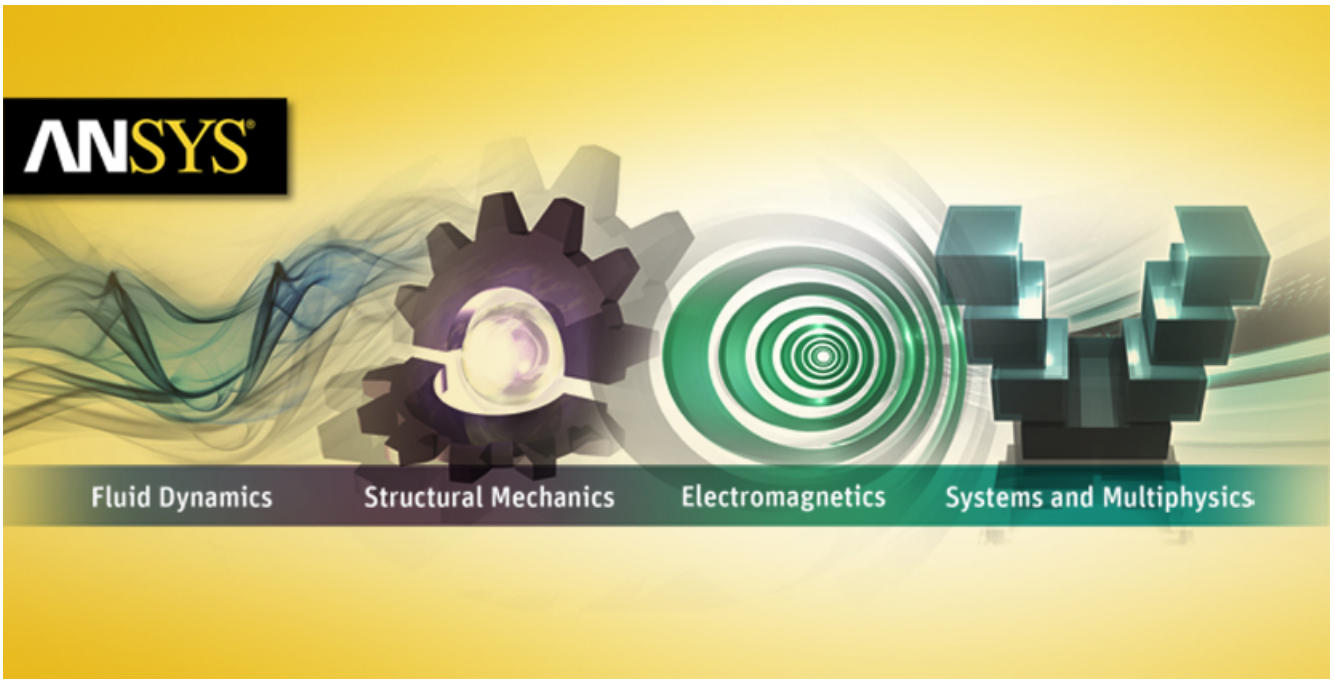


ANSYS CFX Reference Guide

ANSYS, Inc.
Southpointe
275 Technology Drive
Canonsburg, PA 15317
ansysinfo@ansys.com
http://www.ansys.com
(T) 724-746-3304
(F) 724-514-9494

Release 14.0
November 2011

Copyright and Trademark Information

Disclaimer Notice

U.S. Government Rights

Third-Party Software

Published in the U.S.A.

Table of Contents

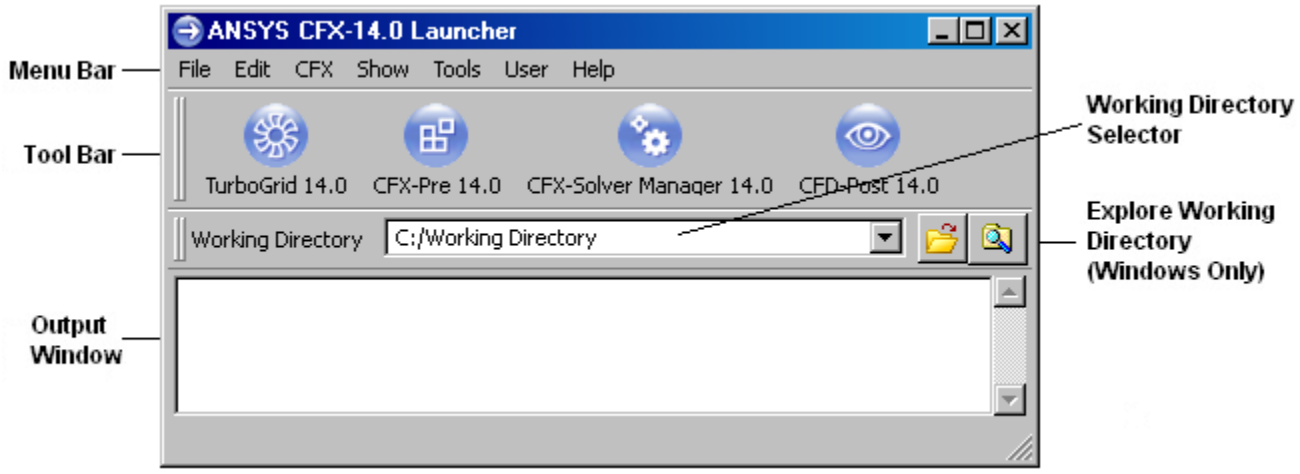
Chapter 1: ANSYS CFX Launcher

This chapter describes the ANSYS CFX Launcher in detail:

1.1. The ANSYS CFX Launcher Interface

The layout of the ANSYS CFX Launcher is shown below:

Figure 1.1 ANSYS CFX Launcher



The launcher consists of a menu bar, a tool bar for launching applications, a working directory selector, and an output window where messages are displayed. On Windows platforms, an icon to start Windows Explorer in the working directory appears next to the directory selector.

1.1.1. Menu Bar

The ANSYS CFX Launcher menus provide the following capabilities:

1.1.1.1. File Menu

Saves the contents of the text output window and to close the ANSYS CFX Launcher.

1.1.1.1.1. Save As

Saves the contents of the output window to a file.

1.1.1.1.2. Quit

Shuts down the ANSYS CFX Launcher. Any programs already launched will continue to run.

1.1.1.2. Edit Menu

Clears the text output window, finds text in the text output window and sets options for the ANSYS CFX Launcher.

1.1.1.2.1. Clear

Clears the output window.

1.1.1.2.2. Find

Displays a dialog box where you can search the text in the output window.

1.1.1.2.3. Options

Presents the **Options** dialog box, which enables you to change the appearance of the ANSYS CFX Launcher.

1.1.1.2.3.1. User Interface Style

You can choose any one of several user interface styles; each style is available on all platforms. For example, choosing `Windows` will change the look and feel of the user interface to resemble that of a Windows application. You can select from Windows, Motif, CDE (Solaris), Plastique, and Cleanlooks styles. Once you have selected a style, click **Apply** to test.

1.1.1.2.3.2. Application Font and Text Window Font

The button to the right of **Application Font** sets the font used anywhere outside the text output window. The button to the right of **Text Window Font** applies only to the text output window. Clicking either of these buttons opens the **Select Font** dialog box.

1.1.1.3. CFX Menu

Enables you to launch CFX-Pre, CFX-Solver Manager, CFD-Post, and, if they are installed, other CFX products (such as ANSYS TurboGrid).

1.1.1.3.1. CFX-Pre

Runs CFX-Pre, with the working directory as specified in *Working Directory Selector* (p. 4).

1.1.1.3.2. CFX-Solver Manager

Runs CFX-Solver Manager, with the working directory as specified in *Working Directory Selector* (p. 4).

1.1.1.3.3. CFD-Post

Runs CFD-Post, in the current working directory as specified in *Working Directory Selector* (p. 4).

1.1.1.3.4. Other CFX Applications

The ANSYS CFX Launcher also searches for other CFX applications (for example, ANSYS TurboGrid) and provides a menu entry to launch the application. If an application is not found, you can add it; for details, see *Customizing the ANSYS CFX Launcher* (p. 5).

1.1.1.4. Show Menu

Enables you to show system, installation, and other information.

1.1.1.4.1. Show Installation

Displays information about the version of CFX that you are running.

1.1.1.4.2. Show All

Displays all of the available information, including information about your system, installation and variables.

1.1.1.4.3. Show System

Displays information about the CFX installation and the system on which it is being run.

1.1.1.4.4. Show Variables

Displays the values of all the environment variables that are used in CFX.

1.1.1.4.5. Show Patches

Displays the output from the command `cfx5info -patches`. This provides information on patches that have been installed after the initial installation of CFX.

1.1.1.5. Tools Menu

Enables you to access license-management tools and a command line for running other CFX utilities.

1.1.1.5.1. ANSYS Client Licensing Utility

Enables you to configure connections to ANSYS License Managers.

1.1.1.5.2. Command Line

Starts a command window from which you can run any of the CFX commands via the command line interface. The command line will be set up to run the correct version of CFX and the commands will be run in the current working directory.

If you do not use the **Tools** > **Command Line** command to open a command window, then you will have to either type the full path of the executable in each command, or explicitly set your operating system path to include the `<CFXROOT>/bin` directory.

You may want to start components of CFX from the command line rather than by clicking the appropriate button on the ANSYS CFX Launcher for the following reasons:

- CFX contains some utilities (for example, a parameter editor) that can be run only from the command line.
- You may want to specify certain command line arguments when starting up a component so that it starts up in a particular configuration.
- If you are having problems with a component, you may be able to get a more detailed error message by starting the component from the command line than you would get if you started the component

from the launcher. If you start a component from the command line, any error messages produced are written to the command line window.

1.1.1.5.3. Configure User Startup Files (Linux only)

Information about creating startup files can be found in the installation documentation.

1.1.1.5.4. Edit File

Opens a browser to edit the text file of your choice in a platform-native text editor. Which text editor is called is controlled by the settings in `<CFXROOT>/etc/launcher/shared.ccl`.

1.1.1.5.5. Edit Site-wide Configuration File

Opens the site-wide configuration file in a text editor. Which text editor is called is controlled by the settings in `<CFXROOT>/etc/launcher/CFX5.ccl`.

1.1.1.6. User Menu

The **User** menu is provided as an example. You can add your own applications to this menu, or create new menus; for details, see *Customizing the ANSYS CFX Launcher* (p. 5).

1.1.1.7. Help Menu

The **Help** menu enables you to view tutorials, user guides, and reference manuals online. For related information, see Accessing Help.

1.1.2. Tool Bar

The toolbar contains shortcuts to the main components of CFX, for example CFX-Pre, CFX-Solver Manager and CFD-Post. Pressing any of the buttons will start up the component in the specified working directory. The equivalent menu entries for launching the components also show a keyboard shortcut that can be used to launch the component.

1.1.3. Working Directory Selector

While running CFX, all the files that are created will be stored in the working directory. To change the working directory, you can do any of the following:

- Type the directory name into the box and press **Enter**.
- Click the down-arrow icon ( ) next to the directory name. This displays a list of recently used directories.
- Click *Browse*  to browse to the directory that you want.

1.1.4. Output Window

The output window is used to display information from commands in the **Show** menu. You can right-click in the output window to show a shortcut menu with the following options:

- **Find**: Displays a dialog box where you can enter text to search for in the output.
- **Select All**: Selects all the text.
- **Copy Selection**: Copies the selected text.

- **Save As**: Saves the output to a file.
- **Clear**: Clears the output window.

1.2. Customizing the ANSYS CFX Launcher

Many parts of the ANSYS CFX Launcher are driven by CCL commands contained in configuration files. Some parts of the launcher are not editable (such as the **File**, **Edit** and **Help** menus), but others parts allow you to edit existing actions and create new ones (for example, launching your own application from the **User** menu). The following sections outline the steps required to configure the launcher. The configuration files are located in the `<CFXROOT>/etc/launcher/` directory (where `<CFXROOT>` is the path to your installation of CFX). You can open these files in any text editor, but you should not edit any of the configuration files provided by CFX, other than the `User.ccl` configuration file.

1.2.1. CCL Structure

The configuration files contain CCL objects that control the appearance and behavior of menus and buttons that appear in the ANSYS CFX Launcher. There are three types of CCL objects: `GROUP`, `APPLICATION` and `DIVIDER` objects. The fact that there are multiple configuration files is not important; applications in one file can refer to groups in other files.

An example of how to add a menu item for the Windows calculator to the launcher is given in *Example: Adding the Windows Calculator* (p. 8).

1.2.1.1. GROUP

`GROUP` objects represent menus and toolbar groups in the ANSYS CFX Launcher. Each new `GROUP` creates a new menu and toolbar. Nothing will appear in the menu or toolbar until you add `APPLICATION` or `DIVIDER` objects to the group. An example of a `GROUP` object is given below:

```
GROUP: CFX
  Position = 200
  Menu Name = &CFX
  Show In Toolbar = Yes
  Show In Menu = Yes
  Enabled = Yes
END
```

- The group name is set after the colon. In this case, it is "CFX". This is the name that `APPLICATION` and `DIVIDER` objects will refer to when you want to add them to this group. This name should be different to all other `GROUP` objects.
- `Position` refers to the position of the menu relative to others. The value should be an integer between 1 and 1000. Groups with a higher `Position` value, relative to other groups, will have their menu appear further to the right in the menu bar. Referring to *Figure 1.1* (p. 1), CFX has a lower position value than the ANSYS group. The **File** and **Edit** menus are always the first two menus and the **Help** menu is always the last menu.
- The title of the menu is set under `Menu Name` (this menu has the title **CFX**). The optional ampersand is placed before the letter that you want to have act as a menu accelerator (for example, **Alt**-**C** displays the **CFX** menu). You must be careful not to use an existing menu accelerator.
- The creation of the menu or toolbar can be toggled by setting the `Show in Menu` and `Show in Toolbar` options to `Yes` or `No` respectively. For example, you may want to create a menu item but not an associated toolbar icon.
- `Enabled` sets whether the menu/toolbar is available for selection or is disabled. Set the option to `No` to disable it.

1.2.1.2. APPLICATION

APPLICATION objects create entries in the menus and toolbars that will launch an application or run a process. Two examples are given below with an explanation for each parameter. The first example creates a menu entry in the **Tools** menu that opens a command line window. The second example creates a menu entry and toolbar button to start CFX-Solver Manager.

```
APPLICATION: Command Line 1
  Position = 300
  Group = Tools
  Tool Tip = Start a window in which CFX commands can be run
  Menu Item Name = Command Line
  Command = <windir>\system32\cmd.exe
  Arguments = /c start
  Show In Toolbar = No
  Show In Menu = Yes
  Enabled = Yes
  OS List = winnt
END
APPLICATION: CFXSM
  Position = 300
  Group = CFX
  Tool Tip = Launches ANSYS CFX-Solver Manager
  Menu Item Name = CFX-&Solver Manager
  Command = cfx5solve
  Show In Toolbar = Yes
  Show In Menu = Yes
  Enabled = Yes
  Toolbar Name = ANSYS CFX-Solver Manager
  Icon = LaunchSolveIcon.xpm
  Shortcut = CTRL+S
END
```

- The application name is set after the colon, in the first example it is "Command Line 1". This name should be different from all other APPLICATION objects.
- Position: sets the relative position of the menu entry. The value should be an integer between 1 and 1000. The higher the value, relative to other applications that have the same group, the further down the menu or the further to the right in a toolbar the entry will appear. If you do not specify a position, the object assumes a high position value (so it will appear at the bottom of a menu or at the right of a group of buttons).
- Group: sets the GROUP object to which this application belongs. The value must correspond to the name that appears after "GROUP:" in an existing GROUP object. The menu and/or toolbar entry will not be created if you do not specify a valid group name. The GROUP object does not have to be in the same configuration file.
- Tool Tip: displays a message when the mouse pointer is held over a toolbar button. In the "Command Line 1" example above, the Tool Tip entry is not used because a toolbar button is not created. This parameter is optional.
- Menu Item Name: sets the name of the entry that will appear in the menu. If you do not specify a name, the name is set to the name of the APPLICATION: object. The optional ampersand is placed before the letter that you want to have act as a menu accelerator (for example, **Alt**-**C** then **S** will start CFX-Solver Manager. **Alt**-**C** selects the **CFX** menu and **S** selects the entry from the menu). You must be careful not to use an existing menu accelerator.
- Command: contains the command to run the application. The path can be absolute (that is, use a forward slash to begin the path on Linux, or a drive letter on Windows). If an absolute path is not specified, a relative path from <CFXROOT>/bin/ is assumed. If no command is specified, the menu item/toolbar button will not appear in the ANSYS CFX Launcher. The path and command are checked when the launcher is started. If the path or command does not exist, the menu item/toolbar button will not appear

in the launcher. You may find it useful to include environment variables in a command path; for details, see *Including Environment Variables* (p. 7).

- `Arguments`: specifies any arguments that need to be passed to the application. The arguments are appended to the value you entered for `Command`. You do not need to include this parameter as there are no arguments to pass. You may find it useful to include environment variables in the arguments; for details, see *Including Environment Variables* (p. 7).

  Distinct arguments are space-separated. If you need to pass an argument that contains spaces (for example, a Windows file path) you should include that argument in double quotes, for example:

  ```
  Arguments = "C:\Documents and Settings\User" arg2 arg3
  ```

- `Show In Toolbar`: determines if a toolbar button is created for the application. This optional parameter has a default value of **Yes**.
- `Show In Menu`: determines if a menu entry is created for the application. This optional parameter has a default value of `Yes`.
- `Enabled`: controls the menu entry and toolbar button. Set this parameter to `No` to disable the application. This optional parameter has a default value of `Yes`.
- `OS List` is an optional parameter that enables you to set which operating system the application is suitable for. If `OS List` is not supplied, the launcher will attempt to create the menu item and toolbar button on all platforms.

  For example, the command to open a command line window varies depending on the operating system. In the 'Command Line 1' example above, the application only applies to Windows platforms. To complete the OS coverage, the launcher configuration files contain more 'Command Line' applications that apply to different operating systems.

  `OS List` can contain `winnt` (Windows, including Windows XP) and `linux-ia64` (64-bit Linux).
- `Toolbar Name`: sets the name that appears on the toolbar button. This parameter is optional (because you may want to show only an icon).
- `Icon`: specifies the icon to use on the toolbar button and in the menu item. The path can be absolute (that is, use a forward slash to begin the path on Linux, or a drive letter on Windows). If an absolute path is not specified, a relative path from `<CFXROOT>/etc/icons` is assumed. The following file formats are supported for icon image files: Portable Network Graphics (`png`), Pixel Maps (`ppm`, `xpm`) and Bitmaps (`bmp`). Other icons used in the launcher are 32 pixels wide and 30 pixels high. This parameter is optional. If it is not included, an icon will not appear.
- `Shortcut`: specifies the keyboard shortcut that can be pressed to launch the application. You must be careful not to use a keyboard shortcut that is used by any other `APPLICATION` object.

1.2.1.2.1. Including Environment Variables

In can be useful to use environment variables in the values for some parameters. You can specify an environment variable value in any parameter by including its name between the < > symbols. In the 'Command Line 1' example above, `<windir>` is used in the `Command` parameter so that the command would work on different versions of Windows. `<windir>` is replaced with the value held by the `windir` environment variable. The `Command` and `Argument` parameters are the only parameters that are likely to benefit from using environment variables. Environment variables included in the `Arguments` parameter are expanded before they are passed to the application.

1.2.1.3. DIVIDER

DIVIDER objects create a divider in a menu and/or toolbar (see the **Tools** menu for an example). An example of the CCL for DIVIDER objects is shown below.

```
DIVIDER: Tools Divider 1
  Position = 250
  Group = Tools
  OS List = winnt, linux-ia64
END
```

The Position, Group and OS List parameters are the same as those used in APPLICATION objects. For details, see *APPLICATION* (p. 6).

1.2.2. Example: Adding the Windows Calculator

The following CCL is the minimum required to add the Windows calculator to the ANSYS CFX Launcher:

```
GROUP: Windows Apps
  Menu Name = Windows
END
APPLICATION: Calc
  Group = Windows Apps
  Command = <windir>\system32\calc.exe
  Toolbar Name = Calc
END
```

Although the parameter Toolbar Name is not strictly required, you would end up with a blank toolbar button if it were not set.

Chapter 2: Volume Mesh Import API

The Mesh Import Application Programming Interface (API) enables you to build a customized executable that reads a 3-dimensional mesh from a 3rd-party mesh file into CFX-Pre and to extend the number of file formats that CFX-Pre can understand and read beyond those supplied as part of the standard installation.

The communication between the executable and CFX-Pre is via a communications channel that is controlled by use of routines in the API provided.

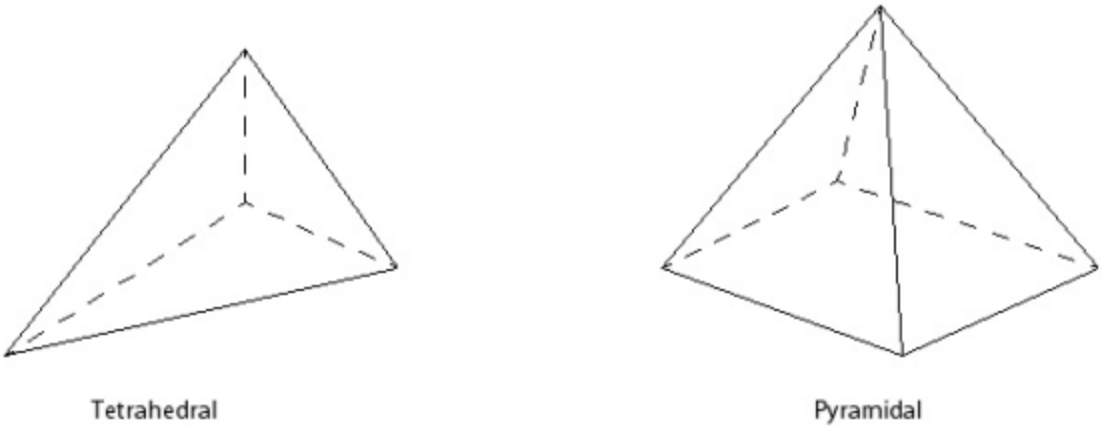
For details on using the Volume Mesh Import API, see User Import.

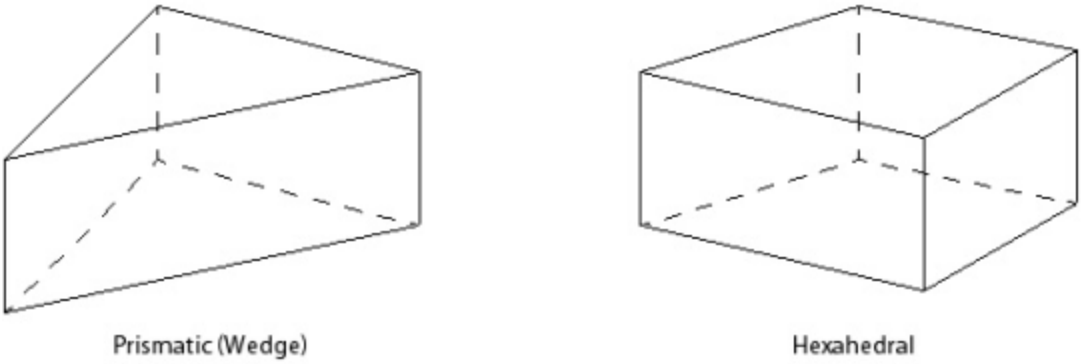
This chapter describes:

2.1. Valid Mesh Elements in CFX

The CFX-Solver technology works with unstructured meshes. This does not prohibit the use of structured meshes. However a structured mesh will always be dealt with internally as an unstructured mesh.

The CFX-Solver can solve flows in any mesh comprising one or more of the following element types:





You must write the program using the API to translate the mesh read from the 3rd-party file into a format that can be processed by CFX-Pre.

2.2. Creating a Custom Mesh Import Executable for CFX-Pre

You can create your own customized program using the 'C' programming language or Fortran programming language. A number of API functions are provided in a library supplied with the ANSYS CFX installation. For details, see *Details of the Mesh Import API* (p. 12).

The installation contains a C source code example file that can be used as the basis of your custom executable. This file, `ImportTemplate.c`, is provided in `<CFXROOT>/examples/`, and is listed in: *An Example of a Customized C Program for Importing Meshes into CFX-Pre* (p. 25).

The basic structure of a program written to import a 3rd-party mesh into CFX-Pre is as follows:

1. Inclusion of the `cfxImport.h` header file (for C programs and not Fortran programs).
2. Initialization for import with the `cfxImportInit` routine.
3. Definition of node data with `cfxImportNode`.
4. Definition of element data with `cfxImportElement`.
5. Optionally, definitions of 2D and 3D regions with either `cfxImportRegion` or the following three functions: `cfxImportBegReg`, `cfxImportAddReg`, `cfxImportEndReg`
6. Data transfer with `cfxImportDone`.

The header files associated with the API are located in <CFXROOT>/include/. If you do not use the header file cfxImport.h, the functionality of the routines contained within the API may not follow defined behavior.

After writing the program, you will need to compile the source code. For details, see *Compiling Code with the Mesh Import API* (p. 11).

You will also need to link your routine with the API routine libraries. For details, see *Linking Code with the Mesh Import API* (p. 11).

After a customized executable has been produced, it can be run in CFX-Pre. For details, see User Import.

2.2.1. Compiling Code with the Mesh Import API

Compilation of a customized executable must be performed using an appropriate compiler and compiler flags.

The customized executable must also be linked with the provided mesh import API library and the provided I/O library as detailed in *Linking Code with the Mesh Import API* (p. 11).

Note

Windows users should note that custom mesh import programs must be compiled as multi-threaded applications.

2.2.1.1. Compiler Flags

No compiler flags are necessary for successful compilation on linux-ia64, but use the icc compiler.

2.2.2. Linking Code with the Mesh Import API

In order to build a customized import utility routine, it must be linked with several libraries. With the exception of bufferoverflowu.lib, these libraries are located in <CFXROOT>/lib/<*os*>/:

- libmeshimport.lib (on Windows), or libmeshimport.a (on Linux)
- libratlas_api.lib (on Windows), or libratlas_api.a (on Linux)
- libratlas.lib (on Windows), or libratlas.a (on Linux)
- libpgtapi.lib (on Windows), or libpgtapi.a (on Linux)
- libunits.lib (on Windows), or libunits.a (on Linux)
- libcclapilt.lib (on Windows), or libcclapilt.a (on Linux)
- libio.lib (on Windows), or libio.a (on Linux)
- bufferoverflowu.lib (on Windows 64-bit)

2.2.2.1. Linking a Customized C Mesh Import Executable on a UNIX Platform

On most UNIX systems you should be able to build the executable with the command:

```
cc myimport.c -I<CFXROOT>/include/ -o myimport -L<CFXROOT>/lib/<os> \
  -lmeshimport -lratlas_api -lratlas -lpgtapi -lunits -lcclapilt -lio \
  -lm -lc
```

Here, `-lmeshimport`, `-lratlas_api`, `-lratlas`, `-lpgtapi`, `-lunits`, `-lcclapillt`, and `-lio` indicate the libraries mentioned above, while `-lm` and `-lc` are system libraries.

In this example, your own import program is named `myimport.c` and the executable file will be called `myimport`. You should ensure that the libraries to which you are linking (which are in the path given after `-L`) appear on the command line after the source file (or object file if you are just linking to an existing object).

2.2.2.2. Linking a Customized Fortran Mesh Import Executable on a UNIX Platform

The following is an example of how to build the executable on Linux, when the source code for the executable is written in Fortran:

```
g77 myImport.F -L<install_dir>/lib/linux -lmeshimport -lratlas_api -lratlas \
  -lpgtapi -lunits -lcclapilt -lio -o myImport.exe
```

2.2.2.3. Linking a Customized Mesh Import Executable on a 32-bit Windows Platform

You can build the executables on 32-bit Windows systems that have Microsoft Visual C++ 2008 Express Edition. An example command line follows:

```
cl /MD /I "C:\Program Files\Ansys Inc\v140\CFX\include" ImportTemplate.c
/link /libpath:"C:\Program Files\Ansys Inc\v140\CFX\lib\winnt" lib-
cclapilt.lib libio.lib libmeshimport.lib libunits.lib libpgtapi.lib lib-
ratlas_api.lib libratlas.lib
```

2.2.2.4. Linking a Customized Mesh Import Executable on a 64-bit Windows Platform

You can build the executables on 64-bit Windows systems that have the compiler from VS2008. An example command line follows:

```
cl /MD /I "C:\Program Files\Ansys Inc\v140\CFX\include" ImportTemplate.c
/link /libpath:"C:\Program Files\Ansys Inc\v140\CFX\lib\winnt-amd64"
libcclapilt.lib libio.lib libmeshimport.lib libunits.lib libpgtapi.lib
libratlas_api.lib libratlas.lib bufferoverflowu.lib
```

2.3. Details of the Mesh Import API

This section contains information about the functions that are used to write a customized import executable in the Mesh Import API.

