Makefile udfexample.obj
        cl /c /Za /DUDF_EXPORTING
    -Ic:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\win64\2d_node
    -Ic:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\src
    -Ic:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\cortex\src
    -Ic:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\client\src
    -Ic:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\tgrid\src
    -Ic:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\multiport\src
        udf_names.c
Microsoft (R) 32-bit C/C++ Standard Compiler Version 13.10.3077 for 80x86
Copyright (C) Microsoft Corporation 1984-2002. All rights reserved.

 udf_names.c
 # Linking libudf.dll because of Makefile user_nt.udf
udf_names.obj udfexample.obj
link /Libpath:c:\Program Files\ANSYS Inc\v212\fluent\fluent21.2.0\win64\2d_node
        /dll
    /out:libudf.dl
l udf_names.obj udfexample.obj  fl2129s.lib
Microsoft (R) Incremental Linker Version 7.10.3077
Copyright (C) Microsoft Corporation. All rights reserved.

    Creating library libudf.lib and object libudf.exp

 C:\Program Files\ANSYS Inc\v212\fluent\ntbin\win64\libudf\win64\2d_node>
```

Important:

Note that if there are problems with the build, you can do a complete rebuild by typing `nmake clean` and then `nmake` again.

---

3. If you installed an older version of Microsoft Visual Studio or Clang on your machine that is not supported, enter the following text command: `define/user-defined/use-built-in-compiler? yes`. This ensures the use of a built-in compiler (Clang) that is provided with the Fluent installation. If Fluent determines that you have not installed a compiler on your computer, the built-in compiler is used automatically, whether or not you have enabled this text command.

Then compile the UDF using the following text command: `define/user-defined/compiled-functions`.

5.3.2.2. Linux Systems

1. Edit every `user.udf` file in each version folder to set the following parameters: `CSOURCES`, `HSOURCES`, and Ansys Fluent path.

CSOURCES =

The name of your source file(s) (for example, `udfexample.c`). Multiple sources can be specified by using a space delimiter (for example, `udfexample1.c udfexample2.c`).

HSOURCES =

The name of your header file(s) (for example, `udfexample.h`). Multiple headers can be specified by using a space delimiter (for example, `udfexample1.h udfexample2.h`).

FLUENT_INC =

The path to your release directory.

2. An excerpt from a sample `user.udf` file is shown below:

```
CSOURCES = udfexample.c
HSOURCES = udfexample.h
FLUENT_INC=/path/ansys_inc/v212/fluent
```

In the previous example, `path` represents the directory where you installed Ansys Fluent.

3. In your library directory (for example, `libudf`), execute the `Makefile` by typing a command that begins with `make` and includes the architecture of the machine you will run Ansys Fluent on, which you identified in a previous step. For example, for the Linux (`lnamd64`) architecture type:

```
make "FLUENT_ARCH=lnamd64"
```

Ansys Fluent will build a shared library for each version you created a directory for (Set Up the Directory Structure (p. 408)) and will display messages about the compile/build process in the console. You can view the compilation history in the `log` file that is saved in your working directory.

For example, when compiling/building a shared library for a source file named `profile.c` and a UDF library named `libudf` on a Linux architecture, the console messages may include the following:

```
Working...
 for d in lnamd64[23]*; do \
   ( \
    cd $d; \
    for f in ../../src*.[ch] ../../src/Makefile; do \
      if [ ! -f 'basename $f' ]; then \
       echo "# linking to " $f "in" $d; \
       ln -s $f .; \
      fi; \
    done; \
    echo ""; \
    echo "# building library in" $d; \
    make -k>makelog 2>&1; \
    cat makelog; \
 ) \
 done
 # linking to ...  myudf.c in lnamd64/2d_node

 # building library in lnamd64/2d_node
 make[1]: Entering directory ..../udf_names.c
 # Generating udf_names
 make[2]: Entering directory ..../profile.c
```

```
make libudf.so ...
# Compiling udf_names.o ...
# Compiling profile.o ...
# Linking libudf.so ...
make[2]: Leaving directory ..../udf_names.c
make[1]: Leaving directory ..../profile.c

You can also see the 'log'-file in
the working directory for compilation history
Done.
```

5.3.3. Load the UDF Library

You can load the shared library you compiled and built using the GUI from the **Compiled UDFs** dialog box or the **UDF Library Manager** dialog box. Follow the procedure outlined in Step 11. of Compiling a UDF Using the GUI (p. 403) or in Load and Unload Libraries Using the UDF Library Manager Dialog Box (p. 418), respectively.

5.4. Link Precompiled Object Files From Non-Ansys Fluent Sources

Ansys Fluent allows you to build a shared library for precompiled object files that are derived from external sources using the text user interface (TUI) option. For example, you can link precompiled objects derived from FORTRAN sources (`.o` objects from `.f` sources) to Ansys Fluent for use by a UDF. The following sections describe the procedures for doing this on a Windows system and a Linux system.

For more information, see the following sections:

5.4.1. Windows Systems
5.4.2. Linux Systems
5.4.3. Example: Link Precompiled Objects to Ansys Fluent

5.4.1. Windows Systems

1. Follow the procedure described in Set Up the Directory Structure (p. 408).

2. Copy your precompiled object files (for example, `myobject1.obj myobject2.obj`) to all of the architecture/version folders you created in Step 1 (for example, `win64/2d_node`, `win64/2d_host`).

> **Important:**
>
> The object files should be compiled using similar flags to those used by Ansys Fluent (for example, `/c /Za`).

3. Edit the `user_nt.udf` files in each architecture/version folder.

5.4.2. Linux Systems

1. Follow the procedure described in Set Up the Directory Structure (p. 408).

2. Copy your precompiled object files (for example, `myobject1.o myobject2.o`) to all of the architecture/version directories you created in Step 1 (for example, `lnamd64/2d_node` and `lnamd64/2d_host`).

> **Important:**
>
> The object files should be compiled using similar flags to those used for Ansys Fluent. Common flags used by Ansys Fluent are: `-KPIC`, `-O`, and `-ansi` which often have equivalents such as `-fpic`, `-O3`, and `-xansi`.

3. Using a text editor, edit the file `user.udf` in each version folder to set the following parameters: `CSOURCES`, `HSOURCES`, and the Ansys Fluent path.

`CSOURCES =`	Put the names of your UDF C files here. They will be calling the functions in the User Objects.
`HSOURCES =`	Put the names of your UDF H files here.
`FLUENT_INC =`	The path to your release directory.

An excerpt from a sample `user.udf` is shown below:

```
CSOURCES = udfexample.c
HSOURCES = udfexample.h
FLUENT_INC=/path/ansys_inc/v212/fluent
```

In the previous example, *path* represents the directory where you installed Ansys Fluent.

4. Using a text editor, edit the file `Makefile` in your `src` directory to set the `USER_OBJECTS` parameters:

`USER_OBJECTS =`	The precompiled object file(s) that you want to build a shared library for (for example, `myobject1.o`). Use a space delimiter to specify multiple object files (for example, `myobject1.o myobject2.o`).

An excerpt from a sample `Makefile` is shown below:

```
#-----------------------------------------------------------#
# Makefile for user defined functions
#
#-----------------------------------------------------------#
# User modifiable section.
#-----------------------------------------------------------#
include user.udf

# Precompiled User Object files (for example .o files from .f sources)
USER_OBJECTS= myobject1.o myobject2.o

#-----------------------------------------------------------#
# Build targets (do not modify below this line).
#-----------------------------------------------------------#
  .
```

```
    .
    .
```

5. In your library directory (for example, `libudf`), execute the `Makefile` by typing a command that begins with `make` and includes the architecture of the machine on which you will run Ansys Fluent, which you identified in a previous step (for example, `lnamd64`).

    ```
    make "FLUENT_ARCH=lnamd64"
    ```

    The following messages will be displayed:

    ```
    # linking to ../../src/Makefile in lnamd64/2d_node
    # building library in lnamd64/2d_node
    # linking to ../../src/Makefile in lnamd64/2d_host
    # building library in lnamd64/2d_host
    ```

5.4.3. Example: Link Precompiled Objects to Ansys Fluent

The following example demonstrates the linking of a FORTRAN object file `test.o` to Ansys Fluent, for use in a UDF named `test_use.c`. This particular UDF is not a practical application but has rather been designed to demonstrate the functionality. It uses data from a FORTRAN-derived object file to display parameters that are passed to the C function named `fort_test`. This on-demand UDF, when executed from the **User-Defined Function Hooks** dialog box, displays the values of the FORTRAN parameters and the common block and common complex numbers that are computed by the UDF, using the FORTRAN parameters.

> **Important:**
>
> Note that the names of the functions and data structures have been changed from the capital form in FORTRAN (for example, `ADDAB` is changed to `addab_`). This name "mangling" is done by the compiler and is strongly system-dependent. Note also that functions returning complex numbers have different forms on different machine types, since C can return only single values and not structures. Consult your system and compiler manuals for details.

1. In the first step of this example, a FORTRAN source file named `test.f` is compiled and the resulting object file (`test.o`) is placed in the shared library folder for the `lnamd64/2d_node` version.

    ```
    libudf/lnamd64/2d_node
    ```

    The source listing for `test.f` is shown below.

    ```
    C FORTRAN function
    C test.f
    C
    C compile to .o file using:
    C f77 -KPIC -n32 -O -c test.f (irix6 & suns)

         REAL*8 FUNCTION ADDAB(A,B,C)

         REAL A
         REAL*8 B
         REAL*8 YCOM
         COMPLEX ZCOM
         INTEGER C
         INTEGER SIZE
    ```

```
      COMMON //SIZE,ARRAY(10)
      COMMON /TSTCOM/ICOM,XCOM,YCOM,ZCOM

      ICOM=C
      XCOM=A
      YCOM=B
      ZCOM=CMPLX(A,REAL(B))

      SIZE=10
      DO 100 I=1,SIZE
     ARRAY(I)=I*A
100 CONTINUE

      ADDAB=(A*C)*B
      END

      COMPLEX FUNCTION CCMPLX(A,B)
      REAL A,B

     CCMPLX=CMPLX(A,B)
      END
```

2. The UDF C source file named `test_use.c` is placed in the source folder for the `lnamd64/2d_node` version:

```
src/lnamd64/2d_node
```

The source listing for `test_use.c` is as follows.

```
#include "udf.h"
#if defined(_WIN32)
    /* Visual Fortran makes uppercase functions provide lowercase
       mapping to be compatible with Linux code */
# define addab_ ADDAB
#endif

typedef struct {float r,i;} Complex;
typedef struct {double r,i;} DComplex;
typedef struct {long double r,i;} QComplex; /* FORTRAN QUAD
                 PRECISION */

/* FORTRAN FUNCTION */
extern double addab_(float *a,double *b,int *c);

/* NOTE on SUN machines that FORTRAN functions returning a complex
 number are actually implemented as void but with an extra
 initial argument.*/

extern void ccmplx_(Complex *z,float *a,float *b);
extern void qcmplx_(QComplex *z,float *a,float *b);

/* BLANK COMMON BLOCK */
extern struct
{
  int size;
  float array[10];
} _BLNK__;

/* FORTRAN NAMED COMMON BLOCK */
extern struct
{
  int int_c;
  float float_a;
  double double_b;
  float cmplx_r;
```

```
      float cmplx_i;
    } tstcom_;

    DEFINE_ON_DEMAND(fort_test)
    {
      float a=3.0,float_b;
      double d,b=1.5;
      int i,c=2;
      Complex z;
      QComplex qz;

      d = addab_(&a,&b,&c);
      Message("\n\nFortran code gives (%f * %d) * %f = %f\n",a,c,b,d);
      Message("Common Block TSTCOM set to: %g
              tstcom_.float_a,tstcom_.double_b,tstcom_.int_c);
      Message("Common Complex Number is (%f + %fj)\n",
              tstcom_.cmplx_r,tstcom_.cmplx_i);
      Message("BLANK Common Block has an array of size
               \n",_BLNK__.size);
      for (i=0; i <_BLNK__.size ; i++)
        {
          Message("array[%d] = %g\n",i,_BLNK__.array[i]);
        }
      float_b=(float)b;
      ccmplx_(&z,&a,&float_b);
      Message("Function CCMPLX returns Complex Number:
                 (%g + %gj)\n",z.r,z.i);
      qcmplx_(&qz,&a,&float_b);
      Message("Function QCMPLX returns Complex Number:
                 (%g + %gj)\n",qz.r,qz.i);
    }
```

3. The `user.udf` is then modified to specify the UDF C source file (`test_use.c`) as shown below.

    ```
    CSOURCES = test_use.c
    HSOURCES =
    FLUENT_INC=/path/ansys_inc/v212/fluent
    ```

    Note that in the previous example, *path* represents the directory where you installed Ansys Fluent.

4. The `Makefile` is then modified to specify the external object file (`test.o`) as shown below.

    ```
    #-----------------------------------------------------------#
    # User modifiable section.
    #-----------------------------------------------------------#
    include user.udf

    # Precompiled User Object files (for example .o files from .f sources)
    USER_OBJECTS= test.o
    ```

5. Finally, the `Makefile` is executed by issuing the following command in the `libudf` folder:

    ```
    make "FLUENT_ARCH=lnamd64"
    ```

5.5. Load and Unload Libraries Using the UDF Library Manager Dialog Box

You can use the **UDF Library Manager** dialog box to load and unload multiple shared libraries in Ansys Fluent.

For more information, see the following sections:

5.5.1. Load the UDF Library

To load a UDF library in Ansys Fluent, open the **UDF Library Manager** dialog box (Figure 5.6: The UDF Library Manager Dialog Box (p. 419)).

 Parameters & Customization → **User Defined Functions** **Manage...**

Figure 5.6: The UDF Library Manager Dialog Box

In the **UDF Library Manager** dialog box, type the name of the shared library in the **Library Name** field and click **Load** (Figure 5.6: The UDF Library Manager Dialog Box (p. 419)).

A message will be displayed in the console providing the status of the load process. For example:
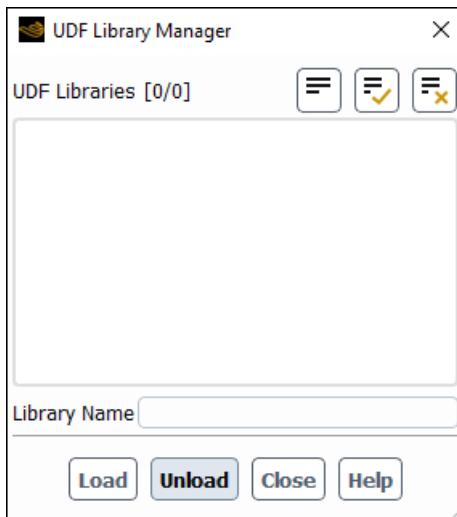
```
Opening library "libudf"...
 Library "libudf\win64\3d_node\libudf.dll" opened
   inlet_x_velocity
 Done.
```

indicates that the shared library named **libudf** was successfully loaded and contains one UDF named `inlet_x_velocity`. In the **UDF Library Manager** dialog box, the library name (for example, `libudf`) will be added under **UDF Libraries**. Repeat this step to load additional libraries.

5.5.2. Unload the UDF Library

To unload a UDF library in Ansys Fluent, open the **UDF Library Manager** dialog box (Figure 5.7: The UDF Library Manager Dialog Box (p. 420)).

 Parameters & Customization → **User Defined Functions** **Manage...**

Figure 5.7: The UDF Library Manager Dialog Box

In the **UDF Library Manager** dialog box, highlight the shared library name (for example, `libudf`) that is listed under **UDF Libraries** (or type the **Library Name**) and click **Unload** (Figure 5.7: The UDF Library Manager Dialog Box (p. 420)).

After it is unloaded, the library (for example, `libudf`) will be removed from the **UDF Libraries** list in the dialog box. Repeat this step to unload additional libraries.

5.6. Common Errors When Building and Loading a UDF Library

A common compiler error occurs when you forget to put an `#include "udf.h"` statement at the beginning of your source file. You'll get a long list of compiler error messages that include illegal declarations of variables. Similarly, if your function requires an auxiliary header file (for example, `sg_pdf.h`) and you forgot to include it, you'll get a similar compiler error message.

Another common error occurs when the argument list for a `DEFINE` statement is placed on multiple lines. (All `DEFINE` macro arguments must be listed on the same line in a C file.) The compiler will typically not report any error message but it will report a single warning message in the log file to indicate that this occurred:

```
warning: no newline at end of file
```

If your compiled UDF library loads successfully, then each function contained within the library will be reported to the console (and log file). For example, if you built a shared library named `libudf` containing two user-defined functions `superfluid_density` and `speed_sound`, a successful library load (on a Linux machine) will result in the following message being reported to the console (and log file):

```
Opening library "libudf"...
Library "path/libudf/lnamd64/3d_node/libudf.so" opened
   superfluid_density
   speed_sound
Done.
```

If, instead, no function names are listed, then it is likely that your source file did not successfully compile. In this case, you must consult the log to view the compilation history, and debug your function(s). Note

that you must unload the UDF library using the **UDF Library Manager** dialog box before you reload the debugged version of your library.

Another common error occurs when you try to read a case file that was saved with a shared library, and that shared library has subsequently been moved to another location. In this case, the following error will be reported to the console (and log file) on a Linux machine:

```
Opening library "path/libudf"...
Error: No such file or directory:
path/libudf/lnamd64/3d_node/libudf.so
```

Similarly, you will get an error message when you try to load a shared library before it has been built.

```
Error: UDF library "libudf" not available at path
```

For more information, see the following section:

5.6.1. Windows Distributed Parallel

Note:

If you have a source file that contains DOS-style line endings, before you can compile the source file in Ansys Fluent on Linux, you must first run the `dos2unix` utility (for example, `dos2unix` *filename*`.c`) in the command line in order to make the source file compatible with the Ansys Fluent Linux compiler.

5.6.1. Windows Distributed Parallel

If you are trying to load a compiled UDF while running Ansys Fluent in distributed parallel, you may receive this error:

```
Error: open_udf_library: The system cannot find the path specified
```

This error occurs because the other computer(s) on the cluster cannot "see" the UDF through the network. To remedy this, you must:

1. Modify the environment variables on the computer where the compiled UDF, case, and data files reside

2. Share the folder where the files reside. See Compiling a UDF Using the GUI (p. 403) for details on file sharing or contact Ansys Fluent installation support for additional assistance.

There are instances when Ansys Fluent can hang when trying to read a compiled UDF using network parallel as a result of a network communicator problem. Contact Ansys Fluent installation support for details.

You may receive an error message when you invoke the command `nmake` or `scons` if you have the wrong compiler installed or if you have not launched the compiler command prompt window prior to building the UDF. See Compilers (p. 401) and Compiling a UDF Using the GUI (p. 403) for details or contact Ansys Fluent installation support for further assistance.

5.7. Special Considerations for Parallel Ansys Fluent

If you are running serial or parallel Ansys Fluent on a Windows system and intend to compile a UDF, it is recommended that you install a supported version of Microsoft Visual Studio or Clang on your machine, preferably on the `C:` drive. If the compiler you installed is an older version that is not supported, then you should enable the use of a built-in compiler (Clang) that is provided as part of the Fluent installation by entering the following text command prior to compiling: `define/user-defined/use-built-in-compiler? yes`. If Fluent determines that you have not installed a compiler on your computer, the built-in compiler is used automatically, whether or not you have enabled this text command.

Also note that if you have compiled a UDF while running Ansys Fluent on a Windows parallel network, you *must* 'share' the folder where the UDF is located so that all computers on the cluster can see this folder. To share the folder in which the case, data, and compiled UDF reside, using the Windows Explorer right-click the folder, choose **Sharing...** from the menu, click **Share this folder**, and then click **OK**.

> **Important:**
>
> If you forget to enable the sharing option for the folder using the Windows Explorer, then Ansys Fluent will hang when you try to load the library in the **Compiled UDFs** dialog box.

See Common Errors When Building and Loading a UDF Library (p. 420) for a list of errors you can encounter that are specific to Windows parallel.

Chapter 6: Hooking UDFs to Ansys Fluent

After you have interpreted or compiled your UDF using the methods described in Interpreting UD-Fs (p. 393) and Compiling UDFs (p. 399), respectively, you are ready to hook the function to Ansys Fluent using a graphic interface dialog box. After it is hooked, the function will be utilized in your Ansys Fluent model. Details about hooking a UDF to Ansys Fluent can be found in the following sections. Note that these sections relate to corresponding sections in `DEFINE` Macros (p. 19).

6.1. Hooking General Purpose UDFs

This section contains methods for hooking general purpose UDFs to Ansys Fluent. General purpose UDFs are those that have been defined using macros described in General Purpose `DEFINE` Macros (p. 20) and then interpreted or compiled and loaded using methods described in Interpreting UD-Fs (p. 393) or Compiling UDFs (p. 399), respectively.

For more information, see the following sections:

6.1.1. Hooking `DEFINE_ADJUST` UDFs

After you interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_ADJUST` UDF, the name of the function you supplied as a `DEFINE` macro argument can be hooked using the **User-Defined Function Hooks** dialog box (Figure 6.1: The User-Defined Function Hooks Dialog Box (p. 424)). Note that you can hook multiple adjust UDFs to your model, if desired.

To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box (Figure 6.1: The User-Defined Function Hooks Dialog Box (p. 424)).

⊟ **Parameters & Customization** → **User Defined Functions** 🖱 **Function Hooks...**
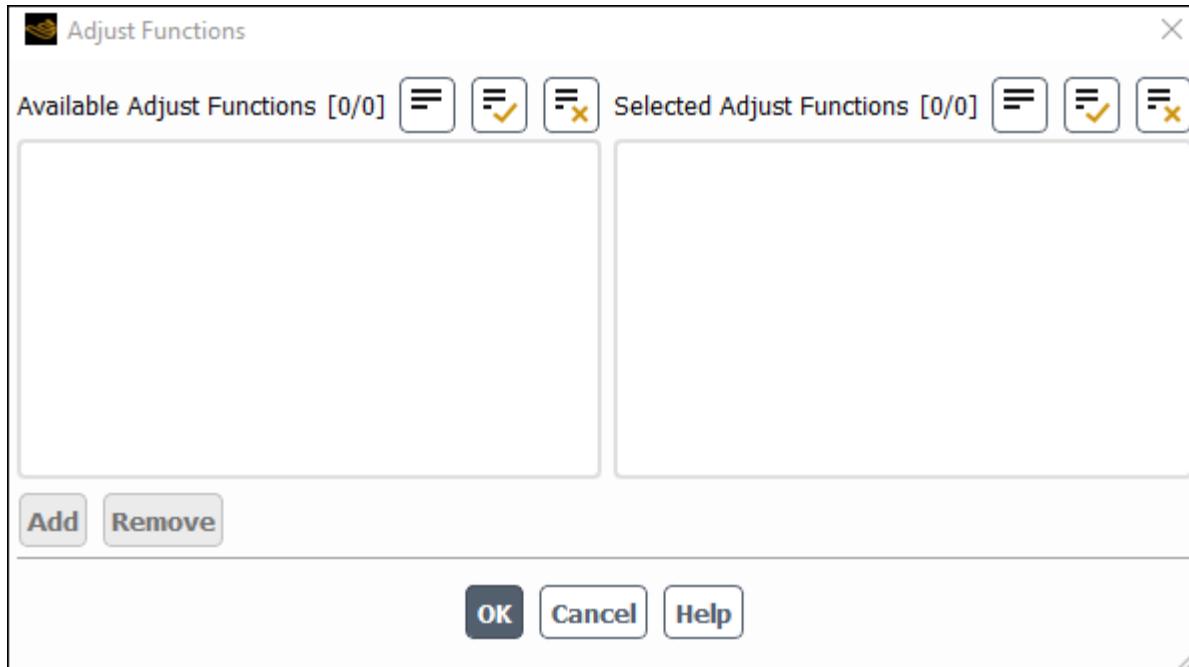
Figure 6.1: The User-Defined Function Hooks Dialog Box

Click the **Edit...** button next to **Adjust** to open the **Adjust Functions** dialog box (Figure 6.2: The Adjust Functions Dialog Box (p. 425)).

Figure 6.2: The Adjust Functions Dialog Box

Select the function(s) you want to hook to your model from the **Available Adjust Functions** list. Click **Add** and then **OK** to close the dialog box. The name of the function you selected will be displayed in the **Adjust** field of the **User-Defined Function Hooks** dialog box. If you select more than one function, the number will be displayed (for example, **2 selected**). Click **OK** in the **User-Defined Function Hooks** dialog box to apply the settings.

See `DEFINE_ADJUST` (p. 21) for details about defining adjust functions using the `DEFINE_ADJUST` macro.

6.1.2. Hooking `DEFINE_DELTAT` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DELTAT` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Run Calculation** task page in Ansys Fluent.

To hook the UDF to Ansys Fluent, right-click the **General** branch of the tree and select **Transient** from the **Analysis Type** sub-menu.

Setup → **General** Analysis Type → **Transient**

Then open the **Run Calculation** task page.

Solution → Run Calculation

Select **User-Defined Function** from the **Type** drop-down list, and then select the function name (for example, **mydeltat::libudf**) from the **User-Defined Time Step Size** drop-down list.

See `DEFINE_DELTAT` (p. 23) for details about defining `DEFINE_DELTAT` functions.
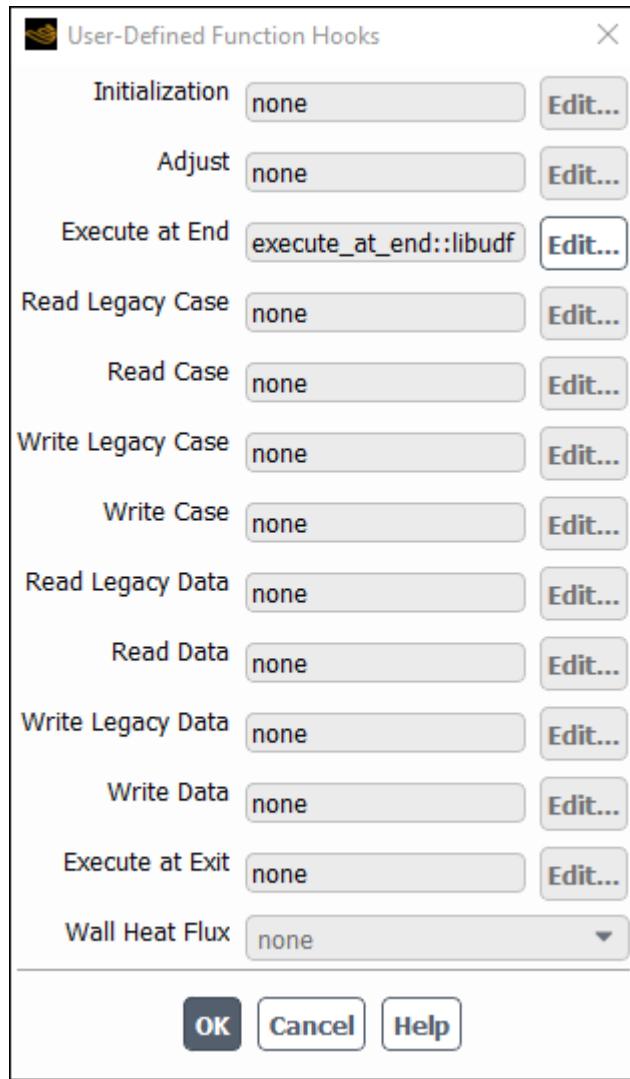
6.1.3. Hooking `DEFINE_EXECUTE_AT_END` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_EXECUTE_AT_END` UDF, it is ready to be hooked to Ansys Fluent. Note that you can hook multiple at-end UDFs to your model, if desired.
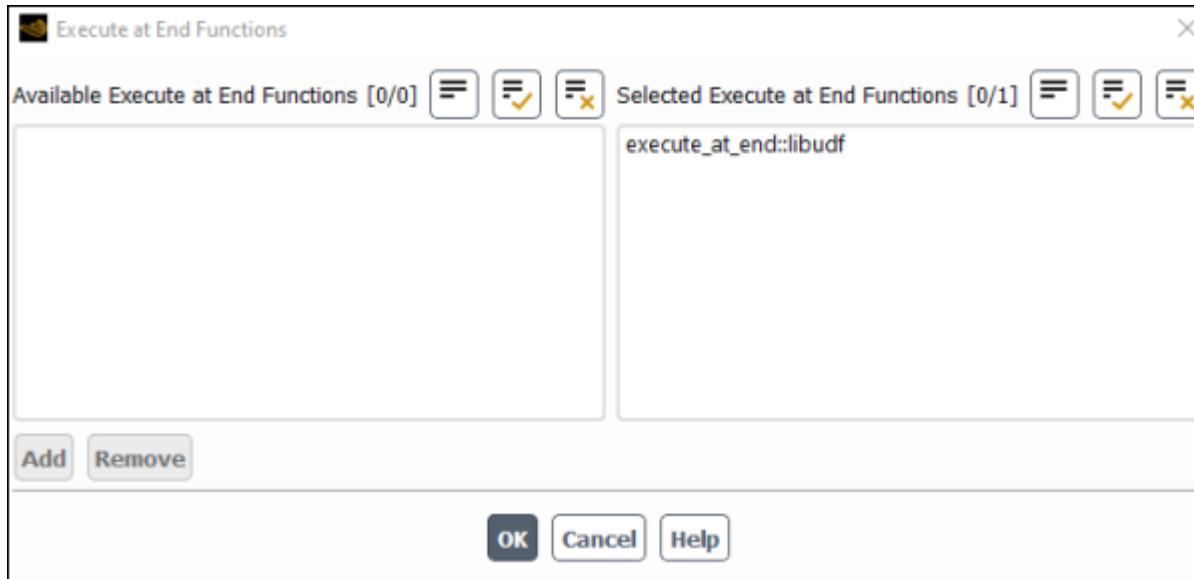
To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box (Figure 6.3: The User-Defined Function Hooks Dialog Box (p. 426)).

📇 **Parameters & Customization** → **User Defined Functions** 🖱 **Function Hooks…**

Figure 6.3: The User-Defined Function Hooks Dialog Box

Click the **Edit…** button next to **Execute At End** to open the **Execute At End Functions** dialog box (Figure 6.4: The Execute At End Functions Dialog Box (p. 427)).

Figure 6.4: The Execute At End Functions Dialog Box

Select the function(s) you want to hook to your model from the **Available Execute at End Functions** list. Click **Add** and then **OK** to close the dialog box. The name of the function you selected will be displayed in the **Execute at End** field of the **User-Defined Function Hooks** dialog box. If you select more than one function, the number will be displayed (for example, **2 selected**). Click **OK** in the **User-Defined Function Hooks** dialog box to apply the settings.
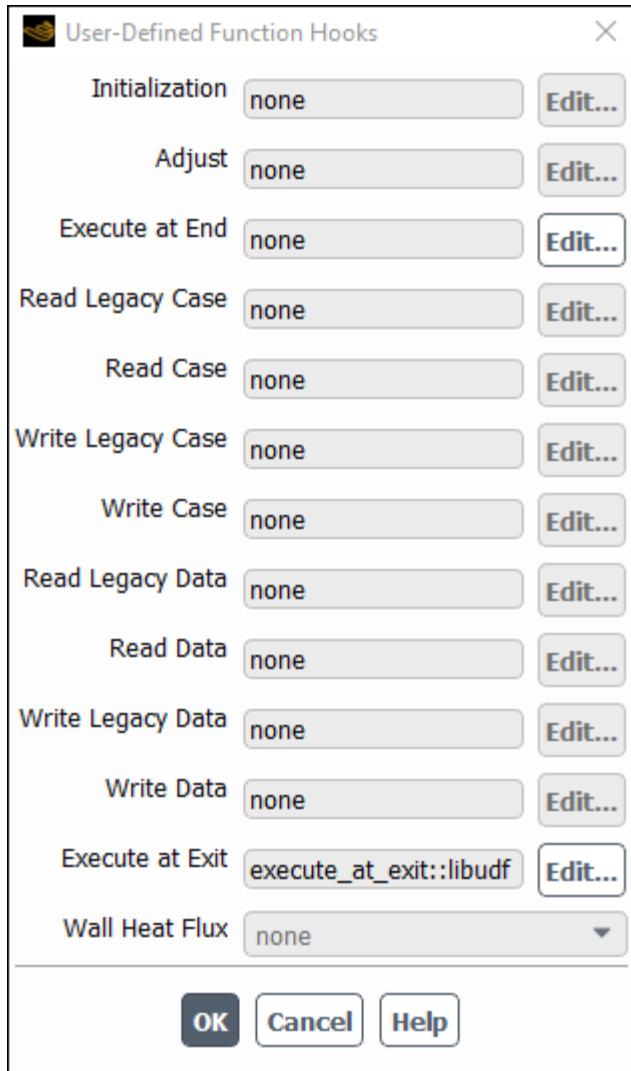
See `DEFINE_EXECUTE_AT_END` (p. 24) for details about defining `DEFINE_EXECUTE_AT_END` functions.

6.1.4. Hooking `DEFINE_EXECUTE_AT_EXIT` UDFs
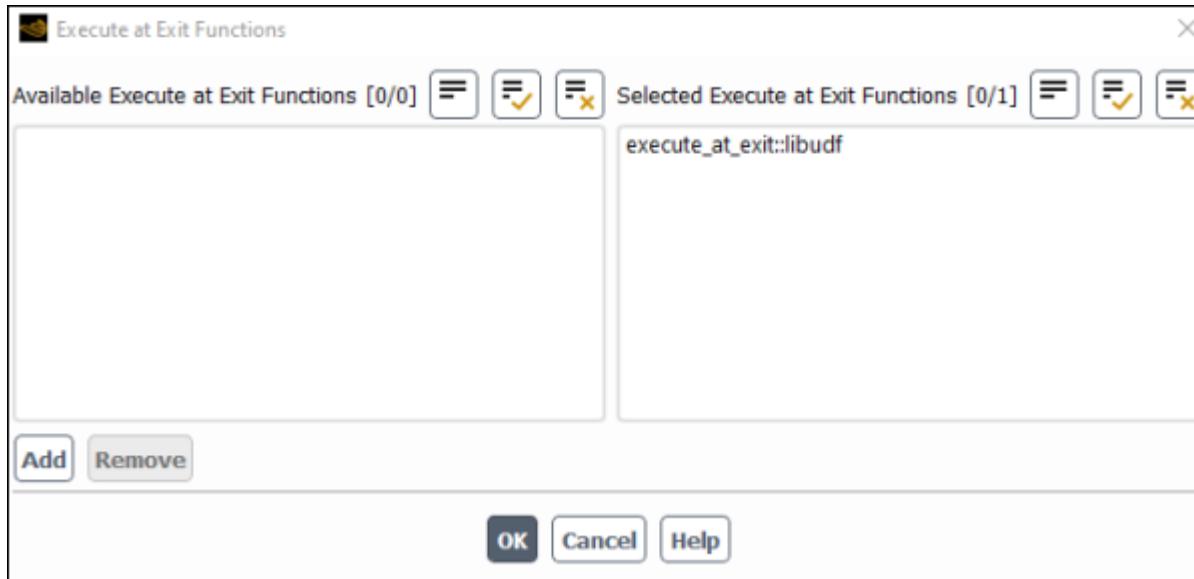
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_EXECUTE_AT_EXIT` UDF, it is ready to be hooked to Ansys Fluent. Note that you can hook multiple at-exit UDFs to your model, if desired.

To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box (Figure 6.5: The User-Defined Function Hooks Dialog Box (p. 428)).

 Parameters & Customization → **User Defined Functions** **Function Hooks...**

Figure 6.5: The User-Defined Function Hooks Dialog Box

Click the **Edit...** button next to **Execute at Exit** to open the **Execute at Exit Functions** dialog box (Figure 6.6: The Execute at Exit Functions Dialog Box (p. 429)).

Figure 6.6: The Execute at Exit Functions Dialog Box

Select the function(s) you want to hook to your model from the **Available Execute at Exit Functions** list. Click **Add** and then **OK** to close the dialog box. The name of the function you selected will be displayed in the **Execute at Exit** field of the **User-Defined Function Hooks** dialog box. If you select more than one function, the number will be displayed (for example, **2 selected**). Click **OK** in the **User-Defined Function Hooks** dialog box to apply the settings.
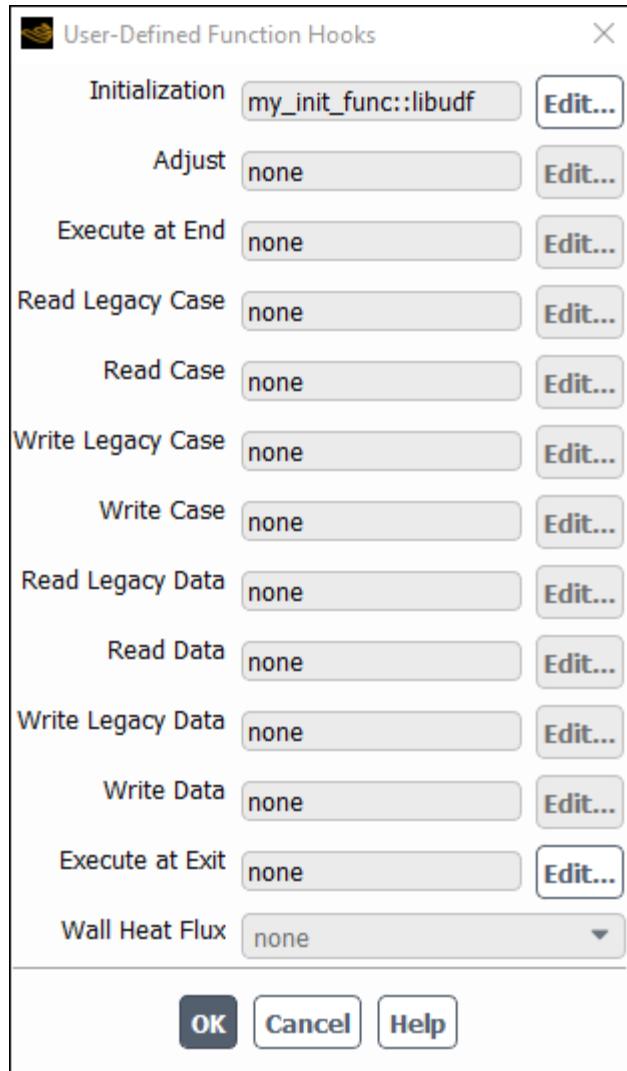
See `DEFINE_EXECUTE_AT_EXIT` (p. 25) for details about defining `DEFINE_EXECUTE_AT_EXIT` functions.

6.1.5. Hooking `DEFINE_INIT` UDFs
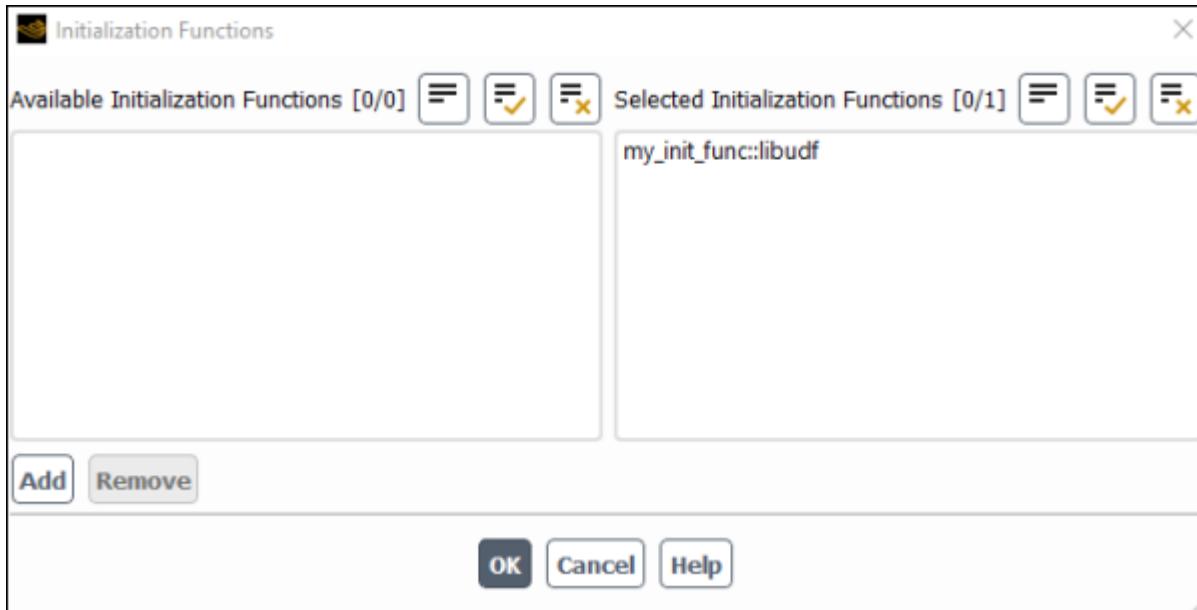
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_INIT` UDF, it is ready to be hooked to Ansys Fluent. Note that you can hook multiple initialization UDFs to your model, if desired.

To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box (Figure 6.7: The User-Defined Function Hooks Dialog Box (p. 430)).

 Parameters & Customization → **User Defined Functions** **Function Hooks...**

Figure 6.7: The User-Defined Function Hooks Dialog Box

Click the **Edit...** button next to **Initialization** to open the **Initialization Functions** dialog box (Figure 6.8: The Initialization Functions Dialog Box (p. 431)).

Figure 6.8: The Initialization Functions Dialog Box

Select the function(s) you want to hook to your model from the **Available Initialization Functions** list. Click **Add** and then **OK** to close the dialog box. The name of the function you selected will be displayed in the **Initialization** field of the **User-Defined Function Hooks** dialog box. If you select more than one function, the number will be displayed (for example, **2 selected**). Click **OK** in the **User-Defined Function Hooks** dialog box to apply the settings.

See `DEFINE_INIT` (p. 31) for details about defining `DEFINE_INIT` functions.

6.1.6. Hooking `DEFINE_ON_DEMAND` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_ON_DEMAND` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Execute On Demand** dialog box (Figure 6.9: The Execute On Demand Dialog Box (p. 431)) in Ansys Fluent.

To hook the UDF to Ansys Fluent, open the **Execute On Demand** dialog box.

📋 **Parameters & Customization** → **User Defined Functions** 🖱 **Execute on Demand...**

Figure 6.9: The Execute On Demand Dialog Box

Select the function name (for example, **on_demand_calc::libudf**) in the **Execute On Demand** drop-down list and click **Execute**. Ansys Fluent will execute the UDF immediately. Click **Close** to close the dialog box.

See `DEFINE_ON_DEMAND` (p. 33) for details about defining `DEFINE_ON_DEMAND` functions.
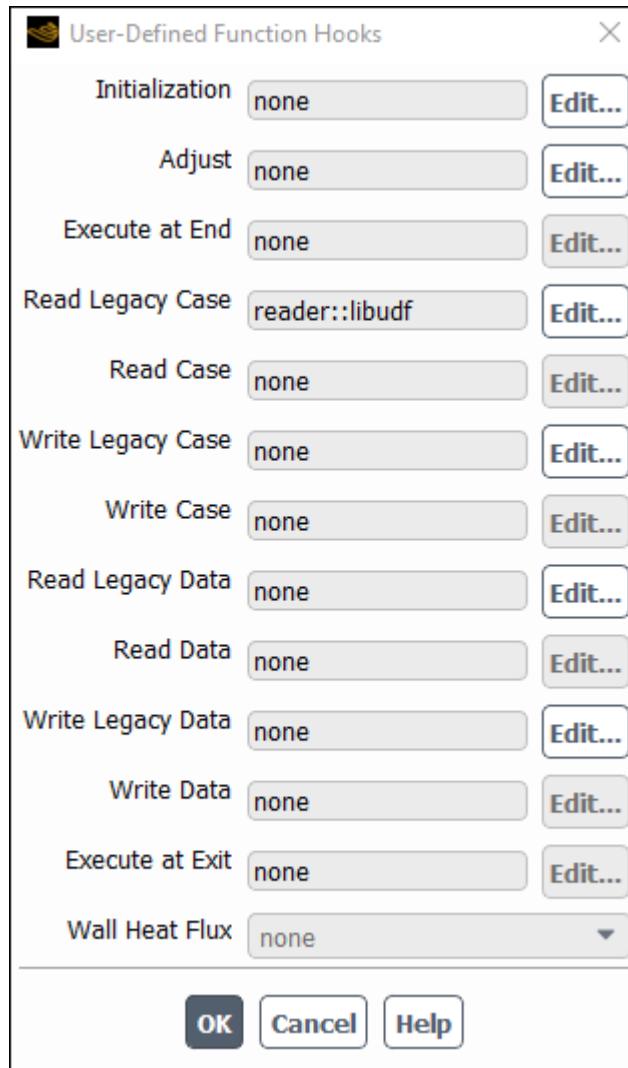
6.1.7. Hooking `DEFINE_RW_FILE` and `DEFINE_RW_HDF_FILE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_RW_FILE` or `DEFINE_RW_HDF_FILE` UDF, it is ready to be hooked to Ansys Fluent. Note that you can hook multiple read/write file UDFs to your model, if desired.

To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box (Figure 6.10: The User-Defined Function Hooks Dialog Box (p. 432)).

⬚ **Parameters & Customization** → **User Defined Functions** ⬚ **Function Hooks...**

Figure 6.10: The User-Defined Function Hooks Dialog Box
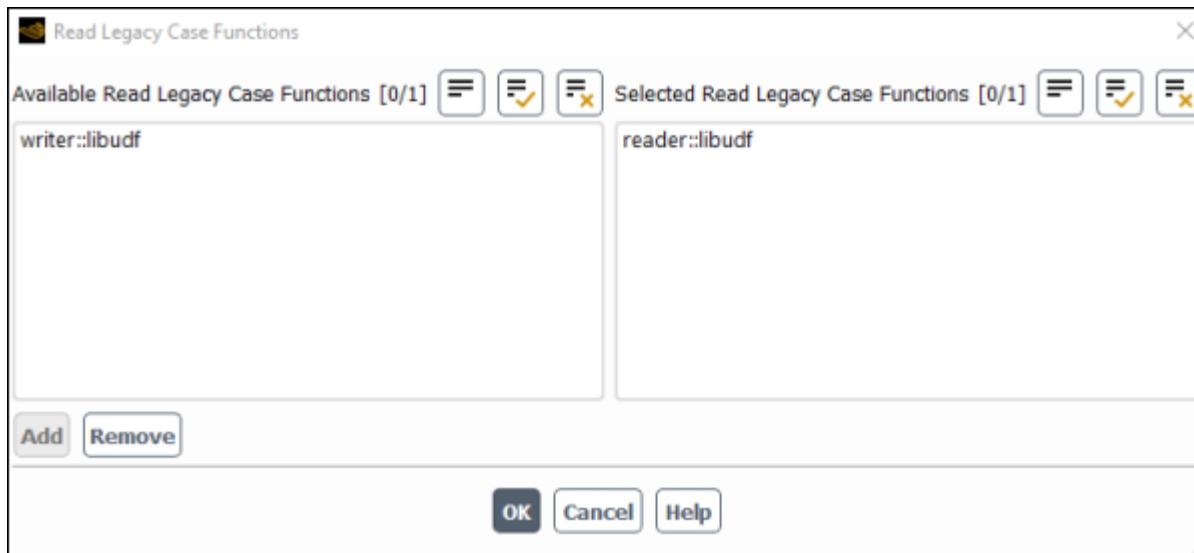
You have the choice of hooking a UDF to read and write a case and data file. Below is a description of what each function does.

- **Read Legacy Case** is called when you read a legacy case file into Ansys Fluent. It will specify the customized section that is to be read from the case file.

- **Read Case** is called when you read a CFF case file into Ansys Fluent. It will specify the customized section that is to be read from the case file.

- **Write Legacy Case** is called when you write a legacy case file from Ansys Fluent. It will specify the customized section that is to be written to the case file.

- **Write Case** is called when you write a CFF case file from Ansys Fluent. It will specify the customized section that is to be written to the case file.

- **Read Legacy Data** is called when you read a legacy data file into Ansys Fluent. It will specify the customized section that is to be read from the data file.

- **Read Data** is called when you read a CFF data file into Ansys Fluent. It will specify the customized section that is to be read from the data file.

- **Write Legacy Data** is called when you write a legacy data file from Ansys Fluent. It will specify the customized section that is to be written to the data file.

- **Write Data** is called when you write a CFF data file from Ansys Fluent. It will specify the customized section that is to be written to the data file.

To hook a read legacy case file UDF, for example, click the **Edit...** button next to **Read Case** to open the **Read Legacy Case Functions** dialog box (Figure 6.11: The Read Case Functions Dialog Box (p. 433)).

Figure 6.11: The Read Case Functions Dialog Box

Select the function(s) you want to hook to your model from the **Available Read Legacy Case Functions** list. Click **Add** and then **OK** to close the dialog box. The name of the function you selected will be displayed in the **Read Case** field of the **User-Defined Function Hooks** dialog box. If you select more than one function, the number will be displayed (for example, **2 selected**). Click **OK** in the **User-Defined Function Hooks** dialog box to apply the settings.

See `DEFINE_RW_FILE` (p. 36) and `DEFINE_RW_HDF_FILE` (p. 38) for details about defining `DEFINE_RW_FILE` and `DEFINE_RW_HDF_FILE` functions.

6.1.8. User-Defined Memory Storage

You can store values computed by your UDF in memory so that they can be retrieved later, either by a UDF or for postprocessing within Ansys Fluent. In order to have access to this memory, you will need to allocate memory by specifying the **Number of User-Defined Memory Locations** and **Number of User-Defined Node Memory Locations** in the **User-Defined Memory** dialog box (Figure 6.12: The User-Defined Memory Dialog Box (p. 434)).

📇 **Parameters & Customization → User Defined Memory** 🖱 **Edit...**

Figure 6.12: The User-Defined Memory Dialog Box

The macros `C_UDMI`, `F_UDMI`, or `N_UDMI` can be used in your UDF to access a particular user-defined memory location in a cell, face, or node. See Cell Macros (p. 308) and Face Macros (p. 322) for details.

Field values that have been stored in user-defined memory or user-defined node memory will be saved to the data file. These fields will also appear in the **User Defined Memory...** category in the drop-down lists in Ansys Fluent's postprocessing dialog boxes. They will be named **User Memory 0**, **User Memory 1**, and so on, and **User Node Memory 0**, **User Node Memory 1**, and so on, based on the memory and node memory location index, respectively. The total number of memory locations is limited to 500. For large numbers of user-defined memory locations, system memory requirements will increase.

> **Important:**
>
> When writing data files in the legacy format (`.dat` files), you must select the UDM quantities using the **Data File Quantities** dialog box in order to postprocess them in CFD-Post. Alternatively, you have the option of exporting the desired quantities to a `.cdat` file. This will ensure that all the UDM quantities are available for postprocessing in CFD-Post. Refer to the Ansys CFD-Post manual for more information.

6.2. Hooking Model-Specific UDFs

This section contains methods for hooking model-specific UDFs to Ansys Fluent that have been defined using `DEFINE` macros found in Model-Specific `DEFINE` Macros (p. 46), and interpreted or compiled using methods described in Interpreting UDFs (p. 393) or Compiling UDFs (p. 399), respectively.

For more information, see the following sections:

6.2.1. Hooking `DEFINE_ANISOTROPIC_CONDUCTIVITY` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_ANISOTROPIC_CONDUCTIVITY` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Create/Edit Materials** dialog box (Figure 6.13: The Create/Edit Materials Dialog Box (p. 437)) in Ansys Fluent.
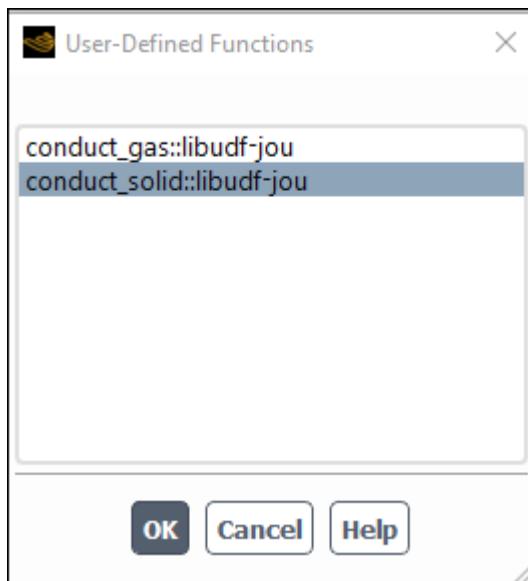
To hook the UDF to Ansys Fluent, first open the **Materials** task page.

 Setup → **Materials**

Make a selection in the **Materials** list and click the **Create/Edit...** button to open the appropriate **Create/Edit Materials** dialog box (Figure 6.13: The Create/Edit Materials Dialog Box (p. 437)).

Figure 6.13: The Create/Edit Materials Dialog Box

To hook an anisotropic conductivity UDF for the conductivity matrix, select **user-defined-anisotropic-k** from the **Thermal Conductivity** drop-down list. The **User-Defined Functions** dialog box ([Figure 6.14: The User-Defined Functions Dialog Box (p. 437)](#)) will open.

Figure 6.14: The User-Defined Functions Dialog Box

Select the name of your UDF (for example, **cyl_ortho_cond::libudf**) and click **OK** in the **User-Defined Functions** dialog box. The name will then be displayed in the field below the **Thermal Conductivity** drop-down list in the **Create/Edit Materials** dialog box. Click **Change/Create** to save your settings.

See `DEFINE_ANISOTROPIC_CONDUCTIVITY` (p. 53) for details about defining `DEFINE_ANISO-TROPIC_CONDUCTIVITY` UDFs and the User's Guide for general information about user-defined anisotropic conductivity.

6.2.2. Hooking `DEFINE_CHEM_STEP` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_CHEM_STEP` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.15: The User-Defined Function Hooks Dialog Box (p. 439)) in Ansys Fluent.

To hook the UDF to Ansys Fluent, first set up the species transport and combustion models in the **Species Model** dialog box.
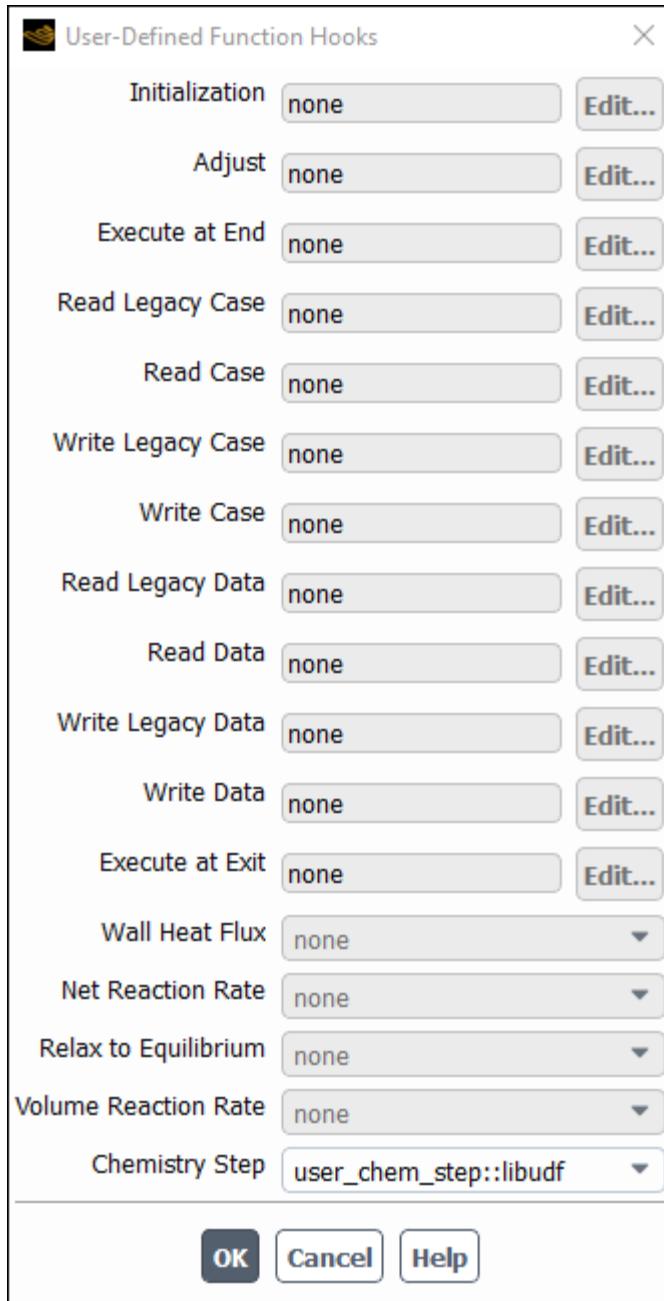
⊡ **Setup** → **Models** → **Species** 🖰 **Edit...**

Note that chemistry step UDFs can only be used with the laminar finite-rate model (with the stiff chemistry solver), the EDC model, or the PDF Transport model. Therefore, you must use one of the following groups of settings in the **Species Model** dialog box:

- To enable the laminar finite-rate model, select **Species Transport**, enable **Volumetric** in the **Reactions** group box, select **Finite-Rate/No TCI** in the **Turbulence-Chemistry Interaction** group box, and select **Stiff Chemistry Solver** from the **Chemistry Solver** drop-down list.

- To enable the EDC model, select **Species Transport**, enable **Volumetric** in the **Reactions** group box, and select **Eddy-Dissipation Concept** in the **Turbulence-Chemistry Interaction** group box.

- To enable the PDF Transport model, select **Composition PDF Transport** and enable **Volumetric** in the **Reactions** group box.

Next, open the **User-Defined Function Hooks** dialog box (Figure 6.15: The User-Defined Function Hooks Dialog Box (p. 439)).

⊡ **Parameters & Customization** → **User Defined Functions** 🖰 **Function Hooks...**

Figure 6.15: The User-Defined Function Hooks Dialog Box

Select the function name (for example, **user_chem_step::libudf**) in the **Chemistry Step** drop-down list in the **User-Defined Function Hooks** dialog box, and click **OK**.
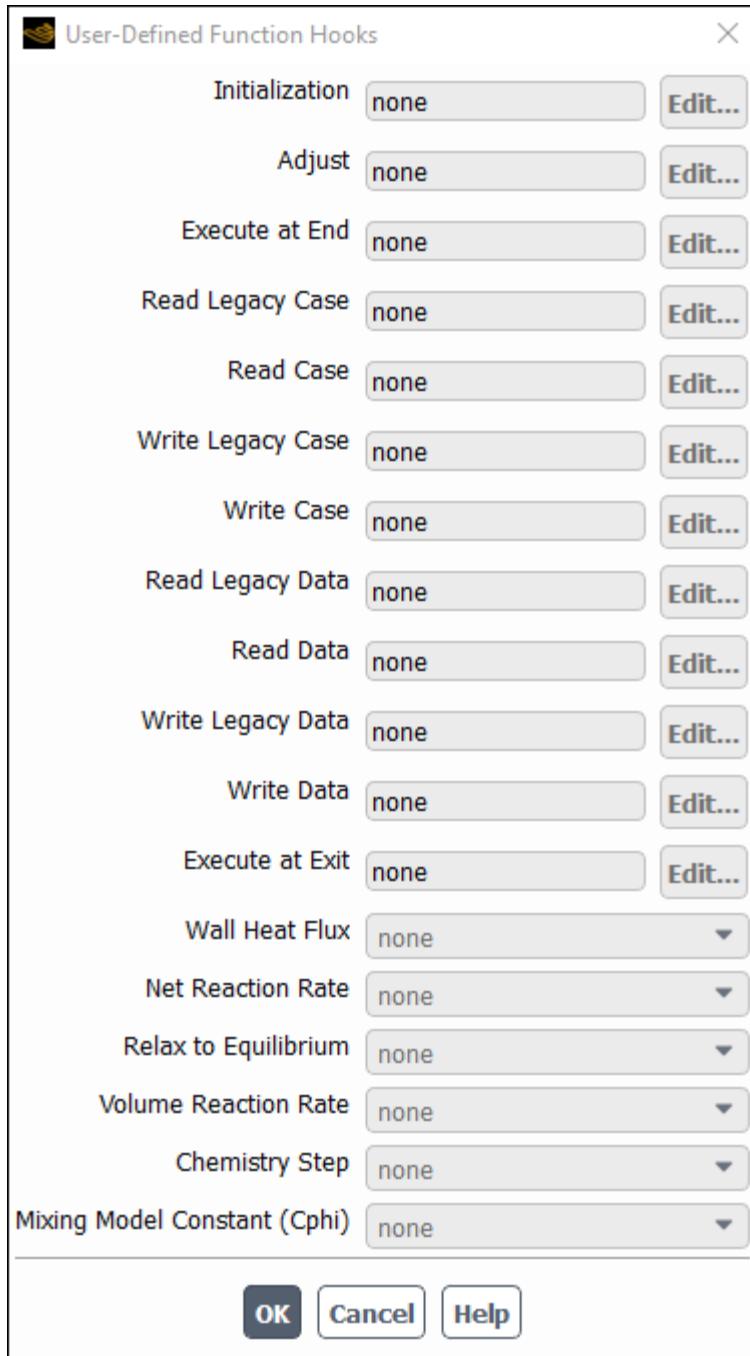
See `DEFINE_CHEM_STEP` (p. 55) for details about defining `DEFINE_CHEM_STEP` functions.

6.2.3. Hooking `DEFINE_CPHI` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_CPHI` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.16: The User-Defined Function Hooks Dialog Box (p. 441)) in Ansys Fluent.

To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box ().

⊞ Parameters & Customization → **User Defined Functions** 🖱 **Function Hooks...**

Figure 6.16: The User-Defined Function Hooks Dialog Box

Important:

EDC or PDF Transport models must be enabled to hook the mixing model constant Cphi UDFs.

Select the function name (for example, **user_cphi::libudf**) from the drop-down list for **Mixing Model Constant (Cphi)**, and click **OK**.

See `DEFINE_CPHI` (p. 57) for details about defining `DEFINE_CPHI` functions.

6.2.4. Hooking `DEFINE_DIFFUSIVITY` UDFs
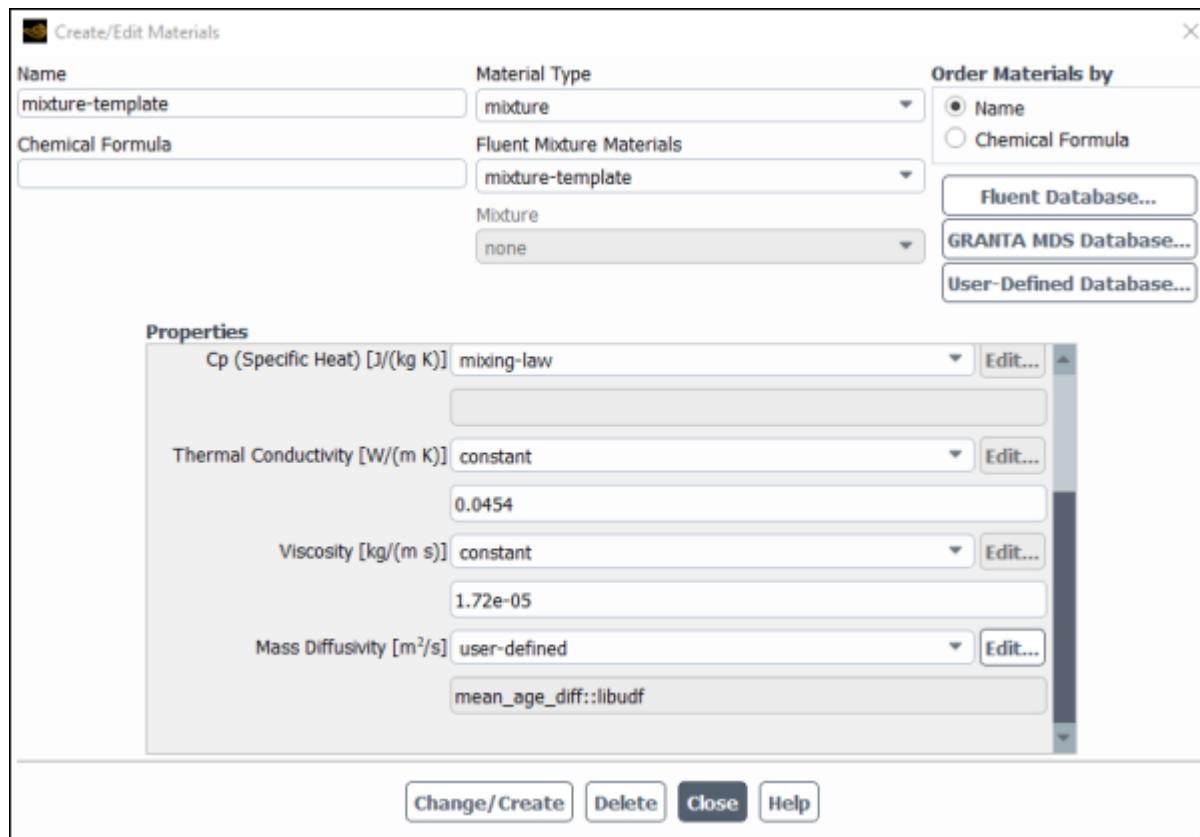
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DIFFUSIVITY` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in Ansys Fluent.

To hook the UDF to Ansys Fluent, first open the **Materials** task page.

Setup → Materials
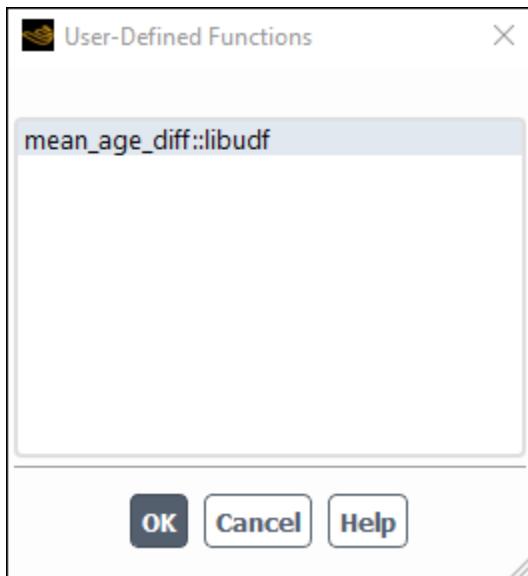
Make a selection in the **Materials** list and click the **Create/Edit...** button to open the appropriate **Create/Edit Materials** dialog box (Figure 6.17: The Create/Edit Materials Dialog Box (p. 442)).

Figure 6.17: The Create/Edit Materials Dialog Box

You then have the following options:

- To hook a mass diffusivity UDF for the species transport equations, select **user-defined** from the **Mass Diffusivity** drop-down list of the **Create/Edit Materials** dialog box (Figure 6.17: The Create/Edit Materials Dialog Box (p. 442)). The **User-Defined Functions** dialog box (Figure 6.18: The User-Defined Functions Dialog Box (p. 443)) will open.
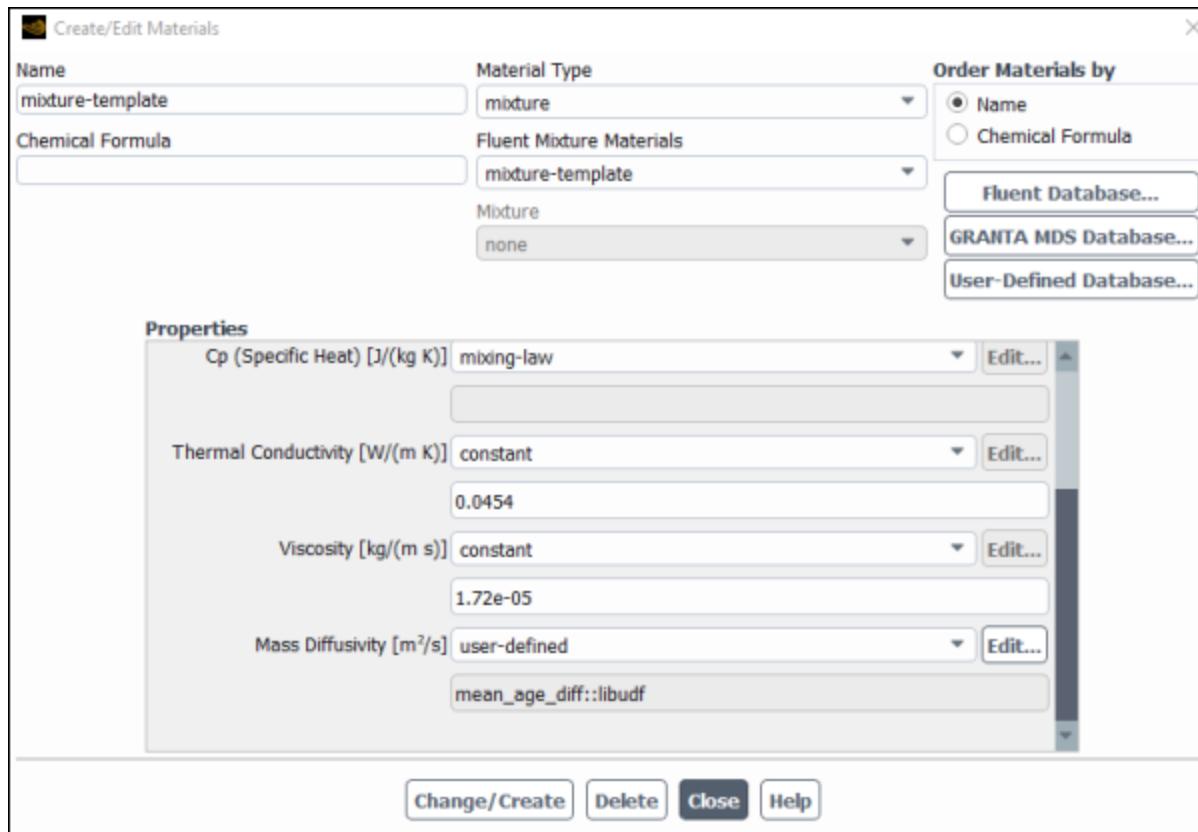
Figure 6.18: The User-Defined Functions Dialog Box

Select the name of your UDF (for example, **mean_age_diff::libudf**) and click **OK** in the **User-Defined Functions** dialog box. The name will then be displayed in the field below the **Mass Diffusivity** drop-down list in the **Create/Edit Materials** dialog box. Click **Change/Create** to save your settings.

- To hook a single diffusion coefficient UDF so that it applies to all UDS equations, first set the number and options of the user-defined scalars using the **User-Defined Scalars** dialog box.

    Parameters & Customization → **User Defined Scalars** **New...**

Then, select **user-defined** from the **UDS Diffusivity** drop-down list in the **Create/Edit Materials** dialog box (Figure 6.19: The Create/Edit Materials Dialog Box (p. 444)).

Figure 6.19: The Create/Edit Materials Dialog Box

Next, select the name of your UDF (for example, **mean_age_diff::libudf**) in the **User-Defined Functions** dialog box that opens (Figure 6.18: The User-Defined Functions Dialog Box (p. 443)) and click **OK**. The name will then be displayed in the field below the **UDS Diffusivity** drop-down list in the **Create/Edit Materials** dialog box. Click **Change/Create** to save your settings.

See `DEFINE_DIFFUSIVITY` (p. 60) for details about defining `DEFINE_DIFFUSIVITY` UDFs and the User's Guide for general information about UDS diffusivity.

6.2.5. Hooking `DEFINE_DOM_DIFFUSE_REFLECTIVITY` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DOM_DIFFUSE_REFLECTIVITY` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.20: The User-Defined Function Hooks Dialog Box (p. 445)) in Ansys Fluent.
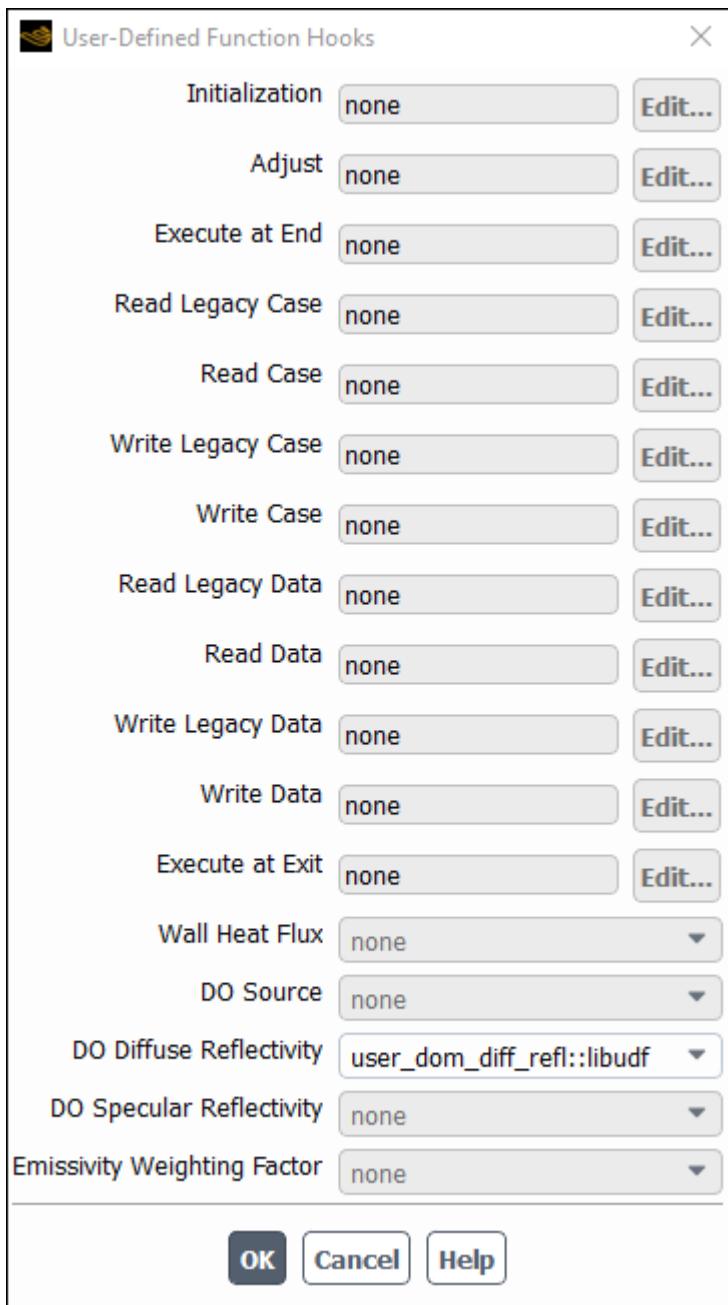
Important:

The discrete ordinates (DO) radiation model must be enabled from the **Radiation Model** dialog box.

To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box (Figure 6.20: The User-Defined Function Hooks Dialog Box (p. 445)).

⊞ Parameters & Customization → User Defined Functions 🖱 Function Hooks...

Figure 6.20: The User-Defined Function Hooks Dialog Box

Select the function name (for example, **user_dom_diff_refl::libudf**) in the **DO Diffuse Reflectivity** drop-down list in the **User-Defined Function Hooks** dialog box, and click **OK**.

See DEFINE_DOM_DIFFUSE_REFLECTIVITY (p. 61) for details about DEFINE_DOM_DIFFUSE_REFLECTIVITY functions.

6.2.6. Hooking `DEFINE_DOM_SOURCE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DOM_SOURCE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.21: The User-Defined Function Hooks Dialog Box (p. 447)) in Ansys Fluent.


Important:

The discrete ordinates (DO) radiation model must be enabled from the **Radiation Model** dialog box.

To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box (Figure 6.21: The User-Defined Function Hooks Dialog Box (p. 447)).

 Parameters & Customization → **User Defined Functions** **Function Hooks...**

Figure 6.21: The User-Defined Function Hooks Dialog Box

Select the function name (for example, **dom::libudf**) in the **DO Source** drop-down list in the **User-Defined Function Hooks** dialog box, and click **OK**.

See `DEFINE_DOM_SOURCE` (p. 63) for details about `DEFINE_DOM_SOURCE` functions.

6.2.7. Hooking `DEFINE_DOM_SPECULAR_REFLECTIVITY` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DOM_SPECULAR_REFLECTIVITY` UDF, the name of the function you supplied as a `DEFINE`
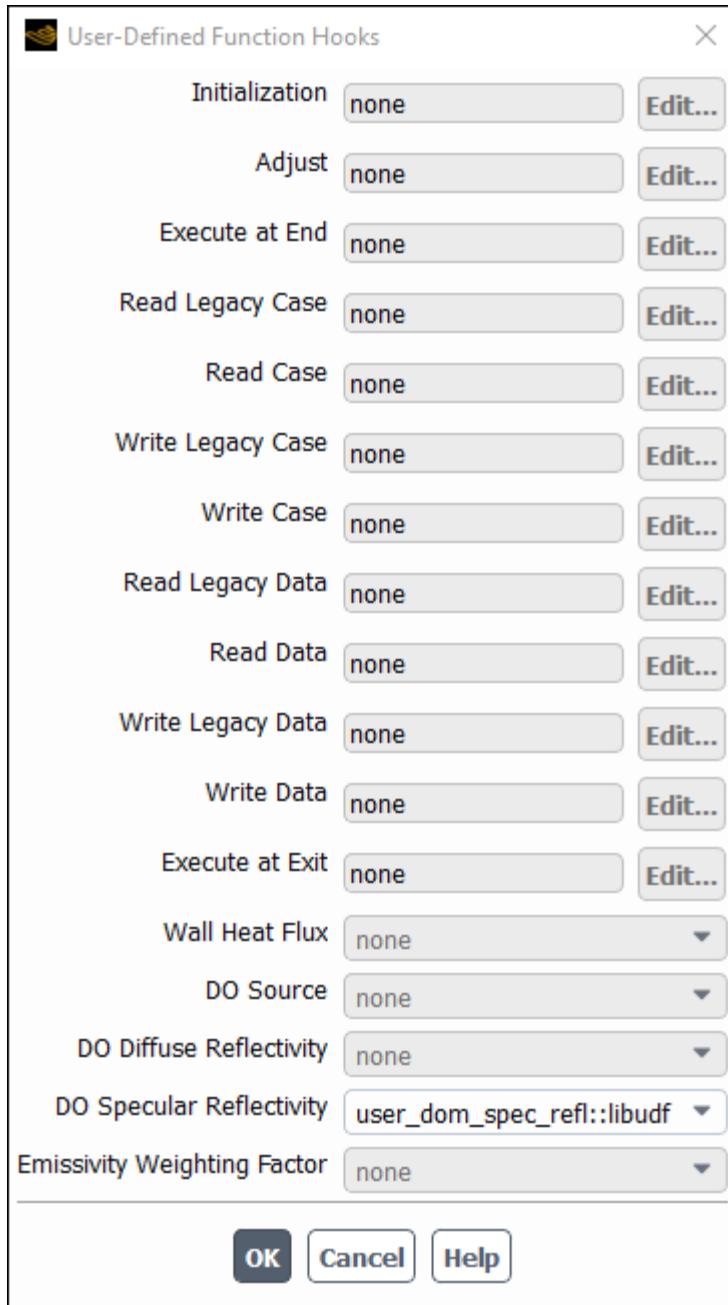
macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.22: The User-Defined Function Hooks Dialog Box (p. 449)) in Ansys Fluent.


Important:

The discrete ordinates (DO) radiation model must be enabled from the **Radiation Model** dialog box.

To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box (Figure 6.22: The User-Defined Function Hooks Dialog Box (p. 449)).

 Parameters & Customization → **User Defined Functions** **Function Hooks...**

Figure 6.22: The User-Defined Function Hooks Dialog Box

Select the function name (for example, **user_dom_spec_refl::libudf**) in the **DO Specular Reflectivity** drop-down list in the **User-Defined Function Hooks** dialog box, and click **OK**.

See `DEFINE_DOM_SPECULAR_REFLECTIVITY` (p. 64) for details about `DEFINE_DOM_SPECU-LAR_REFLECTIVITY` functions.

6.2.8. Hooking `DEFINE_EC_RATE` UDFs

After you have compiled (Compiling UDFs (p. 399)) your `DEFINE_EC_RATE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-**

Defined Function Hooks dialog box ([Figure 6.23: The User-Defined Function Hooks Dialog Box](p. 451)) in Ansys Fluent.

To hook the UDF to Ansys Fluent:
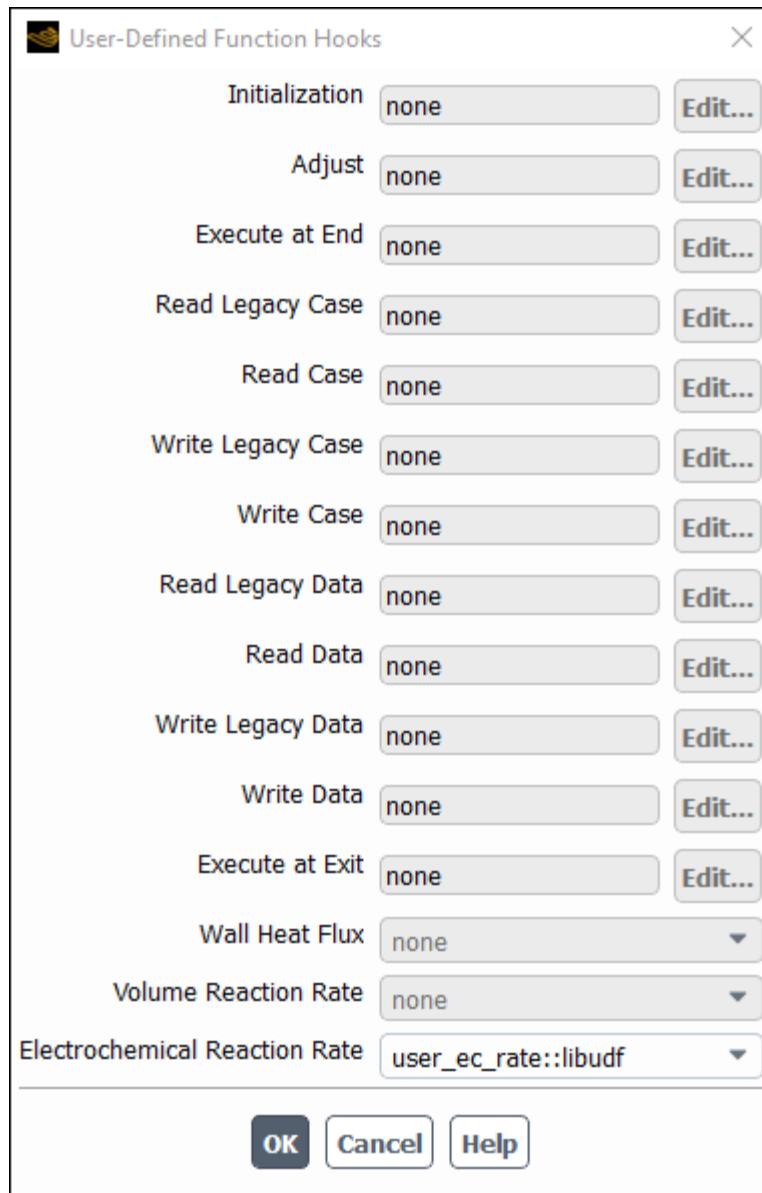
1. Set up an appropriate reaction model in the **Species Model** dialog box.

    🖳 **Setup** → **Models** → **Species** 🖱 **Edit...**

    a. From the **Model** list, select **Species Transport**.

    b. In the **Reactions** group box, enable **Volumetric** and **Electrochemical** and click **OK**.

2. Hook the UDF to Ansys Fluent using the **User-Defined Function Hooks** dialog box ([Figure 6.67: The User-Defined Function Hooks Dialog Box](p. 506)).

    🖳 **Parameters & Customization** → **User Defined Functions** 🖱 **Function Hooks...**

Figure 6.23: The User-Defined Function Hooks Dialog Box

a. From the **Electro-Chemical Reaction Rate** drop-down list, select the function name (for example, `user_ec_rate::libudf`).

b. Click **OK**.

See `DEFINE_EC_RATE` (p. 67) for details about `DEFINE_EC_RATE` functions.
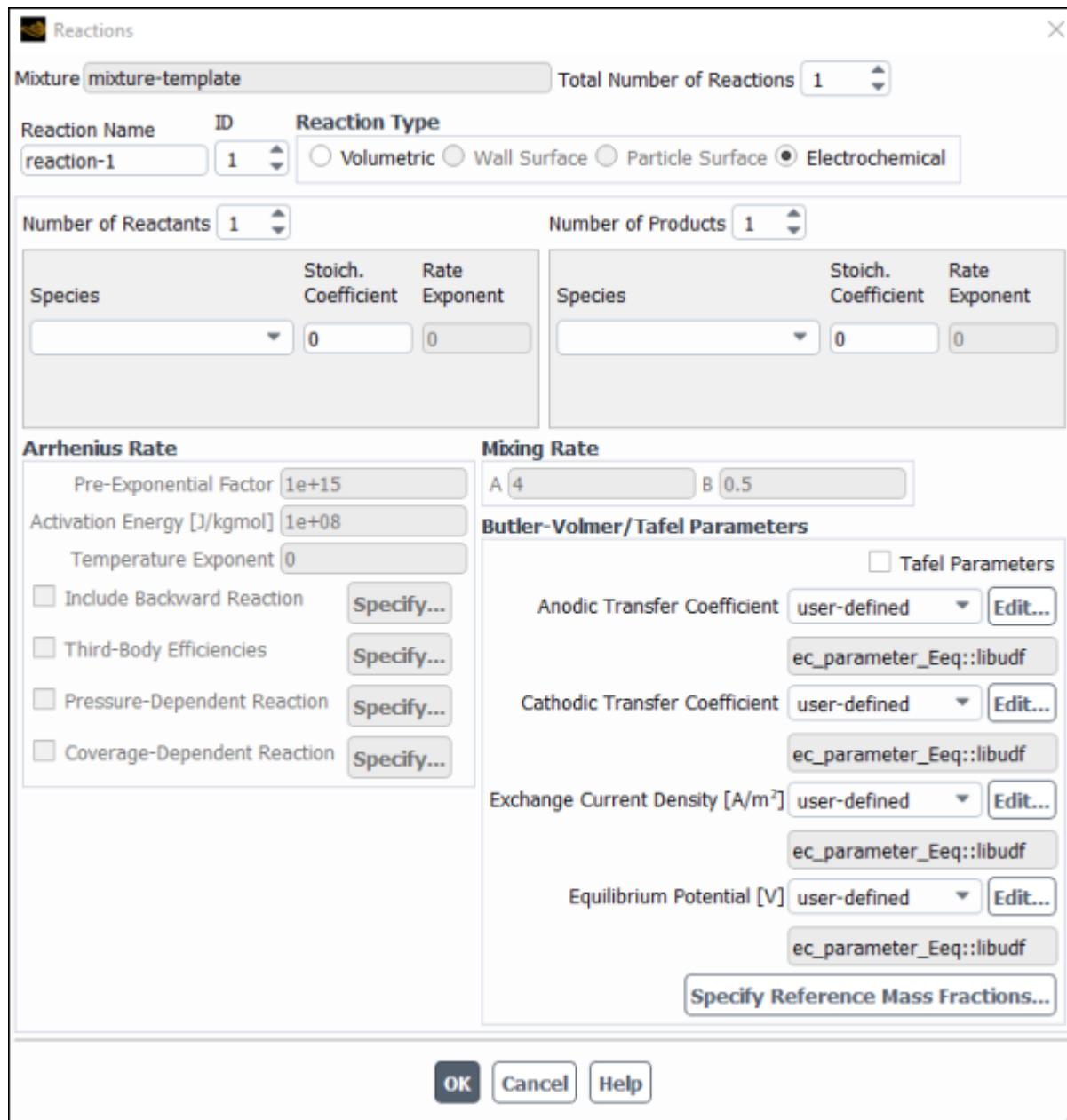
6.2.9. Hooking `DEFINE_EC_KINETICS_PARAMETER` UDFs

After you have compiled (Compiling UDFs (p. 399)) your `DEFINE_EC_KINETICS_PARAMETER` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Functions** dialog box when you choose **user-defined** for a kinetics parameter.

To hook the UDF to Ansys Fluent:

1. In the **Reaction** dialog box, select the **Electrochemical** reaction type.

Figure 6.24: The Reactions Dialog Box

2. Define four kinetics parameters for the Butler-Volmer rate equation. For each kinetics parameter that you want to define via your macro, select **user-defined** from the drop-down list. Next, in the **User-Defined Functions** dialog box that opens, select the name of your UDF (for example, **ec_parameter_Eeq::libudf**) and click **OK**. The name of the function will be displayed in the field below the selected Butler-Volmer/Tafel parameter.

3. Click **OK** to close the **Reaction** dialog box.

See `DEFINE_EC_KINETICS_PARAMETER` (p. 66) for details about `DEFINE_EC_KINETICS_PARA-METER` functions.

6.2.10. Hooking `DEFINE_EDC_MDOT` UDFs

Once the UDF that you have defined using `DEFINE_EDC_MDOT` is compiled (Compiling UDFs (p. 399)), the name of the function that you supplied as a `DEFINE` macro argument will become visible and selectable from the **User Defined EDC Scales** drop-down list in the **Options** group box of the **Species Model** dialog box in Ansys Fluent (Figure 6.26: The Species Model Dialog Box (User Defined EDC Scales) (p. 456)).

> **Note:**
>
> • Note that the UDF hook is only available when the **Eddy Dissipation Concept** model is selected in the **Species Model** dialog box.
>
> • `DEFINE_EDC_MDOT` and `DEFINE_EDC_SCALES` cannot be used together.

To hook the UDF to Ansys Fluent:

1. In the **Species Model** dialog box, ensure that **Eddy Dissipation Concept** is enabled in the **Turbulence-Chemistry Interaction** group box.

    Setting Up Physics → Models → Species...

Figure 6.25: The Species Model Dialog Box (User Defined EDC Scales)

2. From the **User Defined EDC Scales** drop-down list (**Options** group box), select the function name (for example, `edc_mdot::libudf`) and click **Apply**.

See `DEFINE_EDC_MDOT` (p. 69) for details about `DEFINE_EDC_MDOT` functions.

6.2.11. Hooking `DEFINE_EDC_SCALES` UDFs

Once the UDF that you have defined using `DEFINE_EDC_SCALES` is compiled (Compiling UD-Fs (p. 399)), the name of the function that you supplied as a `DEFINE` macro argument will become

visible and selectable from the **User Defined EDC Scales** drop-down list in the **Options** group box of the **Species Model** dialog box in Ansys Fluent (Figure 6.26: The Species Model Dialog Box (User Defined EDC Scales) (p. 456)).

Note:

- Note that the UDF hook is only available when the **Eddy Dissipation Concept** model is selected in the **Species Model** dialog box.

- `DEFINE_EDC_MDOT` and `DEFINE_EDC_SCALES` cannot be used together.

To hook the UDF to Ansys Fluent:

1. In the **Species Model** dialog box, ensure that **Eddy Dissipation Concept** is enabled in the **Turbulence-Chemistry Interaction** group box.

    Setting Up Physics → Models → Species...

Figure 6.26: The Species Model Dialog Box (User Defined EDC Scales)

2. From the **User Defined EDC Scales** drop-down list (**Options** group box), select the function name (for example, `edc_scales::libudf`) and click **Apply**.

See DEFINE_EDC_SCALES (p. 71) for details about `DEFINE_EDC_SCALES` functions.

6.2.12. Hooking `DEFINE_EMISSIVITY_WEIGHTING_FACTOR` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_EMISSIVITY_WEIGHTING_FACTOR` UDF, the name of the function you supplied as a
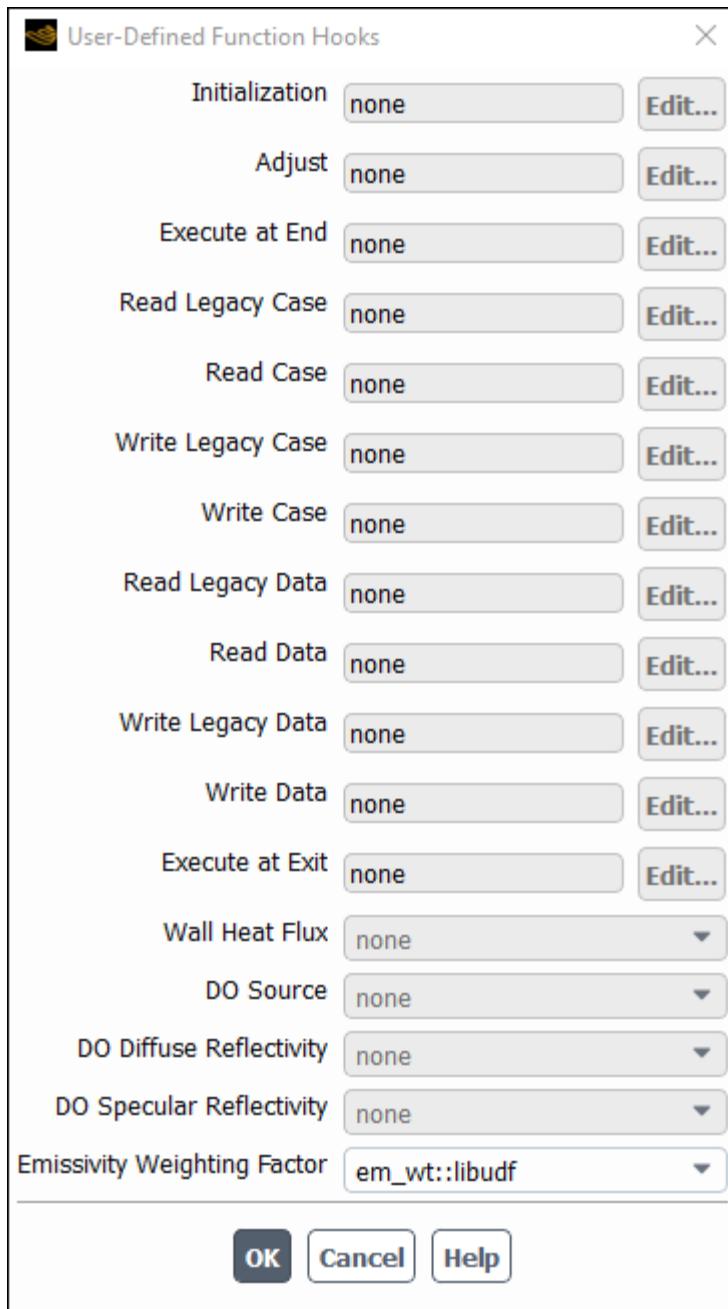
DEFINE macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box () in Ansys Fluent.

Important:

In the **Radiation Model** dialog box, **P1** or **Discrete Ordinates (DO)** must be selected from the **Model** list, and a nonzero value must be entered for **Number of Bands** in the **Non-Gray Model** group box.

To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box ().

⬚ **Parameters & Customization** → **User Defined Functions** 🖱 **Function Hooks...**

Figure 6.27: The User-Defined Function Hooks Dialog Box

Select the function name (for example, **em_wt::libudf**) in the **Emissivity Weighting Factor** drop-down list in the **User-Defined Function Hooks** dialog box, and click **OK**.

See `DEFINE_EMISSIVITY_WEIGHTING_FACTOR` (p. 72) for details about defining `DEFINE_EMISSIVITY_WEIGHTING_FACTOR` UDFs.

6.2.13. Hooking `DEFINE_FLAMELET_PARAMETERS` UDFs

Once the UDF that you have defined using `DEFINE_FLAMELET_PARAMETERS` is compiled (Compiling UDFs (p. 399)), the name of the function that you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function** drop-down list under the **Flamelet** tab of the
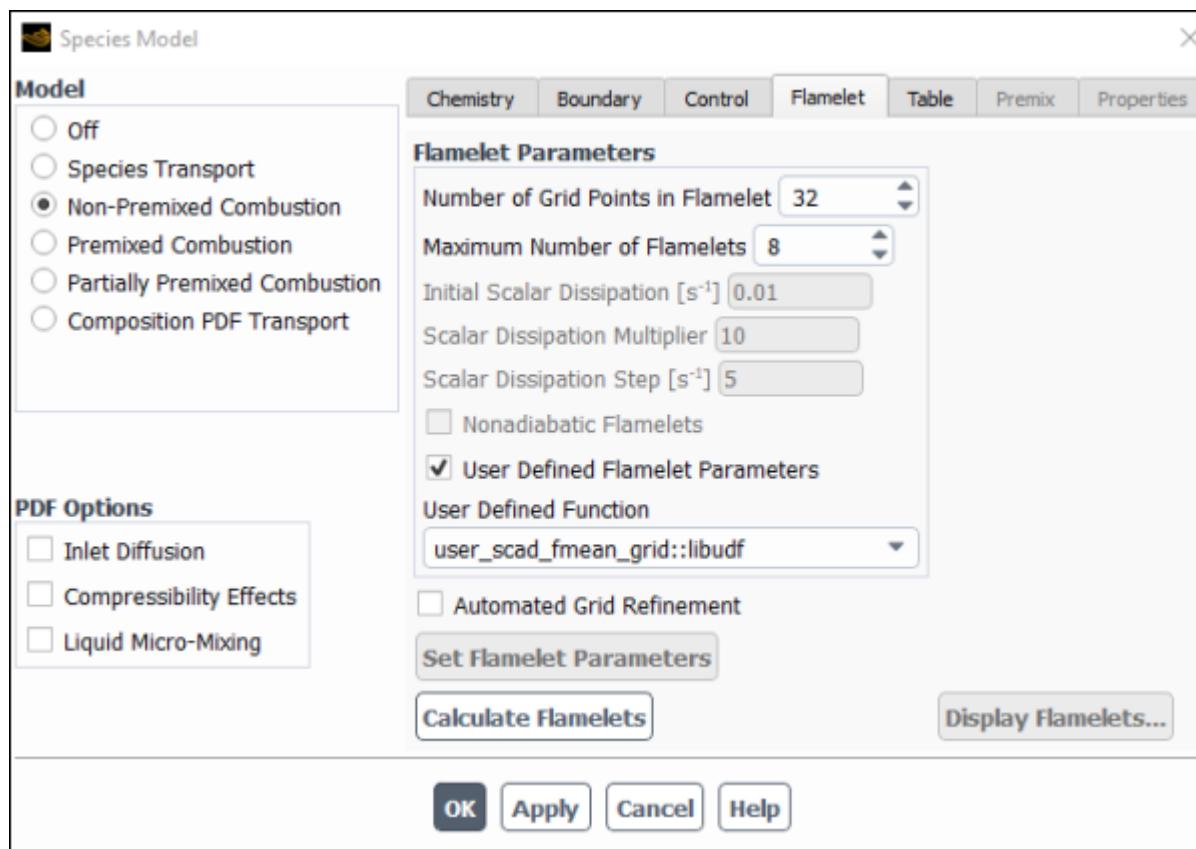
Species Model dialog box in Ansys Fluent (Figure 6.28: The Flamelet Tab Dialog Box (User-Defined Flamelet Parameters) (p. 459)).

Note:

Note that the UDF hook is only available when either the **Non-Premixed Combustion** or **Partially-Premixed Combustion** model is selected in the **Species Model** dialog box.

Setup → Models → Species 🖰 Edit...

Figure 6.28: The Flamelet Tab Dialog Box (User-Defined Flamelet Parameters)

To hook the UDF to Ansys Fluent:

1. In the **Species Model** dialog box, under the **Flamelet** tab, select **User-Defined Flamelet Parameters**.

2. From the **User Defined Function** drop-down list, select the function name (for example, `user_scad_fmean_grid ::libudf`) and click **Apply**.

See `DEFINE_FLAMELET_PARAMETERS` (p. 73) for details about defining `DEFINE_FLAMELET_PARAMETERS` UDFs.

6.2.14. Hooking `DEFINE_ZONE_MOTION` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_ZONE_MOTION` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Fluid** or **Solid** dialog boxes in Ansys Fluent, under the **Reference Frame** tab, the **Mesh Motion** tab, and the **Solid Motion** tab if the **Frame Motion**, **Mesh Motion**, and **Solid Motion** options are enabled, respectively.

⊞ **Setup** → ✛ **Cell Zone Conditions**

Select the fluid or solid zone and click the **Edit…** button to open the **Fluid** or **Solid** dialog box.

Figure 6.29: The Fluid Dialog Box for Frame Motion
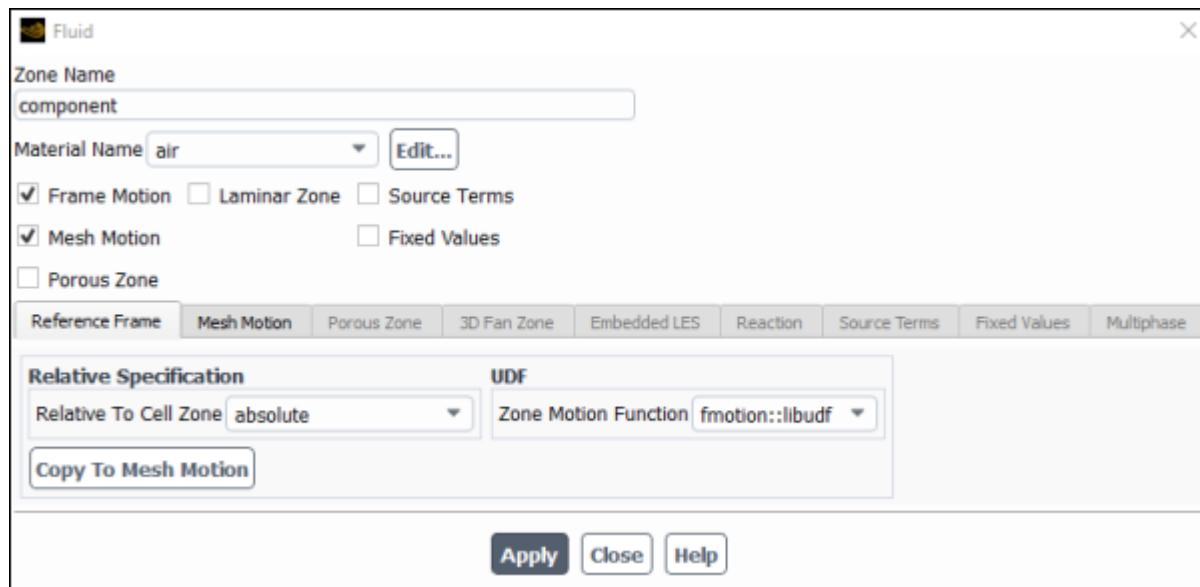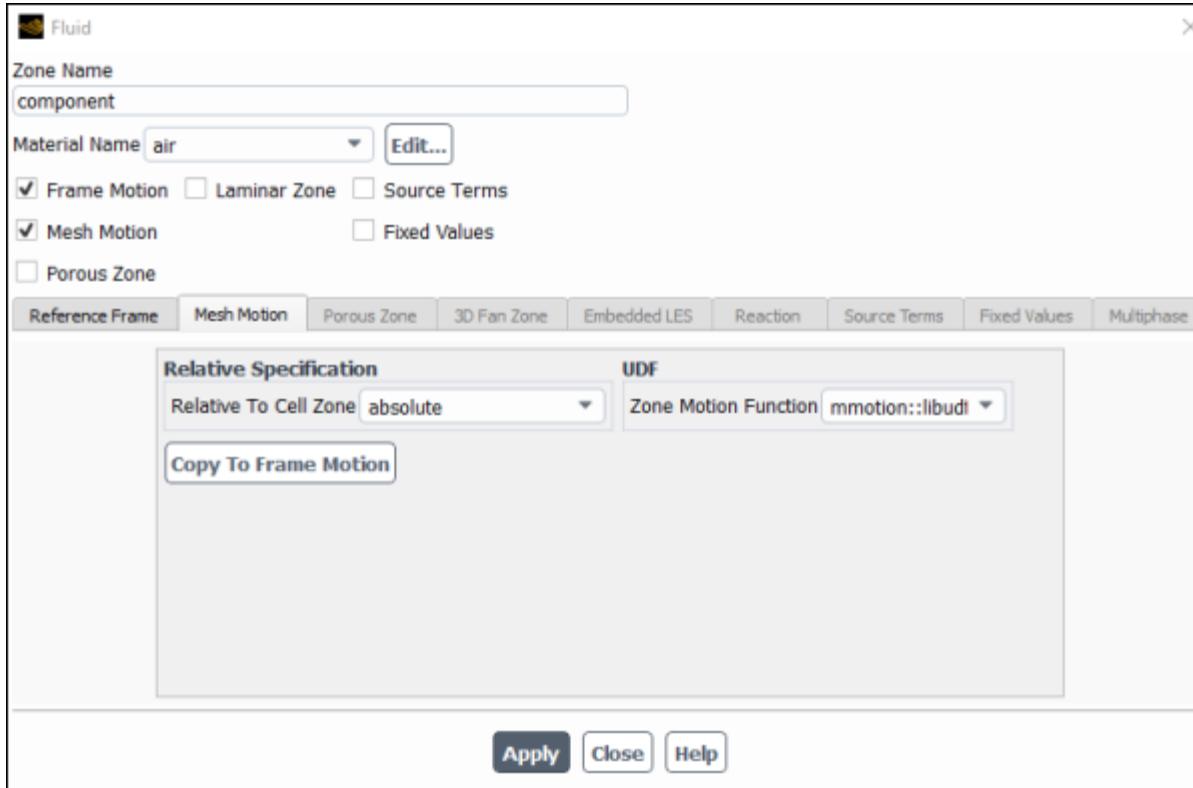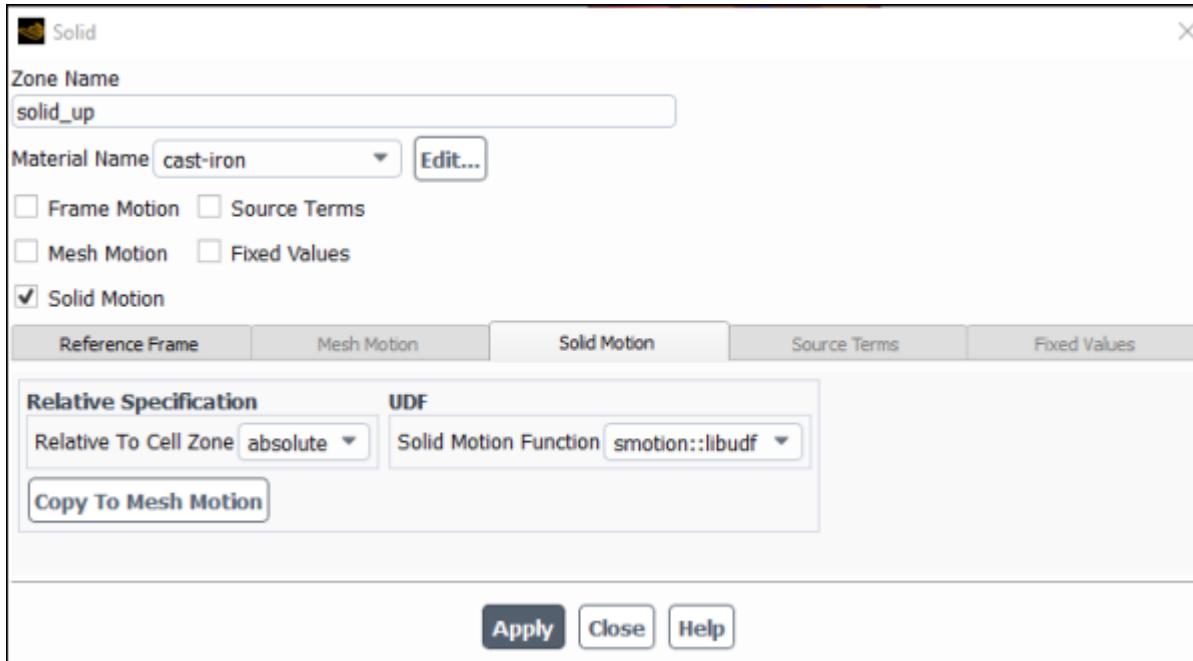
Figure 6.30: The Fluid Dialog Box for Mesh Motion

Figure 6.31: The Solid Dialog Box for Solid Motion

Next, select the UDF from the **Zone Motion Function** drop-down list in the **Fluid** dialog box or the **Zone Motion Function** or **Solid Motion Function** drop-down list in the **Solid** dialog box.

See `DEFINE_ZONE_MOTION` (p. 75) for details about `DEFINE_ZONE_MOTION` functions.

6.2.15. Hooking `DEFINE_GRAY_BAND_ABS_COEFF` UDFs
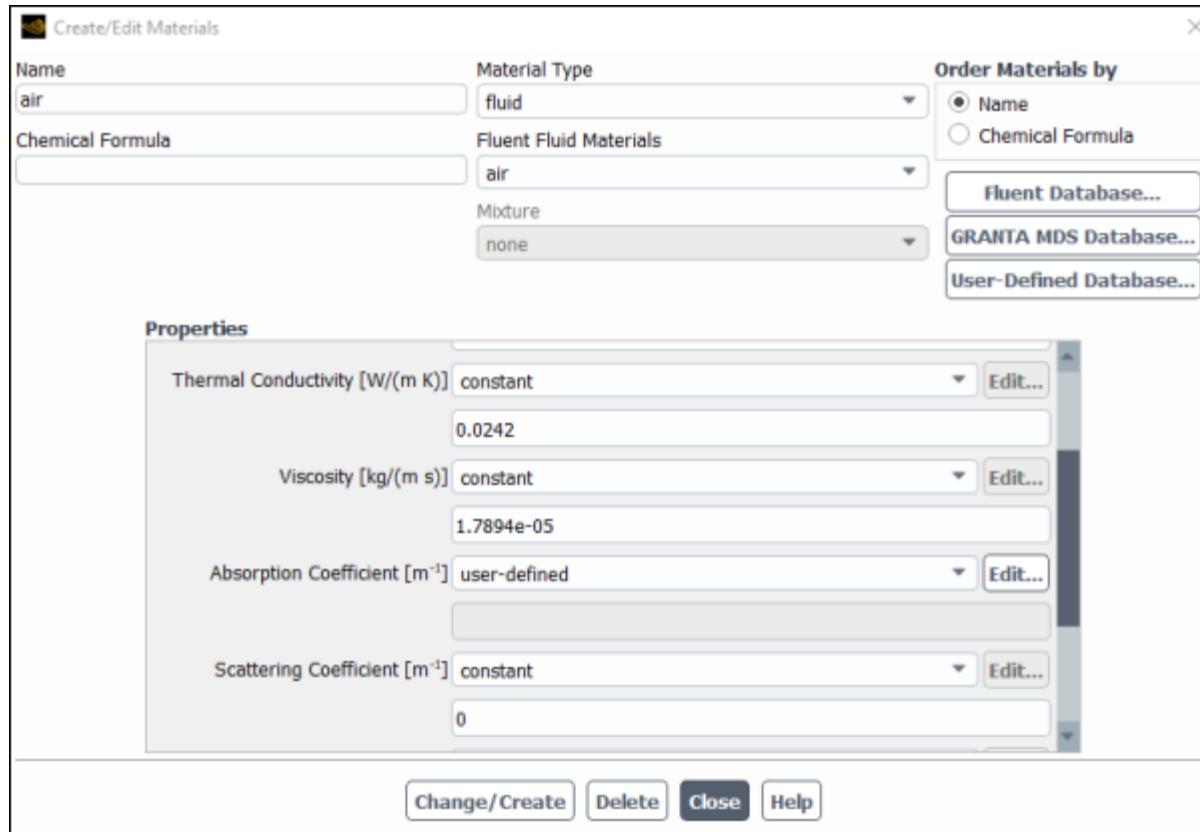
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_GRAY_BAND_ABS_COEFF` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Create/Edit Materials** dialog box in Ansys Fluent.

To hook the UDF to Ansys Fluent, first make sure that the **Discrete Ordinates (DO)** model is selected in the **Radiation Model** dialog box, with a nonzero **Number of Bands** in the **Non-Gray Model** group box. Then open the **Materials** task page.

 Setup → **Materials**

Select the appropriate material from the **Material** selection list and click the **Create/Edit...** button to open the **Create/Edit Materials** dialog box (Figure 6.32: The Create/Edit Materials Dialog Box (p. 462)).

Figure 6.32: The Create/Edit Materials Dialog Box

Next, select **user-defined-gray-band** from the **Absorption Coefficient** drop-down list in the **Create/Edit Materials** dialog box. This opens the **User-Defined Functions** dialog box, where you must select the name of the function (for example, **user_gray_band_abs::libudf**) and click **OK**. Finally, click **Change/Create** in the **Create/Edit Materials** dialog box.

See `DEFINE_GRAY_BAND_ABS_COEFF` (p. 77) for details about `DEFINE_GRAY_BAND_ABS_COEFF` functions.
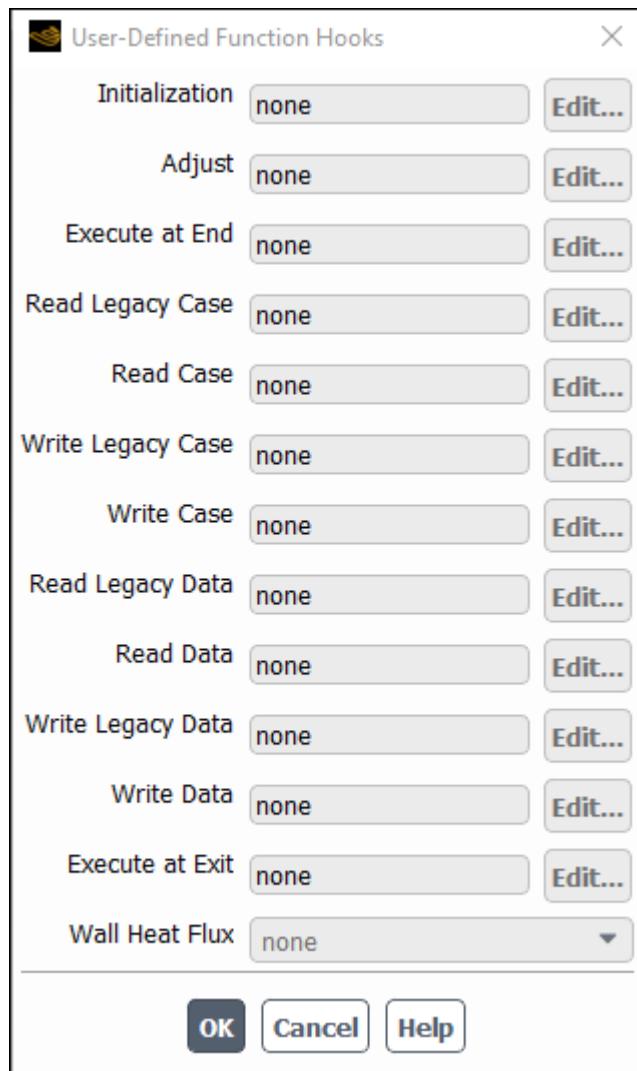
6.2.16. Hooking `DEFINE_HEAT_FLUX` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_HEAT_FLUX` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.33: The User-Defined Function Hooks Dialog Box (p. 463)) in Ansys Fluent.

To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box (Figure 6.33: The User-Defined Function Hooks Dialog Box (p. 463)).

 Parameters & Customization → **User Defined Functions** **Function Hooks...**

Figure 6.33: The User-Defined Function Hooks Dialog Box

Important:

The **Energy Equation** must be enabled.

Select the function name (for example, **user_heat_flux::libudf**) in the **Wall Heat Flux** drop-down list in the **User-Defined Function Hooks** dialog box, and click **OK**.

See `DEFINE_HEAT_FLUX` (p. 78) for details about `DEFINE_HEAT_FLUX` functions.

6.2.17. Hooking `DEFINE_IGNITE_SOURCE` UDFs

After you have compiled (Compiling UDFs (p. 399)) your `DEFINE_IGNITE_SOURCE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.34: The User-Defined Function Hooks Dialog Box (p. 465)) in Ansys Fluent.

To hook the UDF to Ansys Fluent, first open the **General** task page.

⌸ **Setup** → ✛ **General**

Select **Pressure-Based** from the **Type** list, and select **Transient** from the **Time** list.

Then, select a turbulence model in the **Viscous Model** dialog box.

⌸ **Setup** → **Models** → **Viscous Model** 🖱 **Edit...**

Next, set up an appropriate reaction model in the **Species Model** dialog box.

⌸ **Setup** → **Models** → **Species** 🖱 **Edit...**

Select either the **Premixed Combustion** or the **Partially Premixed Combustion** model in the **Species Model** dialog box and click **OK**.
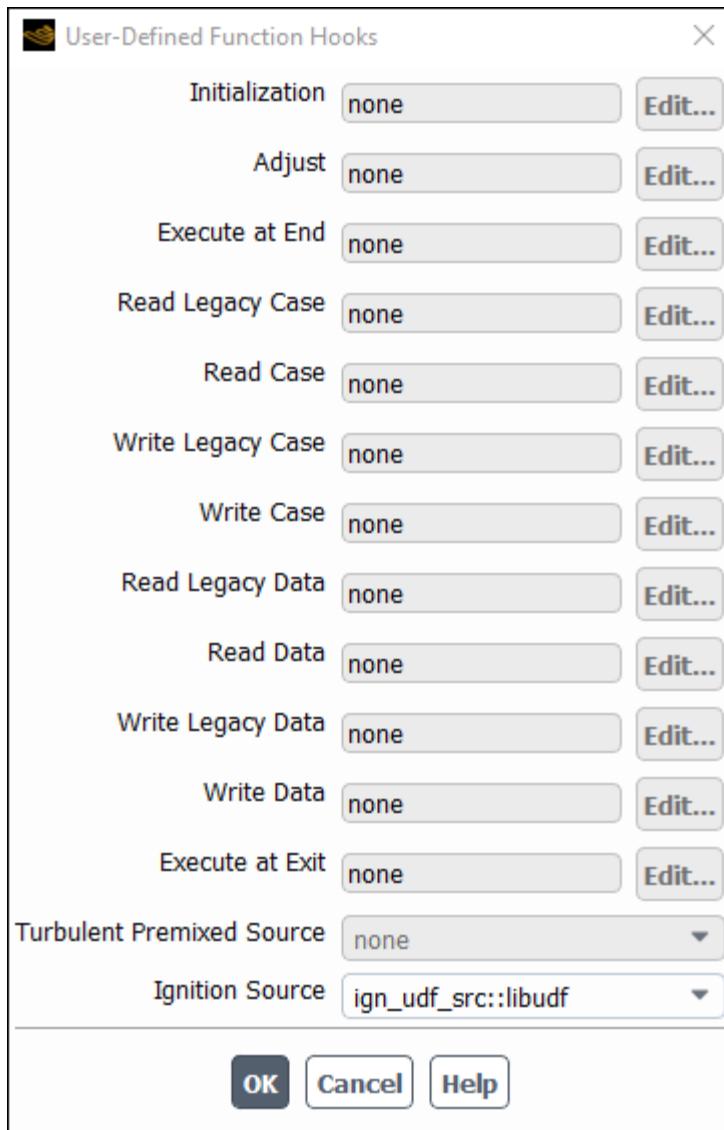
Then open the **Autoignition Model** dialog box.

⌸ **Setup** → **Models** → **Species** → **Autoignition** 🖱 **Edit...**

Select the **Knock Model** from the **Model** list in the **Autoignition Model** dialog box, and click **OK**.

Next, open the **User-Defined Function Hooks** dialog box (Figure 6.34: The User-Defined Function Hooks Dialog Box (p. 465)).

⌸ **Parameters & Customization** → **User Defined Functions** 🖱 **Function Hooks...**

Figure 6.34: The User-Defined Function Hooks Dialog Box

Select the function name (for example, **ign_udf_src::libudf**) in the **Ignition Source** drop-down list in the **User-Defined Function Hooks** dialog box, and click **OK**.

See `DEFINE_IGNITE_SOURCE` (p. 80) for details about `DEFINE_IGNITE_SOURCE` functions.

6.2.18. Hooking `DEFINE_MASS_TR_PROPERTY` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_MASS_TR_PROPERTY` UDF, the name of the function you supplied in the argument of the `DEFINE` macro will become visible and selectable in the **User-Defined Functions** dialog box in Ansys Fluent.
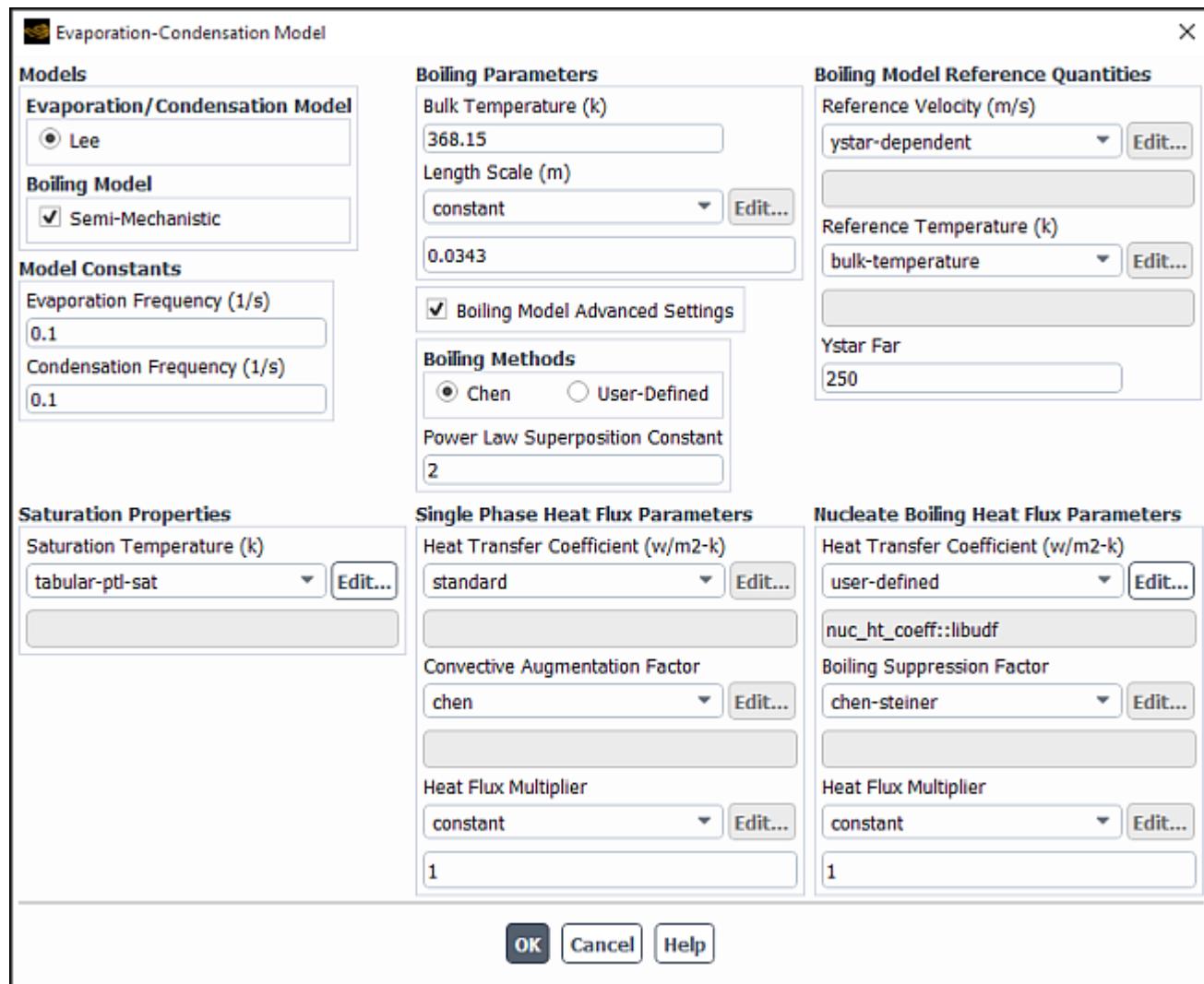
To hook the UDF:

1. In the **Multiphase Model** dialog box, go to the **Phase Interaction** > **Heat, Mass, Reactions** > **Mass** tab and click **Edit...** next to **evaporation-condensation**.
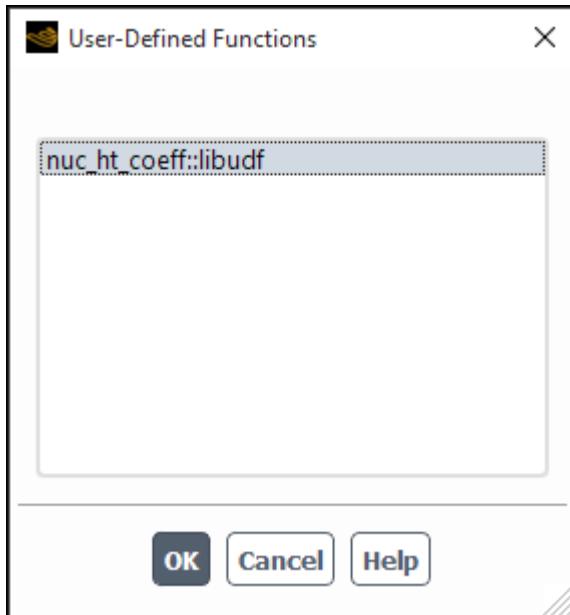
Setup → **Models** → **Multiphase** 🖱 **Edit...**

2. In the **Evaporation-Condensation Model** dialog box that opens, make sure that the **Semi-Mechanistic** boiling model is selected (Figure 6.35: The Evaporation-Condensation Model Dialog Box (p. 466)).

Figure 6.35: The Evaporation-Condensation Model Dialog Box

3. Select **user-defined** in the drop-down list for the appropriate parameter (for example, **Heat Transfer Coefficient**).

Figure 6.36: The User-Defined Functions Dialog Box

4. In the **User-Defined Functions** dialog box, select the function name (for example, `nuc_ht_coeff`) from the list of UDFs displayed and click **OK**.

    The name of the function will subsequently be displayed under the selected parameter (for example, **Heat Transfer Coefficient**) in the **Evaporation-Condensation Model** dialog box (Figure 6.35: The Evaporation-Condensation Model Dialog Box (p. 466)).

See `DEFINE_MASS_TR_PROPERTY` (p. 83) for details about `DEFINE_MASS_TR_PROPERTY` functions.

6.2.19. Hooking `DEFINE_NET_REACTION_RATE` UDFs

After you have compiled (Compiling UDFs (p. 399)) your `DEFINE_NET_REACTION_RATE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.37: The User-Defined Function Hooks Dialog Box (p. 469)) in Ansys Fluent.

To hook the UDF to Ansys Fluent, first set up the species transport and combustion models.

🗁 **Setup** → **Models** → **Species** 🖱 **Edit...**
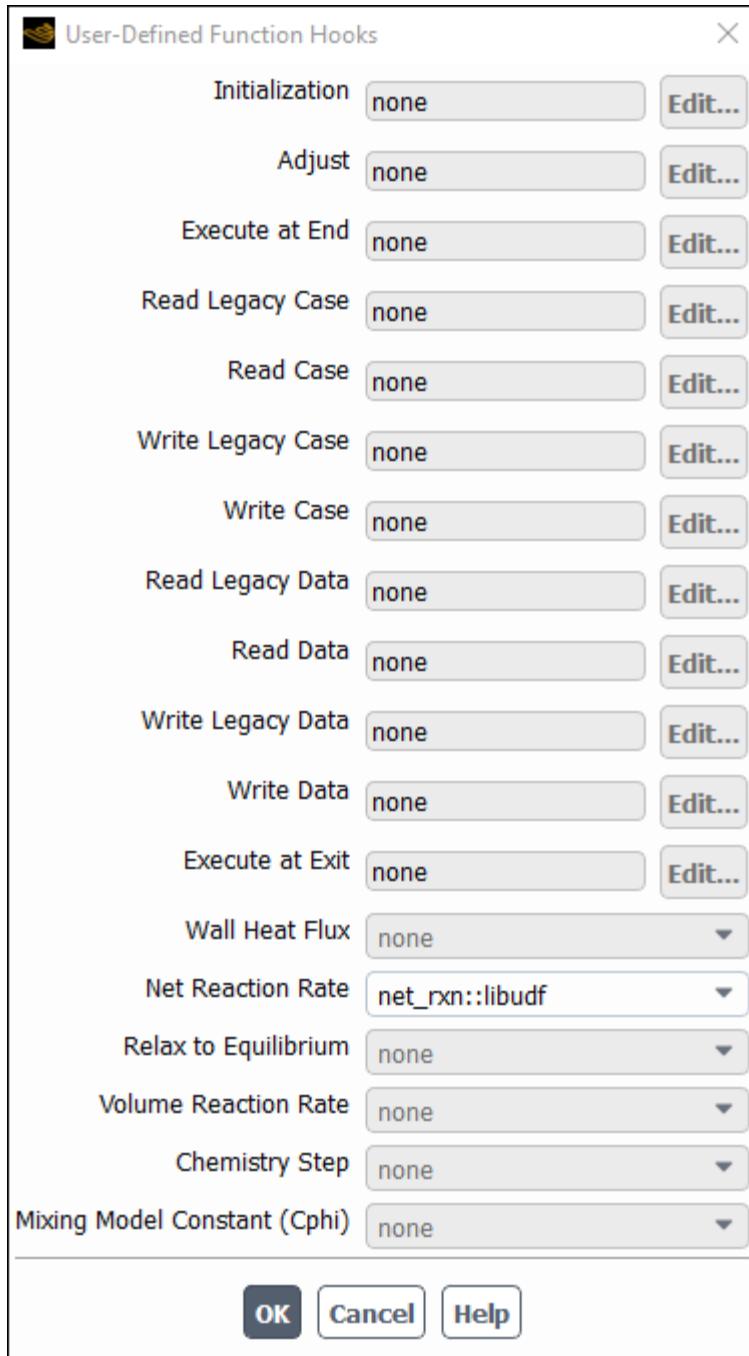
Note that net reaction rate UDFs can only be used with the laminar finite-rate model (with the stiff chemistry solver), the EDC model, the PDF Transport model, or the surface chemistry model. Therefore, you must use one of the following groups of settings in the **Species Model** dialog box:

• To enable the laminar finite-rate model, select **Species Transport**, enable **Volumetric** in the **Reactions** group box, select **Finite-Rate/No TCI** in the **Turbulence-Chemistry Interaction** group box, and select **Stiff Chemistry Solver** from the **Chemistry Solver** drop-down list.

- To enable the EDC model, select **Species Transport**, enable **Volumetric** in the **Reactions** group box, and select **Eddy-Dissipation Concept** in the **Turbulence-Chemistry Interaction** group box.

- To enable the PDF Transport model, select **Composition PDF Transport** and enable **Volumetric** in the **Reactions** group box.

- To enable the surface chemistry model, select **Species Transport** and enable **Volumetric** and **Wall Surface** in the **Reactions** group box.

Next, open the **User-Defined Function Hooks** dialog box (Figure 6.37: The User-Defined Function Hooks Dialog Box (p. 469)).

Parameters & Customization → **User Defined Functions** **Function Hooks...**

Figure 6.37: The User-Defined Function Hooks Dialog Box

Select the function name (for example, **net_rxn::libudf**) in the **Net Reaction Rate Function** drop-down list, and click **OK**.

See `DEFINE_NET_REACTION_RATE` (p. 85) for details about `DEFINE_NET_REACTION_RATE` functions.
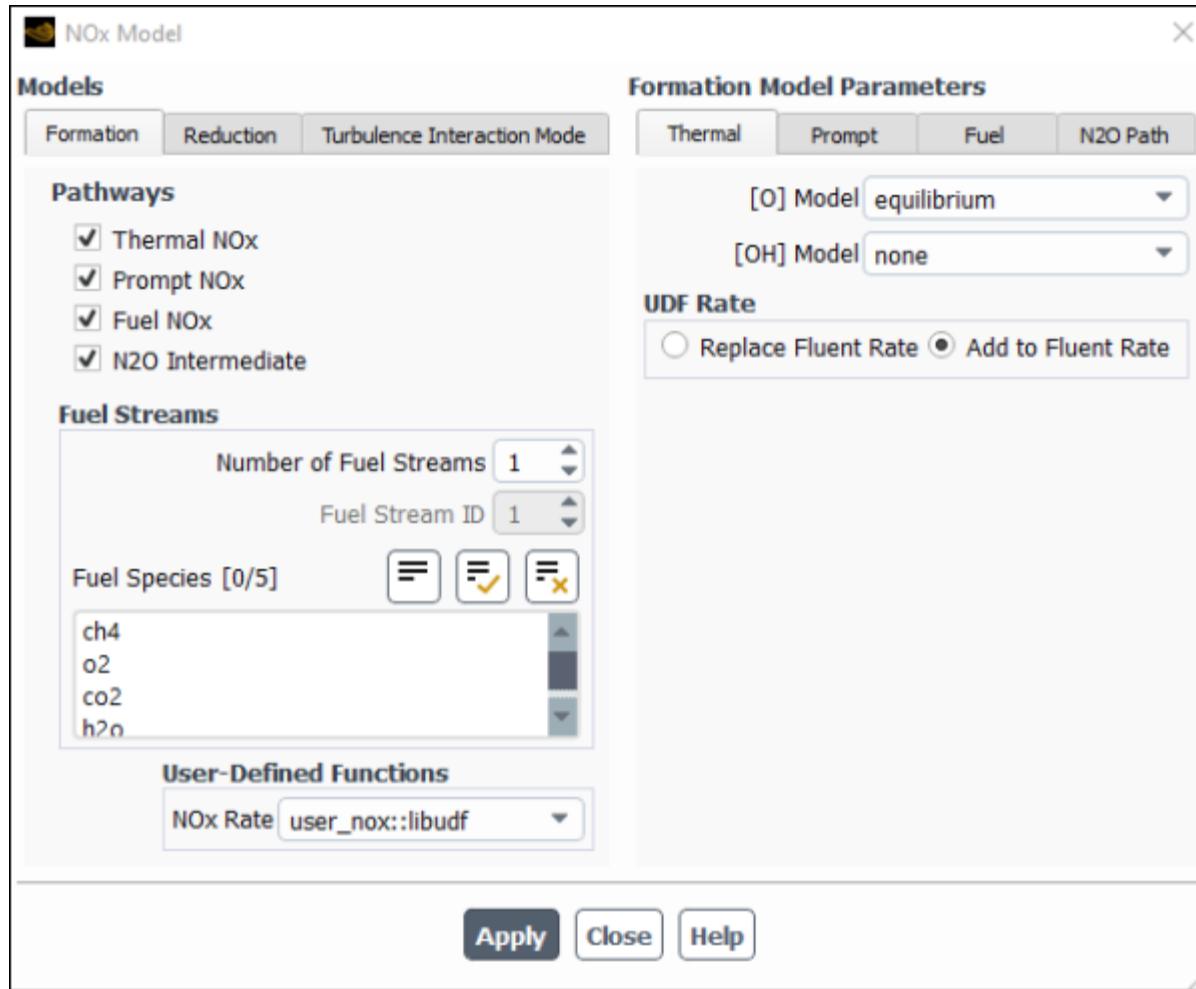
6.2.20. Hooking `DEFINE_NOX_RATE` UDFs

After you have compiled (Compiling UDFs (p. 399)) your `DEFINE_NOX_RATE` UDF in Ansys Fluent, the function name you supplied in the `DEFINE` macro argument will become visible and selectable

in the **NOx Rate** drop-down list in the **Formation** tab of the **NOx Model** dialog box (Figure 6.38: The NOx Model Dialog Box (p. 470)).

⊞ Setup → Models → Species → NOx ⌖ Edit...

Figure 6.38: The NOx Model Dialog Box

Recall that a single UDF is used to define custom rates for the thermal NOx, prompt NOx, fuel NOx, and N_2O NOx pathways. By default, the custom NOx rate of your UDF is added to the rate calculated internally by Ansys Fluent for each pathway. The UDF rate will be added to the forward rate if it is assigned to the `POLLUT_FRATE` macro, or the reverse rate if it is assigned to the `POLLUT_RRATE` macro. If you would rather entirely replace the internally calculated NOx rate with your custom rate, click the desired NOx pathway tab (**Thermal**, **Prompt**, **Fuel**, or **N2O Path**) under **Formation Model Parameters**, select **Replace Fluent Rate** in the **UDF Rate** group box for that pathway, and then click **Apply**. Repeat this process for all of the remaining NOx pathways.

Unless specifically defined in your NOx rate UDF, data and parameter settings for each individual NOx pathway will be derived from the settings in the **NOx Model** dialog box. Therefore, it is good practice to make the appropriate settings in the **NOx Model** dialog box, even though you can use a UDF to replace the default rates with user-specified rates. There is no computational penalty for doing this because the default rate calculations will be ignored when **Replace Fluent Rate** is selected.

To specify a custom maximum limit (T_{max}) for the integration of the temperature PDF for each cell, you must first select the UDF name (for example, **user_nox::libudf**) from the **NOx Rate** drop-down list, as described previously. Then, click the **Turbulence Interaction Mode** tab and select either **temperature** or **temperature/species** from the **PDF Mode** drop-down list. Finally, select **user-defined** from the **Tmax Option** drop-down list and click **Apply**.

See `DEFINE_NOX_RATE` (p. 87) for details about `DEFINE_NOX_RATE` functions.
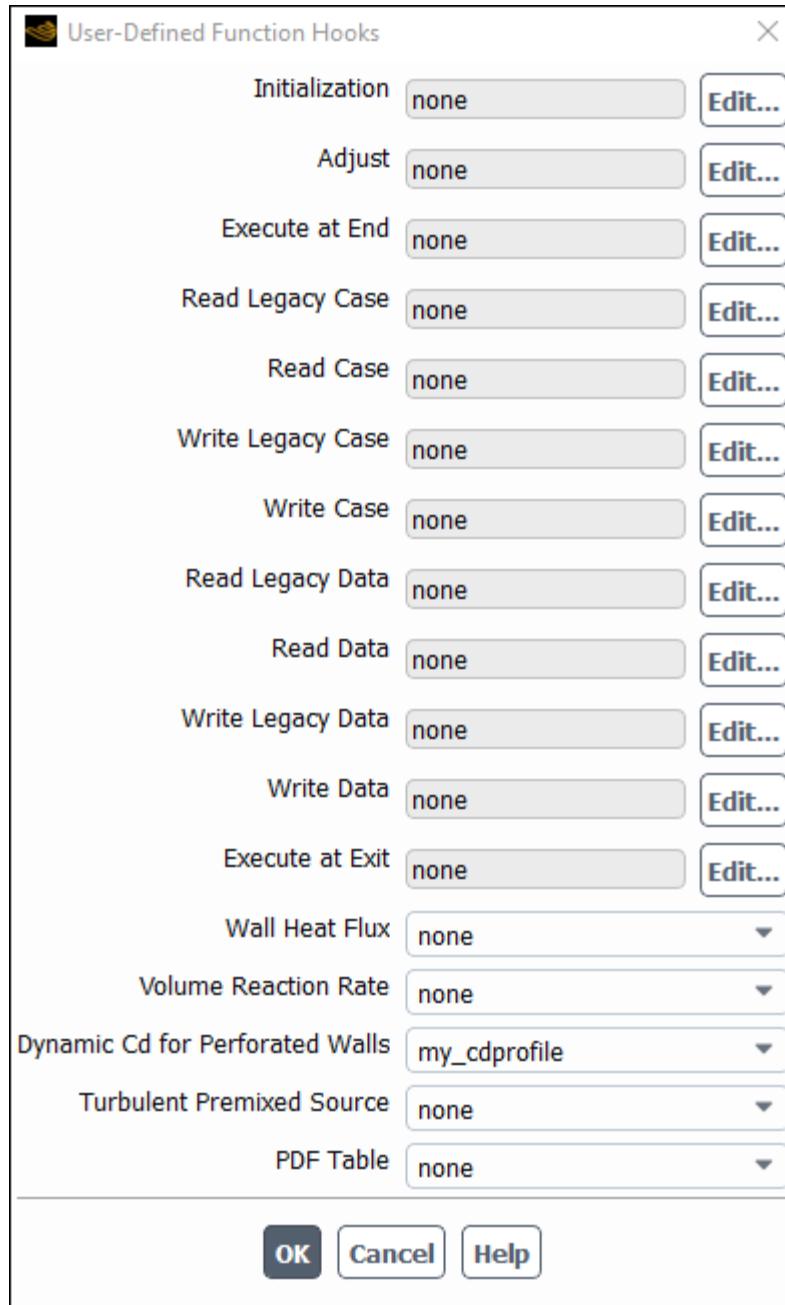
6.2.21. Hooking `DEFINE_PERFORATED_CD` UDFs

After you have compiled (Compiling UDFs (p. 399)) your `DEFINE_PERFORATED_CD` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.39: The User-Defined Function Hooks Dialog Box (p. 472)) in Ansys Fluent.

To hook the UDF to Ansys Fluent:
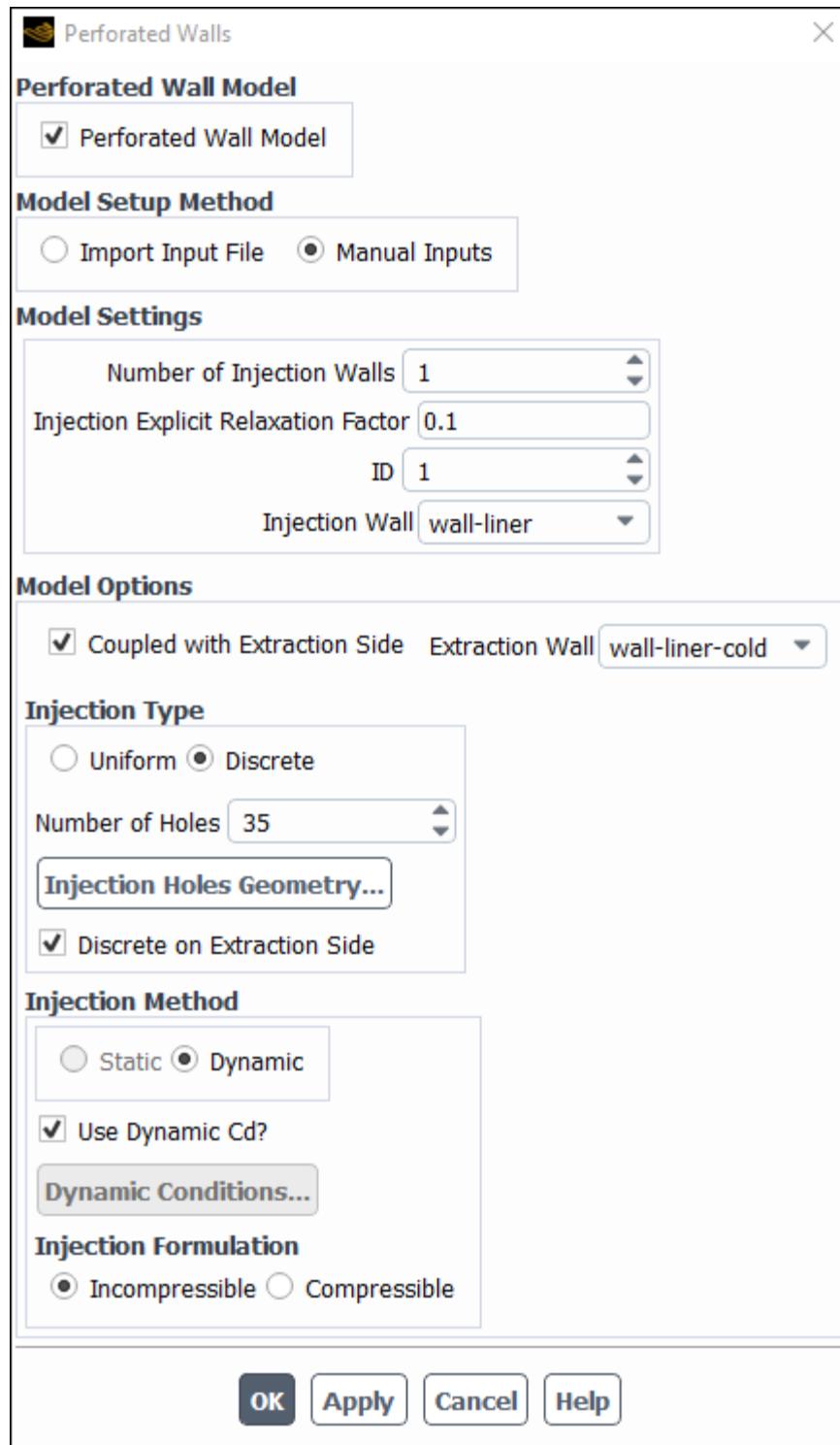
1. In the **User-Defined Function Hooks** dialog box, select the function name (for example, `my_cdprofile::libudf`) in the **Dynamic CD for Perforated Walls** drop-down list (Figure 6.39: The User-Defined Function Hooks Dialog Box (p. 472)).

   ▣ **Parameters & Customization** → **User Defined Functions** ⎊ **Function Hooks...**

Figure 6.39: The User-Defined Function Hooks Dialog Box

2. In the **Perforated Walls** dialog box, configure the appropriate settings for your case. Make sure that the following options are selected:

- **Discrete** (as the injection type)

- **Dynamic** (as the injection method)

- **Discrete on Extraction Side**

⊞ Setup → ⬦ Boundary Conditions → Perforated Walls...

Figure 6.40: The Perforated Walls Dialog Box

3. Select **Use Dynamic Cd?**

> **Note:**
>
> Only one **DEFINE_PERFORATED_CD** can be hooked. That is, if you are using a
> `DEFINE_PERFORATED_CD` UDF for multiple injections, the same UDF will be used for
> each injection.
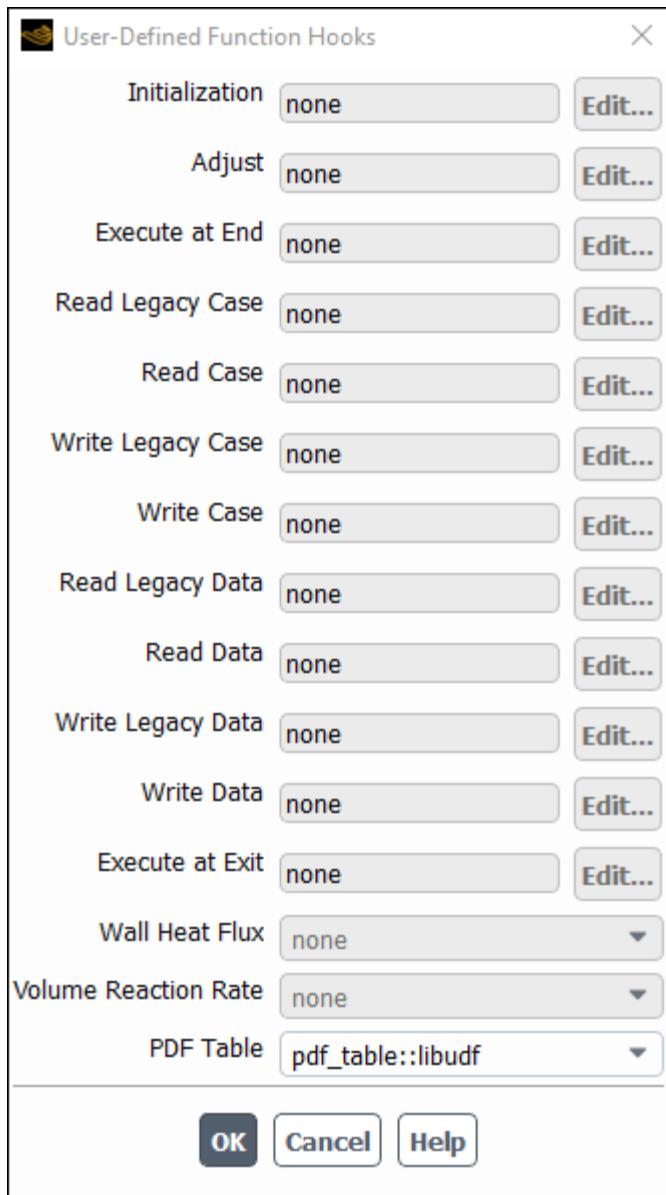
See `DEFINE_PERFORATED_CD` (p. 91) for details about defining `DEFINE_PERFORATED_CD`
functions.

6.2.22. Hooking `DEFINE_PDF_TABLE` UDFs

After you have compiled (Compiling UDFs (p. 399)) your `DEFINE_PDF_TABLE` UDF, the name of the
function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.41: The User-Defined Function Hooks Dialog Box (p. 475))
in Ansys Fluent.

To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box (Figure 6.42: The
User-Defined Function Hooks Dialog Box (p. 476)).

⊡ **Parameters & Customization** → **User Defined Functions** 🖱 **Function Hooks...**

Figure 6.41: The User-Defined Function Hooks Dialog Box

Important:

You must enable the **Non-Premixed** or **Partially-Premixed** models, and generate or read a valid PDF table.

Select the function name (for example, **pdf_table::libudf**) in the **PDF Table** drop-down list in the **User-Defined Function Hooks** dialog box, and click **OK**.

See `DEFINE_PDF_TABLE` (p. 93) for details about defining `DEFINE_PDF_TABLE` functions.

6.2.23. Hooking `DEFINE_PR_RATE` UDFs
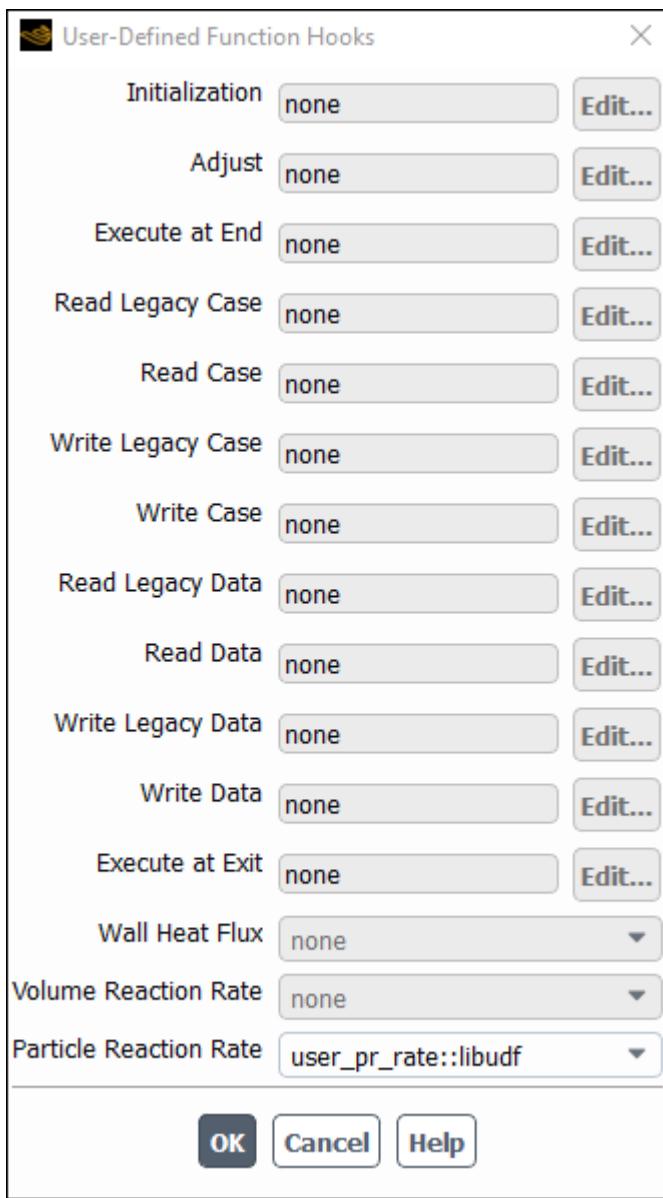
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_PR_RATE` UDF, the name of the function you supplied as a `DEFINE` macro argument will

become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.42: The User-Defined Function Hooks Dialog Box (p. 476)) in Ansys Fluent.

To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box (Figure 6.42: The User-Defined Function Hooks Dialog Box (p. 476)).

⊞ **Parameters & Customization** → **User Defined Functions** 🖱 **Function Hooks...**

Figure 6.42: The User-Defined Function Hooks Dialog Box

Important:

You must enable the particle surface reactions option before you can hook the UDF by selecting **Volumetric** and **Particle Surface** under **Reactions** in the **Species Model** dialog box.
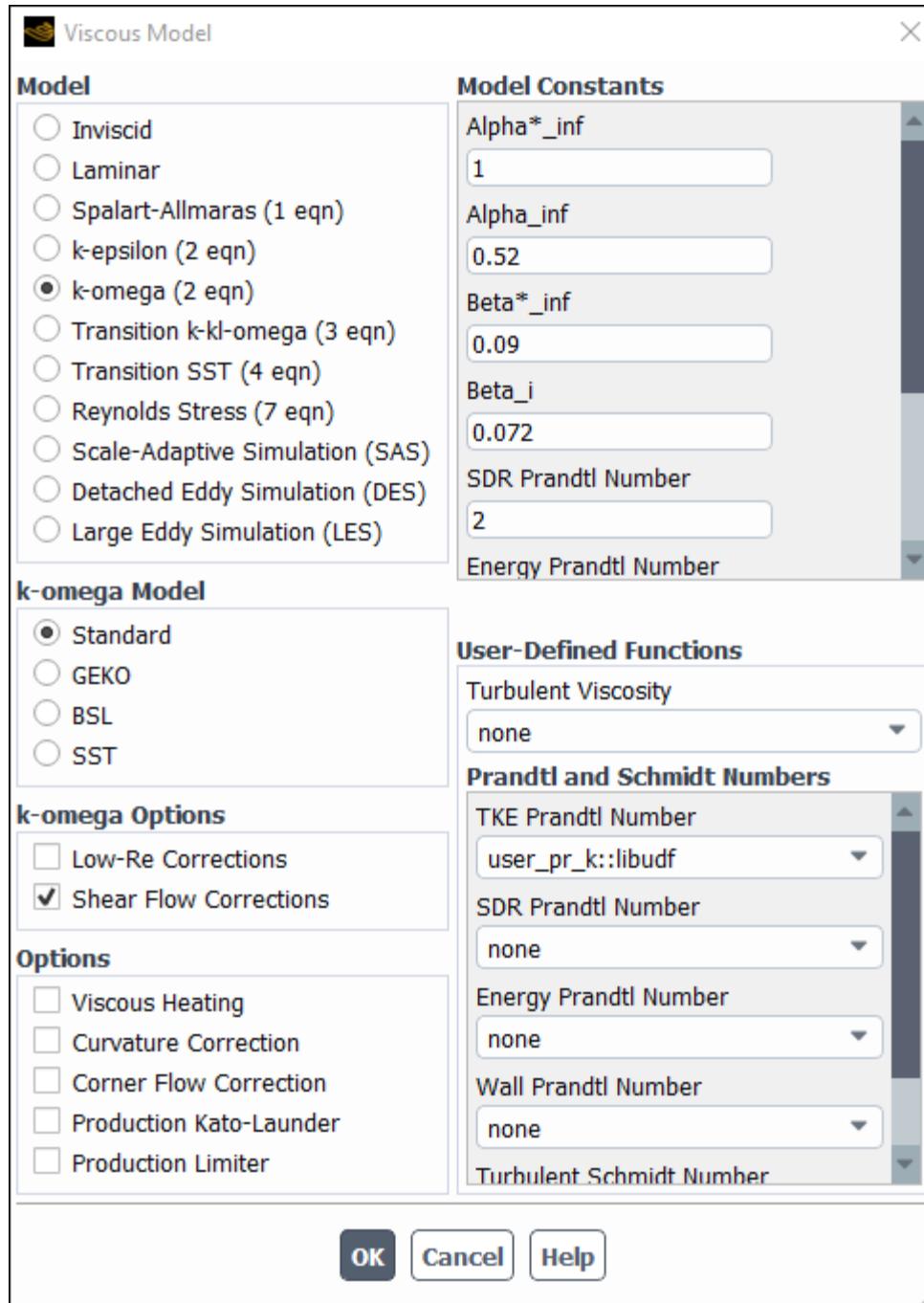
Select the function name (for example, **user_pr_rate::libudf**) in the **Particle Reaction Rate Function** drop-down list in the **User-Defined Function Hooks** dialog box, and click **OK**.

See `DEFINE_PR_RATE` (p. 99) for details about defining `DEFINE_PR_RATE` functions.

6.2.24. Hooking `DEFINE_PRANDTL` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_PRANDTL` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Viscous Model** dialog box (Figure 6.43: The Viscous Model Dialog Box (p. 478)) in Ansys Fluent.

Setup → Models → Viscous → Edit...

Figure 6.43: The Viscous Model Dialog Box

To hook the UDF to Ansys Fluent, select the function name (for example, **user_pr_k::libudf**) from the **TKE Prandtl Number** drop-down list under **User-Defined Functions** in the **Viscous Model** dialog box, and click **OK**.

See `DEFINE_PRANDTL` UDFs (p. 103) for details about `DEFINE_PRANDTL` functions.

6.2.25. Hooking `DEFINE_PROFILE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_PROFILE` UDF, the name of the function you supplied as a `DEFINE` macro argument will

become visible and selectable in the appropriate boundary zone condition dialog box, cell zone condition dialog box, or **Shell Conduction Layers** dialog box in Ansys Fluent. To open the boundary or cell zone condition dialog box, select the zone in the **Boundary Conditions** or **Cell Zone Conditions** task page and click the **Edit...** button.

⊨ **Setup** → **✦** **Boundary Conditions**

or
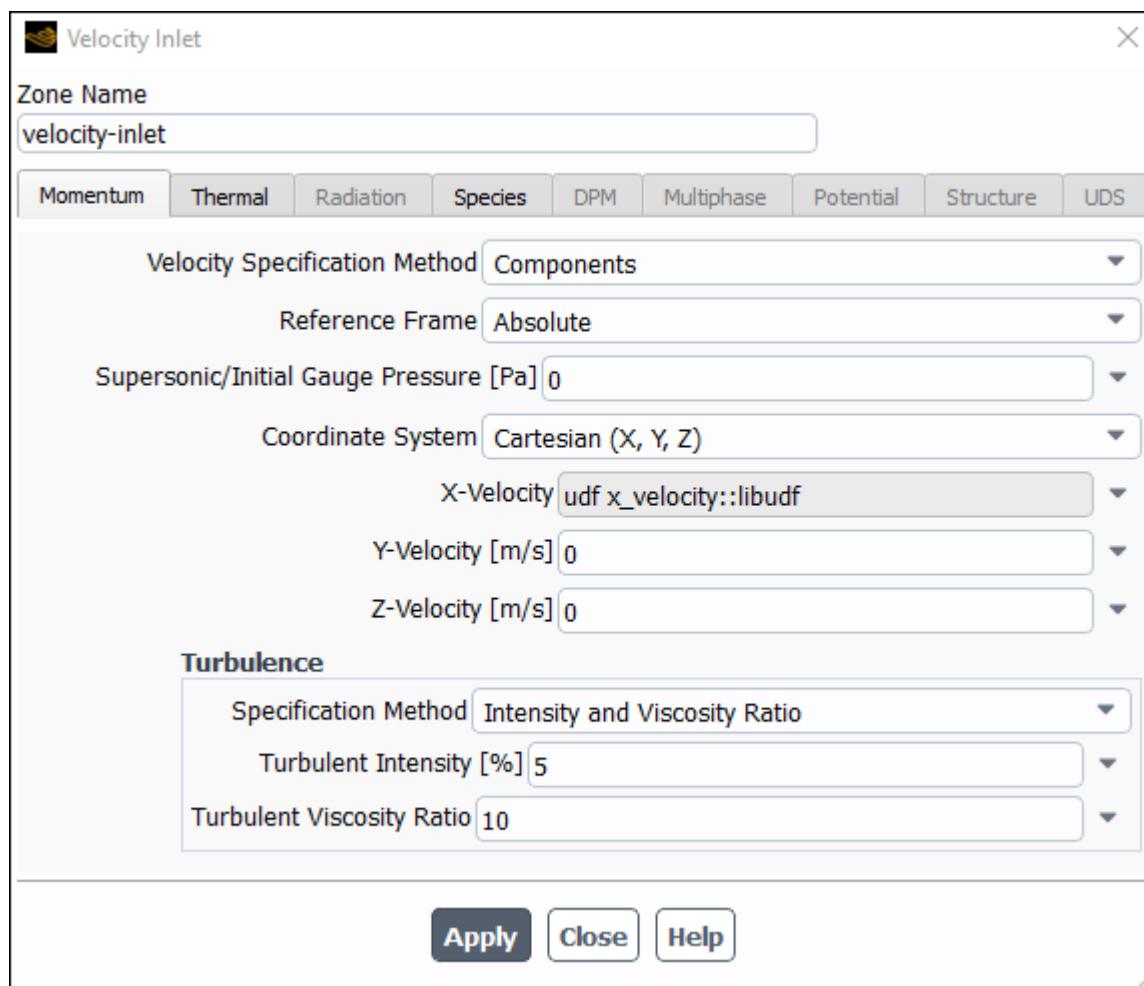
⊨ **Setup** → **✦** **Cell Zone Conditions**

To open the **Shell Conduction Layers** dialog box, enable the **Shell Conduction** option in the **Thermal** tab of the **Wall** dialog box, and then click the **Edit...** button.

To hook the UDF, select the name of your function in the appropriate drop-down list. For example, if your UDF defines a velocity inlet boundary condition, click the **Momentum** tab in the **Velocity Inlet** dialog box (Figure 6.44: The Velocity Inlet Dialog Box (p. 479)), select the function name (for example, **x_velocity::libudf**) from the **X Velocity** drop-down list, and click **OK**. Note that the UDF name that is displayed in the drop-down lists is preceded by the word **udf** (for example, **udf x_velocity::libudf**).

Figure 6.44: The Velocity Inlet Dialog Box

If you are using your UDF to specify a fixed value in a cell zone, you will need to turn on the **Fixed Values** option in the **Fluid** or **Solid** dialog box. Then click the **Fixed Values** tab and select the name of the UDF in the appropriate drop-down list for the value you want to set.

See `DEFINE_PROFILE` (p. 108) for details about `DEFINE_PROFILE` functions.

6.2.25.1. Hooking Profiles for UDS Equations

For each of the *N* scalar equations you have specified in your Ansys Fluent model using the **User-Defined Scalars** dialog box, you can hook a fixed-value UDF for a cell zone (for example, **Fluid** or **Solid**) and a specified-value or flux UDF for all wall, inflow, and outflow boundaries.
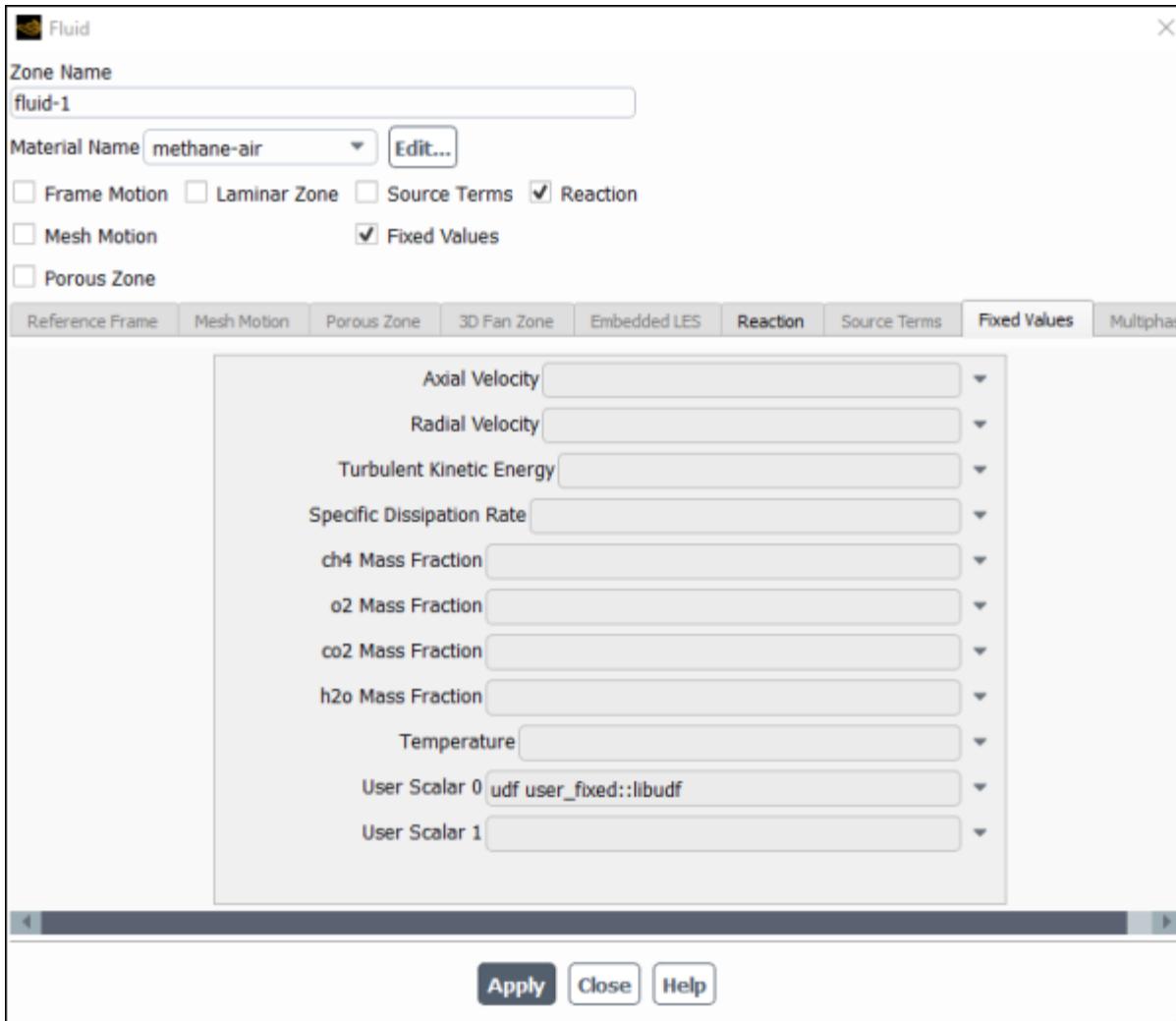
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_PROFILE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the appropriate boundary or cell zone condition dialog box in Ansys Fluent. To open the boundary or cell zone condition dialog box, select the zone in the **Boundary Conditions** or **Cell Zone Conditions** task page and click the **Edit...** button.

Setup → **Boundary Conditions**

or

Setup → **Cell Zone Conditions**
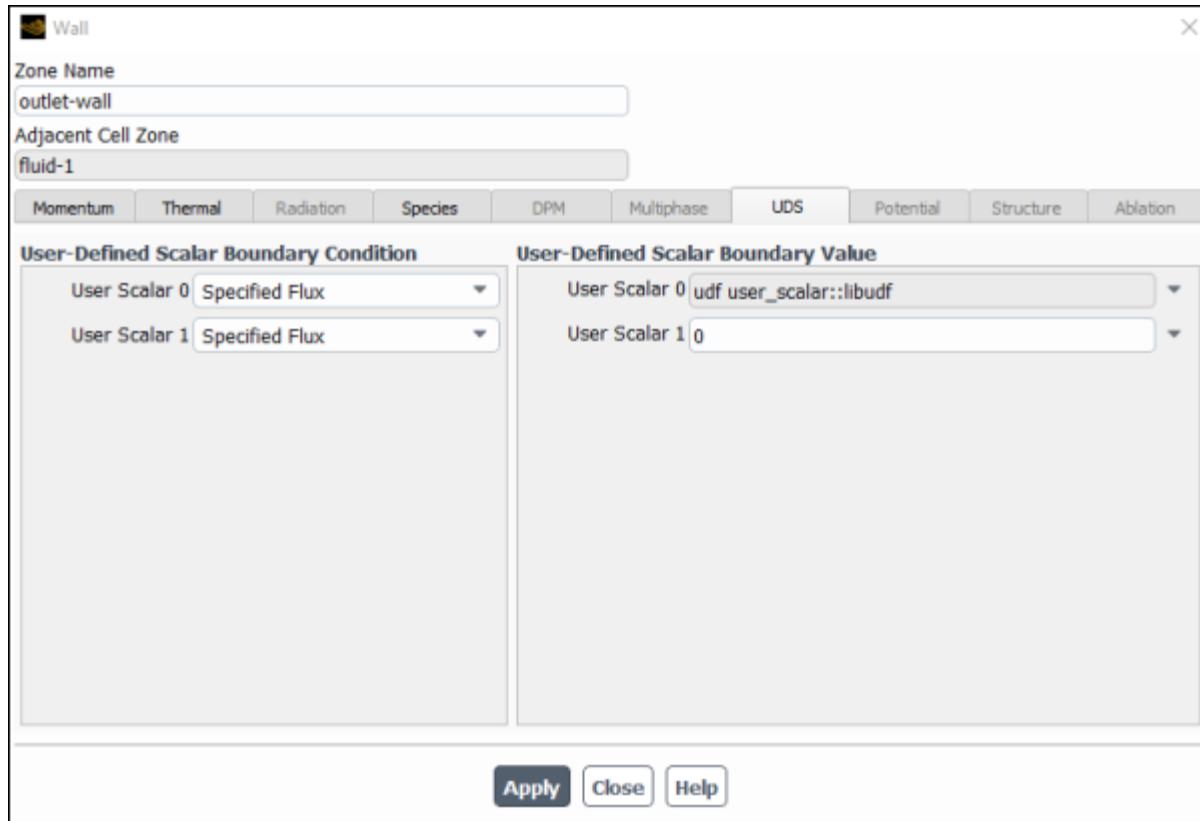
- If you are using your UDF to specify a fixed value in a cell zone, you will need to enable the **Fixed Values** option in the **Fluid** or **Solid** dialog box. Then click the **Fixed Values** tab (Figure 6.45: The Fluid Dialog Box with Fixed Value Inputs for User-Defined Scalars (p. 481)) and select the name of the UDF (for example, **fixed_scalar_0**) in the appropriate drop-down list for the value you want to set.

Figure 6.45: The Fluid Dialog Box with Fixed Value Inputs for User-Defined Scalars

- If you are using your UDF to define a specific value or flux for a scalar equation in a boundary condition dialog box, you will first need to enter a nonzero number in the **User-Defined Scalars** field in the **User-Defined Scalars** dialog box.

Parameters & Customization → **User Defined Scalars** → **Edit...**

Next, select the **UDS** tab in the wall, inflow, or outflow boundary dialog box (Figure 6.46: The Wall Dialog Box with Inputs for User-Defined Scalars (p. 482)).

Figure 6.46: The Wall Dialog Box with Inputs for User-Defined Scalars

For each UDS (**User Scalar 0**, **User Scalar 1**, and so on) specify the boundary condition value as a constant value or a UDF (for example, **user_scalar::libudf**) in the **User-Defined Scalar Boundary Value** group box. If you select **Specified Flux** in the **User-Defined Scalar Boundary Condition** group box for a particular UDS, then your input will be the value of the flux at the boundary (that is, the dot product of the negative of the term in parentheses on the left hand side of Equation 1.9 (in the Theory Guide) with the vector that is normal to the domain); if you instead select **Specified Value**, then your input will be the value of the scalar itself at the boundary. In the sample dialog box shown previously, for example, the **Specified Value** for **User Scalar 0** is set to a `user_scalar` UDF.

Note that for interior walls, you will need to select **Coupled Boundary** if the scalars are to be solved on both sides of a two-sided wall. Note that the **Coupled Boundary** option will show up only in the drop-down list when **all zones** is selected for **Solution Zones** in the **User-Defined Scalars** dialog box.

Important:

In some cases, you may want to exclude diffusion of the scalar at the inlet of your domain. You can do this by disabling **Inlet Diffusion** for the scalar in the **User-Defined Scalars** dialog box.
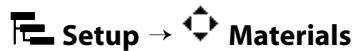
See `DEFINE_PROFILE` (p. 108) for details about `DEFINE_PROFILE` functions.

6.2.26. Hooking `DEFINE_PROPERTY` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your material property UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in Ansys Fluent.
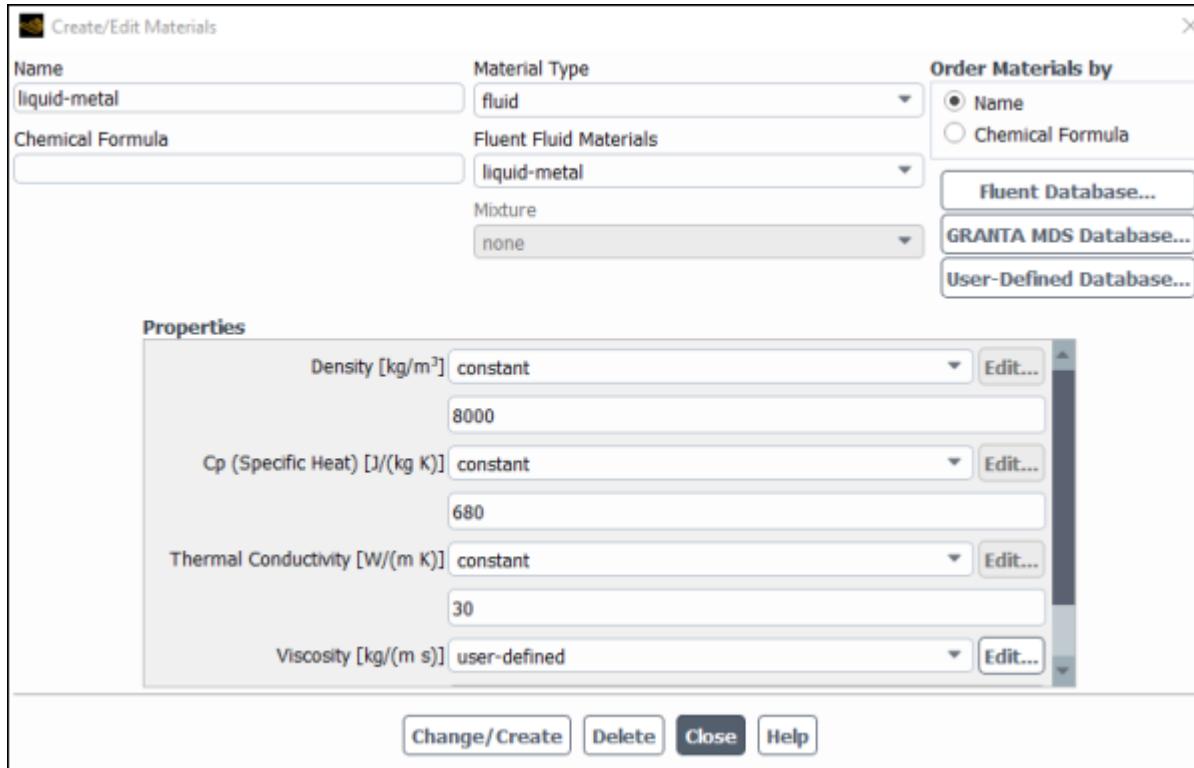
There are various dialog boxes in which you can activate a `DEFINE_PROPERTY` UDF (for example, **Multiphase Model** dialog box), and so the method for hooking it will depend on the property being defined. The following is an example of hooking a UDF that defines viscosity.

First, open the **Materials** task page.
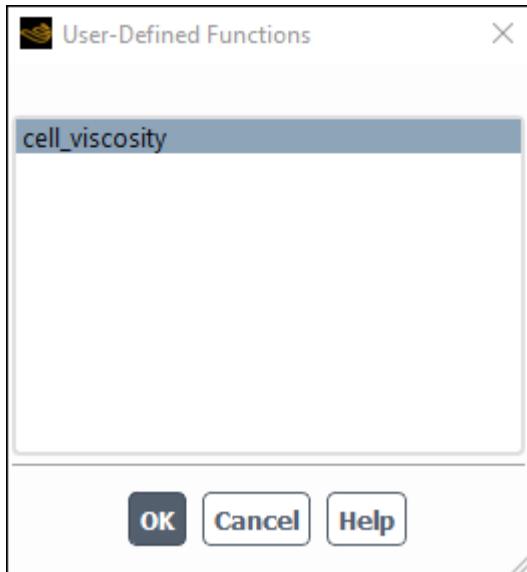
 Setup → ⬩ Materials

Select the appropriate material from the **Material** selection list and click the **Create/Edit...** button to open the **Create/Edit Materials** dialog box (Figure 6.47: The Create/ Edit Materials Dialog Box (p. 483)).

Figure 6.47: The Create/ Edit Materials Dialog Box

Next, open the **User-Defined Functions** dialog box (Figure 6.48: The User-Defined Functions Dialog Box (p. 484)) by choosing **user-defined** in the drop-down list for the appropriate property (for example, **Viscosity**) in the **Create/Edit Materials** dialog box. Then select the function name (for example, **cell_viscosity::libudf**) from the list of UDFs displayed in the **User-Defined Functions** dialog box and click **OK**. The name of the function will subsequently be displayed under the selected property in the **Create/Edit Materials** dialog box.

Figure 6.48: The User-Defined Functions Dialog Box

Important:

If you plan to define density using a UDF, note that the solution convergence will become poor as the density variation becomes large. Specifying a compressible law (density as a function of pressure) or multiphase behavior (spatially varying density) may lead to divergence. It is recommended that you restrict the use of UDFs for density to weakly compressible flows with mild density variations.

See `DEFINE_PROPERTY` UDFs (p. 119) for details about `DEFINE_PROPERTY` functions.

6.2.27. Hooking `DEFINE_REACTING_CHANNEL_BC` UDFs

After you have compiled (Compiling UDFs (p. 399)) your `DEFINE_REACTING_CHANNEL_BC` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User Defined Function** drop-down list under the **Group Inlet Conditions** tab of the **Reacting Channel Model** dialog box in Ansys Fluent. Note that the UDF hook is only available when the **User Defined Inlet Conditions** option is selected.

🔢 **Setup → Models → Reacting Channel Model** 🖱 **Edit...**

Select the function name (for example, **tube3_bc_from_1_and_2::libudf**) from the **User Defined Function** drop-down list under the **Group Inlet Conditions** tab and click **Apply**.

See `DEFINE_REACTING_CHANNEL_BC` (p. 125) for details about defining DEFINE_REACTING_CHANNEL_BC functions.

6.2.28. Hooking `DEFINE_REACTING_CHANNEL_SOLVER` UDFs
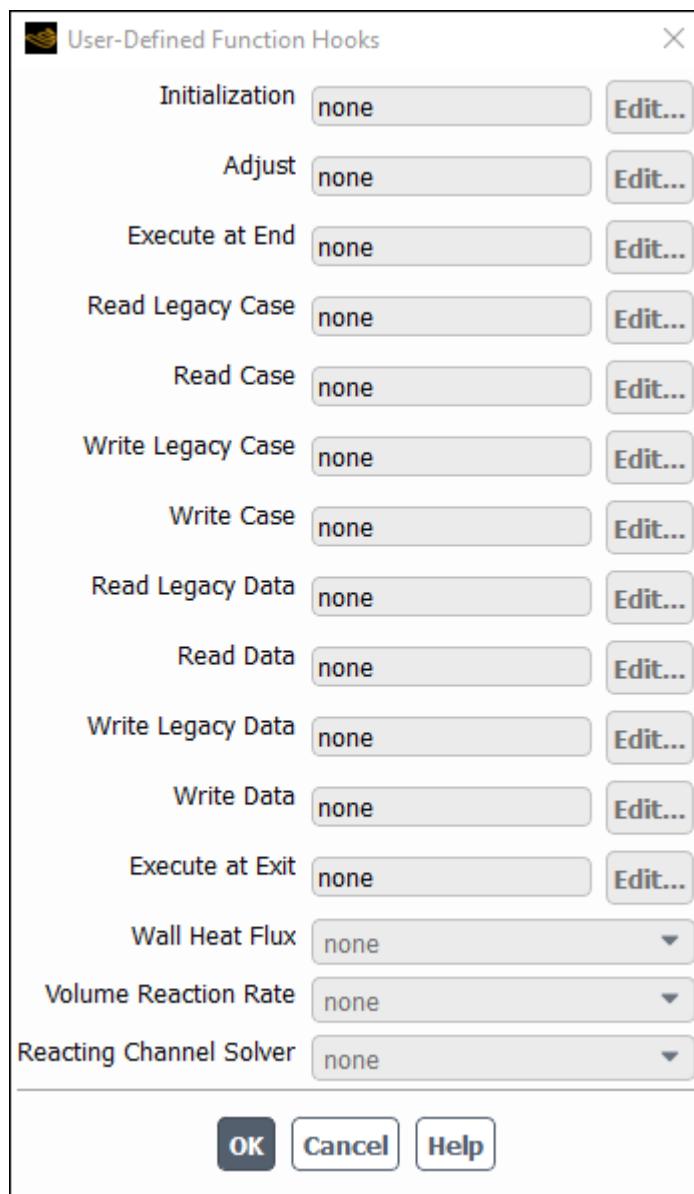
After you have compiled (Compiling UDFs (p. 399)) your `DEFINE_REACTING_CHANNEL_SOLVER` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and

selectable in the **User-Defined Function Hooks** dialog box (Figure 6.49: The User-Defined Function Hooks Dialog Box (p. 485)) in Ansys Fluent.

To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box.

Parameters & Customization → **User Defined Functions** **Function Hooks...**

Figure 6.49: The User-Defined Function Hooks Dialog Box

Important:

You must enable the **Reacting Channel Model** first before hooking your UDF.

Select the function name (for example, **set_channel_htc::libudf**) in the **Reacting Channel Solver** drop-down list in the **User-Defined Function Hooks** dialog box, and click **OK**.

See `DEFINE_REACTING_CHANNEL_SOLVER` for details about defining `DEFINE_REACT-ING_CHANNEL_SOLVER` functions.

6.2.29. Hooking `DEFINE_RELAX_TO_EQUILIBRIUM` UDFs

After you have interpreted or compiled your `DEFINE_RELAX_TO_EQUILIBRIUM` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box ().

To hook the UDF to Ansys Fluent:

1. Set up the species transport model.

    Setting Up Physics → **Models** → **Species...**

2. Enable **Volumetric** in the **Reactions** group box.

3. Select **Relax to Chemical Equilibrium** from the **Chemistry Solver** drop-down list.

4. Open the **User-Defined Function Hooks** dialog box.

    User Defined → **User Defined** → **Function Hooks...**

Figure 6.50: The User-Defined Function Hooks Dialog Box

5. Select the function name (for example, `relax_tau::libudf`) from the **Relax to Equilibrium** drop-down list, and click **OK**.

See `DEFINE_RELAX_TO_EQUILIBRIUM` (p. 131) for details about `DEFINE_RELAX_TO_EQUILIB-RIUM` functions.

6.2.30. Hooking `DEFINE_SBES_BF` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_SBES_BF` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent.
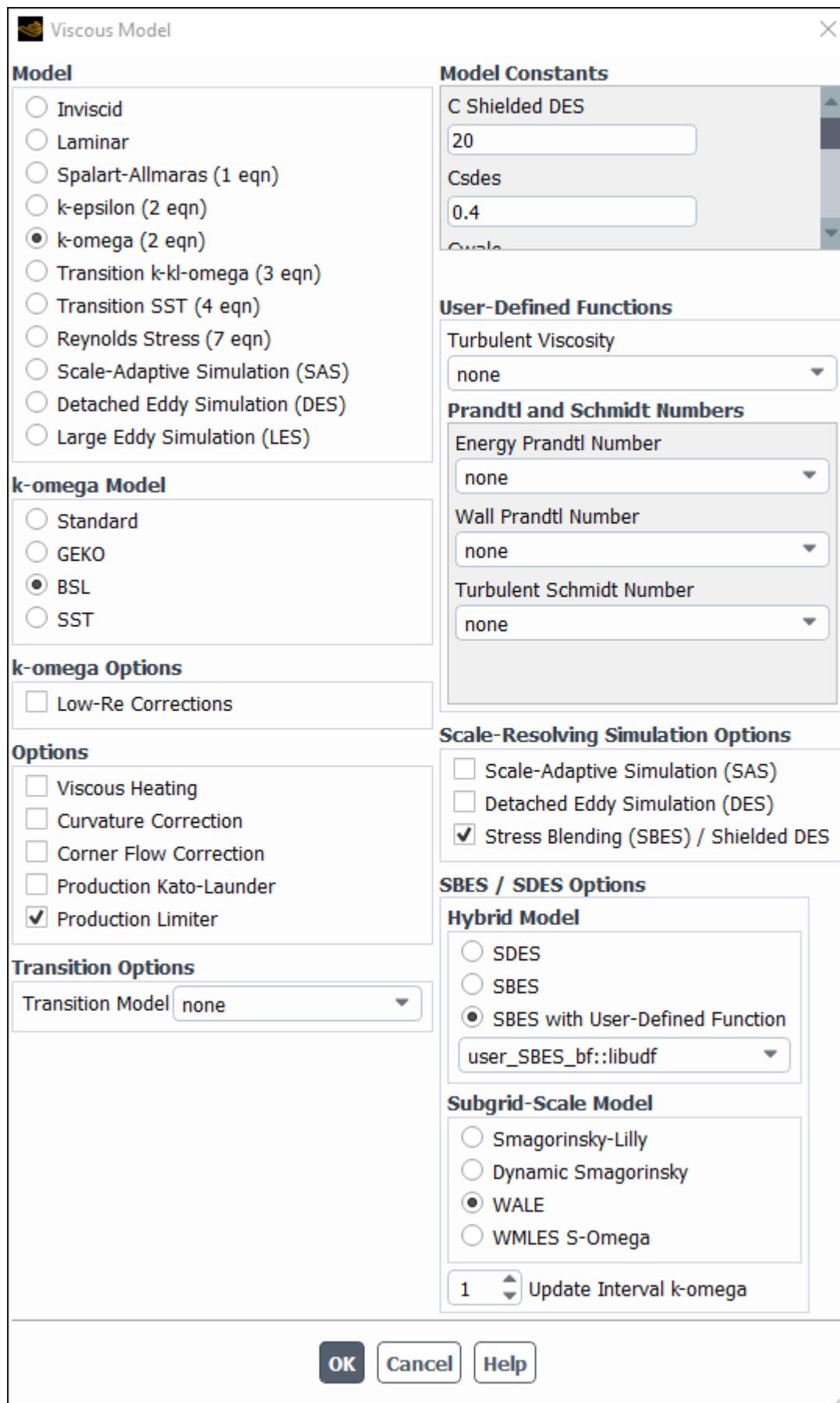
To hook the UDF to the 3D version of Ansys Fluent, you must first right-click the **General** branch of the tree and select **Transient** from the **Analysis Type** sub-menu.

Setup → **General** → **Analysis Type** → **Transient**

Next, open the **Viscous Model** dialog box.

Setup → **Models** → **Viscous** → **Edit...**

Figure 6.51: The Viscous Model Dialog Box

Select either **k-omega** or **Transition SST** from the **Model** list, and (for **k-omega**) either **BSL** or **SST** from the **k-omega Model** list. Next, enable the **Stress Blending (SBES) / Shielded DES** option, and in the **Hybrid Model** group box select **SBES with User-Defined Function** and then the function name (for example, **user_SBES_bf::libudf**) from the drop-down list.

See `DEFINE_SBES_BF` (p. 132) for details about `DEFINE_SBES_BF` functions.

6.2.31. Hooking `DEFINE_SCAT_PHASE_FUNC` UDFs
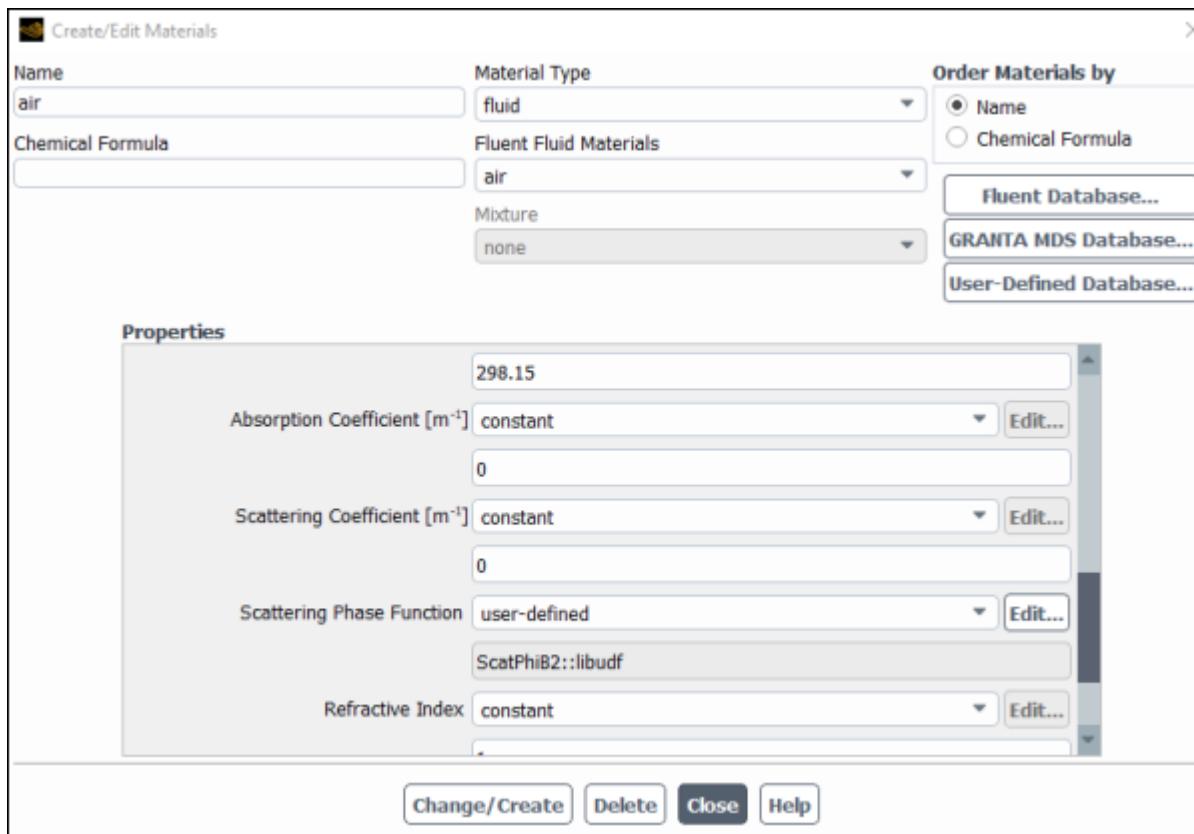
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_SCAT_PHASE_FUNC` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Functions** dialog box in Ansys Fluent.

To hook the UDF to Ansys Fluent, first make sure that the **Discrete Ordinates (DO)** model is selected in the **Radiation Model** dialog box. Then open the **Materials** task page.

🗁 Setup → ✥ Materials

Select the appropriate material from the **Material** selection list and click the **Create/Edit...** button to open the **Create/Edit Materials** dialog box (Figure 6.52: The Create/Edit Materials Dialog Box (p. 491)).
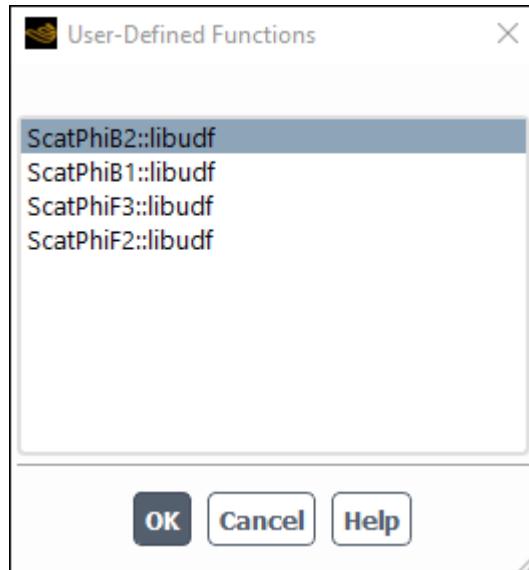
Figure 6.52: The Create/Edit Materials Dialog Box

Open the **User-Defined Functions** dialog box (Figure 6.53: The User-Defined Functions Dialog Box (p. 492)) from the **Create/Edit Material** dialog box by selecting **user-defined** in the drop-down list for the **Scattering Phase Function** property. Then, select the function name (for example, **ScatPhiB2**) from the list of UDFs displayed in the **User-Defined Functions** dialog box, and click **OK**. The

name of the function will subsequently be displayed under the **Scattering Phase Function** property in the **Create/Edit Materials** dialog box.

Figure 6.53: The User-Defined Functions Dialog Box
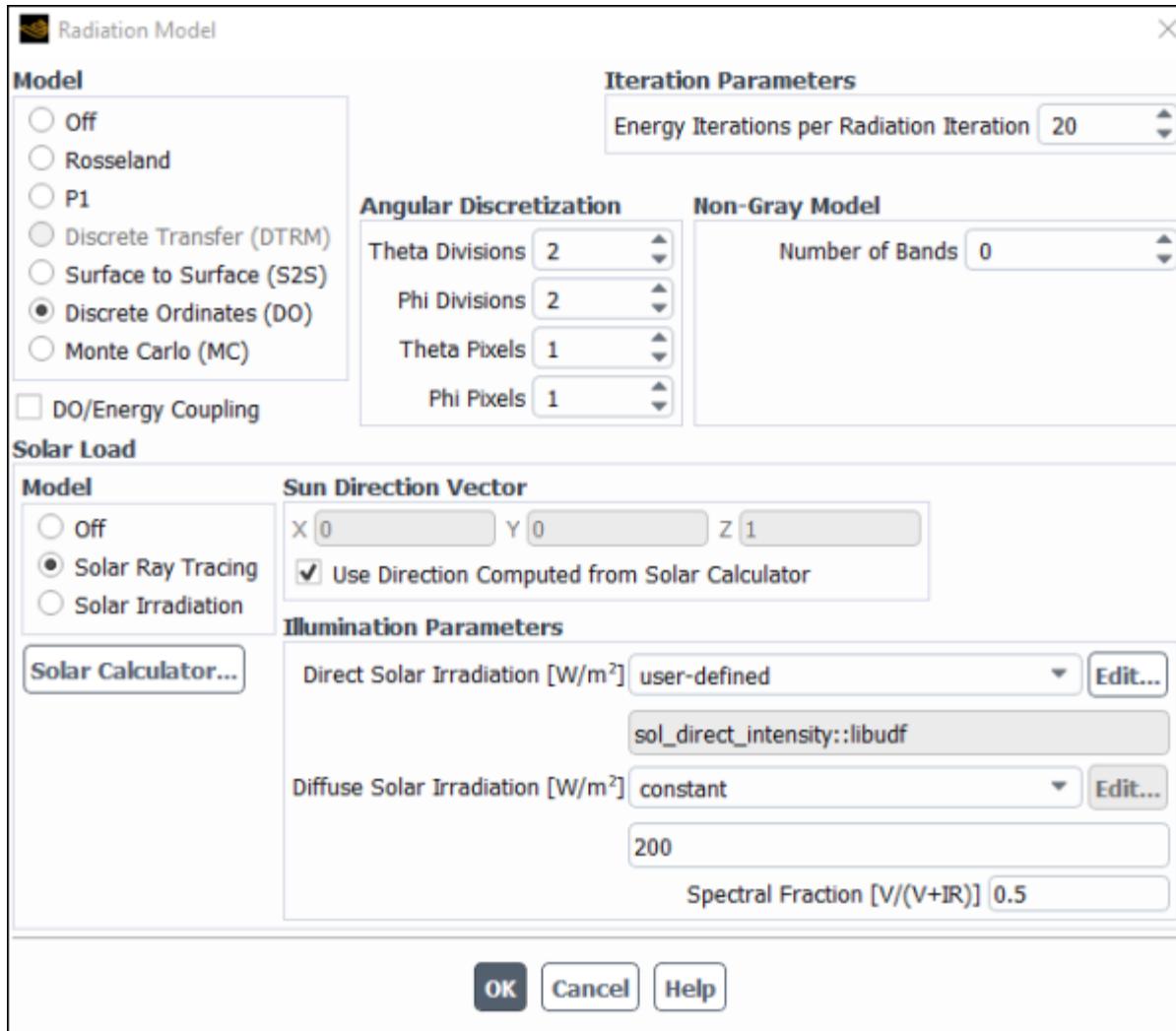
See `DEFINE_SCAT_PHASE_FUNC` for details about `DEFINE_SCAT_PHASE_FUNC` functions.

6.2.32. Hooking `DEFINE_SOLAR_INTENSITY` UDFs
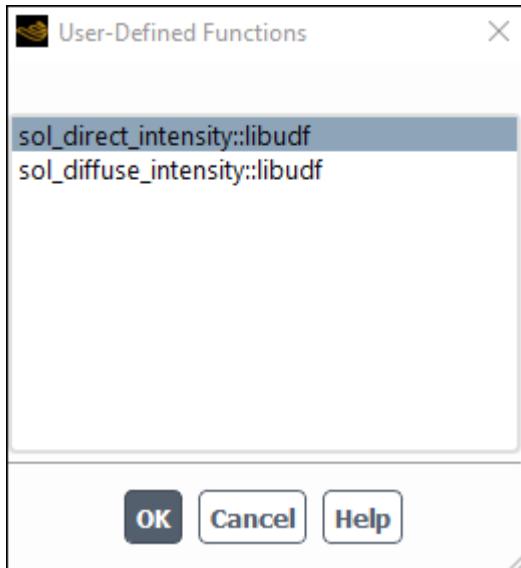
After you have interpreted () or compiled () your `DEFINE_SOLAR_INTENSITY` UDF, the name of the function you supplied in the argument of the `DEFINE` macro will become visible and selectable in the **User-Defined Functions** dialog box in Ansys Fluent.

To hook the UDF, first open the **Radiation Model** dialog box ().

 Setup → **Models** → **Radiation** **Edit...**

Figure 6.54: The Radiation Model Dialog Box

Select **Discrete Ordinates (DO)** from the **Model** list, and select **Solar Ray Tracing** in the **Solar Load** group box. In the **Illumination Parameters** group box that appears, select **user-defined** from the **Direct Solar Irradiation** or **Diffuse Solar Irradiation** drop-down list to open the **User-Defined Functions** dialog box (Figure 6.55: The User-Defined Functions Dialog Box (p. 494)).

Figure 6.55: The User-Defined Functions Dialog Box

Select the function name (for example, **sol_direct_intensity::libudf**) from the list of UDFs displayed in the **User-Defined Functions** dialog box and click **OK**. The name of the function will subsequently be displayed under the selected property (for example, **Direct Solar Irradiation**) in the **Radiation Model** dialog box (Figure 6.54: The Radiation Model Dialog Box (p. 493)).
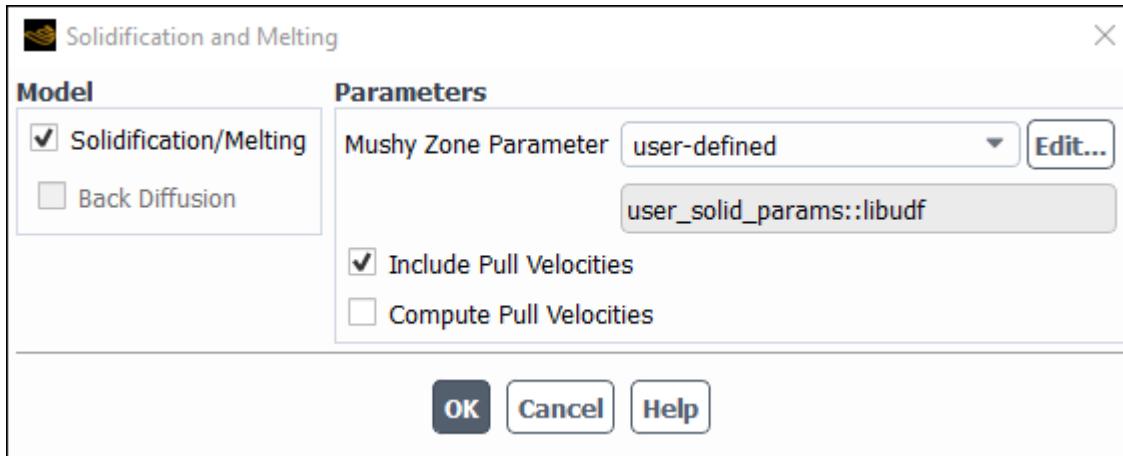
See `DEFINE_SOLAR_INTENSITY` (p. 135) for details about `DEFINE_SOLAR_INTENSITY` functions.

6.2.33. Hooking `DEFINE_SOLIDIFICATION_PARAMS` UDFs
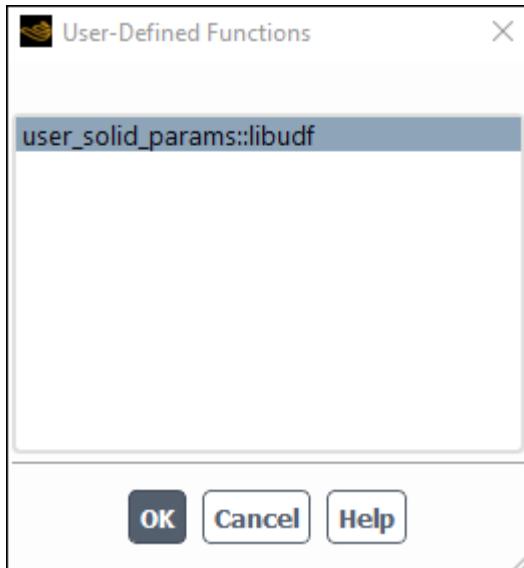
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_SOLIDIFICATION_PARAMS` UDF, the name of the function you supplied in the argument of the `DEFINE` macro will become visible and selectable in the **User-Defined Functions** dialog box in Ansys Fluent.

To hook the UDF, first open the **Solidification and Melting** dialog box (Figure 6.56: The Solidification and Melting Dialog Box (p. 495)).

⊡ **Setup** → **Models** → **Solidification & Melting** 🖱 **Edit...**

Figure 6.56: The Solidification and Melting Dialog Box

Enable **Solidification/Melting** under **Model**. Select **user-defined** from the **Mushy Zone Parameter** drop-down list to open the **User-Defined Functions** dialog box (Figure 6.57: The User-Defined Functions Dialog Box (p. 495)). You can also select **user-defined** from the **Mush Zone Parameter** drop-down list.

Figure 6.57: The User-Defined Functions Dialog Box

Select the function name (for example, **user_solid_params::libudf**) from the list of UDFs displayed in the **User-Defined Functions** dialog box and click **OK**. The name of the function will subsequently be displayed under the selected property.

See `DEFINE_SOLIDIFICATION_PARAMS` (p. 136) for details about `DEFINE_SOLIDIFICA-TION_PARAMS` functions.

6.2.34. Hooking `DEFINE_SOURCE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_SOURCE` UDF, the name of the function you supplied as a `DEFINE` macro argument will
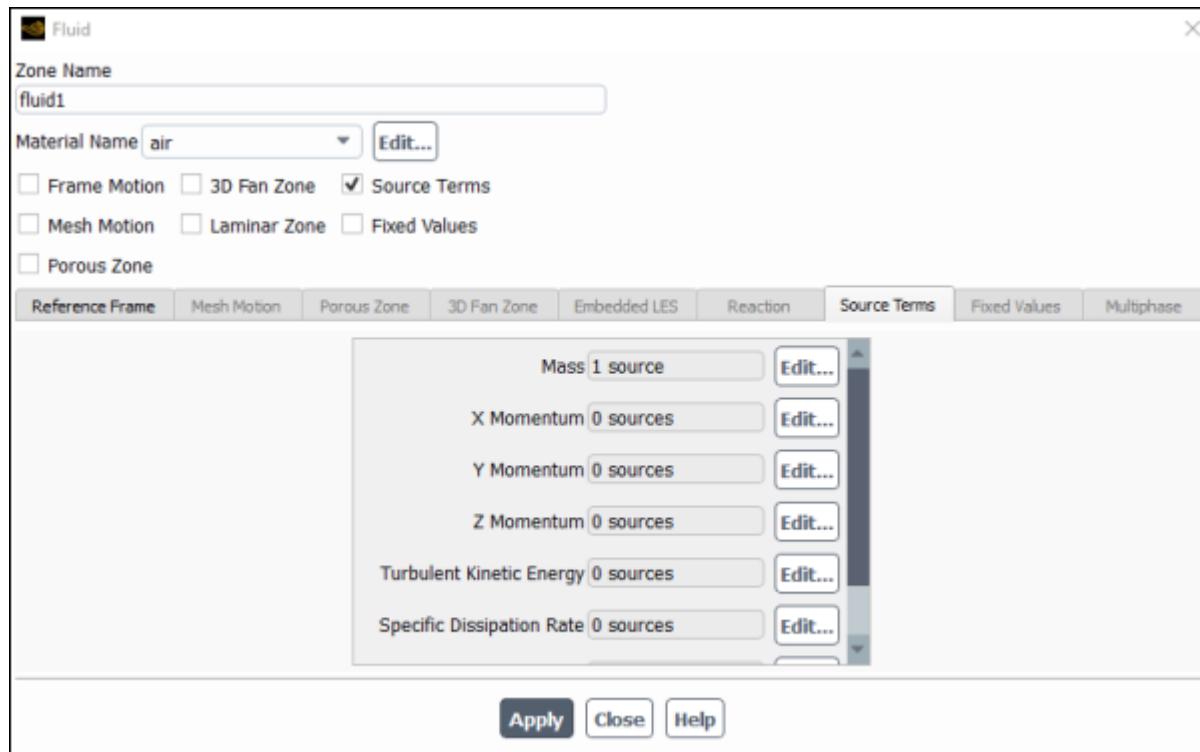
become visible and selectable in a source term dialog box (for example, the **Mass sources** dialog box) in Ansys Fluent.

To hook the UDF to Ansys Fluent, you will first need to open the **Cell Zone Conditions** task page.

📇 Setup → ✛ Cell Zone Conditions
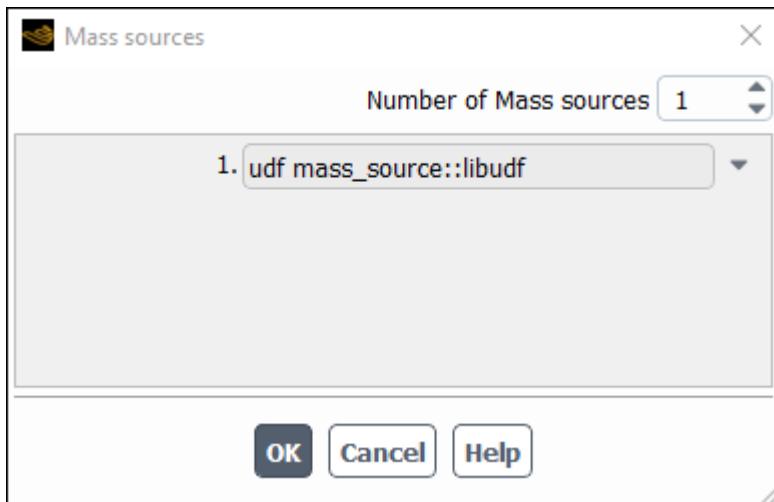
Select the appropriate zone in the **Zone** selection list of the **Cell Zone Conditions** task page and click the **Edit...** button to open the cell zone condition dialog box (for example, the **Fluid** dialog box, as shown in Figure 6.58: The Fluid Dialog Box (p. 496)).

Figure 6.58: The Fluid Dialog Box

Next, enable the **Source Terms** option in the cell zone condition dialog box and click the **Source Terms** tab. This will display the source term parameters (mass, momentum, and so on) in the scrollable window. Click the **Edit...** button next to the source term (for example, **Mass**) you want to customize, in order to open the appropriate source term dialog box (for example, the **Mass sources** dialog box, as shown in Figure 6.59: The Mass sources Dialog Box (p. 497)).

Figure 6.59: The Mass sources Dialog Box

Specify the number of terms you want to model by setting the **Number of Mass Sources** text-entry box (for example, 1) and then select the function name (for example, **mass_source::libudf**) from the appropriate drop-down list.

(Note that the UDF name that is displayed in the drop-down lists is preceded by the word **udf**.) Click **OK** in the **Mass sources** dialog box to accept the new boundary condition. The source term field in the cell zone condition dialog box will display the number of sources (for example, **1 source**). Click **OK** to close the cell zone condition dialog box and fix the new mass source terms for the solution calculation.

Repeat this step for all of the source terms you want to customize using a UDF.

See `DEFINE_SOURCE` (p. 148) for details about `DEFINE_SOURCE` functions.
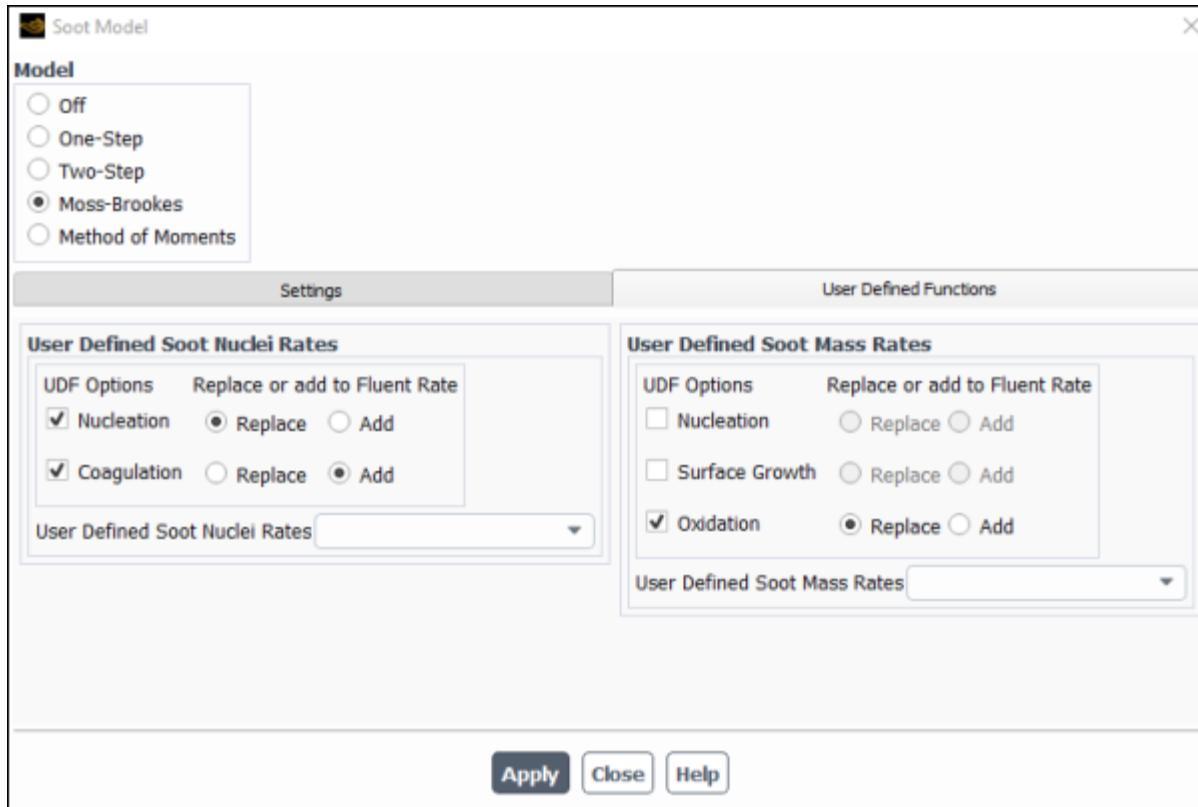
6.2.35. Hooking `DEFINE_SOOT_MASS_RATES` UDFs

After you have compiled (Compiling UDFs (p. 399)) your `DEFINE_SOOT_MASS_RATES` UDF in Ansys Fluent, the function name you supplied as a `DEFINE` macro argument will become visible and selectable from the **User Defined Soot Mass Rates** drop-down list under the **User Defined Functions** tab of the **Soot Model** dialog box (Figure 6.60: The Soot Model Dialog Box (User-Defined Mass and Nucleation Rates) (p. 498)).

> **Note:**
>
> The UDF hook is only available when the **Moss-Brookes** or the **Moss-Brookes-Hall** soot model is enabled.

📇 **Setup** → **Models** → **Species** → **Soot** 🖱 **Edit...** → **Moss-Brookes**

Figure 6.60: The Soot Model Dialog Box (User-Defined Mass and Nucleation Rates)

Under the **User Defined Functions** tab, select the function name (for example, `user_soot_mass_rates::libudf`) from the **User Defined Soot Mass Rates** drop-down list, and click **Apply**.

See `DEFINE_SOOT_MASS_RATES` (p. 137) for details about defining `DEFINE_SOOT_MASS_RATES` functions.

6.2.36. Hooking `DEFINE_SOOT_MOM_RATES` UDFs

After you have compiled (Compiling UDFs (p. 399)) your `DEFINE_SOOT_MOM_RATES` UDF in Ansys Fluent, the function name you supplied as a `DEFINE` macro argument will become visible and select-able from the **User Defined Soot MOM Rates** drop-down list (**User Defined Functions** group box) in the **Soot Model** dialog box (Figure 6.61: The Soot Model Dialog Box (User-Defined Soot MOM Rates) (p. 499)).
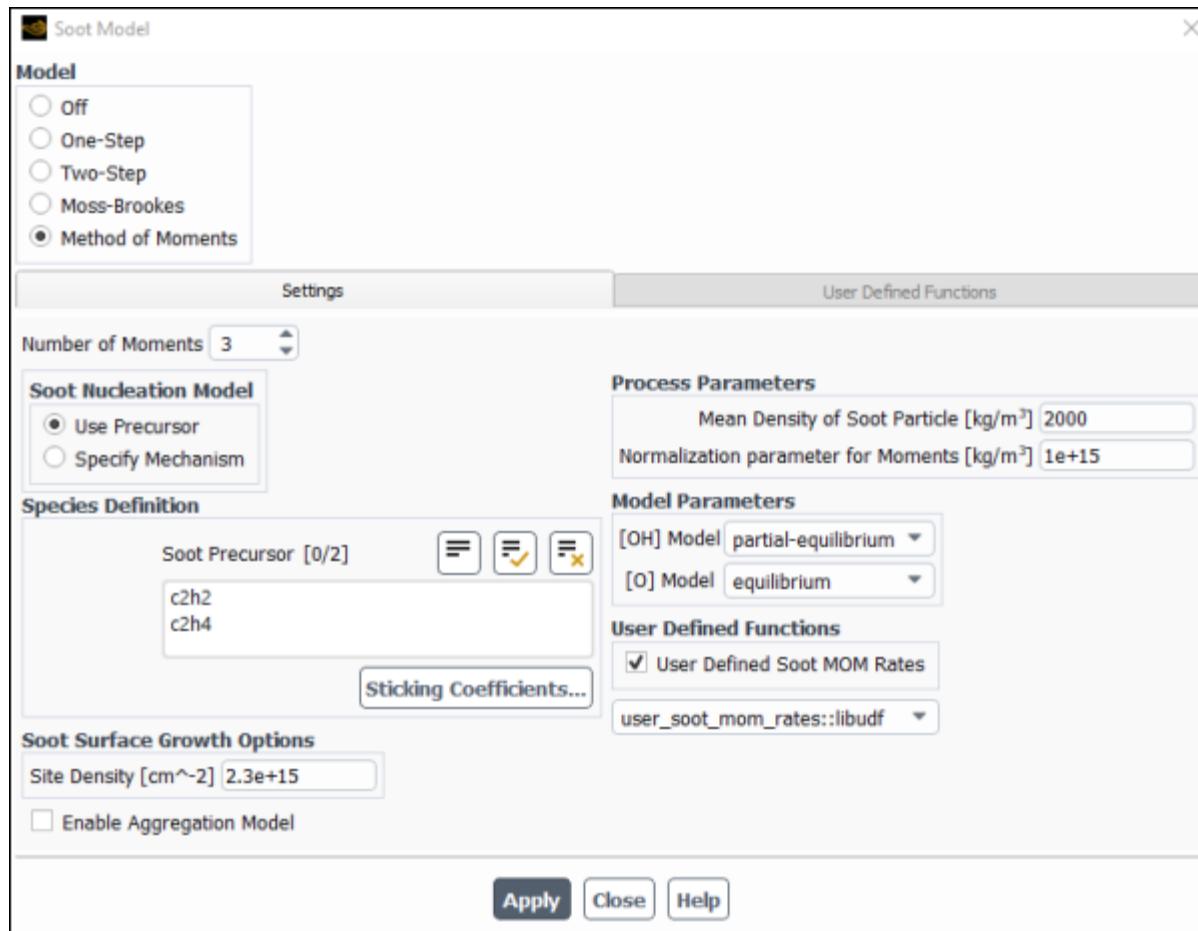
> **Note:**
>
> The UDF hook is only available for the **Method of Moments** soot model.

To hook a soot MOM rates UDF:

1. Open the **Soot Model** dialog box

   🗔 **Setup** → **Models** → **Species** → **Soot** 🖱 **Edit...**

Figure 6.61: The Soot Model Dialog Box (User-Defined Soot MOM Rates)

2. Make sure that the **Method of Moments** is selected as the **Model**.

3. Select the function name (for example, `user_soot_mom_rates::libudf`) from the **User Defined Soot MOM Rates** drop-down list.

4. Click **Apply**.

See `DEFINE_SOOT_MOM_RATES` (p. 140) for details about defining `DEFINE_SOOT_MOM_RATES` functions.

6.2.37. Hooking `DEFINE_SOOT_NUCLEATION_RATES` UDFs

After you have compiled (Compiling UDFs (p. 399)), your `DEFINE_SOOT_NUCLEATION_RATES` UDF in Ansys Fluent, the function name you supplied as a `DEFINE` macro argument will become visible and selectable from the **User Defined Soot Nuclei Rates** drop-down list under the **User Defined**

Functions tab of the **Soot Model** dialog box (Figure 6.60: The Soot Model Dialog Box (User-Defined Mass and Nucleation Rates) (p. 498)).

> **Note:**
>
> The UDF hook is only available when the **Moss-Brookes** or the **Moss-Brookes-Hall** soot model is enabled.

🔁 **Setup** → **Models** → **Species** → **Soot** 🖱 **Edit...** → **Moss-Brookes**

Under the **User Defined Functions** tab, select the function name (for example, `user_soot_nuc_rates::libudf`) from the **User Defined Soot Nuclei Rates** drop-down list, and click **Apply**.

See `DEFINE_SOOT_NUCLEATION_RATES` (p. 142) for details about defining `DEFINE_SOOT_NUC-LEATION_RATES` functions.
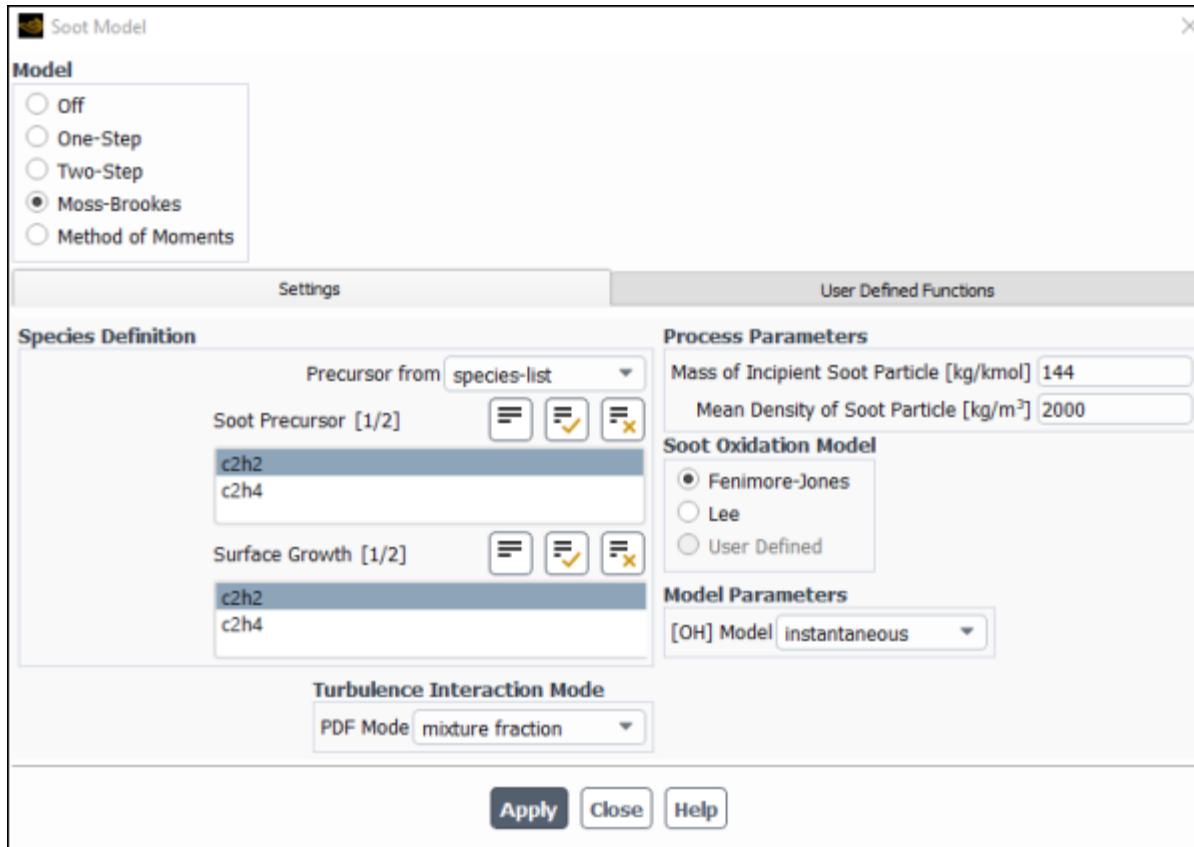
6.2.38. Hooking `DEFINE_SOOT_OXIDATION_RATE` UDFs

After you have compiled (Compiling UDFs (p. 399)), your `DEFINE_SOOT_OXIDATION_RATE` UDF in Ansys Fluent, the function name you supplied as a `DEFINE` macro argument will become visible and selectable from the **User Defined Oxidation Rate** drop-down list in the **Soot Oxidation Model** group box of the **Soot Model** dialog box (Figure 6.62: The Soot Model Dialog Box (User-Defined Oxidation Rate) (p. 501)).

> **Note:**
>
> The UDF hook is only available when the **Moss-Brookes** or **Moss-Brookes-Hall** model is enabled.

🔁 **Setup** → **Models** → **Species** → **Soot** 🖱 **Edit...** → **Moss-Brookes**

Figure 6.62: The Soot Model Dialog Box (User-Defined Oxidation Rate)

Select **User Defined** in the **Soot Oxidation Model** group box, then select the function name (for example, **user_soot_oxid_rate::libudf**) from the **User Defined Oxidation Rate** drop-down list , and click **Apply**.

See `DEFINE_SOOT_OXIDATION_RATE` (p. 145) for details about defining `DEFINE_SOOT_OXIDA-TION_RATE` functions.
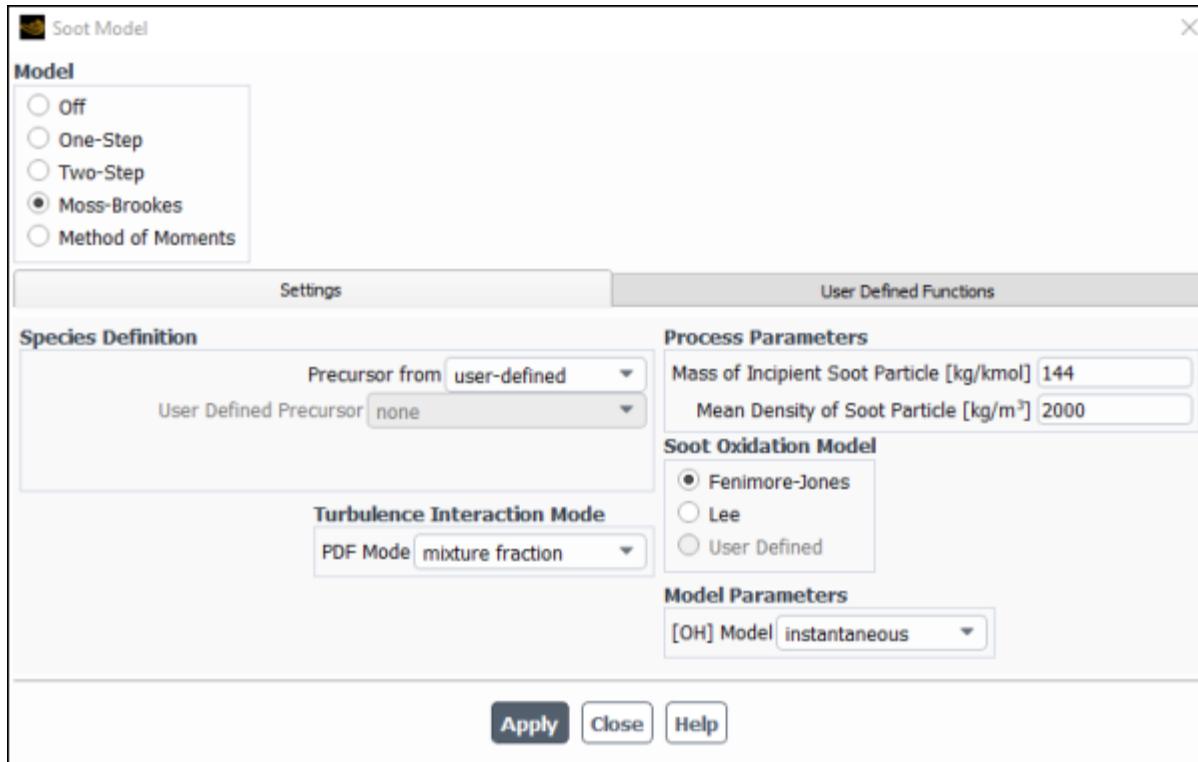
6.2.39. Hooking `DEFINE_SOOT_PRECURSOR` UDFs

After you have compiled (Compiling UDFs (p. 399)), your `DEFINE_SOOT_PRECURSOR` UDF in Ansys Fluent, the function name you supplied as a `DEFINE` macro argument will become visible and select-able from the **User Defined Precursor** drop-down list in the **Species Definition** group box of the **Soot Model** dialog box ().

Note:

The UDF is available only with **Moss-Brookes** model and visible when the **user-defined** option is selected from the **Precursor from** drop-down list in **Species Definition** group box of the **Soot Model** dialog box (Figure 6.63: The Soot Model Dialog Box (User-Defined Precursor) (p. 502)).

 Setup → **Models** → **Species** → **Soot** **Edit...** → **Moss-Brookes**

Figure 6.63: The Soot Model Dialog Box (User-Defined Precursor)

Select the function name (for example, **user_soot_prec::libudf**) from the **User Defined Precursor** drop-down list in the **Species Definition** group box, and click **Apply**.

See `DEFINE_SOOT_PRECURSOR` (p. 147) for details about defining `DEFINE_SOOT_PRECURSOR` functions.

6.2.40. Hooking `DEFINE_SPARK_GEOM` UDFs

After you have compiled (Compiling UDFs (p. 399)) your `DEFINE_SPARK_GEOM` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Set Spark Ignition** dialog box (Figure 6.64: The Set Spark Ignition Dialog Box (p. 503)) in Ansys Fluent.

To hook the UDF to Ansys Fluent, begin by opening the **Species Model** dialog box.

  Setup → **Models** → **Species** 🖱 **Edit...**
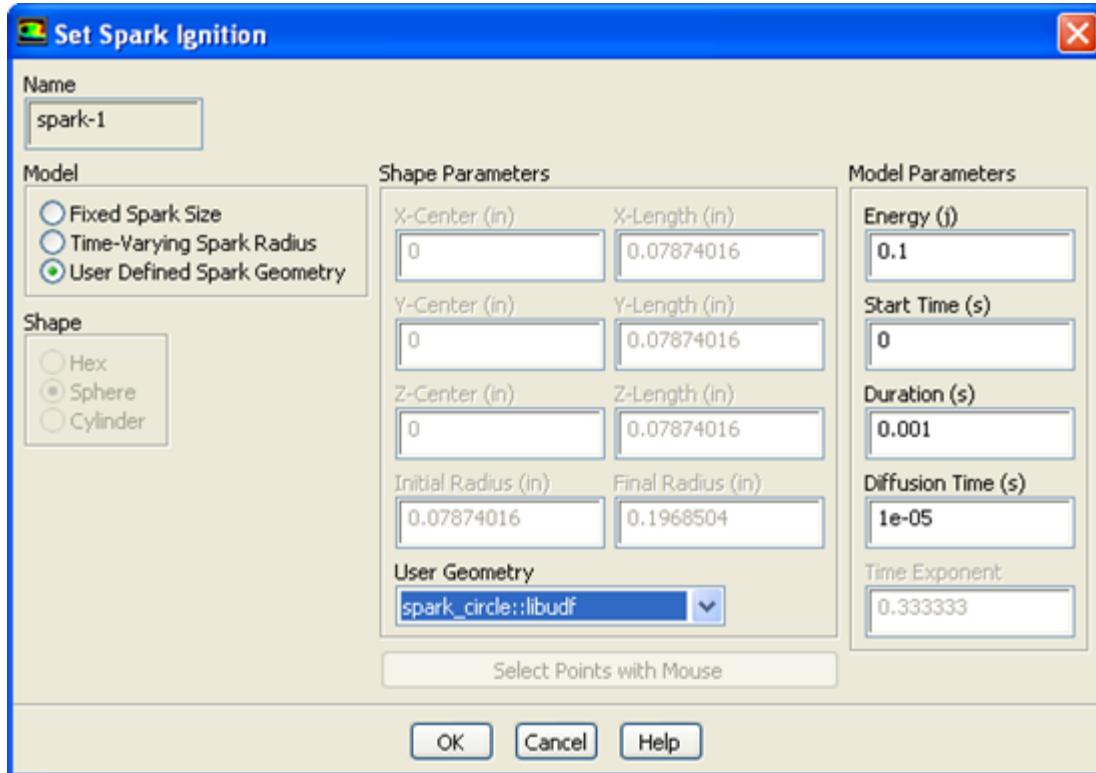
In the **Species Model** dialog box, either select **Species Transport** from the **Model** list and enable the **Volumetric** option in the **Reactions** list, or simply select **Premixed Combustion** from the **Model** list.

Next, open the **Spark Ignition** dialog box.

  Setup → **Models** → **Species** → **Spark Ignition** 🖱 **Edit...**

Make sure that **Number of Sparks** is set to a nonzero number in the **Spark Ignition** dialog box and click the **Define...** button for the spark you want to define, in order to open the **Set Spark Ignition** dialog box.

Figure 6.64: The Set Spark Ignition Dialog Box

In the **Set Spark Ignition** dialog box, select **User Defined Spark Geometry** from the **Model** list. Then select the function name (for example, **spark_circle::libudf**) from the **User Geometry** drop-down list in the **Shape Parameters** group box.

See `DEFINE_SPARK_GEOM (R14.5 spark model)` (p. 152) for details about `DEFINE_SPARK_GEOM` UDFs.
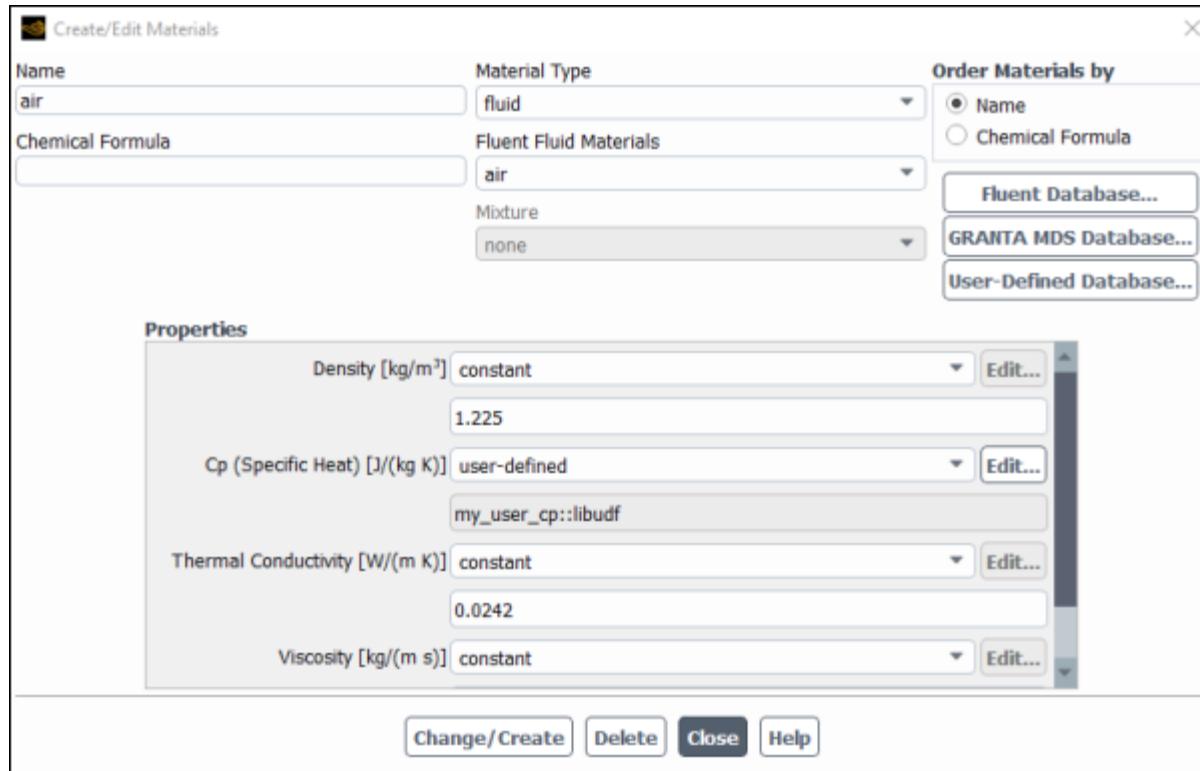
6.2.41. Hooking `DEFINE_SPECIFIC_HEAT` UDFs

After you have compiled your `DEFINE_SPECIFIC_HEAT` UDF (as described in Compiling UD-Fs (p. 399)), the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in Ansys Fluent.
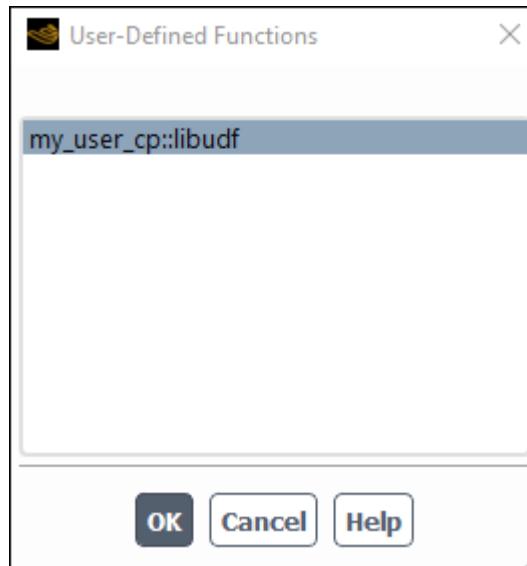
To hook the UDF to Ansys Fluent, you will first need to open the **Materials** task page.

Setup → **Materials**

Select the appropriate material from the **Material** selection list and click the **Create/Edit...** button to open the **Create/Edit Materials** dialog box (Figure 6.65: The Create/Edit Materials Dialog Box (p. 504)).

Figure 6.65: The Create/Edit Materials Dialog Box

Next, select **user-defined** from the drop-down list for **Cp** to open the **User-Defined Functions** dialog box (Figure 6.66: The User-Defined Functions Dialog Box (p. 504)). Select the name you defined in the UDF (for example, **my_user_cp::libudf**) and click **OK**. The name of the function will subsequently be displayed under the **Cp** property in the **Create/Edit Materials** dialog box.

Figure 6.66: The User-Defined Functions Dialog Box

See `DEFINE_SPECIFIC_HEAT` (p. 154) for details about defining `DEFINE_SPECIFIC_HEAT` UDFs.

6.2.42. Hooking `DEFINE_SR_RATE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_SR_RATE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.67: The User-Defined Function Hooks Dialog Box (p. 506)) in Ansys Fluent.
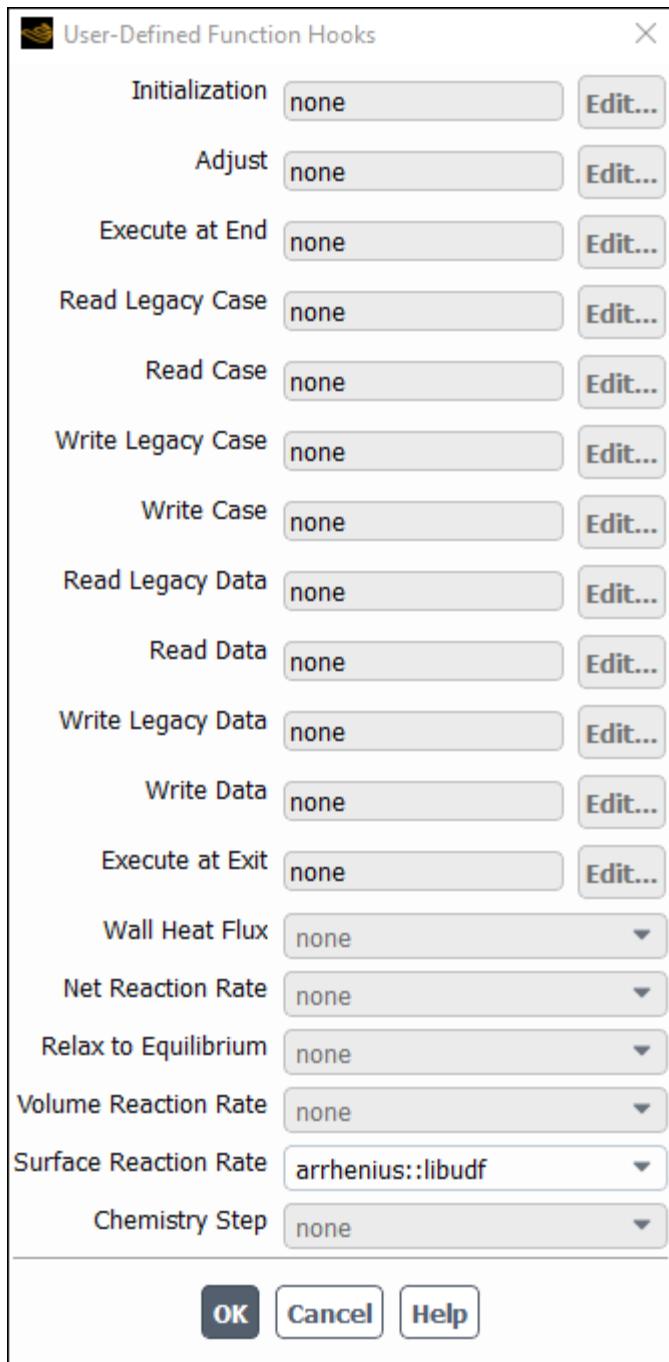
To hook the UDF to Ansys Fluent, first set up an appropriate reaction model in the **Species Model** dialog box.

Setup → **Models** → **Species** Edit...

Select **Species Transport** from the **Model** list in the **Species Model** dialog box, and enable the **Volumetric** and **Wall Surface** options in the **Reactions** group box. Make sure that you do not use the **Stiff Chemistry Solver** (as the **Chemistry Solver**), and click **OK**.

Next, open the **User-Defined Function Hooks** dialog box (Figure 6.67: The User-Defined Function Hooks Dialog Box (p. 506)).

Parameters & Customization → **User Defined Functions** Function Hooks...

Figure 6.67: The User-Defined Function Hooks Dialog Box

Select the function name (for example, **arrhenius::libudf**) in the **Surface Reaction Rate Function** drop-down list in the **User-Defined Function Hooks** dialog box, and click **OK**.

See `DEFINE_SR_RATE` (p. 156) for details about `DEFINE_SR_RATE` functions.

6.2.43. Hooking `DEFINE_THICKENED_FLAME_MODEL` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_THICKENED_FLAME_MODEL` UDF, the name of the function you supplied as a `DEFINE`

macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.67: The User-Defined Function Hooks Dialog Box (p. 506)) in Ansys Fluent.
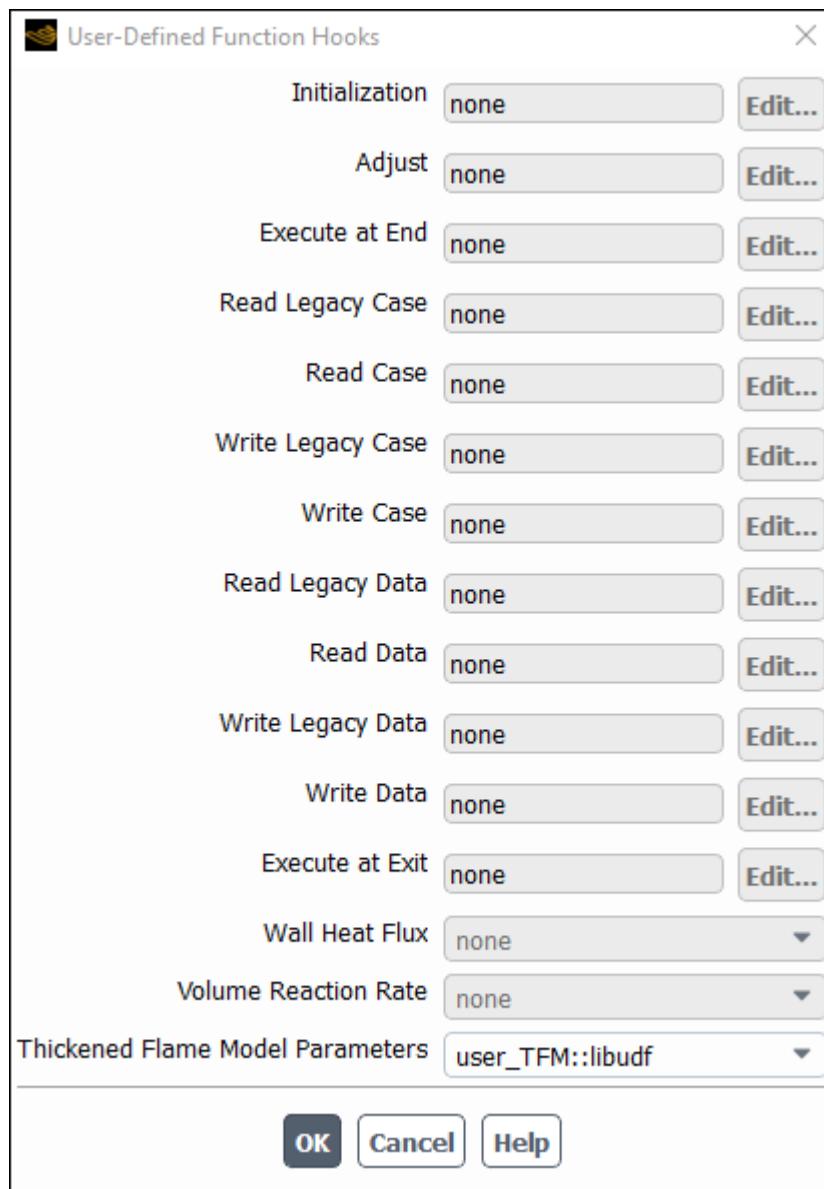
To hook the UDF to Ansys Fluent, select **Species Transport** with **Volumetric Reactions** enabled in the **Species Model** dialog box. Enable the **Thickened Flame Model**. Note that this option is only available for unsteady, laminar or LES/DES/SAS turbulent cases.

🖳 **Setup** → **Models** → **Species** 🖱 **Edit...**

Next, open the **User-Defined Function Hooks** dialog box (Figure 6.68: The User-Defined Function Hooks Dialog Box (p. 507)).

🖳 **Parameters & Customization** → **User Defined Functions** 🖱 **Function Hooks...**

Figure 6.68: The User-Defined Function Hooks Dialog Box
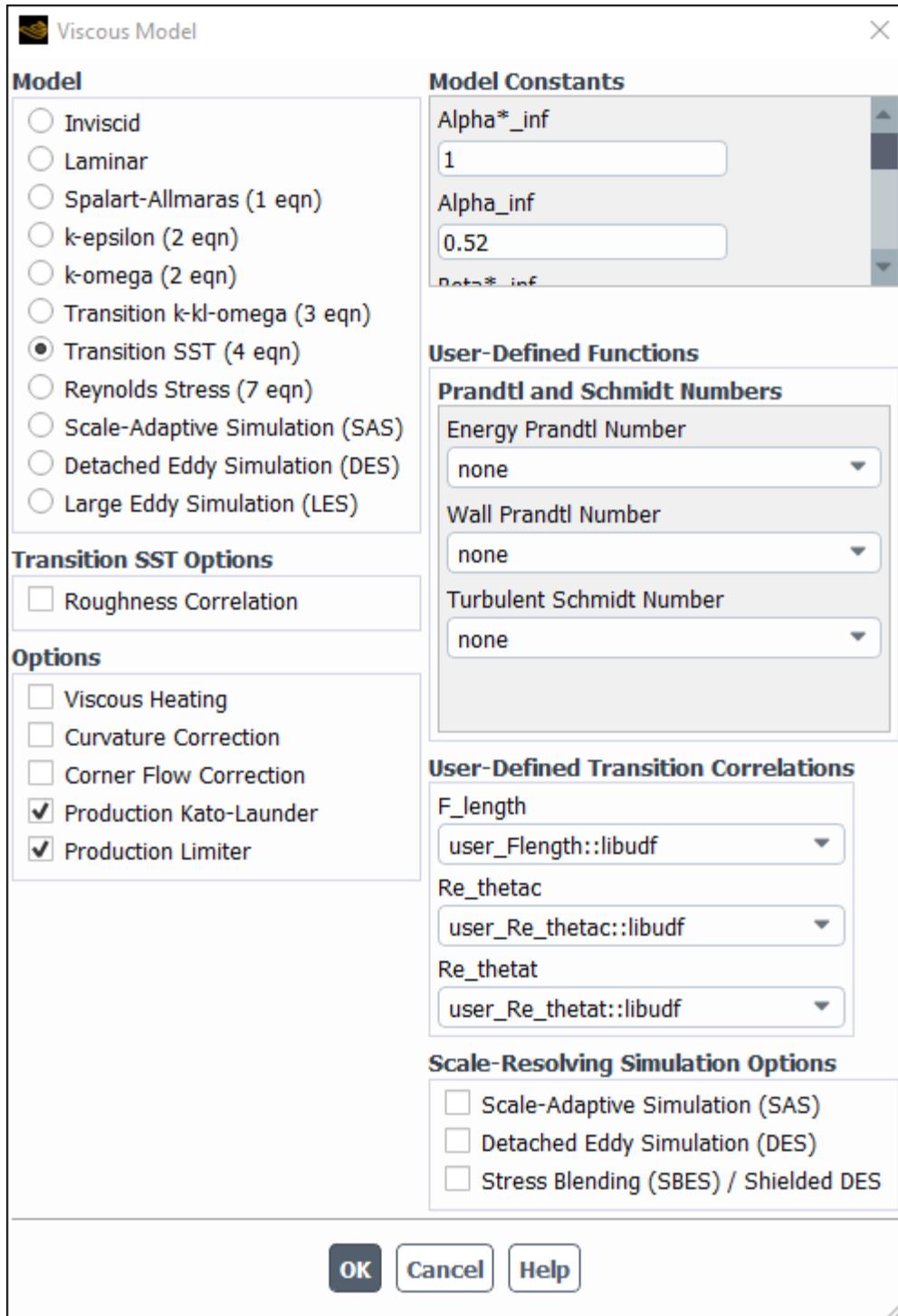
Select the function name (for example, **user_TFM::libudf**) in the **Thickened Flame Model Parameters** drop-down list in the **User-Defined Function Hooks** dialog box, and click **OK**.

See `DEFINE_THICKENED_FLAME_MODEL` (p. 158) for details about `DEFINE_THICKENED_FLAME_MODEL` functions.

6.2.44. Hooking `DEFINE_TRANS` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_TRANS` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. To hook the UDF, select **Transition SST** from the **Model** list in the **Viscous Model** dialog box (Figure 6.69: The Viscous Model Dialog Box (p. 509)).

Setup → Models → Viscous 🖱 **Edit...**

Figure 6.69: The Viscous Model Dialog Box
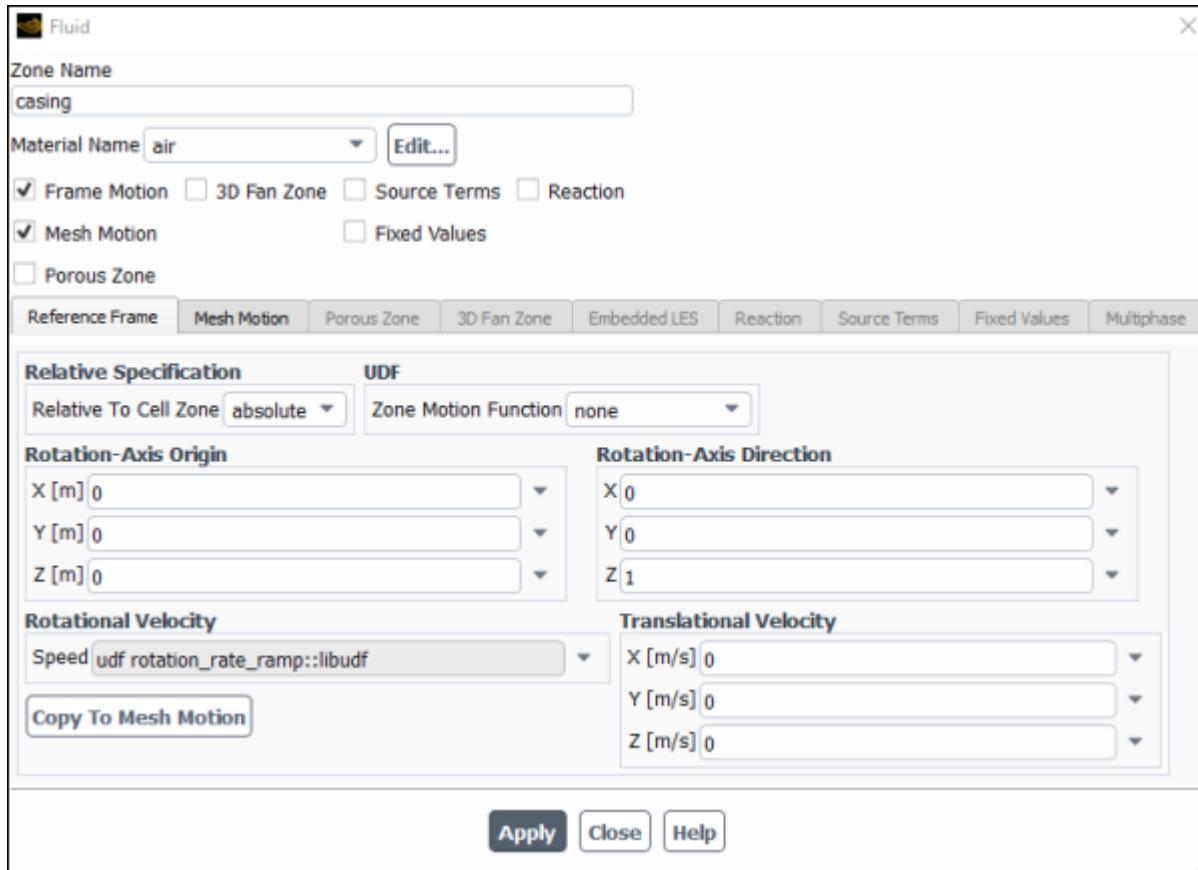
Next, select the function name (for example, **user_Flength::libudf**) from a drop-down list in the **User-Defined Transition Correlations** group box (for example, **Flength**), and click **OK**.

See `DEFINE_TRANS` UDFs (p. 160) for details about `DEFINE_TRANS` functions.

6.2.45. Hooking `DEFINE_TRANSIENT_PROFILE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_TRANSIENT_PROFILE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Fluid** or **Solid** dialog box (Figure 6.70: The Fluid Dialog Box (p. 510)) in Ansys Fluent, under the **Reference Frame** tab, the **Mesh Motion** tab, and the **Solid Motion** tab if the **Frame Motion**, **Mesh Motion**, and **Solid Motion** options are enabled, respectively.

Figure 6.70: The Fluid Dialog Box

Select the function name from a drop-down list in the **Rotational Velocity** or **Rotational Velocity** group boxes in the **Fluid** or **Solid** dialog box, and click **OK**.

See `DEFINE_TRANSIENT_PROFILE` (p. 164) for details about `DEFINE_TRANSIENT_PROFILE` functions.
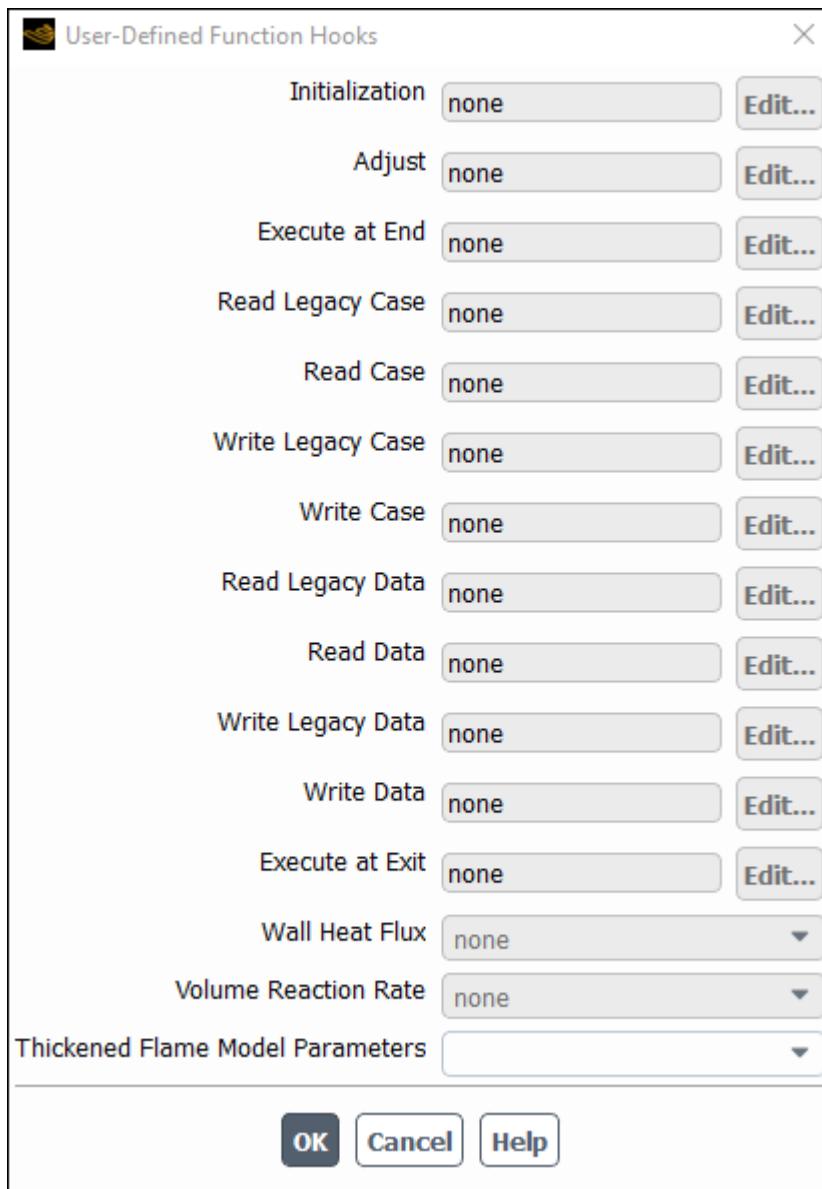
6.2.46. Hooking `DEFINE_TURB_PREMIX_SOURCE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_TURB_PREMIX_SOURCE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.71: The User-Defined Function Hooks Dialog Box (p. 511)) in Ansys Fluent.

To hook the UDF to Ansys Fluent, open the **User-Defined Function Hooks** dialog box (Figure 6.71: The User-Defined Function Hooks Dialog Box (p. 511)).

⊞ Parameters & Customization → User Defined Functions 🖰 Function Hooks...

Figure 6.71: The User-Defined Function Hooks Dialog Box

Important:

You must have a premixed combustion model enabled in the **Species Model** dialog box.

Select the function name (for example, **turb_flame_src::libudf**) in the **Turbulent Premixed Source Function** drop-down list in the **User-Defined Function Hooks** dialog box, and click **OK**.

See DEFINE_TURB_PREMIX_SOURCE (p. 166) for details about DEFINE_TURB_PREMIX_SOURCE functions.
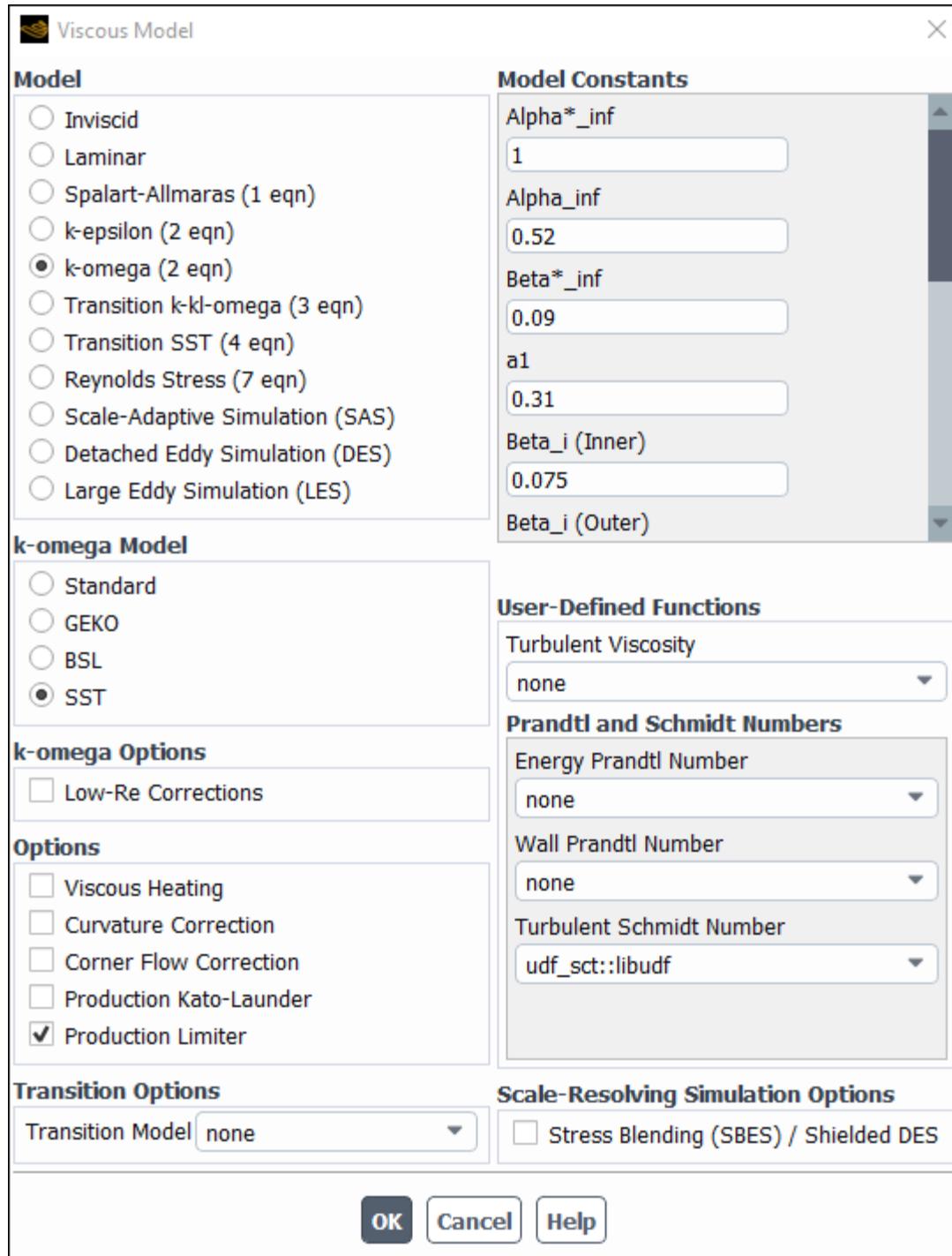
6.2.47. Hooking `DEFINE_TURB_SCHMIDT` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_TURB_SCHMIDT` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Viscous Model** dialog box in Ansys Fluent. To hook the UDF, first open the **Viscous Model** dialog box (Figure 6.72: The Viscous Model Dialog Box (p. 513)) and set up a turbulence model.

> **Important:**
>
> If you select **k-epsilon** from the **Model** list, you must not select **RNG** from the **k-epsilon Model** list.

 Setup → Models → Viscous Edit...

Figure 6.72: The Viscous Model Dialog Box

Next, select the function name (for example, **udf_sct::libudf**) from the **Turbulent Schmidt Number** drop-down list under **User-Defined Functions** in the **Viscous Model** dialog box, and click **OK**.

Important:
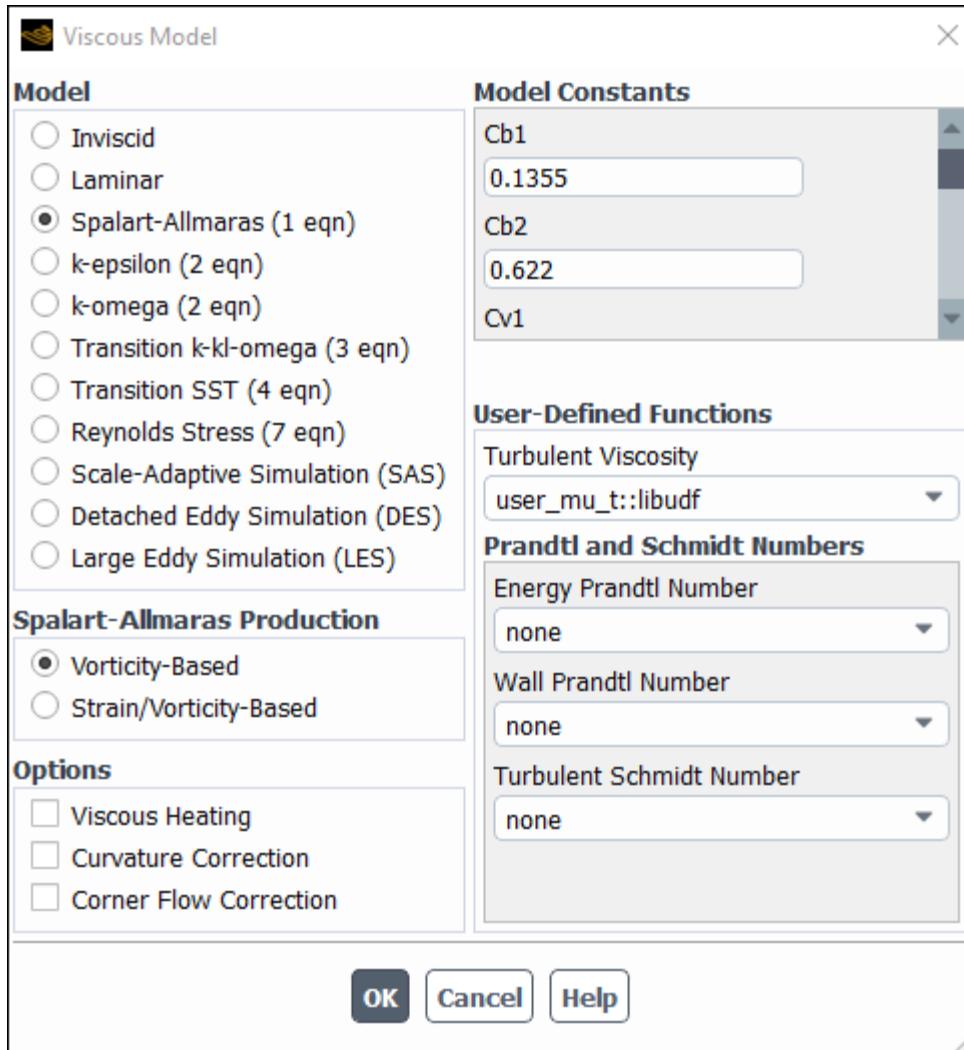
The **Species Transport** model must be selected in the **Species Model** dialog box for the **Turbulent Schmidt Number** drop-down list to be visible in the **Viscous Model** dialog box.

See `DEFINE_TURB_SCHMIDT` UDF (p. 167) for details about `DEFINE_TURB_SCHMIDT` functions.

6.2.48. Hooking `DEFINE_TURBULENT_VISCOSITY` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_TURBULENT_VISCOSITY` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Viscous Model** dialog box (Figure 6.73: The Viscous Model Dialog Box (p. 515)) in Ansys Fluent.

⌸ **Setup** → **Models** → **Viscous** 🖱 **Edit...**

Figure 6.73: The Viscous Model Dialog Box

To hook the UDF to Ansys Fluent, select the function name (for example, **user_mu_t::libudf**) from the **Turbulent Viscosity** drop-down list under **User-Defined Functions** in the **Viscous Model** dialog box, and click **OK**.

See `DEFINE_TURBULENT_VISCOSITY` (p. 168) for details about `DEFINE_TURBULENT_VISCOSITY` functions.

6.2.49. Hooking `DEFINE_VR_RATE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_VR_RATE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.74: The **User-Defined Function Hooks** Dialog Box (p. 516)) in Ansys Fluent.

To hook the UDF to Ansys Fluent, first set up an appropriate reaction model in the **Species Model** dialog box.

 Setup → **Models** → **Species** **Edit...**

Select **Species Transport** from the **Model** list in the **Species Model** dialog box, and enable the **Volumetric** option in the **Reactions** group box. Make sure that **Chemkin-CFD Solver** is not selected as the **Chemistry Solver**, and click **OK**.

Next, open the **User-Defined Function Hooks** dialog box (Figure 6.74: The **User-Defined Function Hooks** Dialog Box (p. 516)).

🗔 **Parameters & Customization** → **User Defined Functions** 🖱 **Function Hooks...**

Figure 6.74: The User-Defined Function Hooks Dialog Box
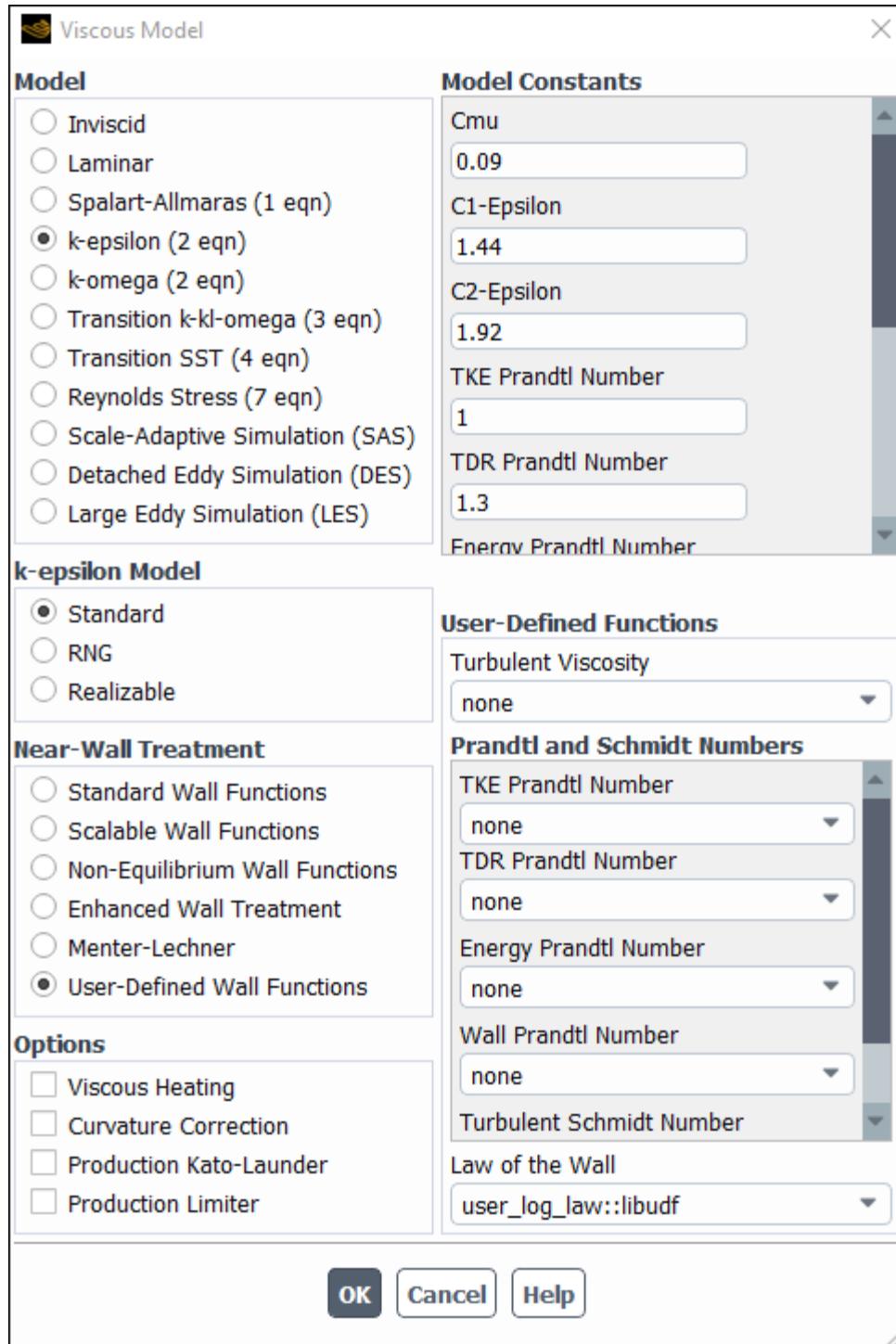
Select the function name (for example, **vol_reac_rate::libudf**) in the **Volume Reaction Rate Function** drop-down list in the **User-Defined Function Hooks** dialog box, and click **OK**.

See `DEFINE_VR_RATE` (p. 170) for details about `DEFINE_VR_RATE` functions.

6.2.50. Hooking `DEFINE_WALL_FUNCTIONS` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_WALL_FUNCTIONS` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Viscous Model** dialog box () in Ansys Fluent.

Setup → **Models** → **Viscous** **Edit...**

Figure 6.75: The Viscous Model Dialog Box

To hook the UDF, select **k-epsilon** from the **Model** list in the **Viscous Model** dialog box, and select **User-Defined Wall Functions** from the **Near-Wall Treatment** list. Then, select the function name (for example, **user_log_law::libudf**) from the **Law of the Wall** drop-down list, and click **OK**.

See DEFINE_WALL_FUNCTIONS (p. 173) for details about DEFINE_WALL_FUNCTIONS functions in Ansys Fluent.

6.2.51. Hooking `DEFINE_SOURCE_FE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_SOURCE_FE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable when defining displacement source terms for solid cell zones in Ansys Fluent.

To hook the UDF to Ansys Fluent, first select the **Linear Elasticity** model in the **Structural Model** dialog box.
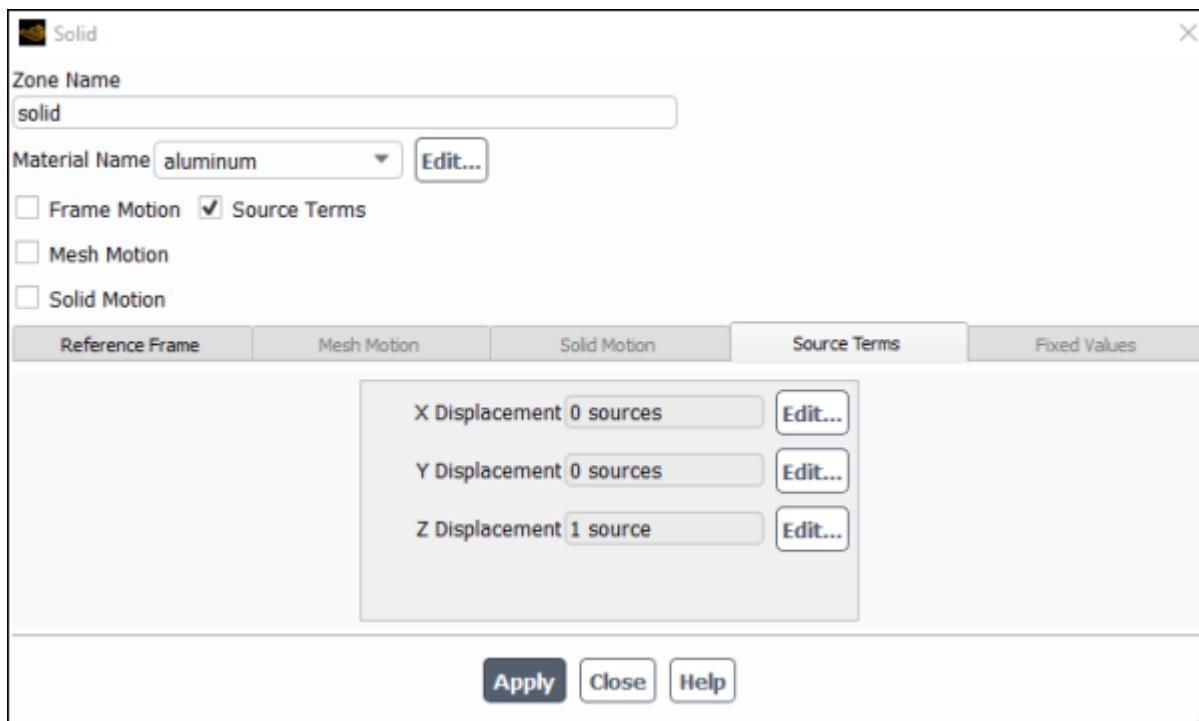
 Setup → Models → Structure **Edit...**

Next, open the **Cell Zones** task page.
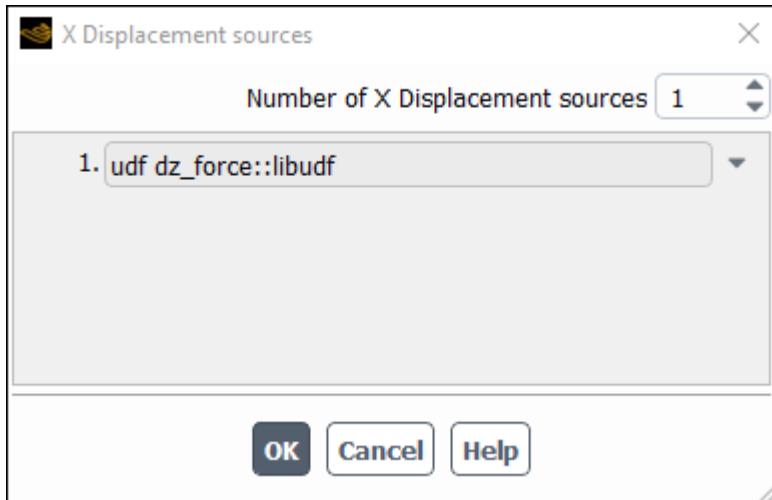
 Setup → Cell Zones

Select a solid zone in the **Zone** list and click **Edit...** to open the **Solid** dialog box (Figure 6.76: The Solid Dialog Box (p. 519)).

Figure 6.76: The Solid Dialog Box

In the **Source Terms** tab, select an appropriate directional displacement (such as **Z Displacement**) and click **Edit...** to open, for example, the **Z Displacement source** dialog box.

Figure 6.77: The Z Displacement source Dialog Box
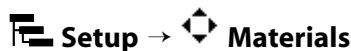
Increase the **Number of Z Displacement sources** and hook the UDF by, for example selecting the function name (**udf dz_force**) from the drop-down list.

See `DEFINE_SOURCE_FE` (p. 176) for details about `DEFINE_SOURCE_FE` function in Ansys Fluent.
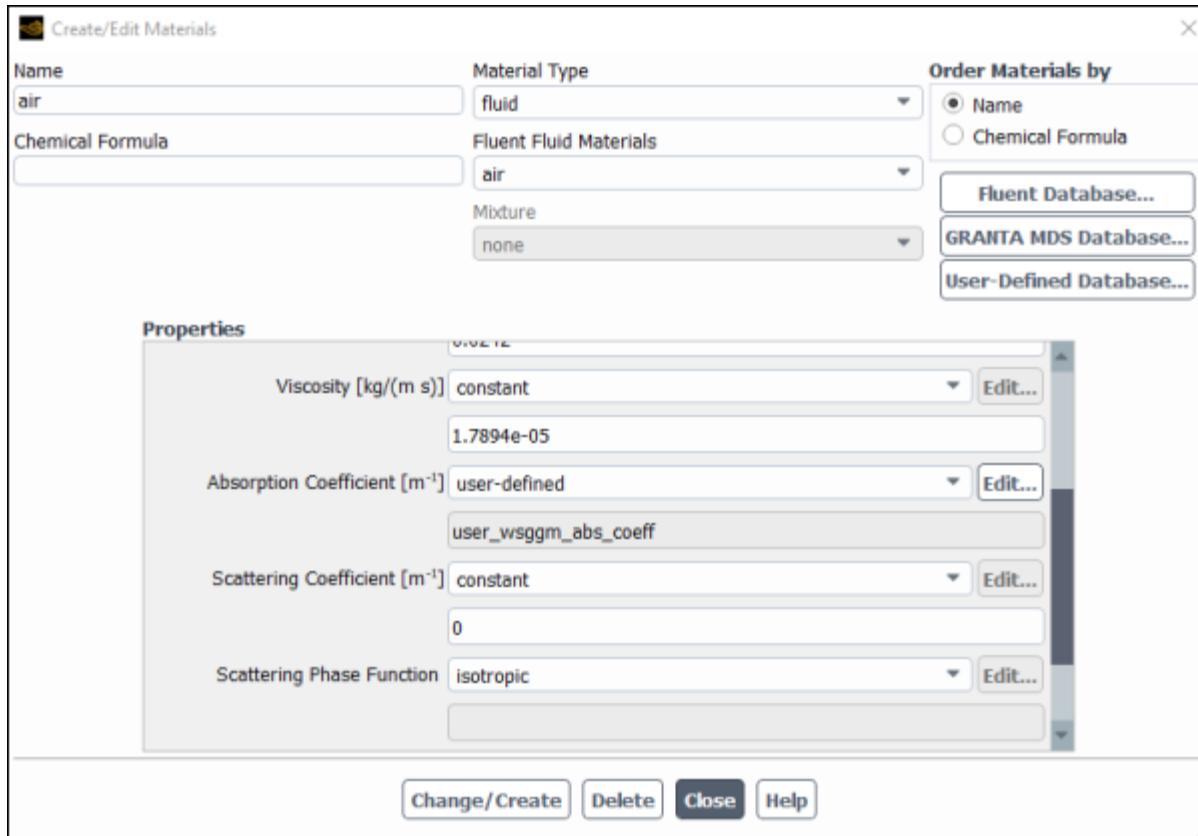
6.2.52. Hooking `DEFINE_WSGGM_ABS_COEFF` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_WSGGM_ABS_COEFF` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Create/Edit Materials** dialog box in Ansys Fluent.
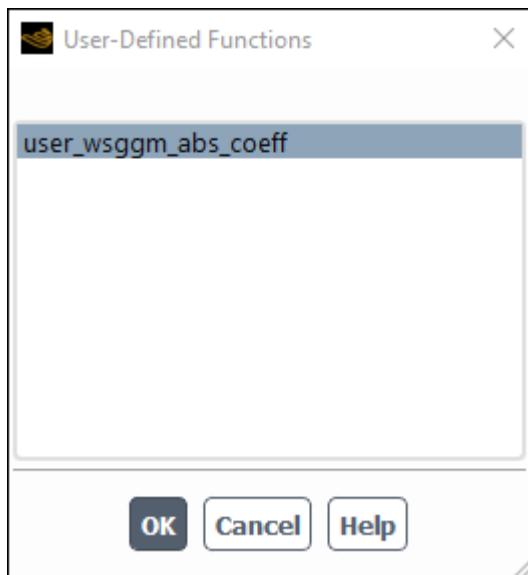
To hook the UDF to Ansys Fluent, first make sure that you have selected a radiation model from the **Model** list of the **Radiation Model** dialog box and have set up a species transport or soot formation model. Then open the **Materials** task page.

 Setup → ⬦ **Materials**

Select the mixture in the **Materials** list and click the **Create/Edit...** button to open the **Create/Edit Materials** dialog box (Figure 6.78: The Create/Edit Materials Dialog Box (p. 521)).

Figure 6.78: The Create/Edit Materials Dialog Box

Next, select **user-defined-wsggm** from the **Absorption Coefficient** drop-down list in the **Properties** list, which opens the **User-Defined Functions** dialog box (Figure 6.79: The User-Defined Functions Dialog Box (p. 521)).

Figure 6.79: The User-Defined Functions Dialog Box

Select the function name (for example, **user_wsggm_abs_coeff::libudf**) from the list of UDFs displayed in the **User-Defined Functions** dialog box and click **OK**. The function name will then be displayed

in a field under the **Absorption Coefficient** drop-down list in the **Create/Edit Materials** dialog box. Finally, click **Change/Create** in the **Create/Edit Materials** dialog box to save the settings.

See `DEFINE_WSGGM_ABS_COEFF` (p. 177) for details about `DEFINE_WSGGM_ABS_COEFF` functions, and Inputs for a Composition-Dependent Absorption Coefficient in the User's Guide for further information about inputs for composition-dependent absorption coefficients.

6.3. Hooking Multiphase UDFs

This section contains methods for hooking UDFs to Ansys Fluent that have been defined using `DEFINE` macros (described in Multiphase `DEFINE` Macros (p. 179)), and interpreted or compiled using methods (described in Interpreting UDFs (p. 393) or Compiling UDFs (p. 399)), respectively.

For more information, see the following sections:

6.3.1. Hooking `DEFINE_BOILING_PROPERTY` UDFs
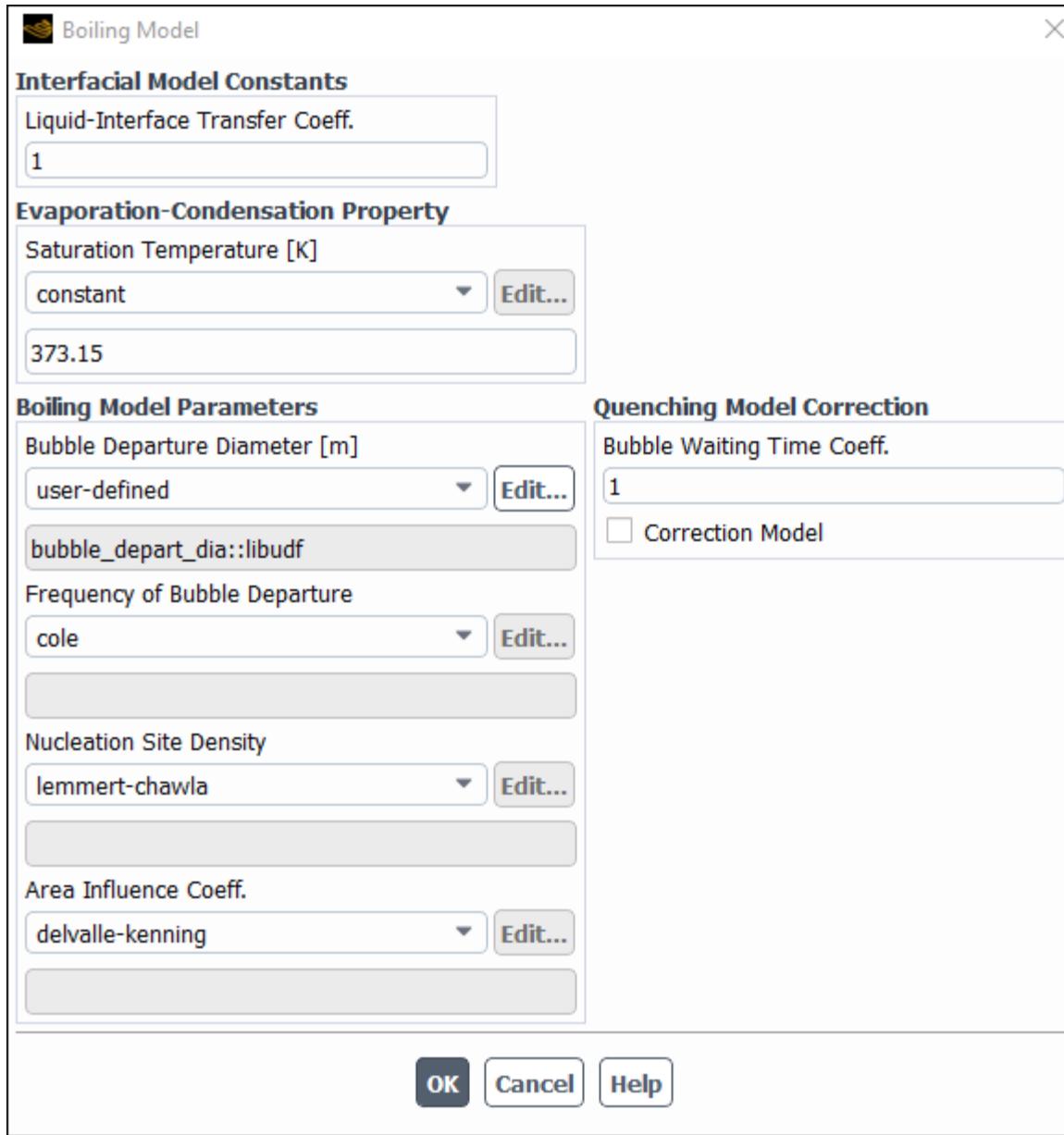
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_BOILING_PROPERTY` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Boiling Model** dialog box (Figure 6.80: The Boiling Model Dialog Box (p. 523)) in Ansys Fluent. Note that this type of UDF can be applied only to the Eulerian multiphase boiling model.

To hook the UDF to Ansys Fluent:

1. In the **Multiphase Model** dialog box, select the **Eulerian** model (**Models** tab).

    📇 **Setup → Models → Multiphase** 🖱 **Edit...**

2. In the **Eulerian Parameters** group box, select **Boiling Model**.

3. Click **Apply**.

4. In the **Phase Interaction** > **Heat, Mass, Reactions** > **Mass** tab, set the **Number of Mass Transfer Mechanisms** and select **boiling** from the **Mechanism** drop-down list. The **Boiling Model** dialog box will open where you can hook your boiling parameter UDFs, as shown in Figure 6.80: The Boiling Model Dialog Box (p. 523).

Figure 6.80: The Boiling Model Dialog Box

5. Click **Apply**.

See `DEFINE_BOILING_PROPERTY` (p. 181) for details about `DEFINE_BOILING_PROPERTY` functions.

6.3.2. Hooking `DEFINE_CAVITATION_RATE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_CAVITATION_RATE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Function Hooks** dialog box (Figure 6.82: The User-Defined Function Hooks Dialog Box (p. 525)) in Ansys Fluent. Note that cavitation rate UDFs can be applied only to the mixture multiphase model.
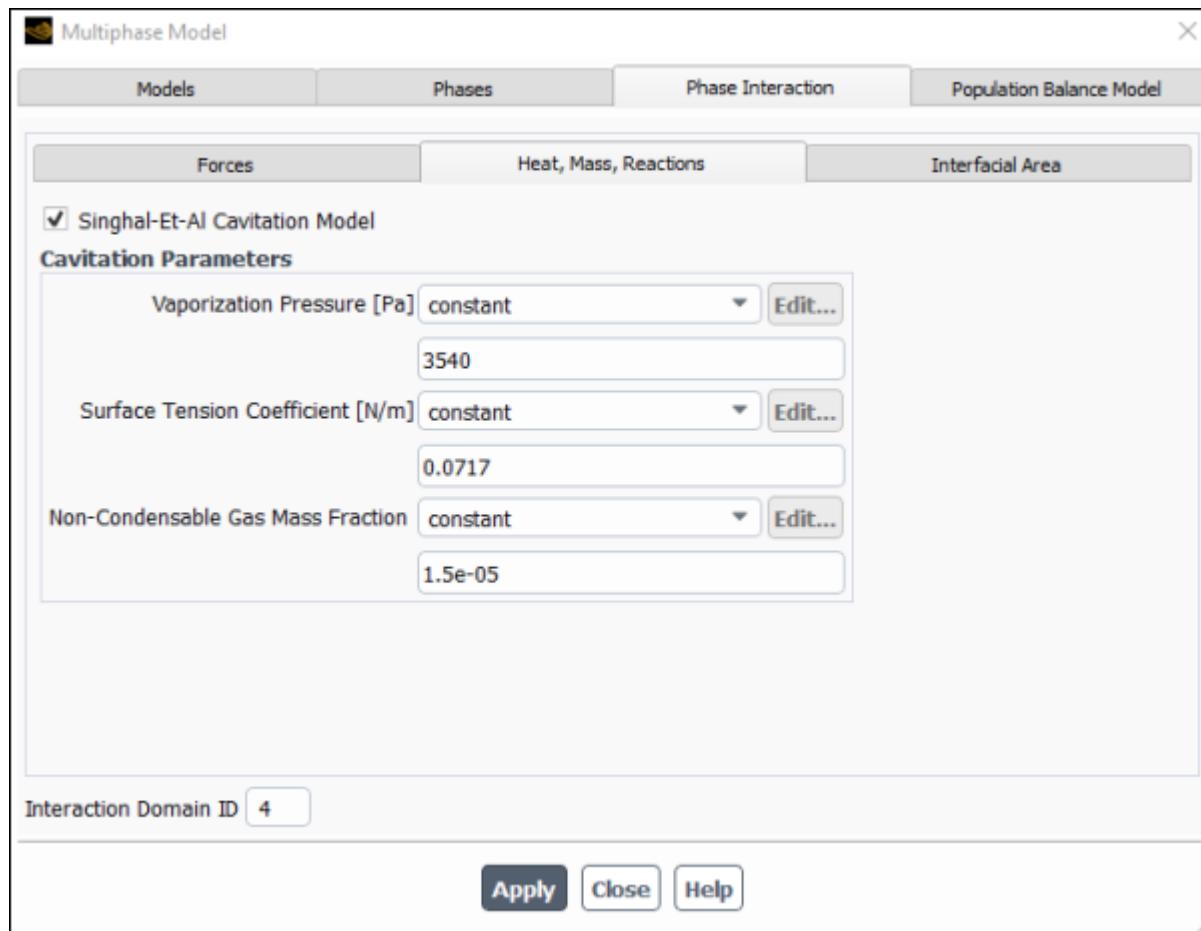
To hook the UDF to Ansys Fluent:

1. In the **Multiphase Model** dialog box, select the **Mixture** model (**Models** tab).

   🖱 **Setup** → **Models** → **Multiphase** 🖱 **Edit...**
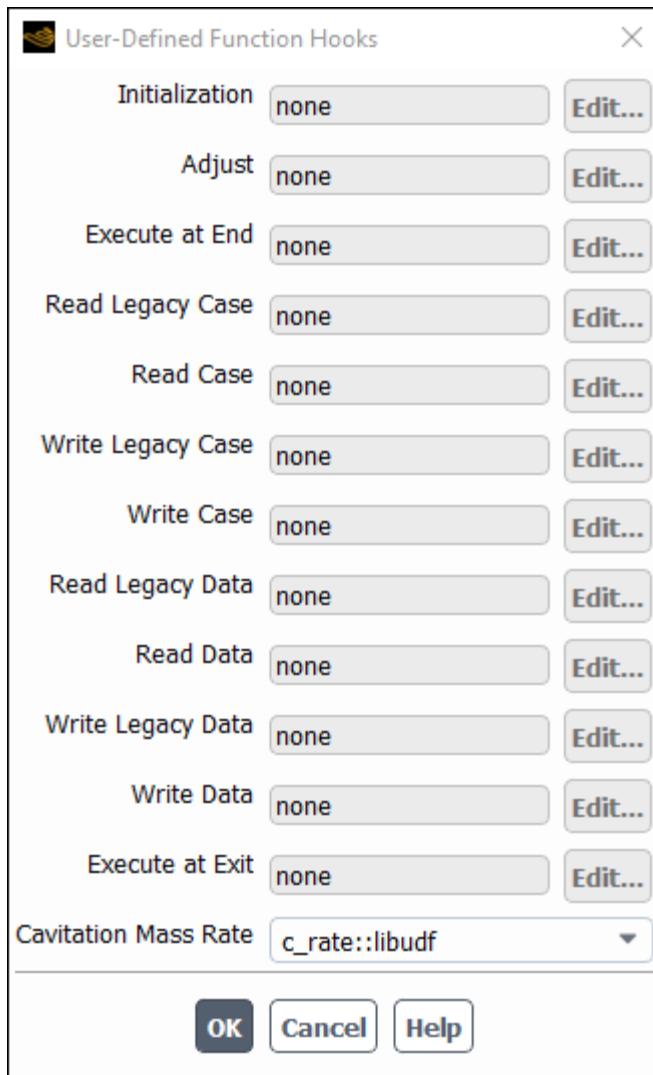
2. Click **Apply**.

3. Enable the following text command: `solve/set/advanced/singhal-et-al-cavitation-model`.

4. In the **Phase Interaction** > **Heat, Mass, Reactions** tab of the **Multiphase Model** dialog box, enable **Singhal-Et-Al Cavitation Model** (Figure 6.81: The Multiphase Model Dialog Box - Heat, Mass, Reactions Tab (p. 524)), and click **OK**.

   Figure 6.81: The Multiphase Model Dialog Box - Heat, Mass, Reactions Tab

5. Open the **User-Defined Function Hooks** dialog box (Figure 6.82: The User-Defined Function Hooks Dialog Box (p. 525)).

   🖱 **Parameters & Customization** → **User Defined Functions** 🖱 **Function Hooks...**

Figure 6.82: The User-Defined Function Hooks Dialog Box

6. To hook the UDF to Ansys Fluent, select the function name (for example, **c_rate::libudf**) in the **Cavitation Mass Rate** drop-down list (Figure 6.82: The User-Defined Function Hooks Dialog Box (p. 525)), and click **OK**.

7. Click **Apply** and close the **Multiphase Model** dialog box.

See `DEFINE_CAVITATION_RATE` (p. 183) for details about `DEFINE_CAVITATION_RATE` functions.

6.3.3. Hooking `DEFINE_EXCHANGE_PROPERTY` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_EXCHANGE_RATE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in Ansys Fluent.

To hook an exchange property UDF to Ansys Fluent:

1. Open the **Multiphase Model** dialog box.
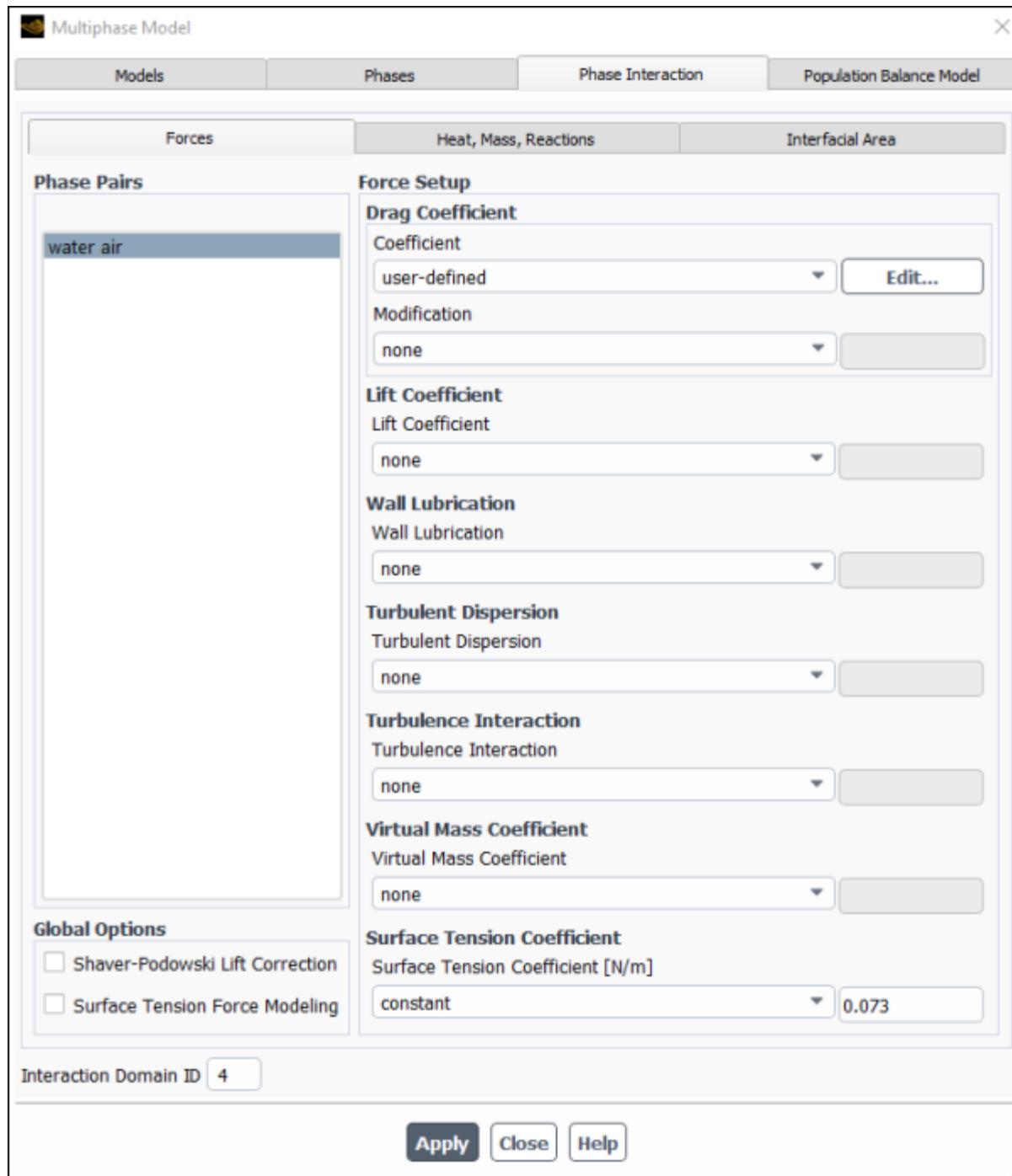
 Setup → **Models** → **Multiphase** **Edit...**

Customized mass transfer UDFs can be applied to VOF, Mixture, and Eulerian multiphase models. Drag coefficient UDFs can be applied to Mixture and Eulerian models, while heat transfer and lift coefficient UDFs can be applied only to the Eulerian model. Select the appropriate model from the **Model** list in the **Multiphase Model** dialog box.
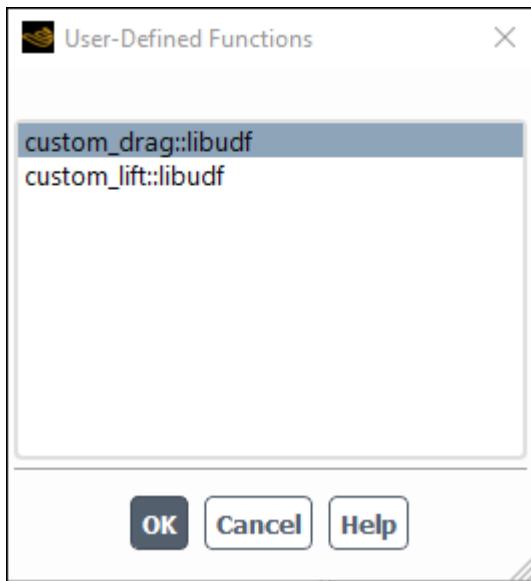
Important:

For the **Mixture** model, make sure that you enable **Slip Velocity** in the **Mixture Parameters** group box (**Models** tab) in order to display the drag coefficient in the **Forces** tab of the **Multiphase Model** dialog box.

---

2. Click **Apply**.

3. In the **Phase Interaction** > **Forces** tab, select **user-defined** from the drop-down list for the corresponding exchange property (for example, for **Coefficient** (**Drag Coefficient** group box)).

Figure 6.83: The Multiphase Model Dialog Box - Forces Tab

4. In the **User-Defined Functions** dialog box that opens, select the function name (for example, **custom_drag::libudf**) from the list of UDFs (Figure 6.84: The User-Defined Functions Dialog Box (p. 528)) and click **OK**.

Figure 6.84: The User-Defined Functions Dialog Box

5. Click **Apply** and close the **Multiphase Model** dialog box.

See `DEFINE_EXCHANGE_PROPERTY` (p. 185) for details about `DEFINE_EXCHANGE_PROPERTY` functions.

6.3.4. Hooking `DEFINE_HET_RXN_RATE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_HET_RXN_RATE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in Ansys Fluent.

To hook the UDF:
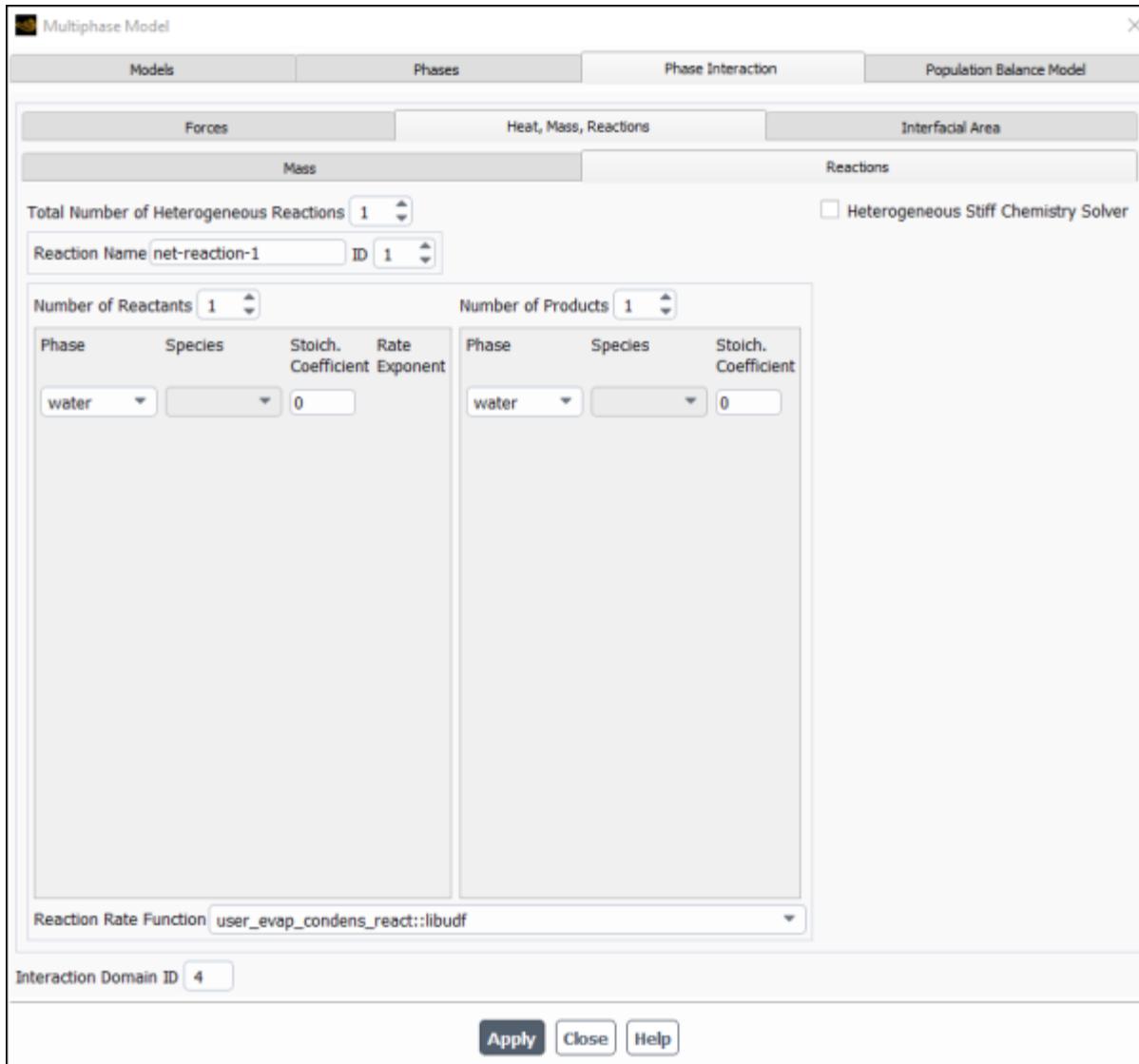
1. Select **Species Transport** from the **Model** list of the **Species Model** dialog box.

   Setup → **Models** → **Species** Edit...

2. In the **Multiphase Model** dialog box, make a selection from the **Model** list.

   Setup → **Models** → **Multiphase** Edit...

3. Click **Apply**.

4. Go to the **Phase Interaction** > **Heat, Mass, Reactions** > **Reactions** tab (Figure 6.85: The Multiphase Model Dialog Box - Reactions Tab (p. 529)).

Figure 6.85: The Multiphase Model Dialog Box - Reactions Tab

5. Enter a nonzero number in the **Total Number of Heterogeneous Reactions** field.

6. Select the function name (for example, **user_evap_condens_react::libudf**) from the **Reaction Rate Function** drop-down list.

7. Click **Apply** and close the **Multiphase Model** dialog box.
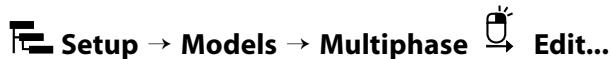
See `DEFINE_HET_RXN_RATE` (p. 190) for details about `DEFINE_HET_RXN_RATE` functions.

6.3.5. Hooking `DEFINE_LINEARIZED_MASS_TRANSFER` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_LINEARIZED_MASS_TRANSFER` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Phase Interaction** tab of the **Multiphase Model** dialog box.

To hook the UDF to Ansys Fluent:

1. Make a selection from the **Model** list of the **Multiphase Model** dialog box.

   🗁 **Setup** → **Models** → **Multiphase** 🖱 **Edit...**

2. Click **Apply**.

3. Go to the **Phase Interaction** > **Heat, Mass, Reactions** tab (Figure 6.86: The Multiphase Model Dialog Box - Heat, Mass, Reactions Tab (p. 531)).

4. Specify the **Number of Mass Transfer Mechanisms** greater than 0.

   The **Mechanism** drop-down list will appear.

5. Select **user-defined** from the **Mechanism** drop-down list.

6. In the **User-Defined Functions** dialog box that opens (Figure 6.87: The User-Defined Functions Dialog Box (p. 532)), select the function name and click **OK**.

   The UDF name will appear in the field below the **Mechanism** drop-down list in the **Heat, Mass, Reactions** tab of the **Multiphase Model** dialog box.

7. Click **Apply** and close the **Multiphase Model** dialog box.

See `DEFINE_LINEARIZED_MASS_TRANSFER` (p. 193) for details about `DEFINE_LINEARIZED_MASS_TRANSFER` functions.

6.3.6. Hooking `DEFINE_MASS_TRANSFER` UDFs
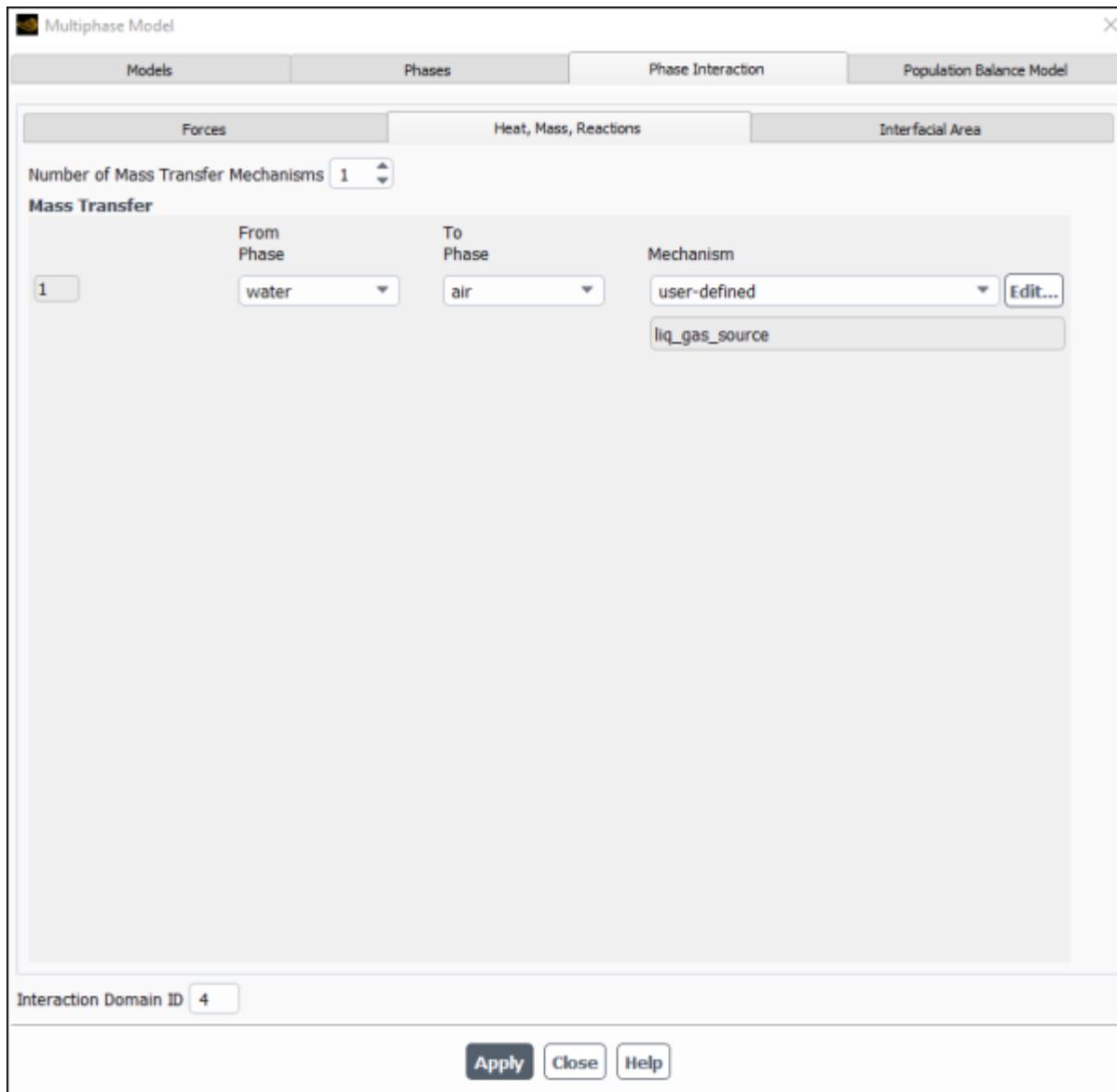
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_MASS_TRANSFER` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Phase Interaction** > **Heat, Mass, Reactions** tab of the **Multiphase Model** dialog box (Figure 6.86: The Multiphase Model Dialog Box - Heat, Mass, Reactions Tab (p. 531)).

To hook the UDF to Ansys Fluent:
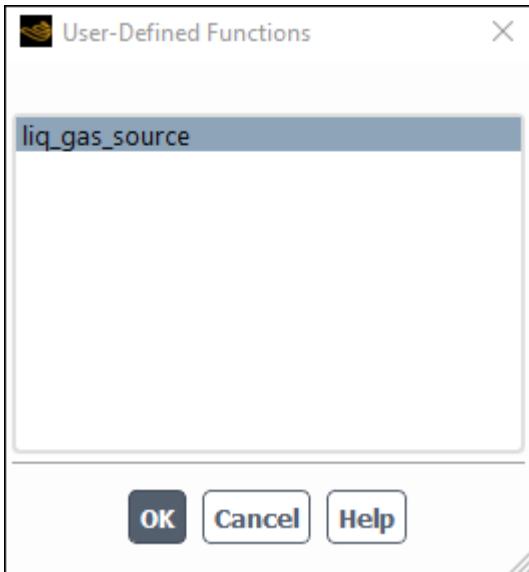
1. Make a selection from the **Model** list of the **Multiphase Model** dialog box.

   🗁 **Setup** → **Models** → **Multiphase** 🖱 **Edit...**

2. Click **Apply**.

3. Go to the **Phase Interaction** > **Heat, Mass, Reactions** tab (Figure 6.86: The Multiphase Model Dialog Box - Heat, Mass, Reactions Tab (p. 531)).

Figure 6.86: The Multiphase Model Dialog Box - Heat, Mass, Reactions Tab

4. Specify the **Number of Mass Transfer Mechanisms** greater than 0.

5. Select **user-defined** from the **Mechanism** drop-down list to open the **User-Defined Functions** dialog box (Figure 6.87: The User-Defined Functions Dialog Box (p. 532)).

Figure 6.87: The User-Defined Functions Dialog Box

6. Select the function name (for example, **liq_gas_source::udf**) and click **OK**.

   The UDF name will appear in the field below the **Mechanism** drop-down list in the **Heat, Mass, Reactions** tab of the **Multiphase Model** dialog box.

7. Click **Apply** and close the **Multiphase Model** dialog box.

See `DEFINE_MASS_TRANSFER` (p. 196) for details about `DEFINE_MASS_TRANSFER` functions.

6.3.7. Hooking `DEFINE_VECTOR_EXCHANGE_PROPERTY` UDFs
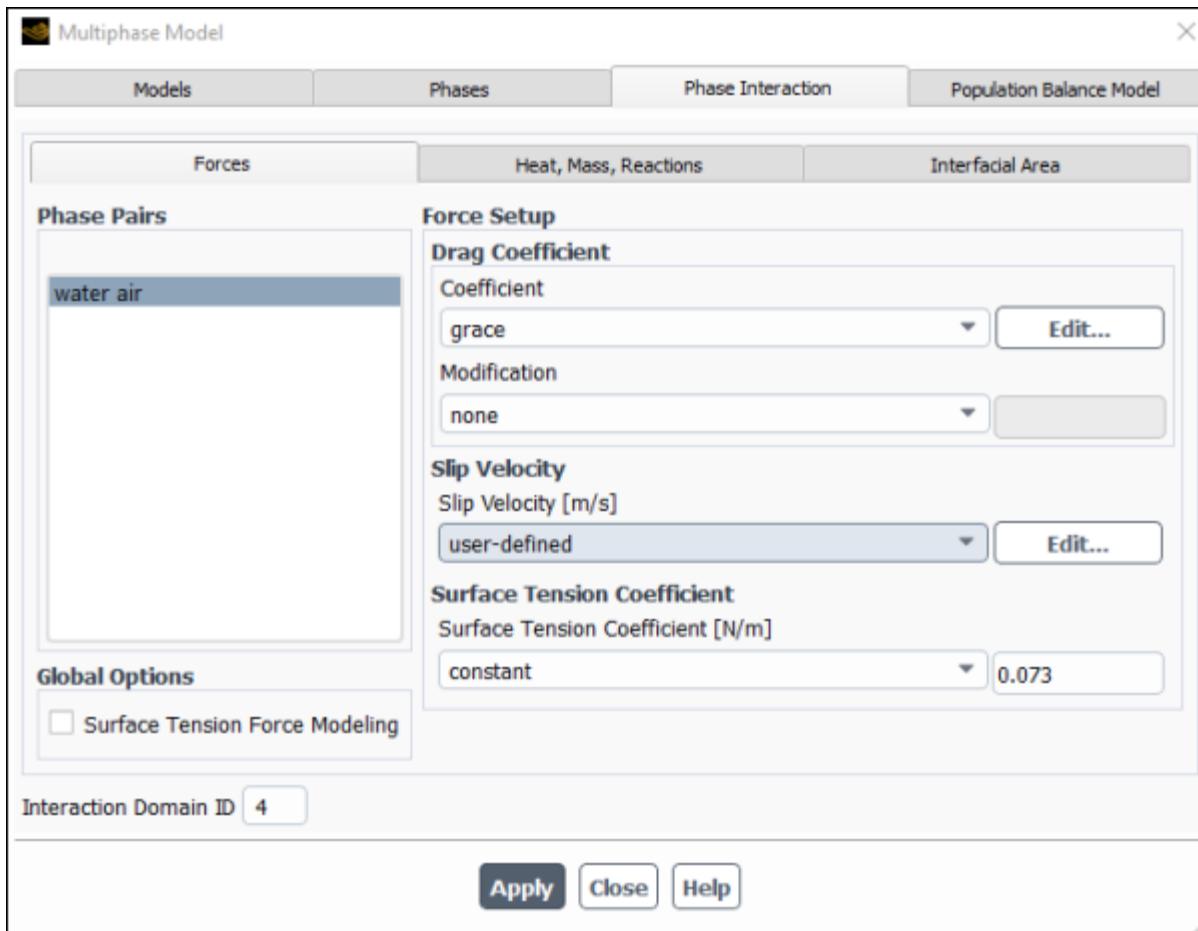
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_VECTOR_EXCHANGE_RATE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Functions** dialog box (Figure 6.89: The User-Defined Functions Dialog Box (p. 534)) in Ansys Fluent.

To hook the UDF to Ansys Fluent:

1. Select **Mixture** from the **Model** list of the **Multiphase Model** dialog box and make sure that the **Slip Velocity** option is enabled.

   Setup → **Models** → **Multiphase** → **Edit...**
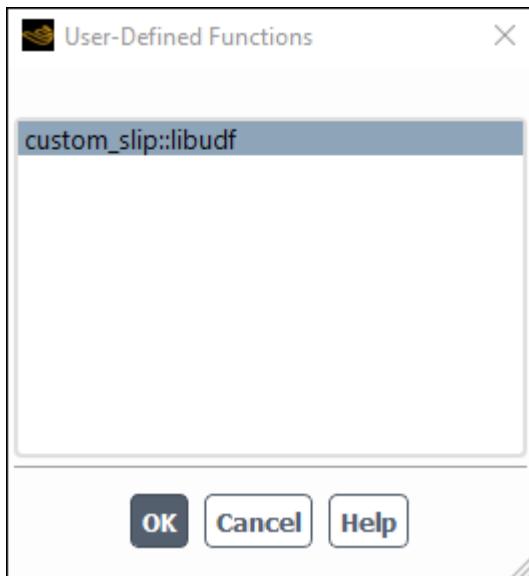
2. Click **Apply**.

3. Go to the **Phase Interaction** > **Forces** tab (Figure 6.88: The Multiphase Model Dialog Box - Force Tab (p. 533)).

Figure 6.88: The Multiphase Model Dialog Box - Force Tab

4. In the **Force Setup** group box, select **user-defined** from the drop-down list for the **Slip Velocity**.

   The **User-Defined Functions** dialog box opens automatically.

Figure 6.89: The User-Defined Functions Dialog Box

5. Select the function name (for example, **custom_slip::libudf**) from the list of UDFs displayed in the **User-Defined Functions** dialog box (Figure 6.89: The User-Defined Functions Dialog Box (p. 534)), and click **OK**.

6. Click **Apply** and close the **Multiphase** dialog box.

See `DEFINE_VECTOR_EXCHANGE_PROPERTY` (p. 199) for details about `DEFINE_VECTOR_EX-CHANGE_PROPERTY` functions.

6.4. Hooking Discrete Phase Model (DPM) UDFs

This section contains methods for hooking UDFs to Ansys Fluent that have been

- defined using `DEFINE` macros described in Discrete Phase Model (DPM) `DEFINE` Macros (p. 201), and

- interpreted or compiled using methods described in Interpreting UDFs (p. 393) or Compiling UD-Fs (p. 399), respectively.

For more information, see the following sections:

6.4.1. Hooking `DEFINE_DPM_BC` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_BC` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the appropriate boundary condition dialog box (Figure 6.90: The Wall Dialog Box (p. 536)) in Ansys Fluent.
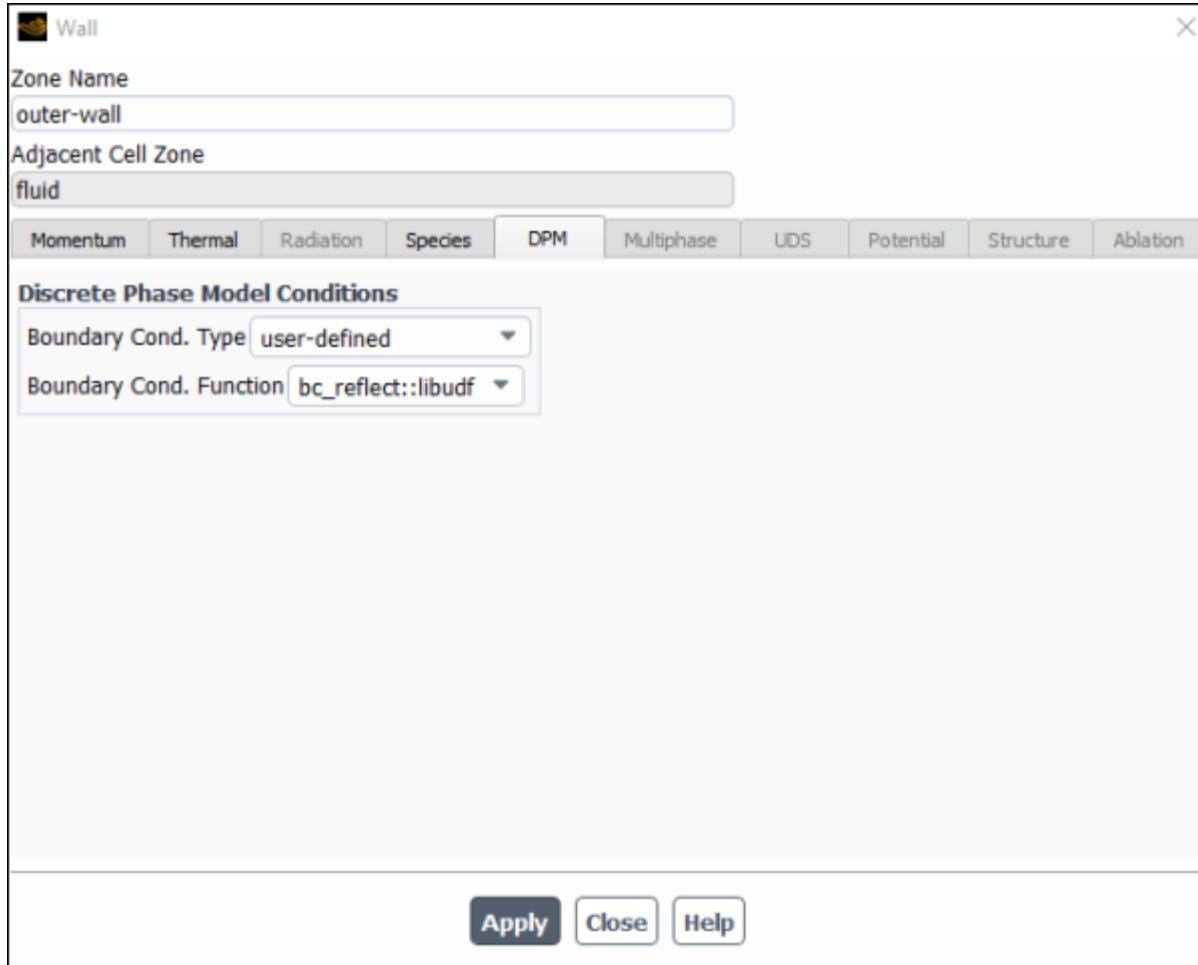
To hook the UDF, first create an injection using the **Injections** dialog box.

Setup → **Models** → **Discrete Phase** → **Injections** ⟶ **New...**

Next, open the **Boundary Conditions** task page.

Setup → **Boundary Conditions**

Select the boundary in the **Zone** list and click **Edit...** to open the boundary condition dialog box (for example, the **Wall** dialog box, as shown in Figure 6.90: The Wall Dialog Box (p. 536)).

Figure 6.90: The Wall Dialog Box

Click the **DPM** tab and select **user_defined** from the **Boundary Cond. Type** drop-down list in the **Discrete Phase Model Conditions** group box. This will expand the dialog box to allow you to select the function name (for example, **bc_reflect::libudf**) from the **Boundary Cond. Function** drop-down list (Figure 6.90: The Wall Dialog Box (p. 536)). Click **OK**.
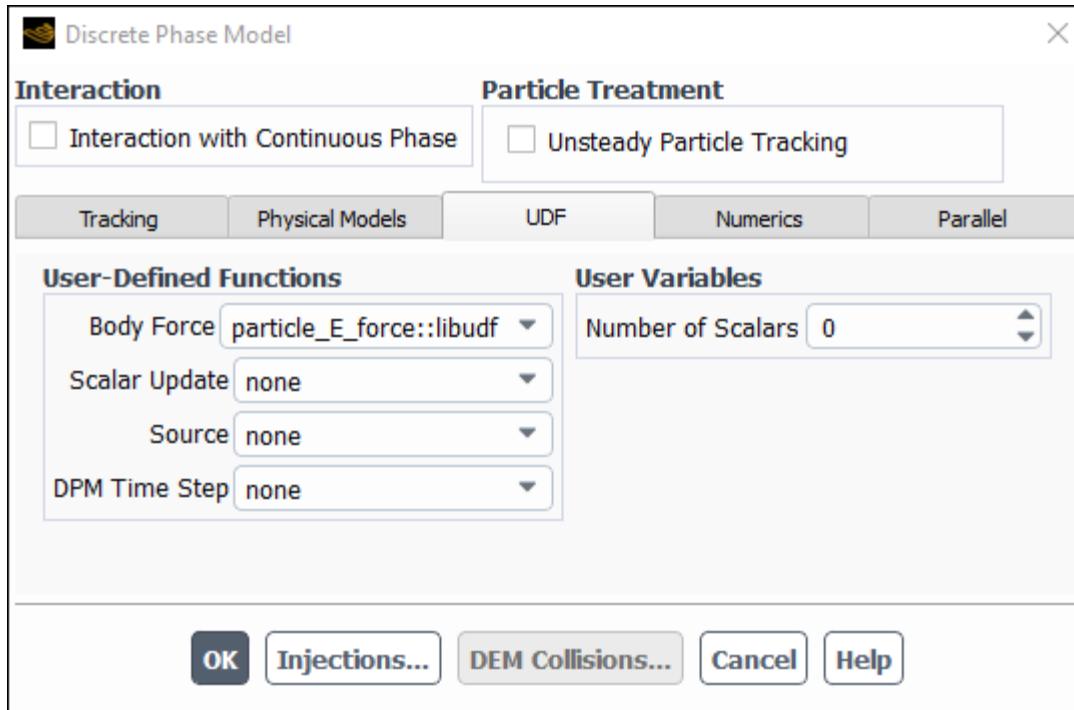
See `DEFINE_DPM_BC` (p. 203) for details about `DEFINE_DPM_BC` functions.

6.4.2. Hooking `DEFINE_DPM_BODY_FORCE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_BODY_FORCE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Discrete Phase Model** dialog box (Figure 6.91: The Discrete Phase Model Dialog Box (p. 537)) in Ansys Fluent.

To hook the UDF, first open the **Discrete Phase Model** dialog box.

📇 **Setup** → **Models** → **Discrete Phase** 🖱 **Edit...**

Figure 6.91: The Discrete Phase Model Dialog Box

Click the **Injections...** button to open the **Injections** dialog box. Create an injection and then click **Close** in the **Injections** dialog box.
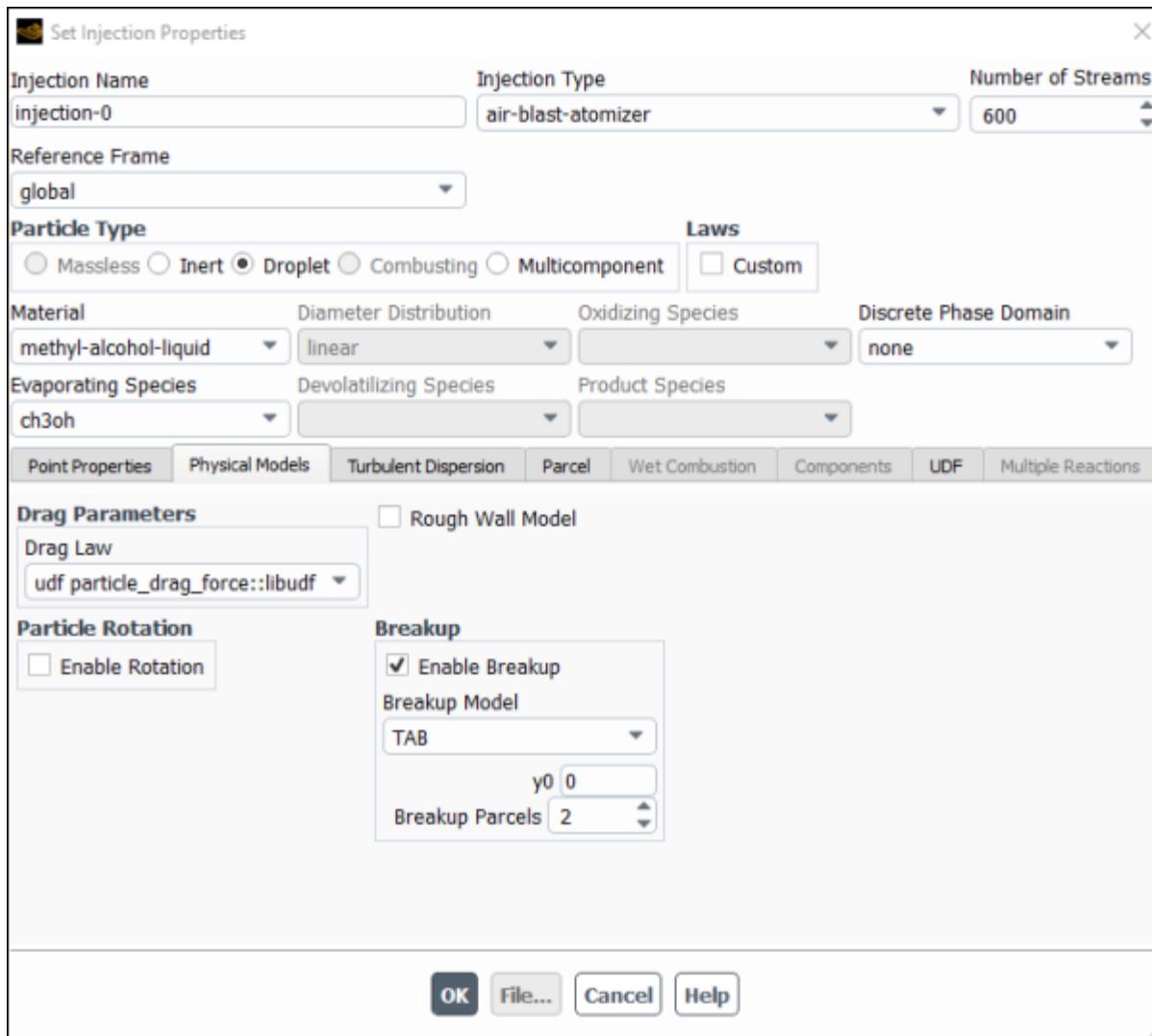
Next, click the **UDF** tab in the **Discrete Phase Model** dialog box. Select the function name (for example, **particle_body_force::libudf**) from the **Body Force** drop-down list under **User-Defined Functions** (Figure 6.91: The Discrete Phase Model Dialog Box (p. 537)), and click **OK**.

See `DEFINE_DPM_BODY_FORCE` (p. 210) for details about `DEFINE_DPM_BODY_FORCE` functions.

6.4.3. Hooking `DEFINE_DPM_DRAG` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_DRAG` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Set Injection Properties** dialog box (Figure 6.92: The Set Injection Properties Dialog Box (p. 538)) in Ansys Fluent.

To hook the UDF, open the **Injections** dialog box by clicking **Discrete Phase** and selecting **Injections...** in the **Setting Up Physics** ribbon tab (**Model Specific** group box) or by clicking **Injections...** in the **Discrete Phase Model** dialog box. In the **Injections** dialog box, **Set...** or **Create** the injection that the user-defined DPM drag function specifies. This will bring up the **Set Injection Properties** dialog box.

Figure 6.92: The Set Injection Properties Dialog Box

Click the **Physical Models** tab in the **Set Injection Properties** dialog box. Select the function name (for example, **particle_drag_force::libudf**) from the **Drag Law** drop-down list in the **Drag Parameters** group box (Figure 6.92: The Set Injection Properties Dialog Box (p. 538)), and click **OK**. (Note that function names listed in the drop-down list are preceded by the word **udf**, as in **udf particle_drag_force::libudf**.)

See `DEFINE_DPM_DRAG` (p. 212) for details about `DEFINE_DPM_DRAG` functions.
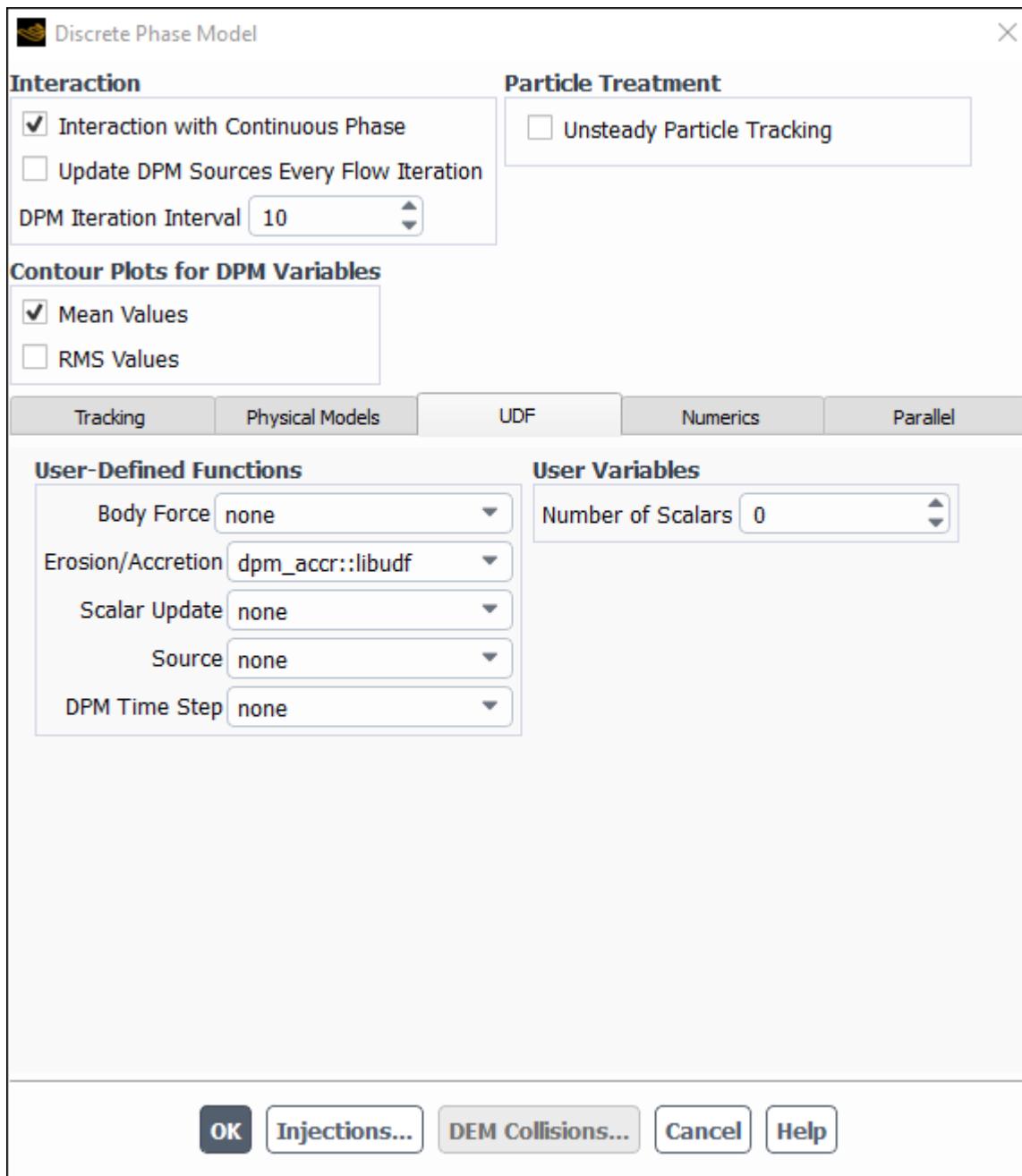
6.4.4. Hooking `DEFINE_DPM_EROSION` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_EROSION` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Discrete Phase Model** dialog box (Figure 6.93: The Discrete Phase Model Dialog Box (p. 539)) in Ansys Fluent.

To hook the UDF, first open the **Discrete Phase Model** dialog box.

⊞ Setup → Models → Discrete Phase 🖱 Edit...

Figure 6.93: The Discrete Phase Model Dialog Box

Click the **Injections...** button to open the **Injections** dialog box. Create an injection and then click **Close** in the **Injections** dialog box.

Next, enable the **Interaction with Continuous Phase** option under **Interaction** in the **Discrete Phase Model** dialog box. Then, click the **Physical Models** tab and enable the **Erosion/Accretion** option. Finally, click the **UDF** tab and select the function name (for example, **dpm_accr::libudf**) from the **Erosion/Accretion** drop-down list in the **User-Defined Functions** group box (Figure 6.92: The Set Injection Properties Dialog Box (p. 538)), and click **OK**.

See `DEFINE_DPM_EROSION` (p. 213) for details about `DEFINE_DPM_EROSION` functions.

6.4.5. Hooking `DEFINE_DPM_HEAT_MASS` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_HEAT_MASS` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Set Injection Properties** dialog box (Figure 6.94: The Set Injections Dialog Box (p. 541)) in Ansys Fluent.
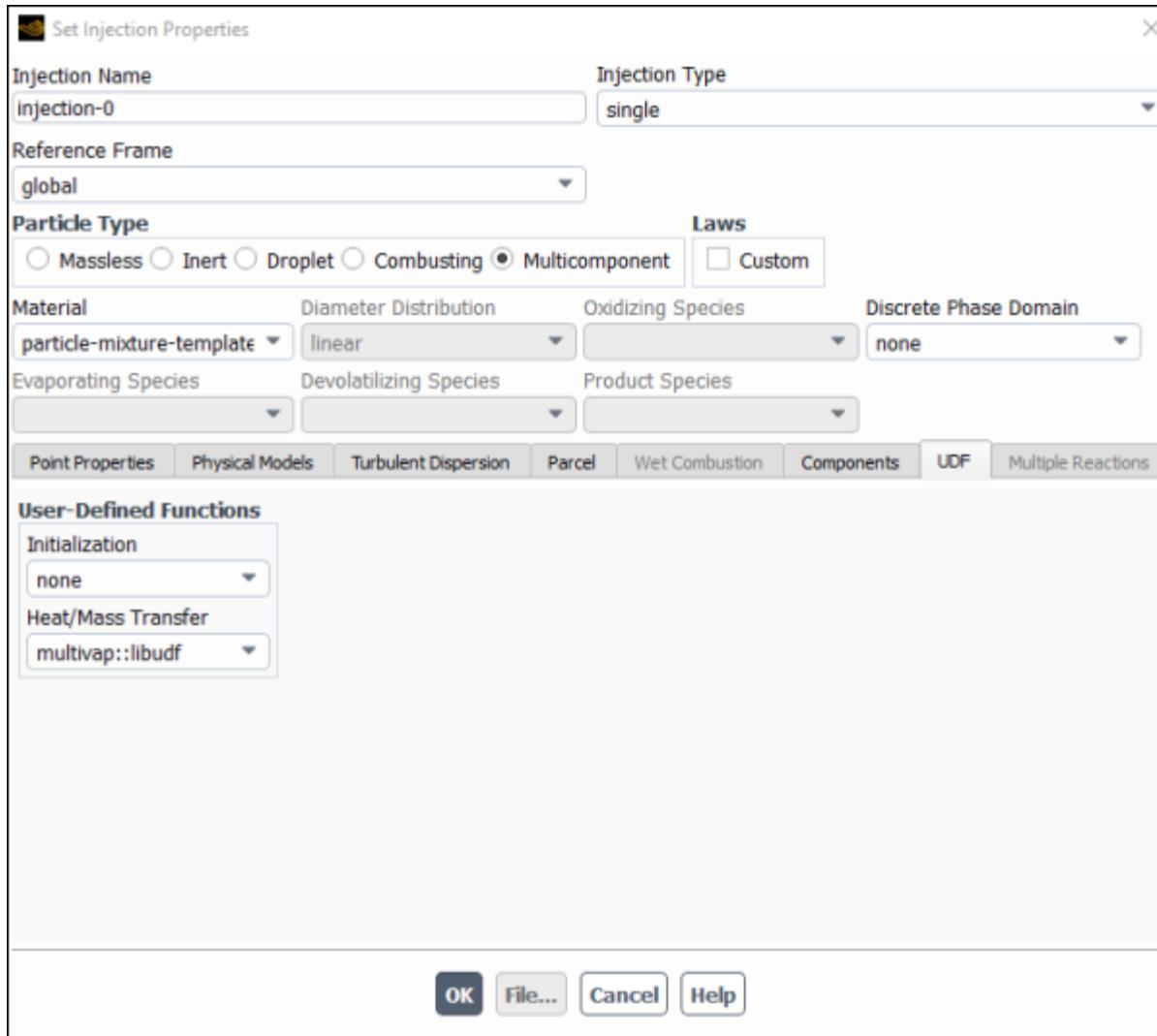
To hook the UDF, first set up your species transport model in the **Species Model** dialog box.

Setup → **Models** → **Species** 🖱 **Edit...**

Select **Species Transport** from the **Model** list and click **OK**.

Next, create a particle injection in the **Injections** dialog box.

Setup → **Models** → **Discrete Phase** → **Injections** 🖱 **New...**

Figure 6.94: The Set Injections Dialog Box

Set up the particle injection in the **Set Injection Properties** dialog box, being sure to select **Multicomponent** in the **Particle Type** group box. Then click the **UDF** tab, and select the function name (for example, **multivap::libudf**) from the **Heat/Mass Transfer** drop-down list in the **User-Defined Functions** group box. Click **OK**.

See `DEFINE_DPM_INJECTION_INIT` (p. 220) for details about `DEFINE_DPM_INJECTION_INIT` functions.
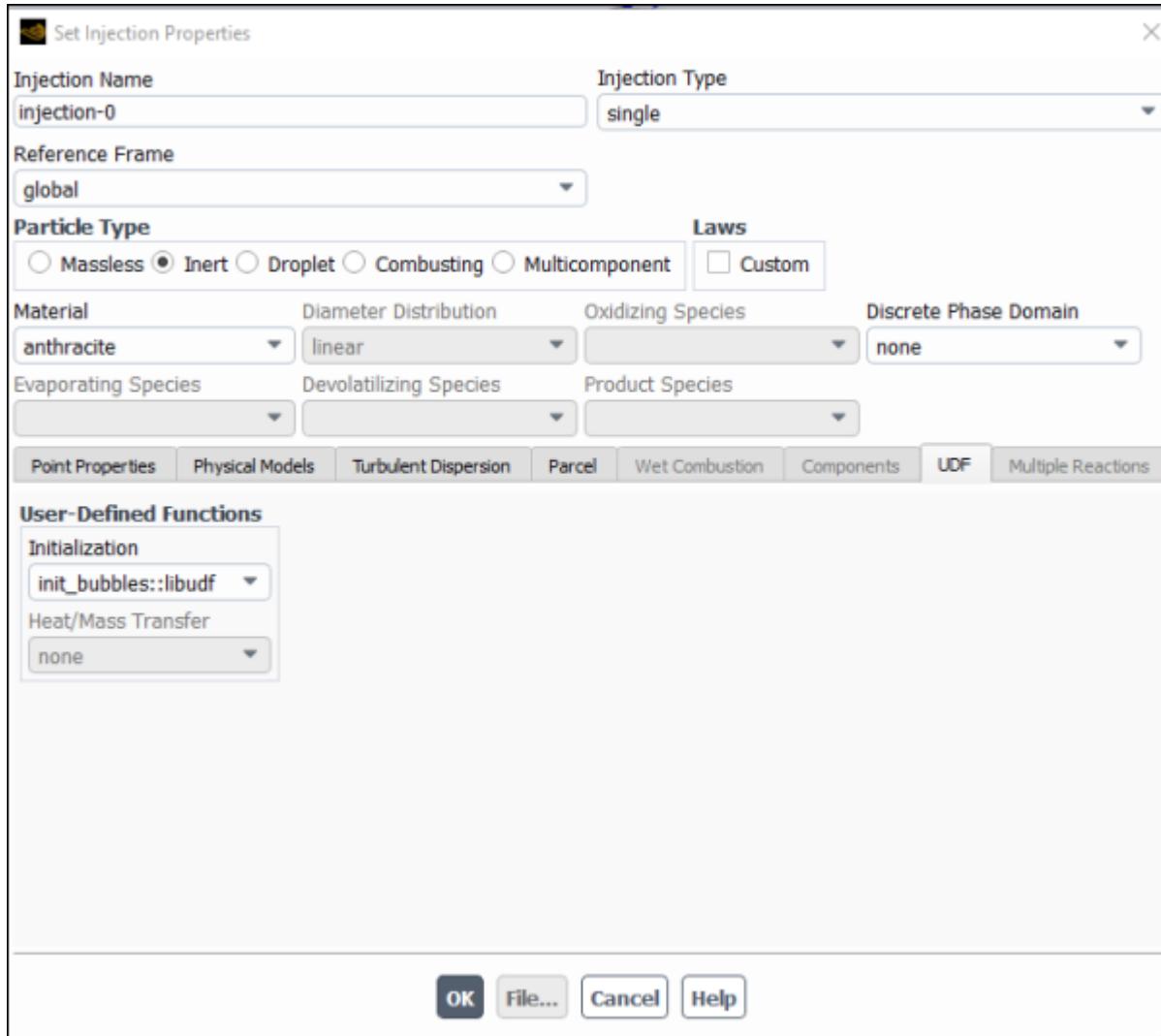
6.4.6. Hooking `DEFINE_DPM_INJECTION_INIT` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_INJECTION_INIT` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Set Injection Properties** dialog box (Figure 6.95: The Injections Dialog Box (p. 542)) in Ansys Fluent.

To hook the UDF, first create a particle injection in the **Injections** dialog box.

⊞ Setup → Models → Discrete Phase → Injections 🖱 New...

Figure 6.95: The Injections Dialog Box

Set up the particle injection in the **Set Injection Properties** dialog box. Then click the **UDF** tab and select the function name (for example, **init_bubbles::libudf**) from the **Initialization** drop-down list under **User-Defined Functions**. Click **OK**.

See `DEFINE_DPM_INJECTION_INIT` (p. 220) for details about `DEFINE_DPM_INJECTION_INIT` functions.

6.4.7. Hooking `DEFINE_DPM_LAW` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_LAW` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Custom Laws** dialog box (Figure 6.96: The Custom Laws Dialog Box (p. 543)) in Ansys Fluent.

To hook the UDF to Ansys Fluent, first create a particle injection in the **Injections** dialog box.

⊞ Setup → Models → Discrete Phase → Injections 🖱 New...

Enable the **Custom** option in the **Laws** group box in the **Set Injection Properties** dialog box, in order to open the **Custom Laws** dialog box (Figure 6.96: The Custom Laws Dialog Box (p. 543)).

Figure 6.96: The Custom Laws Dialog Box

In the **Custom Laws** dialog box, select the function name (for example, **Evapor_Swelling_Law::libudf**) from the appropriate drop-down list located to the left of each of the particle laws (for example, **First Law**), and click **OK**.

See `DEFINE_DPM_LAW` (p. 226) for details about `DEFINE_DPM_LAW` functions.

6.4.8. Hooking `DEFINE_DPM_OUTPUT` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_OUTPUT` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Sample Trajectories** dialog box (Figure 6.97: The Sample Trajectories Dialog Box (p. 544)) in Ansys Fluent.

In order to hook the UDF, you must first set up the discrete phase model (for example, create a particle injection). After you have run the calculation and generated data, open the **Sample Trajectories** dialog box (Figure 6.97: The Sample Trajectories Dialog Box (p. 544)).

⊞ Results → Reports → Discrete Phase → Sample 🖱 Edit...

Figure 6.97: The Sample Trajectories Dialog Box

1. Select the appropriate injection in the **Release From Injections** list.

2. Indicate where the sample will be written (for example, make a selection in the **Planes** list (in 3D) or **Lines** list (in 2D)).

3. Select the function name (for example, **discrete_phase_sample::libudf**) from the **Output** drop-down list under **User-Defined Functions**.

4. Click **Compute** (for steady calculations) or **Start** (for transient calculations).

> **Note:**
>
> If you want to use your `DEFINE_DPM_OUTPUT` UDF only for the VOF-to-DPM lump conversion transcript, you can just select the function name from the **Output** drop-down list and click **Apply**. In this case, no injection must be selected in the **Release From Injections** list.

See `DEFINE_DPM_OUTPUT` (p. 227) for details about `DEFINE_DPM_OUTPUT` functions.

6.4.9. Hooking `DEFINE_DPM_PROPERTY` UDFs
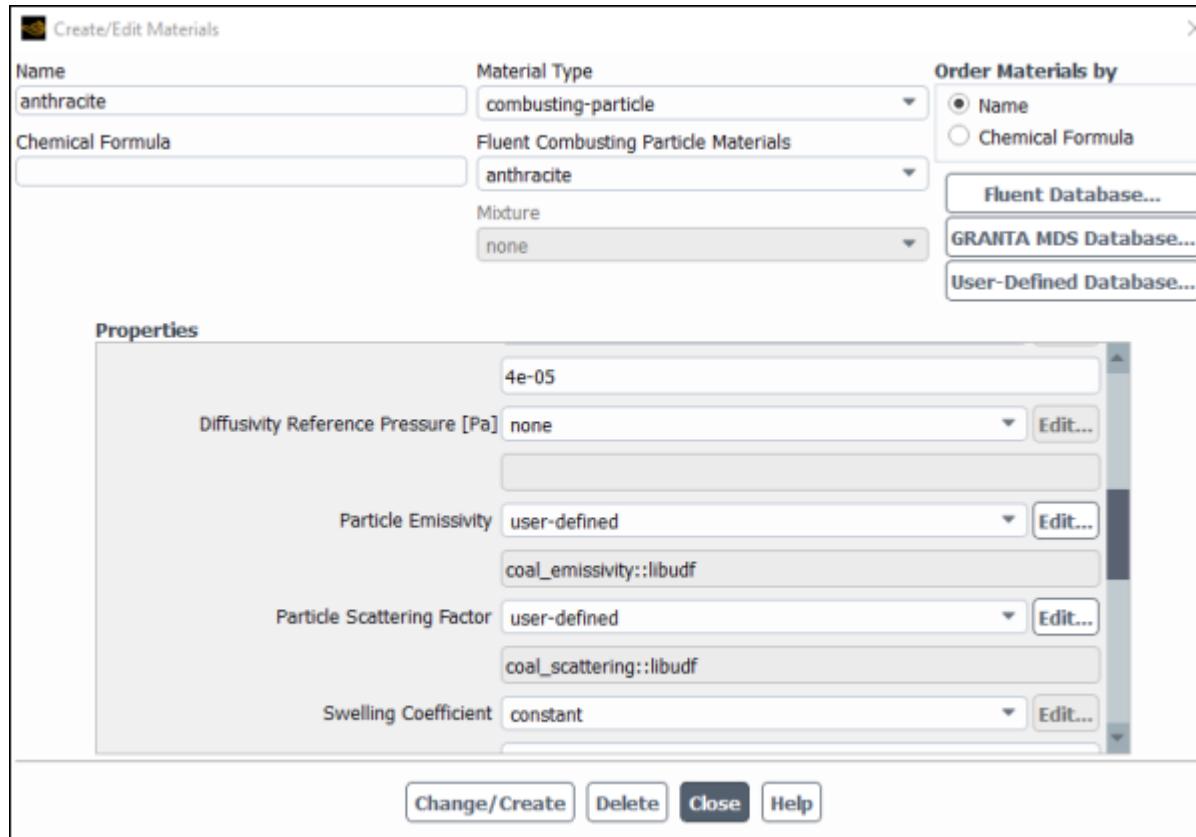
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_PROPERTY` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **User-Defined Functions** dialog box.

To hook the UDF to Ansys Fluent, you will first need to open the **Materials** task page.

 Setup → **Materials**

Select a material from **Materials** list and click the **Create/Edit...** button to open the **Create/Edit Materials** dialog box (Figure 6.98: The Create/Edit Materials Dialog Box (p. 545)).
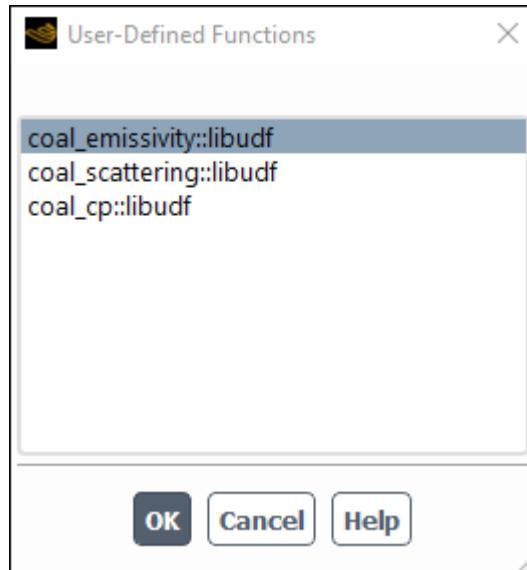
Figure 6.98: The Create/Edit Materials Dialog Box

Select **user-defined** in the drop-down list for one of the properties (for example, **Particle Emissivity**) in the **Create/Edit Materials** dialog box, in order to open the **User-Defined Functions** dialog box (Figure 6.99: The User-Defined Functions Dialog Box (p. 546)).

Important:

In order for the **Particle Emissivity** property to be displayed in the sample dialog box shown above, you must enable a radiation model, turn on the **Particle Radiation Interaction** option in the **Discrete Phase Model** dialog box, and introduce a particle injection in the **Injections** dialog box.

Figure 6.99: The User-Defined Functions Dialog Box

Select the function name (for example, **coal_emissivity::libudf**) from the list of UDFs displayed in the **User-Defined Functions** dialog box, and click **OK**. The name of the function will subsequently be displayed under the selected property (for example, **Particle Emissivity**) in the **Create/Edit Materials** dialog box.
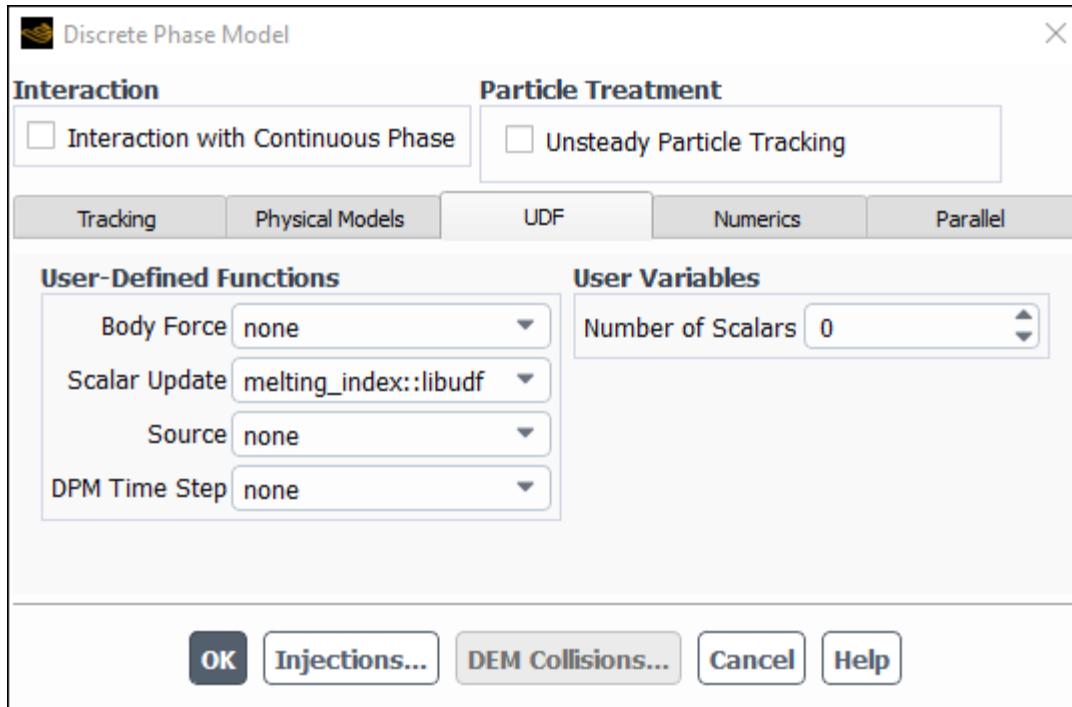
See `DEFINE_PROPERTY` UDFs (p. 119) for details about `DEFINE_DPM_PROPERTY` functions.

6.4.10. Hooking `DEFINE_DPM_SCALAR_UPDATE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_SCALAR_UPDATE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Discrete Phase Model** dialog box (Figure 6.100: The Discrete Phase Model Dialog Box (p. 547)) in Ansys Fluent.

To hook the UDF, first open the **Discrete Phase Model** dialog box.

Setup → **Models** → **Discrete Phase** 🖱 **Edit...**

Figure 6.100: The Discrete Phase Model Dialog Box

Click the **Injections...** button to open the **Injections** dialog box. Create an injection and then click **Close** in the **Injections** dialog box.

Next, click the **UDF** tab in the **Discrete Phase Model** dialog box. Select the function name (for example, **melting_index::libudf**) from the **Scalar Update** drop-down list under **User-Defined Functions** (Figure 6.100: The Discrete Phase Model Dialog Box (p. 547)), and click **OK**.
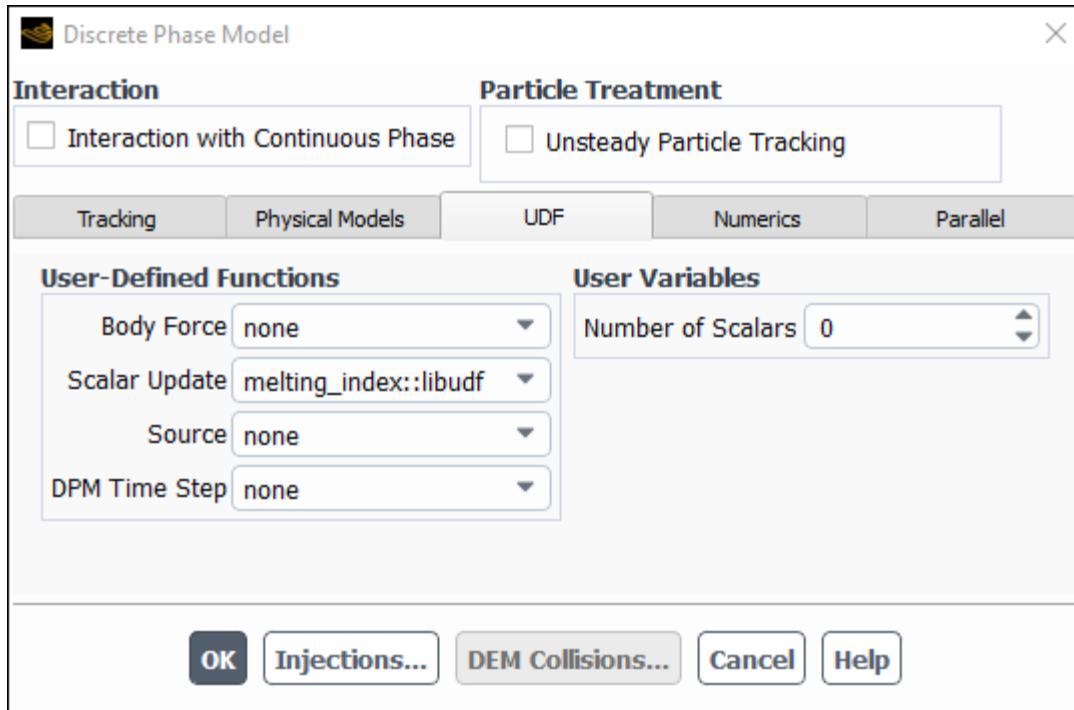
See `DEFINE_DPM_SCALAR_UPDATE` (p. 241) for details about `DEFINE_DPM_SCALAR_UPDATE` functions.

6.4.11. Hooking `DEFINE_DPM_SOURCE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_SOURCE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Discrete Phase Model** dialog box (Figure 6.101: The Discrete Phase Model Dialog Box (p. 548)) in Ansys Fluent.

To hook the UDF, first open the **Discrete Phase Model** dialog box.

   Setup → Models → Discrete Phase 🖱 **Edit...**

Figure 6.101: The Discrete Phase Model Dialog Box

Click the **Injections...** button to open the **Injections** dialog box. Create an injection and then click **Close** in the **Injections** dialog box.

Next, click the **UDF** tab in the **Discrete Phase Model** dialog box. Select the function name (for example, **dpm_source::libudf**) from the **Source** drop-down list under **User-Defined Functions** (Figure 6.101: The Discrete Phase Model Dialog Box (p. 548)), and click **OK**.
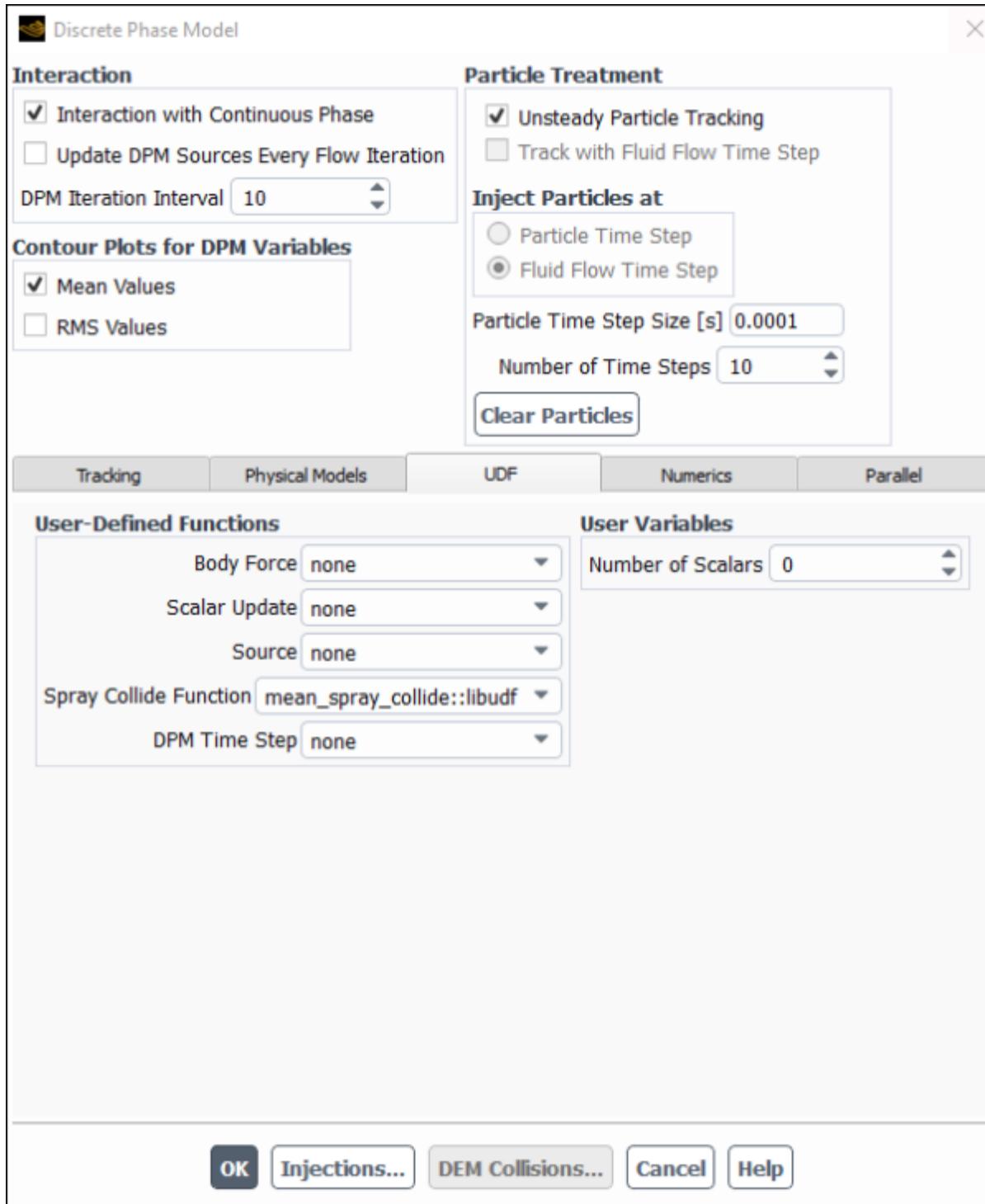
See `DEFINE_DPM_SOURCE` (p. 244) for details about `DEFINE_DPM_SOURCE` functions.

6.4.12. Hooking `DEFINE_DPM_SPRAY_COLLIDE` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_SPRAY_COLLIDE` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Discrete Phase Model** dialog box (Figure 6.102: The Discrete Phase Model Dialog Box (p. 549)) in Ansys Fluent.

To hook the UDF, first open the **Discrete Phase Model** dialog box.

Setup → **Models** → **Discrete Phase** Edit...

Figure 6.102: The Discrete Phase Model Dialog Box

Click the **Injections...** button to open the **Injections** dialog box. Create an injection and then click **Close** in the **Injections** dialog box.

Next, click the **Physical Models** tab in the **Discrete Phase Model** dialog box and enable the **Droplet Collision** option in the **Spray Model** group box. Then, click the **UDF** tab and select the function name (for example, **mean_spray_collide::libudf**) from the **Spray Collide Function** drop-down list

in the **User-Defined Functions** group box (Figure 6.92: The Set Injection Properties Dialog Box (p. 538)), and click **OK**.

See `DEFINE_DPM_SPRAY_COLLIDE` (p. 245) for details about `DEFINE_DPM_SPRAY_COLLIDE` functions.

6.4.13. Hooking `DEFINE_DPM_SWITCH` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_SWITCH` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Custom Laws** dialog box (Figure 6.103: The Custom Laws Dialog Box (p. 550)) in Ansys Fluent.

To hook the UDF to Ansys Fluent, first create a particle injection in the **Injections** dialog box.

📁 **Setup** → **Models** → **Discrete Phase** → **Injections** 🖱 **New...**

Enable the **Custom** option in the **Laws** group box in the **Set Injection Properties** dialog box, in order to open the **Custom Laws** dialog box (Figure 6.103: The Custom Laws Dialog Box (p. 550)).

Figure 6.103: The Custom Laws Dialog Box

In the **Custom Laws** dialog box, select the function name (for example, **dpm_switch::libudf**) from the **Switching** drop-down list and click **OK**.

See `DEFINE_DPM_SWITCH` (p. 247) for details about `DEFINE_DPM_SWITCH` functions.
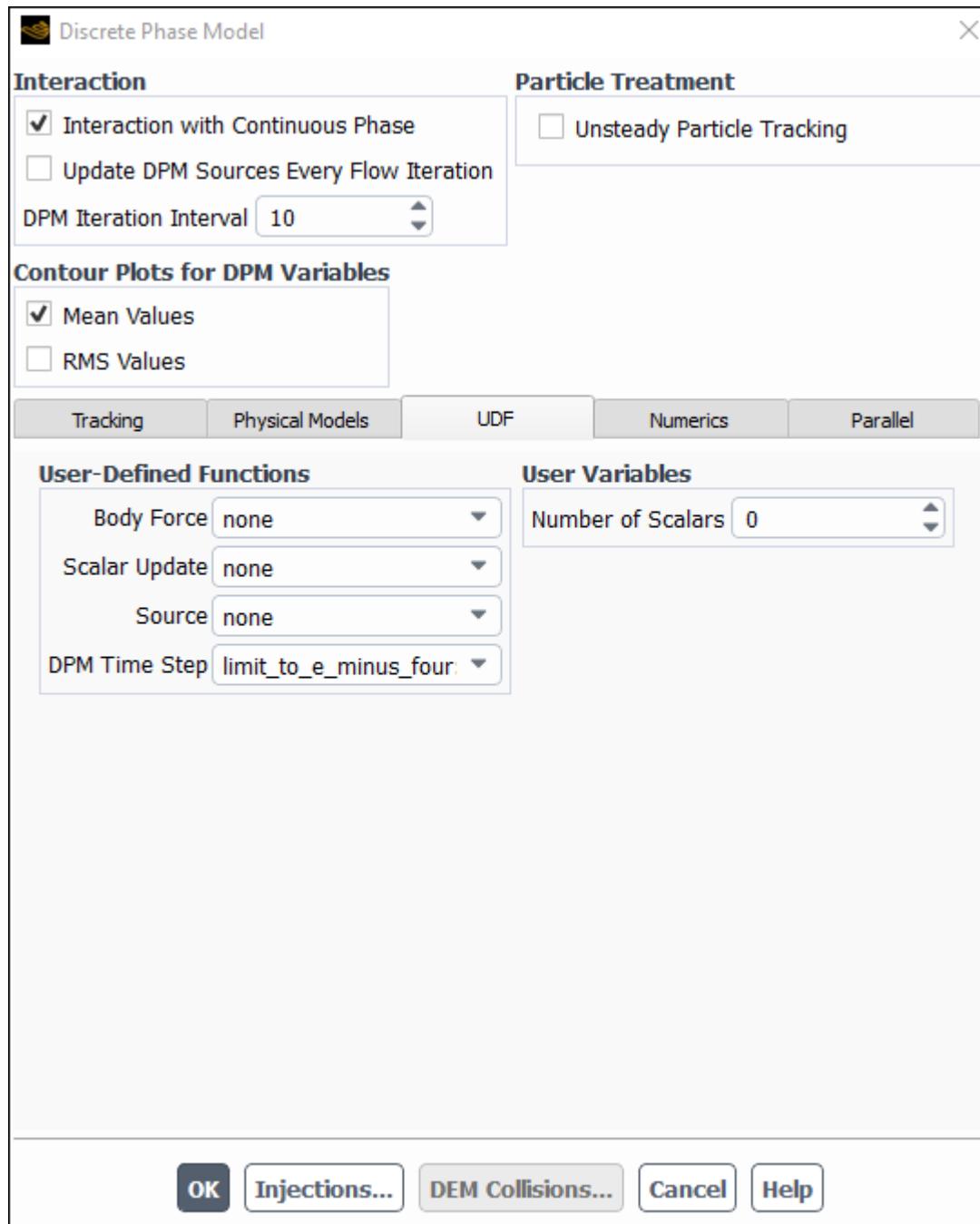
6.4.14. Hooking `DEFINE_DPM_TIMESTEP` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_TIMESTEP` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in Ansys Fluent.

To hook the UDF, first open the **Discrete Phase Model** dialog box.

Setup → **Models** → **Discrete Phase** ⌯ **Edit...**

Figure 6.104: The Discrete Phase Model Dialog Box

Click the **Injections...** button to open the **Injections** dialog box. Create an injection and then click **Close** in the **Injections** dialog box.

Next, click the **UDF** tab in the **Discrete Phase Model** dialog box. Select the function name (for example, **limit_to_e_minus_four::libudf**) from the **DPM Time Step** drop-down list under **User-Defined Functions** (Figure 6.104: The Discrete Phase Model Dialog Box (p. 551)), and click **OK**.

See `DEFINE_DPM_TIMESTEP` (p. 250) for details about `DEFINE_DPM_TIMESTEP` functions.

6.4.15. Hooking `DEFINE_DPM_VP_EQUILIB` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_DPM_VP_EQUILIB` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable from the **Create/Edit Materials** dialog box in Ansys Fluent.

To hook the UDF, first set up your species transport and combustion models in the **Species Model** dialog box.

 Setup → **Models** → **Species** **Edit...**

Then, create a particle injection using the **Injections** dialog box.
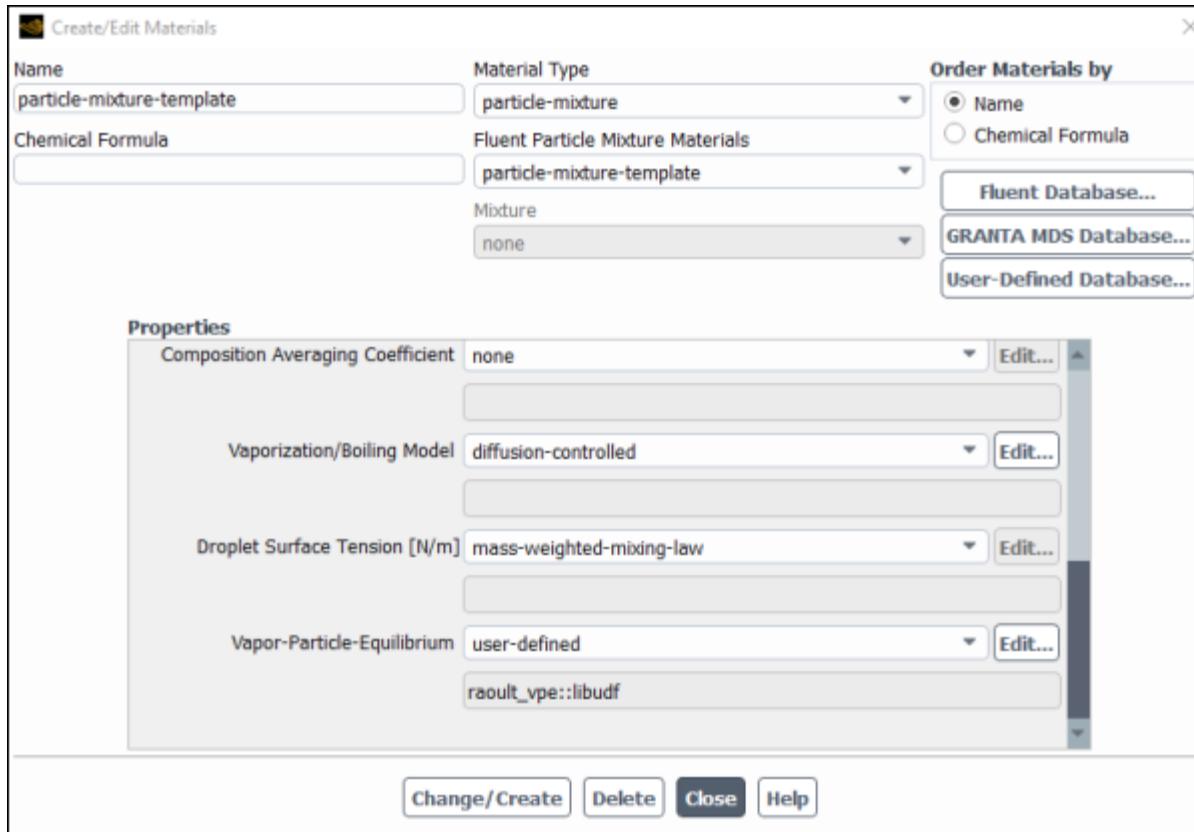
 Setup → **Models** → **Discrete Phase** → **Injections** **New...**

Set up the particle injection, making sure to select **Multicomponent** for the **Particle Type**.
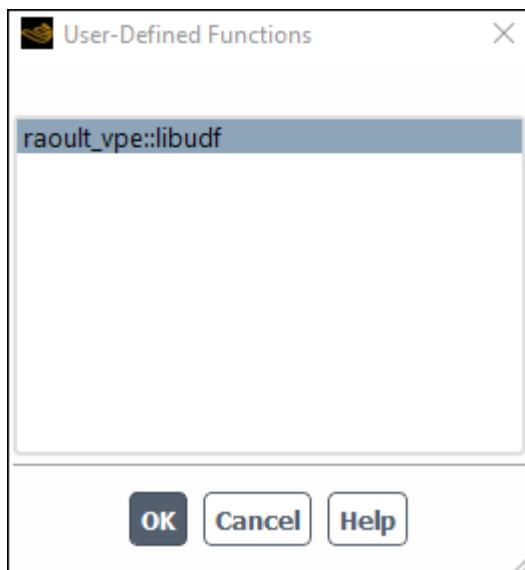
Next, open the **Materials** task page.

 Setup → **Materials**

Select the appropriate material in the **Materials** list (for example, **particle-mixture-template**) and click **Create/Edit** to open the **Create/Edit Materials** dialog box (Figure 6.105: The Create/Edit Materials Dialog Box (p. 553)).

Figure 6.105: The Create/Edit Materials Dialog Box

Select **user-defined** from the drop-down list for **Vapor-Particle-Equilibrium** in the **Properties** group box. This will open the **User-Defined Functions** dialog box. Select the UDF name (for example, **raoult_vp::libudf**) from the list of UDFs displayed and click **OK**. Then click **Change/Create** in the **Create/Edit Materials** dialog box.

Figure 6.106: The User-Defined Functions Dialog Box

See `DEFINE_DPM_VP_EQUILIB` (p. 252) for details about DEFINE_DPM_VP_EQUILIBRIUM functions.

6.4.16. Hooking `DEFINE_IMPINGEMENT` UDFs
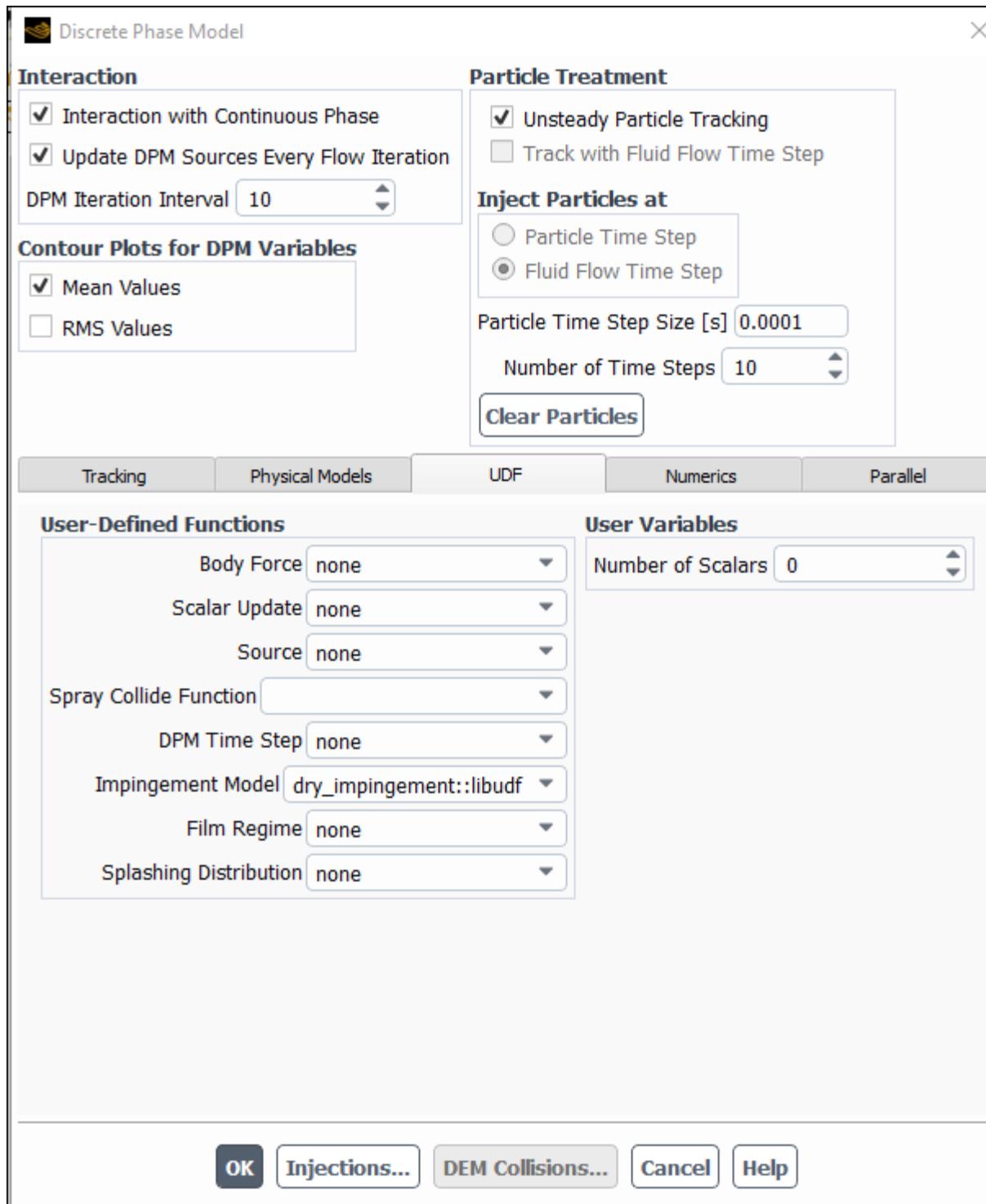
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your DEFINE_IMPINGEMENT UDF, the name of the function you supplied as a DEFINE macro argument will become visible and selectable from in the **Discrete Phase Model** dialog box in Ansys Fluent.

To hook the UDF, follow these steps:

1. Open the **Discrete Phase Model** dialog box.

    Setup → **Models** → **Discrete Phase** 🖱 **Edit...**

2. Click the **Injections...** button and create an injection using the **Injections** dialog box. Close the **Injections** dialog box.

3. Do one of the following:

    •   In the boundary conditions dialog box for the wall of interest, select **wall-film** in the **Boundary Cond. Type** entry on the **DPM** tab.

    •   Enable the Eulerian Wall Film model.

4. In the **Discrete Phase Model** dialog box, open the **UDF** tab and select your function (for example, **dry_impingement::libudf**) from the **Impingement Model** drop-down list.

Figure 6.107: The Discrete Phase Model Dialog Box

See `DEFINE_IMPINGEMENT` (p. 254) for details about `DEFINE_IMPINGEMENT` functions.

6.4.17. Hooking `DEFINE_FILM_REGIME` UDFs
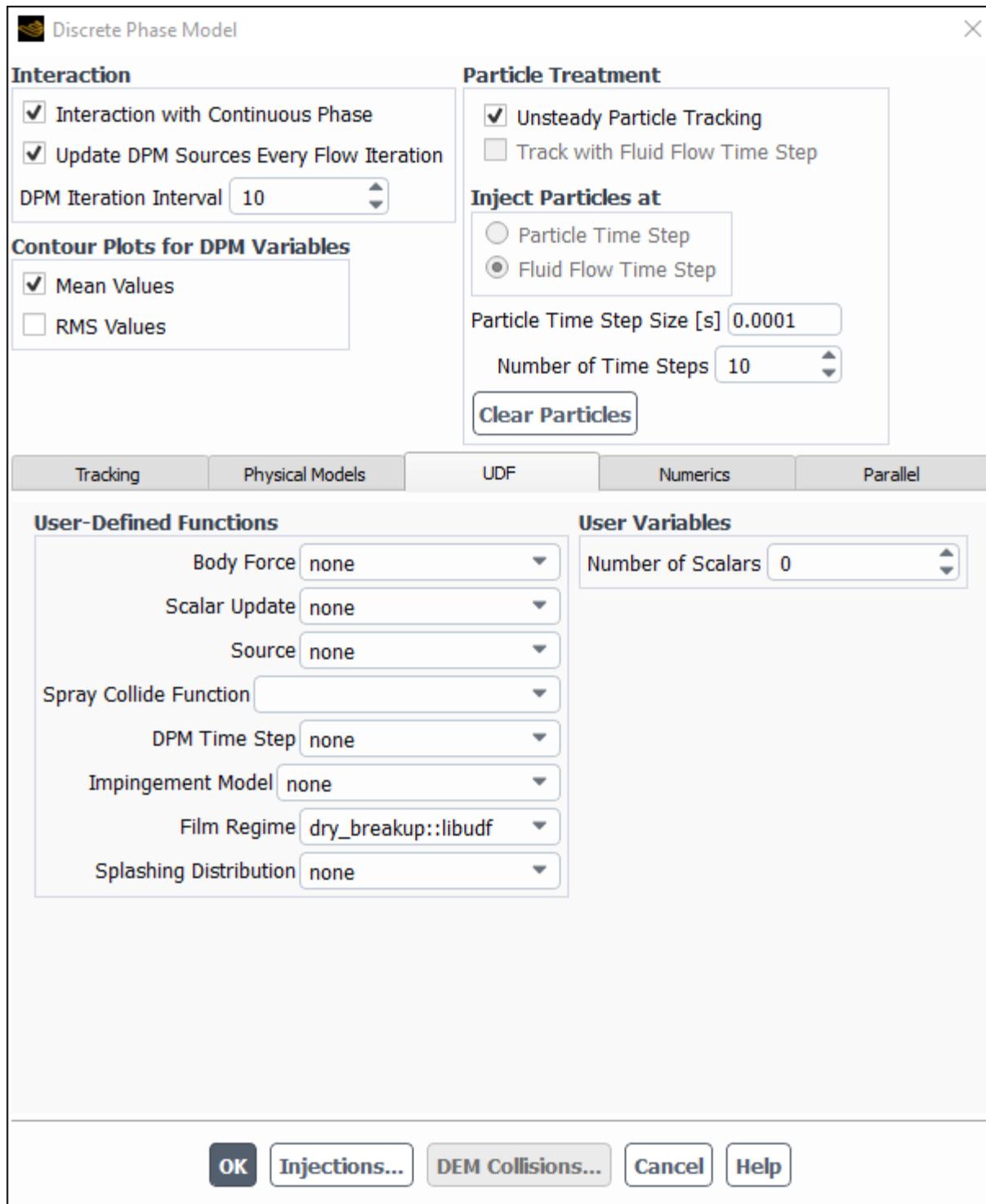
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_FILM_REGIME` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable from in the **Discrete Phase Model** dialog box in Ansys Fluent.

To hook the UDF, follow these steps:

1. Open the **Discrete Phase Model** dialog box.

    🗁 **Setup** → **Models** → **Discrete Phase** 🖱 **Edit...**

2. Click the **Injections...** button and create an injection using the **Injections** dialog box. Close the **Injections** dialog box.

3. Do one of the following:

    • In the boundary conditions dialog box for the wall of interest, select **wall-film** in the **Boundary Cond. Type** entry on the **DPM** tab.

    • Enable the Eulerian Wall Film model.

4. In the **Discrete Phase Model** dialog box, open the **UDF** tab and select your function (for example, **dry_breakup::libudf**) from the **Film Regime** drop-down list.

Figure 6.108: The Discrete Phase Model Dialog Box

See `DEFINE_FILM_REGIME` (p. 257) for details about `DEFINE_FILM_REGIME` functions.

6.4.18. Hooking `DEFINE_SPLASHING_DISTRIBUTION` UDFs
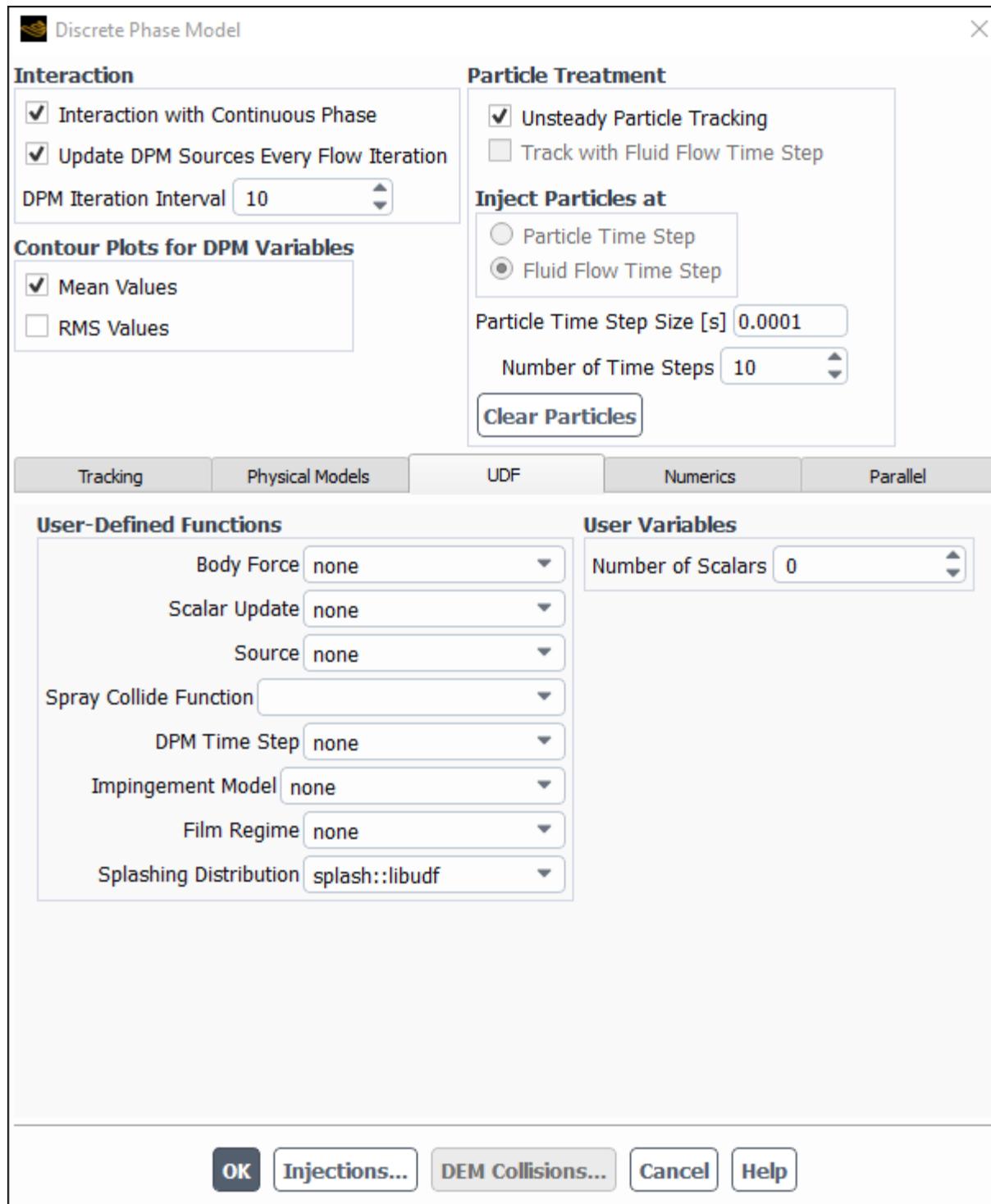
After you have compiled ([Compiling UDFs (p. 399)](#)) your `DEFINE_SPLASHING_DISTRIBUTION` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable from in the **Discrete Phase Model** dialog box in Ansys Fluent.

To hook the UDF, follow these steps:

1. Open the **Discrete Phase Model** dialog box.

   Setup → **Models** → **Discrete Phase**   **Edit...**

2. Click the **Injections...** button and create an injection using the **Injections** dialog box. Close the **Injections** dialog box.

3. Do one of the following:

   • In the boundary conditions dialog box for the wall of interest, select **wall-film** in the **Boundary Cond. Type** entry on the **DPM** tab.

   • Enable the Eulerian Wall Film model.

4. In the **Discrete Phase Model** dialog box, open the **UDF** tab and select your function (for example, **splash::libudf**) from the **Splash Distribution** drop-down list.

Figure 6.109: The Discrete Phase Model Dialog Box

See `DEFINE_SPLASHING_DISTRIBUTION` (p. 259) for details about `DEFINE_SPLASHING_DISTRIBUTION` functions.

6.5. Hooking Dynamic Mesh UDFs

This section contains methods for hooking UDFs to Ansys Fluent that have been defined using `DEFINE` macros described in Dynamic Mesh `DEFINE` Macros (p. 263), and interpreted or compiled using methods described in Interpreting UDFs (p. 393) or Compiling UDFs (p. 399), respectively.

For more information, see the following sections:

6.5.1. Hooking `DEFINE_CG_MOTION` UDFs
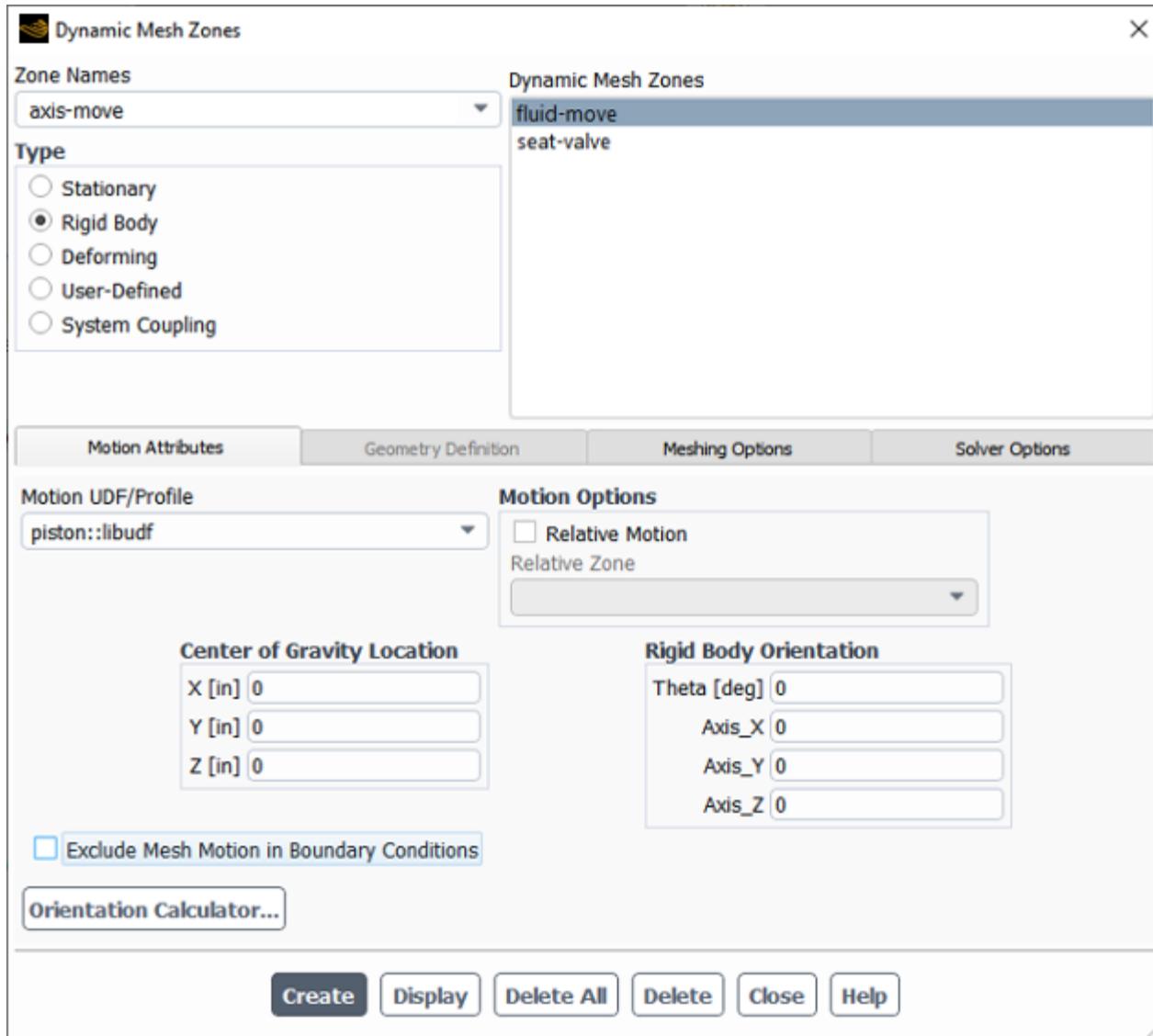
After you have compiled (Compiling UDFs (p. 399)) your `DEFINE_CG_MOTION` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Dynamic Mesh Zones** dialog box (Figure 6.110: The Dynamic Mesh Zones Dialog Box (p. 561)).

To hook the UDF to Ansys Fluent, you will first need to enable the **Dynamic Mesh** option in the **Dynamic Mesh** task page.

 Setup → **Dynamic Mesh** → **Dynamic Mesh**

Next, open the **Dynamic Mesh Zones** dialog box.

 Setup → **Dynamic Mesh** → **Create/Edit...**

Figure 6.110: The Dynamic Mesh Zones Dialog Box

Select **Rigid Body** under **Type** in the **Dynamic Mesh Zones** dialog box (Figure 6.110: The Dynamic Mesh Zones Dialog Box (p. 561)) and click the **Motion Attributes** tab. Finally, select the function name (for example, **piston::libudf**) from the **Motion UDF/Profile** drop-down list, and click **Create** then **Close**.

See `DEFINE_CG_MOTION` (p. 263) for details about `DEFINE_CG_MOTION` functions.

6.5.2. Hooking `DEFINE_DYNAMIC_ZONE_PROPERTY` UDFs

The `DEFINE_DYNAMIC_ZONE_PROPERTY` UDF can be hooked in order to define the following:

- the swirl center for in-cylinder applications

- a variable cell layering height

6.5.2.1. Hooking a Swirl Center UDF

After you have compiled your `DEFINE_DYNAMIC_ZONE_PROPERTY` UDF (as described in Compiling UDFs (p. 399)), the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **In-Cylinder Output Controls** dialog box (Figure 6.111: In-Cylinder Output Controls Dialog Box (p. 562)).

To hook the UDF to Ansys Fluent, you must first right-click the **General** branch of the tree and select **Transient** from the **Analysis Type** sub-menu.
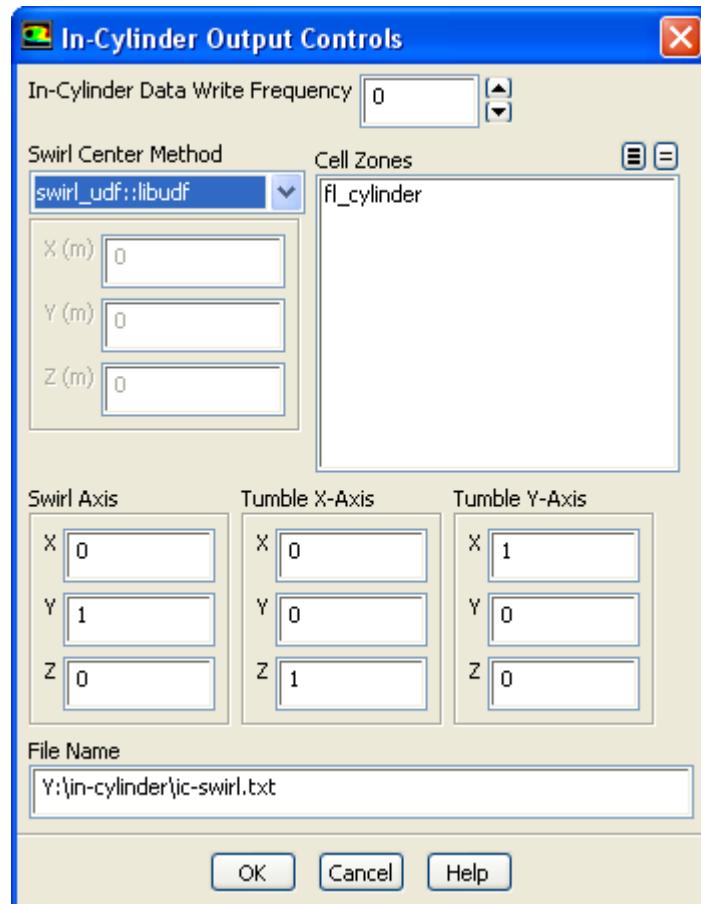
⌨ **Setup** → **General** 🖱 **Analysis Type** → **Transient**

Next, enable the **Dynamic Mesh** option in the **Dynamic Mesh** task page.

⌨ **Setup** → ✛ **Dynamic Mesh** → ☑ **Dynamic Mesh**

Then, enable the **In-Cylinder** option in the **Options** group box, and click the **Settings** button to open the **Options** dialog box. After you have updated the parameters in the **In-Cylinder** tab of this dialog box, click the **Output Controls...** button to open the **In-Cylinder Output Controls** dialog box (Figure 6.111: In-Cylinder Output Controls Dialog Box (p. 562)).

Figure 6.111: In-Cylinder Output Controls Dialog Box

Select the UDF library (for example, swirl_udf::libudf) from the **Swirl Center Method** drop-down list in the **In-Cylinder Output Controls** dialog box. Click **OK** and close the **In-Cylinder Output Controls** dialog box.

See `DEFINE_DYNAMIC_ZONE_PROPERTY` (p. 266) for further details about `DEFINE_DYNAMIC_ZONE_PROPERTY` functions.

6.5.2.2. Hooking a Variable Cell Layering Height UDF

After you have compiled your `DEFINE_DYNAMIC_ZONE_PROPERTY` UDF (as described in Compiling UDFs (p. 399)), the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Dynamic Mesh Zones** dialog box (Figure 6.112: The Dynamic Mesh Zones Dialog Box (p. 564)).

Important:

Since the `DEFINE_DYNAMIC_ZONE_PROPERTY` UDF is a function of time or crank angle, you must make sure that you have selected **Transient** from the **Time** list in the **Solver** group box of the **General** task page before proceeding.
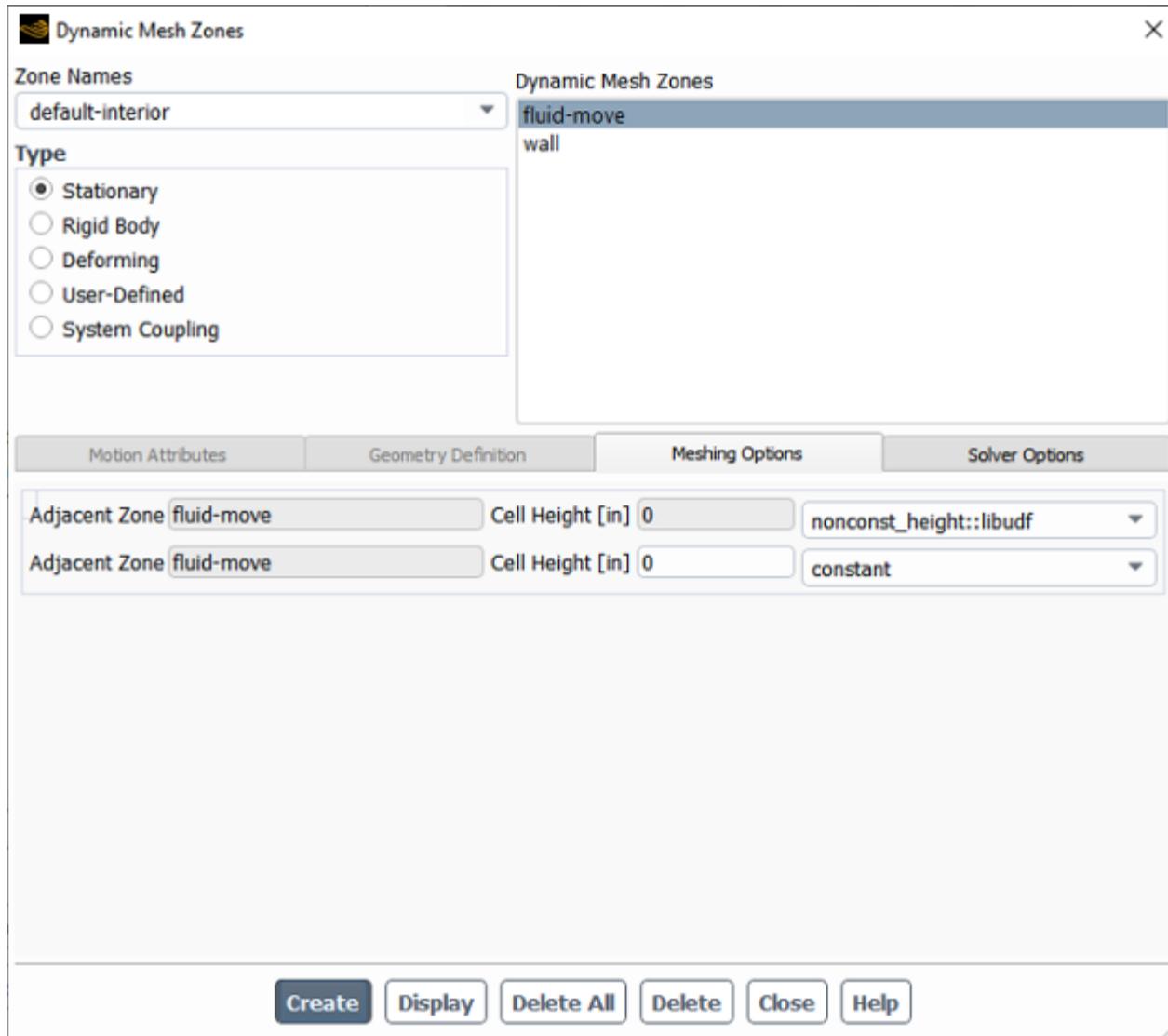
To hook the UDF to Ansys Fluent, you will first need to enable the **Dynamic Mesh** option in the **Dynamic Mesh** task page.

⯐ **Setup** → ✛ **Dynamic Mesh** → ☑ **Dynamic Mesh**

Then, enable the **Layering** option in the **Mesh Methods** list, and click the **Settings...** button to open the **Mesh Methods Settings** dialog box. In the **Layering** tab, select **Height Based** from the **Options** list, and set the **Split Factor** and **Collapse Factor** to appropriate values. Then click **OK**.

Next, specify the meshing options in the **Dynamic Mesh Zones** dialog box (Figure 6.112: The Dynamic Mesh Zones Dialog Box (p. 564)).

⯐ **Setup** → ✛ **Dynamic Mesh** → **Create/Edit...**

Figure 6.112: The Dynamic Mesh Zones Dialog Box

Select **Stationary**, **Rigid Body**, or **User-Defined** from the **Type** list in the **Dynamic Mesh Zones** dialog box. Click the **Meshing Options** tab, and select the UDF library (for example, **nonconst_height::libudf**) from the **Cell Height** drop-down list. Finally, click **Create** and close the **Dynamic Mesh Zones** dialog box.

See `DEFINE_DYNAMIC_ZONE_PROPERTY` for further details about `DEFINE_DYNAMIC_ZONE_PROPERTY` functions.

6.5.3. Hooking `DEFINE_GEOM` UDFs

After you have compiled () your `DEFINE_GEOM` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Dynamic Mesh Zones** dialog box.
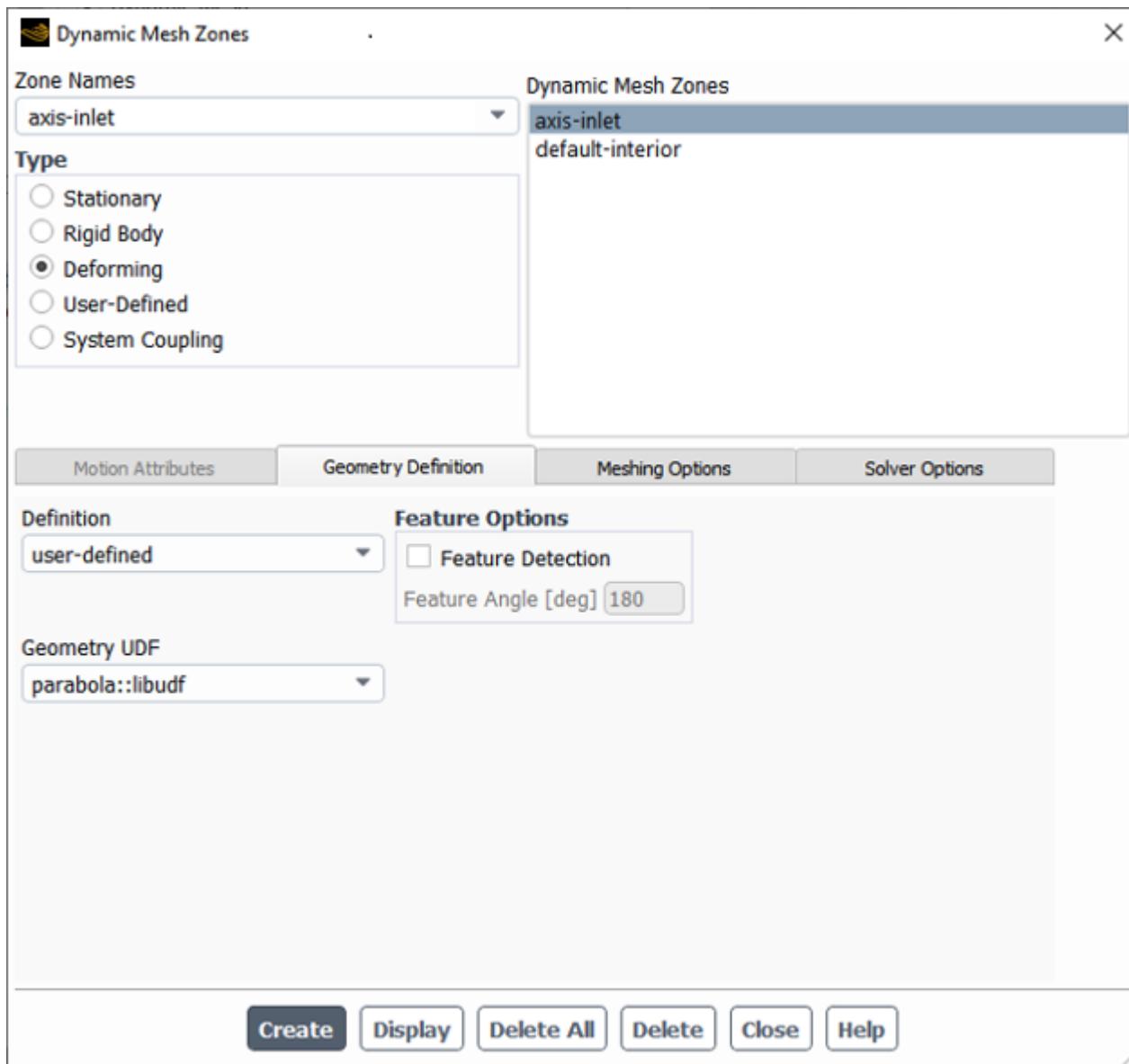
To hook the UDF to Ansys Fluent, you will first need to enable the **Dynamic Mesh** option in the **Dynamic Mesh** task page.

![icon] **Setup** → ![icon] **Dynamic Mesh** → ☑ **Dynamic Mesh**

Next, open the **Dynamic Mesh Zones** dialog box (Figure 6.113: The Dynamic Mesh Zones Dialog Box (p. 565)).

![icon] **Setup** → ![icon] **Dynamic Mesh** → **Create/Edit...**

Figure 6.113: The Dynamic Mesh Zones Dialog Box

Select **Deforming** under **Type** in the **Dynamic Mesh Zones** dialog box (Figure 6.113: The Dynamic Mesh Zones Dialog Box (p. 565)) and click the **Geometry Definition** tab. Select **user-defined** in the **Definition** drop-down list, and select the function name (for example, **parabola::libudf**) from the **Geometry UDF** drop-down list. Click **Create** and then **Close**.

See `DEFINE_GEOM` (p. 270) for details about `DEFINE_GEOM` functions.

6.5.4. Hooking `DEFINE_GRID_MOTION` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_GRID_MOTION` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Dynamic Mesh Zones** dialog box (Figure 6.114: Dynamic Mesh Zones Dialog Box (p. 566)).
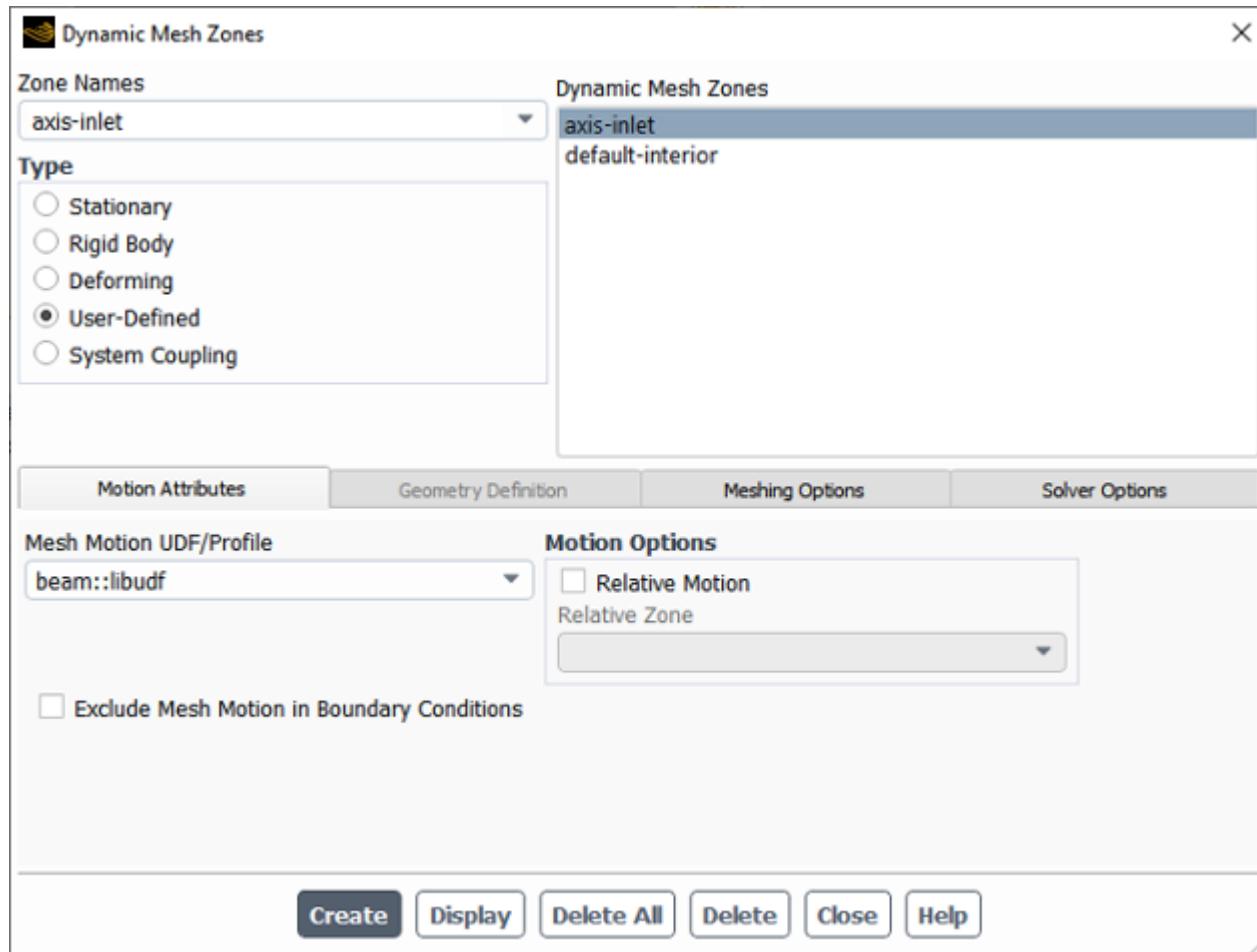
To hook the UDF to Ansys Fluent, you will first need to enable the **Dynamic Mesh** option in the **Dynamic Mesh** task page.

 Setup → ⬦ **Dynamic Mesh** → ☑ **Dynamic Mesh**

Next, open the **Dynamic Mesh Zones** dialog box.

 Setup → ⬦ **Dynamic Mesh** → **Create/Edit...**

Figure 6.114: Dynamic Mesh Zones Dialog Box

Select **User-Defined** under **Type** in the **Dynamic Mesh Zones** dialog box (Figure 6.114: Dynamic Mesh Zones Dialog Box (p. 566)) and click the **Motion Attributes** tab. Select the function name (for example, **beam::libudf**) from the **Mesh Motion UDF** drop-down list. Click **Create** then **Close**.

See `DEFINE_GRID_MOTION` (p. 271) for details about `DEFINE_GRID_MOTION` functions.

6.5.5. Hooking `DEFINE_SDOF_PROPERTIES` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_SDOF_PROPERTIES` UDF, the name of the function you supplied as a `DEFINE` macro argument will become visible and selectable in the **Dynamic Mesh Zones** dialog box in Ansys Fluent.

To hook the UDF to Ansys Fluent, you must first right-click the **General** branch of the tree and select **Transient** from the **Analysis Type** sub-menu.
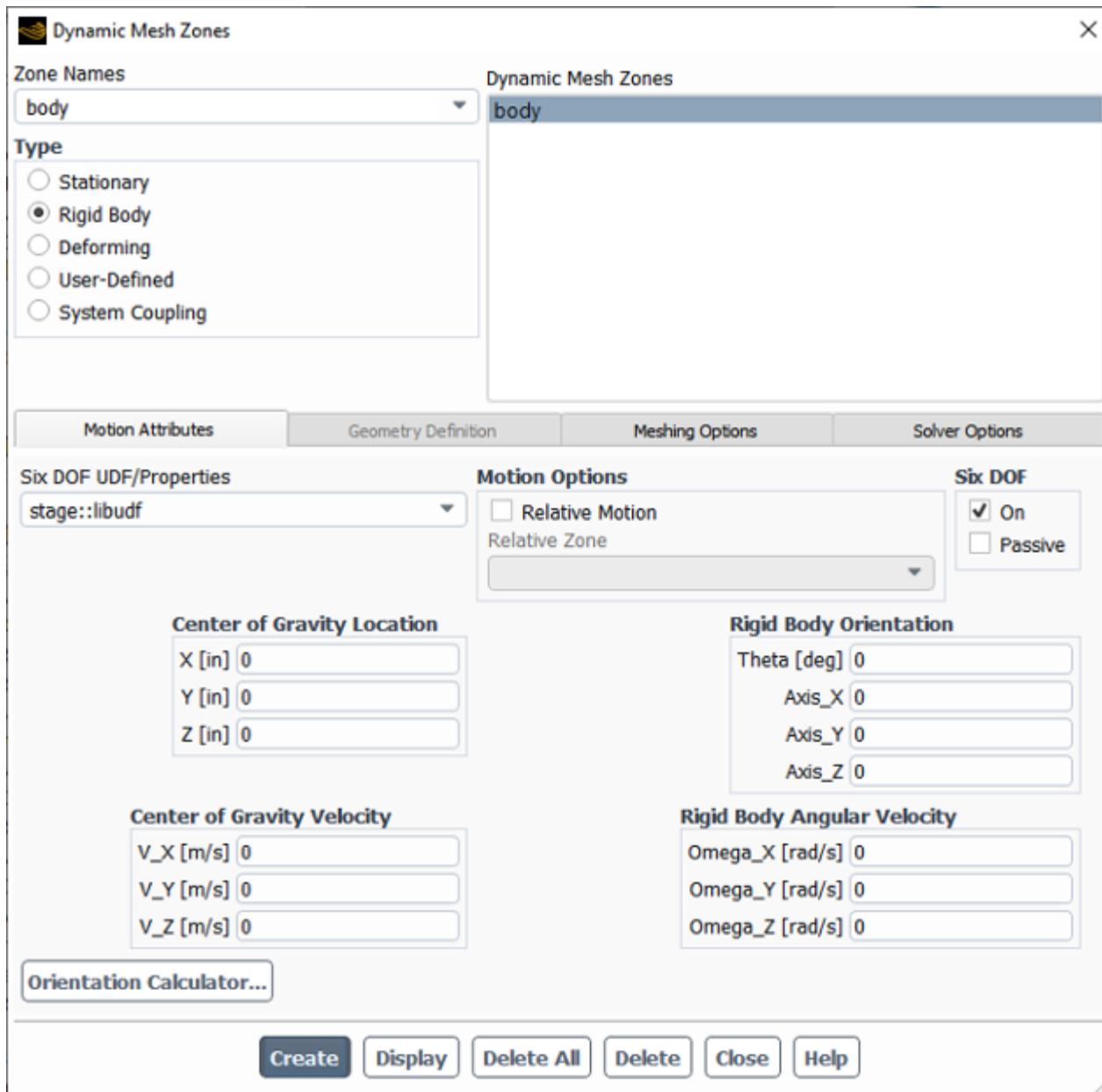
Setup → **General** **Analysis Type** → **Transient**

Next, enable the **Dynamic Mesh** option in the **Dynamic Mesh** task page.

Setup → **Dynamic Mesh** → **Dynamic Mesh**

Then, enable the **Six DOF** option in the **Options** group box, and open the **Dynamic Mesh Zones** dialog box (Figure 6.115: The Dynamic Mesh Zones Dialog Box (p. 568)).

Setup → **Dynamic Mesh** → **Create/Edit...**

Figure 6.115: The Dynamic Mesh Zones Dialog Box

Select **Rigid Body** under **Type** in the **Dynamic Mesh Zones** dialog box (Figure 6.115: The Dynamic Mesh Zones Dialog Box (p. 568)) and click the **Motion Attributes** tab. Make sure that the **On** option in the **Six DOF** group box is enabled, and select the function name (for example, **stage::libudf**) from the **Six DOF UDF/Properties** drop-down list. Click **Create** then **Close**.

See `DEFINE_SDOF_PROPERTIES` (p. 274) for details about `DEFINE_SDOF_PROPERTIES` functions.

6.5.6. Hooking `DEFINE_CONTACT` UDFs

After you have compiled (Compiling UDFs (p. 399)) your `DEFINE_CONTACT` UDF, the name of the argument you supplied as the first `DEFINE` macro argument will become visible and selectable in the **Contact UDF** drop-down box of the **Contact Detection** tab in the **Options** dialog box of Ansys Fluent.

To hook the UDF to Ansys Fluent, you will must first right-click the **General** branch of the tree and select **Transient** from the **Analysis Type** sub-menu.
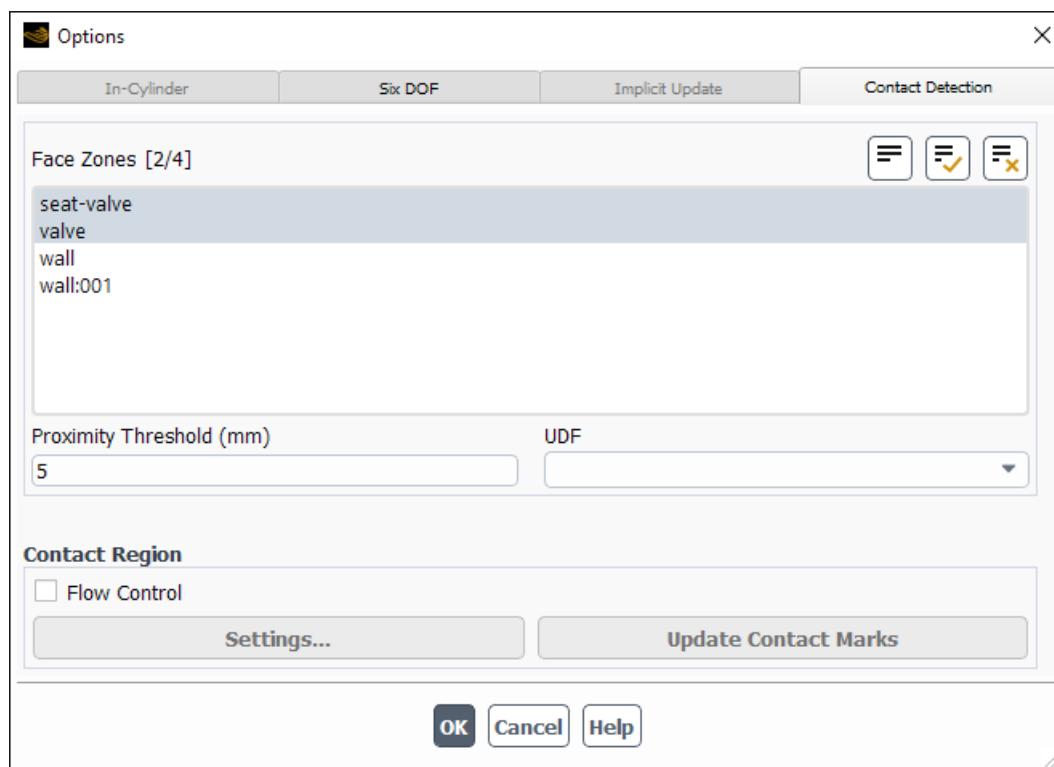
 Setup → **General** **Analysis Type** → **Transient**

Next, enable the **Dynamic Mesh** option in the **Dynamic Mesh** task page.

 Setup → **Dynamic Mesh** → **Dynamic Mesh**

Then, select the **Contact Detection** check box in the **Options** group box, and click the **Settings...** button to open the **Options** dialog box (Figure 6.116: The Options Dialog Box Showing the Contact Detection Tab (p. 569)).

Figure 6.116: The Options Dialog Box Showing the Contact Detection Tab

Select the function name (for example, **contact_props::libudf**) from the **UDF** drop-down list.

See `DEFINE_CONTACT` (p. 278) for details about `DEFINE_CONTACT` functions.

6.6. Hooking User-Defined Scalar (UDS) Transport Equation UDFs

This section contains methods for hooking anisotropic diffusion coefficient, flux, and unsteady UDFs for scalar equations that have been defined using `DEFINE` macros described in User-Defined Scalar (UDS) Transport Equation `DEFINE` Macros (p. 295) and interpreted or compiled using methods described in Interpreting UDFs (p. 393) or Compiling UDFs (p. 399), respectively. See Hooking `DEFINE_PROFILE` UDFs (p. 478), Hooking `DEFINE_SOURCE` UDFs (p. 495), and Hooking `DEFINE_DIFFUSIVITY` UDFs (p. 442) to hook scalar source term, profile, or isotropic diffusion coefficient UDFs.

For more information, see the following sections:

6.6.1. Hooking `DEFINE_ANISOTROPIC_DIFFUSIVITY` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_ANISOTROPIC_DIFFUSIVITY` UDF, the name of the function you supplied as the first `DEFINE` macro argument will become visible and selectable in Ansys Fluent.

To hook the UDF to Ansys Fluent, you will first need to open the **User-Defined Scalars** dialog box.
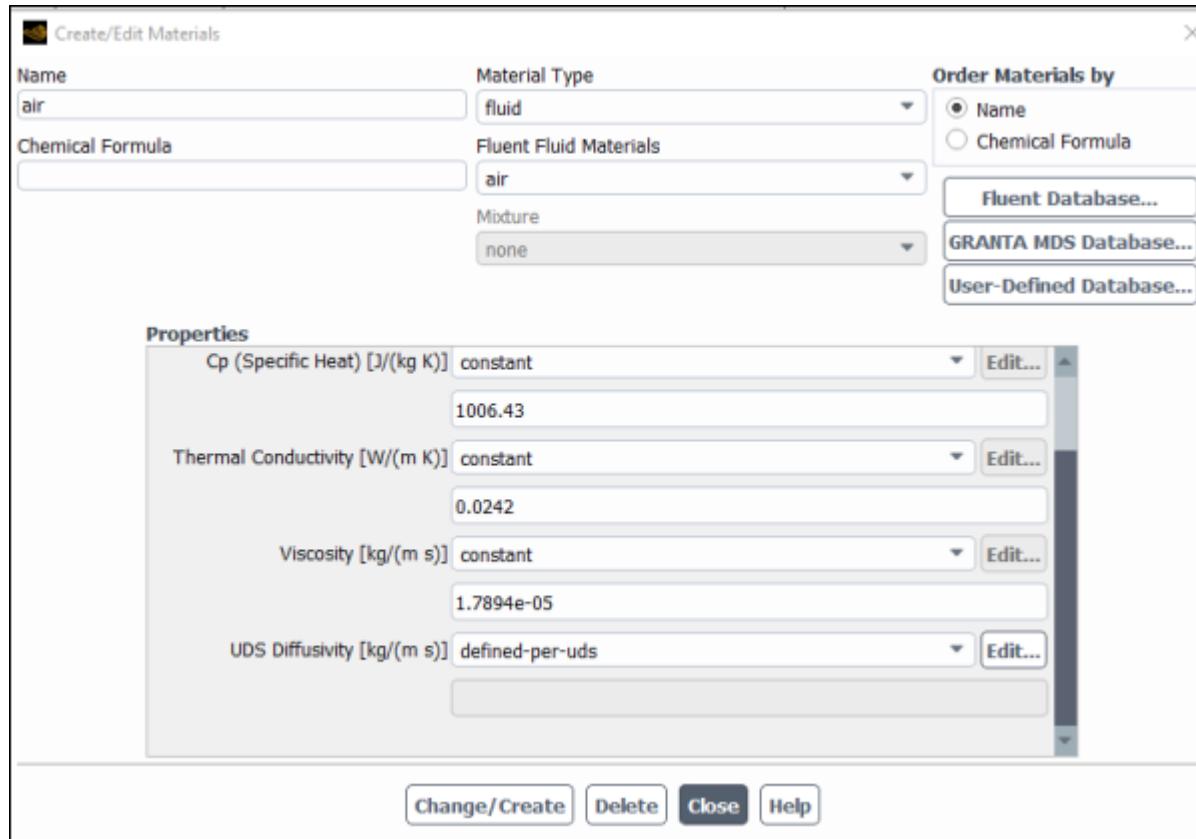
 Parameters & Customization → **User Defined Scalars** **Edit...**

In the **User-Defined Scalars** dialog box, specify the **Number of User-Defined Scalars** (for example, **2**) and click **OK**. Next, open the **Materials** task page.

 Setup → **Materials**
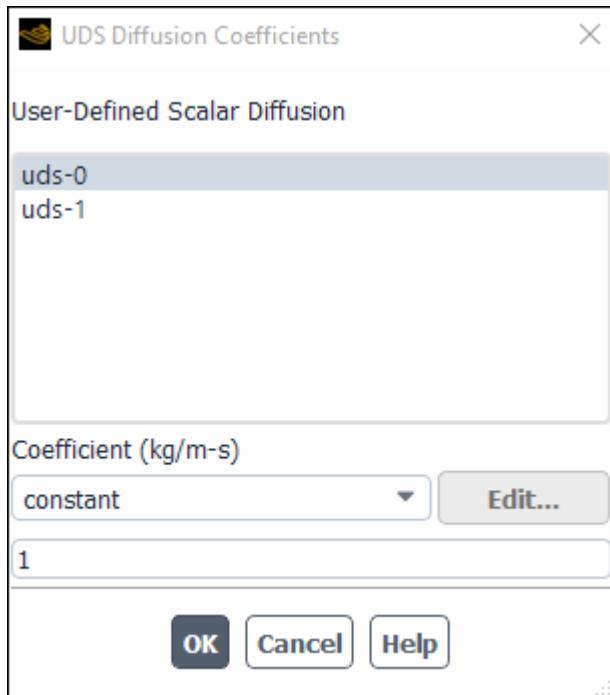
Select one of the materials in the **Materials** list and click **Create/Edit...** to open the **Create/Edit Materials** dialog box (Figure 6.117: The Create/Edit Materials Dialog Box (p. 570)).

Figure 6.117: The Create/Edit Materials Dialog Box

Scroll down the **Properties** group box in the **Create/Edit Materials** dialog box (Figure 6.117: The Create/Edit Materials Dialog Box (p. 570)), and select **defined-per-uds** from the **UDS Diffusivity** drop-down list. This will open the **UDS Diffusion Coefficients** dialog box (Figure 6.118: The UDS Diffusion Coefficients Dialog Box (p. 571)).

Figure 6.118: The UDS Diffusion Coefficients Dialog Box

In the **UDS Diffusion Coefficients** dialog box, select a scalar equation (for example, **uds-0**) and select **user-defined-anisotropic** from the drop-down list under **Coefficient**. This will open the **User-Defined Functions** dialog box. Select the name of the UDF (for example, **cyl_ortho_diff::libudf**) you want to hook, and click **OK**. The name of the UDF will be displayed in the field below the **Coefficient** drop-down list in the **UDS Diffusion Coefficients** dialog box. Click **OK**, and then click **Change/Create** in the **Create/Edit Materials** dialog box.
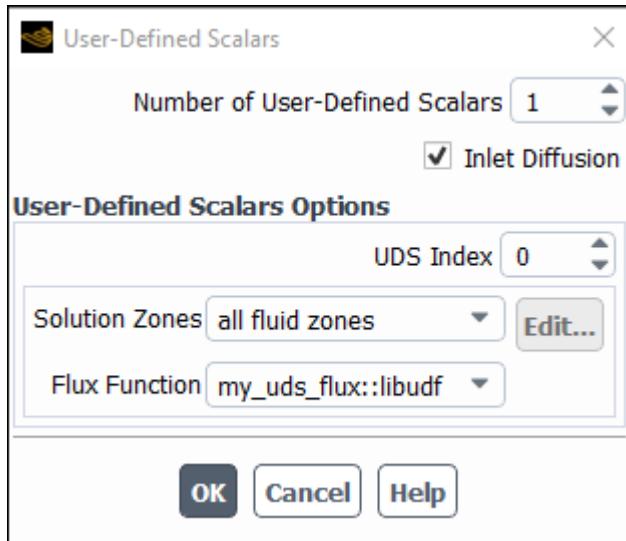
Note that you can hook a unique diffusion coefficient UDF for each scalar transport equation you have defined in your model.

See `DEFINE_ANISOTROPIC_DIFFUSIVITY` (p. 297) for details about defining `DEFINE_ANISO-TROPIC_DIFFUSIVITY` UDFs and the User's Guide for general information about UDS anisotropic diffusivity.

6.6.2. Hooking `DEFINE_UDS_FLUX` UDFs

After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your `DEFINE_UDS_FLUX` UDF, the name of the argument that you supplied as the first `DEFINE` macro argument will become visible and selectable in the **User-Defined Scalars** dialog box (Figure 6.119: The User-Defined Scalars Dialog Box (p. 572)) in Ansys Fluent.

 Parameters & Customization → **User Defined Scalars** **Edit...**

Figure 6.119: The User-Defined Scalars Dialog Box

To hook the UDF to Ansys Fluent, first specify the **Number of User-Defined Scalars** (for example, **2**) in the **User-Defined Scalars** dialog box (Figure 6.119: The User-Defined Scalars Dialog Box (p. 572)). As you enter the number of user-defined scalars, the dialog box will expand to show the **User-Defined Scalars Options** group box. Next, for each scalar you have defined, increment the **UDS Index** and select the **Solution Zones** (for example, **all fluid zones**) and the name of the function (for example, **my_uds_flux::libudf**) from the **Flux Function** drop-down list, and click **OK**.

> **Note:**
>
> Only fluid zones should be selected from the **Solution Zones** drop-down list, as
> DEFINE_UDS_FLUX UDFs cannot be used to calculate convective flux in solid zones.

6.6.3. Hooking `DEFINE_UDS_UNSTEADY` UDFs
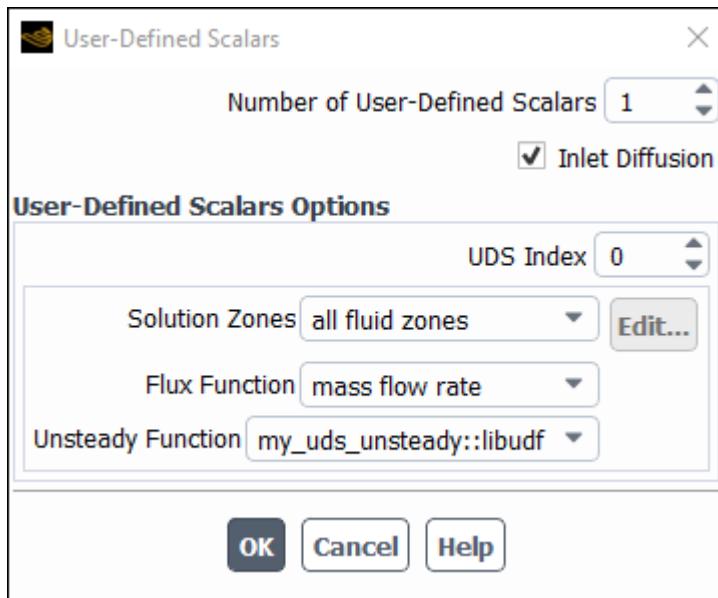
After you have interpreted (Interpreting UDFs (p. 393)) or compiled (Compiling UDFs (p. 399)) your DEFINE_UDS_UNSTEADY UDF, the name of the argument that you supplied as the first DEFINE macro argument will become visible and selectable in the **User-Defined Scalars** dialog box in Ansys Fluent.

> **Important:**
>
> Make sure that you have selected **Transient** from the **Time** list in the **Solver** group box
> of the **General** task page.

To hook the UDF to Ansys Fluent, first open the **User-Defined Scalars** dialog box.

Parameters & Customization → **User Defined Scalars** Edit...

Figure 6.120: The User-Defined Scalars Dialog Box

In the **User-Defined Scalars** dialog box (Figure 6.120: The User-Defined Scalars Dialog Box (p. 573)), specify the **Number of User-Defined Scalars** (for example, **2**) in the **User-Defined Scalars** dialog box (Figure 6.120: The User-Defined Scalars Dialog Box (p. 573)). As you enter the number of user-defined scalars, the dialog box will expand to show the **User-Defined Scalars Options** group box. Next, for each scalar you have defined, increment the **UDS Index** and select the **Zone Type** and the **Flux Function**. Then select the name of your UDF (for example, **my_uds_unsteady::libudf**) from the **Unsteady Function** drop-down list, and click **OK**.

6.7. Hooking Battery Model UDFs
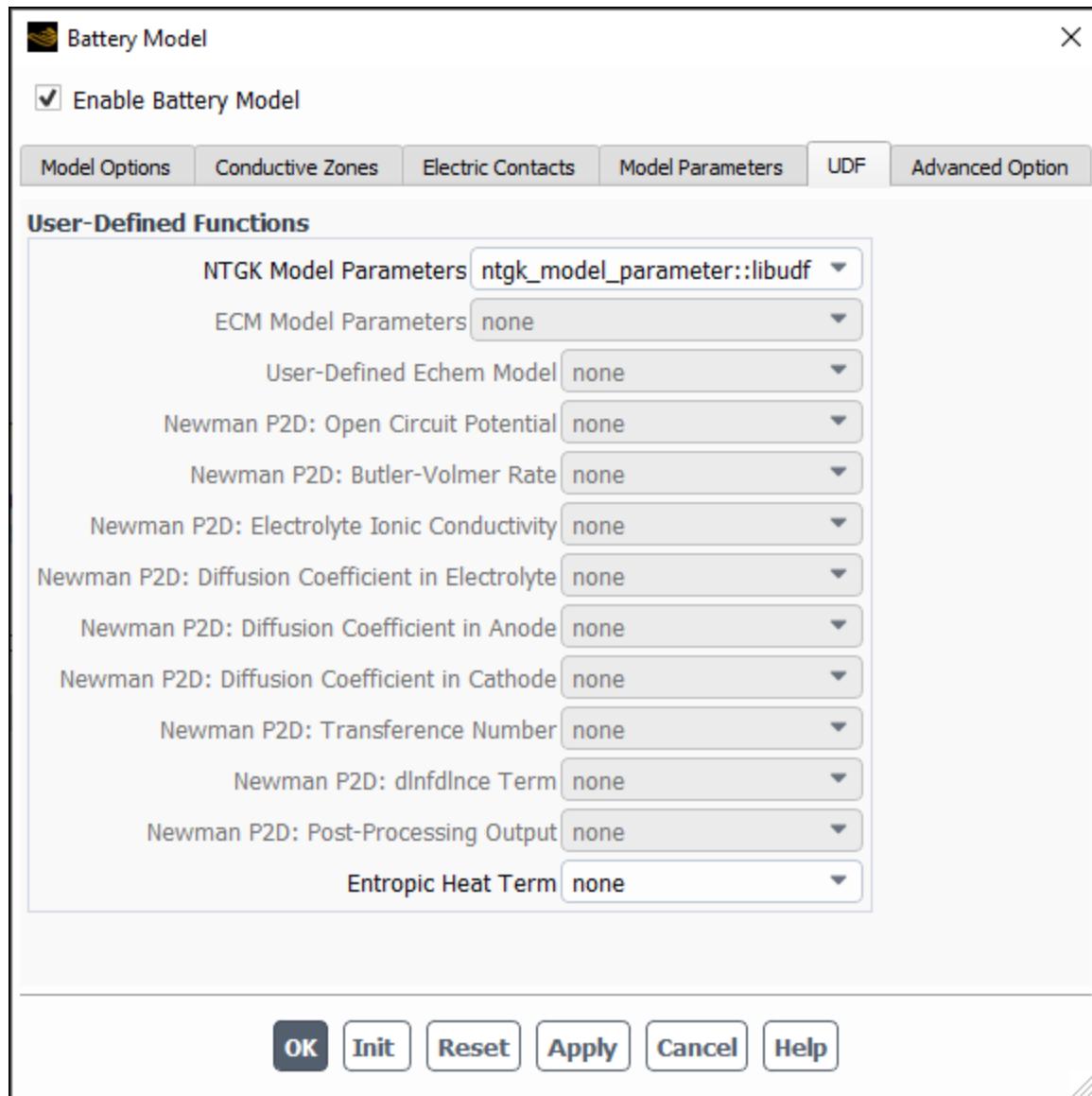
After you have compiled (Compiling UDFs (p. 399)) your MSMD battery model UDFs, the names of the functions you supplied as the first `DEFINE` macro argument will become visible and selectable in Ansys Fluent.

> **Note:**
>
> All the battery model UDFs can only be compiled and cannot be run in the interpreted mode.

To hook your UDFs to Ansys Fluent:

1. In the **Battery Model** dialog box, go to the **UDF** tab.

Figure 6.121: The Battery Model Dialog Box - UDF Tab

2. In the **User-Defined Functions** group box, select the function name (for example, **ntgk_model_para-meter::libudf**) from the appropriate drop-down list located to the right of each of the battery model UDFs (for example, **NTGK Model Parameters**).

3. Click **OK**.

See for details about MSMD battery model UDFs.
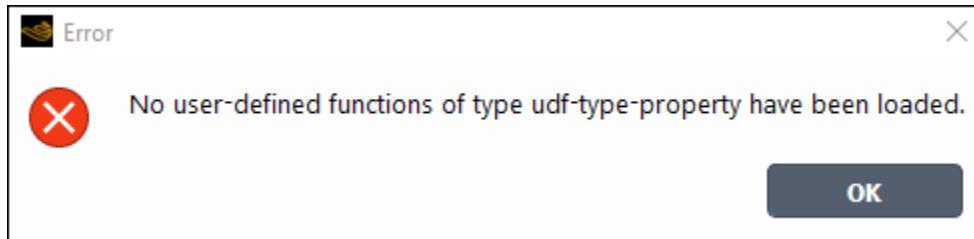
6.8. Common Errors While Hooking a UDF to Ansys Fluent

In some cases, if you select **user-defined** as an option in a graphics dialog box but have not previously interpreted or compiled/loaded a UDF, you will get an error message.

In other graphics dialog boxes, the **user-defined** option will become visible as an option for a parameter only *after* you have interpreted or compiled the UDF. After you have interpreted or compiled the UDF, you can then select **user-defined** option and the list of interpreted and compiled/loaded UDFs will be displayed.

If you inadvertently hook a UDF to the wrong parameter in an Ansys Fluent graphics dialog box (for example, profile UDF for a material property), you will either get a real-time error message, or when you go to initialize or iterate the solution, Ansys Fluent will report an error in the dialog box (Figure 6.122: The Error Dialog (p. 575)).

Figure 6.122: The Error Dialog

A message will also be reported to the console (and log file):