Before trying to use any of the routines listed in this section, it is highly recommended that you read *Creating a Custom Mesh Import Executable for CFX-Pre* (p. 10).

This section contains details of:

Note

In past releases of ANSYS CFX the API has defined IDs of nodes and elements as integers (int). This release now uses a datatype ID_t to represent these quantities. This type is currently defined as an unsigned integer (unsigned int). This allows a greater number of nodes and elements to be imported than in the past.

2.3.1. Defined Constants

The following are defined in the header file `cfxImport.h`, which should be included in the import program.

2.3.1.1. Element Types

There are currently 4 types of elements, which are identified by the number of nodes: Tetrahedrons (4 nodes), pyramids (5 nodes), wedges or prisms (6 nodes), and hexahedrons (8 nodes). The element types may be identified by the defined constants:

```
#define cfxELEM_TET  4
#define cfxELEM_PYR  5
#define cfxELEM_WDG  6
#define cfxELEM_HEX  8
```

The element node ordering and local face numbering follow Patran Neutral file conventions for element descriptions.

2.3.1.2. Region Types

Regions may be defined in terms of nodes, faces or elements, based on the type argument to the `cfxImportBegReg` or `cfxImportRegion` routines. The three types are defined by the defined constants:

```
#define cfxImpREG_NODES  1
#define cfxImpREG_FACES  2
#define cfxImpREG_ELEMS  3
```

Node and Face regions define 2D regions of the imported mesh. Element regions define 3D regions of the imported mesh.

It is best to use face regions to define 2D regions of the mesh and element regions to define 3D regions of the mesh.

Node regions will be automatically transformed into a face region by the import process. This transformation requires the node IDs specified to define vertices of valid element faces. If no element faces can be constructed from the defined node region the node region will be deleted.

Note

Due to the limited topological information recoverable from a set of nodes it is not advisable to define 2D regions internal to a 3D region using nodes. In this case it is advisable to use Face regions.

Node regions are specified by a list of node IDs.

Face regions are defined by a list of face IDs. These face IDs are a combination of an element ID and a local face number in the element.

2.3.2. Initialization Routines

The following routines check and initialize the Import API. With the exception of `cfxImportStatus` the first call to the Import API must be either `cfxImportInit` for communication with CFX, or `cfxImportTest` for testing the import routine in stand-alone mode.

2.3.2.1. cfxImportStatus

```
int cfxImportStatus ()
```

Returns 0 if descriptor is not opened and -1 if not opened for writing. In the normal case, 1 is returned if opened for writing to CFX, and 2 if opened for writing to a file.

2.3.2.2. cfxImportInit

```
void cfxImportInit ()
```

Performs initialization to begin communicating with CFX. This routine should be called early on in the import program to let CFX know that data is to be sent. If not called within 60 seconds, CFX will terminate the import process. If called and there is no connection with CFX, then the routine `cfxImportTest("/dev/null")` (UNIX) or `cfxImportTest("null")` (Windows) will be called. This routine will be automatically called by most of the API routines if not already called.

There is no return value for this routine. In the case of an error, `cfxImportFatal` will be called.

2.3.2.3. cfxImportTest

```
int cfxImportTest (filename)
    char *filename;
```

This routine allows testing of import program in isolation from CFX by writing data to a file *`filename`* instead of attempting to write it to the CFX communication channel.

The routine will return the file descriptor of the output file or will terminate with a call to `cfxImportFatal` on error.

2.3.3. Termination Routines

With the exception of `cfxImportTotals` the last call to the Import API must always be `cfxImportDone`. This function performs the final processing of the import data, and then transfers the data to CFX.

2.3.3.1. cfxImportDone

```
long cfxImportDone ()
```

Indicate to the import API that all mesh data has been given and the API should now send the data to CFX. Except for `cfxImportTotals`, this should be last call made to the API. Returns the total number of bytes transferred to CFX by the import program.

2.3.3.2. cfxImportTotals

```
long cfxImportTotals (counts)
size_t counts[cfxImpCNT_SIZE];
```

Get the total number of nodes, elements, regions and other useful information given to the mesh import API by the program. This information is returned in the array `counts`, which should be of size at least `cfxImpCNT_SIZE` (currently defined as 9). The values returned in `counts` may be indexed by the `enum` list in `cfxImport.h`, which is:

```
counts[cfxImpCNT_NODE]    = number of nodes
counts[cfxImpCNT_ELEMENT] = number of elements
counts[cfxImpCNT_REGION]  = number of regions
counts[cfxImpCNT_UNUSED]  = number of unused nodes
counts[cfxImpCNT_DUP]     = number of duplicate nodes
counts[cfxImpCNT_TET]     = number of tetrahedral elements
counts[cfxImpCNT_PYR]     = number of pyramid elements
counts[cfxImpCNT_WDG]     = number of wedge elements
counts[cfxImpCNT_HEX]     = number of hexahedral elements
```

The return value for the function is the total number of bytes of data sent to CFX or written to the test file given when `cfxImportTest` was called.

2.3.4. Error Handling Routines

The first error handling routine allows the programmer to define an error callback function that is called when a fatal error is generated by the API or explicitly by the programmers code.

The second routine performs a method for clean termination of the program, shutting down the program and communication with ANSYS CFX.

2.3.4.1. cfxImportError

```
void cfxImportError (callback)
void (*callback)(char *errmsg);
```

Define a user routine to be called before terminating due to a fatal error. `callback` is the application-supplied function to be called in the case of an error. The callback routine takes a single argument, `errmsg`, which will be passed by `cfxImportFatal` and should be processed by the callback function as a brief message describing the error that has occurred. If this function is not called or callback is not specified, then the normal termination behavior of the mesh import API will be that the any fatal errors will write the error message to `stderr` as well as being sent to CFX.

2.3.4.2. cfxImportFatal

```
void cfxImportFatal (errmsg)
char *errmsg;
```

Terminate with an error message, `errmsg`. This routine will send the message to CFX, shut down the communication channel or test file and call the user callback function (if specified by a call to cfxImportError).

There is no return from this call. The import program will terminate immediately after clean up tasks have been performed.

2.3.5. Node Routines

These routines define the 3D coordinates of points in space(nodes) that will be used to define elements or 2D regions that are to be imported to CFX. Each node has a unique identifier called a node ID.

2.3.5.1. cfxImportNode

```
ID_t cfxImportNode (nodeid, x, y, z)
ID_t nodeid;
double x, y, z;
```

Define a node in the import API to be subsequently imported into CFX. The unique identifier of the node is given by `nodeid`, and the coordinates of the node by x, y, and z.

Returns 0 if `nodeid` is invalid (less than 1), or `nodeid` is successfully defined. If a node with the same identity has already been defined, the coordinate values will alter to the supplied values.

2.3.5.2. cfxImportGetNode

```
ID_t cfxImportGetNode (nodeid, x, y, z)
ID_t nodeid;
double *x, *y, *z;
```

Get the coordinates for the node identified by `nodeid` and return the values in `x`, `y`, and `z`. Returns 0 if the node has not been defined or the node ID for the node.

2.3.5.3. cfxImportNodeList

```
ID_t * cfxImportNodeList ()
```

Returns an array of all node identifiers currently defined or `NULL` if no nodes have been defined. The first entry in the array is the number of nodes currently defined.

The memory for the array returned is allocated using `malloc` by the routine, consequently it should be destroyed when no longer required by calling `free`.

2.3.6. Element Routines

The following routines define the topology of elements (using node IDs) that are to be imported to CFX. Also included here are routines that get the local face number and vertices of an element.

2.3.6.1. cfxImportElement

```
ID_t cfxImportElement (elemid, elemtype, nodelist)
ID_t elemid, *nodelist; int elemtype;
```

Define a new element to be imported to CFX. The unique identifier of the element is given by `elemid`, the element type by `elemtype` and the list of vertices by `nodelist`. If an element with the same ID has already been defined, it will be replaced by the new element being defined.

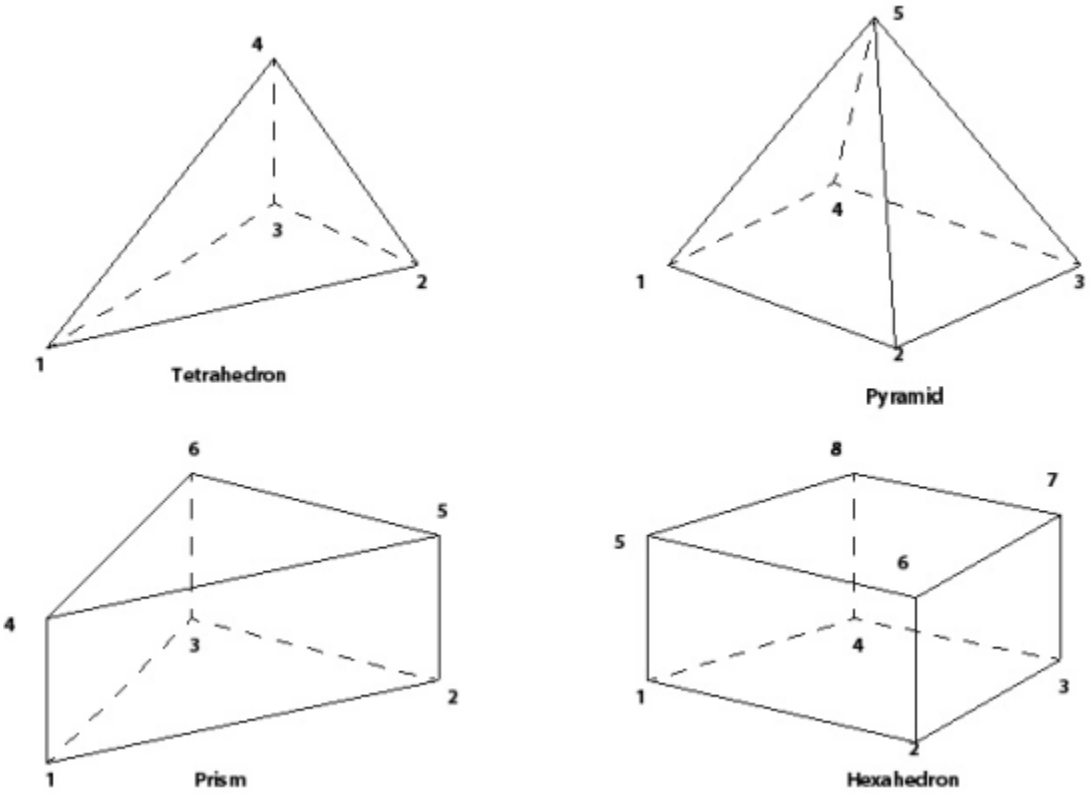
Only volume elements are currently supported by CFX; these may be tetrahedrons (4 vertices), pyramids (5 vertices), prisms (6 vertices) or hexahedrons (8 vertices). `elemtype` is the number of vertices for the element.

The following defines are included in the header file, `cfxImport.h` for convenience:

```
#define cfxELEM_TET  4  /* tet element (4 nodes)      */
#define cfxELEM_PYR  5  /* pyramid element (5 nodes)  */
#define cfxELEM_WDG  6  /* wedge element (6 nodes)    */
#define cfxELEM_HEX  8  /* hex element (8 nodes)      */
```

The list of vertices in `nodelist` refers to IDs of nodes that on termination of the import program by a call to `cfxImportDone` must have been defined by calls to `cfxImportNode`. If this is not the case a fatal error will be reported and the API will terminate.

The vertex ordering for the elements follows Patran Neutral File element conventions, and is shown in the following figure.



Note

The vertex ordering for the export API is different. For details, see *cfxExportElementList* (p. 55).

Returns 0 in the case of an `elemid` is invalid (less than 1) or an unsupported value is given by `elemtype`, or `elemid` if the element is successfully defined. If the element already exists the vertices of the element will be redefined.

2.3.6.2. cfxImportGetElement

```
ID_t cfxImportGetElement (elemid, nodelist)
ID_t elemid, nodelist[];
```

Get the node IDs for corresponding to the vertices of element identified by `elemid` and store in the array nodelist. This array needs to be at least as large the number of vertices for the element (a size of 8 will handle all possible element types).

Returns 0 if the element is not defined, or the element type (number of vertices). The node IDs will be ordered in the order expected by `cfxImportElement` if the program was to redefine the element.

2.3.6.3. cfxImportElementList

```
ID_t * cfxImportElementList ()
```

Returns an array of all the currently defined element IDs or NULL if no elements have been defined. The first entry in the array is the number of elements.

The memory for the array returned is allocated using `malloc` by the routine, consequently it should be destroyed when no longer required by calling `free`.

2.3.6.4. cfxImportGetFace

```
ID_t cfxImportGetFace (elemid, facenum, nodelist)
ID_t elemid, nodelist[]; int facenum;
```

Gets the node IDs for the local `facenum`'th face of the element identified by `elemid`.

The node IDs are returned in `nodelist`, which should be of at least of size 4. The nodes correspond to the vertices of the face and are ordered counter-clockwise such that the normal for the face points away from the element. The face numbers and associated node indices are modeled after Patran Neutral File elements, and are tabulated here:

Element Type	Face	Nodes			
tetrahedron	1	1	3	2	
	2	1	2	4	
	3	2	3	4	
	4	1	4	3	
pyramid	1	1	4	3	2
	2	1	2	5	
	3	2	3	5	
	4	3	4	5	
	5	1	5	4	
prism	1	1	3	2	
	2	4	5	6	
	3	1	2	5	4
	4	1	4	6	3
	5	2	3	6	5
hexahedron	1	1	2	6	5
	2	3	4	8	7
	3	1	4	3	2
	4	2	3	7	6
	5	5	6	7	8
	6	1	5	8	4

Note

The face numbers and associated node indices are different when exporting elements. For details, see *cfxExportFaceNodes* (p. 58).

Returns -1 if the element has not been defined, 0 if the face number is out of range, or the number of nodes for the face (3 or 4):

2.3.6.5. cfxImportFindFace

```
ID_t cfxImportFindFace (elemid, nnodes, nodeid)
ID_t elemid, nodeid[]; int nnodes;
```

Gets the local face number in element identified by `elemid` that contains all the nodes supplied by the calling routine in `nodeid`. `nnodes` is the number of nodes for the face (3 or 4).

Returns -1 if the element is not found or `nodeid` is not supplied or `nnodes` is greater than 4 or less than 3. Returns 0 if there is no match, or the local face number (1 to 6) of the element.

2.3.7. Primitive Region Routines

The following routines allow for the specification of 2D regions as a group of nodes or faces, or a 3D region as a group of elements. In the case of nodes and faces, only those that define faces of valid imported elements will be imported; others are ignored by CFX.

2.3.7.1. cfxImportBegReg

```
int cfxImportBegReg (regname, regtype)
char *regname;
int regtype;
```

Initialize for the specification of a region. If a region is currently being defined, `cfxImportEndReg` will be called.

The name of the region is given by `regname`. If the region name is `NULL`, the name `Unnamed Region 2D` or `Unnamed Region 3D`, with a sequential integer appended, will be used. If a region named `regname` has already been defined, then additional objects will be added to the previous region.

The type of region is given by regtype, which should be one of `cfxImpREG_NODES`, `cfxImpREG_FACES` or `cfxImpREG_ELEMS` depending on whether the region is to be defined by nodes, faces or elements, respectively. It is not currently possible to mix types in a region; doing so will cause the import API to terminate with an error message.

Returns the number of objects (node, faces or elements) currently in the region.

2.3.7.2. cfxImportAddReg

```
int cfxImportAddReg (numobjs, objlist)
int numobjs, *objlist;
```

Add IDs of objects being defined to the current region.

A region must be currently defined or reactivated by `cfxImportBegReg` or an error will occur, and the API will terminate.

The number of objects to add is given by `numobjs` and the IDs of the objects are supplied in `objlist`.

The objects are interpreted as node IDs, face IDs, or element IDs, depending on the type of the region indicated when `cfxImportBegReg` was called.

On calling `cfxImportDone`, any node IDs , face IDs or element IDs specified in the object list must have been defined by the appropriate routine or they will be removed from the region.

Returns the total number of objects in the current region after the object IDs have been added.

2.3.7.3. cfxImportEndReg

```
int cfxImportEndReg ()
```

End the specification of the current region.

Returns the number of objects (nodes, faces or elements) in the region.

2.3.7.4. cfxImportRegion

```
int cfxImportRegion (regname, regtype, numobjs, objlist)
char *regname;
int regtype, numobjs, *objlist;
```

Import a region named `regname` of type `regtype`. The number of objects to add to the region is given by `numobjs`, and the list of object IDs by `objlist`. This routine combines calls to `cfxImport-BegReg`, `cfxImportAddReg` and `cfxImportEndReg`.

Returns the total number of objects in the region on termination of the routine.

2.3.7.5. cfxImportRegionList

```
char ** cfxImportRegionList ()
```

Return a NULL terminated list of currently defined region names.

The memory for the array and each character string in the array returned is allocated using `malloc` by the routine, consequently each array member and the array itself should be destroyed when no longer required by calling `free`.

2.3.7.6. cfxImportGetRegion

```
int * cfxImportGetRegion (regname)
char *regname;
```

Returns a list of objects in the region named `regname`, or `NULL` if the region does not exist. The first entry in the returned list is the region type and the second entry is the number of object IDs.

The memory for the array is allocated using `malloc` by the routine, consequently the array itself should be destroyed when no longer required by calling `free`.

2.3.8. Composite Regions Routines

The following routines allow composite regions to be defined in terms of primitive regions or other composite regions.

2.3.8.1. cfxImportBegCompRegion

```
cfxImportBegCompReg()
char *regionName;
```

Begin defining a composite region with the name `regionName`,

Returns -1 if a primitive region `regionName` is already defined or memory couldn't be allocated, or 0 if successfully created.

2.3.8.2. cfxImportAddCompRegComponents

```
int cfxImportAddCompRegComponents(componentCount,components)
int componentCount;
char **components;
```

Add a set of component region names specified in `components` to the composite region currently being defined. `componentCount` specified how many components are specified in the `components` array,

Returns -1 if a composite region isn't being defined or insufficient memory is available to add the components of the composite region, or 0 if the components were successfully added.

2.3.8.3. cfxImportEndCompReg

```
int cfxImportEndCompReg()
```

Finish defining the current composite region.

Returns -1 if a composite region isn't currently being defined or 0 otherwise.

2.3.8.4. cfxImportCompositeRegion

```
int cfxImportCompositeRegion(regionName, componentCount, components)
char *regionName, **components;
int componentCount;
```

Define a composite region named `regionName` with `componentCount` components supplied in character array `components`.

Returns 0 if successful or -1 if an error occurred preventing the composite region being defined.

2.3.9. Explicit Node Pairing

The following routine provides a method for explicitly marking two nodes as being identical (or in the same position in space).

2.3.9.1. cfxImportMap

```
ID_t cfxImportMap (nodeid, mapid)
ID_t nodeid, mapid;
```

Explicitly map the node identified by `nodeid` to the node identified by `mapid`.

On calling `cfxImportDone` the Mesh Import API will update regions and elements referencing the mapped node to the node it is mapped to. This therefore reduces the total node count imported to CFX and eliminates the duplicate nodes.

Duplicate nodes may also be removed by CFX if the appropriate options are selected in the CFX interface and an appropriate tolerance set. For details, see Importing Meshes in the *CFX-Pre User's Guide*.

2.3.10. Fortran Interface

The following routines are callable from Fortran, and interface with the corresponding C routine. There are currently no return values.

2.3.10.1. cfxinit

```
call cfxinit
```

Interface to `cfxImportInit`. Initializes for import.

2.3.10.2. cfxtest

```
CHARACTER*n filename
call cfxtest(filename)
```

Interface to `cfxImportTest`. *`filename`* is a CHARACTER*n value that gives the name of the file to dump the output to.

2.3.10.3. cfxunit

```
CHARACTER*n units
call cfxunit(units)
```

Interface to `cfxImportUnits`. Specify the units the mesh is specified in.

2.3.10.4. cfxwarn

```
CHARACTER*n mesg
call cfxwarn(mesg)
```

Interface to `cfxImportWarning`. Emit a warning message `mesg`.

2.3.10.5. cfxfatl

```
CHARACTER*n mesg
call cfxfatl(mesg)
```

Interface to `cfxImportFatal`. Emit a warning message `mesg` and terminate the program cleanly.

2.3.10.6. cfxdone

```
call cfxdone
```

Interface to `cfxImportDone`. Terminates the program and transfers the data to CFX-Pre.

2.3.10.7. cfxnode

```
INTEGER idnode
DOUBLE PRECISION x,y,z
call cfxnode(idnode,x,y,z)
```

Interface to `cfxImportNode`. Imports a node with the specified coordinates. `idnode` is an INTEGER value for the node ID, and x, y, and z are the DOUBLE PRECISION coordinates of the node.

2.3.10.8. cfxnodg

```
INTEGER idnode
DOUBLE PRECISION x,y,z
call cfxnodg(idnode,x,y,z)
```

Interface to `cfxImportGetNode`. Queries the current coordinates or a node referenced by `idnode`. `idnode` is an INTEGER value for the node ID, and x, y, and z are the DOUBLE PRECISION coordinates of the node.

2.3.10.9. cfxnods

```
INTEGER ids(*)
call cfxnods(ids)
```

Interface to `cfxImportNodeList`. Retrieves the list of all valid node IDs having been imported into the API. `ids` is an INTEGER array that must be at least as large as the number of nodes currently imported.

2.3.10.10. cfxelem

```
INTEGER idelem,itelem,nodes(*)
call cfxelem(idelem,itelem,nodes)
```

Interface to `cfxImportElement`. `idelem` is element ID, and `itelem` is the element type (number of nodes - 4,5,6, or 8). Both are of type INTEGER. `nodes` is an array of INTEGER node IDs dimensioned of size at least `itelem`.

2.3.10.11. cfxeleg

```
INTEGER idelem,itelem,nodes(*)
call cfxeleg(idelem,itelem,nodes)
```

Interface to `cfxImportGetElement`. Queries the current node ids that define the vertices of the element referenced by the id `idelem`. `idelem` is element ID, and `itelem` is the element type (number of nodes - 4, 5, 6, or 8). Both are of type INTEGER. `nodes` is an array of INTEGER values that will contain the node IDs on successful return. It should be dimensioned of size at least `itelem`.

2.3.10.12. cfxeles

```
INTEGER ids(*)
call cfxeles(ids)
```

Interface to `cfxImportElemList`. Retrieves the list of all valid element IDs having been imported into the API. `ids` is an INTEGER array that must be at least as large as the number of elements currently imported.

2.3.10.13. cfxfacd

```
INTEGER eleid, elefc, id
call cfxfacd(eleid, elefc, id)
```

Interface to `cfxImportFaceID`. Defines a face id (`id`) in terms of an element ID (`eleid`) and local face (`elefc`) of that element.

2.3.10.14. cfxface

```
INTEGER eleid, elefc, vtx(*)
INTEGER cfxface(eleid, elefc, vtx)
```

Interface to `cfxImportGetFace`. Returns the node IDs of the vertices defining a face located by the element ID (`eleid`) and local face (`elefc`) of that element.

2.3.10.15. cfxffac

```
INTEGER eleid, nvtx, vtx(*), elefc
call cfxffac(eleid, nvtx, vtx, elefc)
```

Interface to `cfxImportFindFace`. Returns the local face (`elefc`) of an element (`eleid`) that is defined by the vertices (`vtx`).

2.3.10.16. cfxregn

```
CHARACTER*n regname
INTEGER type,nobjs,objs(*)
call cfxregn(regname,type,nobjs,objs)
```

Interface to `cfxImportRegion`. Regname is a CHARACTER*n string defining the region name, type is an INTEGER value specifying the type of region, either 1 for nodes, 2 for faces, or 3 for elements. `nobjs` is an INTEGER value that gives the number of objects in the region, and objs is an INTEGER array of object IDs dimensioned at least size `nobjs`.

2.3.10.17. cfxregb

```
CHARACTER*n regname
INTEGER type
call cfxregb(regname,type)
```

Interface to `cfxImportBegReg`. Start defining a new region or make an existing region of the same name the current one if it already exists and is of the same type. `regname` is a CHARACTER*n string defining the region name, type is an INTEGER value specifying the type of region, either 1 for nodes, 2 for faces, or 3 for elements.

2.3.10.18. cfxrega

```
INTEGER nobjs,objs(*)
call cfxrega(nobjs,objs)
```

Interface to `cfxImportAddReg`. Add the objects (`objs`) to the current region. `nobjs` is an INTEGER value that gives the number of objects to add to the region, and `objs` is an INTEGER array of object IDs dimensioned at least size `nobjs`.

2.3.10.19. cfxrege

```
call cfxrege()
```

Interface to `cfxImportEndReg`. Finish defining the current region (after the call there will be no current region).

2.3.10.20. cfxregs

```
CHARACTER*n regname
INTEGER numobj
call cfxregs(regname,numobj)
```

Query how many objects (returned in `numobj`) are referenced by the region `regname`. `regname` is a CHARACTER*n string specifying the region name.

2.3.10.21. cfxregg

```
CHARACTER*n regname
INTEGER type, obj(*)
call cfxregg(regname, type, objs)
```

Get the type (`type`) and object IDs (`objs`) referenced by the region `regname`. `regname` is a CHARACTER*n string specifying the region name. `type` is INTEGER and `objs` is an INTEGER array at least of the size returned by `cfxregs`.

2.3.10.22. cfxcmpb

```
CHARACTER*n regname
call cfxcmpb(regname)
```

Interface to `cfxImportBegCompReg`. Start defining a new composite region or make an existing composite region of the same name as the current one if it already exists. `regname` is a CHARACTER*n string defining the region name.

2.3.10.23. cfxcmpa

```
INTEGER nregs
CHARACTER*(n) regs
call cfxcmpa(nregs,regs)
```

Interface to `cfxImportAddCompReg`. Add the region names (`regs`) to the current composite region being defined. `nregs` is an INTEGER value that gives the number of regions to add to the region, and `regs` is a CHARACTER*(*) array of region names dimensioned at least size `nregs`.

2.3.10.24. cfxcmpe

```
call cfxcmpe()
```

Interface to `cfxImportEndCompReg`. Finish defining the current composite region (after the call there will be no current composite region).

2.3.11. Unsupported Routines Previously Available in the API

In ANSYS CFX 14.0 certain functionality available in previous releases is no longer supported. These routines have been removed because they are directly implemented in CFX.