```
 Error: No user-defined functions of type udf-type-property have been loaded.
Error Object: #f
```

Chapter 7: Parallel Considerations

This chapter contains an overview of user-defined functions (UDFs) for parallel Ansys Fluent and their usage. Details about parallel UDF functionality can be found in the following sections:

7.1. Overview of Parallel Ansys Fluent

Ansys Fluent's parallel solver computes a solution to a large problem by simultaneously using multiple processes that may be executed on the same machine, or on different machines in a network. It does this by splitting up the computational domain into multiple partitions (Figure 7.1: Partitioned Mesh in Parallel Ansys Fluent (p. 577)) and assigning each data partition to a different compute process, referred to as a compute node (Figure 7.2: Partitioned Mesh Distributed Between Two Compute Nodes (p. 578)). Each compute node executes the same program on its own data set, simultaneously, with every other compute node. The host process, or simply the host, does not contain mesh cells, faces, or nodes (except when using the DPM shared-memory model). Its primary purpose is to interpret commands from Cortex (the Ansys Fluent process responsible for user-interface and graphics-related functions) and in turn, to pass those commands (and data) to a compute node which distributes it to the other compute nodes.

Figure 7.1: Partitioned Mesh in Parallel Ansys Fluent
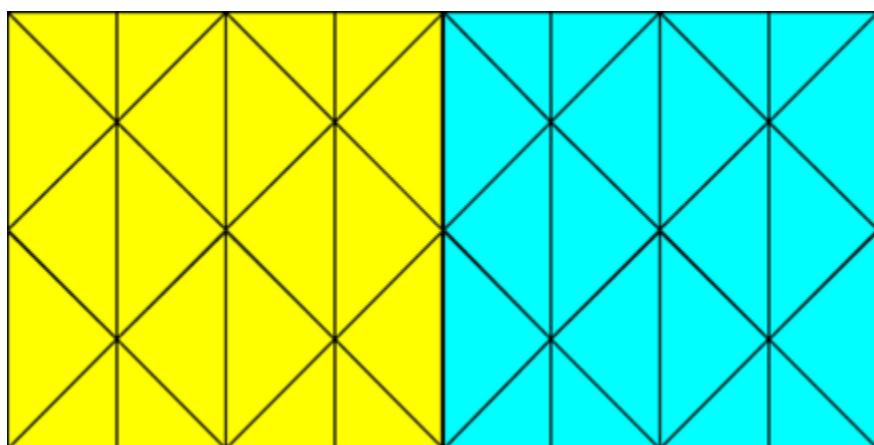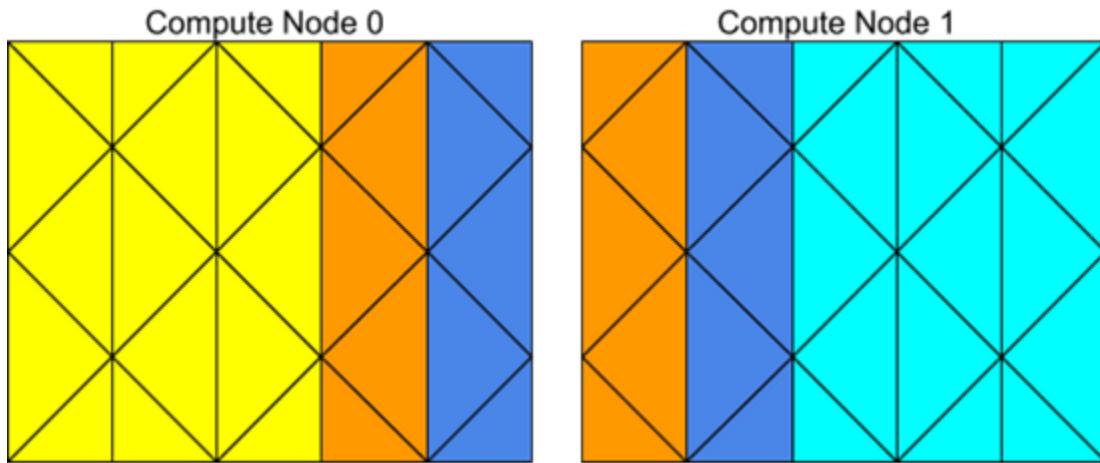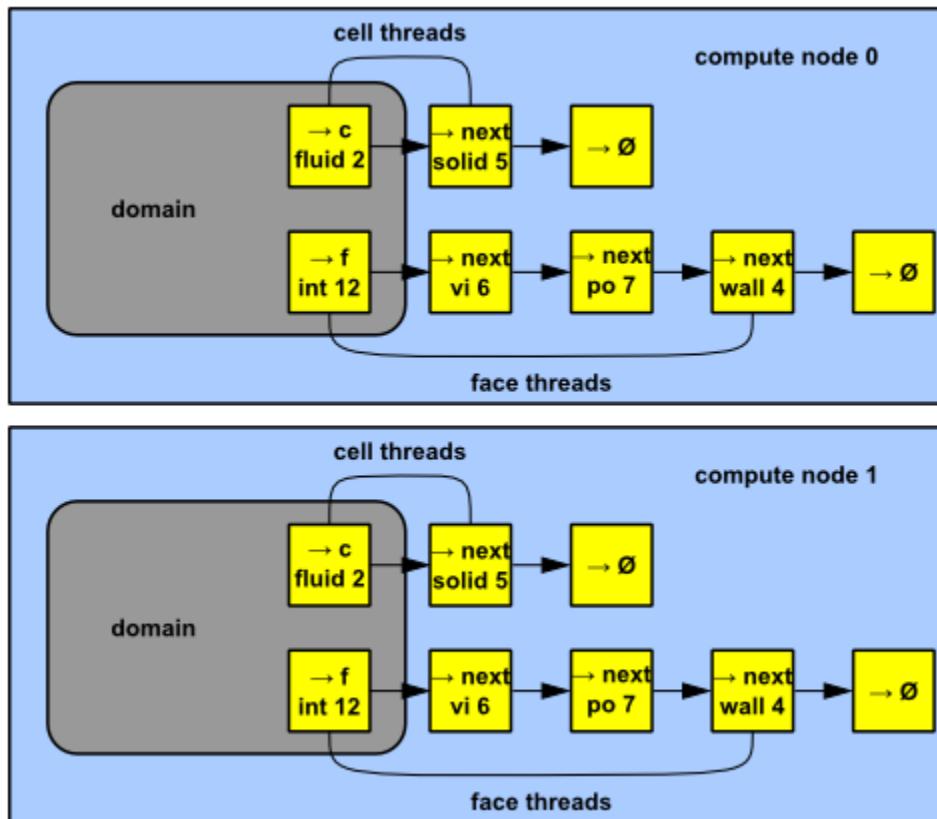
Figure 7.2: Partitioned Mesh Distributed Between Two Compute Nodes

Compute nodes store and perform computations on their portion of the mesh while a single layer of overlapping cells along partition boundaries provides communication and continuity across the partition boundaries (Figure 7.2: Partitioned Mesh Distributed Between Two Compute Nodes (p. 578)). Even though the cells and faces are partitioned, all of the domains and threads in a mesh are mirrored on each compute node (Figure 7.3: Domain and Thread Mirroring in a Distributed Mesh (p. 578)). The threads are stored as linked lists as in the serial solver. The compute nodes can be implemented on a massively parallel computer, a multiple-CPU workstation, or a network of workstations using the same or different operating systems.

Figure 7.3: Domain and Thread Mirroring in a Distributed Mesh

For more information, see the following section:

7.1.1. Command Transfer and Communication

The processes that are involved in an Ansys Fluent session are defined by Cortex, a host process, and a set of *n* compute node processes (referred to as compute nodes), with compute nodes being labeled from `0` to `n-1` (Figure 7.4: Ansys Fluent Architecture (p. 580)). The host receives commands from Cortex and passes commands to compute node-0. Compute node-0, in turn, sends commands to all additional compute nodes. All compute nodes (except 0) receive commands from compute node-0. Before the compute nodes pass messages to the host (via compute node-0), they synchronize with each other. Figure 7.4: Ansys Fluent Architecture (p. 580) shows the relationship of processes in Ansys Fluent.

Each compute node is "virtually" connected to every other compute node and relies on its "communicator" to perform such functions as sending and receiving arrays, synchronizing, performing global reductions (such as summations over all cells), and establishing machine connectivity. An Ansys Fluent communicator is a message-passing library. For example, it could be a vendor implementation of the Message Passing Interface (MPI) standard, as depicted in Figure 7.4: Ansys Fluent Architecture (p. 580).

All of the Ansys Fluent processes (including the host process) are identified by a unique integer ID. The host process is assigned the ID `host(=999999)`. The host collects messages from compute node-0 and performs operation (such as printing, displaying messages, and writing to a file) on all of the data (see Figure 7.5: Example of Command Transfer in Ansys Fluent (p. 581)).

Figure 7.4: Ansys Fluent Architecture
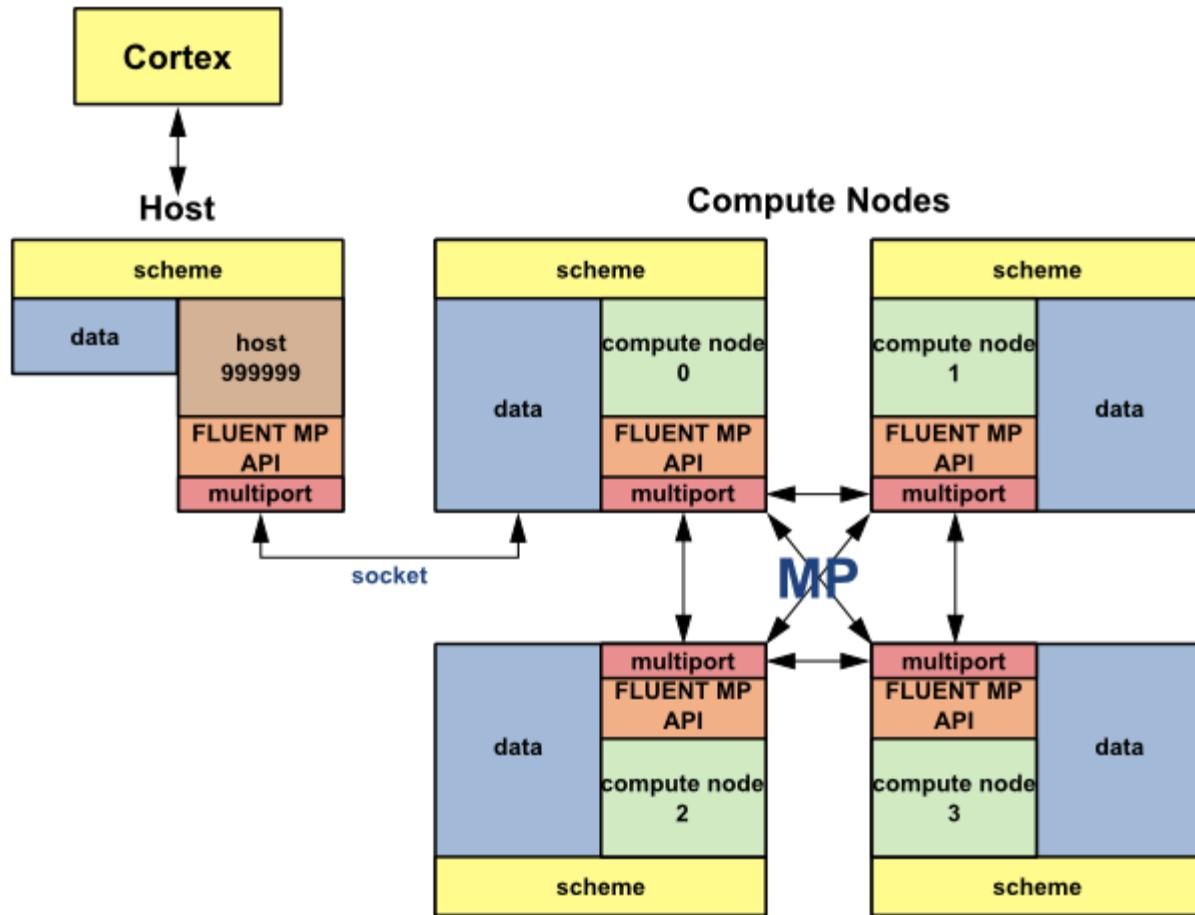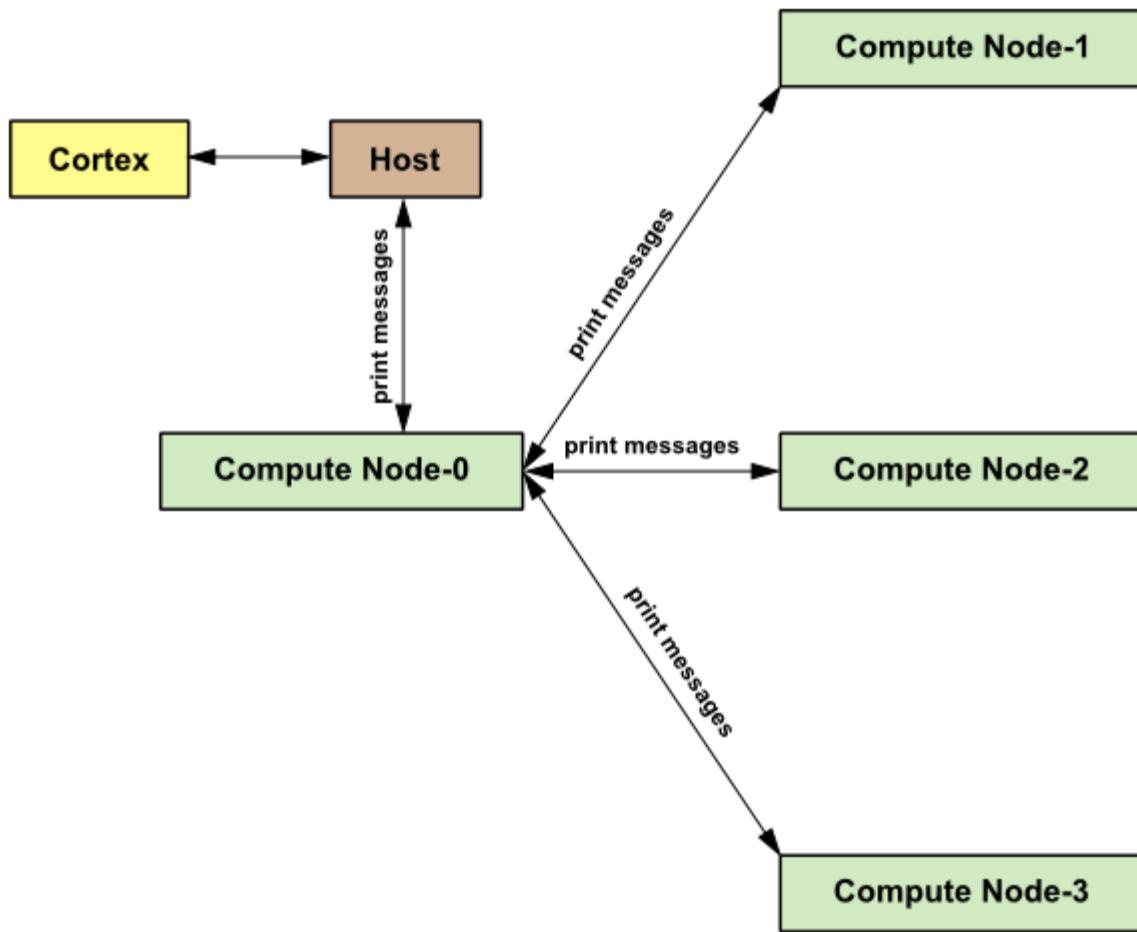
Figure 7.5: Example of Command Transfer in Ansys Fluent

7.2. Cells and Faces in a Partitioned Mesh

Some terminology must be introduced to distinguish between different types of cells and faces in a partitioned mesh. Note that this nomenclature applies only to parallel coding in Ansys Fluent.

7.2.1. Cell Types in a Partitioned Mesh

7.2.2. Faces at Partition Boundaries

7.2.3. PRINCIPAL_FACE_P

7.2.4. Exterior Thread Storage
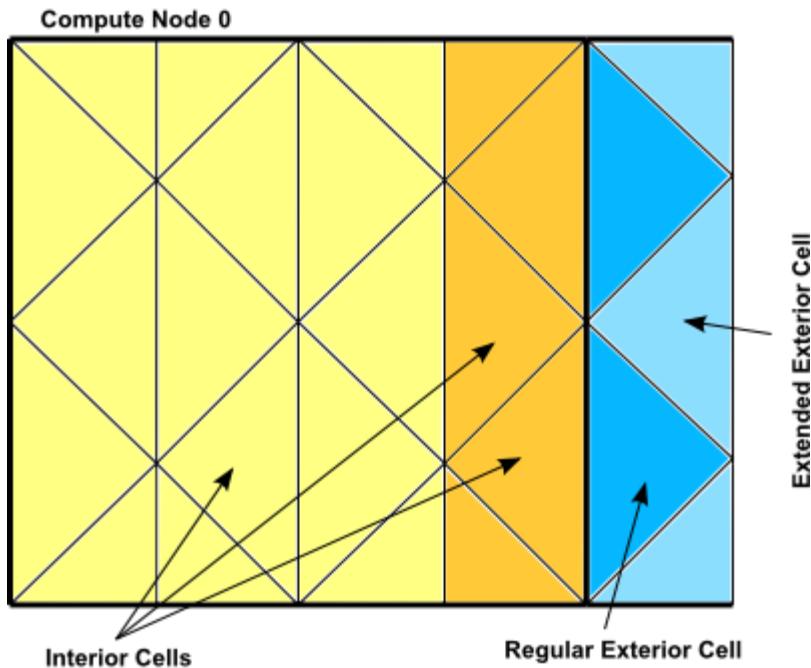
7.2.5. Extended Neighborhood

7.2.1. Cell Types in a Partitioned Mesh

There are broadly two types of cells in a partitioned mesh: *interior cells* and *exterior cells* (Figure 7.6: Partitioned Mesh: Cells (p. 582)). Interior cells are fully contained within a mesh partition. Exterior cells to a partition are not contained within that mesh partition but are connected to a node on its interface with one or more neighboring partitions. If an exterior cell shares a face with an interior cell then it is referred to as a regular exterior cell. If an exterior cell shares only an edge or a node with an interior cell then it is referred to as an extended exterior cell. Exterior cells on one compute node correspond to the same interior cells in the adjacent compute node. (Figure 7.2: Partitioned Mesh Distributed
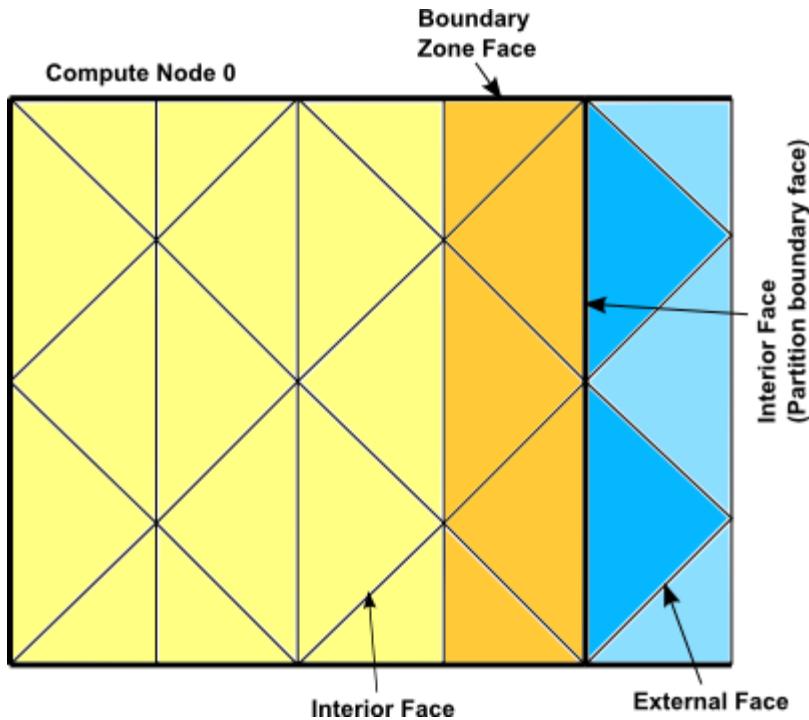
Between Two Compute Nodes (p. 578)). This duplication of cells at a partition boundary becomes important when you want to loop over cells in a parallel mesh. There are separate macros for looping over interior cells, exterior cells, and all cells. See Looping Macros (p. 595) for details.

Figure 7.6: Partitioned Mesh: Cells

7.2.2. Faces at Partition Boundaries

There are three classifications of faces in a partitioned mesh: *interior*, *boundary zone*, and *external* (Figure 7.7: Partitioned Mesh: Faces (p. 583)). Interior faces have two neighboring cells. Interior faces that lie on a partition boundary are referred to as "partition boundary faces." Boundary zone faces lie on a physical mesh boundary and have only one adjacent cell neighbor. External faces are non-partition boundary faces that belong to exterior cells. External faces are generally not used in parallel UDFs and, therefore, will not be discussed here.

Figure 7.7: Partitioned Mesh: Faces

Note that each partition boundary face is duplicated on adjacent compute nodes (Figure 7.2: Partitioned Mesh Distributed Between Two Compute Nodes (p. 578)). This is necessary so that each compute node can calculate its own face values. However, this duplication can result in face data being counted twice when UDFs are involved in operations that involve summing data in a thread that contains partition boundary faces. For example, if your UDF sums data over all of the faces in a mesh, then as each node loops over its faces, duplicated partition boundary faces can be counted twice. For this reason, one compute node in every adjacent set is assigned by Ansys Fluent as the "principal" compute node, with respect to partition boundary faces. In other words, although each face can appear on one or two partitions, it can only "officially" belong to one of them. The Boolean macro `PRINCIPAL_FACE_P(f,t)` returns `TRUE` if the face `f` is a principal face on the current compute node.

7.2.3. `PRINCIPAL_FACE_P`

You can use `PRINCIPAL_FACE_P` to test whether a given face is the principal face, before including it in a face loop summation. In the sample source code below, the area of a face is added to the total area only if it is the principal face.

> **Important:**
>
> `PRINCIPAL_FACE_P` can be used *only* in compiled UDFs.

Example