The following is a list of routines removed from the mesh import API:

```
cfxImportFixElements
cfxImportTolerance
cfxImportGetTol
cfxImportSetCheck
cfxImportRange
cfxImportCheck
cfxtol
cfxset
cfxchk
```

2.4. An Example of a Customized C Program for Importing Meshes into CFX-Pre

This example, `ImportTemplate.c`, can be found in `<CFXROOT>/examples`.

```
/*
 * ImportTemplate.c - Patran Neutral File Import
 * reads packets 1 (nodes), 2 (elements) and 21 (named groups)
 * and optionally packet 6 (loads), and sends data to TfC
 */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include <ctype.h>
```

```
#include <math.h>
#include <sys/types.h>
#include <sys/stat.h>
#include "cfxImport.h"
#include "getargs.h"
static char options[] = "vlF:";
static char *usgmsg[] = {
    "usage  : ImportTemplate.c [options] Patran_file",
    "options:",
    "   -v      = verbose output",
    "   -l      = process packet 6 - distributed loads",
    NULL
};
/*---------- print_error ------------------------------------------
 * print error message and line number
 *-----------------------------------------------------------------*/
static int lineno = 0;
static void print_error (
#ifdef PROTOTYPE
    char *errmsg)
#else
    errmsg)
char *errmsg;
#endif
{
    fprintf (stderr, "%s on line %d\n", errmsg, lineno);
}
/*---------- add_face ---------------------------------------------
 * add an element face to the region list
 *-----------------------------------------------------------------*/
static void add_face (
#ifdef PROTOTYPE
    int elemid, char *data)
#else
    elemid, data)
int elemid;
char *data;
#endif
{
    int n, nnodes, nodes[8];
    ID_t nodeid[8];
    char errmsg[81];
    /* check for node flags set */
    for (nnodes = 0, n = 0; n < 8; n++) {
        if ('1' == data[n])
            nodes[nnodes++] = n;
    }
    /* if node flags set, use the node values */
    if (nnodes) {
        ID_t elemnodes[8];
        int elemtype = cfxImportGetElement (elemid, elemnodes);
        if (!elemtype) {
            sprintf (errmsg,
                "element %d not found for packet 6\n", elemid);
            cfxImportFatal (errmsg);
        }
        for (n = 0; n < nnodes; n++) {
            if (nodes[n] >= elemtype) {
                sprintf (errmsg,
                    "invalid node flags for element %d\n", elemid);
                cfxImportFatal (errmsg);
            }
            nodeid[n] = elemnodes[nodes[n]];
        }
    }
    /* else get nodes from face number */
    else {
        int faceid = atoi (&data[8]);
        nnodes = cfxImportGetFace (elemid, faceid, nodeid);
        if (nnodes < 0) {
            sprintf (errmsg,
                "element %d not found for packet 6\n", elemid);
```

```
            cfxImportFatal (errmsg);
        }
        if (0 == nnodes) {
            sprintf (errmsg,
                "invalid face number for element %d\n", elemid);
            cfxImportFatal (errmsg);
        }
    }
    cfxImportAddReg (nnodes, nodeid);
}
/*========== main ================================================*/
#define getline()   \
    {if (NULL == fgets (buffer, sizeof(buffer), fp)) \
        cfxImportFatal ("premature EOF"); \
    lineno++;}
void main (argc, argv)
int argc;
char *argv[];
{
    int n, packet, nlines;
    int nnodes;
    int  elemid;
    ID_t nodeid[8];
    int lastid = -1, loadid;
    int verbose = 0, do_loads = 0;
    double xyz[3];
    char *p, buffer[256];
    char *testfile = NULL;
    FILE *fp;
    struct stat st;
    if (argc < 2)
        usage (usgmsg, NULL);
    while ((n = getargs (argc, argv, options)) > 0) {
        switch (n) {
            case 'v':
                verbose = 7;
                break;
            case 'l':
                do_loads = 1;
                break;
            case 'F':
                testfile = argarg;
                break;
        }
    }
    if (argind >= argc)
        usage (usgmsg, "filename not specified\n");
    if (stat (argv[argind], &st)) {
        fprintf (stderr, "can't stat <%s>\n", argv[argind]);
        exit (1);
    }
    if (S_IFREG != (st.st_mode & S_IFMT)) {
        fprintf (stderr, "<%s> is not a regular file\n", argv[argind]);
        exit (1);
    }
    if (NULL == testfile)
        cfxImportInit ();
    else {
        if (verbose)
            printf ("writing output to <%s>\n", testfile);
        cfxImportTest (testfile);
    }
    if (NULL == (fp = fopen (argv[argind], "r"))) {
        sprintf (buffer, "can't open <%s> for reading", argv[argind]);
        cfxImportFatal (buffer);
    }
    if (verbose) {
        printf ("reading Patran Neutral file from <%s>\n", argv[argind]);
        fflush (stdout);
    }
    cfxImportError (print_error);
    getline ();
```

```
/* header - packet 25 */
if ((packet = atoi (buffer)) == 25) {
    getline ();
    if (verbose)
        fputs (buffer, stdout);
    getline ();
    packet = atoi (buffer);
}
/* summary - packet 26 */
if (packet == 26) {
    getline ();
    getline ();
    packet = atoi (buffer);
}
/* get remaining packets */
while (packet != 99) {
    /* node */
    if (packet == 1) {
        if (0 != (verbose & 4)) {
            printf ("reading packet 01 (nodes)...\n");
            fflush (stdout);
            verbose &= 3;
        }
        *nodeid = atoi (&buffer[2]);
        getline ();
        p = buffer + 48;
        for (n = 2; n >= 0; n--) {
            *p = 0;
            p -= 16;
            xyz[n] = atof (p);
        }
        getline ();
        cfxImportNode (*nodeid, xyz[0], xyz[1], xyz[2]);
    }
    /* element */
    else if (packet == 2) {
        if (0 != (verbose & 2)) {
            printf ("reading packet 02 (elements)...\n");
            fflush (stdout);
            verbose &= 1;
        }
        elemid = atoi (&buffer[2]);
        n = atoi (&buffer[10]);
        nlines = atoi (&buffer[18]);
        if (n == 5 || n == 7 || n == 8) {
            cfxImpElemType_t typ;
            getline ();
            nnodes = n == 8 ? n : n-1;
            if (nnodes == 4) {
              typ = cfxELEM_TET;
            } else if (nnodes == 5) {
              typ = cfxELEM_PYR;
            } else if (nnodes == 6) {
              typ = cfxELEM_WDG;
            } else if (nnodes == 8) {
              typ = cfxELEM_HEX;
            } else {
              cfxImportFatal("invalid number of nodes for element.");
  }
            lineno++;
            for (n = 0; n < nnodes; n++) {
      int tmp;
                if (1 != fscanf (fp, "%d", &tmp) || tmp < 1) {
                    cfxImportFatal ("missing or invalid node ID");
      } else {
        nodeid[n] = (ID_t)tmp;
      }
            }
            while (getc (fp) != '\n')
                ;
            nlines -= 2;
            cfxImportElement ((ID_t)elemid, typ, nodeid);
```

```
    }
    while (nlines-- > 0)
        getline ();
}
/* distributed loads */
else if (packet == 6 && do_loads) {
    elemid = atoi (&buffer[2]);
    loadid = atoi (&buffer[10]);
    nlines = atoi (&buffer[18]);
    if (loadid != lastid) {
        sprintf (buffer, "PatranLoad%d", loadid);
        if (verbose) {
            printf ("reading packet 06 (loads) as region <%s>...\n",
                buffer);
            fflush (stdout);
        }
        cfxImportBegReg (buffer, cfxImpREG_NODES);
        lastid = loadid;
    }
    getline ();
    /* add if element load flag is set */
    if ('1' == buffer[0])
        add_face (elemid, &buffer[9]);
    while (--nlines > 0)
        getline ();
}
/* named component */
else if (packet == 21) {
    int cnt, type, id;
    elemid = atoi (&buffer[2]);
    nnodes = atoi (&buffer[10]) / 2;
    getline ();

    /* strip leading and trailing spaces */
    buffer[sizeof(buffer)-1] = 0;
    p = buffer + strlen (buffer);
    while (--p >= buffer && isspace(*p))
        ;
    *++p = 0;
    for (p = buffer; *p && isspace(*p); p++)
        ;
    if (verbose) {
        printf ("reading packet 21 (group) as region <%s>...\n", p);
        fflush (stdout);
    }
    cfxImportBegReg (p, cfxImpREG_NODES);
    /* currently only handle type 5 (nodes) in groups */
    for (n = 0, cnt = 0; n < nnodes; n++) {
        if (0 == (n % 5))
            lineno++;
        fscanf (fp, "%d%d", &type, &id);
        if (5 == type) {
            nodeid[cnt++] = id;
            if (8 == cnt) {
                cfxImportAddReg (8, nodeid);
                cnt = 0;
            }
        }
    }
    while (getc (fp) != '\n')
        ;
    if (cnt)
        cfxImportAddReg (cnt, nodeid);
    cfxImportEndReg ();
}
/* all others */
else {
    nlines = atoi (&buffer[18]);
    while (nlines--)
        getline ();
}
if (NULL == fgets (buffer, sizeof(buffer), fp))
```

```
            break;
        lineno++;
        packet = atoi (buffer);
    }
    fclose (fp);
    cfxImportError (NULL);
    /* finish up and send the data */
    if (verbose) {
        printf ("transferring data...\n");
        fflush (stdout);
    }
    cfxImportDone ();
    /* print summary */
    if (verbose) {
        size_t stats[cfxImpCNT_SIZE];
        long bytes;
        static char *statname[] = {
            "imported nodes    ",
            "imported elements ",
            "imported regions  ",
            "unreferenced nodes",
            "duplicate nodes   ",
            "tet elements      ",
            "pyramid elements  ",
            "wedge elements    ",
            "hex elements      ",
            "total bytes sent  "
        };
        bytes = cfxImportTotals (stats);
        putchar ('\n');
        for (n = 0; n < 9; n++)
            printf ("%s = %ld\n", statname[n], stats[n]);
        printf ("%s = %ld\n", statname[9], bytes);
    }
    exit (0);
}
```

2.5. Import Programs

The following sections detail the standard import programs currently available within CFX-Pre and their command line equivalents.

Information about importing meshes from the CFX-Pre interface is given in Importing Meshes in the *CFX-Pre User's Guide*.

If you want to use command line options that cannot be specified through the CFX-Pre User Interface, then you may want to run these programs as user-defined mesh import programs. User Import details how to run a mesh import program.

The executables are located in `<CFXROOT>/bin/<os>`.

2.5.1. ANSYS

Imports an ANSYS file. The external import routine is `ImportANSYS`. Available options are:

`-v` Verbose output. Echo additional data to `stdout` during the import.

`-E` Import Elements of the same type as regions.

`-A` Import ANSA parts as regions.

`-S` Display a list of all supported element types.

2.5.2. CFX Def/Res

Imports the mesh from a CFX-Solver input or results file. The external import routine is `ImportDef`. Available options are:

`-v` Verbose output. Echo additional data to `stdout` during the import.

`-I` Read mesh from the initial timestep in the file.

`-L` Read mesh from the last timestep in the file.

`-T<timestep>` Read mesh from the timestep specified (Transient files)

2.5.3. CFX-4

Imports a CFX-4 grid file. The external import routine is `ImportCFX4`.

Available options are:

`-v` Verbose output. Echo additional data to `stdout` during the import.

`-C` Read coordinates as being in cylindrical coordinates.

`-i` Included interfaces in regions.

`-3` Include USER3D and POROUS regions as 3D regions.

`-c` Import blocked-off conducting solid regions as 3D regions.

`-l` Include blocked-off solid regions as 3D regions.

`-X` Import axisymmetric problem with default values in geometry file.

`-a <nk>` Override the number of planes created in the k direction by `nk` (for example, split theta with `nk` planes) for axisymmetric import.

`-A <theta>` Create a total sector of *theta* degrees for axisymmetric import.

`-S` Rename multiple symmetry planes with the same name to conform to CFX-Solver requirements (that is, must lie in a plane).

2.5.4. CFX-5.1

Imports a CFX-5.1 results file. The external import routine is `ImportCFX5`.

Available options are:

`-v` Verbose output. Echo additional data to `stdout` during the import.

`-f` Input file is formatted.

`-u` Input file is unformatted (Fortran).

`-M <machine type>` Set the machine type in the case of a binary or unformatted file so that data conversion may be done if needed. The default file format is 32-bit IEEE (Iris, Sun, HP, IBM). The currently recognized machine types are:

- IEEE - generic 32-bit IEEE machine.
- BSIEEE - generic 32-bit byteswapped IEEE machine.
- IBM - IBM 32-bit IEEE.
- IRIS - Iris 32-bit IEEE.
- HP - HP 32-bit IEEE.
- SUN - Sun 32-bit IEEE.
- ALPHA - Compaq Tru64 UNIX Alpha 64-bit byte-swapped IEEE.
- DOS - DOS 16-bit byte-swapped IEEE.
- Compaq Tru64 UNIX - Compaq Tru64 UNIX 32-bit byte-swapped IEEE.
- CRAY - Cray 64-bit format.
- CONVEX - native Convex floating point format.
- Windows - 32-bit Windows.

The argument machine type is case insensitive, and only the first 2 characters are needed (any others are ignored).

`-M <machine type>` Set the machine type in the case of a binary or unformatted file so that data conversion may be done if needed. The default file format is 32-bit IEEE (Iris, Sun, HP, IBM). The currently recognized machine types are:

- IEEE - generic 32-bit IEEE machine.
- BSIEEE - generic 32-bit byteswapped IEEE machine.
- IBM - IBM 32-bit IEEE.
- IRIS - Iris 32-bit IEEE.
- HP - HP 32-bit IEEE.
- SUN - Sun 32-bit IEEE.
- ALPHA - Compaq Tru64 UNIX Alpha 64-bit byte-swapped IEEE.
- DOS - DOS 16-bit byte-swapped IEEE.

- Compaq Tru64 UNIX - Compaq Tru64 UNIX 32-bit byte-swapped IEEE.
- CRAY - Cray 64-bit format.
- CONVEX - native Convex floating point format.
- Windows - 32-bit Windows.

The argument machine type is case-insensitive, and only the first two characters are needed (any others are ignored).

2.5.5. CFX-TfC

Imports a CFX-TfC 1.3 mesh file. The external import routine is `ImportGEM`.

Available options are:

`-v` Verbose output. Echo additional data to `stdout` during the import.

`-f` Input file is formatted.

`-u` Input file is unformatted (Fortran).

`-r` Read regions from a `BFI` file.

`-b` <file> Use file as a BFI file name instead of default name.

`-M` <*machine type*> Set the machine type in the case of a binary or unformatted file so that data conversion may be done if needed. The default file format is 32-bit IEEE (Iris, Sun, HP, IBM). The currently recognized machine types are:

- IEEE - generic 32-bit IEEE machine.
- BSIEEE - generic 32-bit byteswapped IEEE machine.
- IBM - IBM 32-bit IEEE.
- IRIS - Iris 32-bit IEEE.
- HP - HP 32-bit IEEE.
- SUN - Sun 32-bit IEEE.
- ALPHA - Compaq Tru64 UNIX Alpha 64-bit byte-swapped IEEE.
- DOS - DOS 16-bit byte-swapped IEEE.
- Compaq Tru64 UNIX - Compaq Tru64 UNIX 32-bit byte-swapped IEEE.
- CRAY - Cray 64-bit format.
- CONVEX - native Convex floating point format.
- Windows - 32-bit Windows.

The argument machine type is case insensitive, and only the first 2 characters are needed (any others are ignored).

2.5.6. CGNS

Imports a CGNS file The external import routine is `ImportCGNS`. Available options are:

`-v` Verbose output. Echo additional data to `stdout` during the import.

`-b` Read a grid from the specific CGNS base.

`-B` Read all CGNS bases. (default)

`-c` Read BOCO information as 2D regions.

`-f` Import Family Information as regions.

`-E` Import each Element Section as a separate region.

`-I` Import each side of a connection as a separate region.

`-P` Do not add the Zone name as a prefix to any region being defined.

2.5.6.1. SplitCGNS.exe

The `SplitCGNS.exe` program will take a single CGNS file and split it into multiple files on a "file per problem basis". The method for running this is:

```
SplitCGNS.exe [ -l ] <filename> <basename>
```

If the file contains two problems called "Pipe" and "Elbow", the import filter will only currently read "Pipe", but using SplitCGNS will produce two files called `basename_Pipe.cgns` and `basename_Elbow.cgns` each containing a single problem that can then be selected for import via the normal method.

Specifying the "`-l`" option "links" the part of the data in the original file to the created file using a relative pathname. The created file does not therefore need to duplicate data.

The "`-l`" option should only be used if the original file and resulting files are going to be kept relative to each other (that is, if when SplitCGNS was run the original file was in `../../example.cgns`, it must always remain in this position relative to the created files).

2.5.7. FLUENT

Imports FLUENT `msh` and `cas` files. The external import routine is `ImportFluent`. The import routine will read the mesh information from the `.cas` or `.msh` file.

Available command line options are:

`-v` Verbose output. Echo additional data to `stdout` during the import.

`-I` Import interior boundary conditions.

2.5.8. GridPro/az3000

Imports a GridPro/az3000 grid and connectivity file from Program Development Corporation (PDC).

The external import routine is `ImportPDC`. The import routine will attempt to determine the connectivity file associated with the grid file by appending the extension `conn` to the grid file name. If the file is not found, then the grid file name extension will be replaced by `conn` and the new file checked for. If neither of these are found, the import routine will look for a file named `conn.tmp`, and if found will use it. A command line option (`-c`) is also available to explicitly name the connectivity file.

If a connectivity file is found, the interface information in the file will be used to eliminate the duplicate nodes at block interfaces, and boundaries conditions will be imported as regions into CFX. If the boundary condition is named in the connectivity file, then that name will be used for the region name, else the default name **UnnamedRegionX** with the X replaced by a number will be used. If a connectivity file is not found, or the command line option to ignore the connectivity file is given (`-i`), then only the grid file will be imported, resulting in duplicate nodes at the block interfaces. You may then want to eliminate these duplicate nodes with the command line option (`-d` or `-D`).

Available options are:

`-v` Verbose output. Echo additional data to `stdout` during the import.

`-i` Ignore the connectivity file. Duplicate nodes will result and no regions will be imported.

`-c <connfile>` Set the name of the connectivity file associated with the grid file to *`connfile`*.

`-p` Include periodic boundary conditions as regions. These are not normally included in the import. Setting this flag will result in these being imported as regions.

`-q` Read from the property file

`-P <propfile>` Set the name of the property file associated with the grid file to *`propfile`*.

`-3` Import grid blocks as 3D regions

2.5.9. I-DEAS

Imports an I-DEAS Universal file from SDRC. The external import routine is ImportIDEAS. Reads datasets 781 and 2411 (nodes) as nodes, 780 and 2412 (elements) as elements, and nodes (type 7) from datasets 752 and 2417 (permanent groups) as regions. All other datasets are read, but not processed.

Available options are:

`-v` Verbose output. Echo additional data to `stdout` during the import.

`-n` Import nodes in a PERMANENT group as a 2D region.

`-l` Import elements in a PERMANENT group as a 3D region.

`-f` Import faces in a PERMANENT group as a 2D region.

2.5.10. ICEM CFX

Imports a file written for CFX by ICEM Tetra. The external import routine is `ImportICEM`. Available options are:

`-v` Verbose output. Echo additional data to `stdout` during the import.

`-P` Read coordinate data from a binary file as double precision.

2.5.11. PATRAN

Imports a PATRAN Neutral file. The external import routine is `ImportPatran`. Reads packet 01 (nodes) as nodes, packet 02 (elements) as elements, and nodes (type 5) from packet 21 (named groups) as regions.

A command line option is available to read packet 06 (loads) as regions also. All other packets are read, but not processed.

Available options are:

`-v` Verbose output. Echo additional data to `stdout` during the import.

`-l` Import packet 06 (distributed loads) as regions. The regions will be assigned the name PatranLoad*X* where the *X* is replaced by the load ID number.

2.5.12. NASTRAN

Imports a NASTRAN file. The external import routine is `ImportMSC`. Currently reads only nodes (GRID), tet (CTETRA) and hex (CHEXA) elements.

Available options are:

`-v` Verbose output. Echo additional data to `stdout` during the import.

`-l` Import PLOAD4 datasets as 2D regions.

`-s` Import PSOLID datasets as 3D regions.

2.5.13. CFX-TASCflow

Imports TASCflow Version 2 files. The external import routine is `ImportGRD`. The import routine will read the mesh information from the GRD file and automatically remove duplicate nodes where interfaces are defined and are 1:1.

Available command line options are:

`-v` Verbose output. Echo additional data to `stdout` during the import.

`-V` More verbose output.

`-i` Ignore the blockoff file (BCF).

`-c` Ignore GCI file.

`-o` Old style 2.4 format.

`-b <file>` Specifies a `bcf` file that contains blocked-off regions (boundary condition information is ignored). For details, see CFX-TASCflow Files in the *CFX-Pre User's Guide*.

`-g <file>` Specifies the `gci` file to import. For details, see CFX-TASCflow Files in the *CFX-Pre User's Guide*.

`-f` Formatted (ASCII) GRD file.

`-u` Fortran unformatted GRD file.

`-3` Import labelled 3D regions.

Chapter 3: Mesh and Results Export API

This chapter describes how to create a custom program for exporting mesh and results data. Information on using such a program is given in Using a Customized Export Program.

This chapter describes:

3.1. Creating a Customized Export Program

The mesh and results contained within an ANSYS CFX results file can be exported in many formats, ready for input into post-processing software other than CFD-Post, MSC/PATRAN, EnSight and Fieldview. To do this, you would write a customized export program that calls routines from the Export Application Programming Interface (API). However, this is recommended only for advanced users, because it involves at least some knowledge of C or C++ programming language.

Once an export program has been created, it can be used by any number of users; so if other ANSYS CFX users at a site regularly use a different post-processor, it may be worth contacting a system administrator to find out if such a format has already been defined.

To define a new format, use the export API. The general steps to follow are:

1. Create a file that contains instructions needed to build the format in C.

   This is most easily done by editing the template file provided (which is written in C). For details, see *An Example of an Export Program* (p. 38).

2. Compile your C program.

   For details, see *Compiling Code with the Mesh and Results Export API* (p. 49).

3. Link the C program into the CFX code.

   For details, see *Linking Code with the Mesh and Results Export API* (p. 49).

4. Use the program.

   For details, see Using a Customized Export Program.

Numerous keywords are required for development and use of custom export files. For details, see cfx5export Arguments.

An example source routine can be used as the basis of a customized program; one is given in the next section.

3.1.1. An Example of an Export Program

The following is an annotated listing of the C source code for a reasonably simple example of a customized Export program. The full source code is available for use as a template and is located in `CFX/examples/ExportTemplate.c`, where CFX is the directory in which CFX is installed.

The example program is a reasonably simple example of an export program, which opens a CFX results file, writes a geometry file (ignoring pyramid elements) and several files containing results. After the program listing, a sample of the output produced is shown.

3.1.1.1. File Header

The file header uses several #include entries. The first set includes standard header files.

```
#include <stdio.h>
#include <string.h>
#include <stdlib.h>
#include <io.h>
```

The second set includes cfx5export header files.

```
#include "cfxExport.h"
#include "getargs.h"
```

Obtaining CFX-Mesh and Results Export API header files is described in more detail. For details, see *Linking Code with the Mesh and Results Export API* (p. 49).

3.1.1.2. Allowed Arguments

The definition of allowed arguments appears as:

```
static char options[] = "u:d:t:cif";
```

The following piece of code simply defines the message that is printed if the user enters incorrect options to the program.

```
static char *usgmsg[] = {
   "usage: ExportTemplate [options] res_file [basename]",
   " options are:",
   " -u<level>     = user level of interest",
   " -d<domain>    = domain of interest (default is 0 - all the domains",
   "                 are combined into a single domain)",
   " -t<timestep>  = timestep of interest (if set to -1, all timesteps",
   "                 are exported)"
   " -c            = use corrected boundary node data",
   " -i            = include boundary node only data",
   " -f            = get info on the res_file (No output is created)",
   " <basename> is the base filename for Template file output.",
   "If not specified, it defaults to 'res_file'. The Template",
   "geometry file will be written to <basename>.geom, the",
   "results file to <basename>.res, and the variables to",
   "<basename>.s## or <basename>.v## where ## is the variable",
   "number and s indicates a scalar and v a vector.",
    NULL
};
```

3.1.1.3. Main Program Initialization

As is standard, the variables `argc` and `argv` are the number of arguments and a pointer to the argument list. The variable `cfxCNT_SIZE` and the types `cfxNode` and `cfxElement` are defined in the header file `cfxExport.h` as are all variables and functions starting with the letters `cfx`. For details, see *Mesh*

and Results Export API (p. 37). The variables `level`, `zone`, `alias`, `bndfix` and `bnddat` are used for setting the default values for the various parameters that can be set on the command line of the program.

```
void main (int argc, char *argv[])
{
   char *pptr;
   char baseFileName[256], fileName[256], errmsg[256];
   int i, n, counts[cfxCNT_SIZE], dim, length, namelen;
   int nnodes, nelems, nscalars, nvectors, nvalues;
   int level = 1, zone = 0, alias = 1, bndfix = 0, bnddat = 0;
   int timestep = -1, infoOnly = 0;
   int ts, t, t1, t2;
   int nTimeDig = 1;     /* number of digits in transient file suffix */
   char zoneExt[256];    /* zone extension added to the base filename */
   int isTimestep = 0;
   float timeVal = 0.0;  /* time value in the single timestep mode */
   char *wildcard = { "******" }; /* used in transient file specification */
   FILE *fp;
   cfxNode *nodes;
   cfxElement *elems;
   float *var;
```

The variable `cfxCNT_SIZE` and the types `cfxNode` and `cfxElement` are defined in the header file `cfxExport.h` as are all variables and functions starting with the letters `cfx`. For details, see *Mesh and Results Export API* (p. 37). The variables `level`, `zone`, `alias`, `bndfix` and `bnddat` are used for setting the default values for the various parameters that can be set on the command line of the program.

The following line prints an error message if there are not enough arguments to proceed.

```
if (argc < 2)
   cfxUsage (usgmsg, NULL);
```

The following piece of code reads the specified options and assigns values to certain variables accordingly. If an invalid or incomplete option is specified, then `getargs` prints an error message and the export program stops.

```
while ((n = getargs (argc, argv, options)) > 0) {
   switch (n) {
      case ‘u’:
         level = atoi (argarg);
         break;
      case ‘d’:
         zone = atoi (argarg);
         break;
      case ‘t’:
         timestep = atoi (argarg);
         isTimestep = 1;
         break;
      case ‘c’:
         bndfix = 1;
         break;
      case ‘i’:
         bnddat = 1;
         break;
      case ‘f’:
         infoOnly = 1;
         break;
   }
}
```

After this, the `level` variable contains the user level specified. All results are output if they are of this user level or below it. The `zone` variable contains the domain number that you specified. The variable `alias` determines whether the variables are referred to by their long names or short names. The default here is for short names to be used because some post-processors need variable names to contain no spaces, but you are encouraged to use long variable names wherever possible. The variable `bndfix` determines whether the variables are exported with corrected boundary node values - if `bndfix` is set

to 1, then corrected values are used. Finally, `bnddat` determines whether variables that contain meaningful values only on the boundary (such as Yplus) are exported or not; if `bnddat` is set to 1, then these variables are exported.

3.1.1.4. Checking File Names

The following code checks to make sure that a CFX results file has been specified, and that it can be read by the export program. If this is not the case, the export program exits.

```
/* CFX-5 results file */
   if (argind >= argc)
       cfxUsage (usgmsg, "CFX-5 results file not specified");
   if (access (argv[argind], 0)) {
      fprintf (stderr, "result file <%s> does not exist\n", argv[argind]);
      exit (1);
   }
```

The following code writes the basename specified to the character array `baseFileName`. If one was not specified, then it defaults to the name of the results file specified. A basename name may be specified in another directory (for example, "../template/output"). However, later in the code this basename without the preceding directory information is required (in this example "output"); and so the pointer `pptr` is assigned to point to the first character of this name.

```
/* base file name */
   if (argind + 1 < argc)
      strcpy (baseFileName, argv[argind+1]);
   else
      strcpy (baseFileName, argv[argind]);
if (NULL != (pptr = strrchr (baseFileName, '/')))
      pptr++;
   else if (NULL != (pptr = strrchr (baseFileName, '\\')))
      pptr++;
   else
      pptr = baseFileName;
```

The following code checks that the results file that will be produced by the export program will not overwrite an existing results file.

```
/* don't overwrite results file */
sprintf (fileName, "%s.res", baseFileName);
if (0 == strcmp (argv[argind], fileName)) {
fprintf (stderr, "Template res file would overwrite CFX results file\n");
fprintf (stderr, "Need to select new Template output base file name\n");
exit (1);
   }
```

3.1.1.5. Opening the CFX Results File

The following code prints a message to the screen telling you that the program is reading the results file. It then calls `cfxExportInit`, which must always be called before any of the other export routines. The variable `n` is set to equal the number of zones in the results file. If the `-f` option has been selected, information about the results file will be displayed. The number of domains to be exported is also determined so that the format of the exported file includes the appropriate suffix. Finally, a check is made to make sure that the zone (if any) that you specified in the program options is a valid zone for this results file.

```
/* open CFX-5 results file */
printf ("\nreading CFX results from <%s>\n", argv[argind]);
   n = cfxExportInit (argv[argind], NULL);
   if (infoOnly) {
      int nt;
      printf("\n%d domains:\n", n);
      for(i = 1; i <= n; i++)
```

```
         printf("  %d   %s\n", i, cfxExportZoneName(i));
      nt = cfxExportTimestepCount();
      printf("%d timesteps:\n", nt);
      if(nt) {
         for(i = 1; i <= nt; i++)
            printf("  %d\n", cfxExportTimestepNumGet(i));
      }
      cfxExportDone();
      exit (0);
   }
   /* determine the zone suffix for the export files */
   strcpy(zoneExt, "");
   if(zone == 0) {
      printf ("processing all domains\n");
      cfxExportSetVarParams(bndfix, level);
   }
   else {
      printf ("processing domain %d\n", zone);
if(n != 1) {
         float f;
         int nZoneDig = 0;
/* count number of digits needed to fit any zone number */
         f = (float) n;
         while((f /= 10) >= 1) nZoneDig++;
         sprintf(zoneExt, "_d%*.*d", nZoneDig, nZoneDig, zone);
      }
   }
if (cfxExportZoneSet (zone, counts) < 0)
         cfxExportFatal ("invalid zone number");
```

The following code is ignoring any pyramid elements (elements with 5 nodes) and decreases `nelems` by the number of pyramid elements. It then checks to make sure that neither the number of nodes nor the number of elements is zero; if so, the program exits with return code -1.