```
begin_f_loop(f,t)
 if PRINCIPAL_FACE_P(f,t) /* tests if the face is the principal face
              FOR COMPILED UDFs ONLY */
 {
   F_AREA(area,f,t);  /* computes area of each face */
   total_area +=NV_MAG(area); /* computes total face area by
                accumulating magnitude of each
```
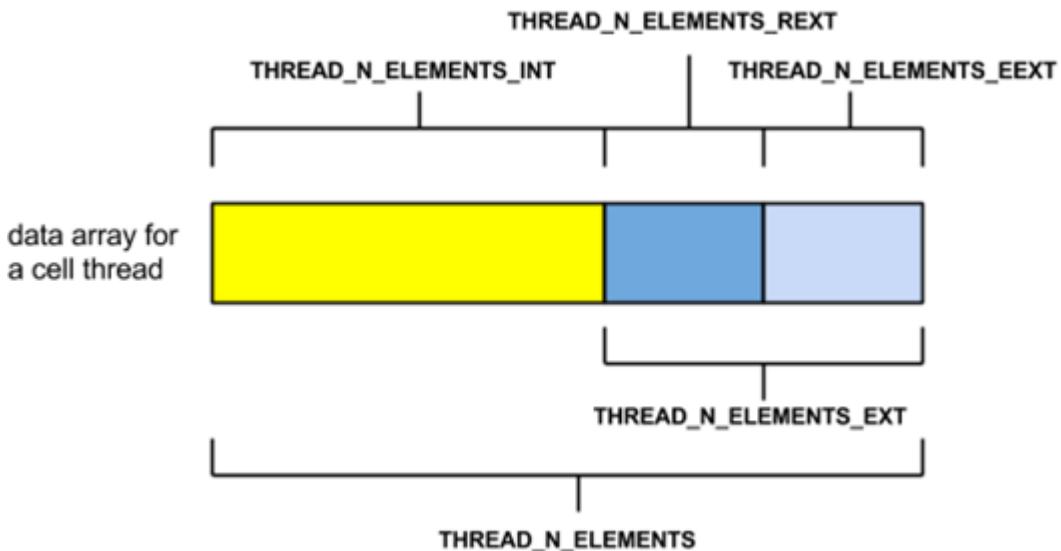
```
                    face's area */
  }
  end_f_loop(f,t)
```

7.2.4. Exterior Thread Storage

Each thread stores the data associated with its cells or faces in a set of arrays. For example, pressure is stored in an array and the pressure for cell `c` is obtained by accessing element `c` of that array. Storage for exterior cell and face data occurs at the end of every thread data array. For a cell thread, data for regular exterior cells occurs before that for extended exterior cells, as shown in .

Figure 7.8: Exterior Thread Data Storage at End of a Thread Array

7.2.5. Extended Neighborhood

Ansys Fluent creates a complete extended exterior cell neighborhood based on interface faces and corner nodes. This makes certain operations easier (such as mesh manipulations and solver gradient reconstructions) and enhances the performance of such operations. The solver uses only the regular neighborhood and hence solver-related data is filled only for the regular exterior cells and not for the extended exterior cells.

Figure 7.9: Regular Neighborhood Includes the Dark Blue Triangles (Connected to the Partition Interface Faces)
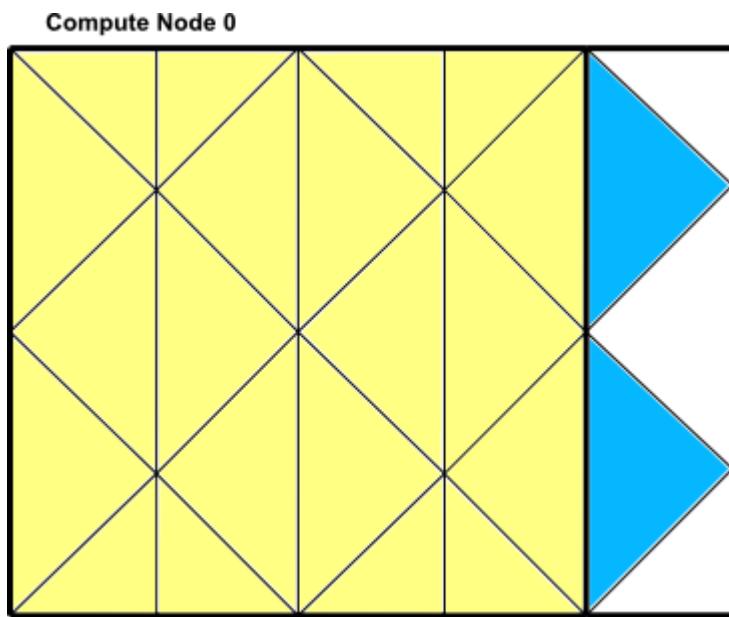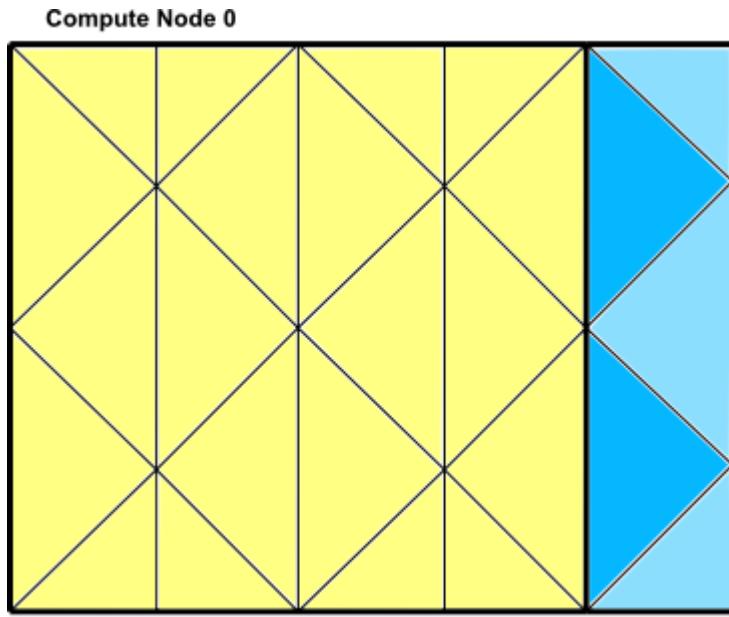
Figure 7.10: Extended Neighborhood Includes Both the Dark Blue and the Light Blue Triangles (Connected to the Partition Interface Nodes)

7.3. Parallelizing Your Serial UDF

Ansys Fluent's solver contains three types of executable: Cortex, host, and compute node (or simply "node" for short). When Ansys Fluent runs, an instance of Cortex starts, followed by one host and n compute nodes, thereby giving a total of $n+2$ running processes. For this reason, it is necessary when running in parallel (and recommended when running in serial) that you make sure that your function will successfully execute as a host and a node process. At first it may appear that you should write two

different versions of your UDF: one for host, and one for node. Good programming practice, however, would suggest that you write a single UDF that, when compiled, can execute on the two versions. This process is referred to in this manual as "parallelizing" a serial UDF (that is, a UDF that does not account for the separate processes). You can do this by adding special macros as well as compiler directives to your UDF, as described below. Compiler directives, (such as `#if RP_NODE`, `RP_HOST`) and their negated forms, direct the compiler to include only portions of the function that apply to a particular process, and ignore the rest (see Compiler Directives (p. 587)).

A general rule of thumb is that your serial UDF must be "parallelized" if it performs an operation that is dependent on sending or receiving data from another compute node (or the host) or uses a macro type that was introduced after Ansys Fluent version 18.2 (which are not guaranteed to be supported in serial UDFs). Some types of operations that require parallelization of serial source code when used with multiple nodes include the following:

- Reading and Writing Files

- Global Reductions

- Global Sums

- Global Minimums and Maximums

- Global Logicals

- Certain Loops over Cells and Faces

- Displaying Messages on a Console

- Printing to a Host or Node Process

After the source code for your "parallelized" UDF has been written, it can be compiled using the same methods for serial UDFs. Instructions for compiling UDFs can be found in Compiling UDFs (p. 399).

7.3.1. Parallelization of Discrete Phase Model (DPM) UDFs

The DPM model can be used for the following parallel options:

- Shared Memory

- Message Passing

When you are using a DPM-specific UDF (see Discrete Phase Model (DPM) `DEFINE` Macros (p. 201)), it will be executed on the machine that is in charge of the considered particle, based on the above-mentioned parallel options. Since all fluid variables needed for DPM models are held in data structures of the tracked particles, no special care is needed when using DPM UDFs in parallel Ansys Fluent with the following two exceptions.

Firstly, when you are writing to a sampling output file using the `DEFINE_DPM_OUTPUT` macro, you are not allowed to use the C function `fprintf`. Instead new functions `par_fprintf` and `par_fprintf_head` are provided to enable the parallel file writing. Each compute node writes its information to a separate temporary file. These individual files are put together and sorted into the final output file by Ansys Fluent. The new functions can be used with the same parameter lists as the C function `fprintf` with the stipulation that the sorting of the files by Ansys Fluent requires the

specification of an extended parameter list. For details on the use of these macros refer to The par_fprintf_head and par_fprintf Functions (p. 386) and DEFINE_DPM_OUTPUT (p. 227).

The second exception arises when storing information about particles. In the case of parallel simulations, you must use particle-specific user variables as they can be accessed with the macros TP_USER_REAL(tp, i)(tp being of the type Tracked_Particle *) and PP_USER_REAL(p, i)(p being of the type Particle *). Only this information is carried with the particles across partition boundaries, while other local or global variables are not carried across partition boundaries.

Note that if you need to access other data, such as cell values, then for the parallel options except Shared Memory you will have access to all fluid and solver variables. When you choose the Shared Memory option, however, you will have access only to the variables defined in the macros SV_DPM_LIST and SV_DPMS_LIST. These macros are defined in the file dpm.h.

7.3.2. Macros for Parallel UDFs

This section contains macros that you can use to parallelize your serial UDF. Where applicable, definitions for these macros can be found in the referenced header file (such as para.h).

7.3.2.1. Compiler Directives

When converting a UDF to run in parallel, some parts of the function may need to be done by the host and some by the compute nodes. This distinction is made when the UDF is compiled. By using Ansys Fluent-provided compiler directives, you can specify portions of your function to be assigned to the host or to the compute nodes. The UDF that you write will be written as a single file for the host and node versions, but different parts of the function will be compiled to generate different versions of the dynamically linked shared object file libudf.so (libudf.dll on Windows). Print tasks, for example, may be assigned exclusively to the host, while a task such as computing the total volume of a complete mesh will be assigned to the compute nodes. Since most operations are executed by either the host or compute nodes, negated forms of compiler directives are more commonly used.

Note that the primary purpose of the host is to interpret commands from Cortex and to pass those commands (and data) to compute node-0 for distribution. Since the host does not contain mesh data, you will need to be careful not to include the host in any calculations that could, for example result in a division by zero. In this case, you will need to direct the compiler to ignore the host when it is performing mesh-related calculations, by wrapping those operations around the #if !RP_HOST directive. For example, suppose that your UDF will compute the total area of a face

thread, and then use that total area to compute a flux. If you do not exclude the host from these operations, the total area on the host will be zero and a floating point exception will occur when the function attempts to divide by zero to obtain the flux.

Example

```
#if !RP_HOST avg_pres = total_pres_a / total_area; /* if you do not exclude the host
    this operation will result in a division by zero and error!
   Remember that host has no data so its total will be zero.*/
 #endif
```

You will need to use the `#if !RP_NODE` directive when you want to exclude compute nodes from operations for which they do not have data.

Below is a list of parallel compiler directives and what they do:

```
/********************************************************************/
 /*  Compiler Directives                                          */
 /********************************************************************/

 #if RP_HOST
    /* only host process is involved */
 #endif

 #if RP_NODE
    /* only compute nodes are involved */
 #endif


 /********************************************************************/
 /*  Negated forms that are more commonly used                    */
 /********************************************************************/

 #if !RP_HOST
    /* only compute nodes are involved */
 #endif

 #if !RP_NODE
    /* only host process is involved */
 #endif
```

The following simple UDF shows the use of compiler directives. The adjust function is used to define a function called `where_am_i`. This function queries to determine which type of process is executing and then displays a message on that computed node's monitor.

Example

```
/*****************************************************
  Simple UDF that uses compiler directives
 *****************************************************/
#include "udf.h"
DEFINE_ADJUST(where_am_i, domain)
{
#if RP_HOST
    Message("I am in the host process\n");
#endif /* RP_HOST */

#if RP_NODE
    Message("I am in the node process with ID %d\n",myid);
    /* myid is a global variable which is set to the multiport ID for
    each node */
#endif /* RP_NODE */
}
```

This simple allocation of functionality between the different types of processes is useful in a limited number of practical situations. For example, you may want to display a message on the compute nodes when a particular computation is being run (by using RP_NODE or !RP_HOST). Or, you can also choose to designate the host process to display messages (by using RP_HOST or !RP_NODE). Usually you want messages written only once by the host process. Simple messages such as "Running the Adjust Function" are straightforward. Alternatively, you may want to collect data from all the nodes and print the total once, from the host. To perform this type of operation your UDF will need some form of communication between processes. The most common mode of communication is between the host and the node processes.

7.3.2.2. Communicating Between the Host and Node Processes

There are two sets of similar macros that can be used to send data between the host and the compute nodes: host_to_node_type_num and node_to_host_type_num.

7.3.2.2.1. Host-to-Node Data Transfer

To send data from the host process to *all* the node processes (indirectly via compute node-0), macros of the following form is used:

```
host_to_node_type_num(val_1,val_2,...,val_num);
```

where 'num' is the number of variables that will be passed in the argument list and 'type' is the data type of the variables that will be passed. The maximum number of variables that can be passed is 7. Arrays and strings can also be passed from host to nodes, one at a time, as shown in the examples below.

For information about transferring a file from the host to a node, see Reading Files in Parallel (p. 608).

Examples

```
/* integer and real variables passed from host to nodes */
 host_to_node_int_1(count);
 host_to_node_real_7(len1, len2, width1, width2, breadth1, breadth2, vol);

/* string and array variables passed from host to nodes */
 char wall_name[]="wall-17";
 int thread_ids[10] = {1,29,5,32,18,2,55,21,72,14};

 host_to_node_string(wall_name,8); /* remember terminating NUL character */
 host_to_node_int(thread_ids,10);
```

Note that these host_to_node communication macros do not need to be "protected" by compiler directives for parallel UDFs, because all of these macros automatically do the following:

• send the variable value if compiled as the host version

• receive and then set the local variable if compiled as a compute node version

The most common use for this set of macros is to pass parameters or boundary conditions from the host to the nodes processes. See the example UDF in Parallel UDF Example (p. 606) for a demonstration of usage.

7.3.2.2.2. Node-to-Host Data Transfer

To send data from compute node-0 to the host process, use macros of the form:

```
node_to_host_type_num(val_1,val_2,...,val_num);
```

where 'num' is the number of variables that will be passed in the argument list and 'type' is the data type of the variables that will be passed. The maximum number of variables that can be passed is 7. Arrays and strings can also be passed from host to nodes, one at a time, as shown in the examples below.

Note that unlike the `host_to_node` macros, which pass data from the host process to *all* of the compute nodes (indirectly via compute node-0), `node_to_host` macros pass data *only* from compute node-0 to the host.

Examples

```
/* integer and real variables passed from compute node-0 to host */
 node_to_host_int_1(count);
 node_to_host_real_7(len1, len2, width1, width2, breadth1, breadth2, vol);

 /* string and array variables passed from compute node-0 to host */
 char *string;
 int string_length;
 real vel[ND_ND];

 node_to_host_string(string,string_length);
 node_to_host_real(vel,ND_ND);
```

`node_to_host` macros do not need to be protected by compiler directives (such as `#if RP_NODE`) since they automatically do the following:

- send the variable value if the node is compute node-0 and the function is compiled as a node version

- do nothing if the function is compiled as a node version, but the node is not compute node-0

- receive and set variables if the function is compiled as the host version

The most common usage for this set of macros is to pass global reduction results from compute node-0 to the host process. In cases where the value that is to be passed is computed by all of the compute nodes, there must be some sort of collection (such as a summation) of the data from all the compute nodes onto compute node-0 before the single collected (summed) value can be sent. Refer to the example UDF in Parallel UDF Example (p. 606) for a demonstration of usage and Global Reduction Macros (p. 591) for a full list of global reduction operations.

7.3.2.3. Predicates

There are a number of macros available in parallel Ansys Fluent that expand to logical tests. These logical macros, referred to as "predicates", are denoted by the suffix `P` and can be used as test conditions in your UDF. The following predicates return `TRUE` if the condition in the parenthesis is met.

```
/* predicate definitions from para.h header file */
```

```
# define MULTIPLE_COMPUTE_NODE_P (compute_node_count > 1)
# define ONE_COMPUTE_NODE_P (compute_node_count == 1)
# define ZERO_COMPUTE_NODE_P (compute_node_count == 0)
```

There are a number of predicates that allow you to test the identity of the node process in your UDF, using the compute node ID. A compute node's ID is stored as the global integer variable `myid` (see Process Identification (p. 605)). Each of the macros listed below tests certain conditions of `myid` for a process. For example, the predicate `I_AM_NODE_ZERO_P` compares the value of `myid` with the compute node-0 ID and returns `TRUE` when they are the same. `I_AM_NODE_SAME_P(n)`, on the other hand, compares the compute node ID that is passed in `n` with `myid`. When the two IDs are the same, the function returns `TRUE`. Node ID predicates are often used in conditional-if statements in UDFs.

```
/* predicate definitions from para.h header file */

# define I_AM_NODE_HOST_P (myid == host)
# define I_AM_NODE_ZERO_P (myid == node_zero)
# define I_AM_NODE_ONE_P (myid == node_one)
# define I_AM_NODE_LAST_P (myid == node_last)
# define I_AM_NODE_SAME_P(n) (myid == (n))
# define I_AM_NODE_LESS_P(n) (myid < (n))
# define I_AM_NODE_MORE_P(n) (myid > (n))
```

Recall that from Cells and Faces in a Partitioned Mesh (p. 581), a face may appear in one or two partitions but in order that summation operations do not count it twice, it is officially allocated to only one of the partitions. The tests above are used with the neighboring cell's partition ID to determine if it belongs to the current partition. The convention that is used is that the smaller-numbered compute node is assigned as the "principal" compute node for that face. `PRINCIPAL_FACE_P` returns `TRUE` if the face is located on its principal compute node. The macro can be used as a test condition when you want to perform a global sum on faces and some of the faces are partition boundary faces. Below is the definition of `PRINCIPAL_FACE_P` from `para.h`. See Cells and Faces in a Partitioned Mesh (p. 581) for more information about `PRINCIPAL_FACE_P`.

```
/* predicate definitions from para.h header file */
# define PRINCIPAL_FACE_P(f,t) (!TWO_CELL_FACE_P(f,t) || \
  PRINCIPAL_TWO_CELL_FACE_P(f,t))

# define PRINCIPAL_TWO_CELL_FACE_P(f,t) \
  (!(I_AM_NODE_MORE_P(C_PART(F_C0(f,t),THREAD_T0(t))) || \
  I_AM_NODE_MORE_P(C_PART(F_C1(f,t),THREAD_T1(t))))))
```

7.3.2.4. Global Reduction Macros

Global reduction operations are those that collect data from all of the compute nodes, and reduce the data to a single value, or an array of values. These include operations such as global summations, global maximums and minimums, and global logicals. These macros begin with the prefix `PRF_G` and are defined in `prf.h`. Global summation macros are identified by the suffix `SUM`, global maximums by `HIGH`, and global minimums by `LOW`. The suffixes `AND` and `OR` identify global logicals.

The variable data types for each macro are identified in the macro name, where `R` denotes real data types, `I` denotes integers, and `L` denotes logicals. For example, the macro `PRF_GISUM` finds the summation of integers over the compute nodes.

Each of the global reduction macros discussed in the following sections has two different versions: one takes a single variable argument, while the other takes a variable array. Macros with a `1` (one) appended to the end of the name take one argument, and return a single variable as the global

reduction result. For example, the macro `PRF_GIHIGH1(x)` expands to a function that takes one argument `x` and computes the maximum of the variable `x` among all of the compute nodes, and returns it. The result can then be assigned to another variable (such as `y`), as shown in the following example.

Example: Global Reduction Variable Macro

```
{
  int y;
  int x = myid;
  y = PRF_GIHIGH1(x); /* y now contains the same number (compute_node_count
        - 1) on all the nodes */
  }
```

Macros *without* a `1` suffix, on the other hand, compute global reduction variable arrays. These macros take three arguments: `x`, `N`, and `iwork` where `x` is an array, `N` is the number of elements in the array, and `iwork` is an array that is of the same type and size as `x` which is needed for temporary storage. Macros of this type are passed an array `x` and the elements of array `x` are filled with the new result after returning from the function. For example, the macro `PRF_GIHIGH(x,N,iwork)` expands to a function that computes the maximum of each element of the array `x` over all the compute nodes, uses the array `iwork` for temporary storage, and modifies array `x` by replacing each element with its resulting global maximum. The function does not return a value.

Example: Global Reduction Variable Array Macro

```
{
  real x[N], iwork[N];
  /* The elements of x are set in the working array here and will
   have different values on each compute node.
   In this case, x[0] could be the maximum cell temperature of all
   the cells on the compute node. x[1] the maximum pressure, x[2]
   the maximum density, etc.
  */
   PRF_GRHIGH(x,N,iwork); /* The maximum value for each value over
        all the compute nodes is found here */
   /* The elements of x on each compute node now hold the same
   maximum values over all the compute nodes for temperature,
   pressure, density, etc. */
  }
```

7.3.2.4.1. Global Summations

Macros that can be used to compute global sums of variables are identified by the suffix `SUM`. `PRF_GISUM1` and `PRF_GISUM` compute the global sum of `integer` variables and `integer` variable arrays, respectively.

`PRF_GRSUM1(x)` computes the global sum of a `real` variable `x` across all compute nodes. The global sum is of type `float` when running a single precision version of Ansys Fluent and type `double` when running the double precision version. Alternatively, `PRF_GRSUM(x,N,iwork)` computes the global sum of a `float` variable array for single precision and `double` when running double precision.

Global Summations

Macro	Action
PRF_GISUM1(x)	Returns sum of integer x over all compute nodes.
PRF_GISUM(x,N,iwork)	Sets x to contain sums over all compute nodes.
PRF_GRSUM1(x)	Returns sum of x over all compute nodes; float if single precision, double if double precision.
PRF_GRSUM(x,N,iwork)	Sets x to contain sums over all compute nodes; float array if single precision, double array if double precision.

7.3.2.4.2. Global Maximums and Minimums

Macros that can be used to compute global maximums and minimums of variables are identified by the suffixes HIGH and LOW, respectively. PRF_GIHIGH1 and PRF_GIHIGH compute the global maximum of integer variables and integer variable arrays, respectively.

PRF_GRHIGH1(x) computes the global maximum of a real variable x across all compute nodes. The value of the global maximum is of type float when running the single precision version of Ansys Fluent and type double when running the double precision version.

PRF_GRHIGH(x,N,iwork) computes the global maximum of a real variable array, similar to the description of PRF_GRSUM(x,N,iwork) on the previous page. The same naming convention used for PRF_GHIGH macros applies to PRF_GLOW.

Global Maximums

Macro	Action
PRF_GIHIGH1(x)	Returns maximum of integer x over all compute nodes.
PRF_GIHIGH(x,N,iwork)	Sets x to contain maximums over all compute nodes.
PRF_GRHIGH1(x)	Returns maximums of x over all compute nodes; float if single precision, double if double precision.
PRF_GRHIGH(x,N,iwork)	Sets x to contain maximums over all compute nodes; float array if single precision, double array if double precision.

Global Minimums

Macro	Action
PRF_GILOW1(x)	Returns minimum of integer x over all compute nodes.
PRF_GILOW(x,N,iwork)	Sets x to contain minimums over all compute nodes.

Global Minimums

Macro	Action
PRF_GRLOW1(x)	Returns minimum of x over all compute nodes; float if single precision, double if double precision.
PRF_GRLOW(x,N,iwork)	Sets x to contain minimums over all compute nodes; float array if single precision, double array if double precision.

7.3.2.4.3. Global Logicals

Macros that can be used to compute global logical ANDs and logical ORs are identified by the suffixes AND and OR, respectively. PRF_GBOR1(x) computes the global logical OR of variable x across all compute nodes. PRF_GBOR(x,N,iwork) computes the global logical OR of variable array x. The elements of x are set to TRUE if any of the corresponding elements on the compute nodes are TRUE.

By contrast, PRF_GBOR(x) computes the global logical AND across all compute nodes and PRF_GBOR(x,N,iwork) computes the global logical AND of variable array x. The elements of x are set to TRUE if all of the corresponding elements on the compute nodes are TRUE.

The PRF_GBOR1(x) and PRF_GBAND1(x) macros expect a cxboolean argument x and return a cxboolean result value. The PRF_GBOR(x, N, iwork) and PRF_GBAND(x, N, iwork) macros expect cxboolean array (pointer) arguments x and iwork.

Global Logicals

Macro	Action
PRF_GBOR1(x)	TRUE when variable x is TRUE for *any* of the compute nodes
PRF_GBOR(x,N,work)	TRUE when *any* of the elements in variable array x is TRUE
PRF_GBOR1(x)	TRUE when variable x is TRUE for *all* compute nodes
PRF_GBOR(x,N,iwork)	TRUE when every element in variable array x is TRUE

Similar macros containing "I" instead of "B" in their names are available for the int data type.

7.3.2.4.4. Global Synchronization

PRF_GSYNC() can be used when you want to globally synchronize compute nodes before proceeding with the next operation. When you insert a PRF_GSYNC macro in your UDF, no commands beyond it will execute until the preceding commands in the source code have been completed on all of the compute nodes. Synchronization may also be useful when debugging your function.

7.3.2.5. Looping Macros

There are different looping macros for interior and exterior cells and faces available for parallel coding.

7.3.2.5.1. Looping Over Cells

A partitioned mesh in parallel Ansys Fluent is made up of interior cells and exterior cells (see Figure 7.6: Partitioned Mesh: Cells (p. 582)). There is a set of cell-looping macros you can use to loop over interior cells only, exterior cells only, or both interior and exterior cells.

7.3.2.5.2. Interior Cell Looping Macro

The macro `begin,end_c_loop_int` loops over interior cells in a partitioned mesh (Figure 7.11: Looping Over Interior Cells in a Partitioned Mesh Using `begin,end_c_loop_int` (indicated by the green cells) (p. 595)) and is identified by the suffix `int`. It contains a `begin` and `end` statement, and between these statements, operations can be performed on each of the thread's interior cells in turn. The macro is passed a cell index `c` and a cell thread pointer `tc`.

```
begin_c_loop_int(c, tc)
  {
  } end_c_loop_int(c, tc)
```

Figure 7.11: Looping Over Interior Cells in a Partitioned Mesh Using `begin,end_c_loop_int` (indicated by the green cells)

Example

```
real total_volume = 0.0;
 begin_c_loop_int(c,tc)
 {
  /* C_VOLUME gets the cell volume and accumulates it. The end
   result will be the total volume of each compute node's
   respective mesh  */
  total_volume += C_VOLUME(c,tc);
 } end_c_loop_int(c,tc)
```

7.3.2.5.3. Exterior Cell Looping Macro

There are three macros to loop over exterior cells in a partitioned mesh (Figure 7.12: Looping Over Exterior Cells in a Partitioned Mesh Using `begin,end_c_loop_[re]ext` (indicated by the green cells) (p. 596)).

- `begin, end_c_loop_rext` loops over regular exterior cells.

- `begin, end_c_loop_eext` loops over extended exterior cells.

- `begin, end_c_loop_ext` loops over both regular and extended exterior cells.
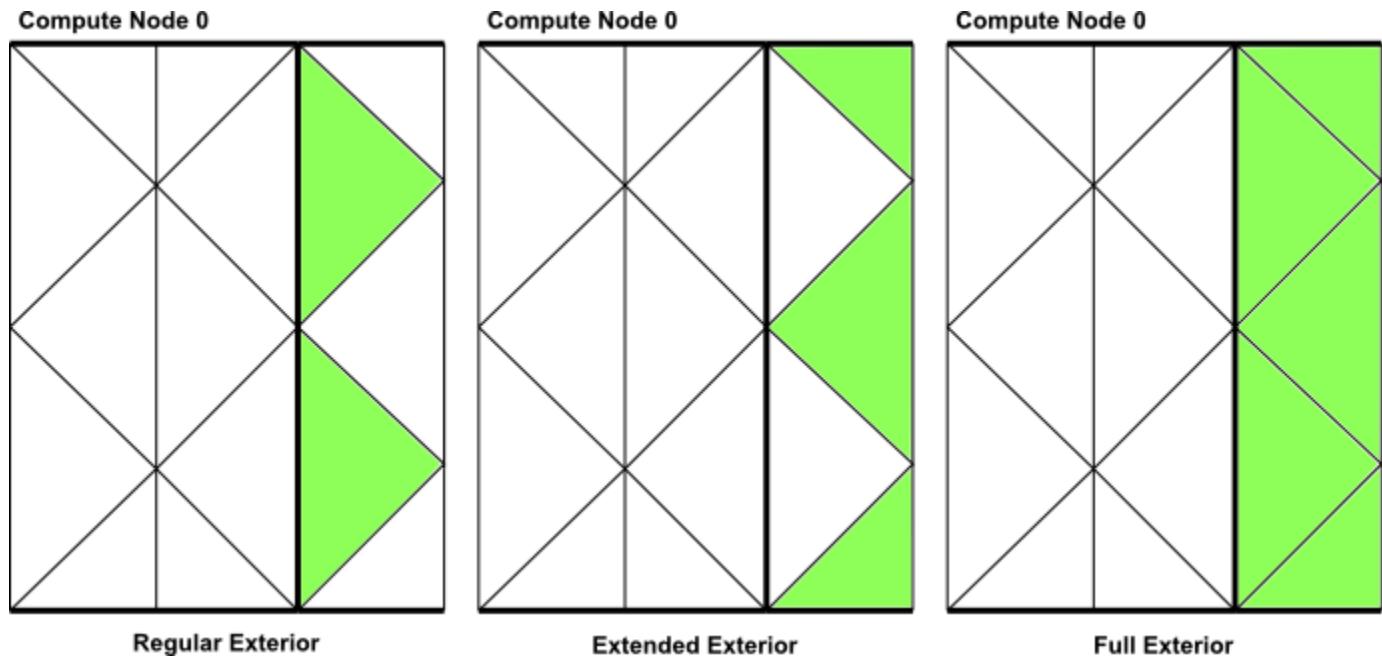
Each macro contains a `begin` and `end` statement, and between these statements, operations can be performed on each of the thread's exterior cells in turn. The macro is passed a cell index `c` and cell thread pointer `tc`. In most situations, there is no need to use the exterior cell loop macros. They are only provided for convenience if you come across a special need in your UDF.

```
begin_c_loop_ext(c, tc)
 {
  } end_c_loop_ext(c,tc)
```

Figure 7.12: Looping Over Exterior Cells in a Partitioned Mesh Using `begin,end_c_loop_[re]ext` (indicated by the green cells)

Regular Exterior   Extended Exterior   Full Exterior

7.3.2.5.4. Interior and Exterior Cell Looping Macro
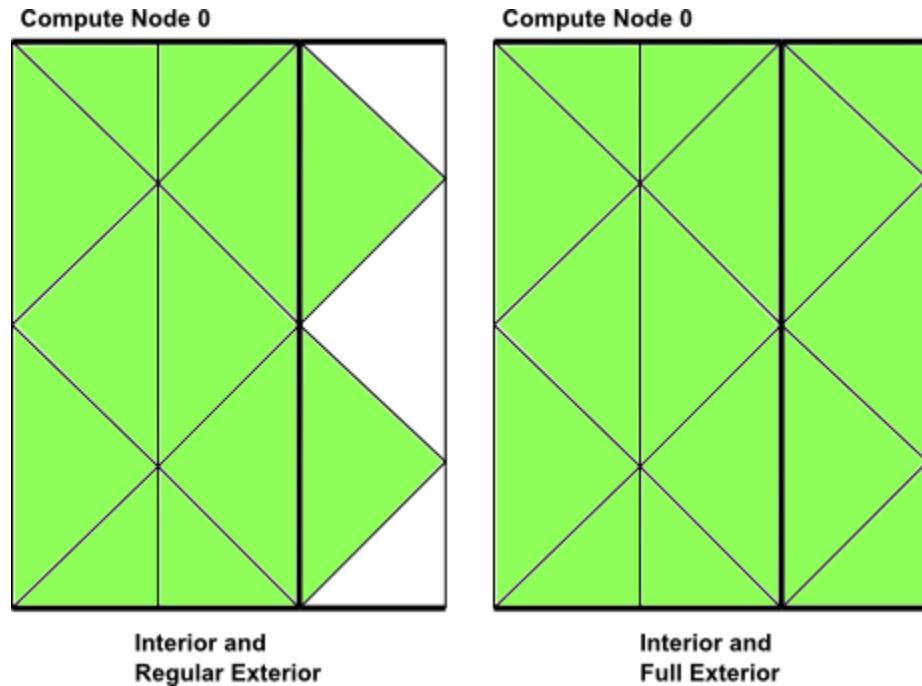
There are two macros to loop over interior and some or all exterior cells in a partitioned mesh (Figure 7.13: Looping Over Both Interior and Exterior Cells in a Partitioned Mesh Using `begin,end_c_loop_int_ext` (p. 597)).

- `begin,end_c_loop` loops over interior and regular exterior cells.

- `begin,end_c_loop_int_ext` loops over interior and all exterior cells.

Each macro contains a `begin` and `end` statement, and between these statements, operations can be performed on each of the thread's interior and exterior cells in turn. The macro is passed a cell index `c` and a cell thread pointer `tc`.

```
begin_c_loop(c, tc)
  {
  } end_c_loop(c ,tc)
```

Figure 7.13: Looping Over Both Interior and Exterior Cells in a Partitioned Mesh Using `begin,end_c_loop_int_ext`

Example

```
real temp;
 begin_c_loop(c,tc)
  {
   /* get cell temperature, compute temperature function and store
    result in user-defined memory, location index 0. */
   temp = C_T(c,tc);
   C_UDMI(c,tc,0) = (temp - tmin) / (tmax - tmin);
   /* assumes a valid tmax and tmin has already been computed */
  } end_c_loop(c,tc)
```

7.3.2.5.5. Looping Over Faces

For the purpose of discussing parallel Ansys Fluent, faces can be categorized into two types: interior faces and boundary zone faces (Figure 7.7: Partitioned Mesh: Faces (p. 583)). Partition boundary faces are interior faces that lie on the partition boundary of a compute node's mesh.

`begin`,`end_f_loop` is a face looping macro available in parallel Ansys Fluent that loops over all interior and boundary zone faces in a compute node. The macro `begin`,`end_f_loop` contains a `begin` and `end` statement, and between these statements, operations can be performed on each of the faces of the thread. The macro is passed a face index `f` and face thread pointer `tf`.

```
begin_f_loop(f, tf)
  {
  } end_f_loop(f,tf)
```

> **Important:**
>
> `begin_f_loop_int` and `begin_f_loop_ext` are looping macros that loop around interior and exterior faces in a compute node, respectively. The `_int` form is equivalent to `begin_c_loop_int`. Although these macros exist, they do not have a practical application in UDFs and should not be used.

Recall that partition boundary faces lie on the boundary between two adjacent compute nodes and are represented on both nodes. Therefore, there are some computations (such as summations) when a partition boundary face will get counted twice in a face loop. This can be corrected by testing whether the current node is a face's principal compute node inside your face looping macro, using `PRINCIPAL_FACE_P`. This is shown in the example below. See Cells and Faces in a Partitioned Mesh (p. 581) for details.

Example

```
begin_f_loop(f,tf)
/* each compute node checks whether or not it is the principal compute
 node with respect to the given face and thread  */

if PRINCIPAL_FACE_P(f,tf)
 /* face is on the principal compute node, so get the area and pressure
 vectors, and compute the total area and pressure for the thread
 from the magnitudes */
 {
F_AREA(area,f,tf);
total_area += NV_MAG(area);
total_pres_a += NV_MAG(area)*F_P(f,tf);
} end_f_loop(f,tf)

total_area = PRF_GRSUM1(total_area);
total_pres_a = PRF_GRSUM1(total_pres_a);
```

7.3.2.6. Cell and Face Partition ID Macros
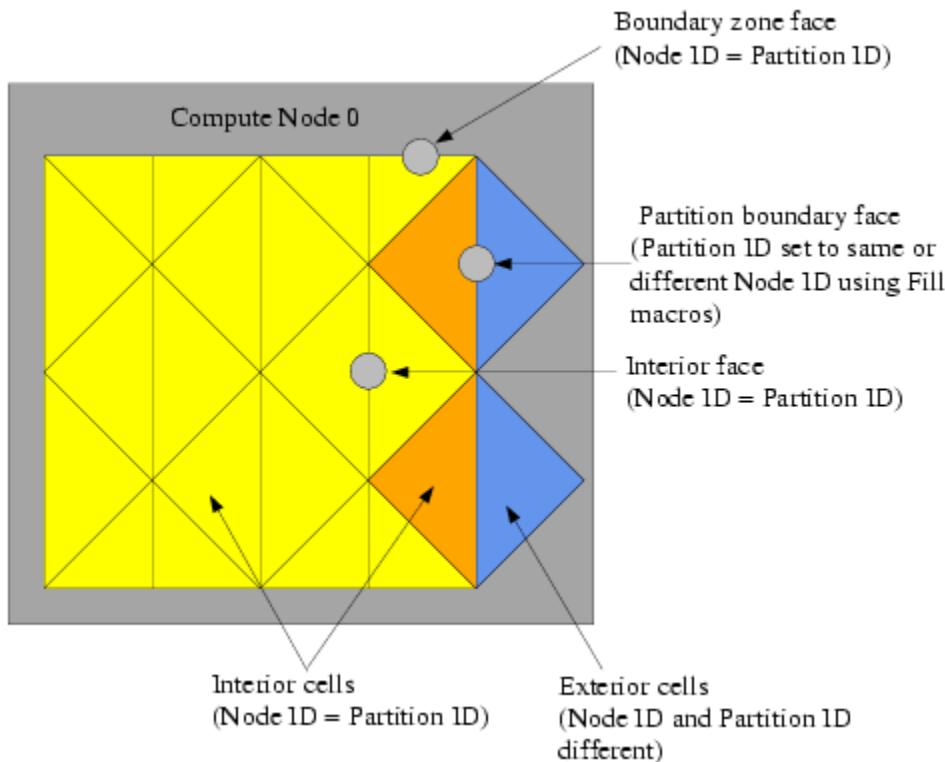
In general, cells and faces have a partition ID that is numbered from `0` to `n-1`, where `n` is the number of compute nodes. The partition IDs of cells and faces are stored in the variables `C_PART` and `F_PART`, respectively. `C_PART(c,tc)` stores the integer partition ID of a cell and `F_PART(f,tf)` stores the integer partition ID of a face.

Note that `myid` can be used in conjunction with the partition ID, since the partition ID of an exterior cell is the ID of the neighboring compute node.

7.3.2.6.1. Cell Partition IDs

For interior cells, the partition ID is the same as the compute node ID. For exterior cells, the compute node ID and the partition ID are different. For example, in a parallel system with two compute nodes (0 and 1), the exterior cells of compute node-0 have a partition ID of 1, and the exterior cells of compute node-1 have a partition ID of 0 (Figure 7.14: Partition Ids for Cells and Faces in a Compute Node (p. 599)).

Figure 7.14: Partition Ids for Cells and Faces in a Compute Node

7.3.2.6.2. Face Partition IDs

For interior faces and boundary zone faces, the partition ID is the same as the compute node ID. The partition ID of a partition boundary face, however, can be either the same as the compute node, or it can be the ID of the adjacent node, depending on what values `F_PART` is filled with (Figure 7.14: Partition Ids for Cells and Faces in a Compute Node (p. 599)). Recall that an exterior cell of a compute node has only partition boundary faces; the other faces of the cell belong to the adjacent compute node. Therefore, depending on the computation you want to do with your UDF, you may want to fill the partition boundary face with the same partition ID as the compute node (using `Fill_Face_Part_With_Same`) or with different IDs (using `Fill_Face_Part_With_Different`). Face partition IDs will need to be filled before you can access them with the `F_PART` macro. There is rarely a need for face partition IDs in parallel UDFs.

7.3.2.7. Message Displaying Macros

You can direct Ansys Fluent to display messages on a host or node process using the `Message` utility. To do this, simply use a conditional `if` statement and the appropriate compiler directive (such as `#if RP_NODE`) to select the process(es) you want the message to come from. This is demonstrated in the following example:

Example

```
#if RP_NODE
  Message("Total Area Before Summing %f\n",total\_area);
 #endif /* RP_NODE */
```

In this example, the message will be sent by the compute nodes. (It will not be sent by the host.)

`Message0` is a specialized form of the `Message` utility. `Message0` will send messages from compute node-0 only and is ignored on the other compute nodes, without having to use a compiler directive.

Example

```
/* Let Compute Node-0 display messages */

 Message0("Total volume = %f\n",total_volume);
```

7.3.2.8. Message Passing Macros

High-level communication macros of the form `node_to_host...` and `host_to_node...` that are described in Communicating Between the Host and Node Processes (p. 589) are typically used when you want to send data from the host to all of the compute nodes, or from node-0 to the host. You cannot, however, use these high-level macros when you need to pass data between compute nodes, or pass data from all of the compute nodes to compute node-0. In these cases, you can use special message passing macros described in this section.

Note that the higher-level communication macros expand to functions that perform a number of lower-level message passing operations which send sections of data as single arrays from one process to another process. These lower-level message passing macros can be easily identified in the macro name by the characters `SEND` and `RECV`. Macros that are used to send data to processes have the prefix `PRF_CSEND`, whereas macros that are used to receive data from processes have the prefix `PRF_CRECV`. Data that is to be sent or received can belong to the following data types: character (`CHAR`), integer (`INT`), `REAL` and logical (`BOOLEAN`). `BOOLEAN` variables are `TRUE` or `FALSE`. `REAL` variables are assigned as `float` data types when running a single precision version of Ansys Fluent and `double` when running double precision. Message passing macros are defined in the `prf.h` header file and are listed below.

```
/* message passing macros */

 PRF_CSEND_CHAR(to, buffer, nelem, tag)
 PRF_CRECV_CHAR (from, buffer, nelem, tag)
 PRF_CSEND_INT(to, buffer, nelem, tag)
 PRF_CRECV_INT(from, buffer, nelem, tag)
 PRF_CSEND_REAL(to, buffer, nelem, tag)
 PRF_CRECV_REAL(from, buffer, nelem, tag)
 PRF_CSEND_BOOLEAN(to, buffer, nelem, tag)
 PRF_CRECV_BOOLEAN(from, buffer, nelem, tag)
```

There are four arguments to the message-passing macros. For 'send' messages:

- `to` is the node ID of the process that data is being sent to.

- `buffer` is the name of an array of the appropriate type that will be sent.

- `nelem` is the number of elements in the array.

- `tag` is a user-defined message tag. The tag convention is to use `myid` when sending messages.

For 'receive' messages:

- `from` is the ID of the sending node.

- `buffer` is the name of an array of the appropriate type that will be received.

- `nelem` is the number of elements in the array.

- `tag` is the ID of the sending node, as the convention is to have the `tag` argument the same as the `from` argument (that is, the first argument) for receive messages.

Note that if variables that are to be sent or received are defined in your function as `real` variables, then you can use the message passing macros with the `_REAL` suffix. The compiler will then substitute `PRF_CSEND_DOUBLE` or `PRF_CRECV_DOUBLE` if you are running double precision and `PRF_CSEND_FLOAT` or `PRF_CRECV_FLOAT`, for single precision.

Because message-passing macros are low-level macros, you will need to make sure that when a message is sent from a node process, a corresponding receiving macro appears in the receiving-node process. Note that your UDF cannot directly send messages from a compute node (other than 0) to the host using message-passing macros. They can send messages indirectly to the host through compute node-0. For example, if you want your parallel UDF to send data from all of the compute nodes to the host for postprocessing purposes, the data will first have to be passed from each compute node to compute node-0, and then from compute node-0 to the host. In the case where the compute node processes send a message to compute node-0, compute node-0 must have a loop to receive the `N` messages from the `N` nodes.

Below is an example of a compiled parallel UDF that utilizes message passing macros `PRF_CSEND` and `PRF_CRECV`. Refer to the comments (*/) in the code, for details about the function.

Example: Message Passing

```
#include "udf.h"
 #define WALLID 3

 DEFINE_ON_DEMAND(face_p_list)
 {
   #if !RP_HOST /* Host will do nothing in this udf. */
      face_t f;
      Thread *tf;
      Domain *domain;
      real *p_array;
      real x[ND_ND], (*x_array)[ND_ND];
      int n_faces, i, j;

      domain=Get_Domain(1); /* Each Node will be able to access
             its part of the domain */

      tf=Lookup_Thread(domain, WALLID); /* Get the thread from the domain */
```

```
        /* The number of faces of the thread on nodes 1,2... needs to be sent
        to compute node-0 so it knows the size of the arrays to receive
        from each */

        n_faces=THREAD_N_ELEMENTS_INT(tf);

        /* No need to check for Principal Faces as this UDF
        will be used for boundary zones only */

   if(! I_AM_NODE_ZERO_P) /* Nodes 1,2... send the number of faces */
      {
       PRF_CSEND_INT(node_zero, &n_faces, 1, myid);
      }

  /* Allocating memory for arrays on each node */
  p_array=(real *)malloc(n_faces*sizeof(real));
   x_array=(real (*)[ND_ND])malloc(ND_ND*n_faces*sizeof(real));

   begin_f_loop(f, tf)
      /* Loop over interior faces in the thread, filling p_array
       with face pressure and x_array with centroid  */
      {
        p_array[f] = F_P(f, tf);
        F_CENTROID(x_array[f], f, tf);
      }
   end_f_loop(f, tf)

/* Send data from node 1,2, ... to node 0 */
Message0("\nstart\n");

   if(! I_AM_NODE_ZERO_P) /* Only SEND data from nodes 1,2... */
   {
      PRF_CSEND_REAL(node_zero, p_array, n_faces, myid);
      PRF_CSEND_REAL(node_zero, x_array[0], ND_ND*n_faces, myid);
   }
 else

   {/* Node-0 has its own data,
     so list it out first */
    Message0("\n\nList of Pressures...\n");
    for(j=0; j<n_faces; j++)
      /* n_faces is currently node-0 value */
      {
# if RP_3D
        Message0("%12.4e %12.4e %12.4e %12.4e\n",
        x_array[j][0], x_array[j][1], x_array[j][2], p_array[j]);
# else /* 2D */
        Message0("%12.4e %12.4e %12.4e\n",
        x_array[j][0], x_array[j][1], p_array[j]);
# endif
    }
 }

/* Node-0 must now RECV data from the other nodes and list that too */
   if(I_AM_NODE_ZERO_P)
   {
     compute_node_loop_not_zero(i)
      /* See para.h for definition of this loop */
     {
        PRF_CRECV_INT(i, &n_faces, 1, i);
         /* n_faces now value for node-i */
        /* Reallocate memory for arrays for node-i */
             p_array=(real *)realloc(p_array, n_faces*sizeof(real));
        x_array=(real(*)[ND_ND])realloc(x_array,ND_ND*n_faces*sizeof(real));

        /* Receive data */
        PRF_CRECV_REAL(i, p_array, n_faces, i);
        PRF_CRECV_REAL(i, x_array[0], ND_ND*n_faces, i);
        for(j=0; j<n_faces; j++)
           {
# if RP_3D
```

```
            Message0("%12.4e %12.4e %12.4e %12.4e\n",
            x_array[j][0], x_array[j][1], x_array[j][2], p_array[j]);
# else /* 2D */
            Message0("%12.4e %12.4e %12.4e\n",
            x_array[j][0], x_array[j][1], p_array[j]);
# endif
        }
      }
    }

    free(p_array); /* Each array has to be freed before function exit */
    free(x_array);

#endif /* ! RP_HOST */
}
```

7.3.2.9. Macros for Exchanging Data Between Compute Nodes

EXCHANGE_SVAR_MESSAGE, EXCHANGE_SVAR_MESSAGE_EXT, and EXCHANGE_SVAR_FACE_MES-SAGE can be used to exchange storage variables (SV_…) between compute nodes. EX-CHANGE_SVAR_MESSAGE and EXCHANGE_SVAR_MESSAGE_EXT exchange cell data between compute nodes, while EXCHANGE_SVAR_FACE_MESSAGE exchanges face data. EX-CHANGE_SVAR_MESSAGE is used to exchange data over regular exterior cells, while EX-CHANGE_SVAR_MESSAGE_EXT is used to exchange data over regular and extended exterior cells. Note that compute nodes are 'virtually' synchronized when an EXCHANGE macro is used; receiving compute nodes wait for data to be sent, before continuing.

```
/* Compute Node Exchange Macros  */

EXCHANGE_SVAR_FACE_MESSAGE(domain, (SV_P, SV_NULL));
EXCHANGE_SVAR_MESSAGE(domain, (SV_P, SV_NULL));
EXCHANGE_SVAR_MESSAGE_EXT(domain, (SV_P, SV_NULL));
```

EXCHANGE_SVAR_FACE_MESSAGE() is rarely needed in UDFs. You can exchange multiple storage variables between compute nodes. Storage variable names are separated by commas in the argument list and the list is ended by SV_NULL. For example, EXCHANGE_SVAR_MESSAGE(domain, (SV_P, SV_T, SV_NULL)) is used to exchange cell pressure and temperature variables. You can determine a storage variable name from the header file that contains the variable's definition statement. For example, suppose you want to exchange the cell pressure (C_P) with an adjacent compute node. You can look at the header file that contains the definition of C_P (mem.h) and determine that the storage variable for cell pressure is SV_P. You will need to pass the storage variable to the exchange macro.

7.3.3. Limitations of Parallel UDFs

The macro PRINCIPAL_FACE_P can be used *only* in compiled UDFs.

PRF_GRSUM1 and similar global reduction macros (Global Reduction Macros (p. 591)) cannot be used within macros such as DEFINE_SOURCE and DEFINE_PROPERTY UDFs, which are generally called for each cell (or face) and thus are called a different number of times on each compute node. As a workaround, you can use macros that are called only once on each node, such as DEFINE_ADJUST, DEFINE_ON_DEMAND, and DEFINE_EXECUTE_AT_END UDFs. For example, you could write a DEFINE_ADJUST UDF that calculates a global sum value in the adjust function, and then save the variable in user-defined memory. You can subsequently retrieve the stored variable from user-defined memory and use it inside a DEFINE_SOURCE UDF. This is demonstrated below.

In the following example, the spark volume is calculated in the DEFINE_ADJUST function and the value is stored in user-defined memory using C_UDMI. The volume is then retrieved from user-defined memory and used in the DEFINE_SOURCE UDF.

```
#include "udf.h"

/* These variables will be passed between the ADJUST and SOURCE UDFs */

static real spark_center[ND_ND] = {ND_VEC(20.0e-3, 1.0e-3, 0.0)};
static real spark_start_angle = 0.0, spark_end_angle = 0.0;
static real spark_energy_source = 0.0;
static real spark_radius = 0.0;
static real crank_angle = 0.0;

DEFINE_ADJUST(adjust, domain)
{
#if !RP_HOST

   const int FLUID_CHAMBER_ID = 2;

   real cen[ND_ND], dis[ND_ND];
   real crank_start_angle;
   real spark_duration, spark_energy;
   real spark_volume;
   real rpm;
   cell_t c;
   Thread *ct;

   rpm = RP_Get_Real("dynamesh/in-cyn/crank-rpm");
   crank_start_angle = RP_Get_Real("dynamesh/in-cyn/crank-start-angle");
   spark_start_angle = RP_Get_Real("spark/start-ca");
   spark_duration = RP_Get_Real("spark/duration");
   spark_radius = RP_Get_Real("spark/radius");
   spark_energy = RP_Get_Real("spark/energy");

   /* Set the global angle variables [deg] here for use in the SOURCE UDF */
   crank_angle = crank_start_angle + (rpm * CURRENT_TIME * 6.0);
   spark_end_angle = spark_start_angle + (rpm * spark_duration * 6.0);

   ct = Lookup_Thread(domain, FLUID_CHAMBER_ID);
   spark_volume = 0.0;

   begin_c_loop_int(c, ct)
     {
       C_CENTROID(cen, c, ct);
       NV_VV(dis,=,cen,-,spark_center);

       if (NV_MAG(dis) < spark_radius)
  {
    spark_volume += C_VOLUME(c, ct);
  }
     }
   end_c_loop_int(c, ct)

   spark_volume = PRF_GRSUM1(spark_volume);
   spark_energy_source = spark_energy/(spark_duration*spark_volume);

   Message0("\nSpark energy source = %g [W/m3].\n", spark_energy_source);
#endif
}

DEFINE_SOURCE(energy_source, c, ct, dS, eqn)
{
  /* Don't need to mark with #if !RP_HOST as DEFINE_SOURCE is only executed
     on nodes as indicated by the arguments "c" and "ct" */
  real cen[ND_ND], dis[ND_ND];

  if((crank_angle >= spark_start_angle) &&
     (crank_angle < spark_end_angle))
     {
```

```
        C_CENTROID(cen, c, ct);
        NV_VV(dis,=,cen,-,spark_center);

        if (NV_MAG(dis) < spark_radius)
  {
    return spark_energy_source;
  }
     }

  /* Cell is not in spark zone or within time of spark discharge */

  return 0.0;
}
```

Important:

Interpreted UDFs cannot be used with an Infiniband interconnect. The compiled UDF approach should be used in this case.

7.3.4. Process Identification

Each process in parallel Ansys Fluent has a unique integer identifier that is stored as the global variable `myid`. When you use `myid` in your parallel UDF, it will return the integer ID of the current compute node (including the host). The host process has an ID of `host(=999999)` and is stored as the global variable `host`. Compute node-0 has an ID of `0` and is assigned to the global variable `node_zero`. Below is a list of global variables in parallel Ansys Fluent.

Global Variables in Parallel Ansys Fluent

```
int node_zero = 0;
 int node_host = 999999;
 int node_one = 1;

 int node_last; /* returns the id of the last compute node */
 int compute_node_count; /* returns the number of compute nodes */
 int myid; /* returns the id of the current compute node (and host) */
```

`myid` is commonly used in conditional-if statements in parallel UDF code. Below is some sample code that uses the global variable `myid`. In this example, the total number of faces in a face thread is first computed by accumulation. Then, if `myid` is not compute node-0, the number of faces is passed from all of the compute nodes to compute node-0 using the message passing macro `PRF_CSEND_INT`. (See Message Passing Macros (p. 600) for details on `PRF_CSEND_INT`.)

Example: Usage of myid

```
int noface=0;
  begin_f_loop(f, tf) /* loops over faces in a face thread and
        computes number of faces */
  {
      noface++;
  }
  end_f_loop(f, tf)

 /* Pass the number of faces from node 1,2, ... to node 0 */

 #if RP_NODE if(myid!=node_zero)
  {
      PRF_CSEND_INT(node_zero, &noface, 1, myid);
  }
 #endif
```

7.3.5. Parallel UDF Example

The following is an example of a serial UDF that has been parallelized, so that it can run on any version of Ansys Fluent (host, node). Explanations for the various changes from the simple serial version are provided in the `/* comments */` and discussed below. The UDF, named `face_av`, is defined using an adjust function, computes a global sum of pressure on a specific face zone, and computes its area average.

Example: Global Summation of Pressure on a Face Zone and its Area Average Computation

```
#include "udf.h"

 DEFINE_ADJUST(face_av,domain)
 {
/* Variables used by host, node versions */
int surface_thread_id=0;
real total_area=0.0;
real total_force=0.0;

/* "Parallelized" Sections */
#if !RP_HOST  /* Compile this section for computing processes only since
          these variables are not available on the host */
    Thread* thread;
    face_t face;
    real area[ND_ND];
#endif /* !RP_HOST */


/* Get the value of the thread ID from a user-defined Scheme variable */
#if !RP_NODE
    surface_thread_id = RP_Get_Integer("pres_av/thread-id");
    Message("\nCalculating on Thread # %d\n",surface_thread_id);
#endif /* !RP_NODE */

/* To set up this user Scheme variable in cortex type */
/* (rp-var-define 'pres_av/thread-id 2 'integer #f) */
/* After set up you can change it to another thread's ID using : */
/* (rpsetvar 'pres_av/thread-id 7) */
/* Send the ID value to all the nodes */
host_to_node_int_1(surface_thread_id);

#if RP_NODE Message("\nNode %d is calculating on thread # %d\n",myid,
     surface_thread_id);
#endif /* RP_NODE */

#if !RP_HOST
/* thread is only used on compute processes */
thread = Lookup_Thread(domain,surface_thread_id);
begin_f_loop(face,thread)

  /* If this is the node to which face "officially" belongs,*/
  /* get the area vector and pressure and increment    */
  /* the total area and total force values for this node */
   if (PRINCIPAL_FACE_P(face,thread))
     {
      F_AREA(area,face,thread);
      total_area += NV_MAG(area);
      total_force += NV_MAG(area)*F_P(face,thread);
     }

 end_f_loop(face,thread)
 Message("Total Area Before Summing %f\n",total_area);
 Message("Total Normal Force Before Summing %f\n",total_force);

 # if RP_NODE /* Perform node synchronized actions here */
    total_area = PRF_GRSUM1(total_area);
    total_force = PRF_GRSUM1(total_force);
```

```
# endif /* RP_NODE */

#endif /* !RP_HOST */

/* Pass the node's total area and pressure to the Host for averaging */
node_to_host_real_2(total_area,total_force);

#if !RP_NODE
Message("Total Area After Summing: %f (m2)\n",total_area);
Message("Total Normal Force After Summing %f (N)\n",total_force);
Message("Average pressure on Surface %d is %f (Pa)\n",
    surface_thread_id,(total_force/total_area));
#endif /* !RP_NODE */
}
```

The function begins by initializing the variables `surface_thread_id`, `total_area`, and `total_force` for all processes. This is done because the variables are used by the host and node processes. The compute nodes use the variables for computation purposes and the host uses them for message-passing and displaying purposes. Next, the preprocessor is directed to compile `thread`, `face`, and `area` variables only on the node versions (and not the host), since faces and threads are only defined in the node versions of Ansys Fluent. (Note that in general, the host will ignore these statements since its face and cell data are zero, but it is good programming practice to exclude the host. See Macros for Parallel UDFs (p. 587) for details on compiler directives.)

Next, a user-defined Scheme variable named `pres_av/thread-id` is obtained by the host process using the `RP_Get_Integer` utility (see Scheme Macros (p. 382)), and is assigned to the variable `surface_thread_id`. (Note that this user-defined Scheme variable was previously set up in Cortex and assigned a value of 2 by typing the text commands shown in the comments.) After a Scheme-based variable is set up for the thread ID, it can be easily changed to another thread ID from the text interface, without the burden of modifying the source code and recompiling the UDF. Since the host communicates with Cortex and the nodes are not aware of Scheme variables, it is essential to direct the compiler to exclude the nodes from compiling them using `#if !RP_NODE`. Failure to do this will result in a compile error.

The `surface_thread_id` is then passed from the host to compute node-0 using the `host_to_node` macro. Compute node-0, in turn, automatically distributes the variable to the other compute nodes. The node processes are directed to loop over all faces in the thread associated with the `surface_thread_id`, using `#if !RP_HOST`, and compute the total area and total force. Since the host does not contain any thread data, it will ignore these statements if you do not direct the compiler, but it is good programming practice to do so. The macro `PRINCIPAL_FACE_P` is used to ensure that faces at partition boundaries are not counted twice (see Cells and Faces in a Partitioned Mesh (p. 581)). The nodes display the total area and force on the monitors (using the `Message` utility) before the global summation. `PRF_GRSUM1` (Global Reduction Macros (p. 591)) is a global summation macro that is used to compute the total area and force of all the compute nodes. These operations are directed for the compute nodes using `#if RP_NODE`.

7.4. Reading and Writing Files in Parallel

Although compute nodes can perform computations on data simultaneously when Ansys Fluent is running in parallel, when data is read from or written to a single, common file, the operations have to be sequential. The file has to be opened and read from or written to by processes that have access to the desired file system. It is often the case that the compute nodes are running on a dedicated parallel machine without disk space. This means that all of the data has to be read and/or written from the host process which always runs on a machine with access to a file system, since it reads and writes the case

and data files. This implies that unlike the example in Message Passing Macros (p. 600), where data is only passed to compute node-0 to be collated, data must now be passed from all the compute nodes to compute node-0, which then passes it on to the host node which writes it to the file. This process is known as "marshalling."

The following sections describe the processes of reading and writing files in parallel in more detail:

7.4.1. Reading Files in Parallel

To copy a file from the host to nodes, before reading it within parallel UDFs, use the following function:

```
host_to_node_sync_file(const char* filename);
```

This handles the situation (for example, within Ansys Fluent UDFs) when the current working directory is not shared between the host and the nodes. For the host, the input argument `filename` is the path to the file that is to be copied to the nodes. For the nodes, the input argument is the directory on the nodes under which the file is copied. Upon successful completion, `host_to_node_sync_file()` returns the number of bytes copied, otherwise, `-1` is returned.

Example

In the following example, the host process on Windows copies the file from its local directory `e:\udfs\test.bat` to the directory `/tmp` on the remote Linux nodes.

```
DEFINE_ON_DEMAND(host_to_node_sync)
{
#if RP_HOST
  int total_bytes_copied = host_to_node_sync_file("e:\\udfs\\test.dat.h5");
#endif
#if RP_NODE
  int total_bytes_copied = host_to_node_sync_file("/tmp");
  /* The file /tmp/test.dat.h5 can be opened now */
#endif
  printf("Total number of bytes copied is %d\n", total_bytes_copied);
}
```

7.4.2. Writing Files in Parallel

Writing files in parallel is done in the following stages:

1.   The node_host process opens the file.

2.   Compute node-0 sends its data to the node_host.

3.   The other compute nodes send their data to compute node-0.

4.   Compute node-0 receives the data from the other compute nodes and sends it to the node_host.

5.   The node_host receives the data sent from *all* the compute nodes and writes it to the file.

6.   The node_host closes the file.

Since the NODE_HOST and NODE processes are performing different tasks, the example below appears long and utilizes a large number of compiler directives. If, however, as an exercise you make three copies of this example and in each copy delete the unused sections for either the NODE_HOST or NODE versions, then you will see that it is actually quite a simple routine.

Example: Writing Data to a Common File on the node_host Process's File System

```
/*******************************************************************
   This function will write pressures and positions
    for a fluid zone to a file on the host machine
 *******************************************************************/
#include "udf.h"

# define FLUID_ID 2

DEFINE_ON_DEMAND(pressures_to_file)
{
    /* Different variables are needed on different nodes */
#if !RP_HOST
  Domain *domain=Get_Domain(1);
  Thread *thread;
  cell_t c;
#else
  int i;
#endif

#if !RP_NODE
  FILE *fp = NULL;
  char filename[]="press_out.txt";
#endif

  int size; /* data passing variables */
  real *array;
  int pe;

#if !RP_HOST
  thread=Lookup_Thread(domain,FLUID_ID);
#endif
#if !RP_NODE
  if ((fp = fopen(filename, "w"))==NULL)
     Message("\n Warning: Unable to open %s for writing\n",filename);
  else
     Message("\nWriting Pressure to %s...",filename);
#endif
/* UDF Now does 2 different things depending on NODE or HOST */

#if RP_NODE
  /* Each Node loads up its data passing array */
  size=THREAD_N_ELEMENTS_INT(thread);
  array = (real *)malloc(size * sizeof(real));
  begin_c_loop_int(c,thread)
    array[c]= C_P(c,thread);
  end_c_loop_int(c,thread)

  /* Set pe to destination node */
  /* If on node_0 send data to host */
  /* Else send to node_0 because */
  /*  compute nodes connect to node_0 & node_0 to host */
  pe = (I_AM_NODE_ZERO_P) ? node_host : node_zero;
  PRF_CSEND_INT(pe, &size, 1, myid);
  PRF_CSEND_REAL(pe, array, size, myid);
  free(array);/* free array on nodes after data sent */

  /* node_0 now collect data sent by other compute nodes */
  /*  and sends it straight on to the host */

 if (I_AM_NODE_ZERO_P)
   compute_node_loop_not_zero (pe)
   {
```

```
        PRF_CRECV_INT(pe, &size, 1, pe);
        array = (real *)malloc(size * sizeof(real));
        PRF_CRECV_REAL(pe, array, size, pe);
        PRF_CSEND_INT(node_host, &size, 1, myid);
        PRF_CSEND_REAL(node_host, array, size, myid);
        free((char *)array);
    }
#endif /* RP_NODE */

#if RP_HOST
  compute_node_loop (pe) /* only acts as a counter in this loop */
    {
        /* Receive data sent by each node and write it out to the file */
        PRF_CRECV_INT(node_zero, &size, 1, node_zero);
        array = (real *)malloc(size * sizeof(real));
        PRF_CRECV_REAL(node_zero, array, size, node_zero);

  for (i=0; i<size; i++)
     fprintf(fp, "%g\n", array[i]);

  free(array);
}
#endif /* RP_HOST */


#if !RP_NODE
  fclose(fp);
  Message("Done\n");
#endif

}
```

7.5. Enabling Fluent UDFs to Execute on General Purpose Graphics Processing Units (GPGPUs)

UDFs can be compiled in Ansys Fluent with OpenCL support for execution on General Purpose Graphics Processing Units (GPGPUs) on lnamd64 and win64 platforms. To enable OpenCL support, you should use the following text user interface (TUI) command: `/define/user-defined/enable-udf-on-gpu` before compiling the UDF. The UDF library is linked with `libOpenCL.so` on lnamd64 and `OpenCL.dll` on win64. For execution on GPU systems, the appropriate environment variable (`LD_LIBRARY_PATH` on lnamd64 or `%path%` on win64) should be correctly set so that `libOpenCL.so`/`OpenCL.dll` is loaded along with the UDF library.

Chapter 8: Examples

This chapter provides examples of UDFs that range from simple to complex. It begins with a step-by-step process that takes you through the seven basic steps of programming and using a UDF in Ansys Fluent. Some examples for commonly used applications are subsequently presented.

8.1. Step-By-Step UDF Example

You can use the following process to code a UDF and use it effectively in your Ansys Fluent model.

For more information, see the following sections:

8.1.1. Process Overview

To begin the process, you'll need to define the problem you want to solve using a UDF (Step 1). For example, suppose you want to use a UDF to define a custom boundary profile for your model. You will first need to define the set of mathematical equations that describes the profile.

Next you will need to translate the mathematical equation (conceptual design) into a function written in the C programming language (Step 2). You can do this using any text editor. Save the file with a

`.c` suffix (for example, `udfexample.c`) in your working folder. (See Appendix A: C Programming Basics (p. 677) for some basic information on C programming.)
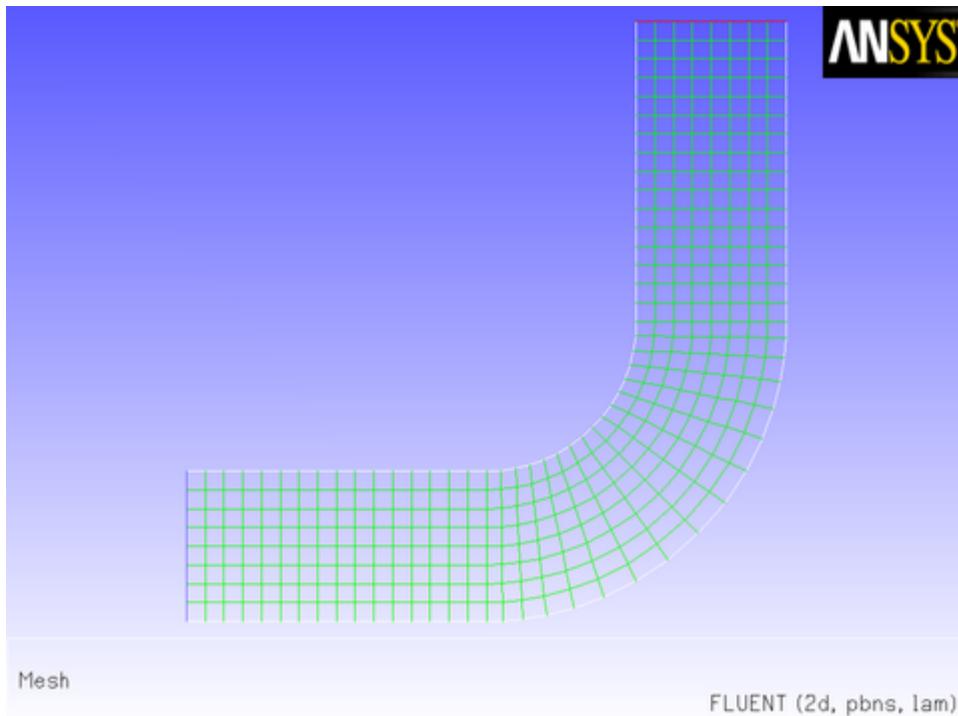
After you have written the C function, you are ready to start Ansys Fluent and read in (or set up) your case file (Step 3). You will then need to interpret or compile the source code, debug it (Step 4), and then hook the function to Ansys Fluent (Step 5). Finally you'll run the calculation (Step 6), analyze the results from your simulation, and compare them to expected results (Step 7). You may loop through this entire process more than once, depending on the results of your analysis. Follow the step-by-step process in the sections below to see how this is done.

8.1.2. Step 1: Define Your Problem
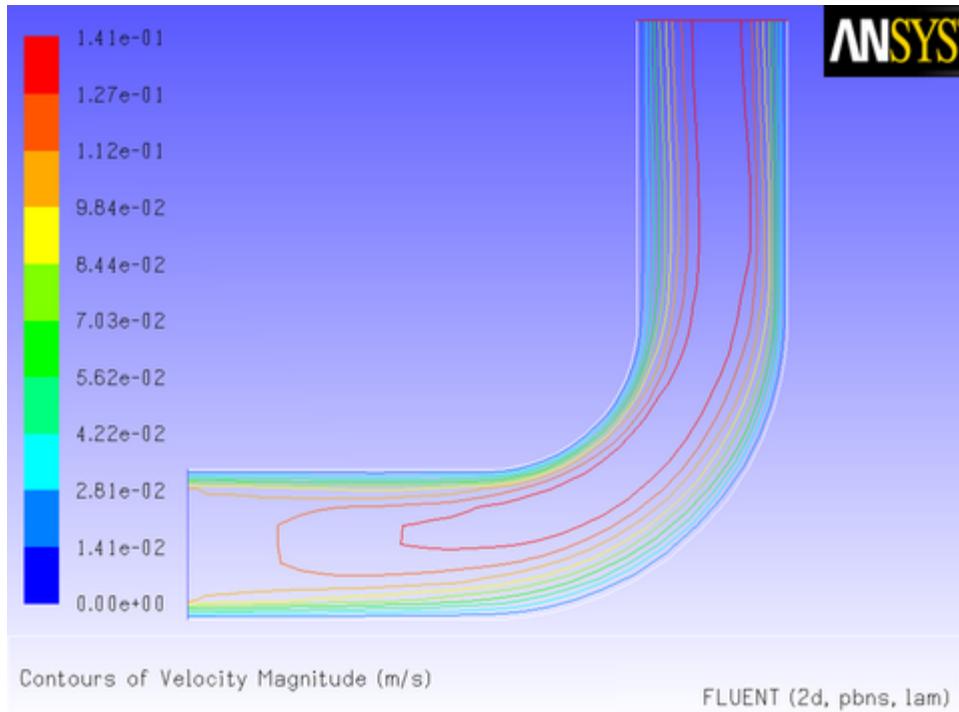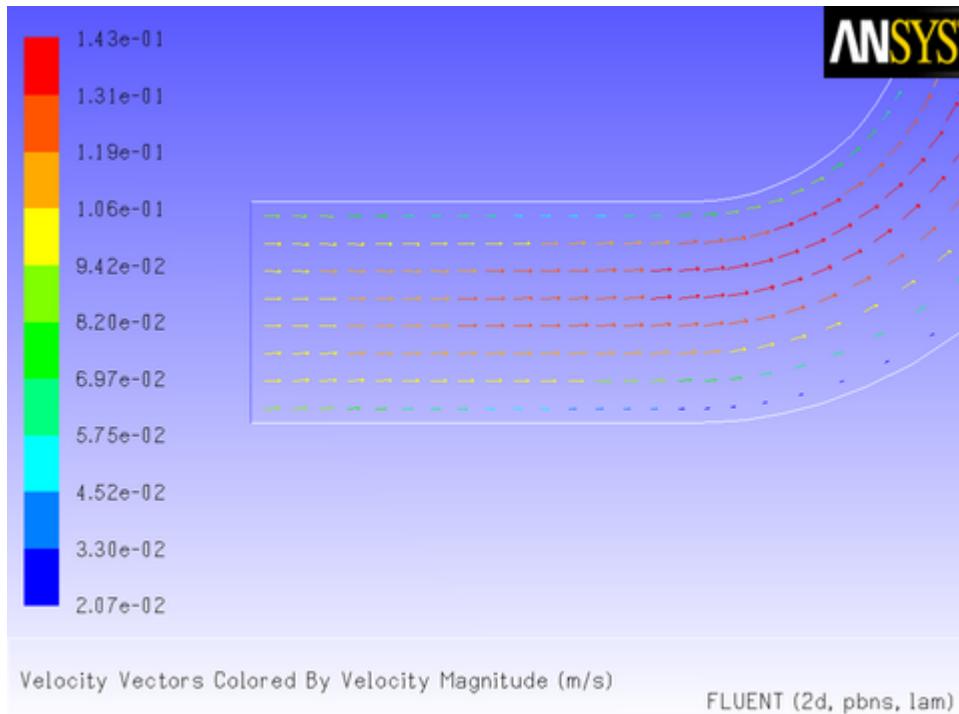
The first step in creating a UDF and using it in your Ansys Fluent model involves defining your model equation(s).

Consider the elbow duct illustrated in Figure 8.1: The Mesh for the Elbow Duct Example (p. 612). The domain has a velocity inlet on the left side, and a pressure outlet at the top of the right side.

Figure 8.1: The Mesh for the Elbow Duct Example

A flow field in which a constant *x* velocity is applied at the inlet will be compared with one where a parabolic *x* velocity profile is applied. The results of a constant velocity (of 0.1 m/s) at the inlet are shown in Figure 8.2: Velocity Magnitude Contours for a Constant Inlet x Velocity (p. 613) and Figure 8.3: Velocity Vectors for a Constant Inlet x Velocity (p. 613).

Figure 8.2: Velocity Magnitude Contours for a Constant Inlet x Velocity

Figure 8.3: Velocity Vectors for a Constant Inlet x Velocity

Now suppose that you want to impose a non-uniform *x* velocity to the duct inlet, which has a parabolic shape. The velocity is 0 m/s at the walls of the inlet and 0.1 m/s at the center.

To solve this type of problem, you can write a custom profile UDF and apply it to your Ansys Fluent model.

8.1.3. Step 2: Create a C Source File

Now that you have determined the shape of the velocity profile that defines the UDF, you can use any text editor to create a file containing C code that implements the function. Save the source code file with a `.c` extension (for example, `udfexample.c`) in your working folder. The following UDF source code listing contains only a single function. Your source file can contain multiple concatenated functions. (Refer to Appendix A: C Programming Basics (p. 677) for basic information on C programming.)

Below is an example of how the profile described in Step 1 can be implemented in a UDF. The functionality of the UDF is designated by the leading `DEFINE` macro. Here, the `DEFINE_PROFILE` macro is used to indicate to the solver that the code that follows will provide profile information at boundaries. Other `DEFINE` macros will be discussed later in this manual. (See DEFINE Macros (p. 19) for details about `DEFINE` macro usage.)

```
/*************************************************************************
   udfexample.c
 UDF for specifying steady-state velocity profile boundary condition
 *************************************************************************/

#include "udf.h"

DEFINE_PROFILE(inlet_x_velocity, thread, position)
{
   real x[ND_ND]; /* this will hold the position vector */
   real y, h;
   face_t f;
   h = 0.016; /* inlet height in m */
   begin_f_loop(f,thread)
   {
     F_CENTROID(x, f, thread);
     y = 2.*(x[1]-0.5*h)/h; /* non-dimensional y coordinate */
     F_PROFILE(f, thread, position) = 0.1*(1.0-y*y);
   }
   end_f_loop(f, thread)

}
```

The first argument of the `DEFINE_PROFILE` macro, `inlet_x_velocity`, is the name of the UDF that you supply. The name will appear in the boundary condition dialog box after the function is interpreted or compiled, enabling you to hook the function to your model. Note that the UDF name you supply cannot contain a number as the first character. The equation that is defined by the function will be applied to all cell faces (identified by `f` in the face loop) on a given boundary zone (identified by `thread`). The thread is defined automatically when you hook the UDF to a particular boundary in the Ansys Fluent GUI. The index is defined automatically through the `begin_f_loop` utility. In this UDF, the `begin_f_loop` macro (Looping Macros (p. 364)) is used to loop through all cell faces in the boundary zone. For each face, the coordinates of the face centroid are accessed by `F_CENTROID` (Face Centroid (`F_CENTROID`) (p. 322)). The y coordinate `y` is used in the parabolic profile equation and the returned velocity is assigned to the face through `F_PROFILE`. `begin_f_loop` and `F_PROFILE` (Set Boundary Condition Value (`F_PROFILE`) (p. 331)) are Ansys Fluent-supplied macros. Refer to Additional Macros for Writing UDFs (p. 305) for details on how to utilize predefined macros and functions supplied by Ansys Fluent to access Ansys Fluent solver data and perform other tasks.

8.1.4. Step 3: Start Ansys Fluent and Read (or Set Up) the Case File

After you have created the source code for your UDF, you are ready to begin the problem setup in Ansys Fluent.

1. Start Ansys Fluent in Windows using Fluent Launcher with the following settings:

    •   Specify the folder that contains your case, data, and UDF source files in the **Working Directory** field in the **General Options** tab.

    •   If you plan to compile the UDF, make sure that the batch file for the UDF compilation environment settings is correctly specified in the **Environment** tab (see Compilers (p. 401) for further details).

2. Read (or set up) your case file.

8.1.5. Step 4: Interpret or Compile the Source File

You are now ready to interpret or compile the profile UDF (named `inlet_x_velocity`) that you created in Step 2 and that is contained within the source file named `udfexample.c`. In general, you *must* compile your function as a compiled UDF if the source code contains structured reference calls or other elements of C that are not handled by the Ansys Fluent interpreter. To determine whether you should compile or interpret your UDF, see Differences Between Interpreted and Compiled UDFs (p. 8).

8.1.5.1. Interpret the Source File
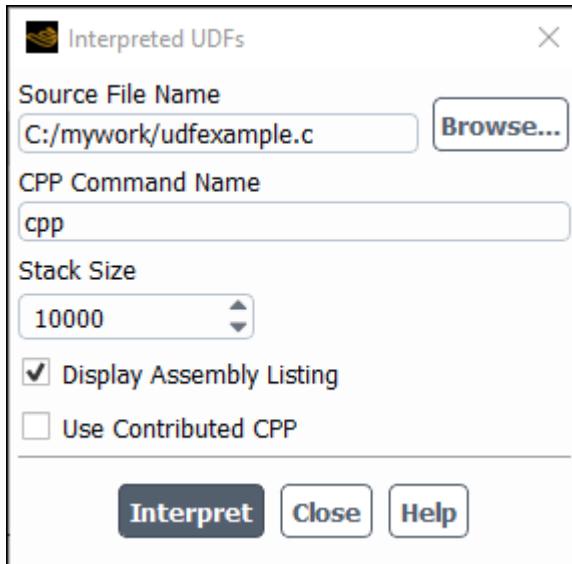
Follow the procedure below to interpret your source file in Ansys Fluent. For more information on interpreting UDFs, see Interpreting UDFs (p. 393).

Important:
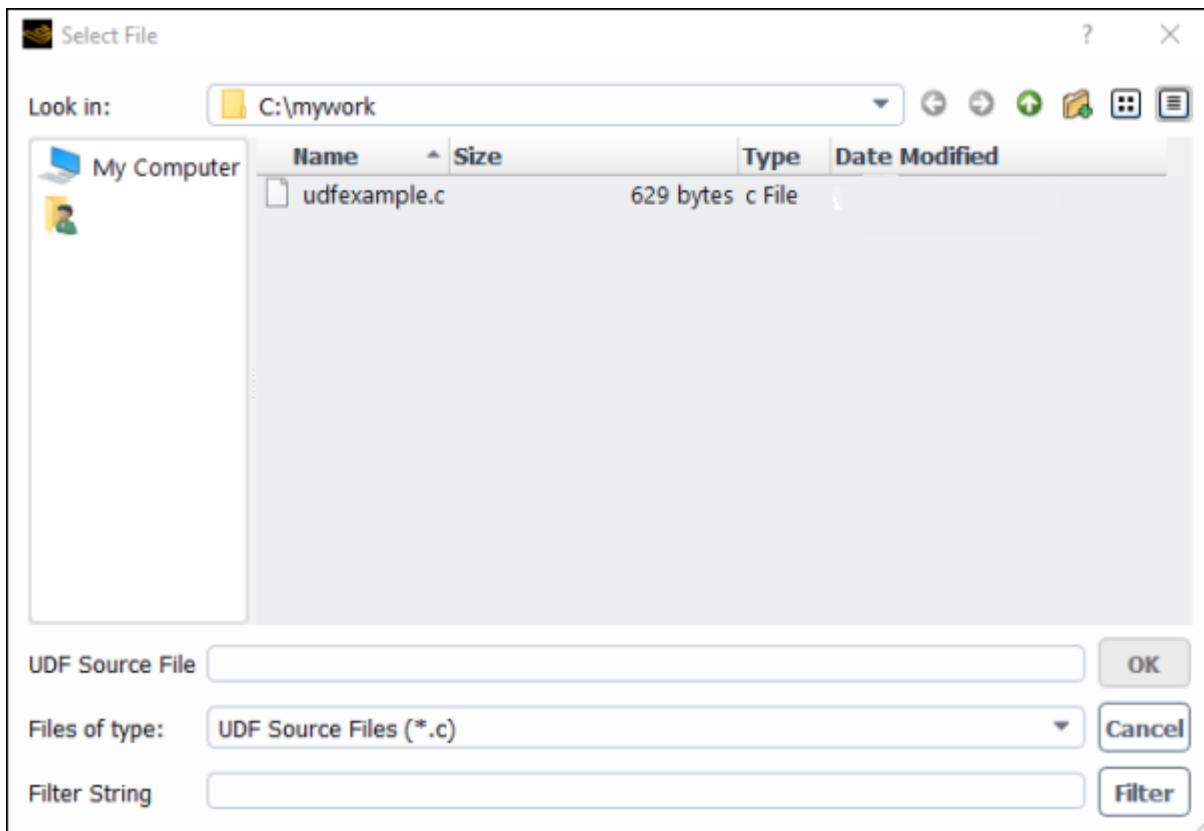
Note that this step does not apply to Windows parallel networks. See Interpreting a UDF Source File Using the Interpreted UDFs Dialog Box (p. 395) for details.

---

1. Open the **Interpreted UDFs** dialog box.

    ⛶ Parameters & Customization → **User Defined Functions** 🖱 **Interpreted...**

Figure 8.4: The Interpreted UDFs Dialog Box

2. In the **Interpreted UDFs** dialog box, indicate the UDF source file you want to interpret by clicking the **Browse...** button. This will open the **Select File** dialog box ().

Figure 8.5: The Select File Dialog Box

In the **Select File** dialog box, select the desired file (for example, `udfexample.c`) and click **OK**. The **Select File** dialog box will close and the complete path to the file you selected will

appear in the **Source File Name** field in the **Interpreted UDFs** dialog box (Figure 8.4: The Interpreted UDFs Dialog Box (p. 616)).

3. In the **Interpreted UDFs** dialog box, specify the C preprocessor to be used in the **CPP Command Name** field. You can keep the default **cpp** or you can select **Use Contributed CPP** to use the preprocessor supplied by Ansys Fluent.

4. Keep the default **Stack Size** setting of 10000, unless the number of local variables in your function will cause the stack to overflow. In this case, set the **Stack Size** to a number that is greater than the number of local variables used.

5. If you want a listing of assembly language code to appear in your console when the function interprets, enable the **Display Assembly Listing** option. This option will be saved in your case file, so that when you read the case in a subsequent Ansys Fluent session, the assembly code will be automatically displayed.

6. Click **Interpret** to interpret your UDF. If the **Display Assembly Listing** option was enabled, then the assembly code will appear in the console when the UDF is interpreted, as shown below.