The first two lines focus on the number of nodes in the zone and the number of elements in the zone.

```
nnodes = cfxExportNodeCount();
nelems = cfxExportElementCount();
if (counts[cfxCNT_PYR]) {
   printf ("%d pyramid elements found - they are being ignored\n",
      counts[cfxCNT_PYR]);
   nelems -= counts[cfxCNT_PYR];
}
if (!nnodes || !nelems)
   cfxExportFatal ("no nodes and/or elements");
```

3.1.1.6. Timestep Setup

The following code determines whether all of the timesteps, a specific timestep or the final timestep (steady-state) have been selected for export.

```
if(isTimestep && timestep == -1 && !cfxExportTimestepCount()) {
      isTimestep = 0;
   }
   if(isTimestep) {
      int i;
      float f;
      if(timestep == -1) {
         printf("processing all timesteps\n");
         t1 = 1;
         t2 = cfxExportTimestepCount() + 1;
      }
      else {
         int isFound = 0;
         printf("processing timestep %d\n", timestep);
         for(i = 1; i <= cfxExportTimestepCount() + 1; i++)
            if(cfxExportTimestepNumGet(i) == timestep) {
               timeVal = cfxExportTimestepTimeGet(i);
               t1 = t2 = i;
```

```
               isFound = 1;
               break;
            }
         if(!isFound) {
            sprintf(errmsg, "\nTimestep %d not found. "
               "Use -f to see the list of valid timesteps.\n", timestep);
            cfxExportFatal (errmsg);
         }
      }
      /* count number of digits needed to fit any timestep number */
      f = (float) cfxExportTimestepCount();
      while((f /= 10) >= 1) nTimeDig++;
   }
   else {
      timeVal = cfxExportTimestepTimeGet(cfxExportTimestepCount() + 1);
      timestep = cfxExportTimestepNumGet(cfxExportTimestepCount() + 1);
      t1 = t2 = cfxExportTimestepCount() + 1;
   }
```

3.1.1.7. Geometry File Output

The following code opens the geometry file `basename.geom`, printing an error if it cannot be opened for any reason. A message is then displayed informing the user that the application is writing the geometry file.

```
/* Template geometry output */
   sprintf (fileName, "%s.geom", baseFileName);
   if (NULL == (fp = fopen (fileName, "w+"))) {
      sprintf (errmsg, "can't open <%s> for output", fileName);
      cfxExportFatal (errmsg);
   }
   printf ("writing Template Geometry file to <%s>\n", fileName);
```

The header of this file is shown after the program listing.

```
/* write header */
   fprintf( fp, "Template Geometry file exported from CFX\n");
   fprintf( fp, " \n");
   fprintf( fp, "node id given\n");
   fprintf( fp, "element id off\n");
```

The following code writes first the word "coordinates" and the number of nodes that will be written. The pointer nodes is initialized to point at the data for the first node and the node data is written into the geometry file. For each node, a node number is written, followed by the three coordinates of that node. Note that `n` ranges between 0 and `nnodes-1`. This program adds 1 to each node number so that the nodes in the geometry file are numbered between 1 and `nnodes`. When it has finished, the `cfxExportNodeFree` routine frees the memory that was used to store the node data, and finally the word "done" is printed on the screen to alert you that it has finished writing the node data.

```
/* write nodes */
   fprintf( fp, "coordinates\n");
   fprintf( fp, "%8d\n", nnodes );
   nodes = cfxExportNodeList();
   printf ("  writing %d nodes ...", nnodes);
   fflush (stdout);
   for (n = 0; n < nnodes; n++, nodes++) {
fprintf( fp, "%8d %12.5e %12.5e %12.5e\n", n + 1, nodes->x,
   nodes->y, nodes->z );
}
   cfxExportNodeFree();
printf (" done\n");
```

Next, the data for each element must be written.

Firstly, some general information is written. Then the data for each element type is written in turn.

```
/* write elements */
   fprintf( fp, "part 1\n" );
   fprintf( fp, "volume elements\n");
   printf ("  writing %d elements...", nelems);
   fflush (stdout);
```

For tetrahedral elements, the word "tetra4" is written to the file, followed by the number of tetrahedral elements written.

```
/* tets */
   fprintf( fp, "tetra4\n");
   fprintf( fp, "%8d\n", counts[cfxCNT_TET] );
```

The following code is executed only if the number of tetrahedral elements is non-zero. Assuming this, `elems` is set to point to the list of elements stored in the results file. The index n loops over all the elements. For each element, the following step is carried out: "If the element is a tetrahedron, then loop over its four vertices and write their node numbers to the geometry file, then start a new line (ready for the next set of data)." The output produced can be seen in the examples of the exported files in the next section.

```
if (counts[cfxCNT_TET]) {
   elems = cfxExportElementList();
   for (n = 0; n < nelems; n++, elems++) {
      if (cfxELEM_TET == elems->type) {
         for (i = 0; i < elems->type; i++)
            fprintf (fp, "%8d", elems->nodeid[i]);
         putc ('\n', fp);
      }
   }
}
```

For wedges (triangular prisms) and hexahedral elements, the same procedure is followed. However, there is a slight difference in the way that the `fprintf` line is written for hexahedral elements. This is because the order that the element nodes are written to the geometry file is different to the order in which they were read from the results file. This may need to be done if a post-processor has a different convention for node order than the one that the cfx5export node routines have. The order the nodes are written in will affect which node is connected to which. The node ordering for exported elements is illustrated in *cfxExportElementList* (p. 55).

```
/* wedges */
   fprintf( fp, "penta6\n");
   fprintf( fp, "%8d\n", counts[cfxCNT_WDG] );
   if (counts[cfxCNT_WDG]) {
      elems = cfxExportElementList();
      for (n = 0; n < nelems; n++, elems++) {
         if (cfxELEM_WDG == elems->type) {
            for (i = 0; i < elems->type; i++)
               fprintf (fp, "%8d", elems->nodeid[i]);
         }
         putc ('\n', fp);
      }
   }
/* hexes */
   fprintf( fp, "hexa8\n");
   fprintf( fp, "%8d\n", counts[cfxCNT_HEX] );
   if (counts[cfxCNT_HEX]) {
      elems = cfxExportElementList();
      for (n = 0; n < nelems; n++, elems++) {
         if (cfxELEM_HEX == elems->type)
            fprintf (fp, "%8d%8d%8d%8d%8d%8d%8d%8d\n",
               elems->nodeid[0], elems->nodeid[1],
               elems->nodeid[3], elems->nodeid[2],
               elems->nodeid[4], elems->nodeid[5],
               elems->nodeid[7], elems->nodeid[6]);
      }
   }
```

Then the geometry file is closed and the memory occupied by the element data is freed.

```
printf (" done\n");
   fclose (fp);
   cfxExportElementFree();
```

3.1.1.8. Template Results File

Despite its name, the Template results file does not contain any actual values of results. It simply contains information about how many variables there are and in which file each is stored.

The first job is to make sure that there are some results for export. First, the code checks that there is a nonzero number of variables that have the specified user level. Then it counts the number of scalar and vector variables that will be exported. To be exported, a variable must:

1. Have a dimension of 1 (scalar variable) or 3 (vector variable) and,
2. Either be a variable with useful values everywhere in the zone or be a variable that has values only on the boundaries (in which case it will be exported only if you asked to "include boundary node only data" by specifying the option `-i` when starting the export program, which translated to setting `bnddat = 1` when the arguments were processed).

Review the `cfxExportVariableSize` routine if this logic is unclear. For details, see *cfxExportVariableSize* (p. 62).

Once results are identified, the code calculates the variable `namelen`, which is the length of the longest variable name to be exported (the `alias` variable was set when processing the arguments passed to the export program, and depends upon whether you wanted to use long names or short names). If there are no vector or scalar variables to be exported, the export program exits.

```
/* output results file */
   nscalars = nvectors = namelen = 0;
   if ((nvalues = cfxExportVariableCount(level)) > 0) {
      for (n = 1; n <= nvalues; n++) {
         cfxExportVariableSize (n, &dim, &length, &i);
         if ((1 != dim && 3 != dim) ||
            (length != nnodes && length != bnddat))
            continue;
         if (1 == dim)
            nscalars++;
         else
            nvectors++;
         i = strlen (cfxExportVariableName (n, alias));
         if (namelen < i)
            namelen = i;
      }
   }
   if (0 == (nscalars + nvectors)) {
      cfxExportDone ();
      exit (0);
   }
```

The following code checks that the results file can be opened for writing to, and exits if not. The number of scalar and vector variables are written to the file, followed by some numbers (which EnSight, for example, requires) that are always the same for any export of this kind.

```
sprintf (fileName, "%s.res", baseFileName);
   if (NULL == (fp = fopen (fileName, "w+"))) {
      sprintf (errmsg, "can't open <%s> for writing", fileName);
      cfxExportFatal (errmsg);
   }
   printf ("writing Template results file to <%s>\n", fileName);
   fflush (stdout);
   fprintf( fp, "%d %d 0\n", nscalars, nvectors );
```

```
    fprintf( fp, "%d\n", t2 - t1 + 1 );
    for(i = t1; i <= t2; i++) {
       fprintf( fp, "%13.4e", cfxExportTimestepTimeGet(i));
       if(!(i % 6)) fprintf( fp, "\n");
    }
    fprintf( fp, "\n");
    if(isTimestep && t1 != t2)
       fprintf( fp, "0 1\n");
```

Next, for each scalar variable, a line is written that contains the filename where the scalar will be written, and then the name of the variable. Note that the filename is not the basename, but the basename with all the directory structure (if any) stripped off the front. For details, see *Checking File Names* (p. 40). This is done because these file will be written in the same directory as this Template results file, so there is no need for directory information.

```
 if ( nscalars ) {
    for (n = 1; n <= nvalues; n++) {
       cfxExportVariableSize (n, &dim, &length, &i);
       if (1 == dim && (length == nnodes || length == bnddat))
          if(!isTimestep)
             fprintf (fp, "%s%s.s%2.2d %s\n", pptr, zoneExt,
                n, cfxExportVariableName(n, alias));
          else if(t1 == t2)
             fprintf (fp, "%s%s_t%d.s%2.2d %s\n", pptr, zoneExt,
                cfxExportTimestepNumGet(t1), n,
                cfxExportVariableName(n, alias));
          else
             fprintf (fp, "%s%s_t%*.*s.s%2.2d %s\n", pptr, zoneExt,
                nTimeDig, nTimeDig, wildcard, n,
                cfxExportVariableName(n, alias));
       }
    }
```

The same information is then written for each vector variable and the Template results file is closed.

```
 if ( nvectors ) {
    for (n = 1; n <= nvalues; n++) {
       cfxExportVariableSize (n, &dim, &length, &i);
       if (3 == dim && (length == nnodes || length == bnddat))
          if(!isTimestep)
             fprintf (fp, "%s%s.v%2.2d %s\n", pptr, zoneExt,
                n, cfxExportVariableName(n, alias));
          else if(t1 == t2)
             fprintf (fp, "%s%s_t%d.v%2.2d %s\n", pptr, zoneExt,
                cfxExportTimestepNumGet(t1), n,
                cfxExportVariableName(n, alias));
          else
             fprintf (fp, "%s%s_t%*.*s.v%2.2d %s\n", pptr, zoneExt,
                nTimeDig, nTimeDig, wildcard, n,
                cfxExportVariableName(n, alias));
       }
    }
 fclose( fp );
```

3.1.1.9. Creating Files with Results for Each Variable

The results for each variable are written to separate files, called <*basename*>.s01, <*basename*>.s02, <*basename*>.v03, for example. Each file with an extension containing a letter "s" contains a scalar variable, and each with a "v" contains a vector variable. Which variable is written to each file is tabulated in the Template results file that has just been written.

The following code reads the information for each variable, after you decide that it should be exported - the logic is very similar to that used when counting the relevant variables when creating the Template results file. The marked `if` loop executes if the variable needs to be exported. It checks to make sure that the variable information can be read, and (assuming it can) then builds the filename and checks

to see if it can be opened. Continuing, it writes to the screen where it is putting the variable, and then loops through all the values, writing them to the file, inserting a new line every six values. After each variable, the memory used to store that variable is restored.

After all the variable files have been written, the program calls the `cfxExportDone` routine, which close the CFX results file, and frees up any remaining memory. This routine must be the last call to any of the API routines. The program then exits.

Note

This program makes no use of any of the region routines, which enable access to boundary condition data, nor the volume routines that enable access to the subdomains that are defined for a problem.

- *Region Routines* (p. 56)
- *Volume Routines* (p. 59)

```
/* output each timestep to a different file */
   for(t = t1; t <= t2; t++) {
      ts = cfxExportTimestepNumGet(t);
      if(cfxExportTimestepSet(ts) < 0) {
         continue;
      }
   /* build file name and open file */
   if(!isTimestep)
      sprintf( fileName, "%s%s.%c%2.2d", baseFileName, zoneExt,
         1 == dim ? 's' : 'v', n);
      else if(t1 == t2)
         sprintf( fileName, "%s%s_t%d.%c%2.2d", baseFileName, zoneExt,
            ts, 1 == dim ? 's' : 'v', n);
      else
         sprintf( fileName, "%s%s_t%*.*d.%c%2.2d", baseFileName, zoneExt,
            nTimeDig, nTimeDig, t-1, 1 == dim ? 's' : 'v', n);
      if (NULL == (fp = fopen (fileName, "w+"))) {
         sprintf (errmsg, "can't open <%s> for writing\n", fileName);
         cfxExportFatal (errmsg);
      }
      printf ("  %-*s -> %s ...", namelen,
         cfxExportVariableName(n, alias), fileName);
      fflush (stdout);
      fprintf( fp, "%s\n", cfxExportVariableName(n, alias));
      length = nnodes * dim;
      for ( i = 0; i < length; i++, var++ ) {
         fprintf( fp, "%12.5e ", *var );
         if ( i && 5 == (i % 6) )
            putc ('\n', fp);
      }
      if ( 0 != ( nvalues % 6 ) )
         putc( '\n', fp );
      fclose( fp );
      cfxExportVariableFree (n);
      printf (" done\n");
      }
   }
   } /* loop for each timestep */
      cfxExportDone();
      exit (0);
}
```

3.1.2. Example of Output Produced

If the export program is correctly compiled and run, the following output is obtained. For details, see Using a Customized Export Program.

In this example, the CFX results file contains three variables at user level 1: pressure, temperature and velocity. This is in a file named file.res. No timesteps or domains were specified, and the basename was specified as an example.

The following is displayed on screen:

```
reading CFX results from <file.res>
processing all domains
writing Template Geometry file to <example.geom>
   writing 2365 nodes ... done
   writing 11435 elements... done
writing Template results file to <example.res>
writing variable output files
   Pressure    -> example.s01 ... done
   Temperature -> example.s02 ... done
   Velocity    -> example.v03 ... done
```

Five files are produced: the geometry file `example.geom`, the Template results file `example.res`, and three variable files called `example.s01`, `example.s02` and `example.v03`, which contain the results for pressure, temperature and velocity, respectively. For details, see:

- *example.geom* (p. 47)
- *example.res* (p. 47)
- *example.s01* (p. 48)

3.1.2.1. example.geom

The content of this file appears as:

```
Template Geometry file exported from CFX
node id given
element id off
coordinates
   2365
      1 2.00000e+00 0.00000e+00 0.00000e+00
      2-2.00000e+00-6.51683e-07 0.00000e+00
      3 2.00000e+00 0.00000e+00 2.00000e+00
      4-2.00000e+00-6.51683e-07 2.00000e+00
      5 3.00000e+00 1.00000e+00 5.00000e-01
      ....
      ....
      ....
      2362-1.13337e+00 2.18877e-01 4.02491e-01
      2363-1.12115e+00-3.66598e-01 2.22610e-01
      2364 1.36924e+00 4.78359e-01 1.22588e-01
      2365-3.30703e-01 1.38487e+00 2.23515e+00
part 1
volume elements
tetra4
   11435
      754     230      12     145
      755     216       8     122
      756     212     125     215
      ....
      ....
      ....
      2365   496      475     474
penta6
   0
hexa8
   0
```

3.1.2.2. example.res

The content of this file appears as:

```
2 1 0
1
0.0
0 1
example.s01 Pressure
example.s02 Temperature
example.v03 Velocity
```

3.1.2.3. example.s01

The content of this file appears as:

```
Pressure
   1.42748e+04 1.42621e+04 1.43425e+04 1.43350e+04 1.44118e+04 1.44777e+04
   1.38639e+04 1.37352e+04 1.44130e+04 1.44755e+04 1.37733e+04 1.37626e+04
   ....
   ....
   ....
   1.39092e+04 1.40699e+04 1.24139e+04 1.34786e+04 1.34859e+04 1.37959e+04
```

3.1.3. Source Code for getargs.c

The following code is the C code that defines the functions `cfxUsage` and `getargs`, both of which are called by the example listing above. You do not need to include this code with your custom export program (it is automatically linked in if you use the compiler as described in the next section).

```
#include <stdio.h>
#include <string.h>
#include <ctype.h>
#include "getargs.h"
/*---------- usage -------------------------------------------------
 * display usage message and exit
 *------------------------------------------------------------------*/
void cfxUsage (
#ifdef PROTOTYPE
   char **usgmsg, char *errmsg)
#else
   usgmsg, errmsg)
char **usgmsg, *errmsg;
#endif
{
   int n;
   if (NULL != errmsg)
      fprintf (stderr, "ERROR: %s\n", errmsg);
   for (n = 0; NULL != usgmsg[n]; n++)
      fprintf (stderr, "%s\n", usgmsg[n]);
   exit (NULL != errmsg);
}
/*---------- getargs -----------------------------------------------
 * get option letter from argument vector or terminates on error
 * this is similar to getopt()
 *------------------------------------------------------------------*/
int argind = 0;  /* index into argv array */
char *argarg;    /* pointer to argument string */
int getargs (
#ifdef PROTOTYPE
   int argc, char **argv, char *ostr)
#else
   argc, argv, ostr)
int argc;
char **argv, *ostr;
#endif
{
   int argopt;
   char *oli;
   static char *place;
   static int nextarg;
   /* initialisation */
   if (!argind)
```

```
      nextarg = 1;
   if (nextarg) {  /* update scanning pointer */
      nextarg = 0;
      /* end of arguments */
      if (++argind >= argc || '-' != argv[argind][0])
         return (0);
      place = argarg = &argv[argind][1];
   }
   /* check for valid option */
   if ((argopt = *place++) == ':' ||
      (oli = strchr (ostr, argopt)) == NULL) {
      fprintf (stderr, "invalid command line option `%c'\n", argopt);
      exit (1);
   }
   /* check for an argument */
   if (*++oli != ':') {        /* don't need argument */
      argarg = NULL;
      if (!*place)
         nextarg = 1;
   }
   else {                      /* need an argument */
      if (!*place) {
         if (++argind >= argc) {
            fprintf (stderr, "missing argument for option `%c'\n", argopt);
            exit (1);
         }
         place = argv[argind];
      }
      argarg = place;
      nextarg = 1;
   }
   return (argopt);    /* return option letter */
}
```

3.2. Compiling Code with the Mesh and Results Export API

Compilation of a customized executable must be performed using an appropriate compiler and compiler flags.

The customized executable must also be linked with the provided Mesh and Results Export API library and the provided i/o library as detailed in *Linking Code with the Mesh and Results Export API* (p. 49).

3.2.1. Compiler Flags

No compiler flags are necessary for successful compilation on linux-ia64, but use the icc compiler.

3.3. Linking Code with the Mesh and Results Export API

In order to build a customized export utility, it must be linked with several libraries. With the exception of `bufferoverflowu.lib`, these libraries are located in `<CFXROOT>/lib/<os>/`:

- `libmeshexport.lib` (on Windows), or `libmeshexport.a` (on Linux)
- `libratlas_api.lib` (on Windows), or `libratlas_api.a` (on Linux)
- `libratlas.lib` (on Windows), or `libratlas.a` (on Linux)
- `libpgtapi.lib` (on Windows), or `libpgtapi.a` (on Linux)
- `libunits.lib` (on Windows), or `libunits.a` (on Linux)
- `libcclapilt.lib` (on Windows), or `libcclapilt.a` (on Linux)
- `libio.lib` (on Windows), or `libio.a` (on Linux)
- `bufferoverflowu.lib` (on Windows 64-bit)

3.3.1. UNIX

On most UNIX systems, build the executable with the command:

```
cc export.c -o export.exe -I<CFXROOT>/include/ -L<CFXROOT>/lib/<OSDIR> -lmeshexport -lratlas_api -lratlas -lpgtapi -lcc
```

where <CFXROOT> is the directory in which CFX is installed and <*OSDIR*> is a directory name corresponding to the architecture of the machine that you are running on (that is, `linux-ia64`). In this example, your own export program is named `export.c` and the executable file will be called `export.exe`.

The compiler flags and required libraries may vary, depending on the compiler and the custom program.

3.3.2. Windows (32-bit)

You can build the executables on 32-bit Windows systems that have Microsoft Visual C++ 2008 Express Edition. An example command line follows:

```
cl /MD /I "C:\Program Files\Ansys Inc\v140\CFX\include" ExportTemplate.c
/link /libpath:"C:\Program Files\Ansys Inc\v140\CFX\lib\winnt" lib-
cclapilt.lib libio.lib libmeshexport.lib libunits.lib libpgtapi.lib lib-
ratlas_api.lib libratlas.lib
```

3.3.3. Windows (64-bit)

You can build the executables on 64-bit Windows systems that have the compiler from VS2008. An example command line follows:

```
cl /MD /I “C:\Program Files\Ansys Inc\v140\CFX\include” ExportTemplate.c
/link /libpath:”C:\Program Files\Ansys Inc\v140\CFX\lib\winnt-amd64”
libcclapilt.lib libio.lib libmeshexport.lib libunits.lib libpgtapi.lib
libratlas_api.lib libratlas.lib bufferoverflowu.lib
```

3.4. Details of the Mesh Export API

The full list of constants, data structures, types and functions available to the programmer are given in the following sections:

3.4.1. Defined Constants and Structures

The following constants and data structures are defined in the header file `cfxExport.h`, which should be included in the export program.

3.4.1.1. Element Types

CFX can use 4 types of element, which are identified by the number of nodes: tetrahedrons (4 nodes), pyramids (5 nodes), prisms/wedges (6 nodes), and hexahedrons (8 nodes). The element types are identified in the Export API by the following constants:

```
#define cfxELEM_TET 4
#define cfxELEM_PYR 5
#define cfxELEM_WDG 6
#define cfxELEM_HEX 8
```

3.4.1.2. Volume List Types

The Export API contains functions that enable the user to query how volumes are defined in the results file. It is possible to request how a volume is defined in terms of nodes or elements. (For details, see *Volume Routines* (p. 59).)

The following constants are defined in the header file and should be used as arguments to the Volume routines:

```
#define cfxVOL_NODES 0
#define cfxVOL_ELEMS 1
```

3.4.1.3. Region List Types

The Export API contains functions that enable the user to query how regions are defined in the results file. It is possible to request how a region is defined in terms of nodes or faces. (For details, see *Region Routines* (p. 56).)

The following constants are defined in the header file and should be used as arguments to the Region routines:

```
#define cfxREG_NODES 0
#define cfxREG_FACES 1
```

In the case of nodes, the global node number is returned, while in the case of faces, the returned value is a combination of the global element number and local face number of the element. The following macros are available to enable the user to extract the element and face number from the combined value:

```
#define cfxFACENUM(face) ((face) & 7)
#define cfxELEMNUM(face) ((face) >> 3)
```

3.4.1.4. Count Entries

Two routines exist for initializing the Export API (see *cfxExportInit* (p. 52)) and requesting the totals of certain quantities in a zone (see *cfxExportZoneSet* (p. 53)). The array returned from both of these routines requires the following constants to be used by the calling program to reference the correct quantities.

```
enum cfxCounts {
cfxCNT_NODE = 0, /* number of nodes */
cfxCNT_ELEMENT, /* number of elements */
cfxCNT_VOLUME,   /* number of volumes */
cfxCNT_REGION, /* number of regions */
cfxCNT_VARIABLE, /* number of variables */
cfxCNT_TET, /* number of tetrahedral elements */
cfxCNT_PYR, /* number of pyramid elements */
cfxCNT_WDG, /* number of wedge elements */
cfxCNT_HEX, /* number of hexahedral elements */
cfxCNT_SIZE /* size of count array */
};
```

3.4.1.5. Node Data Structure

Nodes are represented in the Export API using the following structure (note the change in data type of x, y and z):

```
typedef struct cfxNode {
float x, y, z;
} cfxNode;
```

where `x`, `y`, and `z` are the coordinates of the node. A pointer to an array of these structures is returned by `cfxExportNodeList`. For details, see *cfxExportNodeList* (p. 54).

3.4.1.6. Element Data Structure

Elements are represented by the Export API using the following structure:

```
typedef struct cfxElement {
int type;
int *nodeid;
} cfxElement;
```

where `type` is the element type and `nodeid` is an array of node numbers that define the topology of the element. A pointer to an array of these structures is returned by `cfxExportElementList`. For details, see *Element Types* (p. 51) and *cfxExportElementList* (p. 55).

3.4.2. Initialization and Error Routines

The following routines open and close the CFX results file, initialize the Export API, and handle fatal error processing. The first call to any of the API routines must be `cfxExportInit` and the last call should be `cfxExportDone`. For details, see:

- *cfxExportInit* (p. 52)
- *cfxExportDone* (p. 52).

3.4.2.1. cfxExportInit

```
int cfxExportInit (char *resfile, int counts[cfxCNT_SIZE])
```

Opens the CFX results file named `resfile` and initializes the Export API. This should be the first call made to the API.

The routine returns the total number of zones. If the array `counts` is supplied to the routine (that is, it is not `NULL`), the array is filled with values representing the total number of nodes, elements, volumes, regions and variables for all the zones are returned in this array.

3.4.2.2. cfxExportDone

```
void cfxExportDone ()
```

Closes the CFX results file and destroys any internal storage used by the API. This should be the final call made to the Export API.

3.4.2.3. cfxExportError

```
void cfxExportError (void (*callback) (char *errmsg))
```

Specify a callback function that will be executed when a fatal error is generated by a call to `cfxImportFatal` (see *cfxImportFatal* (p. 15)). The argument, callback, is the function that will be called, it should take an argument that is the error message passed to cfxImportFatal. It is the responsibility of the function to terminate the application if required.

3.4.2.4. cfxExportFatal

```
void cfxExportFatal (char *errmsg)
```

Generate a fatal error message (`errmsg`) and close the ANSYS CFX results file. This routine also calls a callback function, if one has been specified by `cfxExportError` (see *cfxExportError* (p. 52)). If no callback function has been specified the function also terminates the application. There is no return from this call.

3.4.3. Zone Routines

A zone is defined as groups of nodes or faces that are located on the external boundaries of the domain. The following routines provide functionality for returning the number of zones in the open CFX results file specifying and requesting the current zone, and destroying any internal storage associated with a zone. All other routines in the Export API refer to quantities in the current zone being accessed by the API. By default the current zone is the global zone (a combination of all zones in the ANSYS CFX results file), but this can be the current zone can be altered by making a call to `cfxExportZoneSet` (see *cfxExportZoneSet* (p. 53)). Once this call has been made, any other function returns information about this zone until a subsequent call is made.

3.4.3.1. cfxExportZoneCount

```
int cfxExportZoneCount ()
```

Return the number of zones in the CFX results file.

3.4.3.2. cfxExportZoneSet

```
int cfxExportZoneSet (int zone, int counts[cfxCNT_SIZE])
```

Set the current zone being accessed by the Export API.

The value of zone should be between 1 and the value returned by `cfxExportZoneCount` (see *cfxExportZoneCount* (p. 53)) or 0 if the global zone is to be accessed.

The function returns 0 if the value of zone is invalid or the value zone if setting of the zone was successful.

The argument counts can be passed as a NULL pointer. In this case no information is returned to the calling function other than the return value mentioned above. If counts is specified it must be at least cfxCNT_SIZE in size, not specifying an array large enough can result in errors. In the case when counts is supplied correctly the total number of nodes, elements, volumes, regions and variables will be returned.

3.4.3.3. cfxExportZoneGet

```
int cfxExportZoneGet ()
```

Returns the current zone number.

3.4.3.4. cfxExportZoneFree

```
void cfxExportZoneFree ()
```

While a zone is being accessed, internal storage is allocated, this storage should be deallocated when no longer required. This can be done by calling `cfxExportZoneFree` or by calling `cfxExportNodeFree`, `cfxExportElementFree`, `cfxExportVolumeFree`, `cfxExportRegionFree` and `cfxExportVariableFree`. Details on each of these routines is available; see:

- *cfxExportNodeFree* (p. 55)
- *cfxExportElementFree* (p. 56)
- *cfxExportVolumeFree* (p. 60)
- *cfxExportRegionFree* (p. 58)
- *cfxExportVariableFree* (p. 63).

3.4.3.5. cfxExportZoneIsRotating

```
int cfxExportZoneIsRotating(double rotationAxis[2][3], double *angularVelocity)
```

Query whether the current zone is rotating and describe axis and angular velocity of the rotation if applicable. Returns 1 if the current zone is rotating and 0 if it is not; for the combined zone the return value is always -1. If successful the rotation axis is returned in `rotationAxis` and the velocity in `angularVelocity` in radians/second.

3.4.3.6. cfxExportZoneMotionAction

```
int cfxExportZoneMotionAction(const int zone, const int flag)
```

Specify whether grid coordinates and variables should have the appropriate rotation applied to them if the zone is rotating so that grid coordinates appear in their correct locations and velocities (for examples) take this rotation into consideration. If `cfxExportZoneList` and `cfxExportVariableList` should return rotated values, flag should be set to `cfxMOTION_USE`. The default behavior for a particular zone will be used if `cfxMOTION_IGNORE` is specified or this function isn't called. If zone is not valid or flag is not `cfxMOTION_USE`, `cfxMOTION_IGNORE` the return value will be -1 otherwise 0 is returned.

3.4.4. Node Routines

Accessing nodes within the current zone (see *cfxExportZoneSet* (p. 53)) is performed by making calls to the following functions.

It should be noted that the nodes for a zone are not loaded into the Export API until either `cfxExportNodeList` (see *cfxExportNodeList* (p. 54)) or `cfxExportNodeGet` (see *cfxExportNodeGet* (p. 55)) are called. This reduces memory overheads in the API by not allocating space until required.

When access to nodes in the current zone is no longer required, a call to `cfxExportNodeFree` (see *cfxExportNodeFree* (p. 55)) should be made to deallocate any internal storage.

3.4.4.1. cfxExportNodeCount

```
int cfxExportNodeCount ()
```

Query the number of nodes defined in the current zone.

3.4.4.2. cfxExportNodeList

```
cfxNode *cfxExportNodeList ()
```

Return a pointer to an array of cfxNode elements (see *cfxnode* (p. 22)) containing the coordinate values of each node in the current zone. The first node in the zone is the first element of the array, the second the second and so on.

The memory allocated to represent this information should be deallocated using `cfxExportNodeFree` (see *cfxExportNodeFree* (p. 55)) when no longer required.

3.4.4.3. cfxExportNodeGet

```
int cfxExportNodeGet (int nodeid, double *x, double *y, double *z)
```

Query the coordinates of a specific node in the current zone.

The index (nodeid) is specified between 1 and the number of nodes returned by `cfxExportNodeCount` (see *cfxExportNodeCount* (p. 54)). If the value of nodeid is out of range the return value is 0 otherwise it is nodeid.

3.4.4.4. cfxExportNodeFree

```
void cfxExportNodeFree ()
```

Deallocate any internal storage allocated by the Export API after calls to `cfxExportNodeList` (see *cfxExportNodeList* (p. 54)) and `cfxExportNodeGet` (see *cfxExportNodeGet* (p. 55)) have been made in the current zone.

3.4.5. Element Routines

Accessing elements within the current zone (see *cfxExportZoneSet* (p. 53)) is performed by making calls to the following functions. It should be noted that the elements for a zone are not loaded into the Export API until either `cfxExportElementList` (see *cfxExportElementList* (p. 55)) or `cfxExportElementGet` (see *cfxExportElementGet* (p. 56)) are called. This reduces memory overheads in the API by not allocating space until required.

When access to elements in the current zone is no longer required a call to `cfxExportElementFree` (see *cfxExportElementFree* (p. 56)) should be made to deallocate any internal storage.

3.4.5.1. cfxExportElementCount

```
int cfxExportElementCount ()
```

Query the number of elements defined in the current zone.

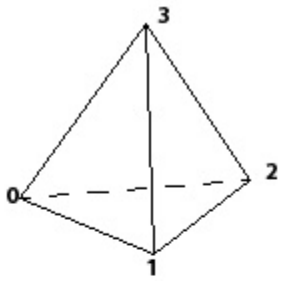
3.4.5.2. cfxExportElementList

```
cfxElement *cfxExportElementList ()
```

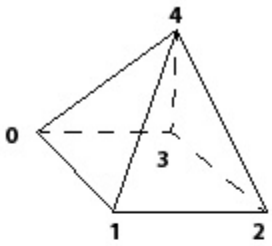
Return a pointer to an array of cfxElement elements (see *cfxelem* (p. 23)) containing the type and vertices of each element in the current zone. The first element in the zone is the first element of the array, the second the second and so on.

The memory allocated to represent this information should be deallocated using `cfxExportElement-Free` (see *cfxExportElementFree* (p. 56)) when no longer required.

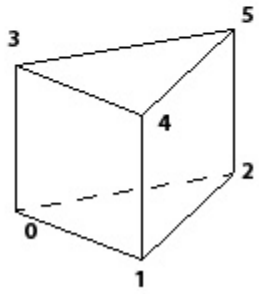
The following diagrams show the order of the nodes and connections that ANSYS CFX uses for exporting elements:



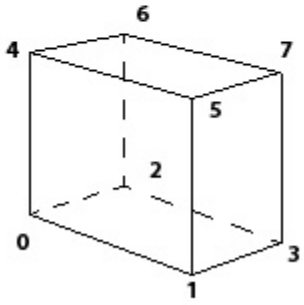
Tetrahedral element



Pyramid element



"Wedge" (prism) element



Hexahedral element

Note

The vertex ordering for the import API is different. For details, see *cfxImportElement* (p. 16).

3.4.5.3. cfxExportElementGet

```
int cfxExportElementGet (int elemid, int elemtype, int *nodelist)
```

Query the type and vertices of a specific element in the current zone.

The index (elemid) is specified between 1 and the number of elements returned by `cfxExportElementCount` (see *cfxExportElementCount* (p. 55)). If the value of elemid is out of range the return value is 0 otherwise it is elemid.

The type of the element is returned in elemtype and the vertices defining the element in nodelist. Note that nodelist must be large enough to hold the element number of vertices in the element (normally an array of 8 integers is used as this allows space enough for all element types to be handled).

3.4.5.4. cfxExportElementFree

```
void cfxExportElementFree ()
```

Deallocates any internal storage allocated by making calls to `cfxExportElementList` (see *cfxExportElementList* (p. 55)) or `cfxExportElementGet` (see *cfxExportElementGet* (p. 56)).

3.4.6. Region Routines

Regions are groups of faces in an ANSYS CFX results file. Accessing regions within the current zone (see *cfxExportZoneSet* (p. 53)) is performed by making calls to the following functions. It should be noted that the region information is not loaded into the Export API until either `cfxExportRegionList`

(see *cfxExportRegionList* (p. 57)) or `cfxExportRegionGet` (see *cfxExportRegionGet* (p. 57)) are called. This reduces memory overheads in the API by not allocating space until required.

When access to region in the current zone is no longer required a call to `cfxExportRegionFree` (see *cfxExportRegionFree* (p. 58)) should be made to deallocate any internal storage.

3.4.6.1. cfxExportRegionCount

```
int cfxExportRegionCount ()
```

Query the number of regions defined in the current zone.

3.4.6.2. cfxExportRegionSize

```
int cfxExportRegionSize (int regnum, int type)
```

Query the number of faces (if type is `cfxREG_FACES`) or nodes (if type is `cfxREG_NODES`) defined in the region identified by `regnum` in the current zone.

The function returns the number of faces or nodes in the current zone or 0 if either `regnum` is out of range or type is invalid.

3.4.6.3. cfxExportRegionName

```
char *cfxExportRegionName (int regnum)
```

Query the name of the region in the current zone identifies by regnum.

The function returns the name of the region or NULL if the region number supplied is out of range. The pointer returned points to static storage, which will be overwritten by the next call to `cfxExport-RegionName`.

3.4.6.4. cfxExportRegionList

```
int *cfxExportRegionList (int regnum, int type)
```

Query the nodes (type is `cfxREG_NODES`) or faces (`cfxREG_FACES`) that define a region. This function returns a pointer to an array of node ids or face ids that define the region identified by regnum or NULL if the region number is out of range or the type is not recognized. If type is specified as `cfxREG_FACES`, the returned ids will represent faces. The element number and local element face number may be extracted from each face id returned by using the macros `cfxELEMNUM` and `cfxFACENUM`. The node numbers for the face may be obtained by calling `cfxExportFaceNodes`. For details, see *cfxExportFaceNodes* (p. 58).

3.4.6.5. cfxExportRegionGet

```
int cfxExportRegionGet (int regnum, int type, int index, int *id)
```

Query the index'th element (type is `cfxREG_ELEM`) or index'th node (type is `cfxREG_NODE`) that defines a the region regnum in the current zone.

If regnum is out of range or type is not recognized or index is out of range, 0 is returned.

Otherwise id will contain the id of the appropriate node or face defining the region and the function will return index.

If type is specified as cfxREG_FACES, the returned id will represent the identity of a face. The element number and local element face number may be extracted from the id by using the macros cfxELEMNUM and cfxFACENUM.

3.4.6.6. cfxExportRegionFree

```
void cfxExportRegionFree (int regnum)
```

Deallocate any internal data storage associated with the region defined by regnum.

3.4.7. Face Routines

Faces are 2 dimensional (2D) units of mesh. Each global face ID is returned from cfxExportBoundaryList (see *cfxExportBoundaryList* (p. 61)) or cfxExportRegionList (see *cfxExportRegionList* (p. 57)).

Within CFX faces are either represented as Triangles (three vertices) or Quadrilaterals (two vertices). Each face in a CFX .res file will be parented by a single 3D element. The parent element of a face can be returned by the cfxELEMNUM macro with the global face ID, and the local face of that element can be determined by calling cfxFACENUM with the same global face ID

3.4.7.1. cfxExportFaceNodes

```
int cfxExportFaceNodes (int faceid, int *nodes)
```

Requests the vertices for the face identified by faceid. The argument faceid should be constructed from the element number and local face number using the following formula:

```
(element_number << 3) & local_face_number
```

Values returned from cfxExportRegionGet and cfxExportRegionList can be supplied directly to this function.

The number of vertices defining the face are returned if faceid is valid, otherwise 0 is returned. The node numbers are returned in the array nodes, which should be dimensioned to a minimum size of 4 in the calling routine.

The face numbers and associated node indices are tabulated here:

Element Type	Face	Nodes			
tetrahedron	1	0	1	2	
	2	0	3	1	
	3	1	3	2	
	4	0	2	3	
pyramid	1	0	3	4	
	2	1	4	2	
	3	0	4	1	
	4	2	4	3	
	5	0	1	2	3
prism	1	0	2	5	3
	2	0	3	4	1

Element Type	Face	Nodes			
	3	1	4	5	2
	4	0	1	2	
	5	3	5	4	
hexahedron	1	0	2	6	4
	2	1	5	7	3
	3	0	4	5	1
	4	2	3	7	6
	5	0	1	3	2
	6	4	6	7	5

Note

The face numbers and associated node indices are different when importing elements. For details, see *cfxImportGetFace* (p. 18).

3.4.8. Volume Routines

Volumes are groups of elements in a CFX results file. Accessing volumes within the current zone (see *cfxExportZoneSet* (p. 53)) is performed by making calls to the following functions. It should be noted that the volume definitions for a zone are not loaded into the Export API until either `cfxExportVolumeList` (see *cfxExportVolumeList* (p. 60)) or `cfxExportVolumeGet` (see *cfxExportVolumeGet* (p. 60)) are called. This reduces memory overheads in the API by not allocating space until required.

When access to volume information in the current zone is no longer required a call to `cfxExportVolumeFree` (see *cfxExportVolumeFree* (p. 60)) should be made to deallocate any internal storage.

3.4.8.1. cfxExportVolumeCount

```
int cfxExportVolumeCount ()
```

Query the number of volumes defined in the current zone.

3.4.8.2. cfxExportVolumeSize

```
int cfxExportVolumeSize (int volnum, int type)
```

Query the number of nodes (if type is `cfxVOL_NODES`) or number of elements (if type is `cfxVOL_ELEMS`) defining the volume indexed by `volnum` in the current zone. The return value will be 0 if `volnum` is out of range or type is invalid.

3.4.8.3. cfxExportVolumeName

```
char *cfxExportVolumeName (int volnum)
```

Query the name of the volume in the current zone indexed by `volnum`. Returns `NULL` if the `volnum` is out of range.

Note

The returned pointer points to internal storage, which will be overwritten by the next call to `cfxExportVolumeName`.

3.4.8.4. cfxExportVolumeList

```
int *cfxExportVolumeList (int volnum, int type)
```

Query the nodes (type is `cfxVOL_NODES`) or elements (`cfxVOL_ELEMS`) that define a volume.

This function returns a pointer to an array of node ids or element ids that define the volume identified by `volnum` or `NULL` if the volume number is out of range or the type is not recognized.

3.4.8.5. cfxExportVolumeGet

```
int cfxExportVolumeGet (int volnum, int type, int index, int *id)
```

Query the [index]th element (type is `cfxVOL_ELEM`) or [index]th node (type is `cfxVOL_NODE`) that defines a the volume `volnum` in the current zone.

If volnum is out of range or type is not recognized or index is out of range, 0 is returned.

Otherwise `id` will contain the id of the appropriate node or element in defining the volume and the function will return `index`.

3.4.8.6. cfxExportVolumeFree

```
void cfxExportVolumeFree (int volnum)
```

Deallocate any internal data storage associated with the volume defined by `volnum`.

3.4.9. Boundary Condition Routines

Boundary condition are located on groups of faces in a CFX results file. Accessing boundary condition locations within the current zone (see *cfxExportZoneSet* (p. 53)) is performed by making calls to the following functions. It should be noted that the boundary condition location information is not loaded into the Export API until either `cfxExportBoundaryList` (see *cfxExportBoundaryList* (p. 61)) or `cfxExportBoundaryGet` (see *cfxExportBoundaryGet* (p. 61)) are called. This reduces memory overheads in the API by not allocating space until required. When access to regions in the current zone are no longer required a call to `cfxExportBoundaryFree` (see *cfxExportBoundaryFree* (p. 62)) should be made to deallocate any internal storage.

3.4.9.1. cfxExportBoundaryCount

```
int cfxExportBoundaryCount ()
```

Query the number of boundary conditions defined in the current zone.

The function returns the number of boundary conditions in the current zone.

3.4.9.2. cfxExportBoundaryName

```
const char *cfxExportBoundaryName (const int bcidx)
```

Query the name of the boundary condition in the current zone identified by `bcidx`.

The function returns the name of the boundary condition or NULL if the bcidx supplied is out of range.

The pointer returned points to static storage, which will be overwritten by the next call to cfxExportBoundaryName.

Note

The following routines use bcidx which must lie between 1 and cfxExportBoundaryCount() and use index which must lie between 1 and cfxExportBoundarySize (bcidx, type).

3.4.9.3. cfxExportBoundaryType

```
const char *cfxExportBoundaryType (const int bcidx)
```

Query the type (for example, Inlet, Outlet etc.) of the boundary condition in the current zone identified by bcidx.

The function returns the type of the boundary condition or NULL if the bcidx supplied is out of range.

The pointer returned points to static storage, which will be overwritten by the next call to cfxExportBoundaryType.

3.4.9.4. cfxExportBoundarySize

```
int cfxExportBoundarySize (const int bcidx, const int type)
```

Query the number of faces (if type is cfxREG_FACES) or nodes (if type is cfxREG_NODES) defined in the boundary condition identified by bcidx in the current zone.

The function returns the number of faces or nodes or 0 if either bcidx is out of range or type is invalid.

3.4.9.5. cfxExportBoundaryList

```
int *cfxExportBoundaryList (const int bcidx, const int type)
```

Query the faces (if type is cfxREG_FACES) or nodes (if type is cfxREG_NODES) that define a boundary condition.

This function returns a pointer to an array of node ids or face ids that define the location of the boundary condition identified by bcidx or NULL if bcidx is out of range or the type is not recognized. If type is specified as cfxREG_FACES, the returned ids will represent faces. The element number and local element face number may be extracted from each face id returned by using the macros cfxELEMNUM and cfxFACENUM respectively. The node numbers for the face may be obtained by calling cfxExportFaceNodes. For details, see *cfxExportFaceNodes* (p. 58).

The returned pointer points to static data which should be destroyed using cfxExportBoundaryFree. Subsequent calls to cfxExportBoundaryList will overwrite the array.

3.4.9.6. cfxExportBoundaryGet

```
int cfxExportBoundaryGet (const int bcidx, const int type, const int index, int *id)
```

Query the index'th face (type is cfxREG_FACES) or index'th node (type is cfxREG_NODES) that defines the boundary condition location indexed by bcidx in the current zone. If bcidx is out of range or type is not recognized or index is out of range (not between 1 and cfxExportBoundarySize), 0 is returned. Otherwise id will contain the identifier of the appropriate node or face defining the boundary condition location and the function will return index.If type is specified as cfxREG_FACES, the returned id will represent the identity of a face. The element number and local element face number may be extracted from the id by using the macros cfxELEMNUM and cfxFACENUM respectively.

3.4.9.7. cfxExportBoundaryFree

```
void cfxExportBoundaryFree (const int bcidx)
```

Deallocate any internal data storage associated with the boundary condition defined by bcidx.

3.4.10. Variable Routines

These routines access the variable data defined on the current zone as defined by cfxExportZoneSet. For details, see *cfxExportZoneSet* (p. 53). The variable data arrays are not loaded into memory until either cfxExportVariableList or cfxExportVariableGet are called, and remain in memory until cfxExportVariableFree is called. For details, see:

- *cfxExportVariableList* (p. 63)
- *cfxExportVariableGet* (p. 63)
- *cfxExportVariableFree* (p. 63).

3.4.10.1. cfxExportVariableCount

```
int cfxExportVariableCount(int usr_level)
```

Query the number of variables at interest level usr_level or below. If usr_level is 0, then the total number of variables is returned.

3.4.10.2. cfxExportVariableSize

```
int cfxExportVariableSize (int varnum, int *dimension, int *length, int *bdnflag)
```

Query the dimension, dimension, and length, length, for the variable identified by varnum, which should be from 1 to the number of variables, returned by cfxExportVariableCount (*cfxExportVariableCount* (p. 62)). The length, length, will either be 1 or the same as the number of nodes returned by cfxExportNodeCount (see *cfxExportNodeCount* (p. 54)). If 1, then the variable has meaningful values only at the boundary nodes, with a constant value in the interior.

The function also returns bdnflag which indicates if the variable contains corrected boundary node values (1) or not (0).

The function returns varnum if successful, or 0 if the variable number is out of range.

3.4.10.3. cfxExportVariableName

```
char *cfxExportVariableName (int varnum, int alias)
```

Query the name of the variable identified by varnum.

The return value of the function is NULL if the variable number is out of range or the name of the variable.

The pointer returned points to static storage, which will be overwritten by the next call to `cfxExportVariableName`.

The argument `alias` indicates whether the short name (`alias=0`) or long name (`alias=1`) should be returned. For example, the short and long names for the total temperature variable are `TEMPTOT` and `Total Temperature`, respectively.

3.4.10.4. cfxExportVariableList

```
float *cfxExportVariableList (int varnum, int correct)
```

Query the results data for a variable identified by `varnum`.

Returns `NULL` if the variable number is out of range or the variable data if successful.

The flag `correct` indicates whether to correct boundary node data (correct=1) or not (correct=0), assuming that it exists.

The data is in the same order as the nodes returned from `cfxExportNodeList` (see *cfxExportNodeList* (p. 54)).

For multidimensional variables, the data is stored with dimension consecutive values for each node.

The storage for the data is created by the Export API when this function is called. When the data is no longer required a call to `cfxExportVariableFree` (see *cfxExportVariableFree* (p. 63)) should be made by the calling function.

3.4.10.5. cfxExportVariableGet

```
int cfxExportVariableGet (int varnum, int correct, int index, float *value)
```

Request the values of the variable identified by varnum at the location given by `index`, which should be from 1 to the length of the variable, inclusively.

The flag `correct` indicates whether to correct boundary node data (`correct=1`) or not (`correct=0`), assuming that it exists.

The function returns `index`, or 0 if the location is out of range.

The values of the variable are returned in value which should be dimensioned at least as large as the dimension of the variable.

3.4.10.6. cfxExportVariableFree

```
void cfxExportVariableFree (int varnum)
```

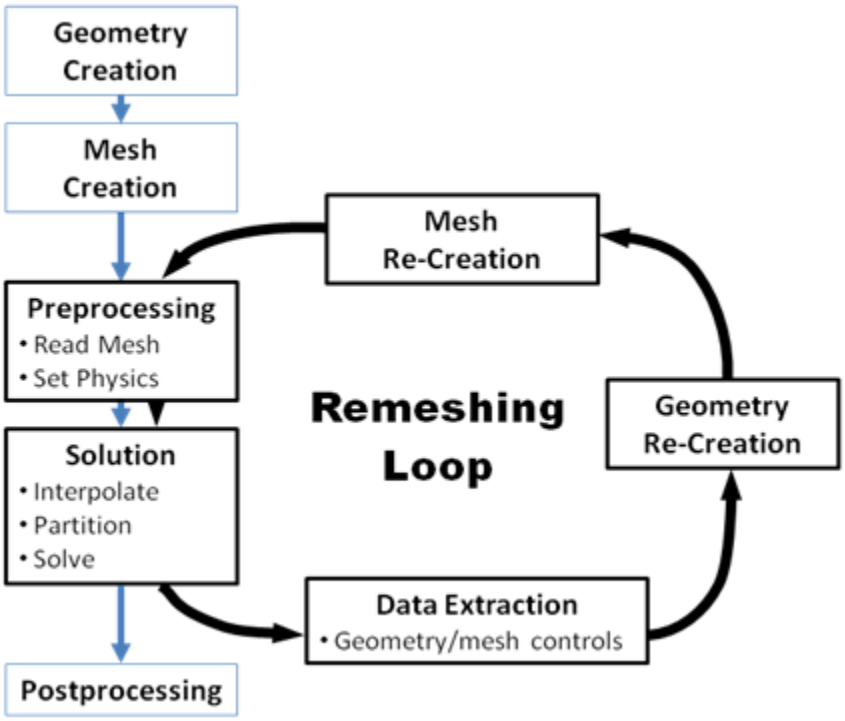
Deallocates the internal data storage for the variable identified by `varnum` for the current zone.

Chapter 4: Remeshing Guide

Periodic remeshing is an important part of running analyses that involve significant mesh deformation. Remeshing is often required simply to maintain acceptable mesh quality, as described in Discretization Errors in the *CFX-Solver Theory Guide* and Measures of Mesh Quality in the *CFX-Solver Modeling Guide*.

A schematic illustrating the integration of remeshing into the general simulation workflow is shown in the figure below.

Figure 4.1 Integration of a Remeshing Loop into the General Simulation Workflow



As shown in *Figure 4.1* (p. 65), in addition to the **Pre-processing** and **Solution** steps of the standard simulation workflow, the remeshing loop includes three additional steps:

- **Data Extraction**
- **Geometry Modification**
- **Mesh Recreation**.

In the context of remeshing, those steps are responsible for completing the following sub-steps;

- **Data Extraction**: Extract any data needed to guide geometry modifications and mesh re-creation from the most recent analysis results and monitor point values.
- **Geometry Re-Creation**: Update the analysis' geometry so that it conforms to that of the most recent analysis results (that is, account for mesh deformation).
- **Mesh Re-Creation**: Generate new mesh(es) that correspond to the updated geometry.

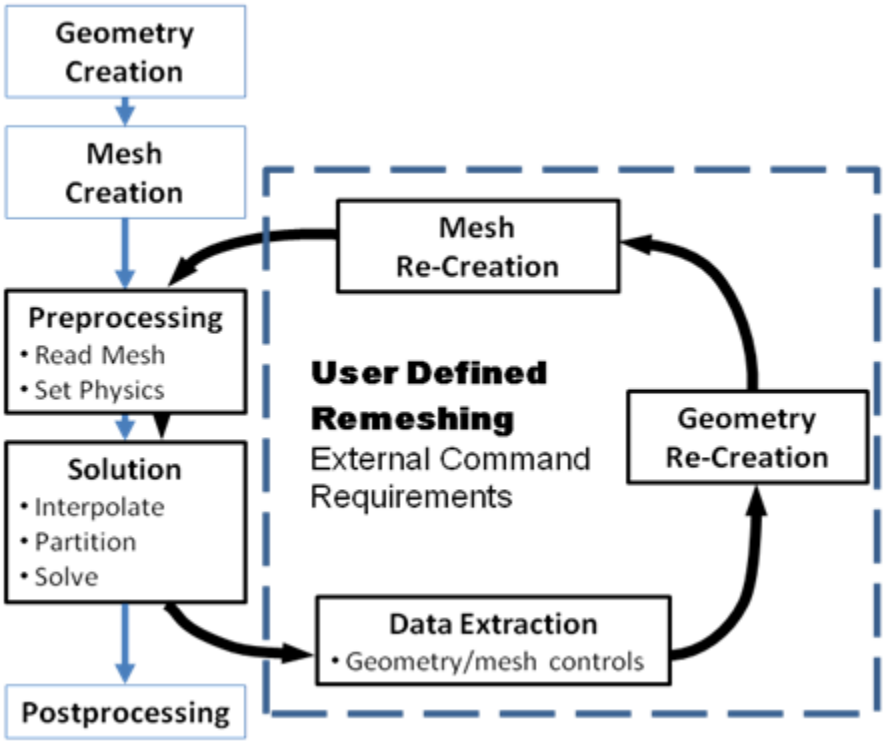
- **Pre-processing**: Insert the new mesh(es) into the analysis definition, and generate an updated CFX-Solver Input File.
- **Solution**: Interpolate the previously generated analysis results onto the new mesh, re-partition the mesh if a parallel run mode is selected, and continue the solution process.

As described in Remeshing Tab in the *CFX-Pre User's Guide*, there are two options available for remeshing: **User Defined** and **ICEM CFD Replay**. As outlined in the discussions that follow, the **Pre-processing** and **Solution** steps (and their respective sub-steps) are automatically executed for both remeshing options. Although the remaining steps are automatically executed for the **ICEM CFD Replay** remeshing option, they become the responsibility of a user defined external command for the **User Defined** remeshing option.

4.1. User Defined Remeshing

User Defined remeshing offers the greatest flexibility to customize the remeshing process. This comes at the expense of requiring that the data extraction, and geometry and mesh re-creation steps are executed by a user-specified external command, as illustrated in the figure below where the dashed line identifies steps that must be executed by the user-specified **External Command**.

Figure 4.2 Schematic for User Defined remeshing



This remeshing option is ideally suited for users who have previously completed an 'in-house' remeshing solution involving scripts or varying degrees of manual user-intervention. When this option is used, the following steps are automatically executed:

- Run the specified external command to generate a new mesh(es)
- Insert the new mesh(es) into the analysis definition, and generate an updated CFX-Solver Input file
- Interpolate the previously generated analysis results onto the new mesh, re-partition the mesh if a parallel run mode is selected, and continue the solution process.

The following examples outline the use of the **User Defined** remeshing option:

- *Remeshing with Key-Frame Meshes* (p. 67)
- *Remeshing with Automatic Geometry Extraction* (p. 67).

4.1.1. Remeshing with Key-Frame Meshes

In some analyses involving mesh deformation, the motion of various boundaries and sub-domains is known beforehand. Thus, the iteration or time step number at which unacceptable mesh quality will occur due to mesh motion is also known. A sequence of 'key-frame' meshes (of any mesh file type) corresponding to these instances of poor mesh quality can consequently be generated and applied during the analysis.

Once the sequence of key-frame meshes has been generated, they should be placed in a location that will be accessible during the analysis' execution. The analysis definition is then modified to include one or more control conditions that will interrupt the solver at the iteration or time step at which a key-frame mesh should be inserted. A configuration is subsequently defined (unless this has already been done), and a remeshing definition is created with the following settings:

- Set **Option** to **User Defined**.
- Set the **Activation Condition**(s) to the previously created interrupt control condition(s).
- Set the **Location** to the mesh region that will be replaced.
- Set the **External Command** to the command that will be used to generate the replacement mesh file.
- Set the **Replacement File** to the name of the file that will be generated by the external command.

The **External Command** is typically a shell script or batch file that completes the following tasks:

- Determine which key-frame mesh to use. This will require parsing the run's output file for the iteration or time step number, or the actual simulation time. Output generated from the `cfx5mondata` executable can also be parsed instead of the run's output file. For details, see Exporting Monitor Data from the Command Line in the *CFX-Solver Manager User's Guide*.
- Copy the key-frame mesh to the path the specified by the **Replacement File** setting.

4.1.2. Remeshing with Automatic Geometry Extraction

In some analyses involving mesh deformation, the motion of various boundaries and sub-domains is not known beforehand and the key-frame remeshing strategy presented above is not applicable. In these analyses, geometrical information must be extracted from the most recent analysis results and applied in the remeshing process.