```
inlet_x_velocity:
         .local.pointer thread (r0)
         .local.int position (r1)
    0    .local.end
    0    save
         .local.int x (r3)
    1    begin.data 8 bytes, 0 bytes initialized:
      .local.float y (r4)
    5    push.float 0
         .local.float h (r5)
    .       .
    .       .
    .       .
  142    pre.inc.int f (r6)
  144    pop.int
  145    b .L3 (28)
    .L2:
  147    restore
  148    restore
  149    ret.v
```

> **Important:**
>
> Note that if your compilation is unsuccessful, then Ansys Fluent will report an error and you will need to debug your program. See Common Errors Made While Interpreting A Source File (p. 397) for details.

7. Click **Close** when the interpreter has finished.

8. Write the case file. The interpreted UDF will be saved with the case file so that the function will be automatically interpreted whenever the case is subsequently read.

8.1.5.2. Compile the Source File
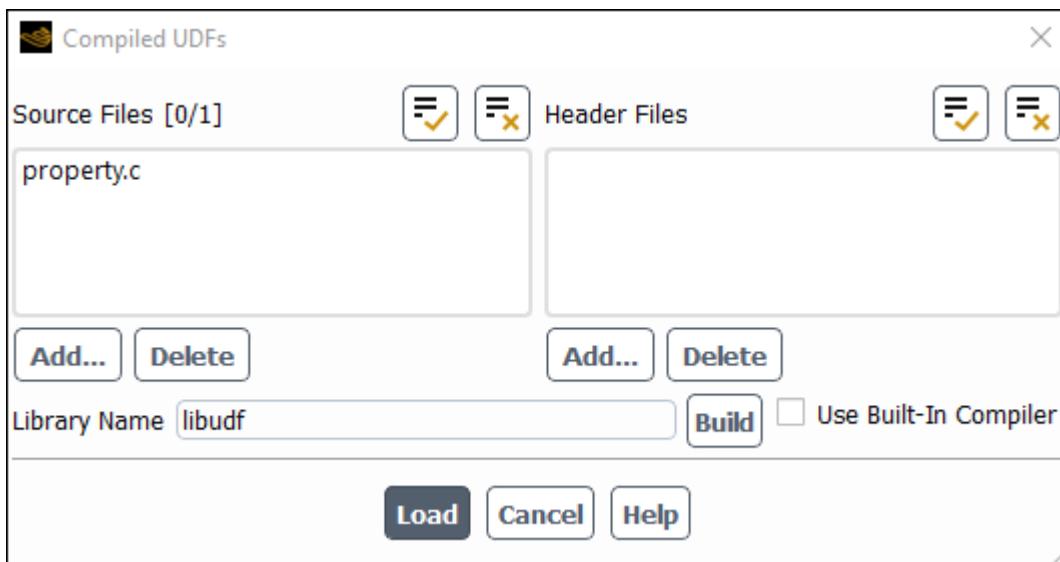
You can compile your UDF using the text user interface (TUI) or the graphical user interface (GUI) in Ansys Fluent. The GUI option for compiling a source file on a Windows system is discussed below.

For details about compiling on other platforms, using the TUI to compile your function, or for general questions about compiling UDFs in Ansys Fluent, see Compiling UDFs (p. 399).
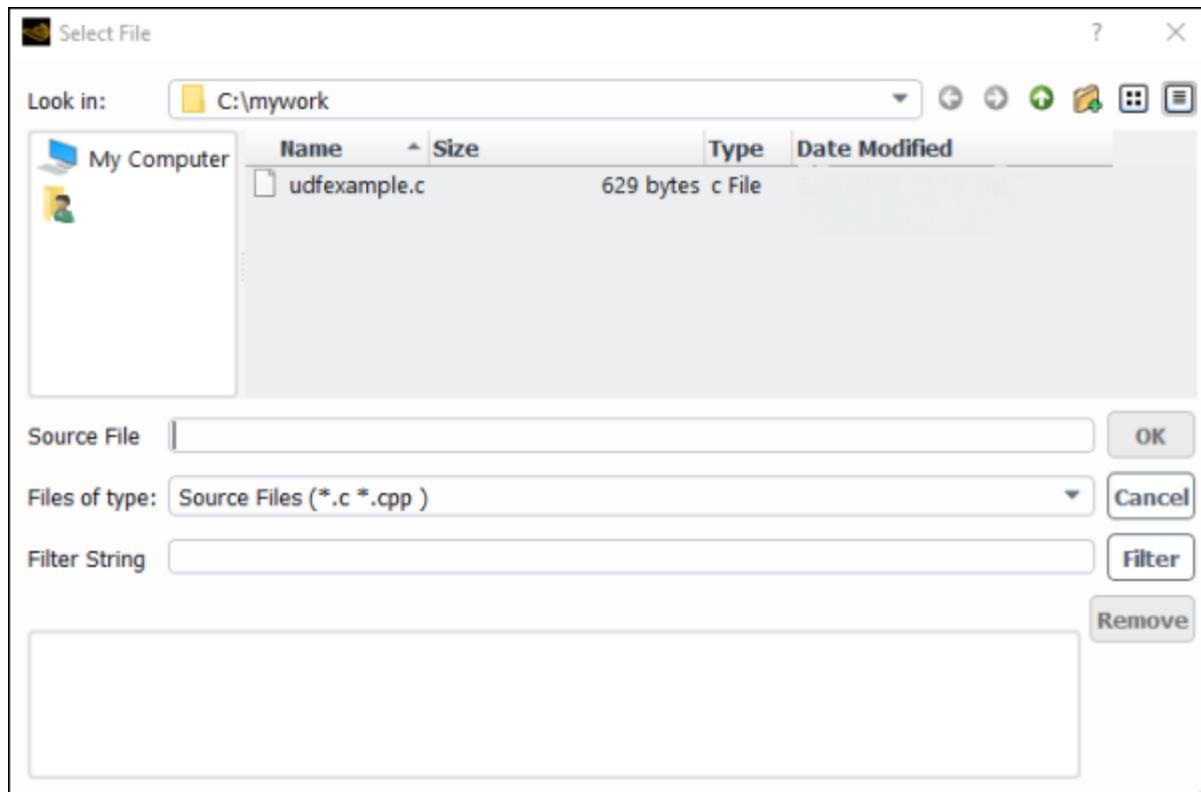
1. As mentioned previously, make sure that you have started Ansys Fluent in Windows using Fluent Launcher with the following settings:

    • Specify the folder that contains your case, data, and UDF source files in the **Working Directory** field in the **General Options** tab.

    • Make sure that the batch file for the UDF compilation environment settings is correctly specified in the **Environment** tab (see Compilers (p. 401) for further details).

2. Open the **Compiled UDFs** dialog box (Figure 8.6: The Compiled UDFs Dialog Box (p. 618)).

   ⊞ **Parameters & Customization** → **User Defined Functions** 🖱 **Compiled...**

Figure 8.6: The Compiled UDFs Dialog Box

3. Click **Add...** under **Source Files** in the **Compiled UDFs** dialog box. This will open the **Select File** dialog box (Figure 8.7: The Select File Dialog Box (p. 619)).

Figure 8.7: The Select File Dialog Box

In the **Select File** dialog box, select the file (for example, `udfexample.c`) you want to compile. The complete path to the source file will then be displayed under **Source File(s)**. Click **OK**. The **Select File** dialog box will close and the file you added will appear in the **Source Files** list in the **Compiled UDFs** dialog box.

In a similar manner, select the **Header Files** that need to be included in the compilation.

4. In the **Compiled UDFs** dialog box, type the name of the shared library in the **Library Name** field (or leave the default name **libudf**).

5. If you installed a version of Microsoft Visual Studio or Clang on your machine that is older and not supported, then you should enable the **Use Built-In Compiler** option, in order to request the use of a built-in compiler (Clang) that is provided with the Fluent installation. If Fluent determines that you have not installed a compiler on your machine, the built-in compiler will be used automatically, whether or not you have enabled the **Use Built-In Compiler** option.

6. Click **Build**. This process will compile the code and will build a shared library in your working folder for the architecture you are running on.

   As the compile/build process begins, a **Question** dialog box will appear, reminding you that the UDF source file must be in the folder that contains your case and data files (that is, your working folder). If you have an existing library folder (for example, **libudf**), then you will need to delete it prior to the build to ensure that the latest files are used. Click **OK** to close the dialog box and resume the compile/build process. The results of the build will be displayed in the

console. You can view the compilation history in the `log` file that is saved in your working folder.

> **Important:**
>
> If the compile/build is unsuccessful, then Ansys Fluent will report an error and you will need to debug your program before continuing. See Common Errors When Building and Loading a UDF Library (p. 420) for a list of common errors.

7. Click **Load** to load the shared library into Ansys Fluent. The console will report that the library has been opened and the function (for example, `inlet_x_velocity`) loaded.
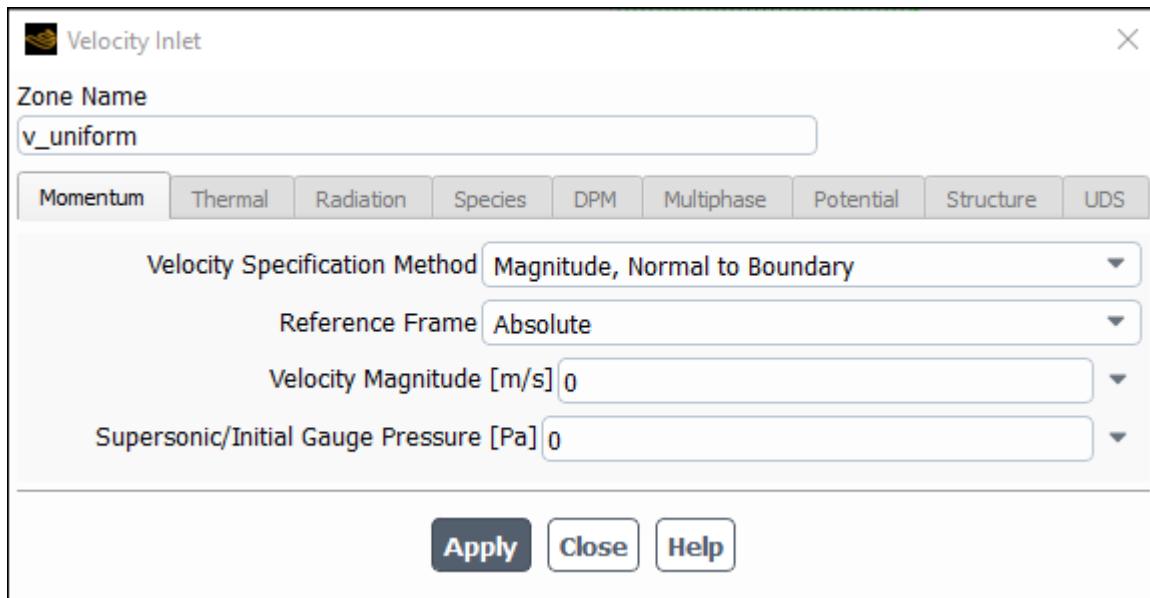
```
Opening library "E:\libudf"...
Library "E:\libudf\win64\3d_host\libudf.dll" opened
Opening library "E:\libudf"...
Library "E:\libudf\win64\3d_node\libudf.dll" opened
 inlet_x_velocity
Done.
```

See Compiling UDFs (p. 399) for more information on the compile/build process.

8.1.6. Step 5: Hook the UDF to Ansys Fluent

Now that you have interpreted or compiled your UDF following the methods outlined in Step 4, you are ready to hook the profile UDF in this sample problem to the **Velocity Inlet** boundary condition dialog box (see Hooking UDFs to Ansys Fluent (p. 423) for details on how to hook UDFs). First, click the **Momentum** tab in the **Velocity Inlet** dialog box (Figure 8.8: The Velocity Inlet Dialog Box (p. 621)) and then choose the name of the UDF that was given in our sample problem with **udf** preceding it (**udf inlet_x_velocity**) from the **X-Velocity** drop-down list. Click **OK** to accept the new boundary condition and close the dialog box. Your user profile will be used in the subsequent solution calculation.

 Setup → ⬥ **Boundary Conditions** → ☰ **cold-inlet** → **Edit...**

Figure 8.8: The Velocity Inlet Dialog Box
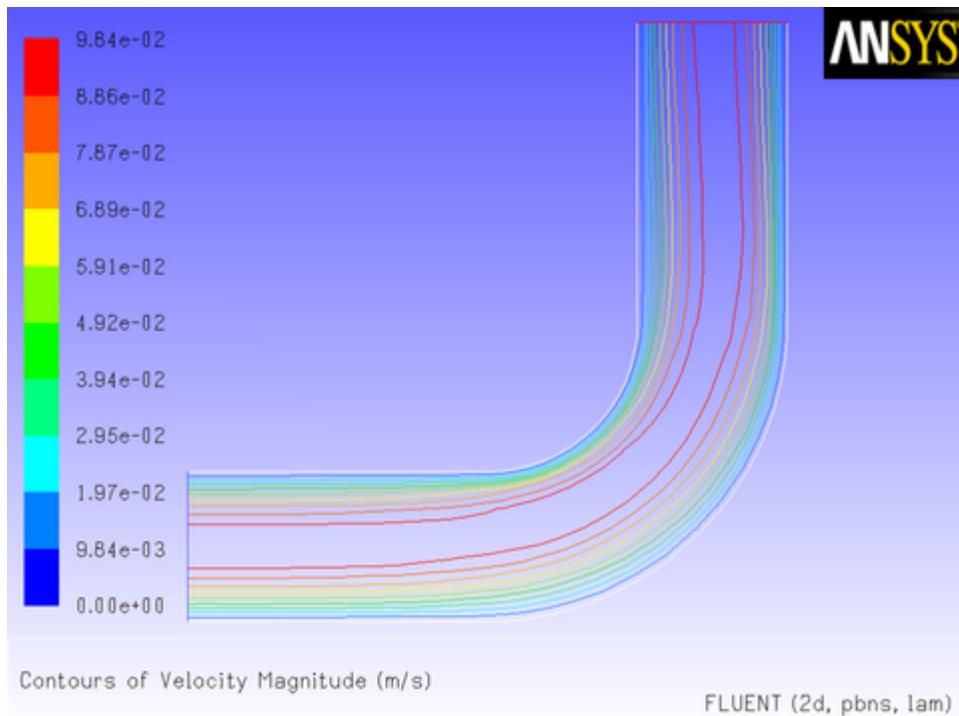
8.1.7. Step 6: Run the Calculation

After initializing the solution, run the calculation.

 Solution → **Run Calculation** **Calculate**

8.1.8. Step 7: Analyze the Numerical Solution and Compare to Expected Results

After the solution is run to convergence, obtain a revised velocity field. The velocity magnitude contours for the parabolic inlet *x* velocity are shown in Figure 8.9: Velocity Magnitude Contours for a Parabolic Inlet Velocity Profile (p. 622), and can be compared to the results of a constant velocity of 0.1 m/s (Figure 8.2: Velocity Magnitude Contours for a Constant Inlet x Velocity (p. 613)). For the constant velocity condition, the velocity profile is seen to develop as the flow passes through the duct. The velocity field for the imposed parabolic profile, however, shows a maximum at the center of the inlet, which drops to zero at the walls.

Figure 8.9: Velocity Magnitude Contours for a Parabolic Inlet Velocity Profile

Contours of Velocity Magnitude (m/s)

FLUENT (2d, pbns, lam)

8.2. Detailed UDF Examples

This section contains detailed examples of UDFs that are used in typical Ansys Fluent applications.

8.2.1. Boundary Conditions

This section contains two applications of boundary condition UDFs.

• Parabolic Velocity Inlet Profile for an Elbow Duct

• Transient Pressure Outlet Profile for Flow in a Tube
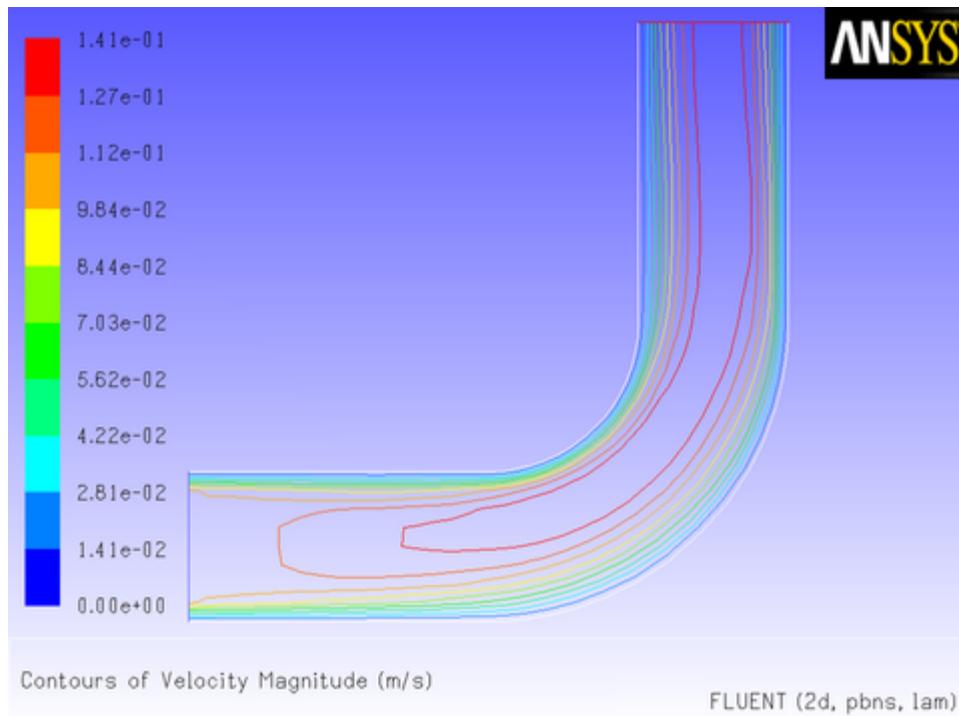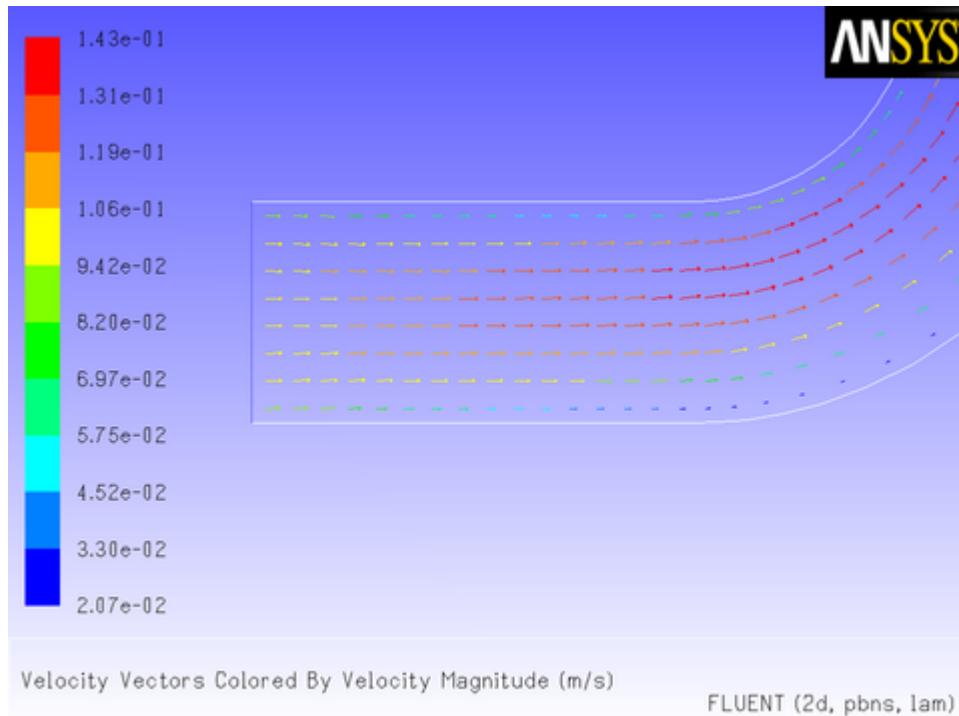
8.2.1.1. Parabolic Velocity Inlet Profile in an Elbow Duct

Consider the elbow duct illustrated in Figure 8.10: The Mesh for the Elbow Duct Example (p. 623). The domain has a velocity inlet on the left side, and a pressure outlet at the top of the right side.

Figure 8.10: The Mesh for the Elbow Duct Example

Mesh

FLUENT (2d, pbns, lam)

A flow field in which a constant *x* velocity is applied at the inlet will be compared with one where a parabolic *x* velocity profile is applied. While the application of a profile using a piecewise-linear profile is available with the boundary profiles option, the specification of a polynomial can be accomplished only by a user-defined function.

The results of a constant velocity (of 0.1 m/sec) at the inlet are shown in Figure 8.11: Velocity Magnitude Contours for a Constant Inlet x Velocity (p. 624) and Figure 8.12: Velocity Vectors for a Constant Inlet x Velocity (p. 624). The velocity profile is seen to develop as the flow passes through the duct.

Figure 8.11: Velocity Magnitude Contours for a Constant Inlet x Velocity

Figure 8.12: Velocity Vectors for a Constant Inlet x Velocity

Now suppose that you want to impose a non-uniform *x* velocity to the duct inlet, which has a parabolic shape. The velocity is 0 m/s at the walls of the inlet and 0.1 m/s at the center.

A UDF is used to introduce this parabolic profile at the inlet. The C source code (`vprofile.c`) is shown below. The function makes use of Ansys Fluent-supplied solver functions that are described in Face Macros (p. 322).

The UDF, named `inlet_x_velocity`, is defined using `DEFINE_PROFILE` and has two arguments: `thread` and `position`. `Thread` is a pointer to the face's thread, and `position` is an integer that is a numerical label for the variable being set within each loop.

The function begins by declaring variable `f` as a `face_t` data type. A one-dimensional array `x` and variable `y` are declared as `real` data types. A looping macro is then used to loop over each face in the zone to create a profile, or an array of data. Within each loop, `F_CENTROID` outputs the value of the face centroid (array `x`) for the face with index `f` that is on the thread pointed to by `thread`. The *y* coordinate stored in `x[1]` is assigned to variable `y`, and is then used to calculate the *x* velocity. This value is then assigned to `F_PROFILE`, which uses the integer `position` (passed to it by the solver based on your selection of the UDF as the boundary condition for *x* velocity in the **Velocity Inlet** dialog box) to set the *x* velocity face value in memory.

```
/**************************************************************************
   vprofile.c
 UDF for specifying steady-state velocity profile boundary condition
 **************************************************************************/

#include "udf.h"

DEFINE_PROFILE(inlet_x_velocity, thread, position)
{
   real x[ND_ND]; /* this will hold the position vector */
   real y, h;
   face_t f;
   h = 0.016; /* inlet height in m */
   begin_f_loop(f,thread)
     {
       F_CENTROID(x, f, thread);
       y = 2.*(x[1]-0.5*h)/h; /* non-dimensional y coordinate */
       F_PROFILE(f, thread, position) = 0.1*(1.0-y*y);
     }
   end_f_loop(f, thread)
}
```
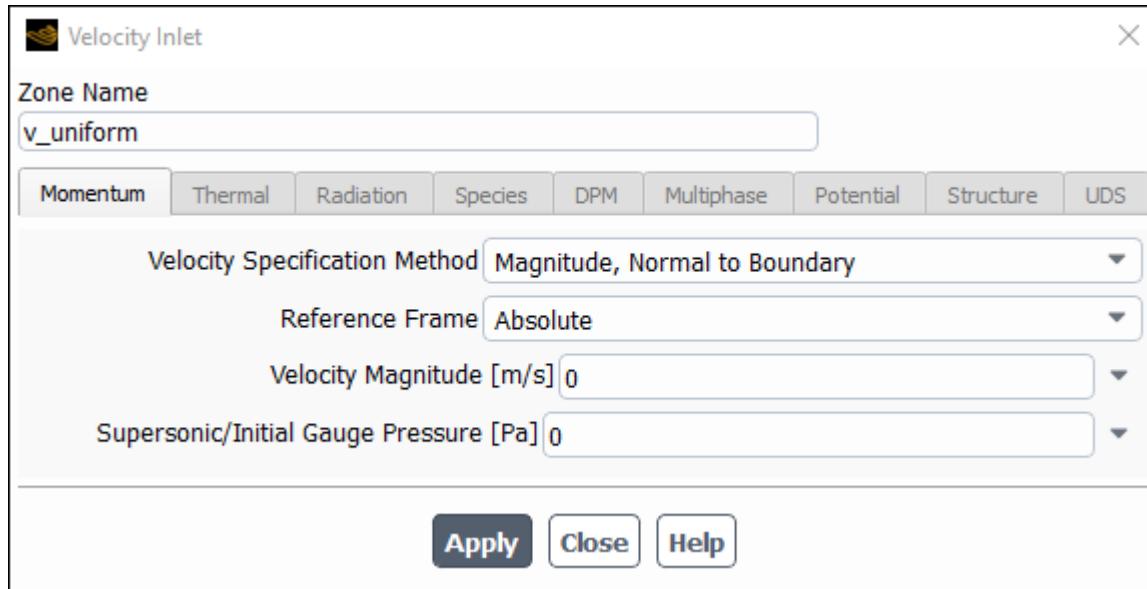
To make use of this UDF in Ansys Fluent, you will first need to interpret (or compile) the function, and then hook it to Ansys Fluent using the graphical user interface. Follow the procedure for interpreting source files using the **Interpreted UDFs** dialog box (Interpreting a UDF Source File Using the Interpreted UDFs Dialog Box (p. 395)), or compiling source files using the **Compiled UDFs** dialog box (Compiling a UDF Using the GUI (p. 403)).

To hook the UDF to Ansys Fluent as the velocity boundary condition for the zone of choice, open the **Velocity Inlet** dialog box and click the **Momentum** tab (Figure 8.13: The Velocity Inlet Dialog Box (p. 626)).

 Setup → ⬥ Boundary Conditions → ≡ velocity-inlet → Edit...

Figure 8.13: The Velocity Inlet Dialog Box

In the **X-Velocity** drop-down list, select **udf inlet_x_velocity**, the name that was given to the function above (with **udf** preceding it). Click **OK** to accept the new boundary condition and close the dialog box. Your user-defined profile will be used in the subsequent solution calculation.

After the solution is initialized and run to convergence, a revised velocity field is obtained as shown in Figure 8.14: Velocity Magnitude Contours for a Parabolic Inlet x Velocity (p. 626) and Figure 8.15: Velocity Vectors for a Parabolic Inlet x Velocity (p. 627). The velocity field shows a maximum at the center of the inlet, which drops to zero at the walls.
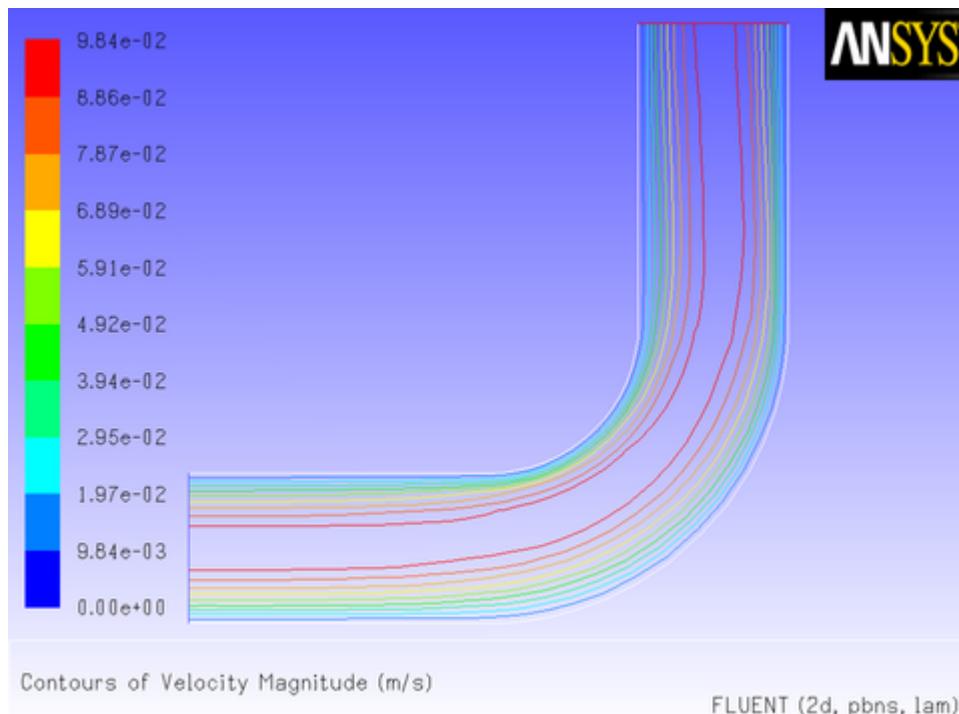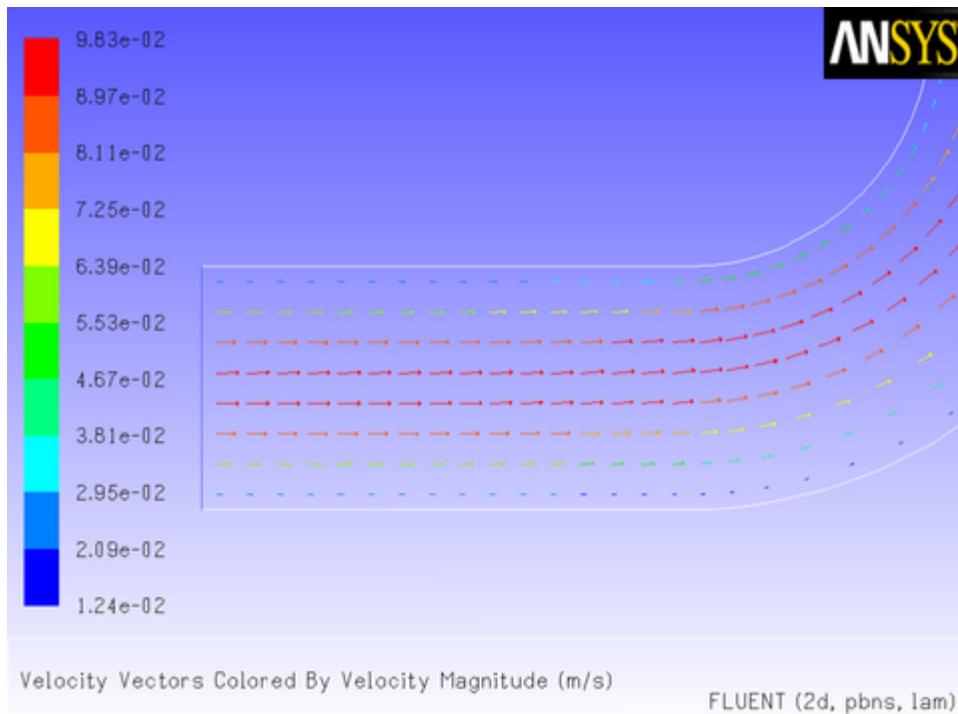
Figure 8.14: Velocity Magnitude Contours for a Parabolic Inlet x Velocity

Figure 8.15: Velocity Vectors for a Parabolic Inlet x Velocity

8.2.1.2. Transient Pressure Outlet Profile for Flow in a Tube

In this example, a temporally periodic pressure boundary condition will be applied to the outlet of a tube using a UDF. The pressure has the form

$$p_x = p_0 + A\sin(\omega t)$$

The tube is assumed to be filled with air, with a fixed total pressure at the inlet. The pressure of the air fluctuates at the outlet about an equilibrium value (p_0) of 101325 Pa, with an amplitude of 5 Pa and a frequency of 10 rad/s.

The source file listing for the UDF that describes the transient outlet profile is shown below. The function, named `unsteady_pressure`, is defined using the `DEFINE_PROFILE` macro. The utility `CURRENT_TIME` is used to look up the `real` flow time, which is assigned to the variable `t`. (See Time-Dependent Macros (p. 380) for details on `CURRENT_TIME`).

```
/***********************************************************************
   unsteady.c
 UDF for specifying a transient pressure profile boundary condition
 ***********************************************************************/

#include "udf.h"

DEFINE_PROFILE(unsteady_pressure, thread, position)
{
   face_t f;
   real t = CURRENT_TIME;
   begin_f_loop(f, thread)
     {
        F_PROFILE(f, thread, position) = 101325.0 + 5.0*sin(10.*t);
     }
   end_f_loop(f, thread)
```

```
        }
```
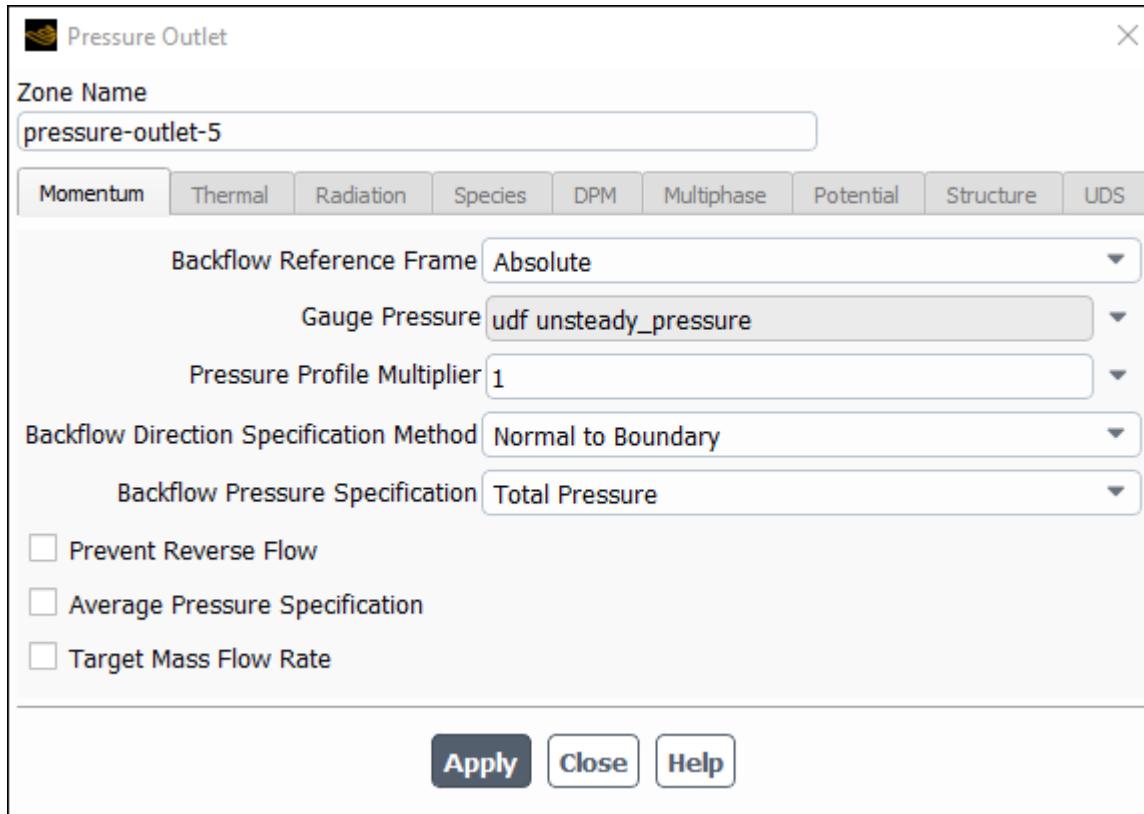
Before you can interpret or compile the UDF, you must specify a transient flow calculation in the **General** task page. Then, follow the procedure for interpreting source files using the **Interpreted UDFs** dialog box (Interpreting a UDF Source File Using the Interpreted UDFs Dialog Box (p. 395)), or compiling source files using the **Compiled UDFs** dialog box (Compiling a UDF Using the GUI (p. 403)).

The sinusoidal pressure boundary condition defined by the UDF can now be hooked to the outlet zone. In the **Pressure Outlet** dialog box (Figure 8.16: The Pressure Outlet Dialog Box (p. 628)), simply select the name of the UDF given in this example with the word **udf** preceding it (**udf unsteady_pressure**) from the **Gauge Pressure** drop-down list. Click **OK** to accept the new boundary condition and close the dialog box. Your user-defined profile will be used in the subsequent solution calculation.

 Setup → ⬦ Boundary Conditions → ☰ pressure-outlet-5 → Edit...

Figure 8.16: The Pressure Outlet Dialog Box

8.2.2. Source Terms

This section contains an application of a source term UDF. It is executed as an interpreted UDF in Ansys Fluent.

8.2.2.1. Adding a Momentum Source to a Duct Flow

When a source term is being modeled with a UDF, it is important to understand the context in which the function is called. When you add a source term, Ansys Fluent will call your function as it performs a global loop on cells. Your function should compute the source term and return it to the solver.

In this example, a momentum source will be added to a 2D Cartesian duct flow. The duct is 4 m long and 2 m wide, and will be modeled using a symmetry boundary through the middle. Liquid metal (with properties listed in Table 8.1: Properties of the Liquid Metal (p. 629)) enters the duct at the left with a velocity of 1 mm/s at a temperature of 290 K. After the metal has traveled 0.5 m along the duct, it is exposed to a cooling wall, which is held at a constant temperature of 280 K. To simulate the freezing of the metal, a momentum source is applied to the metal as soon as its temperature falls below 288 K. The momentum source is proportional to the x component of the velocity, v_x, and has the opposite sign:

$$S_x = -C v_x \tag{8.1}$$

where C is a constant. As the liquid cools, its motion will be reduced to zero, simulating the formation of the solid. (In this simple example, the energy equation will not be customized to account for the latent heat of freezing. The velocity field will be used only as an indicator of the solidification region.)

The solver linearizes source terms in order to enhance the stability and convergence of a solution. To allow the solver to do this, you need to specify the dependent relationship between the source and solution variables in your UDF, in the form of derivatives. The source term, S_x, depends only on the solution variable, v_x. Its derivative with respect to v_x is

$$\frac{\partial S_x}{\partial v_x} = -C \tag{8.2}$$

The following UDF specifies a source term and its derivative. The function, named `cell_x_source`, is defined on a cell using `DEFINE_SOURCE`. The constant C in Equation 8.1 (p. 629) is called `CON` in the function, and it is given a numerical value of 20 kg/m³-s, which will result in the desired units of N/m³ for the source. The temperature at the cell is returned by `C_T(cell,thread)`. The function checks to see if the temperature is below (or equal to) 288 K. If it is, the source is computed according to Equation 8.1 (p. 629) (`C_U` returns the value of the x velocity of the cell). If it is not, the source is set to 0. At the end of the function, the appropriate value for the source is returned to the Ansys Fluent solver.

Table 8.1: Properties of the Liquid Metal

Property	Value
Density	8000 kg/m³
Viscosity	5.5×10^{-3} kg/m-s
Specific Heat	680 J/kg-K
Thermal Conductivity	30 W/m-K

```
/********************************************************************
   UDF that adds momentum source term and derivative to duct flow
 ********************************************************************/

#include "udf.h"

#define CON 20.0
```
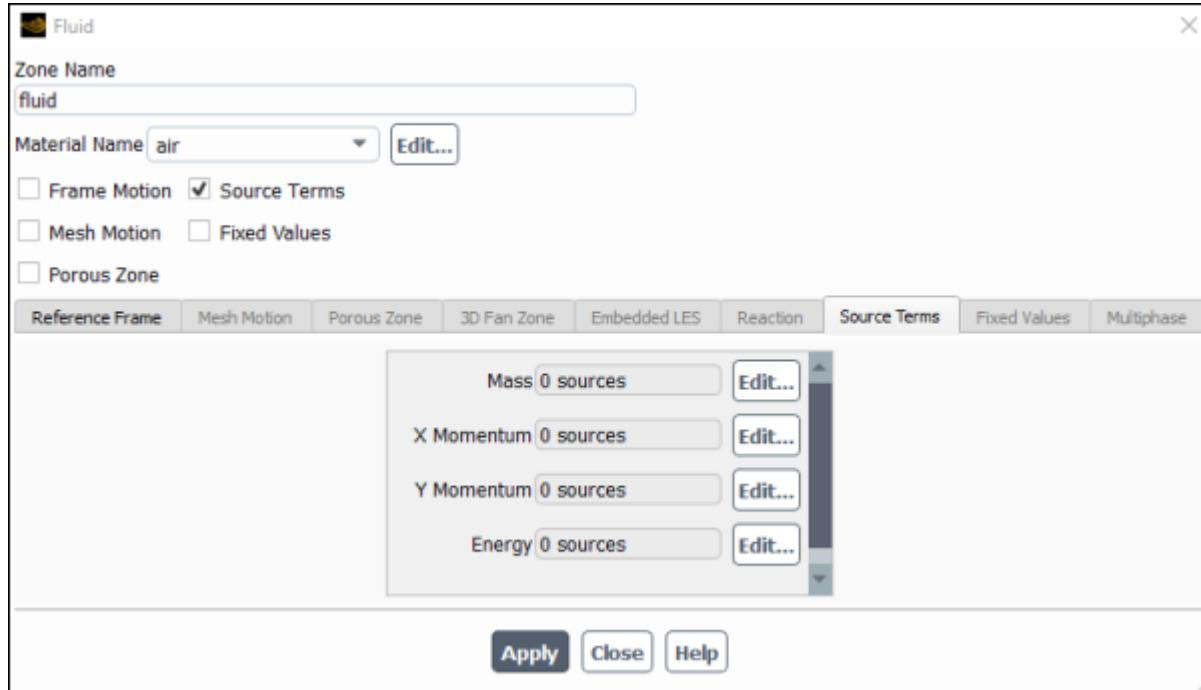
```
DEFINE_SOURCE(cell_x_source, cell, thread, dS, eqn)
{
   real source;
   if (C_T(cell,thread) <= 288.)
   {
     source = -CON*C_U(cell,thread);
     dS[eqn] = -CON;
   }
   else
   {
     source = dS[eqn] = 0.;
   }
   return source;
}
```

To make use of this UDF in Ansys Fluent, you will first need to interpret (or compile) the function, and then hook it to Ansys Fluent using the graphical user interface. Follow the procedure for interpreting source files using the **Interpreted UDFs** dialog box (Interpreting a UDF Source File Using the Interpreted UDFs Dialog Box (p. 395)), or compiling source files using the **Compiled UDFs** dialog box (Compiling a UDF Using the GUI (p. 403)).
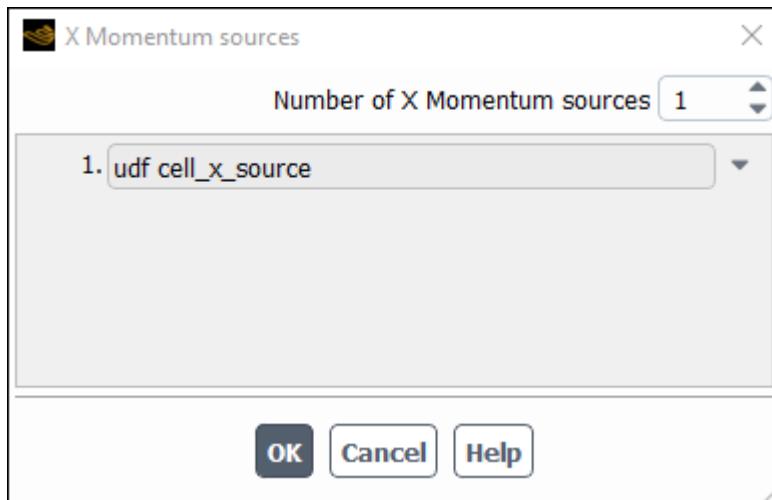
To include source terms in the calculation, you will first need to open the **Fluid** dialog box (Figure 8.17: The Fluid Dialog Box (p. 630)) by selecting the fluid zone in the **Cell Zone Conditions** task page and clicking **Edit...**.

Setup → Cell Zone Conditions → fluid → Edit...

Figure 8.17: The Fluid Dialog Box
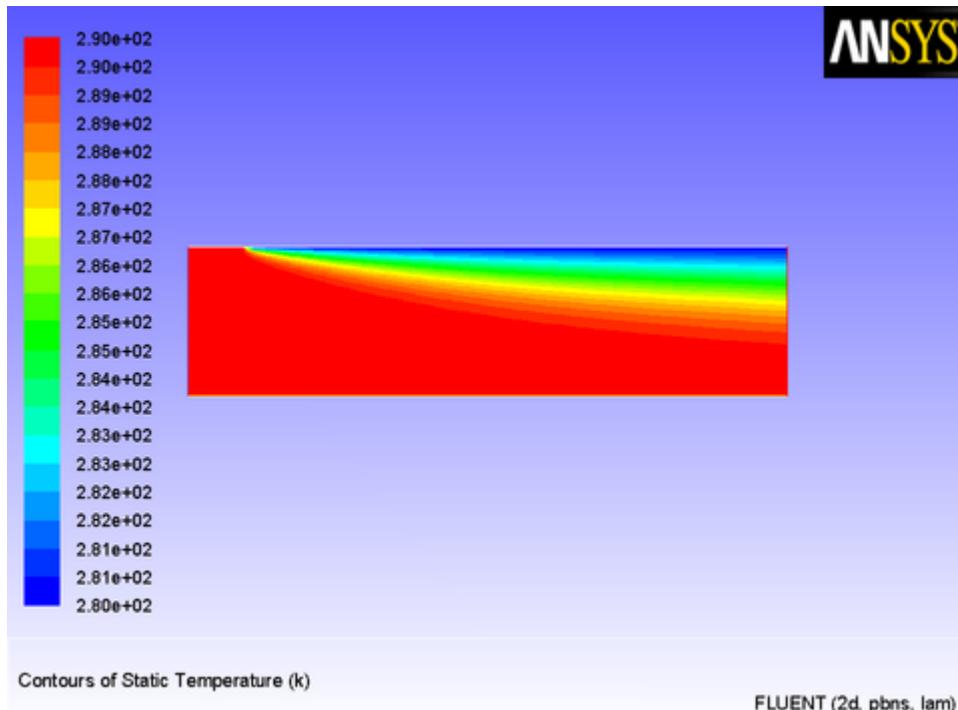
Enable the **Source Terms** option in the **Fluid** dialog box and click the **Source Terms** tab. This will display the momentum source term parameters in the scrollable window. Then, click the **Edit...** button next to the **X Momentum** source term to open the **X Momentum sources** dialog box (Figure 8.18: The X Momentum sources Dialog Box (p. 631)).

Figure 8.18: The X Momentum sources Dialog Box
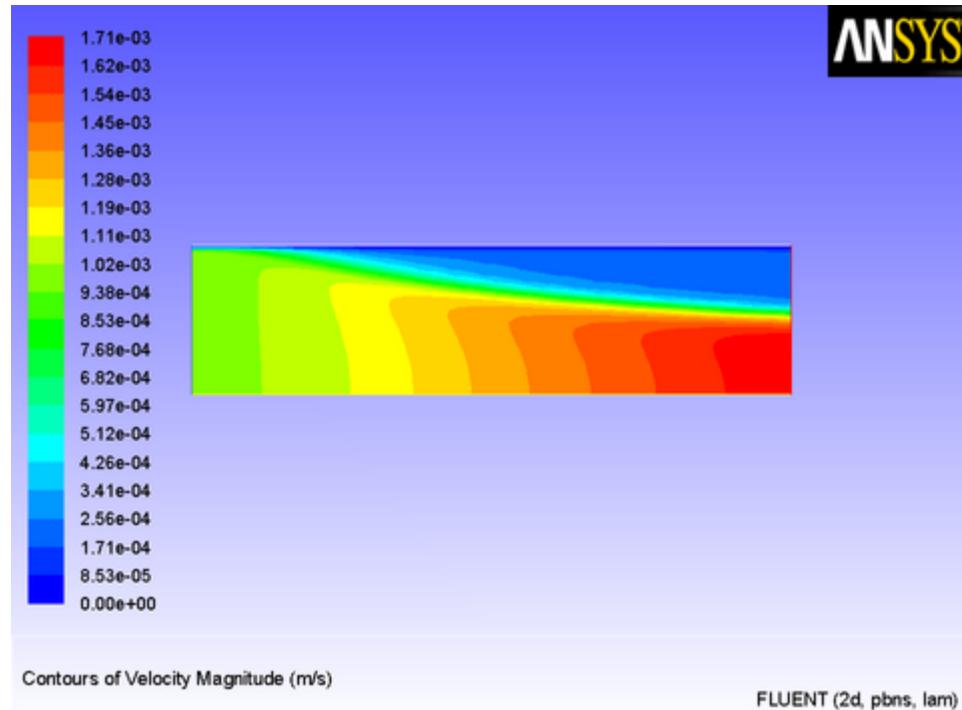
Enter 1 for the **Number of Momentum sources** in the **X Momentum sources** dialog box and then select the function name for the UDF (**udf cell_x_source**) in the drop-down list that appears. (Note that the name that is displayed in the drop-down lists is your UDF name preceded by the word **udf**.) Click **OK** to accept the new cell zone condition and close the dialog box. The **X Momentum** parameter in the **Fluid** dialog box will now display **1 source**. Click **OK** to fix the new momentum source term for the solution calculation and close the **Fluid** dialog box.

After the solution has converged, you can view contours of static temperature to see the cooling effects of the wall on the liquid metal as it moves through the duct (Figure 8.19: Temperature Contours Illustrating Liquid Metal Cooling (p. 631)).
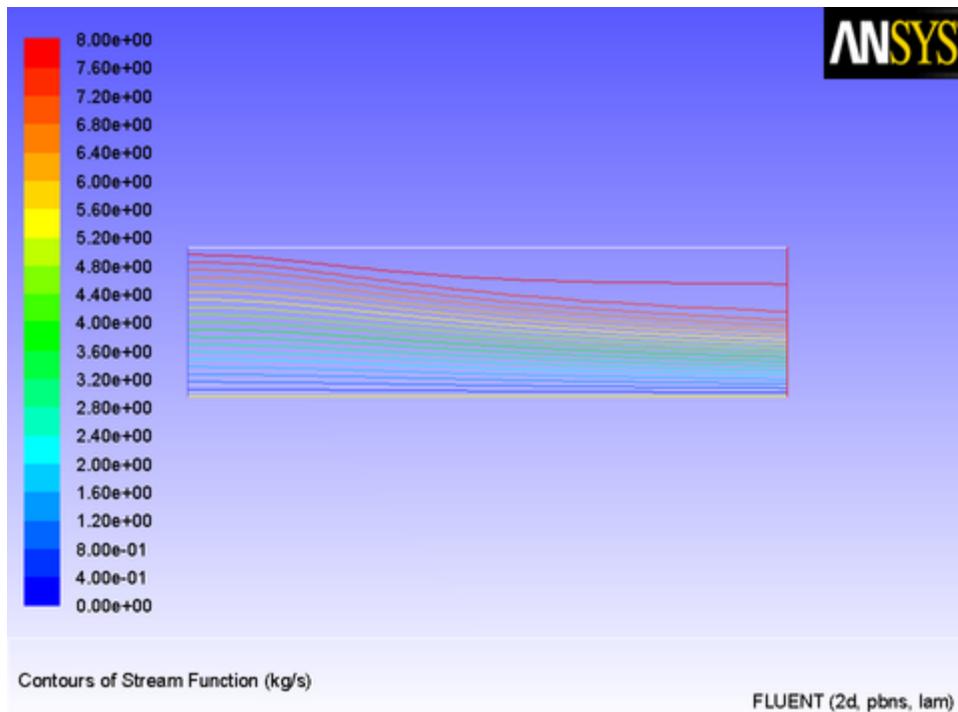
Figure 8.19: Temperature Contours Illustrating Liquid Metal Cooling

Contours of velocity magnitude (Figure 8.20: Velocity Magnitude Contours Suggesting Solidification (p. 632)) show that the liquid in the cool region near the wall has indeed come to rest to simulate solidification taking place.

Figure 8.20: Velocity Magnitude Contours Suggesting Solidification

The solidification is further illustrated by line contours of stream function (Figure 8.21: Stream Function Contours Suggesting Solidification (p. 633)).

Figure 8.21: Stream Function Contours Suggesting Solidification

Contours of Stream Function (kg/s)

FLUENT (2d, pbns, lam)

To more accurately predict the freezing of a liquid in this manner, an energy source term would be needed, as would a more accurate value for the constant appearing in Equation 8.1 (p. 629).

8.2.3. Physical Properties

This section contains an application of a physical property UDF. It is executed as an interpreted UDF in Ansys Fluent.

8.2.3.1. Solidification via a Temperature-Dependent Viscosity

UDFs for properties (as well as sources) are called from within a loop on cells. For this reason, functions that specify properties are required to compute the property for only a single cell, and return the value to the Ansys Fluent solver.

The UDF in this example generates a variable viscosity profile to simulate solidification, and is applied to the same problem that was presented in Adding a Momentum Source to a Duct Flow (p. 629).

The viscosity in the warm ($T > 288$ K) fluid has a molecular value for the liquid (5.5 $\times 10^{-3}$ kg/m-s), while the viscosity for the cooler region ($T < 286$ K) has a much larger value (1.0 kg/m-s). In the intermediate temperature range (286 K $\leq T \leq$ 288 K), the viscosity follows a linear profile (Equation 8.3 (p. 633)) that extends between the two values given above:

$$\mu = 143.2135 - 0.49725T \tag{8.3}$$

This model is based on the assumption that as the liquid cools and rapidly becomes more viscous, its velocity will decrease, thereby simulating solidification. Here, no correction is made for the energy field to include the latent heat of freezing. The C source code for the UDF is shown below.
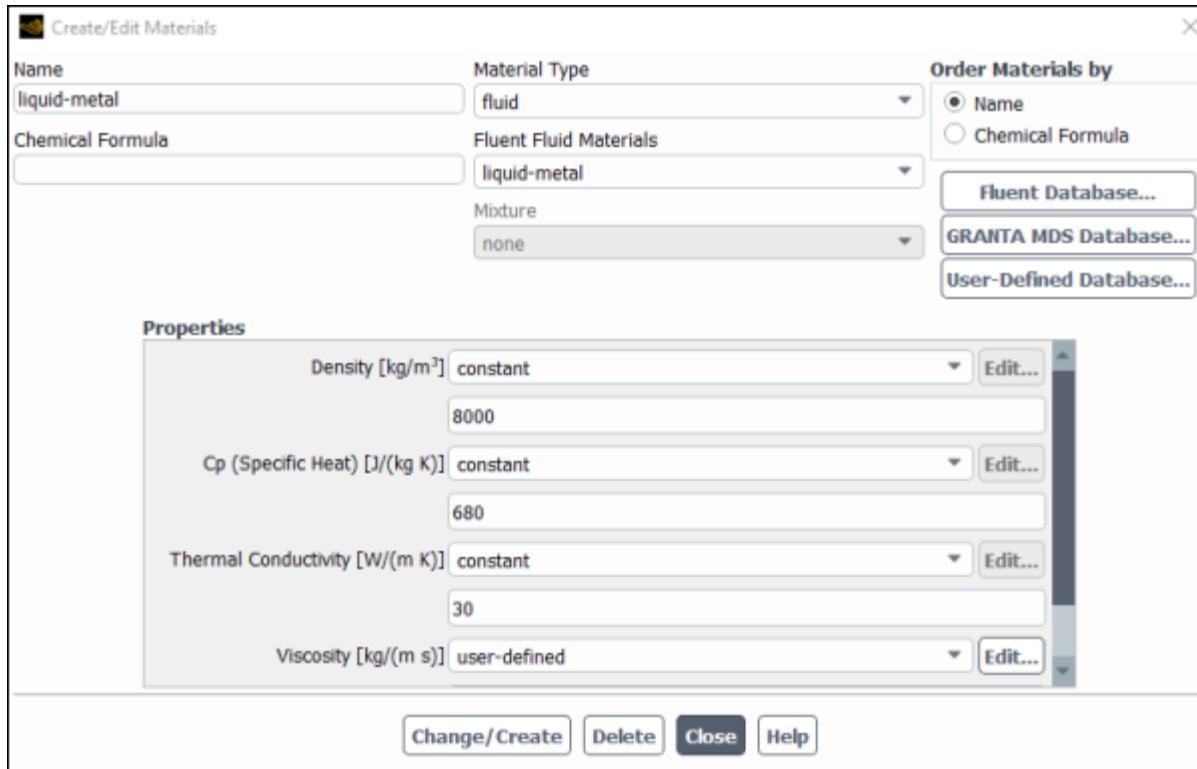
The function, named `cell_viscosity`, is defined on a cell using `DEFINE_PROPERTY`. Two `real` variables are introduced: `temp`, the value of `C_T(cell,thread)`, and `mu_lam`, the laminar viscosity computed by the function. The value of the temperature is checked, and based upon the range into which it falls, the appropriate value of `mu_lam` is computed. At the end of the function, the computed value for `mu_lam` is returned to the solver.

```
/*********************************************************************
   UDF for specifying a temperature-dependent viscosity property
 *********************************************************************/

#include "udf.h"

DEFINE_PROPERTY(cell_viscosity, cell, thread)
{
   real mu_lam;
   real temp = C_T(cell, thread);
   if (temp > 288.)
    mu_lam = 5.5e-3;
   else if (temp > 286.)
    mu_lam = 143.2135 - 0.49725 * temp;
   else
    mu_lam = 1.;
   return mu_lam;
}
```
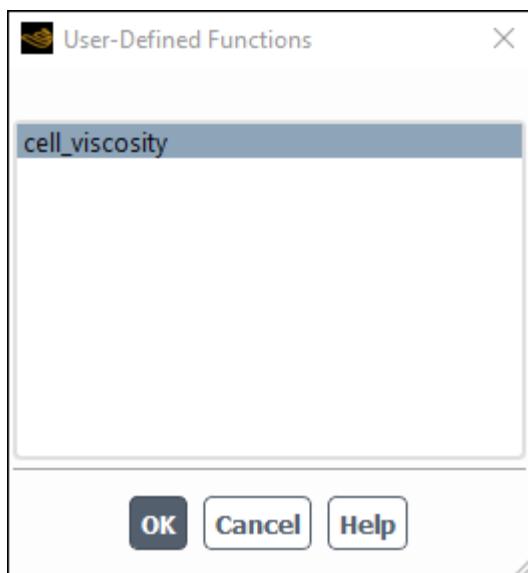
This function can be executed as an interpreted or compiled UDF in Ansys Fluent. Follow the procedure for interpreting source files using the **Interpreted UDFs** dialog box (Interpreting a UDF Source File Using the Interpreted UDFs Dialog Box (p. 395)), or compiling source files using the **Compiled UDFs** dialog box (Compiling a UDF Using the GUI (p. 403)).

To make use of the user-defined property in Ansys Fluent, you will need to open the **Create/Edit Materials** dialog box (Figure 8.22: The Create/Edit Materials Dialog Box (p. 635)) by selecting the liquid metal material in the **Materials** task page and clicking the **Create/Edit...** button.

🗔 **Setup** → ◇ **Materials** → ☰ **liquid_metal** → **Create/Edit...**

Figure 8.22: The Create/Edit Materials Dialog Box

In the **Create/Edit Materials** dialog box, select **user-defined** in the drop-down list for **Viscosity**. This will open the **User-Defined Functions** dialog box (Figure 8.23: The User-Defined Functions Dialog Box (p. 635)), from which you can select the appropriate function name. In this example, only one option is available, but in other examples, you may have several functions from which to choose. (Recall that if you need to compile more than one interpreted UDF, the functions can be concatenated in a single source file prior to compiling.)

Figure 8.23: The User-Defined Functions Dialog Box

The results of this model are similar to those obtained in Adding a Momentum Source to a Duct Flow (p. 629). Figure 8.24: Laminar Viscosity Generated by a User-Defined Function (p. 636) shows the viscosity field resulting from the application of the user-defined function. The viscosity varies rapidly over a narrow spatial band from a constant value of 0.0055 to 1.0 kg/m-s.

The velocity field (Figure 8.25: Contours of Velocity Magnitude Resulting from a User-Defined Viscosity (p. 637)) demonstrates that the liquid slows down in response to the increased viscosity, as expected. In this model, there is a large "mushy" region, in which the motion of the fluid gradually decreases. This is in contrast to the first model, in which a momentum source was applied and a more abrupt change in the fluid motion was observed.

Figure 8.24: Laminar Viscosity Generated by a User-Defined Function
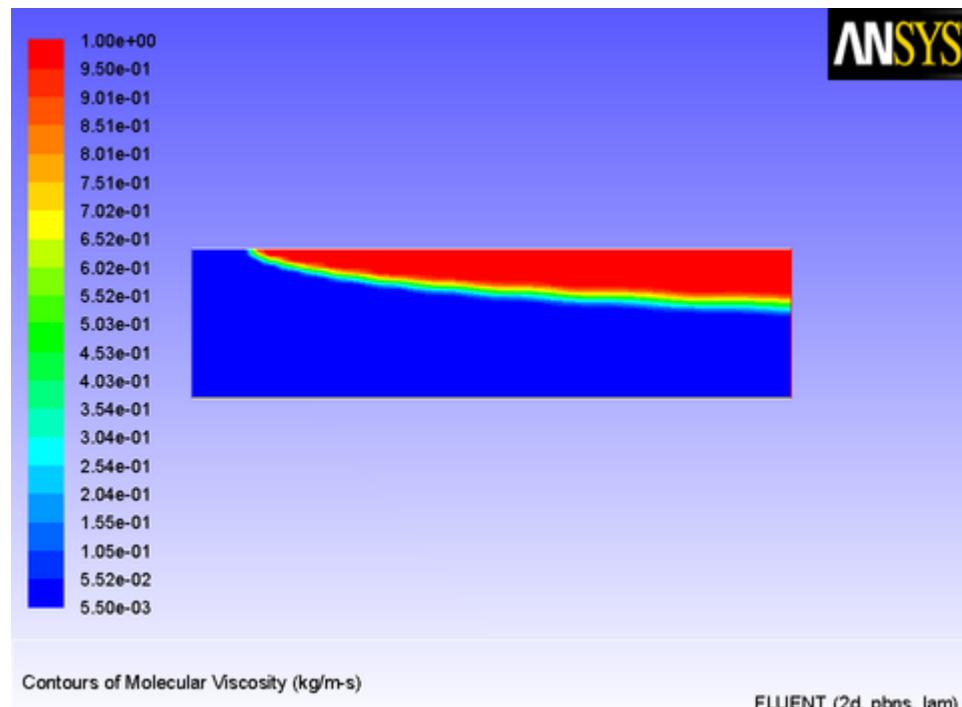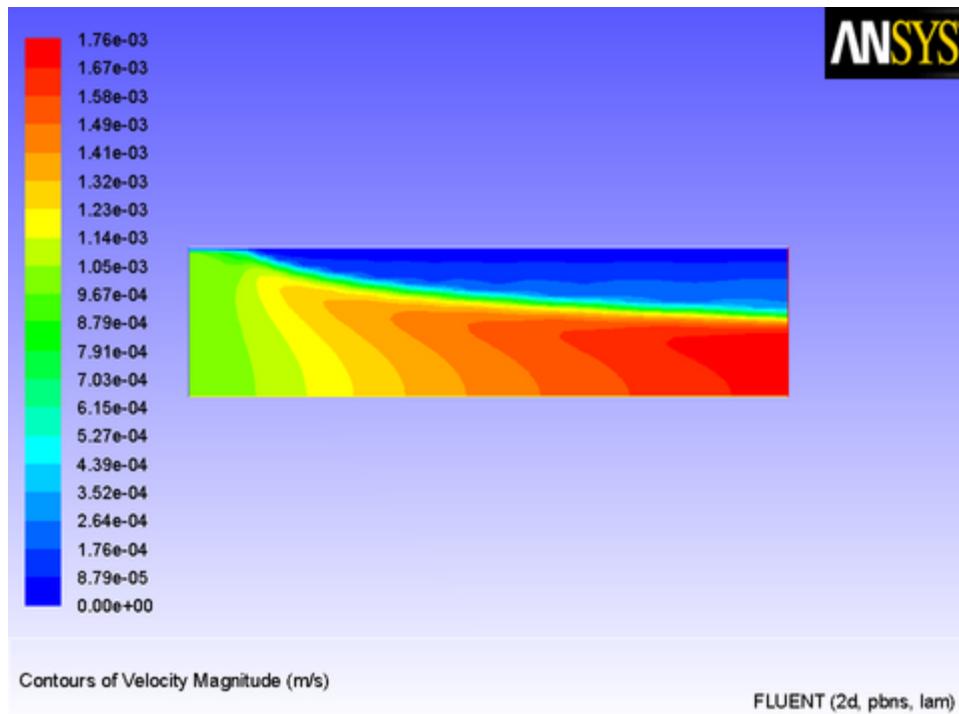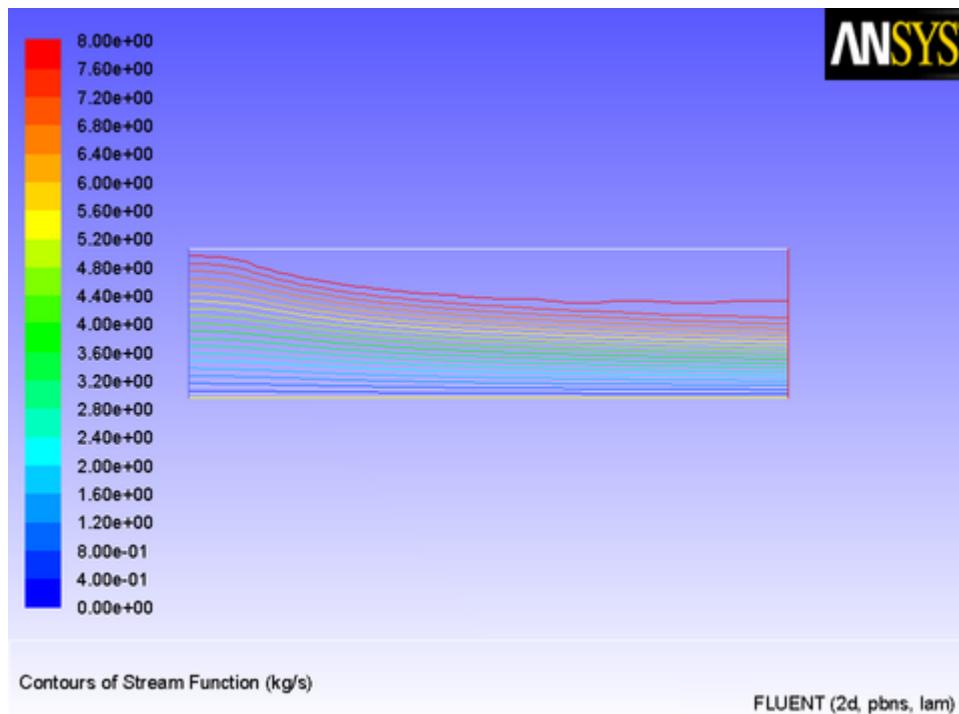
Figure 8.25: Contours of Velocity Magnitude Resulting from a User-Defined Viscosity

Contours of Velocity Magnitude (m/s)

FLUENT (2d, pbns, lam)

Figure 8.26: Stream Function Contours Suggesting Solidification

Contours of Stream Function (kg/s)

FLUENT (2d, pbns, lam)

8.2.4. Reaction Rates

This section contains an example of a custom reaction rate UDF. It is executed as a compiled UDF in Ansys Fluent.

8.2.4.1. Volume Reaction Rate

A custom volume reaction rate for a simple system of two gaseous species is considered. The species are named `species-a` and `species-b`. The reaction rate is one that converts `species-a` into `species-b` at a rate given by the following expression:
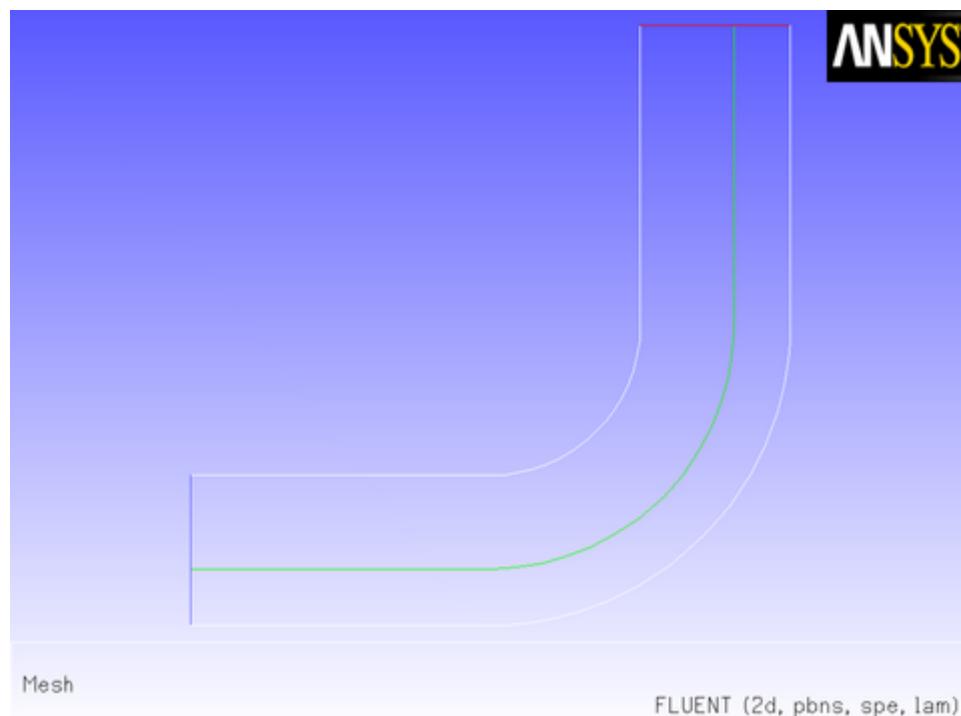
$$R = \frac{K_1 X_a}{\left(1 + K_2 X_a\right)^2}$$

(8.4)

where X_a is the mass fraction of `species-a`, and K_1 and K_2 are constants.

The 2D (planar) domain consists of a 90° bend. The duct has a porous region covering the bottom and right-hand wall, and the reaction takes place in the porous region only. The species in the duct have identical properties. The density is 1.0 kg/m³, and the viscosity is 1.7894 ×10^{-5} kg/m-s.
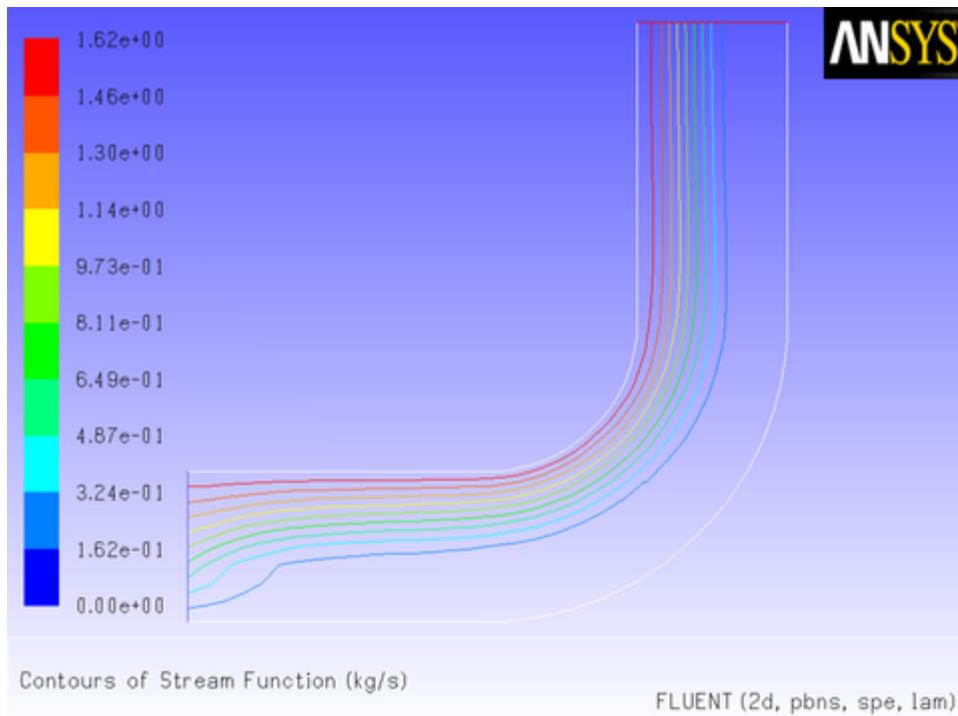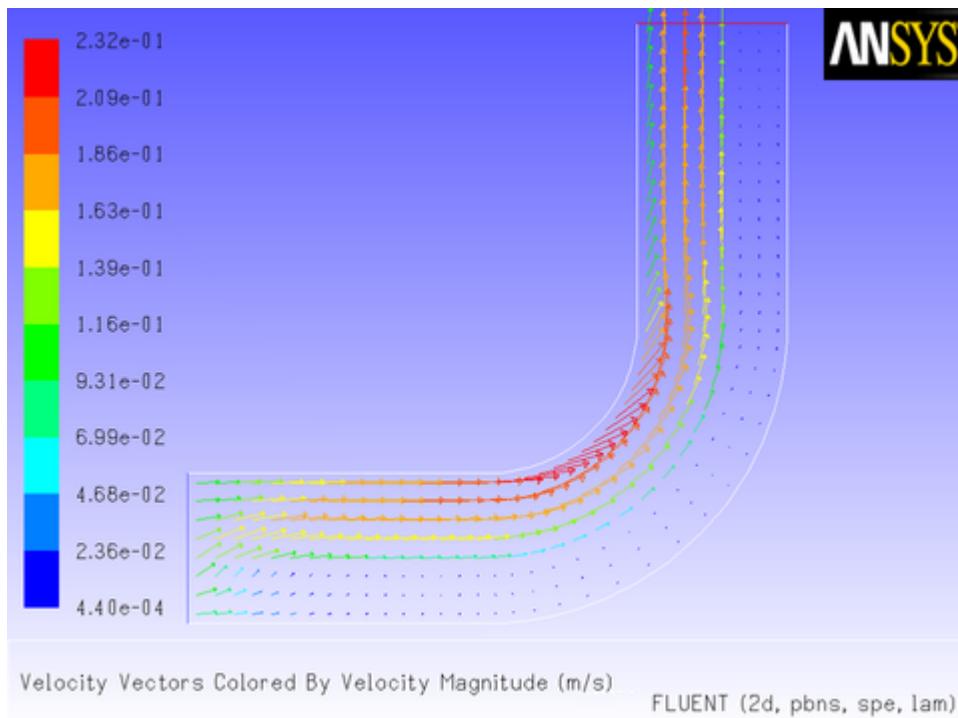
The outline of the domain is shown in Figure 8.27: The Outline of the 2D Duct (p. 638). The porous medium is the region below and to the right of the line that extends from the inlet on the left to the pressure outlet at the top of the domain.

Figure 8.27: The Outline of the 2D Duct

Through the inlet on the left, gas that is purely `species-a` enters with an *x* velocity of 0.1 m/s. The gas enters both the open region on the top of the porous medium and the porous medium itself, where there is an inertial resistance of 5 m^{-1} in each of the two coordinate directions. The laminar flow field (Figure 8.28: Streamlines for the 2D Duct with a Porous Region (p. 639)) shows that most of the gas is diverted from the porous region into the open region.

The flow pattern is further substantiated by the vector plot shown in Figure 8.29: Velocity Vectors for the 2D Duct with a Porous Region (p. 639). The flow in the porous region is considerably slower than that in the open region.

Figure 8.28: Streamlines for the 2D Duct with a Porous Region

Contours of Stream Function (kg/s)   FLUENT (2d, pbns, spe, lam)

Figure 8.29: Velocity Vectors for the 2D Duct with a Porous Region

Velocity Vectors Colored By Velocity Magnitude (m/s)   FLUENT (2d, pbns, spe, lam)

The source code (`rate.c`) that contains the UDF used to model the reaction taking place in the porous region is shown below. The function, named `vol_reac_rate`, is defined on a cell for a given species mass fraction using `DEFINE_VR_RATE`. The UDF performs a test to check if a zone is porous, so that the reaction rate is only calculated for the porous zone. This is done to save calculation time. Note that, in calculating the reaction source terms, Fluent will always use the actual

fluid volume for a reacting zone, so there is no need to multiply the volumetric rate by porosity. This will be done by Fluent internally. The macro `FLUID_THREAD_P(t)` is used to determine if a cell thread is a fluid (rather than a solid) thread. The variable `THREAD_VAR(t).fluid.porous` is used to check if a fluid cell thread is a porous region.

```
/*******************************************************************
   rate.c
 Compiled UDF for specifying a reaction rate in a porous medium
 *****************************************************************/

#include "udf.h"

#define K1 2.0e-2
#define K2 5.

DEFINE_VR_RATE(vol_reac_rate,c,t,r,mole_weight,species_mf,rate,rr_t)
{
   real s1 = species_mf[0];
   real mw1 = mole_weight[0];
   if (FLUID_THREAD_P(t) && THREAD_VAR(t).fluid.porous)
     *rate = K1*s1/pow((1.+K2*s1),2.0)/mw1;
   else
     *rate = 0.;
   *rr_t = *rate;
}
```
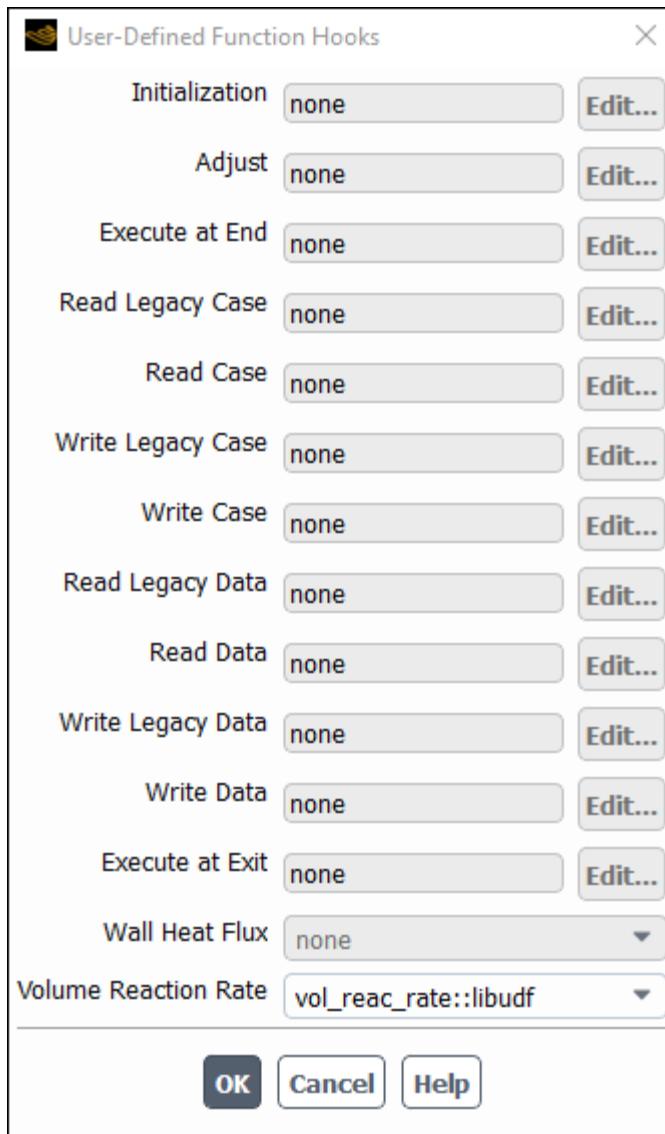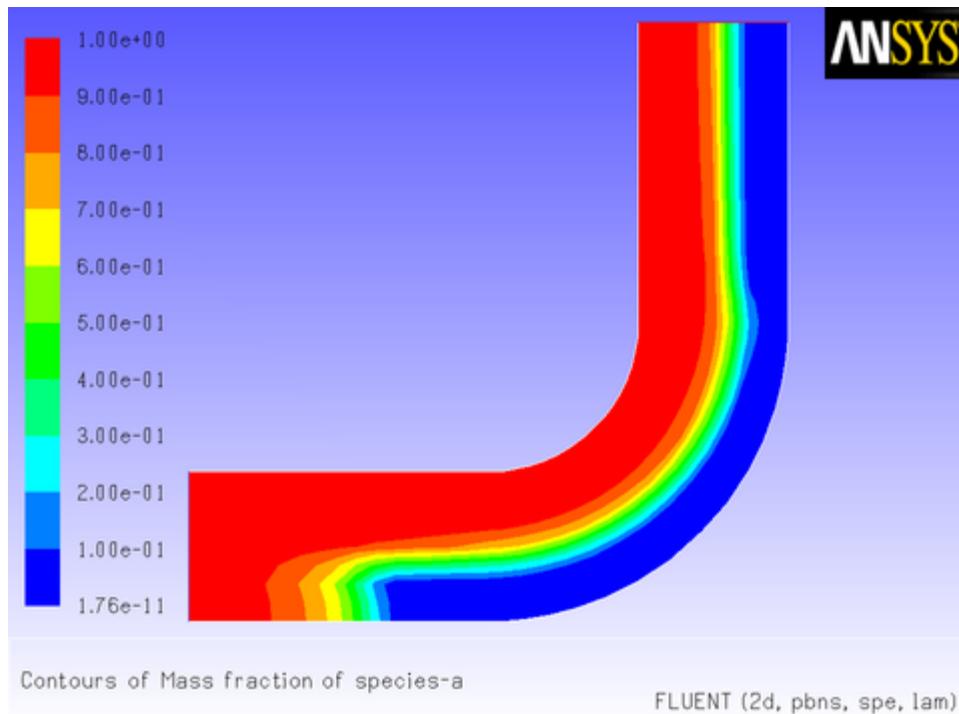
This UDF is executed as a compiled UDF in Ansys Fluent. Follow the procedure for compiling source files using the **Compiled UDFs** dialog box that is described in Compiling a UDF Using the GUI (p. 403).

After the function **vol_reac_rate** is compiled and loaded, you can hook the reaction rate UDF to Ansys Fluent by selecting the function's name in the **Volume Reaction Rate Function** drop-down list in the **User-Defined Function Hooks** dialog box (Figure 6.74: The **User-Defined Function Hooks** Dialog Box (p. 516)).

Parameters & Customization → **User Defined Functions** **Function Hooks...**

Figure 8.30: The User-Defined Functions Hooks Dialog Box

Initialize and run the calculation. The converged solution for the mass fraction of `species-a` is shown in Figure 8.31: Mass Fraction for species-a Governed by a Reaction in a Porous Region (p. 642). The gas that moves through the porous region is gradually converted to `species-b` in the horizontal section of the duct. No reaction takes place in the fluid region, although some diffusion of `species-b` out of the porous region is suggested by the wide transition layer between the regions of 100% and 0% `species-a`.

Figure 8.31: Mass Fraction for species-a Governed by a Reaction in a Porous Region

Contours of Mass fraction of species-a

FLUENT (2d, pbns, spe, lam)

8.2.5. User-Defined Scalars

This section contains examples of UDFs that can be used to customize user-defined scalar (UDS) transport equations. See User-Defined Scalar (UDS) Transport Equation `DEFINE` Macros (p. 295) for information on how you can define UDFs in Ansys Fluent. See User-Defined Scalar (UDS) Transport Equations in the *Fluent Theory Guide* for UDS equation theory and details on how to set up scalar equations.

8.2.5.1. Postprocessing Using User-Defined Scalars

Below is an example of a compiled UDF that computes the gradient of temperature to the fourth power, and stores its magnitude in a user-defined scalar. The computed temperature gradient can, for example, be subsequently used to plot contours. Although the practical application of this UDF is questionable, its purpose here is to show the methodology of computing gradients of arbitrary quantities that can be used for postprocessing.

```
/**********************************************************************/
/* UDF for computing the magnitude of the gradient of T^4            */
/**********************************************************************/

 #include "udf.h"

 /* Define which user-defined scalars to use. */
 enum
 {
    T4,
    MAG_GRAD_T4,
    N_REQUIRED_UDS
 };

 DEFINE_ADJUST(adjust_fcn, domain)
 {
```

```
    Thread *t;
    cell_t c;
    face_t f;

    /* Make sure there are enough user-defined scalars. */
    if (n_uds < N_REQUIRED_UDS)
      Internal_Error("not enough user-defined scalars allocated");
      /* Fill first UDS with temperature raised to fourth power. */
      thread_loop_c (t,domain)
        {
          if (NULL != THREAD_STORAGE(t,SV_UDS_I(T4)))
          {
           begin_c_loop (c,t)
           {
            real T = C_T(c,t);
            C_UDSI(c,t,T4) = pow(T,4.);
           }
           end_c_loop (c,t)
          }
        }

      thread_loop_f (t,domain)
        {
         if (NULL != THREAD_STORAGE(t,SV_UDS_I(T4)))
          {
           begin_f_loop (f,t)
            {
             real T = 0.;
             if (NULL != THREAD_STORAGE(t,SV_T))
               T = F_T(f,t);
             else if (NULL != THREAD_STORAGE(t->t0,SV_T))
               T = C_T(F_C0(f,t),t->t0);
               F_UDSI(f,t,T4) = pow(T,4.);
            }
           end_f_loop (f,t)
          }
        }
      /* Fill second UDS with magnitude of gradient. */
      thread_loop_c (t,domain)
        {
         if (NULL != THREAD_STORAGE(t,SV_UDS_I(T4)) &&
             NULL != T_STORAGE_R_NV(t,SV_UDSI_G(T4)))
          {
           begin_c_loop (c,t)
           {
            C_UDSI(c,t,MAG_GRAD_T4) = NV_MAG(C_UDSI_G(c,t,T4));
           }
           end_c_loop (c,t)
          }
        }
      thread_loop_f (t,domain)
      {
       if (NULL != THREAD_STORAGE(t,SV_UDS_I(T4)) &&
           NULL != T_STORAGE_R_NV(t->t0,SV_UDSI_G(T4)))
        {
         begin_f_loop (f,t)
          {
             F_UDSI(f,t,MAG_GRAD_T4)=C_UDSI(F_C0(f,t),t->t0,MAG_GRAD_T4);
          }
         end_f_loop (f,t)
        }
      }
    }
  }
```

The conditional statement if (NULL != THREAD_STORAGE(t,SV_UDS_I(T4))) is used to check if the storage for the user-defined scalar with index T4 has been allocated, while NULL != T_STORAGE_R_NV(t,SV_UDSI_G(T4)) checks whether the storage of the gradient of the user-defined scalar with index T4 has been allocated.

In addition to compiling this UDF, as described in Compiling UDFs (p. 399), you will need to enable the solution of a user-defined scalar transport equation in Ansys Fluent.

Parameters & Customization → User Defined Scalars → New...

See User-Defined Scalar (UDS) Transport Equations in the *Fluent Theory Guide* for UDS equation theory and details on how to set up scalar equations.

8.2.5.2. Implementing Ansys Fluent's P-1 Radiation Model Using User-Defined Scalars

This section provides an example that demonstrates how the P1 radiation model can be implemented as a UDF, utilizing a user-defined scalar transport equation. In the P1 model, the variation of the incident radiation, G, in the domain can be described by an equation that consists of a diffusion and source term.

The transport equation for incident radiation, G, is given by Equation 8.5 (p. 644). The diffusion coefficient, Γ, is given by Equation 8.6 (p. 644) and the source term is given by Equation 8.7 (p. 644). See P-1 Radiation Model Theory in the *Fluent Theory Guide* for more details.

$$\nabla \cdot (\Gamma \nabla G) + S^G = 0 \tag{8.5}$$

$$\Gamma = \frac{1}{3a + (3 - C)\, \sigma_s} \tag{8.6}$$

$$S^G = a(4\sigma T^4 - G) \tag{8.7}$$

The boundary condition for G at the walls is equal to the negative of the radiative wall heat flux, $q_{r,w}$ (Equation 8.8 (p. 644)), where $\vec{n}$ is the outward normal vector (see P-1 Radiation Model Theory in the *Fluent Theory Guide* for more details). The radiative wall heat flux can be given by Equation 8.9 (p. 644).

$$q_r \cdot \vec{n} = -\Gamma \nabla G \cdot \vec{n} \tag{8.8}$$

$$q_{r,w} = -\frac{\varepsilon_w}{2(2 - \varepsilon_w)} \left(4\sigma T_w^4 - G_w\right) \tag{8.9}$$

This form of the boundary condition is unfortunately specified in terms of the incident radiation at the wall, G_w. This mixed boundary condition can be avoided by solving first for G_w using Equation 8.8 (p. 644) and Equation 8.9 (p. 644), resulting in Equation 8.10 (p. 644). Then, this expression for G_w is substituted back into Equation 8.9 (p. 644) to give the radiative wall heat flux $q_{r,w}$ as Equation 8.11 (p. 644).

$$G_w = \frac{4\sigma T_w^4 E_w + \frac{\alpha_0 \Gamma_0}{A}\left[G_0 - \beta_0(G)\right]}{E_w + \frac{\alpha_0 \Gamma_0}{A}} \tag{8.10}$$

$$q_r = -\frac{\alpha_0 \Gamma_0 E_w}{A\left(E_w + \frac{\alpha_0 \Gamma_0}{A}\right)}\left[4\pi I_b(T_{iw}) - G_0 + \beta_0(G)\right] \tag{8.11}$$

The additional β_0 and G_0 terms that appear in Equation 8.10 (p. 644) and Equation 8.11 (p. 644) are a result of the evaluation of the gradient of incident radiation in Equation 8.8 (p. 644).

In Ansys Fluent, the component of a gradient of a scalar directed normal to a cell boundary (face), $\nabla G \cdot \mathrm{n}$, is estimated as the sum of primary and secondary components. The primary component represents the gradient in the direction defined by the cell centroids, and the secondary component is in the direction along the face separating the two cells. From this information, the face normal component can be determined. The secondary component of the gradient can be found using the Ansys Fluent macro BOUNDARY_SECONDARY_GRADIENT_SOURCE (which is described in Boundary Secondary Gradient Source (BOUNDARY_SECONDARY_GRADIENT_SOURCE) (p. 327)). The use of this macro first requires that cell geometry information be defined, which can be readily obtained by the use of a second macro, BOUNDARY_FACE_GEOMETRY (see Boundary Face Geometry (BOUNDARY_FACE_GEOMETRY) (p. 327)). You will see these macros called in the UDF that defines the wall boundary condition for G.

To complete the implementation of the P1 model, the radiation energy equation must be coupled with the thermal energy equation. This is accomplished by modifying the source term and wall boundary condition of the energy equation. Consider first how the energy equation source term must be modified. The gradient of the incident radiation is proportional to the radiative heat flux. A local increase (or decrease) in the radiative heat flux is attributable to a local decrease (or increase) in thermal energy via the absorption and emission mechanisms. The gradient of the radiative heat flux is therefore a (negative) source of thermal energy. The source term for the incident radiation Equation 8.7 (p. 644) is equal to the gradient of the radiative heat flux and hence its negative specifies the source term needed to modify the energy equation (see P-1 Radiation Model Theory in the *Fluent Theory Guide* for more details).

Now consider how the energy boundary condition at the wall must be modified. Locally, the only mode of energy transfer from the wall to the fluid that is accounted for by default is conduction. With the inclusion of radiation effects, radiative heat transfer to and from the wall must also be accounted for. (This is done automatically if you use Ansys Fluent's built-in P1 model.) The DEFINE_HEAT_FLUX macro allows the wall boundary condition to be modified to accommodate this second mode of heat transfer by specifying the coefficients of the qir equation discussed in DEFINE_HEAT_FLUX (p. 78). The net radiative heat flux to the wall has already been given as Equation 8.9 (p. 644). Comparing this equation with that for qir in DEFINE_HEAT_FLUX (p. 78) will result in the proper coefficients for $cir[\,]$.

In this example, the implementation of the P1 model can be accomplished through six separate UDFs. They are all included in a single source file, which can be executed as a compiled UDF. The single user-defined scalar transport equation for incident radiation, G, uses a DEFINE_DIFFUSIVITY UDF to define Γ of Equation 8.6 (p. 644), and a UDF to define the source term of Equation 8.7 (p. 644). The boundary condition for G at the walls is handled by assigning, in DEFINE_PROFILE, the negative of Equation 8.11 (p. 644) as the specified flux. A DEFINE_ADJUST UDF is used to instruct Ansys Fluent to check that the proper number of user-defined scalars has been defined (in the solver). Lastly, the energy equation must be assigned a source term equal to the negative of that used in the incident radiation equation and the DEFINE_HEAT_FLUX UDF is used to alter the boundary conditions at the walls for the energy equation.

In the solver, at least one user-defined scalar (UDS) equation must be enabled. The scalar diffusivity is assigned in the **Create/Edit Materials** dialog box for the scalar equation. The scalar source and energy source terms are assigned in the boundary condition dialog box for the fluid zones. The boundary condition for the scalar equation at the walls is assigned in the boundary condition dialog box for the wall zones. The DEFINE_ADJUST and DEFINE_HEAT_FLUX functions are assigned in the **User-Defined Function Hooks** dialog box.

Note that the residual monitor for the UDS equation should be reduced from $1e-3$ to $1e-6$ before running the solution. If the solution diverges, then it may be due to the large source terms. In this case, the under-relaxation factor should be reduced to 0.99 and the solution re-run.

```
/****************************************************************/
/* Implementation of the P1 model using user-defined scalars  */
/****************************************************************/

#include "udf.h"
#include "sg.h"

/* Define which user-defined scalars to use. */
enum
{
    P1,
    N_REQUIRED_UDS
};


static real abs_coeff = 0.2; /* absorption coefficient */
static real scat_coeff = 0.0;  /* scattering coefficient */
static real las_coeff = 0.0; /* linear-anisotropic  */
              /* scattering coefficient */
static real epsilon_w = 1.0; /* wall emissivity */


DEFINE_ADJUST(p1_adjust, domain)
{
    /* Make sure there are enough user defined-scalars. */
    if (n_uds < N_REQUIRED_UDS)
      Internal_Error("not enough user-defined scalars allocated");
}


DEFINE_SOURCE(energy_source, c, t, dS, eqn)
{
    dS[eqn] = -16.*abs_coeff*SIGMA_SBC*pow(C_T(c,t),3.);
    return -abs_coeff*(4.*SIGMA_SBC*pow(C_T(c,t),4.) - C_UDSI(c,t,P1)); }

DEFINE_SOURCE(p1_source, c, t, dS, eqn)
{
   dS[eqn] = -abs_coeff;
   return abs_coeff*(4.*SIGMA_SBC*pow(C_T(c,t),4.) - C_UDSI(c,t,P1));
}


DEFINE_DIFFUSIVITY(p1_diffusivity, c, t, i)
{
    return 1./(3.*abs_coeff + (3. - las_coeff)*scat_coeff);
}


DEFINE_PROFILE(p1_bc, thread, position)

{
    face_t f;
    real A[ND_ND],At;
    real dG[ND_ND],dr0[ND_ND],es[ND_ND],ds,A_by_es;
    real aterm,alpha0,beta0,gamma0,Gsource,Ibw;
    real Ew = epsilon_w/(2.*(2. - epsilon_w));
    Thread *t0=thread->t0;
    /* Do nothing if areas are not computed yet or not next to fluid. */
    if (!Data_Valid_P() || !FLUID_THREAD_P(t0)) return;
      begin_f_loop (f,thread)
      {
         cell_t c0 = F_C0(f,thread);
         BOUNDARY_FACE_GEOMETRY(f,thread,A,ds,es,A_by_es,dr0);
         At = NV_MAG(A);
```

```
        if (NULLP(T_STORAGE_R_NV(t0,SV_UDSI_G(P1))))
          Gsource = 0.;   /* if gradient not stored yet */
        else
          BOUNDARY_SECONDARY_GRADIENT_SOURCE(Gsource,SV_UDSI_G(P1),
                 dG,es,A_by_es,1.);
        gamma0 = C_UDSI_DIFF(c0,t0,P1);
        alpha0 = A_by_es/ds;
        beta0 = Gsource/alpha0;
        aterm = alpha0*gamma0/At;
        Ibw = SIGMA_SBC*pow(WALL_TEMP_OUTER(f,thread),4.)/M_PI;
        /* Specify the radiative heat flux. */
        F_PROFILE(f,thread,position) =
             aterm*Ew/(Ew + aterm)*(4.*M_PI*Ibw - C_UDSI(c0,t0,P1) + beta0);
      }
    end_f_loop (f,thread)
}


DEFINE_HEAT_FLUX(heat_flux, f, t, c0, t0, cid, cir)
{
   real Ew = epsilon_w/(2.*(2. - epsilon_w));
   cir[0] = Ew * F_UDSI(f,t,P1);
 cir[3] = 4.0 * Ew * SIGMA_SBC;


}
```

8.2.6. User-Defined Real Gas Models (UDRGM)

This section contains examples of UDFs that can be used to customize user-defined real gas models. See The User-Defined Real Gas Model in the User's Guide for the overview, limitations, and details on how to set up, build and load a library of user-defined real gas functions.

8.2.6.1. UDRGM Example: Redlich-Kwong Equation of State

This section describes another example of a user-defined real gas model. You can use this example as the basis for your own UDRGM code. In this example, the Redlich-Kwong equation of state is used in the UDRGM.

This section summarizes the equations used in developing the UDRGM for the Redlich-Kwong equation of state. The model is based on a modified form of the Redlich-Kwong equation of state described in [1] (p. 713). The equations used in this UDRGM will be listed in the sections below.

The following nomenclature applies to this section:

$a(T)$ = Redlich-Kwong temperature function

c = speed of sound

C_p = specific heat

H = enthalpy

n = exponent in function $a(T)$

p = pressure

R = universal gas constant/molecular weight

T = temperature

S = entropy

$$V = \text{specific volume}$$
$$\rho = \text{density}$$

The superscript 0 designates a reference state, and the subscript c designates a critical point property.

8.2.6.2. Specific Volume and Density

The Redlich-Kwong equation of state can be written in the following form:

$$p=\frac{RT}{\left(V-\tilde{b}\right)}-\frac{a\left(T\right)}{V\left(V+b_0\right)} \tag{8.12}$$

where

$$V=\frac{1}{\rho} \tag{8.13}$$

$$a\left(T\right)=a_0\left(\frac{T_c}{T}\right)^n$$

$$a_0=0.42747\frac{R^2T_c^2}{p_c}$$

$$b_0=0.08664\frac{RT_c}{p_c}$$

$$c_0=\frac{RT_c}{p_c+\frac{a_0}{V_c(V_c+b_0)}}+b_0-V_c$$

$$\tilde{b}=b_0-c$$

Since the real gas model in Ansys Fluent requires a function for density as a function of pressure and temperature, Equation 8.12 (p. 648) must be solved for the specific volume (from which the density can be easily obtained). For convenience, Equation 8.12 (p. 648) can be written as a cubic equation for specific volume as follows:

$$V^3+a_1V^2+a_2V+a_3=0 \tag{8.14}$$

where

$$a_1=c_0-\frac{RT}{p} \tag{8.15}$$

$$a_2=-\left(\tilde{b}b_0+\frac{RTb_0}{p}-\frac{a\left(T\right)}{p}\right)$$

$$a_3=-\frac{a\left(T\right)\tilde{b}}{p}$$

Equation 8.14 (p. 648) is solved using a standard algorithm for cubic equations (see [11] (p. 713) for details). In the UDRGM code, the cubic solution is coded to minimize the number of floating point

operations. This is critical for optimal performance, since this function gets called numerous times during an iteration cycle.

It should be noted that the value of the exponent, n, in the function $a(T)$ will depend on the substance. A table of values can be found in [1] (p. 713) for some common substances. Alternatively, [1] (p. 713) states that values of n are well correlated by the empirical equation

$$n = 0.4986 + 1.1735\omega + 0.475\omega^2 \tag{8.16}$$

where ω is the acentric factor, defined as

$$\omega = -\log\left(\frac{p_v(T)}{p_c}\right) - 1.0 \tag{8.17}$$

In the above equation, $p_v(T)$ is the saturated vapor pressure evaluated at temperature $T = 0.7 T_c$.

8.2.6.3. Derivatives of Specific Volume and Density

The derivatives of specific volume with respect to temperature and pressure can be easily determined from Equation 8.12 (p. 648) using implicit differentiation. The results are presented below:

$$\left(\frac{\partial V}{\partial p}\right)_T = -\frac{(a_1)_p' V^2 + (a_2)_p' V + (a_3)_p'}{3V^2 + 2a_1 V + a_2} \tag{8.18}$$

$$\left(\frac{\partial V}{\partial T}\right)_p = -\frac{(a_1)_T' V^2 + (a_2)_T' V + (a_3)_T'}{3V^2 + 2a_1 V + a_2} \tag{8.19}$$

where

$$(a_1)_p' = \frac{RT}{p^2} \tag{8.20}$$

$$(a_2)_p' = \frac{RTb_0 - a(T)}{p^2}$$

$$(a_3)_p' = \frac{a(T)\tilde{b}}{p^2}$$

$$(a_1)_T' = -\frac{R}{p}$$

$$(a_2)_T' = \frac{-Rb_0 + \frac{da(T)}{dT}}{p}$$

$$(a_3)_T' = -\frac{da(T)}{dT}\frac{\tilde{b}}{p}$$

$$\frac{da(T)}{dT} = -n\frac{a(T)}{T}$$

The derivatives of density can be obtained from the above using the relations

$$\left(\frac{\partial\rho}{\partial p}\right)_T = -\rho^2\left(\frac{\partial V}{\partial p}\right)_T \tag{8.21}$$

$$\left(\frac{\partial\rho}{\partial T}\right)_p = -\rho^2\left(\frac{\partial V}{\partial T}\right)_p \tag{8.22}$$

8.2.6.4. Specific Heat and Enthalpy

Following [1] (p. 713), enthalpy for a real gas can be written

$$H = H^0(T) + pV - RT - \frac{a(T)}{b_0}(1+n)\ln\left(\frac{V+b_0}{V}\right) \tag{8.23}$$

where $H^0(T)$ is the enthalpy function for a thermally perfect gas (that is, enthalpy is a function of temperature alone). In the present case, a fourth-order polynomial for the specific heat for a thermally perfect gas [8] (p. 713) is employed:

$$C_p^0(T) = C_1 + C_2 T + C_3 T^2 + C_4 T^3 + C_5 T^4 \tag{8.24}$$

and obtain the enthalpy from the basic relation

$$H^0(T) = \int_{T_0}^{T} C_p^0(T)\,dT \tag{8.25}$$

The result is

$$H^0(T) = C_1 T + \frac{1}{2}C_2 T^2 + \frac{1}{3}C_3 T^3 + \frac{1}{4}C_4 T^4 + \frac{1}{5}C_5 T^5 - H^0(T^0) \tag{8.26}$$

Note that $H^0(T^0)$ is the enthalpy at a reference state (p^0, T^0), which can be chosen arbitrarily.

The specific heat for the real gas can be obtained by differentiating Equation 8.23 (p. 650) with respect to temperature (at constant pressure): becomes

$$C_p = \left(\frac{\partial H}{\partial T}\right)_p \tag{8.27}$$

The result is

$$C_p = C_p^0(T) + p\left(\frac{\partial V}{\partial T}\right)_p - R - \frac{da(T)}{dT}\frac{(1+n)}{b_0}\ln\left(\frac{V+b_0}{V}\right) + a(T)(1+n)\frac{\left(\frac{\partial V}{\partial T}\right)_p}{V(V+b_0)} \tag{8.28}$$

Finally, the derivative of enthalpy with respect to pressure (at constant temperature) can be obtained using the following thermodynamic relation [8] (p. 713):

$$\left(\frac{\partial H}{\partial p}\right)_T = V - T\left(\frac{\partial V}{\partial T}\right)_p \tag{8.29}$$

8.2.6.5. Entropy

Following [1] (p. 713), the entropy can be expressed in the form

$$S = S^0(T, p^0) + R\ln\left(\frac{V-\check{b}}{V^0}\right) + \frac{\left(\frac{da(T)}{dT}\right)}{b_0}\ln\left(\frac{V+b_0}{V}\right) \tag{8.30}$$

where the superscript 0 again refers to a reference state where the ideal gas law is applicable. For an ideal gas at a fixed reference pressure, p^0, the entropy is given by

$$S^0(T,p^0)=S(T^0,p^0)+\int_{T^0}^{T}\frac{C_p^0(T)}{T}dT \tag{8.31}$$

Note that the pressure term is zero since the entropy is evaluated at the reference pressure. Using the polynomial expression for specific heat, Equation 8.24 (p. 650), Equation 8.31 (p. 651) becomes

$$S^0(T,p^0)=S(T^0,p^0)+C_1\ln(T)+C_2T+\frac{1}{2}C_3T^2+\frac{1}{3}C_4T^3+\frac{1}{4}C_5T^4-f(T^0) \tag{8.32}$$

where $f(T^0)$ is a constant, which can be absorbed into the reference entropy $S(T^0,p^0)$.

8.2.6.6. Speed of Sound

The speed of sound for a real gas can be determined from the thermodynamic relation

$$c^2=\left(\frac{\partial p}{\partial\rho}\right)_S=-\left(\frac{C_p}{C_V}\right)\frac{V^2}{\left(\frac{\partial V}{\partial p}\right)_T} \tag{8.33}$$

Noting that,

$$C_p-C_V=-T\left(\frac{\partial V}{\partial T}\Big|_p\right)^2\frac{\partial p}{\partial v}\Big|_T \tag{8.34}$$

we can write the speed of sound as

$$c=V\sqrt{-\left(\frac{C_p}{C_p-\Delta C}\right)\frac{1}{\left(\frac{\partial V}{\partial p}\right)_T}} \tag{8.35}$$

8.2.6.7. Viscosity and Thermal Conductivity

The dynamic viscosity of a gas or vapor can be estimated using the following formula from [2] (p. 713):

$$\mu(T)=6.3\times10^{-7}\frac{M_w^{0.5}p_c^{0.6666}}{T_c^{0.1666}}\left(\frac{T_r^{1.5}}{T_r+0.8}\right) \tag{8.36}$$

Here, T_r is the reduced temperature

$$T_r=\frac{T}{T_c} \tag{8.37}$$

and M_w is the molecular weight of the gas. This formula neglects the effect of pressure on viscosity, which usually becomes significant only at very high pressures.

Knowing the viscosity, the thermal conductivity can be estimated using the Eucken formula [4] (p. 713):

$$k=\mu\left(C_p+\frac{5}{4}R\right) \tag{8.38}$$

It should be noted that both Equation 8.36 (p. 651) and Equation 8.38 (p. 651) are simple relations, and therefore may not provide satisfactory values of viscosity and thermal conductivity for certain

applications. You are encouraged to modify these functions in the UDRGM source code if alternate formulae are available for a given gas.

8.2.6.8. Using the Redlich-Kwong Real Gas UDRGM

Using the Redlich-Kwong Real Gas UDRGM simply requires the modification of the top block of `#define` macros to provide the appropriate parameters for a given substance. An example listing for CO_2 is given below. The parameters required are:

$\qquad$ `MWT` = Molecular weight of the substance

$\qquad$ `PCRIT` = Critical pressure (Pa)

$\qquad$ `TCRIT` = Critical temperature (K)

$\qquad$ `ZCRIT` = Critical compressibility factor

$\qquad$ `VCRIT` = Critical specific volume (m^3/kg)

$\qquad$ `NRK` = Exponent of $a(T)$ function

$\qquad$ `CC1, CC2, CC3, CC4, CC5` = Coefficients of $C_p(T)$ polynomial curve fit

$\qquad$ `P_REF` = Reference pressure (Pa)

$\qquad$ `T_REF` = Reference temperature (K)

The coefficients for the ideal gas specific heat polynomial were obtained from [8] (p. 713) (coefficients for other substances are also provided in [8] (p. 713)). After the source listing is modified, the UDRGM C code can be recompiled and loaded into Ansys Fluent in the manner described earlier.