In such cases, the analysis definition is modified to include one or more control conditions that will interrupt the solver when, for example, mesh quality deteriorates significantly. A configuration is subsequently defined (unless this has already been done), and a remeshing definition is created with the following settings:

- Set **Option** to **User Defined**.
- Set the **Activation Condition**(s) to the previously created interrupt control condition(s).
- Set the **Location** to the mesh region that will be replaced.

- Set the **External Command** to the command that will be used to generate the replacement mesh file.
- Set the **Replacement File** to the name of the file that will be generated by the external command.

The **External Command** is typically a shell script or batch file that completes the following tasks:

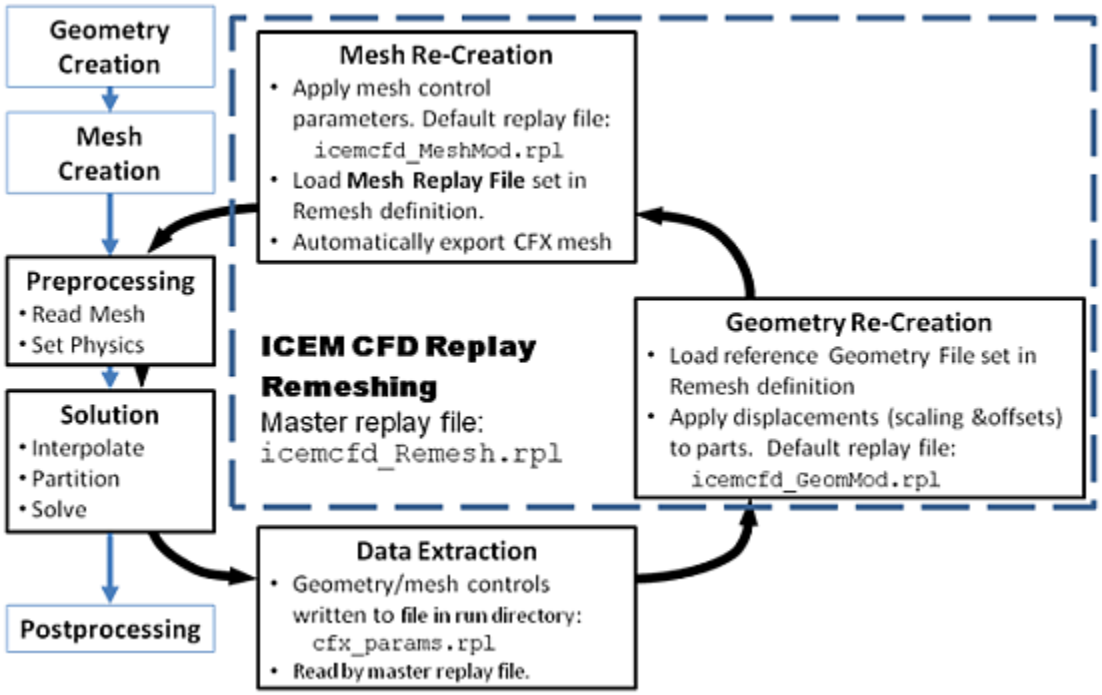
- Extract geometry data from the most recent solution of the analysis, and either update or replace the original geometry. This may be done using mesh-to-geometry conversion tools available in software such as ANSYS ICEM CFD, or by extracting monitor point data values (for example, the Total Centroid Displacement variable) using the `cfx5mondata` executable. For details, see Exporting Monitor Data from the Command Line in the *CFX-Solver Manager User's Guide*.
- Create a replacement mesh file using the updated or newly generated geometry. This may be done in any suitable mesh generation application.

Note that some mesh-to-geometry conversion tools are unable to extract the latest mesh coordinates from the most recent CFX-Solver Results file. If this is the case, then introduce a call to CFX-Pre (within the **External Command**) that executes a session file that simply loads the latest CFX-Solver Results file and writes a new CFX-Solver Input file. That CFX-Solver Input file will contain the required, latest mesh coordinates.

4.2. ICEM CFD Replay Remeshing

ICEM CFD Replay remeshing provides a highly automated remeshing process that is ideally suited for users of the ANSYS ICEM CFD mesh generation software and cases that involve translational mesh motion only (that is, no rotation or general deformation). When this option is used, a master replay file is assembled from other task-oriented replay files and submitted to the ANSYS ICEM CFD mesh generator for batch execution. These replay files are illustrated in the figure below, along with the general process flow for this remeshing option. The dashed line in the figure highlights components of the master replay file and identifies files and steps that you can modify. Unless otherwise noted, files are contained in the `<CFXROOT>/etc/Remeshing` directory.

Figure 4.3 Schematic for ICEM CFD Replay remeshing



When this option is used, the following steps are automatically executed:

- Extract geometry and mesh control data and write them to the `cfx_params.rpl` replay file in the run directory. The data includes:
  - Centroid displacements for boundaries that are included in ANSYS ICEM CFD Part Maps
  - Mesh control parameters (for example, `ehgt` and `emax`)
  - Scalar parameters.
- Run (in batch) the ANSYS ICEM CFD mesh generation program using the master replay file. This master replay file executes the following tasks:
  - Read the `cfx_params.rpl` file.
  - Load the reference geometry from the **Geometry File** identified in the **Remesh** definition.
  - Apply displacements (including scaling and any offsets) corresponding to all ANSYS ICEM CFD Part Map definitions contained in the **Remesh** definition. This is done using the default geometry replay file provided, or using the user defined replay file if specified in the ICEM CFD Geometry Control setting.
  - Apply **ICEM CFD Mesh Controls** defined in the **Remesh** definition. This is done using the provided controls, or using the user-defined replay file if specified in the **ICEM CFD Mesh Control** setting.
  - Load your **Mesh Replay File**, specified in the **Remesh** definition.
  - Export a new mesh for ANSYS CFX.
- Insert the new mesh(es) into the analysis definition, and generate an updated CFX-Solver Input file.
- Interpolate the previously generated analysis results onto the new mesh, re-partition the mesh if a parallel run mode is selected, and continue the solution process.

You must create the reference **Geometry File** and the **Mesh Replay File** because these are specific to each case. However, the generic default replay files (`icemcfd_Remesh.rpl`, `icemcfd_GeomMod.rpl`, and `icemcfd_MeshMod.rpl`) used by this option are provided in the `<CFXROOT>/etc/Remeshing` directory. These files may be edited to provide installation-wide changes to the ICEM CFD Replay remeshing behavior. Alternatively, the geometry and mesh modification files may be copied and edited to provide case-specific changes.

Note

As indicated previously, only translational mesh motion is automatically handled by the ICEM CFD Replay remeshing option. This is accomplished by applying the displacements of centroids of boundaries in the ANSYS CFX analysis definition to parts in the ANSYS ICEM CFD geometry. All other mesh motion (such as rotation about the centroid or another point, or general deformation) will not be applied, and an inconsistency in the analysis geometry before and after remeshing will be introduced.

4.2.1. Steps to Set Up a Simulation Using ICEM CFD Replay Remeshing

The following discussion presents the three general steps required to set up a simulation using the ICEM CFD Replay remeshing option.

The first step involves creating the reference **Geometry File** within the ANSYS ICEM CFD environment. If the geometry was not created within that environment, use one of the **File** > **Import Geometry** options in the ANSYS ICEM CFD environment. At this point, ensure that all required Parts (or Families) are defined and named so that they can be referenced when completing the ICEM CFD Replay remeshing definition later in CFX-Pre. Finally, store the geometry in the ICEM CFD native geometry file format (namely, a `.tin` file).

The second step involves generating the **Mesh Replay File**, again, from within the ANSYS ICEM CFD environment. Start with the previously created geometry loaded, and work sequentially through the mesh generation process until acceptable mesh controls have been specified. This may require fine tuning, which will involve the regeneration of your mesh after moving the geometry through its expected range of motion. Once you are satisfied with the mesh control settings, purge the last mesh using **File** > **Mesh** > **Close Mesh**, and reload the original reference geometry. Complete the following tasks to generate the required **Mesh Replay File**:

1. Use **File** > **Replay Scripts** > **Replay Control** to begin recording the commands for the **Mesh Replay File**. The **Replay Control** dialog box is displayed.
2. Revisit all of the mesh related tabs and settings used to generate the mesh, clicking either the **Apply** or **OK** to commit the settings into the **Replay Control** panel.
3. Generate the mesh.
4. In the **Replay Control** panel, clear the **Record (after current)** toggle and select **Save** to write the settings to replay file.

You may also want to export the mesh that was (re)generated for use in the simulation definition (as in the next step).

The third step involves defining the simulation within CFX-Pre. Complete the following tasks to prepare the simulation:

1. Start a new simulation and import the (previously generated) mesh.

2. Define expressions for the motion of the geometry (for example, see the expressions for the ball movement in Modeling a Ball Check Valve using Mesh Deformation and the CFX Rigid Body Solver in the *CFX Tutorials*).

   Note

   See also the discussion in *Mesh Re-Initialization During Remeshing* (p. 72).

3. Define the flow analysis including the definition of one or more solver interrupt controls, as described in Interrupt Control in the *CFX-Pre User's Guide*, to identify the condition(s) under which solver execution will be interrupted.
4. Define a configuration and complete the ICEM CFD Replay remeshing setup as described in ANSYS ICEM CFD Replay Remeshing in the *CFX-Pre User's Guide*. The **Geometry File** and **Mesh Replay File** created above are referenced here. Note also, that references to one or more of the previously defined solver interrupt control conditions are required to activate remeshing.
5. Complete any execution controls for the simulation and either start the solver or write the CFX-Solver Input file for later use.

4.3. Directory Structure and Files Used During Remeshing

CFX-Solver runs that include remeshing will have a slightly non-standard directory structure during execution. For example, using a CFX-Solver input file named `case.def`, a directory structure similar to the following will exist just after solution execution is interrupted and the second instance of remeshing begins:

```
case.def
case_001/
      1_full.trn
      0_full.trn
      2_full.trn
      3_oldmesh.res
      3_remesh.out
 case_001.dir/
      3_full.trn
      4_full.trn res mon
```

The first instance of remeshing occurred when the solver was interrupted after the third time step. Following this instance of remeshing, all CFX-Solver Results files (such as transient, backup, and remeshing) contained in the run directory, `case_001.dir`, were moved into the final solution directory, `case_001`. The results file written when the solver was interrupted before remeshing was renamed to `3_oldmesh.res`. Any text output to the console window during remeshing was redirected to the file named `3_remesh.out`, which is also placed in the final solution directory.

The second, and currently running, instance of remeshing began when the solver was interrupted after the fifth time step. The results file written by the solver still has the generic name, `res`, and monitor data (contained in the `mon` file) has not yet been inserted into the results file.

Just after inserting the new mesh(es) into the analysis definition, the files contained in the final solution and run directories change slightly. The results and console output files are renamed (to `5_oldmesh.res` and `5_remesh.out`, respectively) and moved from the run directory into the final solution directory. An automatically generated session file, `meshUpdate.pre`, is used by CFX-Pre to generate the updated the solver input file, `5_newmesh.def`, and each of these files are present in the run directory. These files are, however, replaced or removed during the next instance of remeshing or when the analysis ends and the run directory is deleted.

4.4. Additional Considerations

This section discusses additional considerations for remeshing.

4.4.1. Mesh Re-Initialization During Remeshing

The following points are important to note during remeshing:

- The total mesh displacement variable is relative to a specific mesh topology. Because the mesh topology changes, this variable is reset each time remeshing occurs.
- The new variable called `total centroid displacement` tracks the displacement of each boundary's centroid since the beginning of the analysis (that is, relative to the original mesh).
- The specified displacement based mesh motion is relative to the initial mesh and must therefore include an offset to account for mesh re-initialization. The `Mesh Initialisation Time` variable corresponds to the time at which mesh re-initialization last occurred. This can be used to evaluate the required offset for time varying mesh displacement.

An example of the expressions used to evaluate an applied displacement that includes the required offset to account for mesh re-initialization is given below. In this example, the applied displacement is evaluated as the desired displacement minus the value of the desired displacement at the `Mesh Initialisation Time`.

```
Disp Desired = 1[m]*0.5*(1-cos(2.[s^-1]*pi*t))
Disp Mesh ReInit = 1[m]*0.5*(1-cos(2.[s^-1]*pi*Mesh Initialisation Time ))
Disp Applied = Disp Desired - Disp Mesh ReInit
```

4.4.2. Software License Handling

Several software components (for example, CFX-Pre, the CFX-Solver, ANSYS ICEM CFD, and so on) are used while executing steps in the overall remeshing process. Rather than holding all of these licenses for the entire duration of the analysis, they are only 'checked out' as required. Although this frees up the licenses for other users when remeshing is not executing, it also introduces the possibility that required licenses are not available when they are needed for remeshing.

This model for software license handling may cause problems in multi-user environments, but work is underway to provide a broader range of handling options for future releases.

Chapter 5: Reference Guide for Mesh Deformation and Fluid-Structure Interaction

This guide is part of a series that provides advice for using CFX in specific engineering application areas. It is aimed at users with little or moderate experience using CFX for applications involving Mesh Deformation and/or Fluid Structure Interaction.

This guide describes:

5.1. Mesh Deformation

Mesh deformation is an important part of executing simulations with changing domain geometry. In CFX, this capability is available in fluid and solid domains. Motion can be specified on selected regions via CEL or an external solver coupling (for example, ANSYS Multi-field MFX), or on all nodes in an entire domain via a user-Fortran junction box routine.

5.1.1. Mesh Folding: Negative Sector and Element Volumes

It is not uncommon for the mesh to become folded (or tangled) during the mesh deformation process. When this occurs, a message indicating the existence and location of either negative sector volumes or negative (that is, topologically invalid) elements is written to the simulation output file. Notification of negative sector volumes highlights the existence of non-convex mesh elements that still have a positive volume. Although the existence of negative sector volumes is not a fatal condition, it does indicate that:

- Mesh elements are only barely positive
- Further mesh deformation is likely to yield elements with negative volumes, which is a fatal condition

Some of the most common causes for mesh folding during deformation are identified in the following sections.

5.1.2. Applying Large Displacements Gradually

In many simulations that require mesh deformation, the motion is known a priori. In these cases, the motion can be applied gradually, by relating it to the iteration or timestep counters, to reduce the likelihood of mesh folding. Mesh folding is often avoided with this strategy because the mesh displacement equations are assembled using the updated meshes from each deformation step (that is, outer iteration or timestep). In general, the desired total mesh deformation should be split up so that regions where motion is specified move through less than approximately 5 adjacent elements per step.

In some simulations, the motion is not known a priori. Fluid Structure Interaction is an excellent example of this. In these cases, mechanisms available for under-relaxing the displacements applied per deformation step should be used. For details, see Solver Controls, External Coupling Tab in the *CFX-Solver Modeling Guide*.

5.1.3. Consistency of Mesh Motion Specifications

Mesh motion options such as `Specified Displacement` may be applied on multiple boundary and subdomain regions. Because the specified motion is applied directly to mesh nodes, rather than control volume integration points, care is required to ensure that motion specified on adjacent regions is self-consistent. For example, the motion specified on one moving wall should be reduced to zero for any nodes that are shared with another stationary wall. If this is not done, then the motion applied to the shared nodes will be either the moving or stationary condition, depending on which was applied last during the equation assembly process.

Folded meshes often result from the application of inconsistent motion specifications.

5.1.4. Solving the Mesh Displacement Equations and Updating Mesh Coordinates

During each outer iteration or timestep, the mesh displacement equations are solved to the specified convergence level and the resulting displacements are applied to update the mesh coordinates. This occurs before proceeding to solve the general transport (for example, hydrodynamics, turbulence, etc.) equations.

Unlike other equation classes, the convergence level (that is, controls and criteria) applied to mesh displacement equations is unaffected by changes made to the basic settings for all other equations. The default convergence controls and criteria for the mesh displacement equation are tabulated below, and are changed by visiting the **Mesh Displacement** entry in the **Equation Class Settings** tab under **Solver Control**.

Setting	Value
Maximum Number of Coefficient Loops	5
Minimum Number of Coefficient Loops	1
Residual Type	RMS
Residual Target	1.0E-4

Mesh folding occurs and is detected when the displacements are used to update the mesh coordinates. Folded meshes can occur if the displacement equations are incompletely solved. In this case, the unconverged displacement solution field does not vary smoothly enough to ensure that adjacent mesh nodes move by similar amounts.

5.1.5. Mesh Displacement Diffusion Scheme

A number of numerical schemes are available for the solution of the mesh displacement diffusion equation. In many cases the default scheme is appropriate, but in some situations mesh folding can be avoided and improved mesh quality can be obtained by using a different scheme. The expert parameter `meshdisp diffusion scheme` controls which numerical scheme is used. For details, see Discretization Parameters. Setting `meshdisp diffusion scheme` to 3 can be helpful if the mesh folds unexpectedly at sharp corners or if uniform mesh deformation is expected but non-uniform deformation is observed. The figures below show an example of using different mesh displacement diffusion schemes.

Figure 5.1 Original Undeformed Mesh

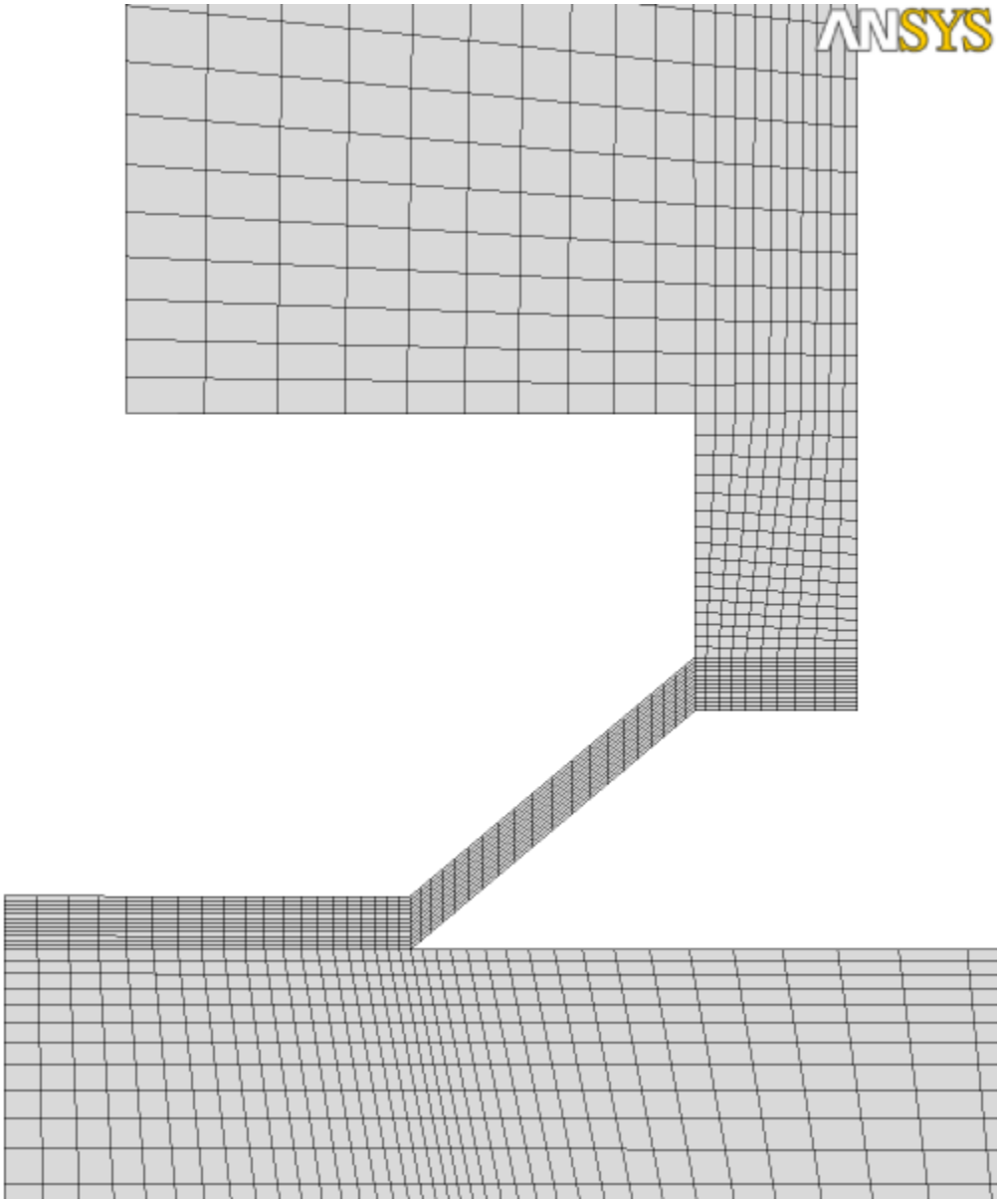


Figure 5.2 Deformed Mesh with the Default Diffusion Scheme

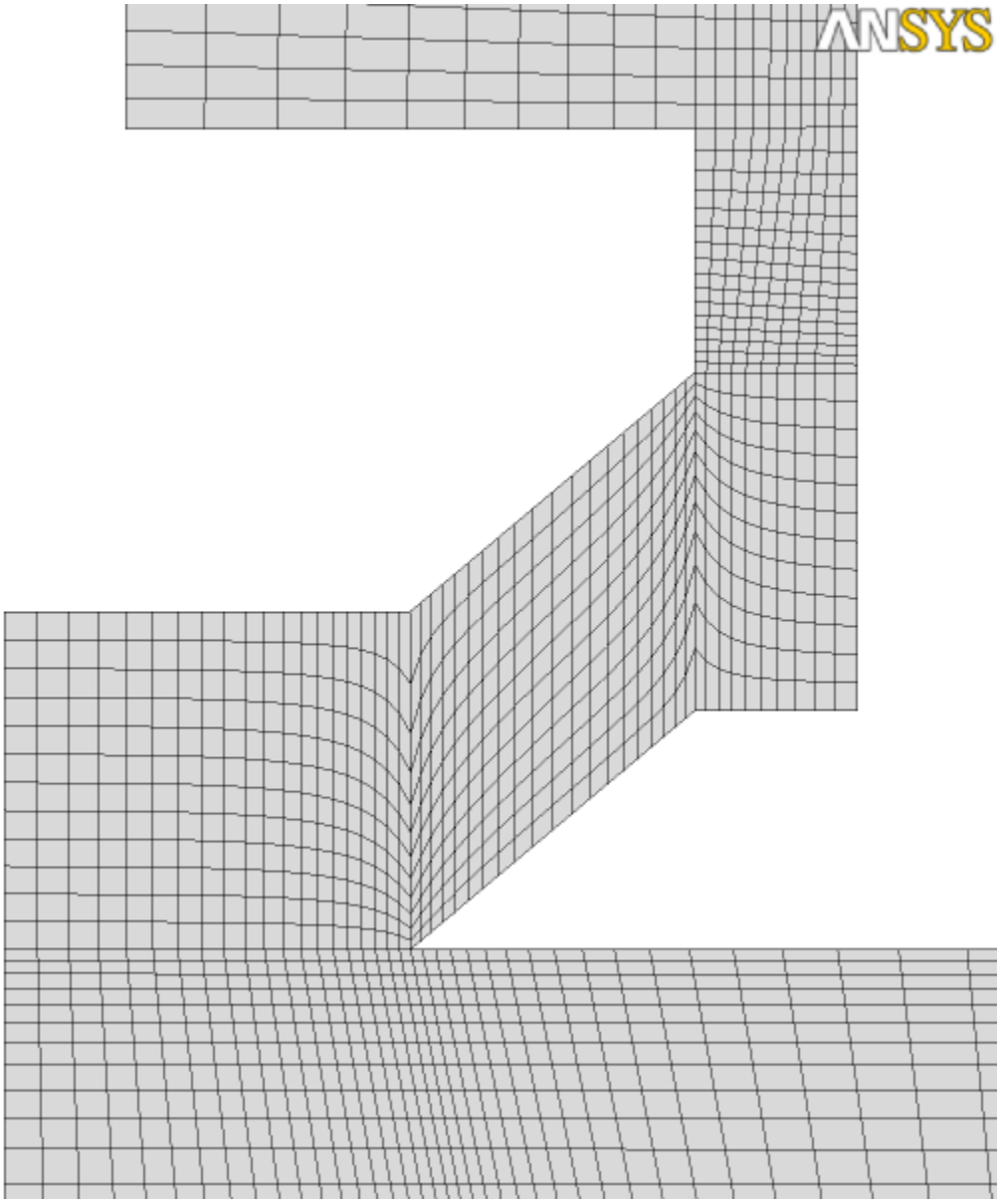
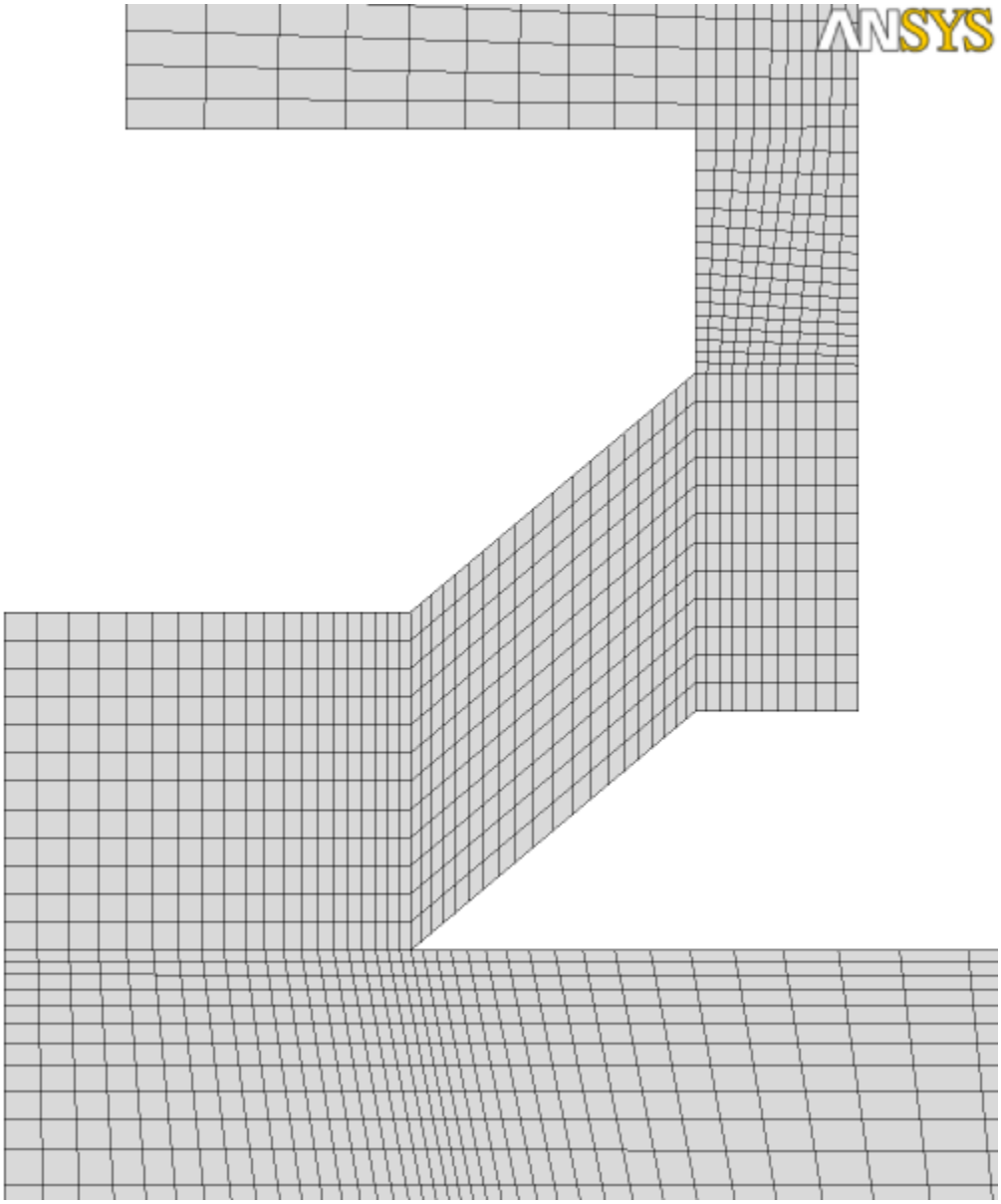


Figure 5.3 Deformed Mesh with "meshdisp diffusion scheme = 3"



5.1.6. Mesh Displacement vs. Total Mesh Displacement

A number of new variables become available when executing simulations with mesh deformation. Two of these variables are `Mesh Displacement` and `Total Mesh Displacement`.

`Mesh Displacement` is the principal variable that is solved for by the mesh motion model (see Mesh Deformation in the *CFX-Solver Modeling Guide*). This variable represents the displacement relative to the previous mesh locations. Conversely, `Total Mesh Displacement` is a derived quantity that represents the displacement relative to the initial mesh.

5.1.7. Simulation Restart Behavior

The following table summarizes the behavior that occurs when simulations with (or without) mesh deformation are restarted with (or without) mesh deformation. With only the exception noted, the simulation type (that is, steady state or transient) used for the initial or restart run does not affect behavior.

Initial Simulation	Restart Simulation	Restart Behavior
No Deformation	Deformation	Mesh from initial run serves as initial mesh for restart run
Deformation	No Deformation	Final mesh from initial run serves as mesh for restart run
Deformation	Deformation	Initial mesh from initial run serves as initial mesh for restart run[a]

[a]If the restart is a transient run with the initial time set to `Value`, then the final mesh from the initial run will serve as the initial mesh for the restart simulation.

5.2. Fluid Structure Interaction

CFX provides the ability to solve, or take part in the solution of cases that involve the coupling of solution fields in fluid and solid domains. This coupling is commonly referred to as Fluid Structure Interaction (FSI). One example of FSI is the simulation of an internal combustion engine, which involves the solution of fluid flow, conjugate heat transfer and combustion problems on deforming meshes.

In the discussion that follows, examples are presented to demonstrate the FSI capabilities using CFX by itself or with other CAE packages like ANSYS Mechanical and ANSYS Multiphysics. These examples are grouped according to the degree of coupling that must be maintained during the simulation in order to ensure that accurate results are obtained.

5.2.1. Unidirectional (One-Way) FSI

In many FSI simulations, the coupling between the solution fields is predominantly unidirectional; a given field may strongly affect, but not be affected by other fields. In CFX, there are a variety of strategies to efficiently execute such simulations. These strategies are identified in the following examples.

5.2.1.1. Using CFX Only

One of the most useful examples of unidirectional FSI within CFX involves prescribed mesh deformation of fluid or solid domains. This is possible using the CEL to specify the motion of sub-domains or domain boundaries, or by reading a sequence of pre-defined meshes.

5.2.1.2. Using CFX and the Mechanical Application

In many FSI simulations, the capabilities of additional solvers are required to compliment those of CFX. In these circumstances, CFX provides tools to facilitate the import and export of solution data in a variety of formats.

5.2.1.2.1. Importing Data from the Mechanical Application Solver

The recommended method for importing boundary condition data from the Mechanical application into CFX is via boundary profile data. For information about the creation and use of profile data files, refer to Chapter 6, Unidirectional Load Transfer and Use Profile Data in the *CFX-Pre User's Guide*.

5.2.1.2.2. Export Data to Other ANSYS Software Products

Two methods exist for exporting data from CFX for use in other ANSYS software products. The first method requires the use of the MFS variant of the ANSYS Multi-field solver and the second method does not.

- For information about exporting mechanical and thermal surface data and thermal volumetric data for use with the *MFS solver*, refer to Export to ANSYS Multi-field Solver Dialog Box in the *CFX-Solver Manager User's Guide*. For information about using the exported results and the MFS Solver, refer to Chapter 3, The ANSYS Multi-field (TM) Solver - MFS Single-Code Coupling.
- For information about exporting mechanical and thermal surface data for *general use*, refer to Mechanical Import/Export Commands in the *CFD-Post User's Guide*. This method involves reading a Mechanical Coded Database (CDB) file and interpolating CFX solution data onto the mesh contained in that file. For more information about how these steps are automated in the Mechanical application, see Custom Systems in the *Workbench User Guide*.

5.2.1.2.3. Mechanical Import/Export Example: One-Way FSI Data Transfer

You can perform one-way FSI operations manually (by exporting CDB files from the Mechanical APDL application, importing the surface in CFD-Post, and exporting the SFE commands).

To create a Mechanical load file using CFD-Post to transfer FSI data:

1. Load the fluids results file, from which you want to transfer results, into CFD-Post
2. Select **File** > **Import** > **Import Mechanical CDB Surface**. The **Import Mechanical CDB Surface** dialog box appears.
3. In the **Import Mechanical CDB Surface** dialog box, either:
   - Select the CDB file that specifies the surface mesh of the solid object to which to transfer data. Also select the **Associated Boundary** for the surface to map onto, and make other selections as appropriate.
   - Select the XML document that provides all transfer information. Click **OK**, and the surface data is loaded.
4. Select **File** > **Export** > **Export Mechanical Load File**. The **Export Mechanical Load File** dialog box appears.
5. In the **Export Mechanical Load File** dialog box, select a filename to which to save the data. For the **Location** parameter value, select the imported ANSYS mesh object. Under **File Format** select **ANSYS Load Commands (FSE or D)**. (Alternatively, you can select **WB Simulation Input (XML)** to get XML output.) Also select the appropriate data to export: Normal Stress Vector, Tangential Stress Vector, Stress Vector, Heat Transfer Coefficient, Heat Flux, or Temperature. Click **Save**, and the data file is created.

The one-way FSI data transfer described above is performed automatically when using the **FSI: Fluid Flow (CFX) > Static Structural** custom system in ANSYS Workbench. For details, see the FSI: Fluid Flow (CFX) > Static Structural in the *Workbench User Guide* section in the ANSYS documentation.

5.2.1.3. Using CFX and Other CAE Software

Solution data can be exported from CFX in a variety of general formats during or after execution of the CFX-Solver. For information about the export of data in CGNS format during the execution of the solver, refer to Export Results Tab in the *CFX-Pre User's Guide*. For information about the extraction and export

of CGNS, MSC Patran, FIELDVIEW, EnSight and custom data from CFX results files, refer to Generic Export Options in the *CFX-Solver Manager User's Guide*.

5.2.2. Bidirectional (Two-Way) FSI

In some simulations, there is a strong and potentially nonlinear relationship between the fields that are coupled in the Fluid Structure Interaction. Under these conditions, the ability to reach a converged solution will likely require the use of bidirectional FSI. As for unidirectional interaction, examples are provided below that demonstrate the variety of strategies to execute such simulations.

5.2.2.1. Using CFX Only

Conjugate heat transfer is an example of bidirectional interaction that can be solved using the CFX-Solver only.

5.2.2.2. Using CFX and the Mechanical Application

Communicating data between CFX and the Mechanical application is automated by the MFX branch of the ANSYS Multi-field solver. In this branch of the ANSYS Multi-field solver, data is communicated between the CFX and the Mechanical application field solvers through standard internet sockets using a custom client-server communication protocol. This custom solution maximizes execution efficiency and robustness, and greatly facilitates future extensibility.

Setup requires creation of the fluid and solid domain/physical models in the CFX-Pre and the Mechanical application user interfaces, respectively, and the specification of coupling data transfers and controls in the CFX-Pre user interface. Execution and run-time monitoring of the coupled simulation is performed from the CFX-Solver Manager. Note that a dedicated MFX-ANSYS/CFX tab is also provided in the ANSYS Product Launcher to begin execution of the coupled simulation (see General Procedure in the *CFX-Solver Manager User's Guide*).

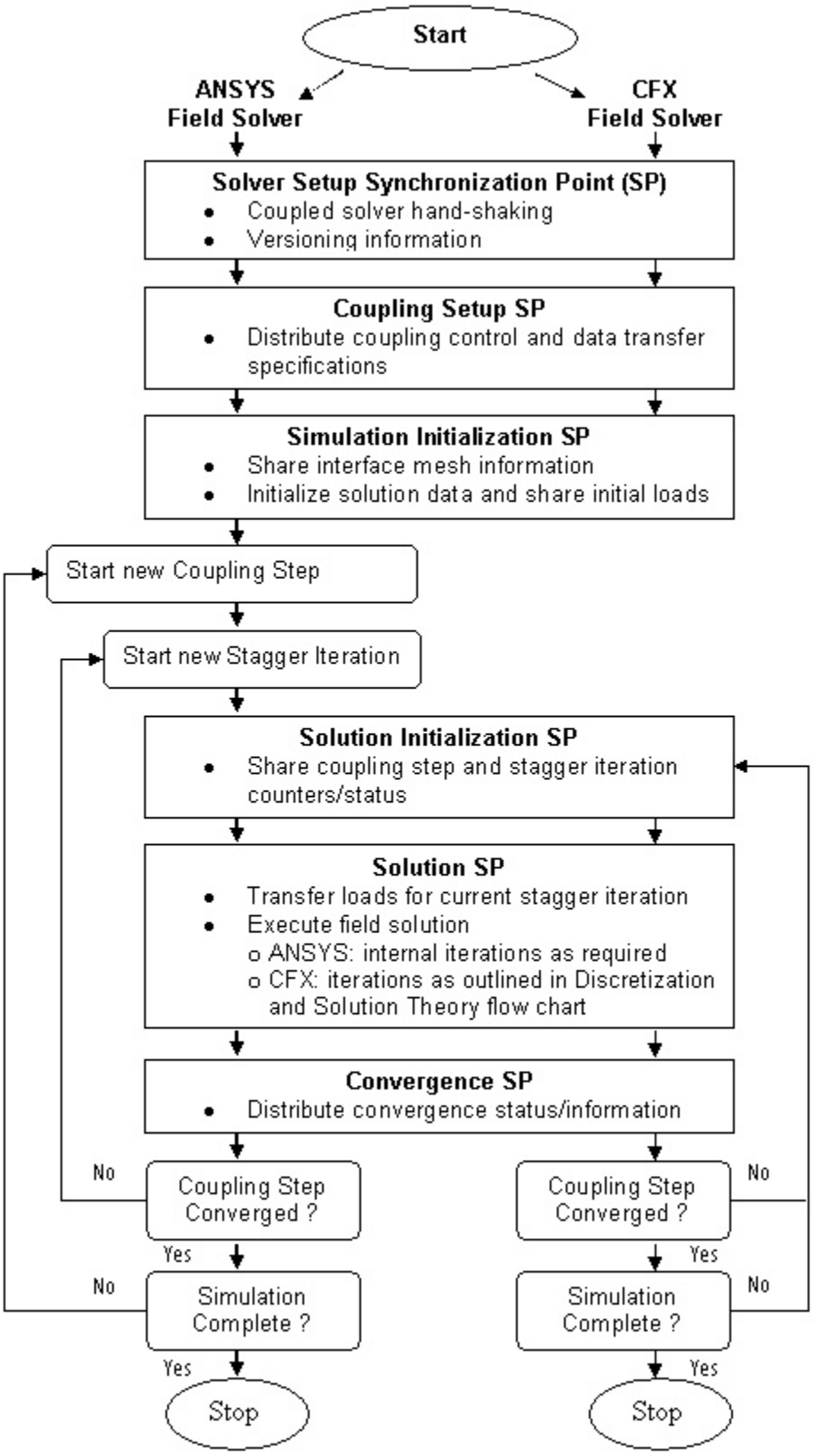
Refer to the following sections for more information:

- Coupling CFX to an External Solver: ANSYS Multi-field Simulations in the *CFX-Solver Modeling Guide*
- Chapter 4, Multi-field Analysis Using Code Coupling.

Coupled simulations begin with the execution of the Mechanical application and CFX field solvers. The Mechanical application solver acts as a coupling master process to which the CFX-Solver connects. Once that connection is established, the solvers advance through a sequence of six pre-defined synchronization points (SPs), as illustrated in *Figure 5.4* (p. 81). At each of these SPs, each field solver gathers the data it requires from the other solver in order to advance to the next point.

The first three SPs are used to prepare the solvers for the calculation intensive solution process, which takes place during the final three SPs. These final SPs define a sequence of coupling steps, each of which consists of one or more stagger/coupling iterations. During every stagger iteration, each field solver gathers the data it requires from the other solver, and solves its field equations for the current coupling step. Stagger iterations are repeated until a maximum number of stagger iterations is reached or until the data transferred between solvers and all field equations have converged. The latter guarantees an implicit solution of all fields for each coupling step.

Figure 5.4 Sequence of Synchronization Points



5.2.2.3. Using CFX and Other CAE Software

Third party code-coupling software or proprietary interfaces provided by the CAE software vendors can also be used in conjunction with CFX. Contact those software providers and your CFX service representative for more information.

Chapter 6: CFX Best Practices Guide for Numerical Accuracy

This guide provides best practice guidelines for Computational Fluid Dynamics (CFD) simulation and documentation of the verification, validation, and demonstration test cases. It describes:

- *An Approach to Error Identification, Estimation and Validation* (p. 83)
- *Definition of Errors in CFD Simulations* (p. 84)
- *General Best Practice Guidelines* (p. 94)
- *Selection and Evaluation of Experimental Data* (p. 104)

This guide is aimed at users who have moderate or little experience using ANSYS CFX. It is part of a series that provides advice for using ANSYS CFX in specific engineering application areas. The current guidelines are adapted from Best Practice Guidelines developed for the nuclear reactor safety applications [*143* (p. 264)].

6.1. An Approach to Error Identification, Estimation and Validation

An evaluation of CFD capabilities has to ensure that the different types of errors are identified and, as far as possible, treated separately. It is known from single-phase studies that the quantification and documentation of modeling errors (as in turbulence models, for example) can be achieved only if the other major sources of errors are reduced below an "acceptable" level. In an ideal world, this would mean, among other demands, that solutions are provided for grids and with timesteps that are fine enough so that numerical errors can be neglected. This is not a trivial task and the separation of errors cannot always be achieved. These difficulties will be greatly increased by the inclusion of multi-phase physics and unsteady effects. Nevertheless, the worst strategy would be to avoid the subject and to provide solutions on a single grid, with a single timestep, and with other uncertainties in initial conditions and boundary conditions not evaluated. This would result in solutions that would be of little use for the validation goals.

An essential quantity in the quality assurance procedure is the definition of target variables. They will mainly be scalar (integral) quantities (for instance, forces, heat transfer rates, and maximum temperature) or one-dimensional distributions, such as the wall heat transfer along a certain line. Convergence studies can be based on these variables without a reference to the grid used in the simulation. They can also be used for an asymptotic evaluation of convergence on unstructured meshes. Even more important, these quantities are of immediate meaning to engineers and allow them to understand the uncertainty from a physical standpoint. A danger of integral or local scalar quantities is that they might not be sensitive enough to detect local changes in the solutions under grid refinement. This should be kept in mind during the analysis.

In order to tackle the problem, it is necessary to first define the different type of errors that can impact a CFD simulation. It is then required that you list the most promising strategies in order to reduce or avoid these errors. Based on these strategies, procedures have to be defined that can be used for the test case simulations.

It might be not possible to rigorously perform the error estimation and reduction procedures described in the following sections for the complex demonstration cases. However, the best attempt should be

made to follow the principal ideas and to avoid single grid solutions without sensitivity studies. For these cases, it is even more important to follow a stringent documentation procedure and to list the possible deficiencies and uncertainties in the simulations.

The strategies for the reduction and evaluation of numerical errors have been developed for single-phase flows. There is no principal difference between the single- and multi-phase flow formulations. They are both based on (ensemble) averaged equations, and are mathematically similar. From a physical standpoint, there are however significant additional challenges due to the presence of the different phases, besides the obviously higher demands on model formulation. One of the additional complication lies in the presence of sharp interfaces between the phases, which require a higher degree of grid resolution than usually necessary for single-phase flows. In addition, multi-phase flows have a higher affinity to physical instabilities that might be suppressed on coarse grids, but appear under grid refinement. (This effect is sometimes also observed in single-phase flows. An example is the blunt trailing edge of an airfoil, where extreme grid refinement will eventually capture the vortex shedding of the mixing layer). It is to be kept in mind that the brute application of procedures might not lead to the desired results. Also in these cases, the spirit behind the guidelines should be followed and carried as far as possible.

Validation studies have to be based on experimental data. These data can introduce significant errors into the comparison. It is therefore required to select the project test cases with attention to potential error sources and experimental uncertainties. Definitions on the different types of test cases as well as on the requirements for the project are given in *Selection and Evaluation of Experimental Data* (p. 104).

6.2. Definition of Errors in CFD Simulations

CFD simulations have the following potential sources for errors or uncertainties:

- *Numerical Errors* (p. 85)

  Numerical errors result from the differences between the exact equations and the discretized equations solved by the CFD code. For consistent discretization schemes, these errors can be reduced by an increased spatial grid density and/or by smaller timesteps.

- *Modeling Errors* (p. 92)

  Modeling errors result from the necessity to describe flow phenomena such as turbulence, combustion, and multi-phase flows by empirical models. For turbulent flows, the necessity for using empirical models derives from the excessive computational effort to solve the exact equations[1] with a *Direct Numerical Simulation* (DNS) approach. Turbulence models are therefore required to bridge the gap between the real flow and the statistically averaged equations. Other examples are combustion models and models for interpenetrating continua, for example, two-fluid models for two-phase flows.

- *User Errors* (p. 92)

  User errors result from incorrect use of CFD software and are usually a result of insufficient expertise by the CFD user. Errors can be reduced or avoided by additional training and experience in combination with high-quality project management and by provision and use of Best Practice Guidelines and associated checklists.

- *Application Uncertainties* (p. 93)

[1] The Navier-Stokes equations for single-phase, Newtonian fluids

Application uncertainties are related to insufficient information to define a CFD simulation. A typical example is insufficient information on the boundary conditions.

- *Software Errors* (p. 93).

  Software errors are the result of an inconsistency between the documented equations and the actual implementation in the CFD software. They are usually a result of programming errors.

A more detailed definition of the different errors follows.

6.2.1. Numerical Errors

Numerical Errors are of the following types:

- *Solution Errors* (p. 85)
- *Spatial Discretization Errors* (p. 86)
- *Time Discretization Errors* (p. 86)
- *Iteration Errors* (p. 88)
- *Round-off Error* (p. 88)
- *Solution Error Estimation* (p. 89)

6.2.1.1. Solution Errors

The most relevant errors from a practical standpoint are solution errors[2]. They are the difference between the exact solution of the model equations and the numerical solution. The relative solution error can be formally defined as:

$$E_s = \frac{f_{\text{exact}} - f_{\text{numeric}}}{f_{\text{exact}}} \qquad \textbf{(6–1)}$$

Equation 6–1 (p. 85) is valid for every grid point for which the numerical solution exists. A global number can be defined by applying suitable norms, as:

$$\hat{E}_s = \frac{\left\| f_{\text{exact}} - f_{\text{numeric}} \right\|}{\left\| f_{\text{exact}} \right\|} \qquad \textbf{(6–2)}$$

The goal of a numerical simulation is to reduce this error below an acceptable limit.

Obviously, this is not a straightforward task, as the exact solution is not known and the error can therefore not be computed. Exceptions are simple test cases for code verification where an analytical solution is available.

Given a grid spacing Δ, and the truncation error order of a consistent discretization scheme, p, a Taylor series can be written to express the exact solution as:

[2] Sometimes also called 'discretization errors'

$$f_{\text{exact}} = f_{\text{numeric}} + c\ \Delta^{p} + \text{HOT} \qquad \textbf{(6–3)}$$

In other words, the numerical solution converges towards the exact solution with the p^{th} power of the grid spacing. Analogous definitions are available for time discretization errors.

6.2.1.2. Spatial Discretization Errors

Spatial discretization errors are the result of replacing the analytical derivatives or integrals in the exact equations by numerical approximations that have a certain truncation error. The truncation error can be obtained by inserting a Taylor series expansion of the numerical solution into the different terms of the discretized equations:

$$f_{\text{numerical}} = f_{\text{exact}} + \sum_{i=1}^{\infty} c_i f^{(i)}\ \Delta^{i} \qquad \textbf{(6–4)}$$

where $f^{(i)}$ is the i^{th} derivative of the exact solution at a given location. An example is a central difference for a spatial derivative:

$$\begin{aligned}
&\frac{\partial f}{\partial x} \approx \frac{f_{i+1} - f_{i-1}}{x_{i+1} - x_{i-1}} \\
&= (f_{\text{exact}} + f^{(1)}\ \Delta x + c_2 f^{(2)}\ \Delta x^2 + c_3 f^{(3)}\ \Delta x^3 + \text{HOT})\ /\ (2\Delta x) \\
&- (f_{\text{exact}} - f^{(1)}\ \Delta x + c_2 f^{(2)}\ \Delta x^2 - c_3 f^{(3)}\ \Delta x^3 + \text{HOT})/(2\Delta x) \\
&= f^{(1)} + o(\Delta x^2)
\end{aligned} \qquad \textbf{(6–5)}$$

This formulation has a truncation error of order 2 and is therefore second-order accurate. The overall truncation error order of the spatial discretization scheme is determined by the lowest order truncation error after all terms have been discretized.

In the $o\ (\Delta x^2)$ term of *Equation 6–5* (p. 86), the leading term is proportional to $f^{(3)}\ \Delta x^2$. First-order upwind differencing of the convective terms yields truncation errors $o\ (\Delta x)$ with leading term proportional to $f^{(2)}\ \Delta x$. This term then contributes to the diffusion term (numerical/false diffusion), which is most dangerous in 3D problems with grid lines not aligned to the flow direction. These schemes enhance the dissipation property of the numerical algorithm (see for example, Ferziger and Peric [*141* (p. 264)]) and are not desirable in high-quality CFD simulations.

From a practical standpoint, it is important to understand that for a first-order method, the error is reduced to 50% by a doubling of the grid resolution in each spatial direction. For a second-order method, it is reduced to 25% for the same grid refinement.

6.2.1.3. Time Discretization Errors

Time adds another dimension to a CFD simulation. The definition of time discretization errors is therefore similar to the definition of the spatial discretization errors. The spatial discretization usually results in a system of nonlinear algebraic equations of the form:

$$\frac{\partial \varphi}{\partial t} = g\ (\varphi) \tag{6–6}$$

The error in the time discretization can again be obtained by a Taylor series expansion of the numerical formulation of this equation. With the example of a backward Euler integration:

$$\frac{\varphi^{n+1} - \varphi^{n}}{\Delta\ t} = g\ \left(\varphi^{n+1}\right) \tag{6–7}$$

the discretization error is:

$$\begin{aligned} &\frac{\varphi^{n+1} - \left(\varphi^{n+1} - \varphi^{(1)\ n+1}\ \Delta\ t + c_2\ \varphi^{(2)\ n+1}\ \Delta\ t^2 + \mathrm{HOT}\right)}{\Delta\ t} \\ &\qquad = \frac{\partial\ \varphi^{n+1}}{\partial\ t} + c_2\ \varphi^{(2)\ n+1}\ \Delta\ t + \mathrm{HOT} \end{aligned} \tag{6–8}$$

The error is therefore first-order for the time derivative.

An additional complication for implicit methods comes from the inclusion of the unknown φ^{n+1} in the right hand side of *Equation 6–7* (p. 87). In order to benefit from an implicit method, a linearization of g has to be included:

$$g\ \left(\varphi^{n+1}\right) = g\ (\varphi^{n})\ + G\ \frac{\Delta\ \varphi}{\Delta\ t}\ \Delta\ t + o\ (\Delta\ t^2)\ ; \qquad \text{with} \qquad G = \frac{\partial\ g}{\partial\ \varphi} \tag{6–9}$$

The resulting discretized equation is therefore:

$$\left[\ \frac{1}{\Delta\ t} - G\right]\ \left(\varphi^{n+1} - \varphi^{n}\right) = g\ (\varphi^{n}) \tag{6–10}$$

This constitutes an implicit formulation with first-order accuracy. A second-order time differencing is not compatible with this linearization of the right hand side, as the linearization introduced a first-order error in $\Delta\ t$. In order to be able to satisfy the implicit dependency of the right hand side on the time level $n+1$ more closely, inner iterations (or coefficient loops) are frequently introduced:

$$\begin{aligned} &\frac{\varphi^{n+1,m+1} - \varphi^{n}}{\Delta\ t} = g\ \left(\varphi^{n+1,m+1}\right) \\ &= g\ \left(\varphi^{n+1,m}\right) + G\ \left(\varphi^{n+1,m+1} - \varphi^{n+1,m}\right) + o\ \left(\varphi^{n+1,m+1} - \varphi^{n+1,m}\right)^2 \end{aligned} \tag{6–11}$$

where an additional iteration over the index m is carried out. This equation can be reformulated as:

$$\left[\frac{1}{\Delta t} - G\right]\left(\varphi^{n+1,m+1} - \varphi^{n+1,m}\right) = g\left(\varphi^{n+1,m}\right) - \frac{\varphi^{n+1,m} - \varphi^{n}}{\Delta t} \quad \textbf{(6–12)}$$

This equation can be converged completely (left hand side goes to zero) in m in order to solve the original exact implicit formulation given by *Equation 6–7* (p. 87). It is obvious that it is not necessary to converge the coefficient loop to zero, while the right hand side has a finite (first-order) error in Δt. It can be shown that for a first-order time integration, one coefficient loop is consistent with the accuracy of the method. In a case where a second-order accurate scheme is used in the time derivative, two coefficient loops will ensure overall second-order accuracy of the method. Note, however, that this is correct only if the coefficient loops are not under-relaxed in any way.

For explicit methods, no coefficient loops are required and the time discretization error is defined solely from a Taylor series expansion.

6.2.1.4. Iteration Errors

The iteration error is similar to the coefficient loop error described above. It occurs in a case where a steady-state solution is sought from an iterative method. In most CFD codes, the iteration is carried out via a (pseudo-) timestepping scheme as given in this example, which also appears above:

$$\left[\frac{1}{\Delta t} - G\right]\left(\varphi^{n+1} - \varphi^{n}\right) = g\left(\varphi^{n}\right) \quad \textbf{(6–13)}$$

Zero iteration error would mean that the left hand side is converged to zero, leading to the converged solution $g(\varphi) = 0$. However, in practical situations, the iterative process is stopped at a certain level, in order to reduce the numerical effort. The difference between this solution and the fully converged solution defines the iteration error.

The iteration error is usually quantified in terms of a residual or a residual norm. This can be the maximum absolute value of the right hand side, $g(\varphi)$, for all grid points, or a root mean square of this quantity. In most CFD methods, the residual is non-dimensionalized to allow for a comparison between different applications with different scaling. However, the non-dimensionalization is different for different CFD codes, making general statements as to the required absolute level of residuals impractical. Typically, the quality of a solution is measured by the overall reduction in the residual, compared to the level at the start of the simulation.

The iteration error should be controlled with the use of the target variables. The value of the target variable can be plotted as a function of the convergence level. In case of iterative convergence, the target variable should remain constant with the convergence level. It is desirable to display the target variable in the solver monitor during the simulation.

6.2.1.5. Round-off Error

Another numerical error is the round-off error. It results from the fact that a computer only solves the equations with a finite number of digits (around 8 for single-precision and around 16 for double-precision). Due to the limited number of digits, the computer cannot differentiate between numbers that are different by an amount below the available accuracy. For flow simulations with large-scale differences (for instance, extent of the domain vs. cell size), this can be a problem for single-precision simulations. Round-off errors are often characterized by a random behavior of the numerical solution.

6.2.1.6. Solution Error Estimation

The most practical method to obtain estimates for the solution error is systematic grid refinement or timestep reduction. In the following, the equations for error estimation are given for grid refinement. The same process can be used for timestep refinement.

If the asymptotic range of the convergence properties of the numerical method is reached, the difference between solutions on successively refined grids can be used as an error estimator. This allows the application of Richardson extrapolation to the solutions on the different grids (Roache [*139* (p. 263)]). In the asymptotic limit, the solution can be written as follows:

$$f_{\text{exact}} = f_i + g_1 h_i + g_2 h_i^2 + \ldots \qquad \textbf{(6–14)}$$

In this formulation, h is the grid spacing (or a linear measure of it) and the g_i are functions independent of the grid spacing. The subscript, i, refers to the current level of grid resolution. Solutions on different grids are represented by different subscripts.

The assumption for the derivation of an error estimate is that the order of the numerical discretization is known. This is usually the case. Assuming a second-order accurate method, the above expansion can be written for two different grids:

$$f_{\text{exact}} = f_1 + g_2 h_1^2 + \ldots \qquad \textbf{(6–15)}$$

$$f_{\text{exact}} = f_2 + g_2 h_2^2 + \ldots$$

Neglecting higher-order terms, the unknown function g_2 can be eliminated from this equation. An estimate for the exact solution is therefore:

$$f_{\text{exact}} = \frac{h_2^2 f_1 - h_1^2 f_2}{h_2^2 - h_1^2} + \text{HOT} \qquad \textbf{(6–16)}$$

The difference between the fine grid solution and the exact solution (defining the error) is therefore:

$$E = f_{\text{exact}} - f_1 = \frac{f_1 - f_2}{r^2 - 1} + \text{HOT} \; ; \qquad \text{with} \qquad r = \frac{h_2}{h_1} \qquad \textbf{(6–17)}$$

For an arbitrary order of accuracy, p, of the underlying numerical scheme, the error is given by:

$$E = f_{\text{exact}} - f_1 = \frac{f_1 - f_2}{r^p - 1} + \text{HOT} \qquad \textbf{(6–18)}$$

In order to build the difference between the solutions f_1 and f_2, it is required that the coarse and the fine grid solution is available at the same location. In the case of a doubling of the grid density without a movement of the coarse grid nodes, all information is available on the coarse grid nodes. The applic-

ation of the correction to the fine-grid solution requires an interpolation of the correction to the fine grid nodes (Roache [*139* (p. 263)]). In the case of a general grid refinement, the solutions are not available on the same physical locations. An interpolation of the solution between the different grids is then required for a direct error estimate. It has to be ensured that the interpolation error is lower than the solution error in order to avoid a contamination of the estimate.