```
/* The variables below need to be set for a particular gas */

/* CO2 */

/* REAL GAS EQUATION OF STATE MODEL - BASIC VARIABLES */
/* ALL VARIABLES ARE IN SI UNITS! */

#define RGASU UNIVERSAL_GAS_CONSTANT
#define PI 3.141592654

#define MWT 44.01
#define PCRIT 7.3834e6
#define TCRIT 304.21
#define ZCRIT 0.2769
#define VCRIT 2.15517e-3
#define NRK 0.77

/* IDEAL GAS SPECIFIC HEAT CURVE FIT */

#define CC1 453.577
#define CC2 1.65014
#define CC3 -1.24814e-3
#define CC4 3.78201e-7
#define CC5 0.00

/* REFERENCE STATE */

#define P_REF 101325
#define T_REF 288.15
```

8.2.6.9. Redlich-Kwong Real Gas UDRGM Code Listing

```c
/****************************************************************/
/*                                                          */
/* User-Defined Function: Redlich-Kwong Equation of State   */
/*          for Real Gas Modeling                           */
/*                                                          */
/* Author: Frank Kelecy                                     */
/*  Date: May 2003                                          */
/* Version: 1.02                                            */
/*                                                          */
/* This implementation is completely general.              */
/* Parameters set for CO2.                                 */
/*                                                          */
/****************************************************************/

#include "udf.h"
#include "stdio.h"
#include "ctype.h"
#include "stdarg.h"

/* The variables below need to be set for a particular gas */

/* CO2 */

/* REAL GAS EQUATION OF STATE MODEL - BASIC VARIABLES */
/* ALL VARIABLES ARE IN SI UNITS! */

#define RGASU UNIVERSAL_GAS_CONSTANT
#define PI  3.141592654

#define MWT 44.01
#define PCRIT 7.3834e6
#define TCRIT 304.21
#define ZCRIT 0.2769
#define VCRIT 2.15517e-3
#define NRK 0.77

/* IDEAL GAS SPECIFIC HEAT CURVE FIT */

#define CC1 453.577
#define CC2 1.65014
#define CC3 -1.24814e-3
#define CC4 3.78201e-7
#define CC5 0.00

/* REFERENCE STATE */

#define P_REF 101325
#define T_REF 288.15

/* OPTIONAL REFERENCE (OFFSET) VALUES FOR ENTHALPY AND ENTROPY */

#define H_REF 0.0
#define S_REF 0.0

static int (*usersMessage)(const char *,...);
static void (*usersError)(const char *,...);

/* Static variables associated with Redlich-Kwong Model */

static double rgas, a0, b0, c0, bb, cp_int_ref;


DEFINE_ON_DEMAND(I_do_nothing)
{
   /* this is a dummy function to allow us */
   /* to use the compiled UDFs utility   */
}
```

```
/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_Setup                                    */
/*----------------------------------------------------------*/

void RKEOS_Setup(Domain *domain, cxboolean vapor_phase, char *species_list,
                 int(*messagefunc)(const char *format,...),
                 void (*errorfunc)(const char *format,...))
{

  rgas = RGASU/MWT;
  a0 = 0.42747*rgas*rgas*TCRIT*TCRIT/PCRIT;
  b0 = 0.08664*rgas*TCRIT/PCRIT;
  c0 = rgas*TCRIT/(PCRIT+a0/(VCRIT*(VCRIT+b0)))+b0-VCRIT;
  bb = b0-c0;
  cp_int_ref = CC1*log(T_REF)+T_REF*(CC2+
               T_REF*(0.5*CC3+T_REF*(0.333333*CC4+0.25*CC5*T_REF)));

  usersMessage = messagefunc;
  usersError = errorfunc;
  usersMessage("\nLoading Redlich-Kwong Library: %s\n", species_list);
}

/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_pressure                                 */
/*      Returns pressure given T and density               */
/*----------------------------------------------------------*/

double RKEOS_pressure(double temp, double density)
{
    double v = 1./density;
    double afun = a0*pow(TCRIT/temp,NRK);
    return rgas*temp/(v-bb)-afun/(v*(v+b0));
}

/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_spvol                                    */
/*      Returns specific volume given T and P               */
/*----------------------------------------------------------*/

double RKEOS_spvol(double temp, double press)
{
    double a1,a2,a3;
    double vv,vv1,vv2,vv3;
    double qq,qq3,sqq,rr,tt,dd;
    double afun = a0*pow(TCRIT/temp,NRK);

    a1 = c0-rgas*temp/press;
    a2 = -(bb*b0+rgas*temp*b0/press-afun/press);
    a3 = -afun*bb/press;

    /* Solve cubic equation for specific volume */

    qq = (a1*a1-3.*a2)/9.;
    rr = (2*a1*a1*a1-9.*a1*a2+27.*a3)/54.;
    qq3 = qq*qq*qq;
    dd = qq3-rr*rr;

  /* If dd < 0, then we have one real root */
  /* If dd >= 0, then we have three roots -> choose largest root */

  if (dd < 0.) {
     tt = pow(sqrt(-dd)+fabs(rr),0.333333);
     vv = (tt+qq/tt)-a1/3.;
  } else {
     tt = acos(rr/sqrt(qq3));
     sqq = sqrt(qq);
     vv1 = -2.*sqq*cos(tt/3.)-a1/3.;
     vv2 = -2.*sqq*cos((tt+2.*PI)/3.)-a1/3.;
     vv3 = -2.*sqq*cos((tt+4.*PI)/3.)-a1/3.;
     vv = (vv1 > vv2) ? vv1 : vv2;
```

```
      vv = (vv > vv3) ? vv : vv3;
  }

  return vv;
}

/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_density                                  */
/*      Returns density given T and P                       */
/*----------------------------------------------------------*/

double RKEOS_density(cell_t cell, Thread *thread, cxboolean vapor_phase, double temp, double press, double yi[]
{
    return 1./RKEOS_spvol(temp, press); /* (Kg/m3) */
}

/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_dvdp                                     */
/*      Returns dv/dp given T and rho                       */
/*----------------------------------------------------------*/

double RKEOS_dvdp(double temp, double density)
{
    double a1,a2,a1p,a2p,a3p,v,press;
    double afun = a0*pow(TCRIT/temp,NRK);

    press = RKEOS_pressure(temp, density);
    v = 1./density;

    a1 = c0-rgas*temp/press;
    a2 = -(bb*b0+rgas*temp*b0/press-afun/press);
    a1p = rgas*temp/(press*press);
    a2p = a1p*b0-afun/(press*press);
    a3p = afun*bb/(press*press);

    return -(a3p+v*(a2p+v*a1p))/(a2+v*(2.*a1+3.*v));
}

/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_dvdt                                     */
/*      Returns dv/dT given T and rho                       */
/*----------------------------------------------------------*/

double RKEOS_dvdt(double temp, double density)
{
    double a1,a2,dadt,a1t,a2t,a3t,v,press;
    double afun = a0*pow(TCRIT/temp,NRK);

    press = RKEOS_pressure(temp, density);
    v = 1./density;

    dadt = -NRK*afun/temp;
    a1 = c0-rgas*temp/press;
    a2 = -(bb*b0+rgas*temp*b0/press-afun/press);
    a1t = -rgas/press;
    a2t = a1t*b0+dadt/press;
    a3t = -dadt*bb/press;

    return -(a3t+v*(a2t+v*a1t))/(a2+v*(2.*a1+3.*v));
}

/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_Cp_ideal_gas                             */
/*      Returns ideal gas specific heat given T             */
/*----------------------------------------------------------*/

double RKEOS_Cp_ideal_gas(double temp)
{
    return (CC1+temp*(CC2+temp*(CC3+temp*(CC4+temp*CC5))));
}
```

```
/*-------------------------------------------------------------*/
/* FUNCTION: RKEOS_H_ideal_gas                             */
/*      Returns ideal gas specific enthalpy given T        */
/*-------------------------------------------------------------*/

double RKEOS_H_ideal_gas(double temp)
{
    return temp*(CC1+temp*(0.5*CC2+temp*(0.333333*CC3+
             temp*(0.25*CC4+temp*0.2*CC5))));
}

/*-------------------------------------------------------------*/
/* FUNCTION: RKEOS_specific_heat                           */
/*      Returns specific heat given T and rho              */
/*-------------------------------------------------------------*/

double RKEOS_specific_heat(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
    double delta_Cp,press,v,dvdt,dadt;
    double afun = a0*pow(TCRIT/temp,NRK);

    press = RKEOS_pressure(temp, density);
    v = 1./density;
    dvdt = RKEOS_dvdt(temp, density);
    dadt = -NRK*afun/temp;
    delta_Cp = press*dvdt-rgas-dadt*(1.+NRK)/b0*log((v+b0)/v)
             + afun*(1.+NRK)*dvdt/(v*(v+b0));

    return RKEOS_Cp_ideal_gas(temp)+delta_Cp; /* (J/Kg-K) */
}

/*-------------------------------------------------------------*/
/* FUNCTION: RKEOS_enthalpy                                */
/*      Returns specific enthalpy given T and rho          */
/*-------------------------------------------------------------*/

double RKEOS_enthalpy(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
    double delta_h,press, v;
    double afun = a0*pow(TCRIT/temp,NRK);

    press = RKEOS_pressure(temp, density);
    v = 1./density;
    delta_h = press*v-rgas*temp-afun*(1+NRK)/b0*log((v+b0)/v);

    return H_REF+RKEOS_H_ideal_gas(temp)+delta_h; /* (J/Kg) */
}

/*-------------------------------------------------------------*/
/* FUNCTION: RKEOS_entropy                                 */
/*      Returns entropy given T and rho                    */
/*-------------------------------------------------------------*/

double RKEOS_entropy(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
    double delta_s,v,v0,dadt,cp_integral;
    double afun = a0*pow(TCRIT/temp,NRK);

    cp_integral = CC1*log(temp)+temp*(CC2+temp*(0.5*CC3+
             temp*(0.333333*CC4+0.25*CC5*temp)))
             - cp_int_ref;
    v = 1./density;
    v0 = rgas*temp/P_REF;
    dadt = -NRK*afun/temp;
    delta_s = rgas*log((v-bb)/v0)+dadt/b0*log((v+b0)/v);

    return S_REF+cp_integral+delta_s; /* (J/Kg-K) */
}

/*-------------------------------------------------------------*/
/* FUNCTION: RKEOS_mw                                      */
```

```
/*      Returns molecular weight                                 */
/*------------------------------------------------------------*/

double RKEOS_mw(double yi[])
{
    return MWT; /* (Kg/Kmol) */
}


/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_speed_of_sound                              */
/*      Returns s.o.s given T and rho                          */
/*------------------------------------------------------------*/

double RKEOS_speed_of_sound(cell_t cell, Thread *thread, double temp, double density, double P,
       double yi[])
{
    double cp = RKEOS_specific_heat(cell, thread, temp, density, P, yi);
    double dvdt = RKEOS_dvdt(temp, density);
    double dvdp = RKEOS_dvdp(temp, density);
    double v = 1./density;
    double delta_c = -temp*dvdt*dvdt/dvdp;

    return sqrt(cp/((delta_c-cp)*dvdp))*v; /* m/s */
}


/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_rho_t                                       */
/*------------------------------------------------------------*/

double RKEOS_rho_t(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
    return -density*density*RKEOS_dvdt(temp, density);
}


/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_rho_p                                       */
/*------------------------------------------------------------*/

double RKEOS_rho_p(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
    return -density*density*RKEOS_dvdp(temp, density);
}


/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_enthalpy_t                                  */
/*------------------------------------------------------------*/

double RKEOS_enthalpy_t(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
    return RKEOS_specific_heat(cell, thread, temp, density, P, yi);
}


/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_enthalpy_p                                  */
/*------------------------------------------------------------*/

double RKEOS_enthalpy_p(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
    double v = 1./density;
    double dvdt = RKEOS_dvdt(temp, density);

    return v-temp*dvdt;
}


/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_viscosity                                   */
/*------------------------------------------------------------*/

double RKEOS_viscosity(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
    double mu,tr,tc,pcatm;
```

```
    tr = temp/TCRIT;
    tc = TCRIT;
    pcatm = PCRIT/101325.;

    mu = 6.3e-7*sqrt(MWT)*pow(pcatm,0.6666)/pow(tc,0.16666)*
        (pow(tr,1.5)/(tr+0.8));

    return mu;
}


/*-----------------------------------------------------------*/
/* FUNCTION: RKEOS_thermal_conductivity                      */
/*-----------------------------------------------------------*/

double RKEOS_thermal_conductivity(cell_t cell, Thread *thread, double temp,
        double density, double P, double yi[])
{
    double cp, mu;

    cp = RKEOS_Cp_ideal_gas(temp);
    mu = RKEOS_viscosity(cell, thread, temp, density, P, yi);

    return (cp+1.25*rgas)*mu;
}

/* Export Real Gas Functions to Solver */

UDF_EXPORT RGAS_Functions RealGasFunctionList =
{
    RKEOS_Setup,                 /* initialize          */
    RKEOS_density,               /* density             */
    RKEOS_enthalpy,              /* enthalpy            */
    RKEOS_entropy,               /* entropy             */
    RKEOS_specific_heat,         /* specific_heat       */
    RKEOS_mw,                    /* molecular_weight    */
    RKEOS_speed_of_sound,        /* speed_of_sound      */
    RKEOS_viscosity,             /* viscosity           */
    RKEOS_thermal_conductivity,  /* thermal_conductivity */
    RKEOS_rho_t,                 /* drho/dT |const p    */
    RKEOS_rho_p,                 /* drho/dp |const T    */
    RKEOS_enthalpy_t,            /* dh/dT |const p      */
    RKEOS_enthalpy_p             /* dh/dp |const T      */
};
```

8.2.6.9.1. UDRGM Example: Multiple-Species Real Gas Model

This is a simple example for multiple-species real gas models that provide you with a template that you can use to write a more complex multiple-species UDRGM.

In this example, a fluid material is defined in the setup function as a mixture of four species (H20, N2, O2, CO2). The equation of state was the simple ideal gas equation of state. The other thermodynamic properties were defined by an ideal-gas mixing law.

Other auxiliary functions are written to provide individual species property to the principle function set.

The example also provide numerical method of computing $\frac{d\rho}{dT}$, $\frac{d\rho}{dp}$, $\frac{dh}{dT}$, and $\frac{dh}{dp}$.

```
/*
 *sccs id: @(#)real_ideal.c 1.10 Copyright 1900/11/09 Ansys, Inc.
 */

/*
 *    Copyright 1988-1998 Ansys, Inc.
```

```
 *    All Rights Reserved
 *
 *    This is unpublished proprietary source code of Ansys, Inc.
 *    It is protected by U.S. copyright law as an unpublished work
 *    and is furnished pursuant to a written license agreement. It
 *    is considered by Ansys, Inc. to be confidential and may not be
 *    used, copied, or disclosed to others except in accordance with
 *    the terms and conditions of the license agreement.
 */


/*
 * Windows Warning!!! Including udf.h is for getting definitions for
 *          Ansys FLUENT constructs such as Domain. You must
 *          NOT reference any Ansys FLUENT globals directly from
 *          within this module nor link this against any Ansys
 *          FLUENT libs, doing so will cause dependencies on a
 *          specific Ansys FLUENT binary such as fl551.exe and
 *          thus won't be version independent.
 */

#include "udf.h"
#include "stdio.h"
#include "ctype.h"
#include "stdarg.h"

#if RP_DOUBLE
#define SMALL 1.e-20
#else
#define SMALL 1.e-10
#endif

#define NCMAX 20
#define NSPECIE_NAME 80


static int (*usersMessage)(const char *,...);
static void (*usersError)(const char *,...);


static double ref_p, ref_T;
static char   gas[NCMAX][NSPECIE_NAME];
static int   n_specs;

double Mixture_Rgas(double yi[]);
double Mixture_pressure(double Temp, double Rho, double yi[]);
double Mw_i(int i);
double Cp_i(double T, double r, int i);
double K_i(double T, double r, int i);
double Mu_i(double T, double r, int i);
double Rgas_i(double T, double r, int i);
double Gm_i(double T, double r, int i);


DEFINE_ON_DEMAND(I_do_nothing)
{
 /*
  This is a dummy function
  must be included to allow for the use of the
  Ansys FLUENT UDF compilation utility
 */
}


/********************************************************************/
/* Mixture Functions                                               */
/* These are the only functions called from ANSYS FLUENT Code      */
/********************************************************************/
void MIXTURE_Setup(Domain *domain, cxboolean vapor_phase, char *specielist,
          int (*messagefunc)(const char *format,...),
          void (*errorfunc)(const char *format,...))
```

```
{
 /* This function will be called from ANSYS FLUENT after the
    UDF library has been loaded.
    User must enter the number of species in the mixture
    and the name of the individual species.
 */

 int i;
 usersMessage = messagefunc;
 usersError  = errorfunc;
 ref_p    = ABS_P(RP_Get_Real("reference-pressure"),op_pres);
 ref_T    = RP_Get_Real("reference-temperature");

 if (ref_p == 0.0)
  {
   Message0("\n MIXTURE_Setup: reference-pressure was not set by user \n");
   Message0("\n MIXTURE_Setup: setting reference-pressure to 101325 Pa \n");
   ref_p = 101325.0;
  }
/*===================================================*/
/*=========  User Input Section =====================*/
/*===================================================*/
/*
 Define Number of species & Species name.
 DO NOT use space for naming species
*/
n_specs = 4;

(void)strcpy(gas[0],"H2O");
(void)strcpy(gas[1],"N2") ;
(void)strcpy(gas[2],"O2") ;
(void)strcpy(gas[3],"CO2");

 /*===================================================*/
 /*=========  End Of User Input Section ===============*/
 /*===================================================*/

 Message0("\n MIXTURE_Setup: RealGas mixture initialization \n");
 Message0("\n MIXTURE_Setup: Number of Species = %d \n",n_specs);
 for (i=0; i<n_specs; ++i)
  {
   Message0("\n MIXTURE_Setup: Specie[%d]    = %s \n",i,gas[i]);
  }

 /*
  concatenate species name into one string
  and send back to fluent
 */
 strcat(specielist,gas[0]);
 for (i=1; i<n_specs; ++i)
  {
     strcat(specielist," ");
     strcat(specielist,gas[i]);
  }
}

double MIXTURE_density(cell_t cell, Thread *thread, cxboolean vapor_phase, double Temp, double P, double yi[]
{
 double Rgas = Mixture_Rgas(yi);

 double r  = P/(Rgas*Temp); /* Density at Temp & P */

 return r;   /* (Kg/m^3) */
}

double MIXTURE_specific_heat(cell_t cell, Thread *thread, double Temp, double density, double P,
        double yi[])
{
 double cp=0.0;
 int i;
```

```
 for (i=0; i<n_specs; ++i)
    cp += yi[i]*Cp_i(Temp,density,i);

 return cp;   /* (J/Kg/K) */
}

double MIXTURE_enthalpy(cell_t cell, Thread *thread, double Temp, double density, double P, double yi[])
{
 double h=0.0;
 int i;

 for (i=0; i<n_specs; ++i)
    h += yi[i]*(Temp*Cp_i(Temp,density,i));

 return h;    /* (J/Kg) */
}

double MIXTURE_entropy(cell_t cell, Thread *thread, double Temp, double density, double P, double yi[])
{
 double s  = 0.0 ;
 double Rgas=0.0;

 Rgas = Mixture_Rgas(yi);

 s  = MIXTURE_specific_heat(cell, thread, Temp,density,P,yi)*log(Temp/ref_T) -
     Rgas*log(P/ref_p);

 return s;    /* (J/Kg/K) */
}

double MIXTURE_mw(double yi[])
{
 double MW, sum=0.0;
 int i;

 for (i=0; i<n_specs; ++i)
    sum += (yi[i]/Mw_i(i));

 MW = 1.0/MAX(sum,SMALL)   ;

 return MW;    /* (Kg/Kmol) */
}

double MIXTURE_speed_of_sound(cell_t cell, Thread *thread, double Temp, double density, double P,
                double yi[])
{
 double a, cp, Rgas;

 cp  = MIXTURE_specific_heat(cell, thread, Temp,density,P,yi);
 Rgas = Mixture_Rgas(yi);


 a  = sqrt(Rgas*Temp* cp/(cp-Rgas));

 return a; /* m/s */
}

double MIXTURE_viscosity(cell_t cell, Thread *thread, double Temp, double density, double P, double yi[])
{
 double mu=0;
 int i;

 for (i=0; i<n_specs; ++i)
    mu += yi[i]*Mu_i(Temp,density,i);

 return mu;    /* (Kg/m/s) */
}

double MIXTURE_thermal_conductivity(cell_t cell, Thread *thread, double Temp, double density, double P,
                  double yi[])
{
```

```
 double kt=0;
 int i;

 for (i=0; i<n_specs; ++i)
    kt += yi[i]*K_i(Temp,density,i);

 return kt;   /* W/m/K */
}

double MIXTURE_rho_t(cell_t cell, Thread *thread, double Temp, double density, double P, double yi[])
{

 double drdT ; /* derivative of rho w.r.t. Temp */
 double p    ;
 double dT=0.01;

 p    = Mixture_pressure(Temp,density, yi);
 drdT = (MIXTURE_density(cell, thread, TRUE,Temp+dT,p,yi) - MIXTURE_density(cell, thread, TRUE,Temp,p,yi)) /d

 return drdT;    /* (Kg/m^3/K) */
}

double MIXTURE_rho_p(cell_t cell, Thread *thread, double Temp, double density, double P, double yi[])
{
 double drdp  ;
 double p    ;
 double dp= 5.0;

 p    = Mixture_pressure(Temp,density, yi);
 drdp = (MIXTURE_density(cell, thread, TRUE,Temp,p+dp,yi) - MIXTURE_density(cell, thread, TRUE,Temp,p,yi)) /d

 return drdp;    /* (Kg/m^3/Pa) */
}

double MIXTURE_enthalpy_t(cell_t cell, Thread *thread, double Temp, double density, double P, double yi[])
{
 double dhdT   ;
 double p     ;
 double rho2  ;
 double dT= 0.01;

 p    = Mixture_pressure(Temp,density, yi);
 rho2 = MIXTURE_density(cell, thread, TRUE,Temp+dT,p,yi)  ;

 dhdT = (MIXTURE_enthalpy(cell, thread, Temp+dT,rho2,P,yi) - MIXTURE_enthalpy(cell, thread, Temp,
      density,P,yi)) /dT;



 return dhdT; /* J/(Kg.K) */
}

double MIXTURE_enthalpy_p(cell_t cell, Thread *thread, double Temp, double density, double P, double yi[])
{
 double dhdp  ;
 double p     ;
 double rho2  ;
 double dp= 5.0 ;

 p    = Mixture_pressure(Temp,density, yi);
 rho2 = MIXTURE_density(cell, thread, TRUE,Temp,p+dp,yi)  ;

 dhdp = (MIXTURE_enthalpy(cell, thread, Temp,rho2,P,yi) - MIXTURE_enthalpy(cell, thread, Temp,density,
      P,yi)) /dp;



 return dhdp; /* J/ (Kg.Pascal) */
}


/****************************************************************/
/* Auxiliary Mixture Functions                              */
```

```
/******************************************************************/

double Mixture_Rgas(double yi[])
{
 double Rgas=0.0;
 int i;

 for (i=0; i<n_specs; ++i)
    Rgas += yi[i]*(UNIVERSAL_GAS_CONSTANT/Mw_i(i));

 return Rgas;
}

double Mixture_pressure(double Temp, double Rho, double yi[])
{
 double Rgas = Mixture_Rgas(yi);

 double P  = Rho*Rgas*Temp  ; /* Pressure at Temp & P */

 return P;    /* (Kg/m^3) */
}


/******************************************************************/
/* Species Property Functions                                     */
/******************************************************************/
double Mw_i(int i)
{
 double mi[20];

 mi[0] = 18.01534; /*H2O*/
 mi[1] = 28.01340; /*N2 */
 mi[2] = 31.99880; /*O2 */
 mi[3] = 44.00995; /*CO2*/

 return mi[i];
}

double Cp_i(double T, double r, int i)
{
 double cpi[20];

 cpi[0] = 2014.00; /*H2O*/
 cpi[1] = 1040.67; /*N2 */
 cpi[2] = 919.31; /*O2 */
 cpi[3] = 840.37; /*CO2*/

 return cpi[i];
}

double K_i(double T, double r, int i)
{
 double ki[20];

 ki[0] = 0.02610; /*H2O*/
 ki[1] = 0.02420; /*N2 */
 ki[2] = 0.02460; /*O2 */
 ki[3] = 0.01450; /*CO2*/

 return ki[i];
}

double Mu_i(double T, double r, int i)
{
 double mui[20];

 mui[0] = 1.340E-05; /*H2O*/
 mui[1] = 1.663E-05; /*N2 */
 mui[2] = 1.919E-05; /*O2 */
 mui[3] = 1.370E-05; /*CO2*/
```

```
 return mui[i];
}

double Rgas_i(double T, double r, int i)
{
 double Rgasi;

 Rgasi = UNIVERSAL_GAS_CONSTANT/Mw_i(i);

 return Rgasi;
}

double Gm_i(double T, double r, int i)
{
 double gammai;

 gammai = Cp_i(T,r,i)/(Cp_i(T,r,i) - Rgas_i(T,r,i));

 return gammai;

}


/*******************************************************************/
/* Mixture Functions Structure                                   */
/*******************************************************************/
UDF_EXPORT RGAS_Functions RealGasFunctionList =
{
    MIXTURE_Setup,                       /* initialize            */
    MIXTURE_density,                     /* density               */
    MIXTURE_enthalpy,                    /* enthalpy              */
    MIXTURE_entropy,                     /* entropy               */
    MIXTURE_specific_heat,               /* specific_heat         */
    MIXTURE_mw,                          /* molecular_weight      */
    MIXTURE_speed_of_sound,              /* speed_of_sound        */
    MIXTURE_viscosity,                   /* viscosity             */
    MIXTURE_thermal_conductivity,        /* thermal_conductivity  */
    MIXTURE_rho_t,                       /* drho/dT |const p      */
    MIXTURE_rho_p,                       /* drho/dp |const T      */
    MIXTURE_enthalpy_t,                  /* dh/dT |const p        */
    MIXTURE_enthalpy_p                   /* dh/dp |const T        */
};

/*******************************************************************/
/*******************************************************************/
```

8.2.6.9.2. UDRGM Example: Real Gas Model with Volumetric Reactions

This is an example of a UDRGM that has been set up for reacting flow simulations. The example UDF code consists of the following sections:

- Definitions and constants for the physical properties of the species in the single-step methane/air reaction mixture (CH_4, O_2, N_2, CO_2, H_2O).

- Functions of the Redlich-Kwong equation of state for the individual species property calculations.

- Functions for the mixture properties. In this example, the mixture properties are computed assuming ideal gas mixing rules.

Important:

In the UDRGM only the mixture species and associated properties are defined. No information about the chemical reactions is required in the UDF. The chemical reaction is set up in a separate step, after the UDF has been compiled and loaded into Ansys Fluent. See Defining Reactions in the *Fluent User's Guide* for details.

```c
/*
   *sccs id: @(#)real_ideal.c 1.10 Copyright 1900/11/09 ANSYS, Inc.
 */

/*
 *   Copyright 1988-1998 ANSYS, Inc.
 *   All Rights Reserved
 *
 *   This is unpublished proprietary source code of ANSYS, Inc.
 *   It is protected by U.S. copyright law as an unpublished work
 *   and is furnished pursuant to a written license agreement. It
 *   is considered by ANSYS, Inc. to be confidential and may not be
 *   used, copied, or disclosed to others except in accordance with
 *   the terms and conditions of the license agreement.
 */


/*
 *  Warning!!! Including udf.h is for getting definitions for
 *          ANSYS FLUENT constructs such as Domain.  You must
 *          NOT reference any ANSYS FLUENT globals directly from
 *          within this module nor link this against any ANSYS
 *          FLUENT libs, doing so will cause dependencies on a
 *          specific ANSYS FLUENT binary such as fl551.exe and
 *          thus won't be version independent.
 */

#include "udf.h"
#include "stdio.h"
#include "ctype.h"
#include "stdarg.h"

#if RP_DOUBLE
#define SMLL 1.e-20
#else
#define SMLL 1.e-10
#endif

#define NSPECIE_NAME 80
#define RGASU UNIVERSAL_GAS_CONSTANT /* 8314.34 SI units: J/Kmol/K */
#define PI  3.141592654
/* Here input the number of species in the mixture */
/* THIS IS A USER INPUT */
#define n_specs 5

static int (*usersMessage)(const char *,...);
static void (*usersError)(const char *,...);

static double ref_p, ref_T;
static char  gas[n_specs][NSPECIE_NAME];

/* static property parameters */
static double cp[5][n_specs]; /* specific heat polynomial coefficients */
static double mw[n_specs];  /* molecular weights */
static double hf[n_specs];  /* formation enthalpy */
static double tcrit[n_specs]; /* critical temperature */
```

```
static double pcrit[n_specs]; /* critical pressure */
static double vcrit[n_specs]; /* critical specific volume */
static double nrk[n_specs];   /* exponent n of function a(T) in Redlich-Kwong
                   equation of state */
static double omega[n_specs]; /* acentric factor */

/* Static variables associated with Redlich-Kwong Model */
static double rgas[n_specs], a0[n_specs], b0[n_specs], c0[n_specs],
 bb[n_specs], cp_int_ref[n_specs];

void Mw();
void Cp_Parameters();
void Hform();
void Tcrit();
void Pcrit();
void Vcrit();
void NRK();
void Omega();

double RKEOS_spvol(double temp, double press, int i);
double RKEOS_dvdp(double temp, double density, int i);
double RKEOS_dvdt(double temp, double density, int i);
double RKEOS_H_ideal_gas(double temp, int i);
double RKEOS_specific_heat(double temp, double density, int i);
double RKEOS_enthalpy(double temp, double density, int i);
double RKEOS_entropy(double temp, double density, int i);
double RKEOS_viscosity(double temp, int i);
double RKEOS_thermal_conductivity(double temp, int i);
double RKEOS_vol_specific_heat(double temp, double density, int i);

DEFINE_ON_DEMAND(I_do_nothing)
{
 /*
  This is a dummy function
  must be included to allow for the use of the
  ANSYS FLUENT UDF compilation utility
 */
}

/*******************************************************************/
/* Mixture Functions                                               */
/* These are the only functions called from ANSYS FLUENT Code      */
/*******************************************************************/
void MIXTURE_Setup(Domain *domain, cxboolean vapor_phase, char *specielist,
          int (*messagefunc)(const char *format,...),
          void (*errorfunc)(const char *format,...))
{
 /* This function will be called from ANSYS FLUENT after the
   UDF library has been loaded.
   User must enter the number of species in the mixture
   and the name of the individual species.
 */

 int i;
 usersMessage = messagefunc;
 usersError  = errorfunc;
 ref_p    = ABS_P(RP_Get_Real("reference-pressure"),op_pres);
 ref_T    = 298.15;

 Message0("\n MIXTURE_Setup: Redlich-Kwong equation of State"
     " with ideal-gas mixing rules \n");
 Message0("\n MIXTURE_Setup: reference-temperature is %f \n", ref_T);

 if (ref_p == 0.0)
  {
     Message0("\n MIXTURE_Setup: reference-pressure was not set by user \n");
     Message0("\n MIXTURE_Setup: setting reference-pressure to 101325 Pa \n");
     ref_p = 101325.0;
  }
 /*=====================================================*/
 /*========= User Input Section =====================*/
```

```
/*===================================================*/
/*
 Define Species name.
 DO NOT use space for naming species
*/
(void)strcpy(gas[0],"H2O");
(void)strcpy(gas[1],"CH4");
(void)strcpy(gas[2],"O2") ;
(void)strcpy(gas[3],"CO2");
(void)strcpy(gas[4],"N2") ;

/*===================================================*/
/*=========  End Of User Input Section ===============*/
/*===================================================*/

Message0("\n MIXTURE_Setup: RealGas mixture initialization \n");
Message0("\n MIXTURE_Setup: Number of Species = %d \n",n_specs);
for (i=0; i<n_specs; ++i)
  {
     Message0("\n MIXTURE_Setup: Specie[%d]     = %s \n",i,gas[i]);
  }

/*
 concatenate species name into one string
 and send back to fluent
*/
strcat(specielist,gas[0]);
for (i=1; i<n_specs; ++i)
  {
     strcat(specielist," ");
     strcat(specielist,gas[i]);
  }

/* initialize */
Mw();
Cp_Parameters();
Hform();
Tcrit();
Pcrit();
Vcrit();
Omega();
NRK();

for (i=0; i<n_specs; ++i)
  {
   rgas[i] = RGASU/mw[i];
   a0[i] = 0.42747*rgas[i]*rgas[i]*tcrit[i]*tcrit[i]/pcrit[i];
   b0[i] = 0.08664*rgas[i]*tcrit[i]/pcrit[i];
   c0[i] = rgas[i]*tcrit[i]/(pcrit[i]+a0[i]/(vcrit[i]*(vcrit[i]+b0[i])))
       +b0[i]-vcrit[i];
   bb[i] = b0[i]-c0[i];
   cp_int_ref[i] = cp[0][i]*log(ref_T)+ref_T*(cp[1][i]+ref_T*(0.5*cp[2][i]
          +ref_T*(0.333333*cp[3][i]+0.25*cp[4][i]*ref_T)));
  }
}

double MIXTURE_mw(double yi[])
{
 double MW, sum=0.0;
 int i;

 for (i=0; i<n_specs; ++i)
    sum += yi[i]/mw[i];

 MW = 1.0/MAX(sum,SMLL)   ;

 return MW;    /* (Kg/Kmol) */
}

double MIXTURE_density(cell_t cell, Thread *thread, cxboolean vapor_phase, double temp, double P, double yi[])
{
```

```
 double den=0.0;
 int i;

 for (i=0; i<n_specs; ++i)
  {
     if (yi[i]> SMLL)
        den += yi[i]*RKEOS_spvol(temp, P, i);
  }

 return 1./den; /* (Kg/m^3) */
}

double MIXTURE_specific_heat(cell_t cell, Thread *thread, double temp, double density, double P,
                double yi[])
{
 double cp=0.0;
 int i;

 for (i=0; i<n_specs; ++i)
     if (yi[i]> SMLL)
        cp += yi[i]*RKEOS_specific_heat(temp,mw[i]*density/MIXTURE_mw(yi),i);

 return cp;    /* (J/Kg/K) */
}

double MIXTURE_enthalpy(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
 double h=0.0;
 int i;

 for (i=0; i<n_specs; ++i)
   if (yi[i]> SMLL)
      h += yi[i]*RKEOS_enthalpy(temp, mw[i]*density/MIXTURE_mw(yi), i);

 return h;    /* (J/Kg) */
}

double MIXTURE_enthalpy_prime(cell_t cell, Thread *thread, double temp, double density, double P,
                double yi[], double hi[])
{
 double h=0.0;
 int i;

 for (i=0; i<n_specs; ++i)
{
  hi[i] = hf[i]/mw[i] + RKEOS_enthalpy(temp, mw[i]*density/MIXTURE_mw(yi),
     i);
   if (yi[i]> SMLL)
      h += yi[i]*(hf[i]/mw[i] + RKEOS_enthalpy(temp, mw[i]*density/MIXTURE_mw(yi), i));
}

 return h;    /* (J/Kg) */
}

double MIXTURE_entropy(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
 double s  = 0.0 ;
 double sum = 0.0;
 double xi[n_specs];
 int i;

 for (i=0; i<n_specs; ++i)
  {
     xi[i] = yi[i] / mw[i];
     sum  += xi[i];
  }
 for (i=0; i<n_specs; ++i)
    xi[i] /= sum;

 for (i=0; i<n_specs; ++i)
   if (yi[i]> SMLL)
```

```
       s += yi[i]*RKEOS_entropy(temp,mw[i]*density/MIXTURE_mw(yi), i)-
            UNIVERSAL_GAS_CONSTANT/MIXTURE_mw(yi)* xi[i] * log(xi[i]);

 return s;    /* (J/Kg/K) */
}

double MIXTURE_viscosity(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
 double mu=0.;
 int i;

 for (i=0; i<n_specs; ++i)
   if (yi[i]> SMLL)
        mu += yi[i]*RKEOS_viscosity(temp,i);

 return mu;    /* (Kg/m/s) */
}

double MIXTURE_thermal_conductivity(cell_t cell, Thread *thread, double temp, double density, double P,
                 double yi[])
{
 double kt=0.;
 int i;

 for (i=0; i<n_specs; ++i)
   if (yi[i]> SMLL)
        kt += yi[i]* RKEOS_thermal_conductivity(temp,i);

 return kt;    /* W/m/K */
}

/*-------------------------------------------------------------*/
/* FUNCTION: MIXTURE_speed_of_sound                            */
/*      Returns s.o.s given T and rho                          */
/*-------------------------------------------------------------*/

double MIXTURE_speed_of_sound(cell_t cell, Thread *thread, double temp, double density, double P,
               double yi[])
{
  double dvdp = 0.;
  double cv = 0.;
  double v = 1./density;
  int i;
  double cp = MIXTURE_specific_heat(cell, thread, temp, density, P, yi);

  for (i=0; i<n_specs; ++i)
   if (yi[i]> SMLL)
   {
      dvdp += yi[i]*RKEOS_dvdp(temp, mw[i]*density/MIXTURE_mw(yi),i);
      cv += yi[i]*RKEOS_vol_specific_heat(temp, mw[i]*density/MIXTURE_mw(yi),
          i);
   }

  return sqrt(- cp/cv/dvdp)*v;
}

/*-------------------------------------------------------------*/
/* FUNCTION: MIXTURE_rho_t                                     */
/*-------------------------------------------------------------*/

double MIXTURE_rho_t(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
 double rho_t = 0.;
 int i;

 for (i=0; i<n_specs; ++i)
   if (yi[i]> SMLL)
        rho_t -= yi[i]*density*density*RKEOS_dvdt(temp,
             mw[i]*density/MIXTURE_mw(yi), i);
 return rho_t;
}
```

```
/*----------------------------------------------------------*/
/* FUNCTION: MIXTURE_rho_p                                  */
/*----------------------------------------------------------*/

double MIXTURE_rho_p(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
 double rho_p = 0.;
 int i;

 for (i=0; i<n_specs; ++i)
   if (yi[i]> SMLL)
         rho_p -= yi[i]*density*density*RKEOS_dvdp(temp,
             mw[i]*density/MIXTURE_mw(yi), i);
 return rho_p;
}

/*----------------------------------------------------------*/
/* FUNCTION: MIXTURE_enthalpy_t                             */
/*----------------------------------------------------------*/

double MIXTURE_enthalpy_t(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
  return MIXTURE_specific_heat(cell, thread, temp, density, P, yi);
}

/*----------------------------------------------------------*/
/* FUNCTION: MIXTURE_enthalpy_p                             */
/*----------------------------------------------------------*/

double MIXTURE_enthalpy_p(cell_t cell, Thread *thread, double temp, double density, double P, double yi[])
{
  double v = 1./density;
  double dvdt = 0.0;
  int i;

  for (i=0; i<n_specs; ++i)
   if (yi[i]> SMLL)
       dvdt += yi[i]*RKEOS_dvdt(temp, mw[i]*density/MIXTURE_mw(yi), i);

  return v-temp*dvdt;
}

/****************************************************************/
/* Species Property Definitions                                */
/****************************************************************/

void Mw() /* molecular weight */
{ /* Kg/Kmol */
 mw[0] = 18.01534; /*H2O*/
 mw[1] = 16.04303; /*CH4*/
 mw[2] = 31.99880; /*O2 */
 mw[3] = 44.00995; /*CO2*/
 mw[4] = 28.01340; /*N2 */
}

void Pcrit() /* critical pressure */
{ /* Pa */
 pcrit[0] = 220.64e5; /*H2O*/
 pcrit[1] = 4.48e6; /*CH4*/
 pcrit[2] = 5066250.; /*O2 */
 pcrit[3] = 7.3834e6; /*CO2*/
 pcrit[4] = 33.98e5; /*N2 */
}

void Tcrit() /* critical temperature */
{ /* K */
 tcrit[0] = 647.; /*H2O*/
 tcrit[1] = 191.; /*CH4*/
 tcrit[2] = 155.; /*O2 */
 tcrit[3] = 304.; /*CO2*/
```

```
  tcrit[4] = 126.2; /*N2 */
}

void Vcrit() /* critical specific volume */
{ /* m3/Kg */
 vcrit[0] = 0.003111; /*H2O*/
 vcrit[1] = 0.006187; /*CH4*/
 vcrit[2] = 0.002294; /*O2 */
 vcrit[3] = 0.002136; /*CO2*/
 vcrit[4] = 0.003196; /*N2 */
}

void NRK() /* exponent n of function a(T) in Redlich-Kwong equation of
        state */
{
 int i;
 for (i=0; i<n_specs; ++i)
      nrk[i]= 0.4986 + 1.1735*omega[i] + 0.475*omega[i]*omega[i];
}
void Omega() /* acentric factor */
{
 omega[0] = 0.348; /*H2O*/
 omega[1] = 0.007; /*CH4*/
 omega[2] = 0.021; /*O2 */
 omega[3] = 0.225; /*CO2*/
 omega[4] = 0.040; /*N2 */
}

void Hform() /* formation enthalpy */
{
 /*J/Kmol*/
 hf[0] = -2.418379e+08; /*H2O*/
 hf[1] = -74895176.;   /*CH4*/
 hf[2] = 0.;       /*O2 */
 hf[3] = -3.9353235e+08;/*CO2*/
 hf[4] = 0.;       /*N2 */
}

void Cp_Parameters() /* coefficients of specific heat polynomials */
{ /* J/Kg/K */
 cp[0][0] = 1609.791 ; /*H2O*/
 cp[1][0] = 0.740494;
 cp[2][0] =-9.129835e-06;
 cp[3][0] =-3.813924e-08;
 cp[4][0] =4.80227e-12;

 cp[0][1] = 872.4671 ; /*CH4*/
 cp[1][1] = 5.305473;
 cp[2][1] = -0.002008295;
 cp[3][1] = 3.516646e-07;
 cp[4][1] = -2.33391e-11 ;

 cp[0][2] = 811.1803 ; /*O2 */
 cp[1][2] =0.4108345;
 cp[2][2] =-0.0001750725;
 cp[3][2] = 3.757596e-08;
 cp[4][2] =-2.973548e-12;

 cp[0][3] = 453.577; /*CO2*/
 cp[1][3] = 1.65014;
 cp[2][3] = -1.24814e-3;
 cp[3][3] = 3.78201e-7;
 cp[4][3] = 0.;

 cp[0][4] = 938.8992; /*N2 */
 cp[1][4] = 0.3017911;
 cp[2][4] = -8.109228e-05;
 cp[3][4] = 8.263892e-09 ;
 cp[4][4] = -1.537235e-13;
}
```

```
/****************************************************************/
/*                                                              */
/* User-Defined Function: Redlich-Kwong Equation of State       */
/*             for Real Gas Modeling                            */
/*                                                              */
/* Author: Frank Kelecy                                         */
/*   Date: May 2003                                             */
/*Modified: Rana Faltsi                                         */
/*   Date: December 2006                                        */
/*                                                              */
/*                                                              */
/****************************************************************/
/* OPTIONAL REFERENCE (OFFSET) VALUES FOR ENTHALPY AND ENTROPY */

#define H_REF 0.0
#define S_REF 0.0
/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_pressure of species i                      */
/*      Returns pressure given T and density                  */
/*------------------------------------------------------------*/

double RKEOS_pressure(double temp, double density, int i)
{
  double v = 1./density;
  double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);
  return rgas[i]*temp/(v-bb[i])-afun/(v*(v+b0[i]));
}


/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_spvol of species i                         */
/*      Returns specific volume given T and P                 */
/*------------------------------------------------------------*/

double RKEOS_spvol(double temp, double press, int i)
{
  double a1,a2,a3;
  double vv,vv1,vv2,vv3;
  double qq,qq3,sqq,rr,tt,dd;

  double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);

  a1 = c0[i]-rgas[i]*temp/press;
  a2 = -(bb[i]*b0[i]+rgas[i]*temp*b0[i]/press-afun/press);
  a3 = -afun*bb[i]/press;

  /* Solve cubic equation for specific volume */

  qq = (a1*a1-3.*a2)/9.;
  rr = (2*a1*a1*a1-9.*a1*a2+27.*a3)/54.;
  qq3 = qq*qq*qq;
  dd = qq3-rr*rr;

  /* If dd < 0, then we have one real root */
  /* If dd >= 0, then we have three roots -> choose largest root */

  if (dd < 0.) {
     tt = -SIGN(rr)*(pow(sqrt(-dd)+fabs(rr),0.333333));
     vv = (tt+qq/tt)-a1/3.;
  } else {
     if (rr/sqrt(qq3)<-1) {
        tt = PI;
     } else if (rr/sqrt(qq3)>1) {
        tt = 0;
     } else {
        tt = acos(rr/sqrt(qq3));
     }
   sqq = sqrt(qq);
   vv1 = -2.*sqq*cos(tt/3.)-a1/3.;
   vv2 = -2.*sqq*cos((tt+2.*PI)/3.)-a1/3.;
   vv3 = -2.*sqq*cos((tt+4.*PI)/3.)-a1/3.;
   vv = (vv1 > vv2) ? vv1 : vv2;
```

```
    vv = (vv > vv3) ? vv : vv3;
    /*Message0("Three roots %f %f %f \n",vv1, vv2, vv3);*/
  }

  return vv;
}

/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_dvdp                                     */
/*       Returns dv/dp given T and rho                      */
/*----------------------------------------------------------*/

double RKEOS_dvdp(double temp, double density, int i)
{
  double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);
  double dterm1,dterm2;
  double v    = 1./ density;

  dterm1  = -rgas[i]*temp*pow((v-b0[i]+c0[i]), -2.0);
  dterm2  = afun*(2.0*v+b0[i])*pow(v*(v+b0[i]),-2.0);

  return 1./(dterm1+dterm2);
}

/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_dvdt                                     */
/*       Returns dv/dT given T and rho                      */
/*----------------------------------------------------------*/

double RKEOS_dvdt(double temp, double density, int i)
{
  double dpdT, dterm1, dterm2;
  double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);
  double v    = 1./density;

  dterm1  = rgas[i]/(v-b0[i]+c0[i]);
  dterm2  = nrk[i]*afun/((v*(v+b0[i]))*temp);
  dpdT = dterm1+dterm2;

  return - RKEOS_dvdp(temp, density, i)* dpdT;
}

/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_Cp_ideal_gas                             */
/*       Returns ideal gas specific heat given T            */
/*----------------------------------------------------------*/

double RKEOS_Cp_ideal_gas(double temp, int i)
{
 double cpi=(cp[0][i]+temp*(cp[1][i]+temp*(cp[2][i]+temp*(cp[3][i]
       +temp*cp[4][i]))));
 if (cpi<SMLL)
    cpi = 1.0;
 return cpi;
}

/*----------------------------------------------------------*/
/* FUNCTION: RKEOS_H_ideal_gas                              */
/*       Returns ideal gas specific enthalpy given T        */
/*----------------------------------------------------------*/

double RKEOS_H_ideal_gas(double temp, int i)
{
 double h = temp*(cp[0][i]+temp*(0.5*cp[1][i]+temp*(0.333333*cp[2][i]
       +temp*(0.25*cp[3][i]+temp*0.2*cp[4][i]))));
 if (h<SMLL)
    h = 1.0;
 return h;
}

/*----------------------------------------------------------*/
```

```
/* FUNCTION: RKEOS_vol_specific_heat                         */
/*      Returns constant volume specific heat given T and rho    */
/*----------------------------------------------------------------*/

double RKEOS_vol_specific_heat(double temp, double density, int i)
{
  double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);
  double v = 1./density;
  double Cv0  = RKEOS_Cp_ideal_gas(temp, i) - rgas[i];
  int np1 = (nrk[i]+1.)/b0[i];
  if (Cv0<SMLL)
     Cv0 = 1.;
  return Cv0 + nrk[i]*np1*afun*log(1.0+b0[i]/v)/temp;

}
/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_specific_heat                            */
/*      Returns specific heat given T and rho            */
/*------------------------------------------------------------*/

double RKEOS_specific_heat(double temp, double density, int i)
{
  double delta_Cp,press,v,dvdt,dadt;
  double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);

  press = RKEOS_pressure(temp, density, i);
  v = 1./density;
  dvdt = RKEOS_dvdt(temp, density, i);
  dadt = -nrk[i]*afun/temp;
  delta_Cp = press*dvdt-rgas[i]-dadt*(1.+nrk[i])/b0[i]*log((v+b0[i])/v)
      + afun*(1.+nrk[i])*dvdt/(v*(v+b0[i]));

  return RKEOS_Cp_ideal_gas(temp, i)+delta_Cp; /* (J/Kg-K) */

}


/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_enthalpy                                 */
/*      Returns specific enthalpy given T and rho          */
/*------------------------------------------------------------*/

double RKEOS_enthalpy(double temp, double density, int i)
{
  double delta_h,press, v;

  double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);

  press = RKEOS_pressure(temp, density, i);
  v = 1./density;
  delta_h = press*v-rgas[i]*temp-afun*(1+nrk[i])/b0[i]*log((v+b0[i])/v);

  return H_REF+RKEOS_H_ideal_gas(temp,i)+delta_h; /* (J/Kg) */
}

/*------------------------------------------------------------*/
/* FUNCTION: RKEOS_entropy                                  */
/*      Returns entropy given T and rho                    */
/*------------------------------------------------------------*/

double RKEOS_entropy(double temp, double density, int i)
{
  double delta_s,v,v0,dadt,cp_integral;

  double afun = a0[i]*pow(tcrit[i]/temp,nrk[i]);

  cp_integral = cp[0][i]*log(temp)+temp*(cp[1][i]+temp*(0.5*cp[2][i]
        +temp*(0.333333*cp[3][i]+0.25*cp[4][i]*temp)))
        - cp_int_ref[i];
  if (cp_integral<SMLL)
     cp_integral = 1.0;
  v = 1./density;
```

```
  v0 = rgas[i]*temp/ref_p;
  dadt = -nrk[i]*afun/temp;
  delta_s = rgas[i]*log((v-bb[i])/v0)+dadt/b0[i]*log((v+b0[i])/v);

  return S_REF+cp_integral+delta_s; /* (J/Kg-K) */
}

/*---------------------------------------------------------------*/
/* FUNCTION: RKEOS_viscosity                                     */
/*---------------------------------------------------------------*/

double RKEOS_viscosity(double temp, int i)
{
  double mu,tr,tc,pcatm;

  tr = temp/tcrit[i];
  tc = tcrit[i];
  pcatm = pcrit[i]/101325.;

  mu = 6.3e-7*sqrt(mw[i])*pow(pcatm,0.6666)/pow(tc,0.16666)
    *(pow(tr,1.5)/(tr+0.8));

  return mu;
}


/*---------------------------------------------------------------*/
/* FUNCTION: RKEOS_thermal_conductivity                          */
/*---------------------------------------------------------------*/

double RKEOS_thermal_conductivity(double temp,int i)
{
  double cp, mu;

  cp = RKEOS_Cp_ideal_gas(temp, i);
  mu = RKEOS_viscosity(temp, i);

  return (cp+1.25*rgas[i])*mu;
}

/*****************************************************************/
/* Mixture Functions Structure                                   */
/*****************************************************************/
UDF_EXPORT RGAS_Functions RealGasFunctionList =
{
 MIXTURE_Setup,                    /* initialize            */
 MIXTURE_density,                  /* density               */
 MIXTURE_enthalpy,                 /* sensible enthalpy     */
 MIXTURE_entropy,                  /* entropy               */
 MIXTURE_specific_heat,            /* specific_heat         */
 MIXTURE_mw,                       /* molecular_weight      */
 MIXTURE_speed_of_sound,           /* speed_of_sound        */
 MIXTURE_viscosity,                /* viscosity             */
 MIXTURE_thermal_conductivity,     /* thermal_conductivity  */
 MIXTURE_rho_t,                    /* drho/dT |const p      */
 MIXTURE_rho_p,                    /* drho/dp |const T      */
 MIXTURE_enthalpy_t,               /* dh/dT |const p        */
 MIXTURE_enthalpy_p,               /* dh/dp |const T        */
 MIXTURE_enthalpy_prime            /* enthalpy              */
};
```

Appendix A. C Programming Basics

This chapter contains an overview of C programming basics for UDFs.

A.1. Introduction

This chapter contains some basic information about the C programming language that may be helpful when writing UDFs in Ansys Fluent. It is not intended to be used as a primer on C and assumes that you are an experienced programmer in C. There are many topics and details that are *not* covered in this chapter including, for example, while and do-while control statements, unions, recursion, structures, and reading and writing files.

If you are unfamiliar with C, consult a C language reference guide (such as [6] (p. 713) or [9] (p. 713)) before you begin the process of writing UDFs for your Ansys Fluent model. Note that Ansys Fluent supports version C99 and earlier.

A.2. Commenting Your C Code

It is good programming practice to document your C code with comments that are useful for explaining the purpose of the function. In a single line of code, your comments must begin with the `/*` identifier, followed by text, and end with the `*/` identifier as shown by the following:

```
/* This is how I put a comment in my C program */
```

Comments that span multiple lines are bracketed by the same identifiers:

```
/* This is how I put a comment in my C program
 that spans more
 than one line.  */
```

> **Important:**
>
> Do not include a DEFINE macro name (such as DEFINE_PROFILE) within a comment in your source code. This will cause a compilation error.

A.3. C Data Types in Ansys Fluent

The UDF interpreter in Ansys Fluent supports the following standard C data types:

`int`	integer number
`long`	integer number of increased range
`float`	floating point (real) number
`double`	double-precision floating point (real) number
`char`	single byte of memory, enough to hold a character

Note that in Ansys Fluent, `real` is a typedef that switches between `float` for single-precision arithmetic, and `double` for double-precision arithmetic. Since the interpreter makes this assignment automatically, it is good programming practice to use the `real` typedef when declaring all `float` and `double` data type variables in your UDF.

A.4. Constants

Constants are absolute values that are used in expressions and need to be defined in your C program using `#define`. Simple constants are decimal integers (such as 0, 1, 2). Constants that contain decimal points or the letter `e` are taken as floating point constants. As a convention, constants are typically declared using all capitals. For example, you may set the ID of a zone, or define constants YMIN and YMAX as shown below:

```
#define WALL_ID 5
#define YMIN 0.0
#define YMAX 0.4064
```

A.5. Variables

A variable (or object) is a place in memory where you can store a value. Every variable has a type (for example, `real`), a name, and a value, and may have a storage class identifier (`static` or `extern`). All variables must be declared before they can be used. By declaring a variable ahead of time, the C compiler knows what kind of storage to allocate for the value.

Global variables are variables that are defined outside of any single function and are visible to all function(s) within a UDF source file. Global variables can also be used by other functions outside of the

source file unless they are declared as `static` (see Static Variables (p. 680)). Global variables are typically declared at the beginning of a file, after preprocessor directives as in

```
#include "udf.h"

 real volume; /* real variable named volume is declared globally */

 DEFINE_ADJUST(compute_volume, domain)
 {
 /* code that computes volume of some zone */
 volume = ....
 }
```

Local variables are variables that are used in a single function. They are created when the function is called, and are destroyed when the function returns unless they are declared as `static` (see Static Variables (p. 680)). Local variables are declared within the body of a function (inside the curly braces "{" and "}"). In the example below, `mu_lam` and `temp` are local variables. The value of these variables is not preserved after the function returns.

```
DEFINE_PROPERTY(cell_viscosity, cell, thread)
 {
    real mu_lam;        /* local variable */
    real temp = C_T(cell, thread);   /* local variable */
    if (temp > 288.)
      mu_lam = 5.5e-3;
    else if (temp > 286.)
      mu_lam = 143.2135 - 0.49725 * temp;
    else
      mu_lam = 1.;
    return mu_lam;
 }
```

A.5.1. Declaring Variables

A variable declaration begins with the data type (for example, `int`), followed by the name of one or more variables of the same type that are separated by commas. A variable declaration can also contain an initial value, and always ends with a semicolon (`;`). Variable names must begin with a letter in C. A name can include letters, numbers, and the underscore (`_`) character. Note that the C preprocessor is case-sensitive (recognizes uppercase and lowercase letters as being different). Below are some examples of variable declarations.

```
 int n;       /* declaring variable n as an integer  */
 int i1, i2;    /* declaring variables i1 and i2 as integers */
 float tmax = 0.;   /* tmax is a floating point real number
            that is initialized to 0      */
 real average_temp = 0.0; /* average_temp is a real number initialized
              to 0.0          */
```

A.5.2. External Variables

If you have a global variable that is declared in one source code file, but a function in another source file needs to use it, then it must be defined in the other source file as an external variable. To do this, simply precede the variable declaration by the word `extern` as in

```
 extern real volume;
```

If there are several files referring to that variable then it is convenient to include the `extern` definition in a header (`.h`) file, and include the header file in all of the `.c` files that want to use the external

variable. Only one `.c` file should have the declaration of the variable without the `extern` keyword. Below is an example that demonstrates the use of a header file.

Important:

`extern` can be used only in compiled UDFs.

A.5.2.1. Example

Suppose that there is a global variable named `volume` that is declared in a C source file named `file1.c`

```
#include "udf.h"
 real volume;  /* real variable named volume is declared globally */

 DEFINE_ADJUST(compute_volume, domain)
 {
 /* code that computes volume of some zone */
 volume = ....
 }
```

If multiple source files want to use `volume` in their computations, then `volume` can be declared as an external variable in a header file (for example, `extfile.h`)

```
/* extfile.h
 Header file that contains the external variable declaration for
 volume */

 extern real volume;
```

Now another file named `file2.c` can declare `volume` as an external variable by simply including `extfile.h`.

```
/* file2.c

 #include "udf.h"
 #include "extfile.h" /* header file containing extern declaration
       is included */

 DEFINE_SOURCE(heat_source,c,t,ds,eqn)
 {
 /* code that computes the per unit volume source using the total
   volume computed in the compute_volume function from file1.c  */
 real total_source = ...;
 real source;
 source = total_source/volume;
 return source;
 }
```

A.5.3. Static Variables

The `static` operator has different effects depending on whether it is applied to local or global variables. When a local variable is declared as `static` the variable is prevented from being destroyed when a function returns from a call. In other words, the value of the variable is preserved. When a global variable is declared as `static` the variable is "file global." It can be used by any function within the source file in which it is declared, but is prevented from being used outside the file, even

if is declared as external. Functions can also be declared as `static`. A static function is visible only to the source file in which it is defined.

> **Important:**
>
> `static` variables and functions can be declared *only* in compiled UDF source files.

A.5.3.1. Example - Static Global Variable

```
/* mysource.c /*

 #include "udf.h"

 static real abs_coeff = 1.0; /* static global variable */
  /* used by both functions in this source file but is
   not visible to the outside */

 DEFINE_SOURCE(energy_source, c, t, dS, eqn)
 {
   real source; /* local variable
   int P1 = ....; /* local variable
        value is not preserved when function returns */
   dS[eqn] = -16.* abs_coeff * SIGMA_SBC * pow(C_T(c,t),3.);
   source =-abs_coeff *(4.* SIGMA_SBC * pow(C_T(c,t),4.) - C_UDSI(c,t,P1));
   return source;
 }

 DEFINE_SOURCE(p1_source, c, t, dS, eqn)
 {
   real source;
   int P1 = ...;
   dS[eqn] = -abs_coeff;
   source = abs_coeff *(4.* SIGMA_SBC * pow(C_T(c,t),4.) - C_UDSI(c,t,P1));
   return source;
 }
```

A.6. User-Defined Data Types

C also allows you to create user-defined data types using structures and `typedef`. (For information about structures in C, see [6] (p. 713).) An example of a structured list definition is shown below.

> **Important:**
>
> `typedef` can only be used for compiled UDFs.

A.6.1. Example

```
typedef struct list{int a;
     real b;
     int c;} mylist; /* mylist is type structure list
mylist x,y,z;        x,y,z are type structure list */
```

A.7. Casting

You can convert from one data type to another by casting. A cast is denoted by type, where the type is `int`, `float`, and so on, as shown in the following example:

```
int x = 1;
real y = 3.14159;
int z = x+((int) y);   /* z = 4 */
```

A.8. Functions

Functions perform tasks. Tasks may be useful to other functions defined within the same source code file, or they may be used by a function external to the source file. A function has a name (that you supply) and a list of zero or more arguments that are passed to it. Note that your function name cannot contain a number in the first couple of characters. A function has a body enclosed within curly braces that contains instructions for carrying out the task. A function may return a value of a particular type. C functions pass data by value.

Functions either return a value of a particular data type (for example, `real`), or do not return any value if they are of type `void`. To determine the return data type for the `DEFINE` macro you will use to define your UDF, look at the macro's corresponding `#define` statement in the `udf.h` file or see Appendix B: `DEFINE` Macro Definitions (p. 693) for a listing.

> **Important:**
>
> C functions cannot alter their arguments. They can, however, alter the variables that their arguments point to.

A.9. Arrays

Arrays of variables can be defined using the notation `name[size]`, where `name` is the variable name and `size` is an integer that defines the number of elements in the array. The index of a C array always begins at 0.

Arrays of variables can be of different data types as shown below.

A.9.1. Examples

```
int a[10], b[10][10];
 real radii[5];

 a[0] = 1;      /* a 1-Dimensional array of variable a  */
 radii[4] = 3.14159265;  /* a 1-Dimensional array of variable radii */
 b[10][10] = 4;    /* a 2-Dimensional array of variable b  */
```

A.10. Pointers

A pointer is a variable that contains an address in memory where the value referenced by the pointer is stored. In other words, a pointer is a variable that points to another variable by referring to the other

variable's address. Pointers contain memory addresses, not values. Pointer variables must be declared in C using the `*` notation. Pointers are widely used to reference data stored in structures and to pass data among functions (by passing the addresses of the data).

For example,

```
int *ip;
```

declares a pointer named `ip` that points to an integer variable.

Now suppose you want to assign an address to pointer `ip`. To do this, you can use the `&` notation. For example,

```
ip = &a;
```

assigns the address of variable `a` to pointer `ip`.

You can retrieve the value of variable `a` that pointer `ip` is pointing to by

```
*ip
```

Alternatively, you can set the value of the variable that pointer `ip` points. For example,

```
*ip = 4;
```

assigns a value of `4` to the variable that pointer `ip` is pointing. The use of pointers is demonstrated by the following:

```
int a = 1;
int *ip;
ip = &a;      /* &a returns the address of variable a */
printf("content of address pointed to by ip = %d\n", *ip);
*ip = 4;      /* a = 4 */
printf("now a = %d\n", a);
```

Here, an integer variable `a` is initialized to `1`. Next, `ip` is declared as a pointer to an integer variable. The address of variable `a` is then assigned to pointer `ip`. Next, the integer value of the address pointed to by `ip` is printed using `*ip`. (This value is `1`.) The value of variable `a` is then indirectly set to `4` using `*ip`. The new value of `a` is then printed. Pointers can also point to the beginning of an array, and are strongly connected to arrays in C.

A.10.1. Pointers as Function Arguments

C functions can access and modify their arguments through pointers. In Ansys Fluent, thread and domain pointers are common arguments to UDFs. When you specify these arguments in your UDF, the Ansys Fluent solver automatically passes data that the pointers are referencing to your UDF so that your function can access solver data. (You do not have to declare pointers that are passed as arguments to your UDF from the solver.) For example, one of the arguments passed to a UDF that specifies a custom profile (defined by the `DEFINE_PROFILE` macro) is the pointer to the thread applied to by the boundary condition. The `DEFINE_PROFILE` function accesses the data pointed to by the thread pointer.

A.11. Control Statements

You can control the order in which statements are executed in your C program using control statements like `if`, `if-else`, and `for` loops. Control statements make decisions about what is to be executed next in the program sequence.

A.11.1. `if` Statement

An `if` statement is a type of conditional control statement. The format of an `if` statement is:

```
if (logical-expression)
  {statements}
```

where `logical-expression` is the condition to be tested, and `statements` are the lines of code that are to be executed if the condition is met.

A.11.1.1. Example

```
if (q != 1)
  {a = 0; b = 1;}
```

A.11.2. `if-else` Statement

`if-else` statements are another type of conditional control statement. The format of an `if-else` statement is:

```
if (logical-expression)
  {statements}
else
  {statements}
```

where `logical-expression` is the condition to be tested, and the first set of `statements` are the lines of code that are to be executed if the condition is met. If the condition is not met, then the statements following `else` are executed.

A.11.2.1. Example

```
if (x < 0.)
    y = x/50.;
else
    {
     x = -x;
     y = x/25.;
    }
```

The equivalent FORTRAN code is shown below for comparison.

```
IF (X.LT.0.) THEN
    Y = X/50.
ELSE
    X = -X
    Y = X/25.
ENDIF
```

A.11.3. `for` **Loops**

`for` loops are control statements that are a basic looping construct in C. They are analogous to `do` loops in FORTRAN. The format of a `for` loop is

```
for (begin ; end ; increment)
  {statements}
```

where `begin` is the expression that is executed at the beginning of the loop; `end` is the logical expression that tests for loop termination; and `increment` is the expression that is executed at the end of the loop iteration (usually incrementing a counter).

A.11.3.1. Example

```
/* Print integers 1-10 and their squares */

 int i, j, n <= 10;

 for (i = 1 ; i = n ; i++)
   {
    j = i*i;
    printf("%d %d\n",i,j);
   }
```

The equivalent FORTRAN code is shown below for comparison.

```
     INTEGER I,J
     N = 10
     DO I = 1,10
     J = I*I
     WRITE (*,*) I,J
     ENDDO
```

A.12. Common C Operators

Operators are internal C functions that, when they are applied to values, produce a result. Common types of C operators are arithmetic and logical.

A.12.1. Arithmetic Operators

Some common arithmetic operators are listed below.

```
= assignment
+ addition
- subtraction
* multiplication
/ division
% modulo reduction
++ increment
-- decrement
```

Note that multiplication, division, and modulo reduction (`%`) operations will be performed before addition and subtraction in any expression. When division is performed on two integers, the result is an integer with the remainder discarded. Modulo reduction is the remainder from integer division. The `++` operator is a shorthand notation for the increment operation.

A.12.2. Logical Operators

Some common logical operators are listed below.

```
<   less than
<=  less than or equal to
>   greater than
>=  greater than or equal to
==  equal to
!=  not equal to
```

A.13. C Library Functions

C compilers include a library of standard mathematical and I/O functions that you can use when you write your UDF code. Lists of standard C library functions are presented in the following sections. Definitions for standard C library functions can be found in various header files (for example, `global.h`). These header files are all included in the `udf.h` file.

A.13.1. Trigonometric Functions

The trigonometric functions shown below are computed (with one exception) for the variable x. Both the function and the argument are double-precision `real` variables. The function `acos(x)` is the arccosine of the argument x, $\cos^{-1}(x)$. The function `atan2(x,y)` is the arctangent of x/y, $\tan^{-1}(x/y)$. The function `cosh(x)` is the hyperbolic cosine function, and so on.

`double acos (double x);`	Returns the arccosine of x
`double asin (double x);`	Returns the arcsine of x
`double atan (double x);`	Returns the arctangent of x
`double atan2 (double x, double y);`	Returns the arctangent of x/y
`double cos (double x);`	Returns the cosine of x
`double sin (double x);`	Returns the sine of x
`double tan (double x);`	Returns the tangent of x
`double cosh (double x);`	Returns the hyperbolic cosine of x
`double sinh (double x);`	Returns the hyperbolic sine of x
`double tanh (double x);`	Returns the hyperbolic tangent of x

A.13.2. Miscellaneous Mathematical Functions

The C functions shown on the left below correspond to the mathematical functions shown on the right.

`double sqrt (double x);`	$\sqrt{x}$
`double pow(double x, double y);`	x^y
`double exp (double x);`	e^x

```
double log (double x);
```
$\ln\left(x\right)$

```
double log10 (double x);
```
$\log_{10}(x)$

```
double fabs (double x);
```
$|x|$

```
double ceil (double x);
```
smallest integer not less than x

```
double floor (double x);
```
largest integer not greater than x

A.13.3. Standard I/O Functions

A number of standard input and output (I/O) functions are available in C and in Ansys Fluent. They are listed below. All of the functions work on a specified file except for `printf`, which displays information that is specified in the argument of the function. The format string argument is the same for `printf`, `fprintf`, and `fscanf`. Note that all of these standard C I/O functions are supported by the interpreter, so you can use them in either interpreted or compiled UDFs. For more information about standard I/O functions in C, you should consult a reference guide (for example, [6] (p. 713)).

Common C I/O Functions

```
fopen("filename", "mode");
```
opens a file

```
fclose(fp);
```
closes a file

```
printf("format", ...);
```
formatted print to the console

```
fprintf(fp, "format",...);
```
formatted print to a file

```
fscanf(fp, "format",...);
```
formatted read from a file

> **Important:**
>
> It is not possible to use the `scanf` C function in Ansys Fluent.

> **Important:**
>
> These standard C I/O functions cannot be used when using the DEFINE_DPM_OUTPUT macro due to certain file operations that must be performed by Ansys Fluent. In this case, Ansys Fluent will handle file opening and closing. The macros `par_fprintf` and `par_fprintf_head` are provided for writing output to the file. For additional details, refer to The `par_fprintf_head` and `par_fprintf` Functions (p. 386) and DEFINE_DPM_OUTPUT (p. 227).

A.13.3.1. `fopen`

```
FILE *fopen(const char *filename, const char *mode);
```

The function `fopen` opens a file in the mode that you specify. It takes two arguments: `filename` and `mode`. `filename` is a pointer to the file you want to open. `mode` is the mode in which you want the file opened. The options for `mode` are read `"r"`, write `"w"`, and append `"a"`. Both arguments must be enclosed in quotes. The function returns a pointer to the file that is to be opened.

Before using `fopen`, you will first need to define a local pointer of type `FILE` that is defined in `stdio.h` (for example, `fp`). Then, you can open the file using `fopen`, and assign it to the local pointer as shown below. Recall that `stdio.h` is included in the `udf.h` file, so you do not have to include it in your function.

```
FILE *fp; /* define a local pointer fp of type FILE */
     fp = fopen("data.txt","r");  /* open a file named data.txt in
                read-only mode and assign it to fp */
```

A.13.3.2. `fclose`

```
int fclose(FILE *fp);
```

The function `fclose` closes a file that is pointed to by the local pointer passed as an argument (for example, `fp`).

```
fclose(fp);  /* close the file pointed to by fp */
```

A.13.3.3. `printf`

```
int printf(const char *format,...);
```

The function `printf` is a general-purpose printing function that prints to the console in a format that you specify. The first argument is the format string. It specifies how the remaining arguments are to be displayed in the console. The format string is defined within quotes. The values of the replacement variables specified as arguments following the format string will be substituted sequentially for each instance of `%type` in the format string. The `%` character is used to designate the format type. Some common format characters are: `%d` for integers, `%f` for floating point numbers, and `%e` for floating point numbers in exponential format (with `e` before the exponent). The format string for `printf` is the same as for `fprintf` and `fscanf`.

In the example below, the text `Content of variable a is:` will be displayed in the console, and the value of the replacement variable, `a`, will be substituted in the message for `%d`.

Example:

```
int a = 5;
 printf("Content of variable a is: %d\n", a); /* \n denotes a new line */
```

> **Important:**
>
> It is recommended that you use the Ansys Fluent `Message` utility instead of `printf` for compiled UDFs. See for details on the `Message` macro.

A.13.3.4. `fprintf`

```
int fprintf(FILE *fp, const char *format,...);
```

The function `fprintf` writes to a file that is pointed to by `fp`, in a format that you specify. The second argument is the format string. It specifies how the remaining arguments are to be written to the file. The format string for `fprintf` is the same as for `printf` and `fscanf`.

Example:

```
FILE *fp;
 fprintf(fp,"%12.4e %12.4e %5d\n",x_array[j][0], x_array[j][1], noface);

 int data1 = 64.25;
 int data2 = 97.33;
 fprintf(fp, "%4.2d %4.2d\n", data1, data2);
```

> **Important:**
>
> Note that the standard C function `fprintf` cannot be used when using a
> `DEFINE_DPM_OUTPUT` macro because certain additional file operations must be handled
> by Ansys Fluent. For file writing in a `DEFINE_DPM_OUTPUT` macro the `par_fprintf`
> and `par_fprintf_head` macros must be used. For additional details, refer to The
> par_fprintf_head and par_fprintf Functions (p. 386) and `DEFINE_DPM_OUTPUT`
> (p. 227).

A.13.3.5. `fscanf`

```
int fscanf(FILE *fp, const char *format,...);
```

The function `fscanf` reads from a file that is pointed to by `fp`, in a format that you specify. The
second argument is the format string. It specifies how the data that is to be read is to be interpreted.
The replacement variables that follow the format string are used to store values that are read. The
replacement variables are preceded by the `&` character. Note that the format string for `fscanf` is
the same as for `fprintf` and `printf`.

In the example below, two floating point numbers are read from the file pointed to by `fp`, and are
stored in the variables `f1` and `f2`.

Example:

```
FILE *fp;
 fscanf(fp, "%f %f", &f1, &f2);
```

> **Important:**
>
> You cannot use the `scanf` I/O function in Ansys Fluent. You must use `fscanf` instead.

A.14. Preprocessor Directives

The UDF interpreter supports C preprocessor directives including `#define` and `#include`.

A.14.1. Macro Substitution Directive Using `#define`

When you use the `#define` macro substitution directive, the C preprocessor (for example, `cpp`)
performs a simple substitution and expands the occurrence of each argument in *macro* using the *re-*
placement-text.

```
#define macro replacement-text
```

For example, the macro substitution directive given by

```
#define RAD 1.2345
```

will cause the preprocessor to replace all instances of the variable `RAD` in your UDF with the number `1.2345`. There may be many references to the variable `RAD` in your function, but you only have to define it once in the macro directive; the preprocessor does the work of performing the substitution throughout your code.

In another example

```
#define AREA_RECTANGLE(X,Y) ((X)*(Y))
```

all of the references to `AREA_RECTANGLE(X,Y)` in your UDF are replaced by the product of `(X)` and `(Y)`.

A.14.2. File Inclusion Directive Using `#include`

When you use the `#include` file inclusion directive, the C preprocessor replaces the line `#include` *filename* with the contents of the named file.

```
#include "filename"
```

The file you name must reside in your current folder. The only exception to this rule is the `udf.h` file, which is read automatically by the Ansys Fluent solver.

For example, the file inclusion directive given by

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

will cause the `udf.h` file to be included with your source code.

The Ansys Fluent solver automatically reads the `udf.h` file from the following folder:

path`\ANSYS Inc\v212\fluent\fluent21.2.0\src\udf\udf.h`

where *path* is the folder in which you have installed Ansys Fluent (by default, the path is `C:\Program Files`).

A.15. Comparison with FORTRAN

Many simple C functions are similar to FORTRAN function subroutines as shown in the example below:

A simple C function	An equivalent FORTRAN function
`int myfunction(int x)`	`INTEGER FUNCTION MYFUNCTION(X)`
`{`	
`int x,y,z;`	`INTEGER X,Y,Z`
`y = 11;`	`Y = 11`
`z = x+y;`	`Z = X+Y`
`printf("z = %d",z);`	`WRITE (*,100) Z`

A simple C function

```
return z;
}
```

An equivalent FORTRAN function

```
MYFUNCTION = Z
END
```

Appendix B. `DEFINE` Macro Definitions

This appendix is divided into the following sections:

B.1. General Solver `DEFINE` Macros

The following definitions for general solver `DEFINE` macros (see General Purpose `DEFINE` Macros (p. 20)) are taken from the `udf.h` header file.

```
#define DEFINE_ADJUST(name, domain) void name(Domain *domain)

#define DEFINE_EXECUTE_AT_END(name) void name(void)

#define DEFINE_EXECUTE_AT_EXIT(name) void name(void)

#define DEFINE_EXECUTE_FROM_GUI(name, libname, mode) \
     void name(char *libname, int mode)

#define DEFINE_EXECUTE_ON_LOADING(name, libname) void name(char *libname)

#define DEFINE_INIT(name, domain) void name(Domain *domain)

#define DEFINE_ON_DEMAND(name) void name(void)

#define DEFINE_RW_FILE(name, fp) void name(FILE *fp)

#define DEFINE_RW_HDF_FILE(name, fname) void name(char *fname)
```

B.2. Model-Specific `DEFINE` Macro Definitions

The following definitions for model-specific `DEFINE` macros (see Model-Specific `DEFINE` Macros (p. 46)) are taken from the `udf.h` header file.

```
#define DEFINE_ANISOTROPIC_CONDUCTIVITY(name, c, t, dmatrix) \
     void name(cell_t c, Thread *t, real dmatrix[ND_ND][ND_ND])

#define DEFINE_CHEM_STEP(name, c, t, p, num_p, n_spe, dt, pres, temp, yk) \
     void name(int cell_t c, Thread *t, Particle *p, int num_p, int n_spe, \
       double *dt, double *pres, double *temp, double *yk)

#define DEFINE_CPHI(name,c,t) \
     real name(cell_t c, Thread *t)
```

```
#define DEFINE_DIFFUSIVITY(name, c, t, i) \
    real name(cell_t c, Thread *t, int i)

#define DEFINE_DOM_DIFFUSE_REFLECTIVITY(name ,t, nb, n_a, n_b, diff_ ref_a, \
    diff_tran_a, diff_ref_b, diff_tran_b) \
    void name(Thread *t, int nb, real n_a, real n_b, real *diff_ref_a, \
      real *diff_tran_a, real *diff_ref_b, real *diff_tran_b)

#define DEFINE_DOM_SPECULAR_REFLECTIVITY(name, f, t, nb, n_a, n_b, \
    ray_direction, e_n, total_internal_reflection, \
    specular_reflectivity, specular_transmissivity) \
    void name(face_t f, Thread *t, int nb, real n_a, real n_b , \
      real ray_direction[], real e_n[], \
      int *total_internal_reflection, real *specular_reflectivity,\
      real *specular_transmissivity)

#define DEFINE_DOM_SOURCE(name, c, t, ni, nb, emission, in_scattering, \
    abs_coeff, scat_coeff) \
    void name(cell_t c, Thread* t, int ni, int nb, real *emission, \
      real *in_scattering, real *abs_coeff, real *scat_coeff)

#define DEFINE_EMISSIVITY_WEIGHTING_FACTOR(name, c, t, T, nb, \
    emissivity_weighting_factor) \
    void name(cell_t c, Thread* t, real T, int nb, \
      real *emissivity_weighting_factor)

#define DEFINE_GRAY_BAND_ABS_COEFF(name, c, t, nb) \
     real name(cell_t c, Thread *t, int nb)

#define DEFINE_HEAT_FLUX(name, f, t, c0, t0, cid, cir) \
    void name(face_t f, Thread *t, cell_t c0, Thread *t0, \
      real cid[], real cir[])

#define DEFINE_NET_REACTION_RATE(name, c, t, particle, pressure, \
    temp, yi, rr, jac) \
    void name(cell_t c, Thread *t, Particle *particle, \
    double *pressure, double *temp, double *yi, double *rr, \
    double *jac)

#define DEFINE_NOX_RATE(name, c, t, Pollut, Pollut_Par, NOx) \
    void name(cell_t c, Thread *t, Pollut_Cell *Pollut, \
    Pollut_Parameter *Poll_Par, NOx_Parameter *NOx)

#define DEFINE_PRANDTL_K(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PRANDTL_D(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PRANDTL_O(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PRANDTL_T(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PRANDTL_T_WALL(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PROFILE(name, t, i) void name(Thread *t, int i)

#define DEFINE_PROPERTY(name, c, t) real name(cell_t c, Thread *t)

#define DEFINE_PR_RATE(name, c, t, r, mw, ci, tp, sf, dif_index, \
 cat_index, rr) \
    void name(cell_t c, Thread *t, Reaction *r, real *mw, real *ci, \
      Tracked_Particle *tp, real *sf , int dif_index, \
      int cat_index, real *rr)

#define DEFINE_SBES_BF(name, c, t) \
   real name(cell_t c, Thread *t)

#define DEFINE_SCAT_PHASE_FUNC(name, c, f) \
   real name(real c, real *f)

#define DEFINE_SOLAR_INTENSITY(name, sun_x, sun_y, sun_z, S_hour, S_minute) \
```

```
     real name(real sun_x, real sun_y, real sun_z, int S_hour,int S_minute)

  #define DEFINE_SOURCE(name, c, t, dS, i) \
       real name(cell_t c, Thread *t, real dS[], int i)

  #define DEFINE_SOX_RATE(name, c, t, Pollut, Pollut_Par, SOx) \
       void name(cell_t c, Thread *t, Pollut_Cell *Pollut, \
         Pollut_Parameter *Poll_Par, SOx_Parameter *SOx)

  #define DEFINE_SR_RATE(name, f, t, r, mw, yi, rr) \
       void name(face_t f, Thread *t, \
          Reaction *r, real *mw, real *yi, real *rr)

  #define DEFINE_TURB_PREMIX_SOURCE(name, c, t, \
       turbulent_flame_speed, source) \
       void name(cell_t c, Thread *t, real *turbulent_flame_speed, \
         real *source)

  #define DEFINE_TURBULENT_VISCOSITY(name, c, t) \
       real name(cell_t c, Thread *t)

  #define DEFINE_VR_RATE(name, c, t, r, mw, yi, rr, rr_t) \
       void name(cell_t c, Thread *t, \
       Reaction *r, real *mw, real *yi, real *rr, real *rr_t)

  #define DEFINE_WALL_FUNCTIONS(name, f, t, c0, t0, wf_ret, yPlus, Emod) \
       real name(face_t f, Thread *t, cell_t c0, Thread *t0, int wf_ret \
         real yPlus, real Emod)

  #define DEFINE_WALL_NODAL_DISP(name, f, t, v, m) \
       real name(face_t f, Thread *t, Node *v, int m)

  #define DEFINE_WALL_NODAL_FORCE(name, f, t, v, m) \
       real name(face_t f, Thread *t, Node *v, int m)

#define DEFINE_SOURCE_FE(name, c, t) \
       real name(cell_t c, Thread *t)

  #define DEFINE_WSGGM_ABS_COEFF(name, c, t, xi, p_t, s, soot_conc, Tcell, \
       nb, ab_wsggm, ab_soot)  \
       void name(cell_t c, Thread *t, real xi[], real p_t, real s, real soot_conc, \
         real Tcell, int nb, real *ab_wsggm, real *ab_soot)
```

B.3. Multiphase **DEFINE** Macros

The following definitions for multiphase DEFINE macros (see) are taken from the udf.h header file.

```
#define DEFINE_CAVITATION_RATE(name, c, t, p, rhoV, rhoL, vofV, p_v, \
 cigma, f_gas, m_dot) \
 void name(cell_t c, Thread *t, real *p, real *rhoV, real *rhoL, \
   real *vofV, real *p_v, real *cigma, real *f_gas, real *m_dot)

#define DEFINE_EXCHANGE_PROPERTY(name, c, mixture_thread, \
 phase_index_1, phase_index_2) \
 real name(cell_t c, Thread *mixture_thread, \
   int phase_index_1, int phase_index_2)

#define DEFINE_HET_RXN_RATE(name, c, t, hr, mw, yi, rr, rr_t) \
 void name(cell_t c, Thread *t, \
 Hetero_Reaction *hr, real mw[MAX_PHASES][MAX_SPE_EQNS], \
   real yi[MAX_PHASES][MAX_SPE_EQNS], real *rr, real *rr_t)

#define DEFINE_MASS_TRANSFER(name, c, mixture_thread, from_phase_index, \
 from_species_index, to_phase_index, to_species_index) \
 real name(cell_t c, Thread *mixture_thread, int from_phase_index, \
 int from_species_index, int to_phase_index, int to_species_index)
```

```
#define DEFINE_VECTOR_EXCHANGE_PROPERTY(name, c, mixture_thread, \
 phase_index_1, phase_index_2, vector_result) \
 void name(cell_t c, Thread *mixture_thread, \
    int phase_index_1, \
    int phase_index_2, real *vector_result)
```

B.4. Dynamic Mesh Model `DEFINE` Macros

The following definitions for dynamic mesh model `DEFINE` macros (see Dynamic Mesh `DEFINE` Macros (p. 263)) are taken from the `udf.h` header file.

```
#define DEFINE_CG_MOTION(name, dt, vel, omega, time, dtime) \
 void name(Dynamic_Thread *dt, real vel[], real omega[], real time,\
    real dtime)

#define DEFINE_DYNAMIC_ZONE_PROPERTY(name, dt, swirl_center) \
 void name(Dynamic_Thread *dt, real *swirl_center)

#define DEFINE_DYNAMIC_ZONE_PROPERTY(name, dt, height) \
 void name(Dynamic_Thread *dt, real *height)

#define DEFINE_GEOM(name, d, dt, position) \
 void name(Domain *d, Dynamic_Thread *dt, real *position)

#define DEFINE_GRID_MOTION(name, d, dt, time, dtime) \
 void name(Domain *d, Dynamic_Thread *dt, real time, real dtime)

#define DEFINE_SDOF_PROPERTIES(name, properties, dt, time, dtime) \
 void name(real *properties, Dynamic_Thread *dt, real time, real dtime)

#define DEFINE_CONTACT(name, dt, contacts) \
 void name(Dynamic_Thread *dt, Objp *contacts)
```

B.5. Discrete Phase Model `DEFINE` Macros

The following definitions for DPM `DEFINE` macros (see Discrete Phase Model (DPM) `DEFINE` Macros (p. 201)) are taken from the `dpm.h` header file. Note that `dpm.h` is included in the `udf.h` header file.

```
#define DEFINE_DPM_BC(name, tp, t, f, normal, dim) \
 int name(Tracked_Particle *tp, Thread *t, face_t f, \
  real normal[], int dim)

#define DEFINE_DPM_BODY_FORCE(name, tp, i) \
 real name(Tracked_Particle *tp, int i)

#define DEFINE_DPM_DRAG(name, Re, tp) \
 real name(real Re, Tracked_Particle *tp)

#define DEFINE_DPM_EROSION(name, tp, t, f, normal, alpha, Vmag, mdot) \
 void name(Tracked_Particle *tp, Thread *t, face_t f, real normal[], \
 real alpha, real Vmag, real mdot)

#define DEFINE_DPM_HEAT_MASS(name, tp, Cp, hgas, hvap, cvap_surf, dydt, dzdt) \
 void name(Tracked_Particle *tp, real Cp, \
    real *hgas, real *hvap, real *cvap_surf, real *dydt, dpms_t *dzdt)

#define DEFINE_DPM_INJECTION_INIT(name, I) void name(Injection *I)

#define DEFINE_DPM_LAW(name, tp, ci) \
 void name(Tracked_Particle *tp, int ci)
```

```
#define DEFINE_DPM_OUTPUT(name, header, fp, tp, t, plane) \
 void name(int header, FILE *fp, Tracked_Particle *tp, \
   Thread *t, Plane *plane)

#define DEFINE_DPM_PROPERTY(name, c, t, tp, T) \
 real name(cell_t c, Thread *t, Tracked_Particle *tp, real T)

#define DEFINE_DPM_SCALAR_UPDATE(name, c, t, initialize, tp) \
 void name(cell_t c, Thread *t, int initialize, Tracked_Particle *tp)

#define DEFINE_DPM_SOURCE(name, c, t, S, strength, tp) \
void name(cell_t c, Thread *t, dpms_t *S, real strength,\
   Tracked_Particle *tp)

#define DEFINE_DPM_SPRAY_COLLIDE(name, tp, p) \
 void name(Tracked_Particle *tp, Particle *p)

#define DEFINE_DPM_SWITCH(name, tp, ci) \
 void name(Tracked_Particle *tp, int ci)

#define DEFINE_DPM_TIMESTEP(name, tp, ts) \
 real name(Tracked_Particle *tp,real ts)

#define DEFINE_DPM_VP_EQUILIB(name, tp, T, cvap_surf, Z) \
 void name(Tracked_Particle *tp, real T, real *cvap_surf, real *Z)

#define DEFINE_IMPINGEMENT(name, tp, rel_dot_n, f, t, y_s, E_imp) \
 int name(Tracked_Particle *tp, real rel_dot_n, face_t f, Thread *t, real *y_s, real *E_imp)

#define DEFINE_FILM_REGIME(name, regime, tp, pn, f, t, f_normal, update) \
 void name(Wall_Film_Regime regime, Tracked_Particle *tp, Particle *pn, face_t f, Thread *t, \
    real f_normal[], cxboolean update)

#define DEFINE_SPLASHING_DISTRIBUTION(name, tp, rel_dot_n, f_normal, n_samp, s) \
 void name(Tracked_Particle *tp, real rel_dot_n, real f_normal[], int n_samp, splashing_distribution_t *s)
```

B.6. User-Defined Scalar (UDS) `DEFINE` Macros

The following definitions for UDS `DEFINE` macros (see User-Defined Scalar (UDS) Transport Equation `DEFINE` Macros (p. 295)) are taken from the `udf.h` header file.

```
#define DEFINE_ANISOTROPIC_DIFFUSIVITY(name, c, t, ns, dmatrix) \
void name(cell_t c, Thread *t, int ns, real dmatrix[ND_ND][ND_ND])

#define DEFINE_UDS_FLUX(name, f, t, i) real name(face_t f, Thread *t, int i)

#define DEFINE_UDS_UNSTEADY(name, c, t, i, apu, su) \
 void name(cell_t c, Thread *t, int i, real *apu, real *su)
```

B.7. Battery Model `DEFINE` Macros

The following definitions for the battery `DEFINE` macros (see Battery Model `DEFINE` Macros (p. 283)) are taken from the `udf.h` header file.

```
#define DEFINE_BATTERY_PARAMETER_NTGK(name, c, t, soc, U, Y) \
   void name(cell_t c, Thread *t, real soc, real *U, real *Y)
#define DEFINE_BATTERY_PARAMETER_ECM(name, c, t, soc, mode, Voc, Rs, R1, C1, R2, C2, R3, C3) \
   void name(cell_t c, Thread *t, real soc, int mode, real *Voc, real *Rs, real *R1, real *C1, real *R2, \
            real *C2, real *R3, real *C3)
#define DEFINE_BATTERY_ENTROPIC_HEAT(name, soc) \
 real name(real soc)
#define DEFINE_BATTERY_ECHEM_MODEL(name, zero_start, mode, T, V_in, I_in, dt, I_out, Qe, V_out) \
 void name(int zero_start, int mode, real T, real V_in, real I_in, real dt, real *I_out, real  *Qe, \
```

```
            real *V_out)
#define DEFINE_BATTERY_NEWMAN_BV_RATE(name, Ce, Cs, Cs_max, T, eta, i_0, alpha_a, alpha_c, mode) \
  real name(real Ce, real Cs, real Cs_max, real T, real eta, real i_0, real alpha_a, real alpha_c, int mode) \
#define DEFINE_BATTERY_NEWMAN_POSTPROCESSING(name, c, t, T, Vp, Vn)  \
  void name(cell_t c, Thread *t, real T, real Vp, real Vn)
#define DEFINE_BATTERY_PROPERTY(name, x, T)  \
  real name(real x, real T)
#define DEFINE_BATTERY_CLUSTER(name, c, t)  \
  real name(cell_t c, Thread *t)
```

Appendix C. Quick Reference Guide for Multiphase `DEFINE` Macros

This appendix is a reference guide that contains a list of general purpose `DEFINE` macros (Model-Specific `DEFINE` Macros (p. 46)) and multiphase-specific `DEFINE` macros (Multiphase `DEFINE` Macros (p. 179)) that can be used to define multiphase model UDFs.

See Special Considerations for Multiphase UDFs (p. 16) for information on special considerations for multiphase UDFs.

This appendix is divided into the following sections:

C.1. VOF Model

Table 1: `DEFINE` Macro Usage for the VOF Model (p. 699) – Table 3: `DEFINE` Macro Usage for the VOF Model (p. 701) list the variables that can be customized using UDFs for the VOF multiphase model, the `DEFINE` macros that are used to define the UDFs, and the phase that the UDF must be hooked to for the given variable.

Table 1: `DEFINE` Macro Usage for the VOF Model

Variable	Macro	Phase Specified On
Boundary Conditions Inlet/Outlet		
volume fraction	DEFINE_PROFILE	secondary phase(s)
velocity magnitude	DEFINE_PROFILE	mixture
pressure	DEFINE_PROFILE	mixture
temperature	DEFINE_PROFILE	mixture
mass flux	DEFINE_PROFILE	primary and secondary phase(s)
species mass fractions	DEFINE_PROFILE	phase-dependent
internal emissivity	DEFINE_PROFILE	mixture
user-defined scalar boundary value	DEFINE_PROFILE	mixture

Variable	Macro	Phase Specified On
discrete phase boundary condition	DEFINE_PROFILE	mixture

Table 2: DEFINE Macro Usage for the VOF Model

Variable	Macro	Phase Specified On
Fluid		
mass source	DEFINE_SOURCE	primary and secondary phase(s)
momentum source	DEFINE_SOURCE	mixture
energy source	DEFINE_SOURCE	mixture
turbulence kinetic energy source	DEFINE_SOURCE	mixture
turbulence dissipation rate source	DEFINE_SOURCE	mixture
user-defined scalar source	DEFINE_SOURCE	mixture
species source	DEFINE_SOURCE	phase-dependent
velocity	DEFINE_PROFILE	mixture
temperature	DEFINE_PROFILE	mixture
user-defined scalar	DEFINE_PROFILE	mixture
turbulence kinetic energy	DEFINE_PROFILE	mixture
turbulence dissipation rate	DEFINE_PROFILE	mixture
species mass fraction	DEFINE_PROFILE	phase-dependent
porosity	DEFINE_PROFILE	mixture
Boundary Conditions Wall		
species boundary condition	DEFINE_PROFILE	phase-dependent
internal emissivity	DEFINE_PROFILE	mixture
irradiation	DEFINE_PROFILE	mixture
roughness height	DEFINE_PROFILE	mixture
roughness constant	DEFINE_PROFILE	mixture
shear stress components	DEFINE_PROFILE	mixture
swirl components	DEFINE_PROFILE	mixture
moving velocity components	DEFINE_PROFILE	mixture
heat flux	DEFINE_PROFILE	mixture
heat generation rate	DEFINE_PROFILE	mixture
heat transfer coefficient	DEFINE_PROFILE	mixture
external emissivity	DEFINE_PROFILE	mixture
external radiation temperature	DEFINE_PROFILE	mixture
free stream temperature	DEFINE_PROFILE	mixture

Variable	Macro	Phase Specified On
user scalar boundary value	`DEFINE_PROFILE`	mixture
discrete phase boundary value	`DEFINE_DPM_BC`	mixture

Table 3: `DEFINE` Macro Usage for the VOF Model

Variable	Macro	Phase Specified On
Other		
surface tension coefficient	`DEFINE_PROPERTY`	phase interaction
mass transfer coefficient	`DEFINE_MASS_TRANSFER`	phase interaction
heterogeneous reaction rate	`DEFINE_HET_RXN_RATE`	phase interaction

C.2. Mixture Model

Table 4:`DEFINE` Macro Usage for the Mixture Model (p. 701) – Table 6:`DEFINE` Macro Usage for the Mixture Model (p. 703) list the variables that can be customized using UDFs for the Mixture multiphase model, the `DEFINE` macros that are used to define the UDFs, and the phase that the UDF must be hooked to for the given variable.

Table 4: `DEFINE` Macro Usage for the Mixture Model

Variable	Macro	Phase Specified On
Boundary Conditions Inlet/Outlet		
volume fraction	`DEFINE_PROFILE`	secondary phase(s)
mass flux	`DEFINE_PROFILE`	primary and secondary phase(s)
velocity magnitude	`DEFINE_PROFILE`	primary and secondary phase(s)
pressure	`DEFINE_PROFILE`	mixture
temperature	`DEFINE_PROFILE`	mixture
species mass fractions	`DEFINE_PROFILE`	phase-dependent
user-defined scalar boundary value	`DEFINE_PROFILE`	mixture
discrete phase boundary condition	`DEFINE_PROFILE`	mixture

Table 5: `DEFINE` Macro Usage for the Mixture Model

Variable	Macro	Phase Specified On
Fluid		
mass source	`DEFINE_SOURCE`	primary and secondary phase(s)
momentum source	`DEFINE_SOURCE`	mixture
energy source	`DEFINE_SOURCE`	mixture

Variable	Macro	Phase Specified On
turbulence kinetic energy source	DEFINE_SOURCE	mixture
turbulence dissipation rate source	DEFINE_SOURCE	mixture
granular temperature source	DEFINE_SOURCE	secondary phase(s)
user scalar source	DEFINE_SOURCE	mixture
species source	DEFINE_SOURCE	phase-dependent
species mass fractions	DEFINE_PROFILE	phase-dependent
velocity	DEFINE_PROFILE	mixture
temperature	DEFINE_PROFILE	mixture
turbulence kinetic energy	DEFINE_PROFILE	mixture
turbulence dissipation rate	DEFINE_PROFILE	mixture
porosity	DEFINE_PROFILE	mixture
granular temperature	DEFINE_PROFILE	secondary phase(s)
viscous resistance	DEFINE_PROFILE	primary and secondary phase(s)
inertial resistance	DEFINE_PROFILE	primary and secondary phase(s)
Wall		
roughness height	DEFINE_PROFILE	mixture
roughness constant	DEFINE_PROFILE	mixture
internal emissivity	DEFINE_PROFILE	mixture
shear stress components	DEFINE_PROFILE	mixture
moving velocity components	DEFINE_PROFILE	mixture
heat flux	DEFINE_PROFILE	mixture
heat generation rate	DEFINE_PROFILE	mixture
heat transfer coefficient	DEFINE_PROFILE	mixture
external emissivity	DEFINE_PROFILE	mixture
external radiation temperature	DEFINE_PROFILE	mixture
free stream temperature	DEFINE_PROFILE	mixture
granular flux	DEFINE_PROFILE	secondary phase(s)
granular temperature	DEFINE_PROFILE	secondary phase(s)
user scalar boundary value	DEFINE_PROFILE	mixture
discrete phase boundary value	DEFINE_DPM_BC	mixture

Variable	Macro	Phase Specified On
species boundary condition	DEFINE_PROFILE	phase-dependent

Table 6: DEFINE Macro Usage for the Mixture Model

Variable	Macro	Phase Specified On
Material Properties		
cavitation surface tension coefficient	DEFINE_PROPERTY	phase interaction
cavitation vaporization pressure	DEFINE_PROPERTY	phase interaction
particle or droplet diameter	DEFINE_PROPERTY	secondary phase(s)
granular diameter	DEFINE_PROPERTY	secondary phase(s)
granular solids pressure	DEFINE_PROPERTY	secondary phase(s)
granular radial distribution	DEFINE_PROPERTY	secondary phase(s)
granular elasticity modulus	DEFINE_PROPERTY	secondary phase(s)
granular viscosity	DEFINE_PROPERTY	secondary phase(s)
granular temperature	DEFINE_PROPERTY	secondary phase(s)
Other		
slip velocity	DEFINE_VECTOR_ EX-CHANGE_PROPERTY	phase interaction
drag coefficient	DEFINE_EXCHANGE	phase interaction
mass transfer coefficient	DEFINE_MASS_TRANSFER	phase interaction
heterogeneous reaction rate	DEFINE_HET_RXN_RATE	phase interaction

C.3. Eulerian Model - Laminar Flow

list the variables that can be customized using UDFs for the laminar flow Eulerian multiphase model, the DEFINE macros that are used to define the UDFs, and the phase that the UDF must be hooked to for the given variable.

Table 7: DEFINE Macro Usage for the Eulerian Model - Laminar Flow

Variable	Macro	Phase Specified On
Boundary Conditions Inlet/Outlet		
volume fraction	DEFINE_PROFILE	secondary phase(s)
species mass fractions	DEFINE_PROFILE	phase-dependent
mass flux	DEFINE_PROFILE	primary and secondary phase(s)
flow direction components	DEFINE_PROFILE	primary and secondary phase(s)

Variable	Macro	Phase Specified On
velocity magnitude	DEFINE_PROFILE	primary and secondary phase(s)
temperature	DEFINE_PROFILE	primary and secondary phase(s)
pressure	DEFINE_PROFILE	mixture
user-defined scalar boundary value	DEFINE_PROFILE	mixture
discrete phase boundary value	DEFINE_DPM_BC	mixture
Fluid		
mass source	DEFINE_SOURCE	primary and secondary phase(s)
momentum source	DEFINE_SOURCE	primary and secondary phase(s)
energy source	DEFINE_SOURCE	primary and secondary phase(s)
species source	DEFINE_SOURCE	phase-dependent
granular temperature source	DEFINE_SOURCE	secondary phase(s)
user-defined scalar source	DEFINE_SOURCE	mixture
velocity	DEFINE_PROFILE	primary and secondary phase(s)
temperature	DEFINE_PROFILE	primary and secondary phase(s)

Table 8: `DEFINE` Macro Usage for the Eulerian Model - Laminar Flow

Variable	Macro	Phase Specified On
Boundary Conditions Fluid		
species mass fraction	DEFINE_PROFILE	phase-dependent
granular temperature	DEFINE_PROFILE	secondary phase(s)
porosity	DEFINE_PROFILE	mixture
user-defined scalar	DEFINE_PROFILE	mixture
viscous resistance	DEFINE_PROFILE	primary and secondary phase(s)
inertial resistance	DEFINE_PROFILE	primary and secondary phase(s)
Wall		
species boundary condition	DEFINE_PROFILE	phase-dependent
shear stress components	DEFINE_PROFILE	primary and secondary phase(s)
moving velocity components	DEFINE_PROFILE	secondary phase(s)
temperature	DEFINE_PROFILE	mixture

Variable	Macro	Phase Specified On
heat flux	DEFINE_PROFILE	mixture
heat generation rate	DEFINE_PROFILE	mixture
heat transfer coefficient	DEFINE_PROFILE	mixture
external emissivity	DEFINE_PROFILE	mixture
external radiation temperature	DEFINE_PROFILE	mixture
free stream temperature	DEFINE_PROFILE	mixture
user-defined scalar boundary value	DEFINE_PROFILE	mixture
discrete phase boundary value	DEFINE_DPM_BC	mixture
Material Properties		
granular diameter	DEFINE_PROPERTY	secondary phase(s)
granular solids pressure	DEFINE_PROPERTY	secondary phase(s)
granular radial distribution	DEFINE_PROPERTY	secondary phase(s)
granular elasticity modulus	DEFINE_PROPERTY	secondary phase(s)
granular viscosity	DEFINE_PROPERTY	secondary phase(s)
granular temperature	DEFINE_PROPERTY	secondary phase(s)

Table 9: `DEFINE` Macro Usage for the Eulerian Model - Laminar Flow

Variable	Macro	Phase Specified On
Other		
drag coefficient	DEFINE_EXCHANGE	phase interaction
lift coefficient	DEFINE_EXCHANGE	phase interaction
heat transfer coefficient	DEFINE_PROPERTY	phase interaction
mass transfer coefficient	DEFINE_MASS_TRANSFER	phase interaction
heterogeneous reaction rate	DEFINE_HET_RXN_RATE	phase interaction

C.4. Eulerian Model - Mixture Turbulence Flow

Table 10:`DEFINE` Macro Usage for the Eulerian Model - Mixture Turbulence Flow (p. 705) –
Table 12:`DEFINE` Macro Usage for the Eulerian Model - Mixture Turbulence Flow (p. 707) list the variables
that can be customized using UDFs for the mixed turbulence flow Eulerian multiphase model, the
`DEFINE` macros that are used to define the UDFs, and the phase that the UDF must be hooked to for
the given variable.

Table 10: `DEFINE` Macro Usage for the Eulerian Model - Mixture Turbulence Flow

Variable	Macro	Phase Specified On
Boundary Conditions Inlet/Outlet		
volume fraction	DEFINE_PROFILE	secondary phase(s)

Variable	Macro	Phase Specified On
species mass fractions	`DEFINE_PROFILE`	phase-dependent
mass flux	`DEFINE_PROFILE`	primary and secondary phase(s)
velocity magnitude	`DEFINE_PROFILE`	primary and secondary phase(s)
temperature	`DEFINE_PROFILE`	primary and secondary phase(s)
pressure	`DEFINE_PROFILE`	mixture

Table 11: `DEFINE` Macro Usage for the Eulerian Model - Mixture Turbulence Flow

Variable	Macro	Phase Specified On
Boundary Conditions Inlet/Outlet - continued		
user-defined scalar boundary value	`DEFINE_PROFILE`	mixture
discrete phase boundary condition	`DEFINE_PROFILE`	mixture
Fluid mass source	`DEFINE_SOURCE`	primary and secondary phase(s)
momentum source	`DEFINE_SOURCE`	primary and secondary phase(s)
energy source	`DEFINE_SOURCE`	primary and secondary phase(s)
turbulence dissipation rate source	`DEFINE_SOURCE`	mixture
turbulence kinetic energy source	`DEFINE_SOURCE`	mixture
user-defined scalar source	`DEFINE_SOURCE`	mixture
user-defined scalar	`DEFINE_PROFILE`	mixture
turbulence kinetic energy	`DEFINE_PROFILE`	mixture
turbulence dissipation rate	`DEFINE_PROFILE`	mixture
velocity	`DEFINE_PROFILE`	primary and secondary phase(s)
temperature	`DEFINE_PROFILE`	primary and secondary phase(s)
porosity	`DEFINE_PROFILE`	mixture
user-defined scalar	`DEFINE_PROFILE`	mixture
viscous resistance	`DEFINE_PROFILE`	primary and secondary phase(s)
inertial resistance	`DEFINE_PROFILE`	primary and secondary phase(s)
Wall		

Variable	Macro	Phase Specified On
species boundary condition	DEFINE_PROFILE	phase-dependent
shear stress components	DEFINE_PROFILE	primary and secondary phase(s)
moving velocity components	DEFINE_PROFILE	mixture
temperature	DEFINE_PROFILE	mixture
heat flux	DEFINE_PROFILE	mixture
heat generation rate	DEFINE_PROFILE	mixture
heat transfer coefficient	DEFINE_PROFILE	mixture
external emissivity	DEFINE_PROFILE	mixture
external radiation temperature	DEFINE_PROFILE	mixture
free stream temperature	DEFINE_PROFILE	mixture

Table 12: DEFINE Macro Usage for the Eulerian Model - Mixture Turbulence Flow

Variable	Macro	Phase Specified On
Wall - continued		
granular flux	DEFINE_PROFILE	secondary phase(s)
granular temperature	DEFINE_PROFILE	secondary phase(s)
discrete phase boundary condition	DEFINE_DPM_BC	secondary phase(s)
user-defined scalar boundary value	DEFINE_PROFILE	secondary phase(s)
Material Properties		
granular diameter	DEFINE_PROPERTY	secondary phase(s)
granular viscosity	DEFINE_PROPERTY	secondary phase(s)
granular bulk viscosity	DEFINE_PROPERTY	secondary phase(s)
granular frictional viscosity	DEFINE_PROPERTY	secondary phase(s)
granular conductivity	DEFINE_PROPERTY	secondary phase(s)
granular solids pressure	DEFINE_PROPERTY	secondary phase(s)
granular radial distribution	DEFINE_PROPERTY	secondary phase(s)
granular elasticity modulus	DEFINE_PROPERTY	secondary phase(s)
turbulent viscosity	DEFINE_TURBULENT_VIS-COSITY	mixture, primary, and secondary phase(s)
Other		
drag coefficient	DEFINE_EXCHANGE	phase interaction
lift coefficient	DEFINE_EXCHANGE	phase interaction
heat transfer coefficient	DEFINE_PROPERTY	phase interaction
mass transfer coefficient	DEFINE_MASS_TRANSFER	phase interaction
heterogeneous reaction rate	DEFINE_HET_RXN_RATE	phase interaction

C.5. Eulerian Model - Dispersed Turbulence Flow

Table 13:`DEFINE` Macro Usage for the Eulerian Model - Dispersed Turbulence Flow (p. 708) –
Table 15:`DEFINE` Macro Usage for the Eulerian Model - Dispersed Turbulence Flow (p. 710) list the variables that can be customized using UDFs for the dispersed turbulence flow Eulerian multiphase model, the `DEFINE` macros that are used to define the UDFs, and the phase that the UDF must be hooked to for the given variable.

Table 13: `DEFINE` Macro Usage for the Eulerian Model - Dispersed Turbulence Flow

Variable	Macro	Phase Specified On
Boundary Conditions Inlet/Outlet		
volume fraction	DEFINE_PROFILE	secondary phase(s)
species mass fractions	DEFINE_PROFILE	phase-dependent
mass flux	DEFINE_PROFILE	primary and secondary phase(s)
velocity magnitude	DEFINE_PROFILE	primary and secondary phase(s)
temperature	DEFINE_PROFILE	primary and secondary phase(s)
pressure	DEFINE_PROFILE	mixture
user-defined scalar boundary value	DEFINE_PROFILE	mixture
discrete phase boundary condition	DEFINE_PROFILE	mixture
Fluid		
mass source	DEFINE_SOURCE	primary and secondary phase(s)
momentum source	DEFINE_SOURCE	primary and secondary phase(s)
energy source	DEFINE_SOURCE	primary and secondary phase(s)
turbulence dissipation rate source	DEFINE_SOURCE	primary and secondary phase(s)
turbulence kinetic energy source	DEFINE_SOURCE	primary and secondary phase(s)
species source	DEFINE_SOURCE	phase-dependent
user-defined scalar source	DEFINE_SOURCE	mixture
turbulence dissipation rate	DEFINE_PROFILE	primary and secondary phase(s)

708

Variable	Macro	Phase Specified On
turbulence kinetic energy	DEFINE_PROFILE	primary and secondary phase(s)

Table 14: `DEFINE` Macro Usage for the Eulerian Model - Dispersed Turbulence Flow

Variable	Macro	Phase Specified On
Fluid		
velocity	DEFINE_PROFILE	primary and secondary phase(s)
temperature	DEFINE_PROFILE	primary and secondary phase(s)
species mass fraction	DEFINE_PROFILE	primary and secondary phase(s)
porosity	DEFINE_PROFILE	mixture
viscous resistance	DEFINE_PROFILE	primary and secondary phase(s)
inertial resistance	DEFINE_PROFILE	primary and secondary phase(s)
user-defined scalar	DEFINE_PROFILE	mixture
Wall		
species mass fraction	DEFINE_PROFILE	mixture
shear stress components	DEFINE_PROFILE	primary and secondary phase(s)
moving velocity components	DEFINE_PROFILE	mixture
heat flux	DEFINE_PROFILE	mixture
temperature	DEFINE_PROFILE	mixture
heat generation rate	DEFINE_PROFILE	mixture
heat transfer coefficient	DEFINE_PROFILE	mixture
external emissivity	DEFINE_PROFILE	mixture
external radiation temperature	DEFINE_PROFILE	mixture
free stream temperature	DEFINE_PROFILE	mixture
granular flux	DEFINE_PROFILE	secondary phase(s)
granular temperature	DEFINE_PROFILE	secondary phase(s)
user-defined scalar boundary value	DEFINE_PROFILE	mixture
discrete phase boundary value	DEFINE_DPM_BC	mixture
Material Properties		
granular diameter	DEFINE_PROPERTY	secondary phase(s)
granular viscosity	DEFINE_PROPERTY	secondary phase(s)

Variable	Macro	Phase Specified On
granular bulk viscosity	`DEFINE_PROPERTY`	secondary phase(s)
granular frictional viscosity	`DEFINE_PROPERTY`	secondary phase(s)
conductivity	`DEFINE_PROPERTY`	secondary phase(s)
granular solids pressure	`DEFINE_PROPERTY`	secondary phase(s)
granular radial distribution	`DEFINE_PROPERTY`	secondary phase(s)
granular elasticity modulus	`DEFINE_PROPERTY`	secondary phase(s)
turbulent viscosity	`DEFINE_TURBULENT_ VISCOSITY`	mixture, primary, and secondary phase(s)

Table 15: `DEFINE` Macro Usage for the Eulerian Model - Dispersed Turbulence Flow

Variable	Macro	Phase Specified On
Other		
drag coefficient	`DEFINE_EXCHANGE`	phase interaction
lift coefficient	`DEFINE_EXCHANGE`	phase interaction
heat transfer coefficient	`DEFINE_PROPERTY`	phase interaction
mass transfer coefficient	`DEFINE_MASS_TRANSFER`	phase interaction
heterogeneous reaction rate	`DEFINE_HET_RXN_RATE`	phase interaction

C.6. Eulerian Model - Per Phase Turbulence Flow

Table 16:`DEFINE` Macro Usage for the Eulerian Model - Per Phase Turbulence Flow (p. 710) –
Table 18:`DEFINE` Macro Usage for the Eulerian Model - Per Phase Turbulence Flow (p. 711) list the
variables that can be customized using UDFs for the per phase turbulence flow Eulerian multiphase
model, the `DEFINE` macros that are used to define the UDFs, and the phase that the UDF must be
hooked to for the given variable.

Table 16: `DEFINE` Macro Usage for the Eulerian Model - Per Phase Turbulence Flow

Variable	Macro	Phase Specified On
Boundary Conditions Inlet/Outlet		
volume fraction	`DEFINE_PROFILE`	secondary phase(s)
species mass fractions	`DEFINE_PROFILE`	phase-dependent
mass flux	`DEFINE_PROFILE`	primary and secondary phase(s)
velocity magnitude	`DEFINE_PROFILE`	primary and secondary phase(s)
temperature	`DEFINE_PROFILE`	primary and secondary phase(s)
pressure	`DEFINE_PROFILE`	mixture

Variable	Macro	Phase Specified On
user-defined scalar boundary value	DEFINE_PROFILE	mixture

Table 17: DEFINE Macro Usage for the Eulerian Model - Per Phase Turbulence Flow

Variable	Macro	Phase Specified On
Fluid		
mass source	DEFINE_SOURCE	primary and secondary phase(s)
momentum source	DEFINE_SOURCE	primary and secondary phase(s)
energy source	DEFINE_SOURCE	primary and secondary phase(s)
turbulence dissipation rate source	DEFINE_SOURCE	primary and secondary phase(s)
turbulence kinetic energy source	DEFINE_SOURCE	primary and secondary phase(s)
user-defined scalar source	DEFINE_SOURCE	mixture
velocity	DEFINE_PROFILE	primary and secondary phase(s)
temperature	DEFINE_PROFILE	primary and secondary phase(s)
turbulence kinetic energy	DEFINE_PROFILE	primary and secondary phase(s)
turbulence dissipation rate	DEFINE_PROFILE	primary and secondary phase(s)
granular flux	DEFINE_PROFILE	secondary phase(s)
granular temperature	DEFINE_PROFILE	secondary phase(s)
porosity	DEFINE_PROFILE	mixture
viscous resistance	DEFINE_PROFILE	primary and secondary phase(s)
inertial resistance	DEFINE_PROFILE	primary and secondary phase(s)
user-defined scalar	DEFINE_PROFILE	mixture

Table 18: DEFINE Macro Usage for the Eulerian Model - Per Phase Turbulence Flow

Variable	Macro	Phase Specified On
Wall		
species boundary condition	DEFINE_PROFILE	phase-dependent
shear stress components	DEFINE_PROFILE	primary and secondary phase(s)
moving velocity components	DEFINE_PROFILE	mixture

Variable	Macro	Phase Specified On
temperature	`DEFINE_PROFILE`	mixture
heat flux	`DEFINE_PROFILE`	mixture
heat generation rate	`DEFINE_PROFILE`	mixture
heat transfer coefficient	`DEFINE_PROFILE`	mixture
external emissivity	`DEFINE_PROFILE`	mixture
external radiation temperature	`DEFINE_PROFILE`	mixture
free stream temperature	`DEFINE_PROFILE`	mixture
granular flux	`DEFINE_PROFILE`	secondary phase(s)
granular temperature	`DEFINE_PROFILE`	secondary phase(s)
user-defined scalar boundary value	`DEFINE_PROFILE`	mixture
discrete phase boundary value	`DEFINE_DPM_BC`	mixture
Material Properties		
granular diameter	`DEFINE_PROPERTY`	secondary phase(s)
granular viscosity	`DEFINE_PROPERTY`	secondary phase(s)
granular bulk viscosity	`DEFINE_PROPERTY`	secondary phase(s)
granular frictional viscosity	`DEFINE_PROPERTY`	secondary phase(s)
granular conductivity	`DEFINE_PROPERTY`	secondary phase(s)
granular solids pressure	`DEFINE_PROPERTY`	secondary phase(s)
granular radial distribution	`DEFINE_PROPERTY`	secondary phase(s)
granular elasticity modulus	`DEFINE_PROPERTY`	secondary phase(s)
turbulent viscosity	`DEFINE_TURBULENT_ VISCOSITY`	mixture, primary, and secondary phase(s)
Other		
drag coefficient	`DEFINE_EXCHANGE`	phase interaction
lift coefficient	`DEFINE_EXCHANGE`	phase interaction
heat transfer coefficient	`DEFINE_PROPERTY`	phase interaction
mass transfer coefficient	`DEFINE_MASS_TRANSFER`	phase interaction
heterogeneous reaction rate	`DEFINE_HET_RXN_RATE`	phase interaction

Bibliography

[1] R. H. Aungier. "A Fast, Accurate Real Gas Equation of State for Fluid Dynamic Analysis Applications". *Journal of Fluids Engineering*. 117. 277–281. 1995.

[2] N. P. Cheremisinoff. *Fluid Flow Pocket Handbook*. Gulf Publishing Company. Houston, TX,1984.

[3] A. M. Douaud and P. Eyzat. "Four-Octane-Number Method for Predicting the Anti-Knock Behavior of Fuels in Engines". *SAE Transactions 780080*. 1978.

[4] E. R. G. Eckert and R. M. Drake. *Analysis of Heat and Mass Transfer*. McGraw-Hill Company. 1972.

[5] S. Jendoubi, H. S. Lee, and T. K. Kim. "Discrete Ordinates Solutions for Radiatively Participating Media in a Cylindrical Enclosure". *J. Thermophys. Heat Transfer*. 7(2). 213–219. 1993.

[6] B. Kernighan and D. Ritchie. *The C Programming Language*. Prentice-Hall, second edition. 1988.

[7] J. C. Livengood and P. C. Wu. "Correlation of Autoignition Phenomena in Internal Combustion Engines and Rapid Compression Machines". *In Fifth Symposium (Int.) on Combustion*. 347–356. 1955.

[8] M. J. Moran and H. N. Shapiro. *Fundamentals of Engineering Thermodynamics*. John Wiley & Sons, Incorporated. 1988.

[9] S. Oualline. *Practical C Programming*. O'Reilly Press. 1997.

[10] P. J. O'Rourke, A. A. Amsden. A Spray/Wall Interaction Submodel for the KIVA-3 Wall Film Model. SAE Paper 2000-01-0271. 2000.

[11] M. R Speigel. *Mathematical Handbook of Formulas and Tables Shaum's Outline Series*. McGraw-Hill Company. 1968.

Part 2: Creating Custom User Interfaces in Fluent

Chapter 1: Introduction to Fluent User Interface Concepts

This chapter contains an overview of the process of creating user interfaces that correspond to user-defined functions (UDFs) in Ansys Fluent.

1.1. Introduction

Interface-oriented user defined functions (UDFs) are UDFs that are meant to correspond to a custom graphical user interface (GUI). They provide an advantage in the ability to change data values whenever you need to without re-compilation. Furthermore, constructing a polished user interface to correspond to a UDF will create a consistent look and feel between Ansys Fluent and your UDF. This part of the manual will guide you through the process of creating a custom interface in Fluent that will allow you to edit and store data for later use by your UDF. Possible data types that are usable on your interface include integers, real numbers, Booleans (toggle buttons), strings, and lists. Throughout this part of the guide, you will be instructed in the process of creating an interface with each of these features, and learn how to save their values in a manner that can be accessed by your UDF.

The fundamental way in which you will create your custom interface is through the use of Fluent macros. Macros are Scheme constructs that are created by Ansys and are used in the development of the Fluent interface. Macros can have a variety of purposes in Fluent, including the use of RP Variables (p. 725), the creation of Interface Elements (p. 737), and the gathering of information from a mesh. As you read through this part of the guide, you will learn how to use a variety of Fluent macros as you learn to build a customized GUI.

1.2. Limitations

This section is about the limitations involved with creating your own interface in Fluent.

1.2.1. Menu Items Read Into Fluent Cannot Be Removed Or Overwritten

It is important to note that any Scheme file that includes a new menu or menu item being added to the right of the Fluent ribbon tabs cannot be removed once it is read into Fluent. This problem does

not exist for Scheme files that do not include a menu item that opens your GUI. If your Scheme file *does not* add a menu item to a menu that you have already created, you can always make changes to the Scheme file and save it before you read it in again. In this case, you can use the cx-show-panel (p. 730) to open the most recent version of the GUI you read into Fluent. If your Scheme file *does* include a menu item to open your interface at will, then that menu item is not removable from Fluent once it is read in.

Furthermore, the menu item *cannot* be overwritten by reading in the same file twice. If you try to make changes to your Scheme file and then read it in again, you will simply have two menu items of the same name that refer to the two versions of your interface that you have loaded into Fluent. Since having different interfaces of the same name can become confusing extremely quickly, it is recommended that if you are trying to make changes to a Scheme file that includes a menu item to open your interface, you should simply close and re-start Fluent in order to remove any former versions of your interface from the program. For more information on adding menus or menu items to the right of the Fluent ribbon tabs, see Adding Menus to the Right of the Ribbon (p. 755).

1.2.2. Help Button Unusable

Dialog boxes have a built-in Ansys **Help** button that appears at the bottom of the interface. Since the interface you are creating is not registered by Fluent, the **Help** button will not work. If you click **Help**, the Ansys Help opens and generates the error message `Cannot find requested page`.

1.3. Scheme Basics

GUI elements in Fluent are written in an implementation of the Scheme programming language. To load a Scheme file into Fluent, click **File**, select the **Read** submenu, and select **Scheme...**. Scheme statements are written in parenthesis, and statements are often nested within other statements. Comment lines in Scheme begin with a semi-colon, and variable names and Scheme commands are not case sensitive. In this section, you will learn about some basic Scheme statements and how you can use them as you create a GUI for your UDF. For a more complete guide to the Scheme language, visit http://www.scheme.com/tspl4/.

1.3.1. Data Types

1.3.2. Important Concepts

1.3.1. Data Types

Scheme allows for the use of a number of important data types. Those provided in this section are the types that Fluent allows you to pass to your user defined functions.

1.3.1.1. Boolean

1.3.1.2. Integers

1.3.1.3. Reals

1.3.1.4. Characters

1.3.1.5. Strings

1.3.1.6. Symbols

1.3.1.7. Pairs and Lists

1.3.1.1. Boolean

```
(define isBool #t)
(set! isBool #f)
(display isBool)
(boolean? isBool)
```

- In Scheme, booleans are initialized using the notation `#t` for true and `#f` for false.

- In the example above, a new variable `isBool` is being initialized to true and then being set to false.

- The `display` command is a console printing method that allows you to check the value of a variable. The line `(display isBool)` would output `#f`.

- The `boolean?` command is used to check if the following argument is a Boolean or not. The line `(boolean? isBool)` would output `#t`.

1.3.1.2. Integers

```
(define isInt 1)
(set! isInt (+ isInt 1))
(display isInt)
(integer? isInt)
(number? isInt)
```

- Integers in Scheme are similar to integers in most other languages.

- In the example above, the new variable `isInt` is initialized to `1`.

- In the next line, it is incremented to `2`. Remember that in Scheme the operation always comes before the data elements being operated on.

- Using the `display` command, we can check that the value of `isInt` is indeed `2`.

- The `integer?` command will check to see if `isInt` is an integer and the `number?` command will check to see if it is one of any of the number data types in Scheme. The output of both of these commands will be `#t`.

1.3.1.3. Reals

```
(define isReal 1.2)
(set! isReal (- isReal .5))
(display isReal)
(real? isReal)
(number? isReal)
```

- Real numbers in Scheme work in a similar way as the integers.

- In the example above, the new variable `isReal` is initialized to `1.2`.

- In the next line, it is set to its own value minus `0.5`. Remember that in Scheme the operation always comes before the data elements being operated on.

- The `display` command allows us to verify that the value of `isReal` is `0.7`.

- The commands `real?` and `number?` will both return `#t`.

1.3.1.4. Characters

```
(define isChar #\a)
(set! isChar #\b)
(display isChar)
(char? isChar)
```

- Characters in Scheme are always preceded by the notation `#\`.

- In the example above the variable `isChar` is initialized to the value a. In the next line, it is set instead to the value of b.

- By using the `display` command, we can check that the value of `isChar` is indeed b.

- The result of the `char?` command will be `#t`.

Note:

Although the example above proves that variables can be set to character values, you will usually just be working the character itself, as it makes little sense to assign a variable for a data type that is only 1 character.

1.3.1.5. Strings

```
(define isString "This is a string")
(set! isString "This is a different string")
(display isString)
(string? isString)
(string-length isString)
(string-ref isString 0)
(set! isString (string-append isString " plus more"))
```

- Strings in Scheme are very similar to strings in most other languages.

- In the example above, the variable `isString` is initialized to `"This is a string"`. In the next line, it is set instead to the value `"This is a different string"`.

- Using the `display` command, we can confirm that the value of `isString` is `"This is a different string"`.

- The `string?` command will confirm that `isString` is a string by outputting `#t`.

- The `string-length` command will return the number of characters in the string, in this case 26, and the `string-ref` command will return the character in the string at the position of the integer passed to it. In this case it would return the character at position 0, or `#\T`.

- The `string-append` command adds two strings together and returns the result as a third string. The example above uses a `string-append` command to combine `isString` with `" plus more"`. It also uses a `set!` command to save the result back to the variable `isString`.

720

The new value of `isString` would now be `"This is a different string plus more"`.

Note:

While the `string-ref` command asserts that a string starts a position 0, the `string-length` command will always treat strings as though they start at position 1. This makes sense, as a string with a `string-length` of 1 has one character at position 0.

1.3.1.6. Symbols

```
(define isSymbol 'abc)
(symbol? isSymbol)
(define isString "abc")
(eq? (string->symbol isString) isSymbol)
```

- Symbols are atomic values that are somewhat similar to strings. Symbols and strings could be used for many of the same purposes, but there are a few important differences that give symbols their own niche in Scheme.

- Multiple variable names that refer to the same symbol will be considered the same object, meaning that when using the `eq?` function on two different variables that refer to the same symbol content, you will get `#t` instead of `#f` like you would with strings.

- This is seen in the example above. A symbol, `isSymbol`, is initialized to `abc`. The `symbol?` command confirms that the variable is a symbol. Then, a string `isString` is initialized to the same value, `abc`. The following line illustrates two important points. First, any string can be cast as a symbol using the `string->symbol` command. Second, any two symbols with the same contents are considered to be the same object and are equal. In this case the `eq?` statement would return `#t`.

- Additionally, symbols are immutable, meaning that they cannot be altered once they are created. Strings on the other hand are mutable, meaning that they can be altered, for example, using `string-append` (see Strings (p. 720)).

- This shows that while strings have more versatility as a data type, symbols are a lightweight and easily comparable data type that is preferable when you know that you will not need to make any alterations once you have created it. As a result, symbols are seen very often in Scheme code in places where strings simply are not necessary.

1.3.1.7. Pairs and Lists

```
(a . b)
(define aPair (cons 1 2))
(a . (b . (c . ())))
(define aList (list 1 2 3 4 5))
(car aList)
(cdr aList)
(list-ref aList 2)
(list-tail aList 1)
```

- Pairs and lists are two related Scheme data constructs that are often used when multiple data items need to be stored together.

- Pairs are two data items that can be accessed from the same variable name. Pairs are often represented as two data items with a period between them, such as the first line in the example above. Pairs are created using the `cons` statement, as seen in the second line of the example above.

- Pairs can be used to create lists. Lists are pairs whose second element is another pair. This makes your list as long as you need it to be by having the right number of appended pairs.

- The third line of the example above shows how a list looks conceptually, made up of individual pairs. The fourth line in the example above shows you how to create a list using the `list` statement.

- The rest of the example above shows how to access certain pieces of data within the list. The `car` statement is used to access the first item in the list. In the example above, it would return `1`. The `cdr` statement is used to access everything but the first item in the list. In the example above, it would return `(2 3 4 5)`.

- The `list-ref` statement is used to access an item in a list by the position it is within the list. The `list-ref` statement in the example above would return `3`. The `list-tail` statement is used to return the remaining items in the list starting at the position you tell it to. In the example above, since it is passed position `1` as a starting point, the `list-tail` statement would return `(2 3 4 5)`.

1.3.2. Important Concepts

There are a number of Scheme concepts that are important to understand when creating your own Fluent interface. This section discusses those concepts that you are most likely to see in examples throughout this guide and use as you build your own interface.

1.3.2.1. Define

```
(define isInt 1)
(define whoKnows)
```

- The `define` statement is used to create a new variable in Scheme.

- Variables in Scheme do not have types, but rather return a value. The return values themselves have types, but using the variable name itself is like calling a function that can return a value of any type. For example, the variable `isInt` in the example above returns the integer `1`.

- Scheme variables do not have to be initialized with a value when they are created. In the example above the variable `whoKnows` is empty and can be assigned to a value of any type.

1.3.2.2. Set!

```
(define someVariable)
(set! someVariable 1)
(set! someVariable 2.5)
```

- The `set!` statement is used to assign a value to a variable that has already been created. If the variable has not been created via a `define` statement you will receive an `unbound variable` error.

- Since there are no variable types in Scheme, a variable can be overwritten with multiple types of data. In the example above, the variable `someVariable` is initially empty.

- After the first `set!` statement, `someVariable` will return the integer value of `1`. After the second `set!` statement however, `someVariable` will return the `real` value of `2.5` instead.

- This flexibility in variable use has its advantages, as it is easy to create a variable and begin working with it immediately. It can also be a problem if you aren't diligent in keeping track of each variable's return type.

1.3.2.3. Let

```
(let ((x 5))
  (let ((x 2)
        (y x))
    (+ y x)))
```

- `let` statements define the scope of variable bindings that occur within them. They are similar to `{}` in C and C++.

- The `let` statement is broken up into two parts. The first part begins with the parenthesis immediately following the word `let`. Within this set of parenthesis are the variable bindings that you want to define for the rest of the `let` statement.

- For the outer `let` in the example above, the first part consists of the variable binding `(x 5)`. For the inner `let` of the example above the first part consists of the variable bindings `(x 2)` and `(y x)`.

- The second part of the `let` statement is known as the body. Once the variable bindings are complete, this is where the operations using these variables occurs.

- For the outer `let` in the example above, the body consists of everything that follows the first part, including the inner `let`, until the final parenthesis. For the inner `let` in the example above, the body includes the statement `(+ y x)`.

- When `let`s are nested, as in the example above, and both the outer and inner `let` bind a value to the same variable, the value bound by the outer `let` is the one used until the first part of the inner `let` is closed.

- Therefore, the `(+ y x)` statement in the example above will use the value `2` for `x` as opposed to `5`. The value for `y` in this statement, however, will still be `5` because when `y` was assigned the value of `x`, it didn't see that `x` had been changed to `2` because the first part of the inner let wasn't closed yet. This makes the result of the addition `7`.

1.3.2.4. Lambda

```
(define doubleProcedure (lambda (x) (+ x x)))
(doubleProcedure 1)
```

- The `lambda` keyword is used to create a new Scheme procedure.

- When using the `lambda` keyword to make a new procedure, you must first provide a list of the variables that will be used in the procedure. Once the variables are declared, you can then specify the functionality of the procedure.

- This process is seen in the example above. Once the keyword `lambda` is used to make `doubleProcedure` a procedure, the variable `x` is declared. Since `x` is the only variable being used in this procedure, we can now move on to the functionality, which consists of the `(+ x x)` portion of the line.

- Now that the procedure has been created, we can call it using any number for `x`, as seen in the second line of the example above, which will return `2`.

1.3.2.5. If

```
(define ifExample (lambda (x) (if (>= x 0) (+ x 1) (- x 1))))
(ifExample 1)
(ifExample -1)
```

- The Scheme `if` statement has the same basic logic as `if` statements in most other programming languages, though the syntax is slightly different.

- In Scheme, the `if` statement is made up of three parts. The first part is the `test`, which determines how you will proceed through the rest of the `if` code.

- The second part is the `consequent`, which is the code block executed if the `test` is true.

- The third part is the `alternative`, which corresponds to an `else` statement in other popular languages. This code block is executed if the `test` is false.

- In the example above, a procedure called `ifExample` is created to show how the `if` statement works. For information about creating procedures in Scheme, see Lambda (p. 724).

- In the body of the procedure is a simple `if` statement. The `test` in this example is `(>= x 0)`, the `consequent` is `(+ x 1)`, and the `alternative` is `(- x 1)`.

- When passed a number, the `ifExample` procedure will increment the number if it is greater than or equal to 0 and decrement it if it is a negative number. Therefore, the function call `(ifExample 1)` will return `2` and the function call `(ifExample -1)` will return `-2`.

1.3.2.6. Map

```
(define halve (lambda (x) (/ x 2)))
(map halve (list 2 4 6 8 10))
```

- The Scheme `map` statement applies a desired function to each element of a list. For more information on lists, see Pairs and Lists (p. 721).

- The example above uses a function called `halve`. For more information on creating functions with a `lambda` statement, see Lambda (p. 724).

- In the example above, the function `halve` is created to divide any number passed to it by 2.

- In the second line, a map statement is used to apply the `halve` function to each number of the list `(list 2 4 6 8 10)`. This map statement returns `(1 2 3 4 5)`.

- The map statement can be used with essentially any function as long as that function is compatible with the list items on which it will operate.

1.4. RP Variables

RP variables are variables that are created for use in Fluent and provide a means of passing data from your GUI (in Scheme) to your compiled or interpreted UDF (in C). RP variables can be accessed from both your GUI and your UDF. Therefore, to pass data from your GUI to your UDF, you simply create and assign values to RP variables in your GUI code, and then access these same variables in your UDF code.

1.4.1. Creating an RP Variable

1.4.2. Changing an RP Variable

1.4.3. Accessing the Value of an RP Variable In Your GUI

1.4.4. Accessing the Value of an RP Variable In Your UDF

1.4.5. Saving and Loading RP Variables

1.4.1. Creating an RP Variable

To create a new RP variable in Scheme, you must use the macro `(rp-var-define)`. For example, the following command creates an integer RP variable named `myInt` with the default value of 1.

```
(rp-var-define 'myInt 1 'integer #f)
```

> **Note:**
>
> Symbols are usually used as arguments in Scheme functions rather than strings. To learn more about symbols, see Symbols (p. 721).

Before you create an RP variable, it is good practice to check that an RP variable by that name has not already been defined. One simple way to do this is to create a function, called `make-new-rpvar`, which checks the presence of a RP variable by that name before creating one. The `make-new-rpvar` function below can be copied into your Scheme file and used to make sure that no RP variables of the names you have chosen already exist before you create them.

```
(define (make-new-rpvar name default type)
  (if (not (rp-var-object name))
    (rp-var-define name default type #f)))

(make-new-rpvar 'myInt 1 'integer)
```

1.4.2. Changing an RP Variable

In order to change an RP variable from your Scheme GUI you must use the `(rpsetvar)` macro. For example, if you want to change the `myInt` variable created in Creating an RP Variable (p. 725) to 3 instead of 1, you can use the following statement:

```
(rpsetvar 'myInt 3)
```

Note:

The new value that you assign to an existing RP variable must be of the type originally declared or else you will receive an error.

`(rpsetvar)` statements are often used in the `apply-cb` function, which is called when you click the **OK** button. In this way, new values for RP variables are set when you click **OK** and the UDF can then access the new values when it needs to run.

1.4.3. Accessing the Value of an RP Variable In Your GUI

It is often necessary to access RP variable values in your GUI as well as your UDF. This can be useful in order to display the existing values of each variable each time the GUI is opened. By doing this, you know the existing value of an RP variable each time you go to change it. In order to access an RP variable in Scheme, you must use the `(%rpgetvar)` macro. This macro is often used to set the value of a local variable to the current value of an RP variable. For example, if you have an integer entry box called `localInt` and an integer RP variable called `rpInt`, you could set the value of `localInt` to the value of `rpInt` by using the following statement.

```
(cx-set-integer-entry localInt (%rpgetvar 'rpInt))
```

Since this statement is meant to update the values in each field when you open your GUI, this type of statement is usually seen in the `update-cb` function, which is called each time a GUI dialog box or task page is opened.

1.4.4. Accessing the Value of an RP Variable In Your UDF

In order to access the value of an RP variable for use in your UDF, see Accessing a Scheme Variable in a UDF (p. 384).

1.4.5. Saving and Loading RP Variables

Once new RP variables have been created their current values are stored in the case file each time the case file is saved. The order in which you load your case file and Scheme file does not matter when trying to load RP variable values from your case file. If the Scheme file is read before the case file, the RP variables will be created with the default values specified in the Scheme file. Then when the case file is read in, these values are overwritten with those in the case file. If the case file is read in before the Scheme file, the RP variables are created and set to the values specified in the case file.

When the Scheme file is read in, it will ignore the RP variable create statements when it recognizes that these RP variables have already been created by the case file (see note below).

> **Note:**
>
> In order to ensure that your Scheme file will recognize when RP variables have already been created by a case file, be sure to use the Scheme function defined in Creating an RP Variable (p. 725) when creating RP variables.

1.5. The .fluent File

When Fluent first starts up, it will look in your home folder for a file called `.fluent`. If it finds the `.fluent` file in your home folder it will load it as a Scheme file. Just as UDFs are automatically loaded into Fluent when you read a case file that includes a UDF, Scheme files can be automatically loaded into Fluent each time the program starts by using the `.fluent` file. The following steps will walk you through the creation and use of the `.fluent` file. The specific `.fluent` file created in this example loads in three separate example Scheme files: `Schemefile1.scm`, `Schemefile2.scm`, and `Schemefile3.scm`.

```
(ti-menu-load-string "file read-journal Schemefile1.scm")
(ti-menu-load-string "file read-journal Schemefile2.scm")
(ti-menu-load-string "file read-journal Schemefile3.scm")
```

• The first step is to copy each Scheme file that you want to load each time Fluent starts into your home folder.

• Next, you must open a new blank document in a basic text editor such as Notepad. In this blank document, you will add one line for each Scheme file you want to load when Fluent starts. These lines are Scheme commands that use the Fluent Text User Interface (TUI) to load each Scheme file accordingly. The example Scheme lines above would allow the `.fluent` file to load the Scheme files `Schemefile1.scm`, `Schemefile2.scm`, and `Schemefile3.scm`. The `ti-menu-load-string` portion of the command is a call to load the string that follows into the Fluent TUI. The string begins with `file read-journal` which tells Fluent to read in the Scheme file that follows. For more information on the Fluent TUI, see the Ansys Fluent Users Guide.

• Finally, you must save the document in your home folder with the name `.fluent`. Be aware that some text editors will force you to change the document type from the default type to `All Files (*.*)` before allowing you to define your own document type. Windows machines generally do not allow document names to begin with a period. In order to get around this problem, you must add a second period at the end of the name. For example, `.fluent.` would be the name you want to enter when you create your `.fluent` file. When you click save, you may receive a warning about your chosen filename depending on the text editor you are using. You should simply ignore this warning. When the document saves, it should remove the second period and name the file `.fluent` instead.

- You should now have all your desired Scheme files and a completed `.fluent` file in your home folder. From now on each Scheme file should be automatically be read in each time Fluent is started.

Note:

Be aware that Scheme files are read into Fluent in the order that they are requested in the `.fluent` file. Therefore, if one Scheme file creates a menu item for a dialog box that is defined in a different Scheme document, you should be sure to read in the Scheme file that defines the dialog box before you read in the Scheme file that creates the menu item for it. For more information on this issue, see the following:

- Adding a New Menu Item (p. 757)

- Example Menu Added to the Right of the Ribbon Tabs (p. 765)

Chapter 2: How to Create an Interface

This chapter contains an overview of how to create a simple interface in Fluent using Fluent Scheme macros. Once you understand the process of creating an interface you will be able to customize it using Interface Elements (p. 737) from the next chapter.

2.1. Dialog Boxes (cx-create-panel)

This section describes how you can create an interface in Fluent using a dialog box.

2.1.1. Description

The process that you will create a user interface to correspond with a user-defined function is to create a dialog box. Dialog boxes are created using the `cx-create-panel` macro. Once a dialog box is created, you must use the `cx-show-panel` macro to display it. Once you are able to create a simple dialog box, it will be easier to learn how to add more complex elements to it. Example One (p. 730) illustrates the simplest way to create a dialog box. It should be noted that this dialog box does not have any functionality. Example Two (p. 731) adds additional GUI elements but still lacks functionality. Dialog boxes use tables to format interface elements. For more information on tables, see Tables (cx-create-table) (p. 735). For more on the individual GUI elements seen in these examples, see Interface Elements (p. 737).

2.1.2. Usage

This section explains the arguments used in the various dialog box macros.

2.1.2.1. cx-create-panel

```
(cx-create-panel title apply-cb update-cb)
```

Argument	Type	Description
title	string	Name of the dialog box

Argument	Type	Description
apply-cb	function	Name of the function called when the **OK** button is clicked
update-cb	function	Name of the function called when the dialog box is opened

Note:

While the `apply-cb` and `update-cb` arguments are normally function names that are called when the dialog box is opened or the **OK** button is clicked, they do not need to be function names for the `cx-create-panel` macro to work. In many examples throughout part two of this guide, the `apply-cb` and `update-cb` arguments will be Boolean values rather than function names. The examples that use Boolean values instead of function names do not have any use when the **OK** button is clicked, but they do allow us to focus more on the specific elements of the interface rather than the overhead of setting up these functions. For examples where the `apply-cb` and `update-cb` arguments are set up as functions and allow functionality through RP variables, see Comprehensive Examples (p. 759).

2.1.2.2. cx-show-panel

```
(cx-show-panel panel)
```

Argument	Type	Description
panel	object	The variable you used to create the dialog box

2.1.3. Examples

This section provides examples of dialog boxes being created in Fluent.

2.1.3.1. Example One
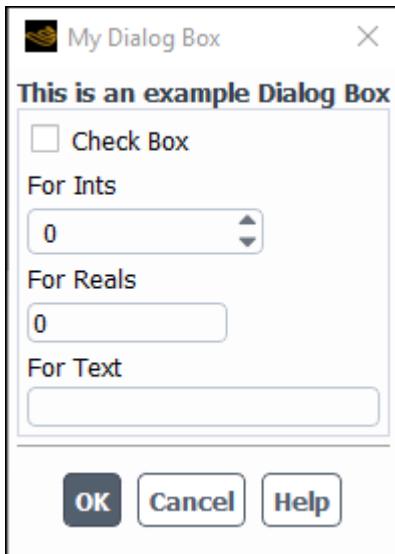
This is an example of a very simple dialog box containing only an empty table. The `apply-cb` and `update-cb` arguments must be included in the `cx-create-panel` statement, but in this case they have been simplified to just Boolean values rather than functions. Note that the `apply-cb` variable must be `#t` if you use a Boolean value instead of a function, because using `#f` will remove the **OK** button from the dialog box. Due to the lack of any data input fields on the dialog box and the use of Boolean values in the place of function calls, this dialog box does not serve any purpose, and is intended to simply illustrate the process of creating a dialog box.

```
(define (apply-cb) #t)
(define update-cb #f)
(define my-dialog-box (cx-create-panel "My Dialog Box" apply-cb update-cb))
(define table)
(set! table(cx-create-table my-dialog-box "This is an example Dialog Box"))
(cx-show-panel my-dialog-box)
```

2.1.3.2. Example Two

This is an example of a dialog box with a few basic data input fields added. The `apply-cb` and `update-cb` arguments must be included in the `cx-create-panel` statement, but in this case they have been simplified to just Boolean values rather than functions. Note that the `apply-cb` variable must be `#t` if you use a Boolean value instead of a function, because using `#f` will remove the **OK** button from the dialog box. Due to the use of boolean values in the place of function calls, this dialog box does not serve any purpose, and is intended to simply illustrate the process of creating a dialog box with various data input fields.


```
(define (apply-cb) #t)
(define update-cb #f)

(define table)
(define checkbox)
(define ints)
(define reals)
(define txt)

(define my-dialog-box (cx-create-panel "My Dialog Box" apply-cb update-cb))

(set! table (cx-create-table my-dialog-box "This is an example Dialog Box"))
```

```
(set! checkbox (cx-create-toggle-button table "Check Box" 'row 0))
(set! ints (cx-create-integer-entry table "For Ints" 'row 1))
(set! reals (cx-create-real-entry table "For Reals" 'row 2))
(set! txt (cx-create-text-entry table "For Text" 'row 3))

(cx-show-panel my-dialog-box)
```

2.1.3.3. Additional Examples

For further examples of dialog boxes with additional elements, see Comprehensive Examples (p. 759).

2.2. Task Pages (cx-create-taskpage)

This section describes how you can create an interface in Fluent using a task page.

2.2.1. Description

Another method of displaying GUI elements in Fluent is by using a task page. Task pages are different from dialog boxes because a task page is integrated directly into the Fluent interface rather than displaying your interface via a new window. This is helpful if you are working with several applications at once and don't have the screen space for a whole new UDF window. Task pages are very similar to dialog boxes, and are created using the `cx-create-taskpage` macro. Once a task page is created, you must use the `cx-show-taskpage` macro to display it. Example One (p. 733) illustrates the simplest way to create a task page. It should be noted that this task page does not have any functionality. Example Two (p. 734) adds additional GUI elements but still lacks functionality. Task pages use tables to format interface elements. For more information on tables, see Tables (cx-create-table) (p. 735).

> **Note:**
>
> The current task page selected in the selection screen on the left-most part of the Fluent interface does not affect your task page being opened.

2.2.2. Usage

This section explains the arguments used in the various task page macros.

2.2.2.1. cx-create-taskpage

```
(cx-create-taskpage title apply-cb update-cb)
```

Argument	Type	Description
title	string	Name of the task page

Argument	Type	Description
apply-cb	function	Name of the function called when the **OK** button is clicked
update-cb	function	Name of the function called when the task page is opened

Note:

While the `apply-cb` and `update-cb` arguments are normally function names that are called when the task page is opened or the **Apply** button is clicked, they do not need to be function names for the `cx-create-taskpage` macro to work. In many examples throughout part two of this guide, the `apply-cb` and `update-cb` arguments will be Boolean values rather than function names. The examples that use Boolean values instead of function names do not have any use when the **Apply** button is clicked, but they do allow us to focus more on the specific elements of the interface rather than the overhead of setting up these functions. For examples where the `apply-cb` and `update-cb` arguments are set up as functions and allow functionality through RP variables, see

2.2.2.2. cx-show-taskpage

```
(cx-show-taskpage taskpage)
```

Argument	Type	Description
taskpage	object	The variable you used to create the task page

2.2.3. Examples

This section provides examples of task pages being created in Fluent.

2.2.3.1. Example One

This is an example of a very simple task page containing only an empty table. The `apply-cb` and `update-cb` arguments must be included in the `cx-create-taskpage` statement, but in this case they have been simplified to just Boolean values rather than functions. Note that the `apply-cb` variable must be `#t` if you use a Boolean value instead of a function, because using `#f` will remove the **Apply** button from the task page. Due to the lack of any data input fields on the dialog box and the use of Boolean values in the place of function calls, this task page does not serve any purpose, and is intended to simply illustrate the process of creating a task page.
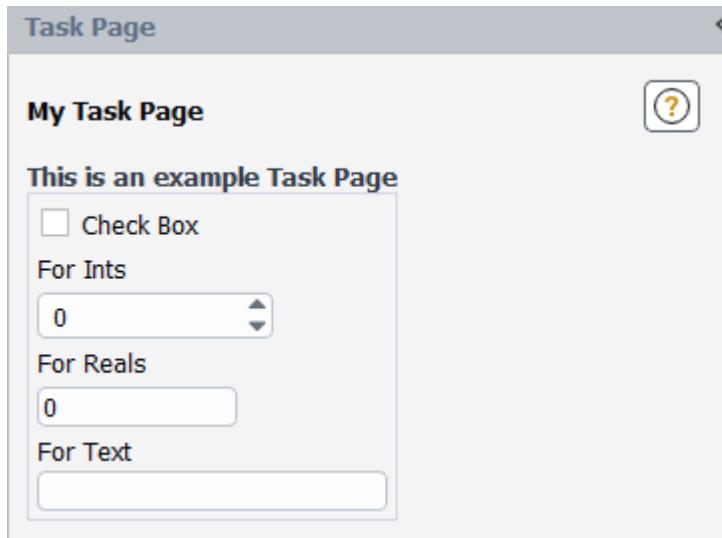
```
(define (apply-cb) #t)
(define update-cb #f)
(define my-taskpage (cx-create-taskpage "My Task Page" apply-cb update-cb))
(define table)
(set! table (cx-create-table my-taskpage "This is an example Task Page"))
(cx-show-taskpage my-taskpage)
```

2.2.3.2. Example Two

This is an example of a task page with a few basic data input fields added. The `apply-cb` and `update-cb` arguments must be included in the `cx-create-taskpage` statement, but in this case they have been simplified to just Boolean values rather than functions. Note that the `apply-cb` variable must be `#t` if you use a Boolean value instead of a function, because using `#f` will remove the **Apply** button from the task page. Due to the use of Boolean values in the place of function calls, this task page does not serve any purpose, and is intended to simply illustrate the process of creating a task page with various data input fields.


```
(define (apply-cb) #t)
(define update-cb #f)

(define table)
(define checkbox)
(define ints)
(define reals)
(define txt)

(define my-task-page (cx-create-taskpage "My Task Page" apply-cb update-cb))
```

```
(set! table (cx-create-table my-task-page "This is an example Task Page"))

(set! checkbox (cx-create-toggle-button table "Check Box" 'row 0))
(set! ints (cx-create-integer-entry table "For Ints" 'row 1))
(set! reals (cx-create-real-entry table "For Reals" 'row 2))
(set! txt (cx-create-text-entry table "For Text" 'row 3))

(cx-show-taskpage my-task-page)
```

2.2.3.3. Additional Examples

For further examples of task pages with additional elements, see Comprehensive Examples (p. 759).

2.3. Tables (cx-create-table)

This section describes how you can format an interface in Fluent using one or more tables.

2.3.1. Description

Tables are used in dialog boxes as a means to organize the various elements on the interface. Tables use the standard row/column style of organization to provide a simple way of formatting interface elements. Multiple tables can be added to the same dialog box in order to create groups of interface elements that are separate from others. Tables are created by using the `cx-create-table` macro. Once the table has been created, it can then be used as the parent attribute of other interface elements.

2.3.2. Usage

```
(cx-create-table parent label border below right-of)
```

Argument	Type	Description
parent	object	The name of the dialog box that you are adding the integer entry field to
label	string	The name of the table to be displayed on the GUI, can be left blank if no name is needed
border	symbol/boolean	Indicates the presence of visible borders in the table
below	symbol/int	Vertical position on the dialog box
right-of	symbol/int	Horizontal position on the dialog box
row	symbol/int	Row number if the table is being added to another table
column	symbol/int	Column number if the table is being added to another table

Note:

The border, below, right-of, row, and column attributes are optional.

2.3.3. Examples

Tables are used in every example in part two of this guide. To see examples of multiple tables used in a larger interface, see Comprehensive Examples (p. 759).

Chapter 3: Interface Elements

This chapter contains an overview of the various interface elements that you can add to your GUI.

3.1. Integer Entry (cx-create-integer-entry)

This section describes how you can add an integer entry field to your interface.

3.1.1. Description

This section discusses the ability to add an integer input field to your dialog box. An integer field is created using the `cx-create-integer-entry` macro. The value of the integer field can be set using the `cx-set-integer-entry` macro, and the value of an integer field can be obtained using the `cx-show-integer-entry` macro.

3.1.2. Usage

This section explains the arguments used in the various integer entry field macros

3.1.2.1. cx-create-integer-entry

```
(cx-create-integer-entry parent label row column)
```

Argument	Type	Description
parent	object	The name of the table that you are adding the integer entry field to
label	string	The name of the integer entry field to be displayed on the GUI
row	symbol/int	When added to a table, signifies the row that the integer entry field is added to
column	symbol/int	When added to a table, signifies the column that the integer entry field is added to

Note:

The row and column attributes are optional. If you leave out one or both of these attributes the integer entry field will be added to the first row/column of the parent attribute and overwrite anything that is already in that spot.

3.1.2.2. cx-set-integer-entry

```
(cx-set-integer-entry intentry value)
```

Argument	Type	Description
intentry	object	The variable you used to create the integer entry field
value	integer	The value you want to set the integer entry field to

3.1.2.3. cx-show-integer-entry
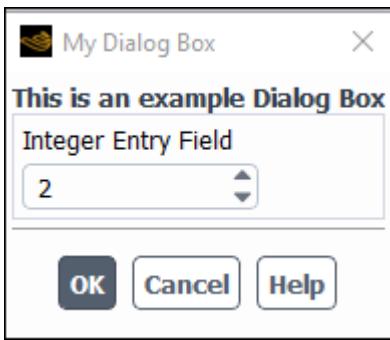
```
(cx-show-integer-entry intentry)
```

Argument	Type	Description
intentry	object	The variable you used to create the integer entry field

3.1.3. Integer Entry Example

This example shows how the `cx-create-integer-entry`, `cx-set-integer-entry` and `cx-show-integer-entry` macros work. Once the integer entry field has been created, the initial value of the field is set to `1` via the statement `(cx-set-integer-entry intField 1)`. Next, the value of that integer entry field is incremented to `2` using another `cx-set-integer-entry` statement. This statement also has a `cx-show-integer-entry` statement nested in it in order to get the value already in the integer entry field and increment it by `1`.

By the time the `cx-show-panel` statement is read, the value of `intField` is now 2, so the number 2 appears in the integer entry field when the dialog box is opened. This dialog box does not do anything when the **OK** button is clicked because we have substituted Boolean values for the `apply-cb` and `update-cb` arguments, which would normally be function calls. For more information on the `apply-cb` and `update-cb` functions, see cx-create-panel (p. 729).

```
(define (apply-cb) #t)
(define update-cb #f)

(define table)
(define intField)

(define my-dialog-box (cx-create-panel "My Dialog Box" apply-cb update-cb))

(set! table (cx-create-table my-dialog-box "This is an example Dialog Box"))

(set! intField (cx-create-integer-entry table "Integer Entry Field"))

(cx-set-integer-entry intField 1)

(cx-set-integer-entry intField (+ 1 (cx-show-integer-entry intField)))

(cx-show-panel my-dialog-box)
```

To view additional examples of integer entry fields, see Comprehensive Examples (p. 759).

3.2. Real Number Entry (cx-create-real-entry)

This section describes how you can add a real number entry field to your interface.

3.2.1. Description

3.2.2. Usage

3.2.3. Real Number Entry Example

3.2.1. Description

This section discusses the ability to add a real number input field to your dialog box. A real number field is created using the `cx-create-real-entry` macro. The value of the real number field can be set using the `cx-set-real-entry` macro, and the value of a real number field can be obtained using the `cx-show-real-entry` macro.

3.2.2. Usage

This section explains the arguments used in the various real number entry field macros

3.2.2.1. cx-create-real-entry

3.2.2.2. cx-set-real-entry

3.2.2.1. cx-create-real-entry

```
(cx-create-real-entry parent label row column)
```

Argument	Type	Description
parent	object	The name of the table that you are adding the real number entry field to
label	string	The name of the real number entry field to be displayed on the GUI
row	symbol/int	When added to a table, signifies the row that the real number entry field is added to
column	symbol/int	When added to a table, signifies the column that the real number entry field is added to

Note:

The row and column attributes are optional. If you leave out one or both of these attributes the real number entry field will be added to the first row/column of the parent attribute and overwrite anything that is already in that spot.

3.2.2.2. cx-set-real-entry

```
(cx-set-real-entry realentry value)
```

Argument	Type	Description
realentry	object	The variable used to create the real number entry field
value	real	The value that you want to set the real number entry field to

3.2.2.3. cx-show-real-entry

```
(cx-show-real-entry realentry)
```

Argument	Type	Description
realentry	object	the variable used to create the real number entry field

3.2.3. Real Number Entry Example

This example shows how the `cx-create-real-entry`, `cx-set-real-entry` and `cx-show-real-entry` macros work. Once the real number entry field has been created, the initial value of the field is set to `0.7` via the statement `(cx-set-real-entry realField 0.7)`. Next, the value of that integer entry field is increased to `1.2` using another `cx-set-real-entry` statement. This statement also has a `cx-show-real-entry` statement nested in it in order to get the value already in the integer entry field and add `0.5`.

By the time the `cx-show-panel` statement is read, the value of `realField` is now `1.2`, so the number `1.2` appears in the real number entry field when the dialog box is opened. This dialog box does not do anything when the **OK** button is clicked because we have substituted boolean values for the `apply-cb` and `update-cb` arguments, which would normally be function calls. For more information on the `apply-cb` and `update-cb` functions, see cx-create-panel (p. 729).


```
(define (apply-cb) #t)
(define update-cb #f)

(define table)
(define realField)

(define my-dialog-box (cx-create-panel "My Dialog Box" apply-cb update-cb))

(set! table (cx-create-table my-dialog-box "This is an example Dialog Box"))

(set! realField (cx-create-real-entry table "Real Entry Field"))

(cx-set-real-entry realField 0.7)

(cx-set-real-entry realField (+ 0.5 (cx-show-real-entry realField)))

(cx-show-panel my-dialog-box)
```

To view additional examples of real number entry fields, see Comprehensive Examples (p. 759).

3.3. Text Entry (cx-create-text-entry)

This section describes how you can add a text entry field to your interface.

3.3.1. Description

3.3.2. Usage

3.3.3. Text Entry Example

3.3.1. Description

This section discusses the ability to add text entry fields to your dialog box. Text entry fields are created with the `cx-create-text-entry` macro. The text of a text entry field can be set using the `cx-set-text-entry` macro, and can be obtained using the `cx-show-text-entry` macro.

3.3.2. Usage

This section explains the arguments used in the various text entry field macros

3.3.2.1. cx-create-text-entry

```
(cx-create-text-entry parent label row column)
```

Argument	Type	Description
parent	object	The name of the table that you are adding the text entry field to
label	string	The name of the text entry field to be displayed on the GUI
row	symbol/int	When added to a table, signifies the row that the text entry field is added to
column	symbol/int	When added to a table, signifies the column that the text entry field is added to

Note:

The row and column attributes are optional. If you leave out one or both of these attributes the text entry field will be added to the first row/column of the parent attribute and overwrite anything that is already in that spot.

3.3.2.2. cx-set-text-entry

```
(cx-set-text-entry text-var value)
```

Argument	Type	Description
text-var	object	The variable you used to create the text entry field
value	string	The value you want to set the text entry field to

3.3.2.3. cx-show-text-entry
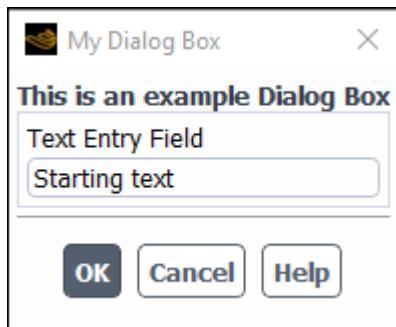
```
(cx-show-text-entry textentry)
```

Argument	Type	Description
text-var	object	The variable you used to create the text entry field

3.3.3. Text Entry Example

This example shows how the `cx-set-text-entry` and `cx-show-text-entry` macros work. Once the text entry field has been created, the initial value of the field is set to `Starting text` via the statement (`cx-set-text-entry txtField "Starting text"`). Next, the value of a random string `isString` is set to `This is different text` via the (`set! isString "This is different text"`) statement. After `isString` is set to `This is different text`, it is next set to the value of the text entry field through the use of a `cx-show-text-entry` statement.

Finally, the value of the text entry field is set to the value of `isString`. Since the text entry field says `Starting text` when the dialog box is opened, we know that the `cx-show-text-entry` statement works because it changed the value of `isString` to `Starting text`. This dialog box does not do anything when the **OK** button is clicked because we have substituted Boolean values for the `apply-cb` and `update-cb` arguments, which would normally be function calls. For more information on the `apply-cb` and `update-cb` functions, see cx-create-panel (p. 729).


```
(define (apply-cb) #t)
(define update-cb #f)

(define table)
(define txtField)
(define isString)

(define my-dialog-box (cx-create-panel "My Dialog Box" apply-cb update-cb))

(set! table (cx-create-table my-dialog-box "This is an example Dialog Box"))

(set! txtField (cx-create-text-entry table "Text Entry Field"))

(cx-set-text-entry txtField "Starting text")
(set! isString "This is different text")
(set! isString (cx-show-text-entry txtField))
(cx-set-text-entry txtField isString)

(cx-show-panel my-dialog-box)
```

To view additional examples of string entry fields, see Comprehensive Examples (p. 759).

3.4. Check Boxes & Radio Buttons (cx-create-toggle-button)

This section describes how you can add a check boxes and radio buttons to your interface.

3.4.1. Description

This section discusses the ability to add check boxes and radio buttons to your dialog box. Check boxes and radio buttons are normally grouped in a construct called a button box. The button box allows check boxes and radio buttons to be stacked on top of each other in a single spaced format. Button boxes also control whether or not the button you are adding is a check box or a option button. The difference between a check box and a option button is that multiple check boxes in a button box can be selected at the same time. Radio buttons are mutually exclusive, so only one option button in the button box can be selected at a time. Button boxes are created with the `cx-create-button-box` macro. Check boxes and radio buttons are created with the `cx-create-toggle-button` macro. The state of a check box or option button can be set with the `cx-set-toggle-button` macro and queried with the `cx-show-toggle-button` macro.

3.4.2. Usage

This section explains the arguments used in the various check box/option button macros

3.4.2.1. cx-create-button-box

```
(cx-create-button-box parent label radio-mode)
```

Argument	Type	Description
parent	object	The name of the table that you are adding the button box to
label	string	The name of the button box to be displayed on the GUI
radio-mode	symbol/Boolean	Allows choosing between check boxes and option buttons

3.4.2.2. cx-create-toggle-button

```
(cx-create-toggle-button parent label row column)
```

Argument	Type	Description
parent	object	The name of the button box that you are adding the check box/option button to

Argument	Type	Description
label	string	The name of the check box/option button to be displayed on the GUI

3.4.2.3. cx-set-toggle-button

```
(cx-set-toggle-button togglebutton value)
```

Argument	Type	Description
togglebutton	object	The variable you used to create the check box/option button
value	boolean	The Boolean value that you want to set the check box/option button to

3.4.2.4. cx-show-toggle-button

```
(cx-show-toggle-button togglebutton)
```

Argument	Type	Description
togglebutton	object	The variable you used to create the check box/option button

3.4.3. Check Box Example

This example shows how the `cx-create-button-box`, `cx-create-toggle-button`, `cx-set-toggle-button` and `cx-show-toggle-button` macros work. In the `cx-create-button-box` line the `radio-mode` argument is set to `#f`, which indicates that check boxes are being used, not radio buttons. Once the two check boxes have been created, the initial value of `checkBox1` is set to `#f` via the statement `(cx-set-toggle-button checkBox1 #f)`. Next, the value of a random Boolean variable `isBool` is set to `#t` via the `(set! isBool #t)` statement. After `isBool` is set to `#t`, it is next set to the value of `checkBox1` through the use of a `cx-show-toggle-button` statement.

Finally, the value of `checkBox2` is set to the value of `isBool`. Since `checkBox2` isn't checked when the dialog box is opened, we know that the `cx-show-toggle-button` statement works because it changed the value of `isBool` to `#f`. This dialog box does not do anything when the **OK** button is clicked because we have substituted Boolean values for the `apply-cb` and `update-cb` arguments, which would normally be function calls. For more information on the `apply-cb` and `update-cb` functions, see cx-create-panel (p. 729).

```
(define (apply-cb) #t)
(define update-cb #f)

(define checkBox1)
(define checkBox2)
(define isBool)

(define my-dialog-box (cx-create-panel "My Dialog Box" apply-cb update-cb))

(define table (cx-create-table my-dialog-box "This is an example Dialog Box"))

(define buttonBox (cx-create-button-box table "Button Box" 'radio-mode #f))

(set! checkBox1 (cx-create-toggle-button buttonBox "Check Box 1"))
(set! checkBox2 (cx-create-toggle-button buttonBox "Check Box 2"))

(cx-set-toggle-button checkBox1 #f)
(set! isBool #t)
(set! isBool (cx-show-toggle-button checkBox1))
(cx-set-toggle-button checkBox2 isBool)

(cx-show-panel my-dialog-box)
```

To view additional examples of check boxes, see Comprehensive Examples (p. 759).

3.4.4. Option Button Example

This example shows how the `cx-create-button-box`, `cx-create-toggle-button`, and `cx-set-toggle-button` macros work. In the `cx-create-button-box` statement the `radio-mode` argument is set to `#t` which indicates that radio buttons are being used, not check boxes. Once all four radio buttons are created, a `cx-set-toggle-button` macro is used to set `radioButton2` to `#t`, meaning that this option button will be selected when the dialog box is opened. Since these are radio buttons and not check boxes, this is the only button in the button box that can be selected. If you select any other button, `radioButton2` will automatically be deselected.

```
(define (apply-cb) #t)
(define update-cb #f)

(define radioButton1)
(define radioButton2)
(define radioButton3)
(define radioButton4)

(define my-dialog-box (cx-create-panel "My Dialog Box" apply-cb update-cb))

(define table (cx-create-table my-dialog-box "This is an example Dialog Box"))

(define buttonBox (cx-create-button-box table "Button Box" 'radio-mode #t))

(set! radioButton1 (cx-create-toggle-button buttonBox "Radio Button 1"))
(set! radioButton2 (cx-create-toggle-button buttonBox "Radio Button 2"))
(set! radioButton3 (cx-create-toggle-button buttonBox "Radio Button 3"))
(set! radioButton4 (cx-create-toggle-button buttonBox "Radio Button 4"))

(cx-set-toggle-button radioButton2 #t)

(cx-show-panel my-dialog-box)
```

To view additional examples of radio buttons, see Comprehensive Examples (p. 759).

3.5. Buttons (cx-create-button)

This section describes how you can add a button to your interface.

3.5.1. Description

While dialog boxes come with standard **OK** and **Cancel** buttons, it can sometimes be useful to add additional buttons with different functionality than the **OK** button. To build a new button, you must use the `cx-create-button` macro.

3.5.2. Usage

```
(cx-create-button parent label callback row column)
```

Argument	Type	Description
parent	object	The name of the table that you are adding the button to
label	string	The name of the button to be displayed on the GUI
callback	symbol/function	The name of the function called when the button is clicked
row	symbol/int	When added to a table, signifies the row that the button is added to
column	symbol/int	When added to a table, signifies the column that the button is added to
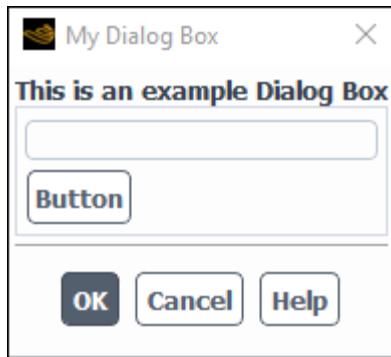
Note:

The row and column attributes are optional. If you leave out one or both of these attributes the button will be added to the first row/column of the parent attribute and overwrite anything that is already in that spot.

3.5.3. Button Example

This example shows how the `cx-create-button` macro is used to create a new button, and how the callback argument works. In the `cx-create-button` statement below, the `'activate-callback button-cb` argument ensures that the `button-cb` procedure is called each time that the button is clicked. The `button-cb` function is set up with the line `(define (button-cb . args)`, which is just like how the `apply-cb` and `update-cb` functions are set up in a fully functional dialog box (see Comprehensive Examples (p. 759)). Once you open the `button-cb` procedure with this line you can then write the code to give it functionality.

In this example, the variable `counter` is incremented by `1` each time the button is clicked and the number of times that the button has been clicked is output to a text entry field `txtField`. This dialog box does not do anything when the **OK** button is clicked because we have substituted Boolean values for the `apply-cb` and `update-cb` arguments, which would normally be function calls like the `button-cb` argument is in the `cx-create-button` line. For more information on the `apply-cb` and `update-cb` functions, see cx-create-panel (p. 729).


```
(define (apply-cb) #t)
(define update-cb #f)
```

```
(define table)
(define txtField)

(define counter 0)
(define (button-cb . args)
  (set! counter (+ counter 1))
  (cx-set-text-entry txtField (string-append "Times Clicked: " (number->string counter)))
)

(define my-dialog-box (cx-create-panel "My Dialog Box" apply-cb update-cb))

(set! table (cx-create-table my-dialog-box "This is an example Dialog Box"))

(set! txtField (cx-create-text-entry table "" 'row 0 'col 0))
(cx-create-button table "Button" 'activate-callback button-cb 'row 1 'col 0)

(cx-show-panel my-dialog-box)
```
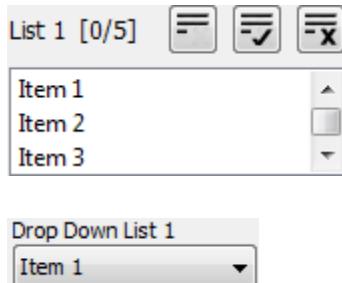
To view additional examples of buttons, see Comprehensive Examples (p. 759).

3.6. Lists & Drop-down Lists (cx-create-list) & (cx-create-drop-down-list)

This section describes how you can add a list or drop-down list to your interface.

3.6.1. Description

This section discusses the ability to add lists and drop-down lists to your dialog box. Lists are created with the `cx-create-list` macro and drop-down lists are created with the `cx-create-drop-down-list` macro. Both lists and drop-down lists use the same list-items macros once they are created. List items can be set using the `cx-set-list-items` macro. List selections can be set with the `cx-set-list-selections` macro and obtained with the `cx-show-list-selections` macro.

3.6.2. Usage

This section explains the arguments used in the various list and drop-down list macros.

3.6.2.1. cx-create-list

```
(cx-create-list parent label visible-lines multiple-selections row
column)
```

Argument	Type	Description
parent	object	The name of the table that you are adding the list to
label	string	The name of the list to be displayed on the GUI
visible-lines	symbol/int	The number of lines of text visible before a scroll bar is needed
multiple-selections	symbol/Boolean	Allows you to choose between having one item at a time selected or multiple
row	symbol/int	When added to a table, signifies the row that the list is added to
column	symbol/int	When added to a table, signifies the column that the list is added to

Note:

The `visible-lines`, `multiple-selections`, `row`, and `column` attributes are optional. If they are not included, the number of visible lines has a default of ten and multiple-selections is `#f`, meaning that you can only select one list item at a time. Furthermore, if the row or column attributes are left out then the list will be added to the first row/column of the parent attribute and overwrite anything that is already in that spot.

3.6.2.2. cx-create-drop-down-list

```
(cx-create-drop-down-list parent label multiple-selections row column)
```

Argument	Type	Description
parent	object	The name of the table that you are adding the drop-down list to
label	string	The name of the drop-down list to be displayed on the GUI
row	symbol/int	When added to a table, signifies the row that the drop-down list is added to

750

Argument	Type	Description
column	symbol/int	When added to a table, signifies the column that the drop-down list is added to

---

Note:

The `row` and `column` attributes are optional. If not included, the drop-down list will be added to the first row/column of the parent attribute and overwrite anything that is already in that spot.

3.6.2.3. cx-set-list-items

```
(cx-set-list-items list items)
```

Argument	Type	Description
list	list	The name of the list you are adding items to
items	strings	The strings that represent each item to be added to the list

3.6.2.4. cx-set-list-selections

```
(cx-set-list-selections list selections)
```

Argument	Type	Description
list	list	The name of the list you are setting the selections of
selections	list	The members of the list that you want selected

3.6.2.5. cx-show-list-selections

```
(cx-show-list-selections list)
```

Argument	Type	Description
list	list	The name of the list that you are obtaining the selections from
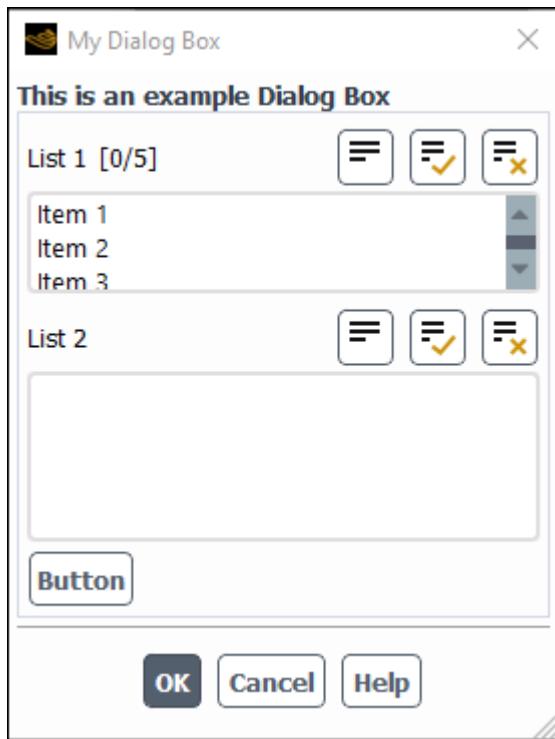
3.6.3. List Example

This example shows how the `cx-create-list`, `cx-set-list-items`, `cx-set-list-selections`, and `cx-show-list-selections` macros work. Both of the lists are created via `cx-create-list` statements. The first list only allows for three visible list items at a time while the second list allows for five to be shown at a time. This is an example of how to set the size of your list via the `visible-lines` argument.

Upon opening the dialog box the first list will automatically load all five list items via a `cx-set-list-items` line while the second list starts empty. By selecting one or multiple list items from the first list and then clicking on **Button**, the selected list items from `list 1` will be gathered via a `cx-show-list-selections` statement and added to `list 2` via a `cx-set-list-items` statement.

The list items added to `list 2` will also automatically be selected via a `cx-set-list-selections` statement.

The extra button is necessary in this example because setting up the functionality of the **OK** button will cause it to close the dialog box each time it is clicked. To review how the **Button** in this example works, see Buttons (cx-create-button) (p. 747). This dialog box does not do anything when the **OK** button is clicked because we have substituted Boolean values for the `apply-cb` and `update-cb` arguments, which would normally be function calls like the `button-cb` argument is in the `cx-create-button` line. For more information on the `apply-cb` and `update-cb` functions, see cx-create-panel (p. 729).


```
(define (apply-cb) #t)
(define update-cb #f)

(define table)
(define myList1)
(define myList2)

(define (button-cb . args)
  (cx-set-list-items myList2 (cx-show-list-selections myList1))
  (cx-set-list-selections myList2 (cx-show-list-selections myList1))
)

(define my-dialog-box (cx-create-panel "My Dialog Box" apply-cb update-cb))

(set! table (cx-create-table my-dialog-box "This is an example Dialog Box"))

(set! myList1 (cx-create-list table "List 1" 'visible-lines 3 'multiple-selections #t 'row 0))
(cx-set-list-items myList1 (list "Item 1" "Item 2" "Item 3" "Item 4" "Item 5"))

(set! myList2 (cx-create-list table "List 2" 'visible-lines 5 'multiple-selections #t 'row 1))

(cx-create-button table "Button" 'activate-callback button-cb 'row 2)

(cx-show-panel my-dialog-box)
```
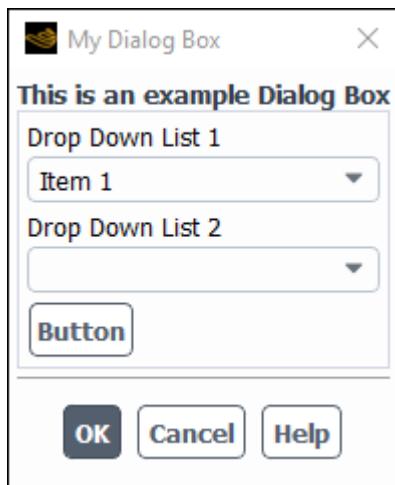
3.6.4. Drop Down List Example

This example shows how the `cx-create-drop-down-list`, `cx-set-list-items`, `cx-set-list-selections`, and `cx-show-list-selections` macros work. Since both lists and drop-down lists use the same `list` data type this example is very similar to the example above. The major difference with drop-down lists is that they don't allow for multiple selections. In the example, both of the drop-down lists are created via `cx-create-drop-down-list` statements.

Upon opening the dialog box the first drop-down list will automatically load all five list items via a `cx-set-list-items` line while the second drop-down list starts empty. By selecting one list item from the first drop-down list and then clicking on **Button**, the selected list item from `Drop Down List 1` will be gathered via a `cx-show-list-selections` statement and added to `Drop Down List 2` via a `cx-set-list-items` statement. The list item added to `list 2` will also automatically be selected via a `cx-set-list-selections` statement.

The extra button is necessary in this example because setting up the functionality of the **OK** button will cause it to close the dialog box each time it is clicked. To review how the **Button** in this example works, see Buttons (cx-create-button) (p. 747). This dialog box does not do anything when the **OK** button is clicked because we have substituted Boolean values for the `apply-cb` and `update-cb` arguments, which would normally be function calls like the `button-cb` argument is in the `cx-create-button` line. For more information on the `apply-cb` and `update-cb` functions, see cx-create-panel (p. 729).


```
(define (apply-cb) #t)
(define update-cb #f)

(define table)
(define myDropList1)
(define myDropList2)

(define (button-cb . args)
  (cx-set-list-items myDropList2 (cx-show-list-selections myDropList1))
)

(define my-dialog-box (cx-create-panel "My Dialog Box" apply-cb update-cb))

(set! table (cx-create-table my-dialog-box "This is an example Dialog Box"))

(set! myDropList1 (cx-create-drop-down-list table "Drop Down List 1" 'row 0))
(cx-set-list-items myDropList1 (list "Item 1" "Item 2" "Item 3" "Item 4" "Item 5"))

(set! myDropList2 (cx-create-drop-down-list table "Drop Down List 2" 'row 1))
```

```
(cx-create-button table "Button" 'activate-callback button-cb 'row 2)

(cx-show-panel my-dialog-box)
```

Chapter 4: Adding Menus to the Right of the Ribbon

This chapter contains an overview on the process of adding a new menu for opening your GUI.

4.1. Adding a New Menu

This section describes how you can add a new menu to the right of the Fluent ribbon tabs.

4.1.1. Description

This section discusses adding new menu to the right of the Fluent ribbon tabs using the `cx-add-menu.` macro.

4.1.2. Usage

`(cx-add-menu name mnemonic)`

Argument	Type	Description
name	string	Name of the Menu

755

Argument	Type	Description
mnemonic	char	This is a placeholder. The only mnemonic is **Alt** + **F** for the **File** menu.

Note:

Mnemonics are not available for user-defined menus, however you must still provide a value for this field, for example `#f`.

Note:

The names of menus and submenus should be unique to avoid conflict when adding menu items. If two menus/submenus of the same name do exist, a menu item being added to that name will be added to the last menu/submenu to be created.

4.1.3. Examples

Since all of the menu macros are best shown and implemented together, see Example Menu Added to the Right of the Ribbon Tabs (p. 765) for examples of menus being added to the right of the Fluent ribbon tabs.

4.2. Adding a New Submenu

This section describes how you can add a new submenu to menus that you have already created.

4.2.1. Description

Using Scheme, you can add submenus to menus you have already defined by using the `cx-add-hitem` macro.

4.2.2. Usage

`(cx-add-hitem menu item mnemonic)`

Argument	Type	Description
menu	string	The name of the menu that you are adding the submenu to
item	string	The name of the submenu you are adding

Argument	Type	Description
mnemonic	char	This is a placeholder. The only mnemonic is **Alt** + **F** for the **File** menu.

> **Note:**
>
> Mnemonics are not available for user-defined menus, however you must still provide a value for this field, for example `#f`.
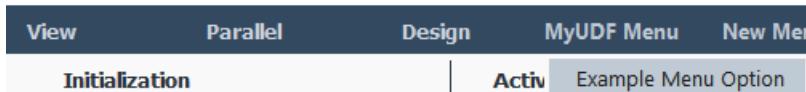
> **Note:**
>
> The names of menus and submenus should be unique to avoid conflict when adding menu items. If two menus/submenus of the same name do exist, a menu item being added to that name will be added to the last menu/submenu to be created.

4.2.3. Examples

Since all of the menu macros are best shown and implemented together, see Example Menu Added to the Right of the Ribbon Tabs (p. 765) for examples of menus being added to the right of the Fluent ribbon tabs.

4.3. Adding a New Menu Item

This section describes how you can add a new menu item to menus you have already created.

4.3.1. Description

This section discusses the ability to add a menu item to an already existing menu or submenu. Menu items are used to execute procedures that will open dialog boxes. This is the primary way that you will access your GUI for your UDF. Once you have created a menu and submenu for your GUI, you must create a menu item in order to open it. Use the `cx-add-item` macro, to add a menu item to an existing menu or submenu.

> **Note:**
>
> The dialog box that the menu item references must be defined in Fluent before the menu item can be created. If the menu item is created before the dialog box, the menu item will not show up in the menu after that interface is read into Fluent.

4.3.2. Usage

```
(cx-add-item menu item mnemonic hotkey test callback)
```

Argument	Type	Description
menu	string	The name of the menu or submenu that you are adding the menu item to
item	string	The name of the item you are adding to the menu or submenu
mnemonic	char	This is a placeholder. The only mnemonic is **Alt** + **F** for the **File** menu.
hotkey	char	This is a placeholder.
test	function	The name of a function that must return #t for the GUI to open
callback	function	The name of the procedure, in Scheme, of the dialog box that will open when you click this menu item

Note:

The hotkey and mnemonic fields are just placeholders; use #f in place of these fields.

Note:

The test parameter is designed for those who only want the menu item to be used if a specific condition is met. In this case, since we want the GUI to open when the menu item is clicked regardless of what else is happening in Fluent, we can simply substitute #t for the name of a function that would return #t.

4.3.3. Examples

Since all of the menu macros are best shown and implemented together, see Example Menu Added to the Right of the Ribbon Tabs (p. 765) for examples of menus being added to the right of the Fluent ribbon tabs.

Chapter 5: Comprehensive Examples

This chapter contains examples of each of the different GUI elements used on both of the two GUI types. Each of these examples corresponds to the user-defined function UDF Example (p. 766).

5.1. Dialog Box Example

This section contains the code for a dialog box that includes all of the features discussed in part two of this guide. This example can be used in conjunction with the UDF Example (p. 766) found later in this chapter in order to verify that the data is passed from the dialog box to the UDF using RP variables. This example contains comments in order to help readers understand the various parts of the example. All Scheme comment lines start with a semi-colon. This code automatically creates a menu called **New Menu**, with a submenu called **New Submenu**. The dialog box is automatically added as a menu item in **New Submenu**.
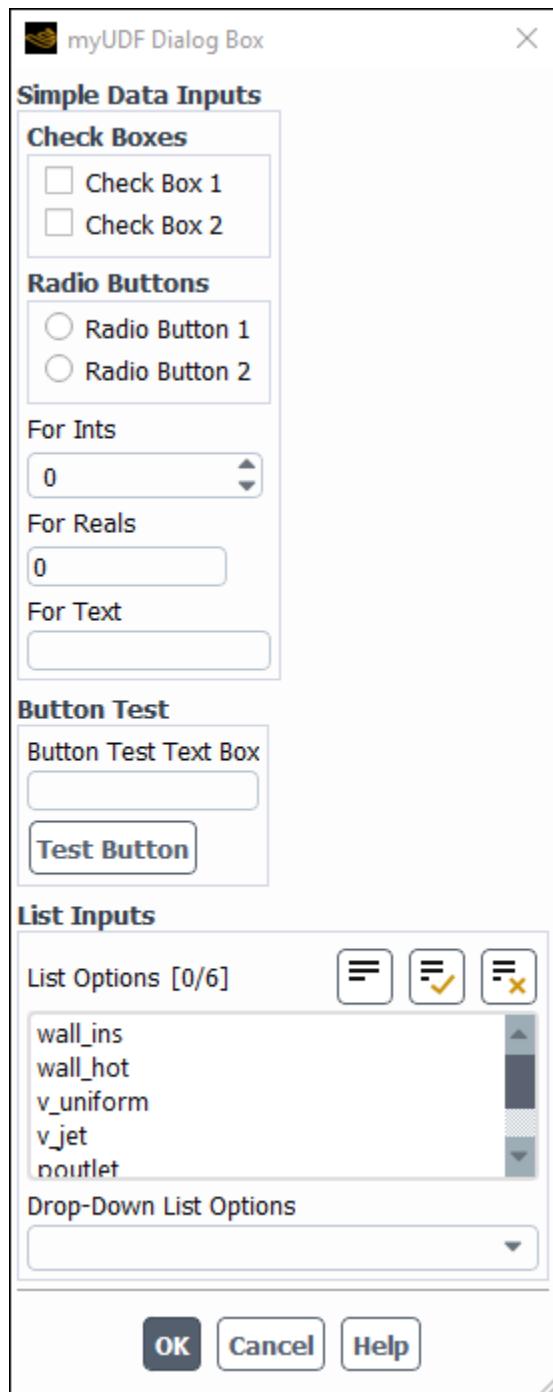
> **Note:**
>
> In order for the **List Options** and **Drop-Down List Options** fields to populate, you must first read a mesh into Fluent because it uses information from the mesh as the list items.

This example uses a couple of Fluent macros that are not seen previously in part two of this guide. The line `(map thread-name(get-face-threads))` in the `update-cb` function is used to populate the list and drop-down list fields using face threads from your current mesh when the dialog box is opened. As seen in chapter one, the Map (p. 724) statement applies a single function over a list of data. This makes `thread-name` the function, which simply gets the name of the thread name as text from the thread itself. The `get-face-threads` part of the statement is the list that the function is being applied to, meaning that this statement returns a list of face threads from the mesh on which the `thread-name` function can operate.

The other macro in this example that is not covered previously in part two of this guide is the `(%run-udf-apply)` macro. This macro is used because the corresponding UDF Example (p. 766) uses a `DEFINE_EXECUTE_FROM_GUI` statement. This allows you to run your UDF from the **OK** button of your GUI. The `(%run-udf-apply)` macro is used in the `apply-cb` function as a call to run the UDF. This macro requires one integer parameter. This integer is used in the UDF to determine what the programmer asked it to do. This is seen in the example below. The **Test Button** has a different job then the **OK** button, so it passes the integer 1 instead of 2. When the UDF runs, it will test the value of that integer and decide what to do depending on what the integer is. The `DEFINE_EXECUTE_FROM_GUI` statement is the only `DEFINE` statement that uses a Scheme macro in the GUI code to invoke the UDF. Other `DEFINE` macros allow the UDF to be run at various times during a simulation or, in the case of

the `DEFINE_EXECUTE_ON_DEMAND` statement, whenever you want by right-clicking **User Defined Functions** under the **Parameters & Customization** branch of the tree and selecting **Execute on Demand...**. For more information on the various `DEFINE` statements available for use in your UDF, see `DEFINE` Macros (p. 19).


```
;RP Variable Create Function
(define (make-new-rpvar name default type)
  (if (not (rp-var-object name))
      (rp-var-define name default type #f)))

;RP Variable Declarations
(make-new-rpvar 'myudf/real 0 'real)
(make-new-rpvar 'myudf/int 0 'int)
```

```
(make-new-rpvar 'myudf/checkbox1 #f 'boolean)
(make-new-rpvar 'myudf/checkbox2 #f 'boolean)
(make-new-rpvar 'myudf/radiobutton1 #f 'boolean)
(make-new-rpvar 'myudf/radiobutton2 #f 'boolean)
(make-new-rpvar 'myudf/string "" 'string)
(make-new-rpvar 'myudf/buttonstr "" 'string)
(make-new-rpvar 'myudf/list '() 'string-list)
(make-new-rpvar 'myudf/droplist '() 'string-list)

;Dialog Box Definition
(define gui-dialog-box
  ;Let Statement, Local Variable Declarations
  (let ((dialog-box #f)
    (table)
        (myudf/box1)
        (myudf/box2)
        (myudf/box3)
        (myudf/buttonbox1)
        (myudf/checkbox1)
        (myudf/checkbox2)
        (myudf/buttonbox2)
        (myudf/radiobutton1)
        (myudf/radiobutton2)
        (myudf/real)
        (myudf/int)
        (myudf/string)
        (myudf/buttonstr)
        (myudf/list)
        (myudf/droplist)
        )

        ;Update-CB Function, Invoked When Dialog Box Is Opened
        (define (update-cb . args)
          (cx-set-toggle-button myudf/checkbox1 (rpgetvar 'myudf/checkbox1))
          (cx-set-toggle-button myudf/checkbox2 (rpgetvar 'myudf/checkbox2))
          (cx-set-toggle-button myudf/radiobutton1 (rpgetvar 'myudf/radiobutton1))
          (cx-set-toggle-button myudf/radiobutton2 (rpgetvar 'myudf/radiobutton2))
          (cx-set-integer-entry myudf/int (rpgetvar 'myudf/int))
          (cx-set-real-entry myudf/real (rpgetvar 'myudf/real))
          (cx-set-text-entry myudf/string (rpgetvar 'myudf/string))
          (cx-set-text-entry myudf/buttonstr (rpgetvar 'myudf/buttonstr))
          (cx-set-list-items myudf/list (map thread-name(get-face-threads)))
          (cx-set-list-selections myudf/list (rpgetvar 'myudf/list))
          (cx-set-list-items myudf/droplist (map thread-name(get-face-threads)))
          (cx-set-list-selections myudf/droplist (rpgetvar 'myudf/droplist))
        )

        ;Apply-CB Function, Invoked When "OK" Button Is Clicked
        (define (apply-cb . args)
          (rpsetvar 'myudf/checkbox1 (cx-show-toggle-button myudf/checkbox1))
          (rpsetvar 'myudf/checkbox2 (cx-show-toggle-button myudf/checkbox2))
          (rpsetvar 'myudf/radiobutton1 (cx-show-toggle-button myudf/radiobutton1))
          (rpsetvar 'myudf/radiobutton2 (cx-show-toggle-button myudf/radiobutton2))
          (rpsetvar 'myudf/int (cx-show-integer-entry myudf/int))
          (rpsetvar 'myudf/real (cx-show-real-entry myudf/real))
          (rpsetvar 'myudf/string (cx-show-text-entry myudf/string))
          (rpsetvar 'myudf/buttonstr (cx-show-text-entry myudf/buttonstr))
          (rpsetvar 'myudf/list (cx-show-list-selections myudf/list))
          (rpsetvar 'myudf/droplist (cx-show-list-selections myudf/droplist))
          (%run-udf-apply 2)
        )

        ;Button-CB Function, Invoked When "Test Button" Is Clicked
        (define (button-cb . args)
          (rpsetvar 'myudf/buttonstr (cx-show-text-entry myudf/buttonstr))
          (%run-udf-apply 1)
        )

        ;Args Function, Used For Interface Setup, Required For Apply-CB, Update-CB, and Button-CB Sections
        (lambda args
          (if (not dialog-box)
```

```
          (let ()
            (set! dialog-box (cx-create-panel "myUDF Dialog Box" apply-cb update-cb))
            (set! table (cx-create-table dialog-box "" 'border #f 'below 0 'right-of 0))

            (set! myudf/box1 (cx-create-table table "Simple Data Inputs" 'row 0))
            (set! myudf/box2 (cx-create-table table "Button Test" 'row 1))
            (set! myudf/box3 (cx-create-table table "List Inputs" 'row 2))

            (set! myudf/buttonbox1 (cx-create-button-box myudf/box1 "Check Boxes" 'radio-mode #f 'row 0))
            (set! myudf/checkbox1 (cx-create-toggle-button myudf/buttonbox1 "Check Box 1"))
            (set! myudf/checkbox2 (cx-create-toggle-button myudf/buttonbox1 "Check Box 2"))

            (set! myudf/buttonbox2 (cx-create-button-box myudf/box1 "Radio Buttons" 'radio-mode #t 'row 1))
            (set! myudf/radiobutton1 (cx-create-toggle-button myudf/buttonbox2 "Radio Button 1"))
            (set! myudf/radiobutton2 (cx-create-toggle-button myudf/buttonbox2 "Radio Button 2"))

            (set! myudf/int (cx-create-integer-entry myudf/box1 "For Ints" 'row 2))
            (set! myudf/real (cx-create-real-entry myudf/box1 "For Reals" 'row 3))
            (set! myudf/string (cx-create-text-entry myudf/box1 "For Text" 'row 4))

            (set! myudf/buttonstr (cx-create-text-entry myudf/box2 "Button Test Text Box" 'row 0))
            (cx-create-button myudf/box2 "Test Button" 'activate-callback button-cb 'row 1)

            (set! myudf/list
                (cx-create-list myudf/box3 "List Options" 'visible-lines 5 'multiple-selections #t 'row 0))
            (cx-set-list-items myudf/list (map thread-name (sort-threads-by-name(get-face-threads))))

            (set! myudf/droplist
                (cx-create-drop-down-list myudf/box3 "Drop-Down List Options" 'multiple-selections #f 'row 1))
            (cx-set-list-items myudf/droplist (map thread-name (sort-threads-by-name(get-face-threads))))

          ) ;End Of Let Statement
        ) ;End Of If Statement

        ;Call To Open Dialog Box
        (cx-show-panel dialog-box)
      ) ;End Of Args Function
  ) ;End Of Let Statement
) ;End Of GUI-Dialog-Box Definition

(cx-add-menu "New Menu" #f)
(cx-add-hitem "New Menu" "New Submenu" #f #f #t #f)
;Menu Item Added To Above Created "New Menu->New Submenu" Submenu In Fluent
(cx-add-item "New Submenu" "MyUDF Dialog Box" #\U #f #t gui-dialog-box)
```

5.2. Task Page Example

This section contains the code for a task page that includes all of the features discussed in part two of this guide. This example can be used in conjunction with the UDF Example (p. 766) found later in this chapter in order to verify that the data is passed from the task page to the UDF using RP variables. This example contains comments in order to help readers understand the various parts of the example. All Scheme comment lines start with a semi-colon. This code automatically creates a menu called **New Menu 2**, with a submenu called **New Submenu 2**. The task page is automatically added as a menu item in **New Submenu 2**.

> **Note:**
>
> In order for the **List Options** and **Drop-Down List Options** fields to populate, you must first read a mesh into Fluent because it uses information from the mesh as the list items.

This example uses a couple of Fluent macros that are not seen previously in part two of this guide. The line `(map thread-name(get-face-threads))` in the `update-cb` function is used to populate the list and drop-down list fields using face threads from your current mesh when the task page is opened. As seen in chapter one, the Map (p. 724) statement applies a single function over a list of data. This makes `thread-name` the function, which simply gets the name of the thread name as text from the thread itself. The `get-face-threads` part of the statement is the list that the function is being applied to, meaning that this statement returns a list of face threads from the mesh on which the `thread-name` function can operate.

The other macro in this example that is not covered previously in part two of this guide is the `(%run-udf-apply)` macro. This macro is used because the corresponding UDF Example (p. 766) uses a `DEFINE_EXECUTE_FROM_GUI` statement. This allows you to run your UDF from the **OK** or **Apply** button of your GUI. The `(%run-udf-apply)` macro is used in the `apply-cb` function as a call to run the UDF. This macro has requires one integer parameter. This integer is used in the UDF to determine what the programmer asked it to do. This is seen in the example below. The **Test Button** has a different job then the **Apply** button, so it passes the integer 1 instead of 2. When the UDF runs, it will test the value of that integer and decide what to do depending on what the integer is. The `DEFINE_EX-ECUTE_FROM_GUI` statement is the only `DEFINE` statement that uses a Scheme macro in the GUI code to invoke the UDF. Other `DEFINE` macros allow the UDF to be run at various times during a simulation or, in the case of the `DEFINE_EXECUTE_ON_DEMAND` statement, whenever you want by right-clicking **User Defined Functions** under the **Parameters & Customization** branch of the tree and selecting **Execute on Demand....** For more information on the various `DEFINE` statements available for use in your UDF, see `DEFINE` Macros (p. 19).

```
;RP Variable Create Function
(define (make-new-rpvar name default type)
  (if (not (rp-var-object name))
      (rp-var-define name default type #f)))


;RP Variable Declarations
(make-new-rpvar 'myudf/real 0 'real)
(make-new-rpvar 'myudf/int 0 'int)
(make-new-rpvar 'myudf/checkbox1 #f 'boolean)
(make-new-rpvar 'myudf/checkbox2 #f 'boolean)
(make-new-rpvar 'myudf/radiobutton1 #f 'boolean)
(make-new-rpvar 'myudf/radiobutton2 #f 'boolean)
(make-new-rpvar 'myudf/string "" 'string)
(make-new-rpvar 'myudf/buttonstr "" 'string)
(make-new-rpvar 'myudf/list '() 'string-list)
(make-new-rpvar 'myudf/droplist '() 'string-list)


;Task Page Definition
(define gui-task-page
  ;Let Statement, Local Variable Declarations
  (let ((task-page #f)
   (table)
        (myudf/box1)
        (myudf/box2)
        (myudf/box3)
        (myudf/buttonbox1)
        (myudf/checkbox1)
        (myudf/checkbox2)
        (myudf/buttonbox2)
        (myudf/radiobutton1)
        (myudf/radiobutton2)
        (myudf/real)
        (myudf/int)
        (myudf/string)
        (myudf/buttonstr)
        (myudf/list)
        (myudf/droplist)
        )

        ;Update-CB Function, Invoked When Task Page Is Opened
        (define (update-cb . args)
          (cx-set-toggle-button myudf/checkbox1 (rpgetvar 'myudf/checkbox1))
          (cx-set-toggle-button myudf/checkbox2 (rpgetvar 'myudf/checkbox2))
          (cx-set-toggle-button myudf/radiobutton1 (rpgetvar 'myudf/radiobutton1))
          (cx-set-toggle-button myudf/radiobutton2 (rpgetvar 'myudf/radiobutton2))
          (cx-set-integer-entry myudf/int (rpgetvar 'myudf/int))
          (cx-set-real-entry myudf/real (rpgetvar 'myudf/real))
          (cx-set-text-entry myudf/string (rpgetvar 'myudf/string))
          (cx-set-text-entry myudf/buttonstr (rpgetvar 'myudf/buttonstr))
          (cx-set-list-items myudf/list (map thread-name(get-face-threads)))
          (cx-set-list-selections myudf/list (rpgetvar 'myudf/list))
          (cx-set-list-items myudf/droplist (map thread-name(get-face-threads)))
          (cx-set-list-selections myudf/droplist (rpgetvar 'myudf/droplist))
        )

        ;Apply-CB Function, Invoked When "Apply" Button Is Clicked
        (define (apply-cb . args)
          (rpsetvar 'myudf/checkbox1 (cx-show-toggle-button myudf/checkbox1))
          (rpsetvar 'myudf/checkbox2 (cx-show-toggle-button myudf/checkbox2))
          (rpsetvar 'myudf/radiobutton1 (cx-show-toggle-button myudf/radiobutton1))
          (rpsetvar 'myudf/radiobutton2 (cx-show-toggle-button myudf/radiobutton2))
          (rpsetvar 'myudf/int (cx-show-integer-entry myudf/int))
          (rpsetvar 'myudf/real (cx-show-real-entry myudf/real))
          (rpsetvar 'myudf/string (cx-show-text-entry myudf/string))
          (rpsetvar 'myudf/buttonstr (cx-show-text-entry myudf/buttonstr))
          (rpsetvar 'myudf/list (cx-show-list-selections myudf/list))
          (rpsetvar 'myudf/droplist (cx-show-list-selections myudf/droplist))
          (%run-udf-apply 2)
        )

        ;Button-CB Function, Invoked When "Test Button" Is Clicked
```

```
       (define (button-cb . args)
         (rpsetvar 'myudf/buttonstr (cx-show-text-entry myudf/buttonstr))
         (%run-udf-apply 1)
       )

       ;Args Function, Used For Interface Setup, Required For Apply-CB, Update-CB, and Button-CB Sections
       (lambda args
         (if (not task-page)
          (let ()
            (set! task-page (cx-create-taskpage "myUDF Task Page" apply-cb update-cb))
            (set! table (cx-create-table task-page "" 'border #f 'below 0 'right-of 0))

            (set! myudf/box1 (cx-create-table table "Simple Data Inputs" 'row 0))
            (set! myudf/box2 (cx-create-table table "Button Test" 'row 1))
            (set! myudf/box3 (cx-create-table table "List Inputs" 'row 2))

            (set! myudf/buttonbox1 (cx-create-button-box myudf/box1 "Check Boxes" 'radio-mode #f 'row 0))
            (set! myudf/checkbox1 (cx-create-toggle-button myudf/buttonbox1 "Check Box 1"))
            (set! myudf/checkbox2 (cx-create-toggle-button myudf/buttonbox1 "Check Box 2"))

            (set! myudf/buttonbox2 (cx-create-button-box myudf/box1 "Radio Buttons" 'radio-mode #t 'row 1))
            (set! myudf/radiobutton1 (cx-create-toggle-button myudf/buttonbox2 "Radio Button 1"))
            (set! myudf/radiobutton2 (cx-create-toggle-button myudf/buttonbox2 "Radio Button 2"))

            (set! myudf/int (cx-create-integer-entry myudf/box1 "For Ints" 'row 2))
            (set! myudf/real (cx-create-real-entry myudf/box1 "For Reals" 'row 3))
            (set! myudf/string (cx-create-text-entry myudf/box1 "For Text" 'row 4))

            (set! myudf/buttonstr (cx-create-text-entry myudf/box2 "Button Test Text Box" 'row 0))
            (cx-create-button myudf/box2 "Test Button" 'activate-callback button-cb 'row 1)

            (set! myudf/list (cx-create-list myudf/box3 "List Options" 'visible-lines 5 'multiple-selections #t 'ro
            (cx-set-list-items myudf/list (map thread-name (sort-threads-by-name(get-face-threads))))

            (set! myudf/droplist (cx-create-drop-down-list myudf/box3 "Drop-Down List Options" 'multiple-selections
            (cx-set-list-items myudf/droplist (map thread-name (sort-threads-by-name(get-face-threads))))
          ) ;End Of Let Statement
         ) ;End Of If Statement

         ;Call To Open Task Page
         (cx-show-taskpage task-page)
       ) ;End Of Args Function
  ) ;End Of Let Statement
) ;End Of GUI-Task-Page Definition

(cx-add-menu "New Menu 2" #f)
(cx-add-hitem "New Menu 2" "New Submenu 2" #f #f #t #f)
;Menu Item Added To Above Created "New Menu 2->New Submenu 2" Submenu In Fluent
(cx-add-item "New Submenu 2" "MyUDF Task Page" #\U #f #t gui-task-page)
```
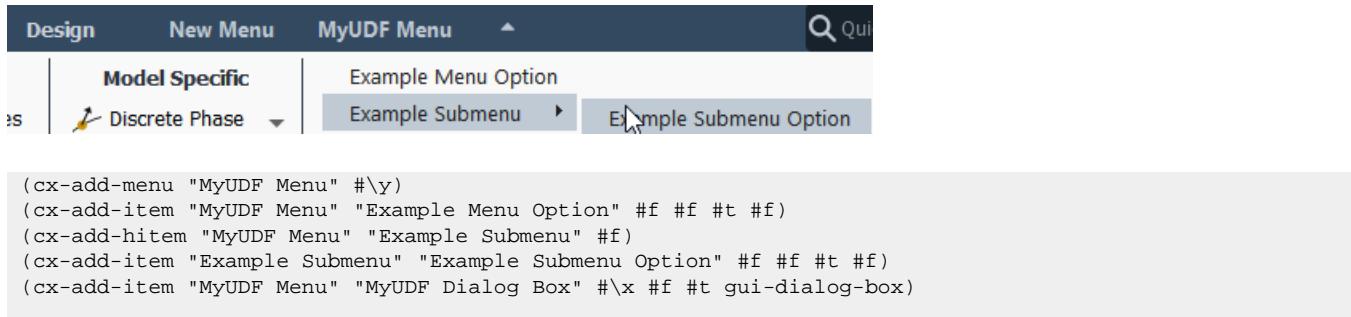
5.3. Example Menu Added to the Right of the Ribbon Tabs

This menu example covers each of the various menu macros covered in the Adding Menus to the Right of the Ribbon (p. 755) chapter. It is important to note that the **Example Menu Option** and **Example Submenu Option** menu items do not open any kind of interface. As you can see in the code, these are just menu items created to illustrate the process of adding menu items to menus and submenus. The **MyUDF Dialog Box** menu item is functional and will open the Dialog Box Example (p. 759) interface. This menu creation code provides examples of menus and menu items that use a mnemonic (MyUDF

Menu, MyUDF Dialog Box) and those that do not, using `#f` in the place of that argument (Example Menu Item, Example Submenu, Example Submenu Item).

Note:

In order for the **MyUDF Dialog Box** menu item to show up in the **MyUDF Menu**, the Dialog Box Example (p. 759) Scheme file must be read into Fluent before this menu creation example code. This is why **New Menu** also appears in the following graphic.

```
(cx-add-menu "MyUDF Menu" #\y)
(cx-add-item "MyUDF Menu" "Example Menu Option" #f #f #t #f)
(cx-add-hitem "MyUDF Menu" "Example Submenu" #f)
(cx-add-item "Example Submenu" "Example Submenu Option" #f #f #t #f)
(cx-add-item "MyUDF Menu" "MyUDF Dialog Box" #\x #f #t gui-dialog-box)
```

5.4. UDF Example

This UDF code is written as an example of how RP variables that were given values in a Scheme interface can then be read for use by a User-Defined Function. This UDF can be used in conjunction with the Dialog Box Example (p. 759) above to show how RP variables are created and set in a dialog box and then read by a UDF. This UDF uses a `DEFINE_EXECUTE_FROM_GUI` (p. 26) statement so that the UDF runs whenever the **OK** button is clicked in a corresponding dialog box. To learn more about the various ways of invoking a UDF, see `DEFINE` Macros (p. 19). For information on how to load your UDF into Fluent via compilation, see Compiling UDFs (p. 399).

```
//Required for all UDFs
#include "udf.h"

//Allows UDF to execute when "OK" button is clicked
DEFINE_EXECUTE_FROM_GUI(check, libudf, mode)
{
  //Variable Declarations
  int listNum;
  int dropDownListNum;
  int i;

  //If "Test Button" was clicked
  if(mode == 1){
    if(strlen(RP_Get_String("myudf/buttonstr")) != 0){
      Message("The test button string is %s\n", RP_Get_String("myudf/buttonstr"));
    }else{
      Message("The test button string is empty\n");
    }
    Message("\n");

  //If "OK" button was clicked
  }else if(mode == 2){

   //Check box RP variables are checked
    if(RP_Get_Boolean("myudf/checkbox1")){
      Message("Check box 1 is checked\n");
    }else{
      Message("Check box 1 is not checked\n");
    }
```

```
    if(RP_Get_Boolean("myudf/checkbox2")){
      Message("Check box 2 is checked\n");
    }else{
      Message("Check box 2 is not checked\n");
    }

  //Radio button RP variables are checked
    if(RP_Get_Boolean("myudf/radiobutton1")){
      Message("Radio button 1 is checked, therefore radio button 2 is not checked\n");
    }else if(RP_Get_Boolean("myudf/radiobutton2")){
      Message("Radio button 2 is check, therefore radio button 1 is not checked\n");
    }else{
      Message("Neither radio button is checked\n");
    }

  //Integer and real number RP variables are checked
   Message("The integer is '%d'\n", RP_Get_Integer("myudf/int"));
   Message("The real number is '%f'\n", RP_Get_Real("myudf/real"));

  //String RP variable checked
   if(strlen(RP_Get_String("myudf/string")) != 0){
     Message("The string is '%s'\n", RP_Get_String("myudf/string"));
   }else{
     Message("The string is empty\n");
   }

  //Test button string RP variable checked
   if(strlen(RP_Get_String("myudf/buttonstr")) != 0){
     Message("The test button string is '%s'\n", RP_Get_String("myudf/buttonstr"));
   }else{
     Message("The test button string is empty\n");
   }

  //List items RP variable checked
   if(listNum = RP_Get_List_Length("myudf/list")){
     for(i=0; i<listNum; i++){
      Message("List item number %d is '%s'\n", i+1, RP_Get_List_Ref_String("myudf/list", i));
      }
   }else{
     Message("The list is empty\n");
   }

  //Drop-down list RP variable checked
   if(dropDownListNum = RP_Get_List_Length("myudf/droplist")){
     Message("The drop-down list item is '%s'\n", RP_Get_List_Ref_String("myudf/droplist", 0));
   }else{
     Message("The drop-down list is empty");
   }
   Message("\n");

  //If neither "Test Button" or "Ok" button was clicked
  }else{
    Message("Error!\n");
  }
}
```

Appendix A. Avoiding Common Mistakes

This appendix is about some of the more common mistakes that can be made as you create your own interface in Fluent.

A.1. Keeping Track Of Parentheses

Since the Scheme programming language uses parentheses for all of its statements, it can sometimes be difficult to keep track of parentheses. This is especially true when you have some statements nested inside of others. When you have a significant amount of nested statements and open parentheses that haven't been closed yet, it is helpful to be meticulous with line indentation. If you indent a nested statement by a few spaces, it will be easier to recognize that the statement exists within a larger statement. By making such use of indentation, in addition to comments within your document, you can help yourself avoid headache down the line.

A.2. Knowing The Type Of Each Variable

As is mentioned in the description of the Set! (p. 723) statement, Scheme variables can change type depending on what is being bound to the variable. For example, you can create a variable as an integer, but then assign it a Boolean value like `#f` at a later time. Once the variable is set to `#f`, it ceases being an integer and becomes Boolean instead. Since Scheme offers a lot of flexibility in variable manipulation, it is important to double check your syntax each time you set a variable to a certain value. Small unnoticed mistakes, such as a period in the middle of a number, can change the type of the variable without you noticing. These kinds of problems will cause more issues down the line as it becomes harder to locate your mistake.

A.3. Overwriting Interface Elements

The following two examples will illustrate how failing to specify a row and column position for each item in a table on dialog box can result in some interface elements being overwritten.
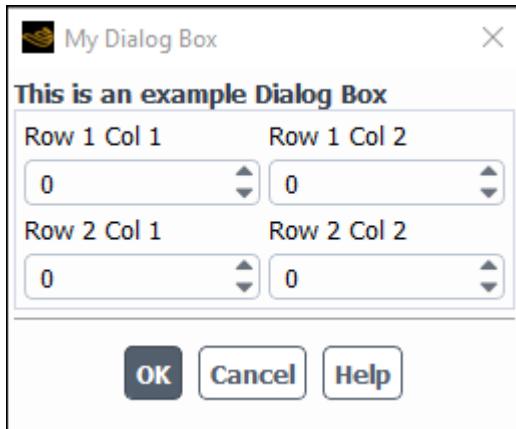
A.3.1. Example One

This example shows how a dialog box should look when all of the elements are placed correctly in a table using the `row` and `col` arguments. The four integer entry fields used in this example have both a row and column specified in their `create` statements. This results in a clean and well structured

dialog box. This dialog box does not do anything when the **OK** button is clicked because we have substituted Boolean values for the `apply-cb` and `update-cb` arguments, which would normally be function calls. For more information on the `apply-cb` and `update-cb` functions, see cx-create-panel (p. 729).


```
(define (apply-cb) #t)
(define update-cb #f)

(define table)
(define int1)
(define int2)
(define int3)
(define int4)

(define my-dialog-box (cx-create-panel "My Dialog Box" apply-cb update-cb))

(set! table (cx-create-table my-dialog-box "This is an example Dialog Box"))

(set! int1 (cx-create-integer-entry table "Row 1 Col 1" 'row 1 'col 1))
(set! int2 (cx-create-integer-entry table "Row 1 Col 2" 'row 1 'col 2))
(set! int3 (cx-create-integer-entry table "Row 2 Col 1" 'row 2 'col 1))
(set! int4 (cx-create-integer-entry table "Row 2 Col 2" 'row 2 'col 2))

(cx-show-panel my-dialog-box)
```

A.3.2. Example Two

This example shows what happens when you decide not to use the row and column arguments to position items in a table on your dialog box. There are actually four integer entry fields on this dialog box, but unfortunately the last integer entry field to be added to the table doesn't have a row and column position specified in its `create` statement. As a result, the final integer entry field will overwrite anything that is already in row 0 column 0. The lesson to be learned here is to specify a position for every item on you interface or else risk overwriting some elements with others. This dialog box does not do anything when the **OK** button is clicked because we have substituted Boolean values for the `apply-cb` and `update-cb` arguments, which would normally be function calls. For more information on the `apply-cb` and `update-cb` functions, see cx-create-panel (p. 729).

```
(define (apply-cb) #t)
(define update-cb #f)

(define table)
(define int1)
(define int2)
(define int3)
(define int4)

(define my-dialog-box (cx-create-panel "My Dialog Box" apply-cb update-cb))

(set! table (cx-create-table my-dialog-box "This is an example Dialog Box"))

(set! int1 (cx-create-integer-entry table "Row 1 Col 1" 'row 1 'col 1))
(set! int2 (cx-create-integer-entry table "Row 1 Col 2" 'row 1 'col 2))
(set! int3 (cx-create-integer-entry table "Row 2 Col 1" 'row 2 'col 1))
(set! int4 (cx-create-integer-entry table "No Row/Col Specified"))

(cx-show-panel my-dialog-box)
```

Appendix B. Reference Table For Fluent Macros

This appendix is a reference location for all of the Fluent-specific Scheme macros described in part two of this guide. Each macro is listed with a short description of its purpose and a link to the section of the guide that explains the macro in greater detail. The arguments of each macro are not listed in this section due to the need for further explanation of each of these arguments. Explanation of the arguments of each macro can be found in the corresponding sections that are linked. Macros are ordered by the order in which they appear in the guide.

Macro	Purpose	Link
(rp-var-define)	Creates a new RP variable.	Creating an RP Variable (p. 725)
(make-new-rpvar)	Though not built into Fluent by default, this macro is recommended for checking for the presence of RP variables before creation.	Creating an RP Variable (p. 725)
(rpsetvar)	Changes the value of an RP variable.	Changing an RP Variable (p. 726)
(%rpgetvar)	Obtains the value of an RP variable.	Accessing the Value of an RP Variable In Your GUI (p. 726)
(ti-menu-load-string)	Loads a string command into the Fluent TUI. Used in this context in the .fluent file.	The .fluent File (p. 727)
(cx-create-panel)	Creates a new dialog box.	cx-create-panel (p. 729)
(cx-show-panel)	Displays an already created dialog box.	cx-show-panel (p. 730)
(cx-create-table)	Creates a table for organization of interface elements on a dialog box.	Usage (p. 735)
(cx-create-integer-entry)	Creates a new integer entry field for a dialog box.	cx-create-integer-entry (p. 737)
(cx-set-integer-entry)	Sets the value of an integer entry field.	cx-set-integer-entry (p. 738)
(cx-show-integer-entry)	Obtains the value of an integer entry field.	cx-show-integer-entry (p. 738)
(cx-create-real-entry)	Creates a new real number entry field for a dialog box.	cx-create-real-entry (p. 740)
(cx-set-real-entry)	Sets the value of a real number entry field.	cx-set-real-entry (p. 740)
(cx-show-real-entry)	Obtains the value of a real number entry field.	cx-show-real-entry (p. 740)
(cx-create-text-entry)	Creates a text entry field for a dialog box.	cx-create-text-entry (p. 742)
(cx-set-text-entry)	Sets the text in a text entry field.	cx-set-text-entry (p. 742)
(cx-show-text-entry)	Obtains the text in a text entry field.	cx-show-text-entry (p. 742)
(cx-create-button-box)	Creates a button box for check boxes and radio buttons.	cx-create-button-box (p. 744)

Macro	Purpose	Link
(cx-create-toggle-button)	Creates a check box or option button for a dialog box depending on the radio-mode attribute of the button box it resides in.	cx-create-toggle-button (p. 744)
(cx-set-toggle-button)	Sets the state of the check box or option button to selected or cleared through the Boolean values true and false.	cx-set-toggle-button (p. 745)
(cx-show-toggle-button)	Obtains the state of a check box or option button.	cx-show-toggle-button (p. 745)
(cx-create-button)	Creates a button for a dialog box.	Usage (p. 748)
(cx-create-list)	Creates a list for a dialog box.	cx-create-list (p. 750)
(cx-create-drop-down-list)	Creates a drop-down-list for a dialog box.	cx-create-drop-down-list (p. 750)
(cx-set-list-items)	Populates a list or drop-down list with list items.	cx-set-list-items (p. 751)
(cx-set-list-selections)	Sets what item/items are selected in a list or drop-down list.	cx-set-list-selections (p. 751)
(cx-show-list-selections)	Obtains the selection/selections of a list or drop-down list.	cx-show-list-selections (p. 751)
(cx-add-menu)	Adds a new menu to the right of the Fluent ribbon tabs.	Usage (p. 755)
(cx-add-hitem)	Adds a new submenu to an already existing menu on the right of the Fluent ribbon tabs.	Usage (p. 756)
(cx-add-item)	Adds a new menu item to an already existing menu or submenu on the right of the Fluent ribbon tabs.	Usage (p. 758)
(map thread-name (get-face-threads))	Obtains all of the names of the face threads of the mesh currently loaded in Fluent.	Dialog Box Example (p. 759)
(%run-udf-apply)	Invokes a currently loaded UDF using the DEFINE_EXECUTE_FROM_GUI statement.	Dialog Box Example (p. 759)