Richardson interpolation can also be applied to integral quantities (target variables), such as lift or drag coefficients. In this case, no interpolation of the solution between grids is required.

Note that the above derivation is valid only if the underlying method has the same order of accuracy everywhere in the domain and if the coarse grid is already in the asymptotic range (the error decreases with the order of the numerical method). In addition, the method magnifies round-off and iteration errors.

The intention of the Richardson interpolation was originally to improve the solution on the fine grid. This requires an interpolation of the correction to the fine grid and introduces additional inaccuracies into the extrapolated solution, such as errors in the conservation properties of the solution. A more practical use of the Richardson extrapolation is the determination of the relative solution error, A_1:

$$A_1 = \frac{f_1 - f_{\text{exact}}}{f_{\text{exact}}} \qquad \textbf{(6–19)}$$

An estimate, E_1, of this quantity can be derived from *Equation 6–16* (p. 89):

$$A_1 = \frac{f_2 - f_1}{f_1} \frac{1}{r^p - 1} \qquad \textbf{(6–20)}$$

It can be shown (Roache [*139* (p. 263)]) that the exact relative error and the approximation are related by:

$$A_1 = E_1 + O\left(h^{p+1}\right) \qquad \textbf{(6–21)}$$

Equation 6–20 (p. 90) can also be divided by the range of f_1 or another suitable quantity in order to prevent the error to become infinite as f_1 goes to zero.

In order to arrive at a practical error estimator, the following definitions are proposed:

Field error:

$$A_f = \frac{|f_2 - f_1|}{\text{range}\left(f_1\right)} \frac{1}{(r^p - 1)} \qquad \textbf{(6–22)}$$

Maximum error:

$$A_{\max} = \frac{\max\left(f_2 - f_1\right)}{\text{range}\left(f_1\right)} \frac{1}{\left(r^p - 1\right)} \tag{6–23}$$

RMS error:

$$A_{\text{rms}} = \frac{\text{rms}\left(f_2 - f_1\right)}{\text{range}\left(f_1\right)} \frac{1}{\left(r^p - 1\right)} \tag{6–24}$$

Target variable error:

$$A_{\text{rms}} = \frac{|\Theta_1 - \Theta_2|}{|\Theta_1|} \frac{1}{\left(r^p - 1\right)} \tag{6–25}$$

where Θ is the defined target variable (list, drag, heat transfer coefficient, maximum temperature, mass flow, and so on).

Similar error measures can be defined for derived variables, which can be specified for each test case. Typical examples would be the total mass flow, the pressure drop, or the overall heat transfer. This will be the recommended strategy, as it avoids the interpolation of solutions between the coarse and the fine grid.

For unstructured meshes, the above considerations are valid only in cases of a global refinement of the mesh. Otherwise, the solution error will not be reduced continuously across the domain. For unstructured refinement the refinement level, r, can be defined as follows:

$$r_{\text{effective}} = \left(\frac{N_1}{N_2}\right)^{1/D} \tag{6–26}$$

where N_i is the number of grid points and D is the dimension of the problem.

It must be emphasized that these definitions do not impose an upper limit on the real error, but are estimates for the evaluation of the quality of the numerical results. Limitations of the above error estimates are:

- The solution has to be smooth
- The truncation error order of the method has to be known
- The solution has to be sufficiently converged in the iteration domain
- The coarse grid solution has to be in the asymptotic range.

For three-dimensional simulations, the demand that the coarse grid solution be in the asymptotic range is often hard to ensure. It is therefore required to compute the error for three different grid levels, to avoid fortuitous results. If the solution is in the asymptotic range, the following indicator should be close to constant:

$$E_h = \frac{\text{error}}{h^p} \qquad \textbf{(6–27)}$$

6.2.2. Modeling Errors

In industrial CFD methods, numerous physical and chemical models are incorporated. Models are usually applied to avoid the resolution of a large range of scales, which would result in excessive computing requirements.

The classical model used in almost all industrial CFD applications is a turbulence model. It is based on time or ensemble averaging of the equations resulting in the so-called *Reynolds Averaged Navier-Stokes* (RANS) equations. Due to the averaging procedure, information from the full Navier-Stokes equations is lost. It is supplied back into the code by the turbulence model. The most widely used industrial models are two-equation models, such as the $\mathrm{k}-\varepsilon$ or $\mathrm{k}-\omega$ models.

The statistical model approach reduces the resolution requirements in time and space by many orders of magnitude, but requires the calibration of model coefficients for certain classes of flows against experimental data. There is a wide variety of models that are introduced to reduce the resolution requirements for CFD simulations, including:

- Turbulence models
- Multi-phase models
- Combustion models
- Radiation models.

In combustion models, the reduction can be both in terms of the chemical species and in terms of the turbulence-combustion interaction. In radiation, the reduction is typically in terms of the wavelength and/or the directional information. For multi-phase flows, it is usually not possible to resolve a large number of individual bubbles or droplets. In this case, the equations are averaged over the different phases to produce continuous distributions for each phase in space and time.

As all of these models are based on a reduction of the 'real' physics to a reduced 'resolution', information has to be introduced from outside the original equations. This is usually achieved by experimental calibration, or by available *DNS* (p. 84) results.

Once a model has been selected, the accuracy of the simulation cannot be increased beyond the capabilities of the model. This is the largest factor of uncertainty in CFD methods, as modeling errors can be of the order of 100% or more. These large errors occur in cases where the CFD solution is very sensitive to the model assumptions and where a model is applied outside its range of calibration.

Because of the complexity of industrial simulations, it cannot be ensured that the models available in a given CFD code are suitable for a new application. While in most industrial codes a number of different models are available, there is no *a priori* criterion as to the selection of the most appropriate one. Successful model selection is largely based on the expertise and the knowledge of the CFD user.

6.2.3. User Errors

User errors result from the inadequate use of the resources available for a CFD simulation. The resources are given by:

- Problem description

- Computing power
- CFD software
- Physical models in the software
- Project time frame.

According to the *ERCOFTAC Best Practice Guidelines* [*140* (p. 263)], some of the sources for user errors are:

- Lack of experience
- Lack of attention to detail or other mistakes.

Often, user errors are related to management errors when insufficient resources are assigned to a project, or inexperienced users are given a too complex application. Typical user errors are:

- Oversimplification of a given problem; for example, geometry, equation system, and so on
- Poor geometry and grid generation
- Use of incorrect boundary conditions
- Selection of non-optimal physical models
- Incorrect or inadequate solver parameters; for example, timestep, and so on
- Acceptance of non-converged solutions
- Post-processing errors.

6.2.4. Application Uncertainties

Application uncertainties result from insufficient knowledge to carry out the simulation. This is in most cases a lack of information on the boundary conditions or of the details of the geometry. A typical example is the lack of detailed information at the inlet. A complete set of inlet boundary conditions is composed of inflow profiles for all transported variables (momentum, energy, turbulence intensity, turbulence length scale, volume fractions, and so on). This information can be supplied from experiments or from a CFD simulation of the upstream flow. In most industrial applications, this information is not known and bulk values are given instead. In some cases, the detailed information can be obtained from a separate CFD simulation (for instance a fully developed pipe inlet flow). In other cases, the boundaries can be moved far enough away from the area of interest to minimize the influence of the required assumptions for the complete specification of the boundary conditions.

Typical application uncertainties are:

- Lack of boundary condition information
- Insufficient information on the geometry
- Uncertainty in experimental data for solution evaluation.

6.2.5. Software Errors

Software errors are defined as any inconsistency in the software package. This includes the code, its documentation, and the technical service support. Software errors occur when the information you have on the equations to be solved by the software is different from the actual equations solved by the code. This difference can be a result of:

- Coding errors (bugs)

- Errors in the graphical user interface (GUI)
- Documentation errors
- Incorrect support information.

6.3. General Best Practice Guidelines

In order to reduce the numerical errors, it is necessary to have procedures for the estimation of the different errors described in *Definition of Errors in CFD Simulations* (p. 84). The main goal is to reduce the solution error to a minimum with given computer resources.

6.3.1. Avoiding User Errors

User errors are directly related to the expertise, the thoroughness, and the experience of the user. For a given user, these errors can only be minimized by good project management and thorough interaction with others. In case of inexperienced users, day-to-day interaction with a CFD expert/manager is required to avoid major quality problems. A structured work plan with intermediate results is important for intermediate and long-term projects.

A careful study of the CFD code documentation and other literature on the numerical methods as well as the physical models is highly recommended. Furthermore, benchmark studies are recommended to enable you to understand the capabilities and limitations of CFD methods. A comparison of different CFD methods is desirable, but not always possible.

6.3.2. Geometry Generation

Before the grid generation can start, the geometry has to be created or imported from CAD-data. In both cases, attention should be given to:

- The use of a correct coordinate system
- The use of the correct units
- The use of geometrical simplification, for example, symmetry planes
- Local details. In general, geometrical features with dimensions below the local mesh size (for example, wall roughness or porous elements) are not included in the geometrical model. These should be incorporated through a suitable model.

In the case that the geometry is imported from CAD-data, the data should be checked beforehand. Frequently, after the import of CAD-data, the CAD-data has to be adapted (cleaned) before it can be used for mesh generation. It is essential for mesh generation to have closed volumes. The various CAD-data formats do not always contain these closed volumes. Therefore, the CAD-data has to be altered in order to create the closed volumes. It has to be ensured that these changes do not influence the flow to be computed.

6.3.3. Grid Generation

In a CFD analysis, the flow domain is subdivided in a large number of computational cells. All these computational cells together form the so-called mesh or grid. The number of cells in the mesh should be taken sufficiently large, such that an adequate resolution is obtained for the representation of the geometry of the flow domain and the expected flow phenomena in this domain.

A good mesh quality is essential for performing a good CFD analysis. Therefore, assessment of the mesh quality before performing a large and complex CFD analysis is very important. Most of the mesh gener-

ators and CFD solvers offer the possibility of checking the mesh on several cells or mesh parameters, such as aspect ratio, internal angle, face warpage, right handiness, negative volumes, cracks, and tetrahedral quality. The reader is referred to the user guides of the various mesh generators and CFD solvers for more information on these cells and mesh parameters.

Recommendations for grid generation are:

- Avoid high grid stretching ratios.
  - Aspect ratios should not be larger than 20 to 50 in regions away from the boundary.
  - Aspect ratios may be larger than that in unimportant regions.
  - Aspect ratios may, and should, be larger than that in the boundary layers. For well resolved boundary layers at high Re numbers, the near-wall aspect ratios can be of the order of 10^5-10^6.
- Avoid jumps in grid density.
  - Growth factors should be smaller than 1.3.
- Avoid poor grid angles.
- Avoid non-scalable grid topologies. Non-scalable topologies can occur in block-structured grids and are characterized by a deterioration of grid quality under grid refinement.
- Avoid non-orthogonal, for example, unstructured tetrahedral meshes, in (thin) boundary layers.
- Use a finer and more regular grid in critical regions, for example, regions with high gradients or large changes such as shocks.
- Avoid the presence of arbitrary grid interfaces, mesh refinements, or changes in element types in critical regions. An arbitrary grid interface occurs when there is no one-to-one correspondence between the cell faces on both sides of a common interface, between adjacent mesh parts.

If possible, determine the size of the cells adjacent to wall boundaries where turbulence models are used, before grid generation has started.

Numerical diffusion is high when computational cells are created that are not orthogonal to the fluid flow. If possible, avoid computational cells that are not orthogonal to the fluid flow.

Judge the mesh quality by using the possibilities offered by the mesh generator. Most mesh generators offer checks on mesh parameters, such as aspect ratio, internal angle, face warpage, right handiness, negative volumes, cracks, and tetrahedral quality.

It should be demonstrated that the final result of the calculations is independent of the grid that is used. This is usually done by comparison of the results of calculations on grids with different grid sizes.

Some CFD methods allow the application of grid adaptation procedures. In these methods, the grid is refined in critical regions (high truncation errors, large solution gradients, and so on). In these methods, the selection of appropriate indicator functions for the adaptation is essential for the success of the simulations. They should be based on the most important flow features to be computed.

As a general rule, any important shear layer in the flow (boundary layer, mixing layer, free jets, wakes, and so on) should be resolved with at least 10 nodes normal to the layer. This is a very challenging requirement that often requires the use of grids that are aligned with the shear layers.

6.3.4. Model Selection and Application

Modeling errors are the most difficult errors to avoid, as they cannot be reduced systematically. The most important factor for the reduction of modeling errors is the quality of the models available in the CFD package and the experience of the user. There is also a strong interaction between modeling errors and the time and space resolution of the grid. The resolution has to be sufficient for the model selected for the application.

In principle, modeling errors can only be estimated in cases where the validation of the model is 'close' to the intended application. Model validation is essential for the level of confidence you can have in a CFD simulation. It is therefore required that you gather all available information on the validation of the selected model, both from the open literature and from the code developers (vendors). In case that CFD is to be applied to a new field, it is recommended that you carry out additional validation studies, in order to gain confidence that the physical models are adequate for the intended simulation.

If several modeling options are available in the code (as is usually the case for turbulence, combustion and multi-phase flow models), it is recommended that you carry out the simulation with different models in order to test the sensitivity of the application with respect to the model selection.

In case you have personal access to a modeling expert in the required area, it is recommended that you interact with the model developer or expert to ensure the optimal selection and use of the model.

6.3.4.1. Turbulence Models

There are different methods for the treatment of turbulent flows. The need for a model results from the inability of CFD simulations to fully resolve all time and length scales of a turbulent motion. In classical CFD methods, the Navier-Stokes equations are usually time- or ensemble-averaged, reducing the resolution requirements by many orders of magnitude. The resulting equations are the *RANS* (p. 92) equations. Due to the averaging procedure, information is lost, which is then fed back into the equations by a turbulence model.

The amount of information that has to be provided by the turbulence model can be reduced if the large time and length scales of the turbulent motion are resolved. The equations for this so-called *Large Eddy Simulation* (LES) method are usually filtered over the grid size of the computational cells. All scales smaller than the resolution of the mesh are modeled and all scales larger than the cells are computed. This approach is several orders of magnitude more expensive than a *RANS* (p. 92) simulation and is therefore not used routinely in industrial flow simulations. It is most appropriate for free shear flows, as the length scales near the solid walls are usually very small and require small cells even for the *LES* (p. 96) method.

RANS (p. 92) methods are the most widely used approach for CFD simulations of industrial flows. Early methods, using algebraic formulations, have been largely replaced by more general transport equation models, for both implementation and accuracy considerations. The use of algebraic models is not recommended for general flow simulations, due to their limitations in generality and their geometric restrictions. The lowest level of turbulence models that offer sufficient generality and flexibility are two-equation models. They are based on the description of the dominant length and time scale by two independent variables. Models that are more complex have been developed and offer more general platforms for the inclusion of physical effects. The most complex *RANS* (p. 92) model used in industrial CFD applications are *Second Moment Closure* (SMC) models. Instead of two equations for the two main turbulent scales, this approach requires the solution of seven transport equations for the independent Reynolds stresses and one length (or related) scale.

The challenge for the user of a CFD method is to select the optimal model for the application at hand from the models available in the CFD method. In most cases, it cannot be specified beforehand which model will offer the highest accuracy. However, there are indications as to the range of applicability of different turbulence closures. This information can be obtained from validation studies carried out with the model.

In addition to the accuracy of the model, consideration has to be given to its numerical properties and the required computer power. It is often observed that more complex models are less robust and require many times more computing power than the additional number of equations would indicate. Frequently, the complex models cannot be converged at all, or, in the worst case, the code becomes unstable and the solution is lost.

It is not trivial to provide general rules and recommendations for the selection and use of turbulence models for complex applications. Different CFD groups have given preference to different models for historical reasons or personal experiences. Even turbulence experts cannot always agree as to which model offers the best cost-performance ratio for a new application.

6.3.4.1.1. One-equation Models

A number of one-equation turbulence models based on an equation for the eddy viscosity have been developed over the last years. Typical applications are:

- Airplane- and wing flows
- External automobile aerodynamics
- Flow around ships.

These models have typically been optimized for aerodynamic flows and are not recommended as general-purpose models.

6.3.4.1.2. Two-equation Models

The two-equation models are the main-stand of industrial CFD simulations. They offer a good compromise between complexity, accuracy and robustness. The most popular models are the standard model and different versions of the $k-\omega$ model, see Wilcox [*30* (p. 247)]. The standard $k-\omega$ model of Wilcox is the most well known of the $k-\omega$ based models, but shows a severe free-stream dependency. It is therefore not recommended for general industrial flow simulations, as the results are strongly dependent on the user input. Alternative formulations are available, see for example, the **Shear Stress Transport** (SST) model, Menter [*9* (p. 244)].

An important weakness of standard two-equation models is that they are insensitive to streamline curvature and system rotation. Particularly for swirling flows, this can lead to an over-prediction of turbulent mixing and to a strong decay of the core vortex. There are curvature correction models available, but they have not been generally validated for complex flows.

The standard two-equation models can also exhibit a strong build-up of turbulence in stagnation regions, due to their modeling of the production terms. Several modifications are available to reduce this effect, for instance by Kato and Launder [*128* (p. 262)]. They should be used for flows around rods, blades, airfoils, and so on.

6.3.4.1.3. Second Moment Closure (SMC) Models

SMC (p. 96) models are based on the solution of a transport equation for each of the independent Reynolds stresses in combination with the ε- or the ω-equation. These models offer generally a wider

modeling platform and account for certain effects due to their exact form of the turbulent production terms. Some of these models show the proper sensitivity to swirl and system rotation, which have to be modeled explicitly in a two-equation framework. *SMC* (p. 96) models are also superior for flows in stagnation regions, where no additional modifications are required.

One of the weak points of the *SMC* (p. 96) closure is that the same scale equations are used as in the two-equation framework. As the scale equation is typically one of the main sources of uncertainty, it is found that *SMC* (p. 96) models do not consistently produce superior results compared to the simpler models. In addition, experience has shown that *SMC* (p. 96) models are often much harder to handle numerically. The model can introduce a strong nonlinearity into the CFD method, leading to numerical problems in many applications.

SMC (p. 96) models are usually not started from a pre-specified initial condition, but from an already available solution from a two-equation (or simpler) model. This reduces some of the numerical problems of the *SMC* (p. 96) approach. In addition, it offers an important sensitivity study, as it allows quantifying the influence of the turbulence model on the solution. It is therefore recommended that you fully converge the two-equation model solution and save it for a comparison with the *SMC* (p. 96) model solution. The difference between the solutions is a measure of the influence of the turbulence model and therefore an indication of the modeling uncertainty. This is possible only in steady state simulations. For unsteady flows, the models usually have to be started from the initial condition.

6.3.4.1.4. Large Eddy Simulation Models

LES (p. 96) models are based on the numerical resolution of the large turbulence scales and the modeling of the small scales. *LES* (p. 96) is not yet a widely used industrial approach, due to the large cost of the required unsteady simulations. For certain classes of applications, *LES* (p. 96) will be applicable in the near future. The most appropriate area will be free shear flows, where the large scales are of the order of the solution domain (or only an order of magnitude smaller). For boundary layer flows, the resolution requirements are much higher, as the near-wall turbulent length scales become much smaller. The internal flows (pipe flows, channel flows) are in between, as they have a restricted domain in the wall normal direction, but small scales have to be resolved in the other two directions.

LES (p. 96) simulations do not easily lend themselves to the application of grid refinement studies both in the time and the space domain. The main reason is that the turbulence model adjusts itself to the resolution of the grid. Two simulations on different grids are therefore not comparable by asymptotic expansion, as they are based on different levels of the eddy viscosity and therefore on a different resolution of the turbulent scales. From a theoretical standpoint, the problem can be avoided, if the *LES* (p. 96) model is not based on the grid spacing, but on a pre-specified filter-width. This would allow reaching grid-independent *LES* (p. 96) solutions above the *DNS* (p. 84) limit. However, *LES* (p. 96) is a very expensive method and systematic grid and timestep studies are prohibitive even for a pre-specified filter. It is one of the disturbing facts that *LES* (p. 96) does not lend itself naturally to quality assurance using classical methods. This property of the *LES* (p. 96) also indicates that (nonlinear) multigrid methods of convergence acceleration are not suitable in this application.

On a more global level, the grid convergence can be tested using averaged quantities resulting from the *LES* (p. 96) simulation. The averaged *LES* (p. 96) results can be analyzed in a similar way as *RANS* (p. 92) solutions (at least qualitatively). Again, it is expensive to perform several *LES* (p. 96) simulations and grid refinement will therefore be more the exception than the rule.

Due to the high computing requirements of *LES* (p. 96), modern developments in the turbulence models focus on a combination of *RANS* (p. 92) and *LES* (p. 96) models. The goal is to cover the wall boundary layers with *RANS* (p. 92) and to allow unsteady (*LES* (p. 96)-like) solutions in largely separated

and unsteady flow regions (for example, flow behind a building, or other blunt bodies). There are two alternatives of such methods available in ANSYS CFX.

The first alternative is called *Scale-Adaptive-Simulation* (SAS) model (Menter and Egorov [*130* (p. 262)], [*131* (p. 262)], [*144* (p. 264)], [*145* (p. 264)]). It is essentially an improved *Unsteady* RANS (URANS) method that develops *LES* (p. 96)-like solutions in unstable flow regimes.

The second alternative is called *Detached Eddy Simulation* (DES) (Spalart [*146* (p. 264)]), implemented in the version of Strelets [*58* (p. 251)]. The current recommendation is to use the *SAS* (p. 99) model, as it has less grid sensitivity than the *DES* (p. 99) formulation. In case that *SAS* (p. 99) does not provide an unsteady solution, the *DES* (p. 99) model should be applied. It should be noted that both model formulations require small timesteps with a Courant number of CFL<1. You are encouraged to read the original references before applying these models.

6.3.4.1.5. Wall Boundary Conditions

There are generally three types of boundary conditions that can be applied to a *RANS* (p. 92) simulation:

- *Wall Function Boundary Conditions* (p. 99)
- *Integration to the wall (low-Reynolds number formulation)* (p. 99)
- *Mixed formulation (automatic near-wall treatment)* (p. 100).

6.3.4.1.5.1. Wall Function Boundary Conditions

Standard wall functions are based on the assumption that the first grid point off the wall (or the first integration point) is located in the universal law-of-the-wall or logarithmic region. Wall functions eliminate the need to resolve the very thin viscous sublayer, leading to a reduction in the number of cells and to a more moderate (and desirable) aspect ratio of the cells (ratio of the longest to the smallest side in a structured grid). High aspect ratios can result in numerical problems due to round-off errors.

On the other hand, standard wall function formulations are difficult to handle, because you have to ensure that the grid resolution near the wall satisfies the wall function requirements. If the grid becomes too coarse, the resolution of the boundary layer is no longer ensured. If the resolution becomes too fine, the first grid spacing can be too small to bridge the viscous sublayer. In this case, the logarithmic profile assumptions are no longer satisfied. You have to ensure that both limits are not overstepped in the grid generation phase.

The lower limit on the grid resolution for standard wall functions is a severe detriment to a systematic grid refinement process, as required by the best practice approach. In other words, instead of an improved accuracy of the solution with grid refinement, the solution will deteriorate from a certain level on, leading eventually to a singularity of the numerical method. Standard wall functions are therefore not recommended for systematic grid refinement studies. Recently, alternative formulations (scalable wall functions) have become available, Menter and Esch [*142* (p. 264)], which allow for a systematic grid refinement when using wall functions.

6.3.4.1.5.2. Integration to the wall (low-Reynolds number formulation)

The use of low-Reynolds (low-Re) number formulations of turbulence models for the integration of the equations through the viscous sublayer is generally more accurate, as no additional assumptions are required concerning the variation of the variables near the wall. On the downside, most low-Re extensions of turbulence models are quite complex and can reduce the numerical performance or even destabilize the numerical method. In addition, classical low-Re models require a very fine near-wall resolution of $y^+ \sim 1$ at all wall nodes. This is very hard to ensure for all walls of a complex industrial application. In

the case that significantly coarser grids are used, the wall shear stress and the wall heat transfer can be reduced significantly below their correct values.

6.3.4.1.5.3. Mixed formulation (automatic near-wall treatment)

In ANSYS CFX, hybrid methods are available for all ω-equation based turbulence models (automatic near-wall treatment), which automatically switch from a low-Re formulation to wall functions based on the grid spacing you provide. These formulations provide the optimal boundary condition for a given grid. From a best practice standpoint, they are the most desirable, as they allow for an accurate near-wall treatment over a wide range of grid spacings. However, accurate boundary layer simulations do not depend only on the y^+ near-wall spacing, but also require a minimum of at least 10 grid nodes inside the boundary layer.

6.3.4.1.5.4. Recommendations for Model Selection

- Avoid the use of classical wall functions, as they are inconsistent with grid refinement.
- Avoid strict low-Re number formulations, unless it is ensured that all near-wall cells are within the resolution requirements of the formulation.
- In combination with the $k-\varepsilon$ model, use scalable wall functions. They can be applied to a range of grids without immediate deterioration of the solution (default in ANSYS CFX).
- For more accurate simulations, use an automatic wall treatment in combination with *SST* (p. 97) turbulence model (default in ANSYS CFX).

6.3.4.2. Heat Transfer Models

The heat transfer formulation is strongly linked to the underlying turbulence model. For eddy viscosity models, the heat transfer simulation is generally based on the analogy between heat and momentum transfer. Given the eddy viscosity of the two-equation model, the heat transfer prediction is based on the introduction of a molecular and a turbulent Prandtl number. The treatment of the energy equation is therefore similar to the treatment of the momentum equations. No additional transport equations are required for the turbulent heat transfer prediction. The boundary conditions are the same as for the momentum equations and follow the same recommendations.

For *SMC* (p. 96) models, three additional transport equations must be solved for the turbulent heat transfer vector in order to be consistent with the overall closure level. Only a few CFD methods offer this option. In most cases, the heat transfer is computed from an eddy diffusivity with a constant turbulent Prandtl number.

6.3.4.3. Multi-Phase Models

Multi-phase models are required in cases where more than one phase is involved in the simulation (phases can also be non-miscible fluids). There is a wide variety of multi-phase flow scenarios, with the two extremes of small-scale mixing of the phases or a total separation of the phases by a sharp interface.

Depending on the flow simulation, different types of models are available. The main distinction of the models is given below.

Lagrange models solve a separate equation for individual particles, bubbles, or droplets in a surrounding fluid. The method integrates the three-dimensional trajectories of the particles based on the forces acting on them from the surrounding fluid and other sources. Turbulence is usually accounted for by a random motion, superimposed on the trajectory.

Lagrange models are usually applied to flows with low particle (bubble) densities. In these flows, the interaction between the particles can usually be neglected, thereby reducing the complexity of the problem significantly.

The Euler-Euler formulation is based on the assumption of interpenetrating continua. A separate set of mass, momentum, and energy conservation equations is solved for each phase. Interphase transfer terms have to be modeled to account for the interaction of the phases. Euler-Euler methods can be applied to separated and dispersed flows by changing the interface transfer model.

Additional models are required for flows with mass transfer between the phases (condensation, evaporation, boiling). These models can be applied in the form of correlations for a large number of particles (bubbles) in a given control volume, or directly at the interface between the resolved phase boundary.

6.3.5. Reduction of Application Uncertainties

Application uncertainties cannot always be avoided because the missing information can frequently not be recovered. The uncertainty can be minimized by interaction with the supplier of the test case. The potential uncertainties have to be documented before the start of the CFD application.

In the case that the assumptions have to be made concerning any input to a CFD analysis, they have to be communicated to the partners in the project. Alternative assumptions have to be proposed and the sensitivity of the solution to these assumptions has to be evaluated by case studies (alteration of inflow profiles, different locations for arbitrary boundary conditions, and so on).

Recommendations are:

- Identify all uncertainties in the numerical setup:
  - Geometry reduction
  - Boundary condition assumptions
  - Arbitrary modeling assumptions, for example, bubble diameter, and so on.
- Perform a sensitivity analysis with at least two settings for each arbitrary parameter.
- Document the sensitivity of the solution on the assumptions.

6.3.6. CFD Simulation

This section provides recommendations concerning the optimal application of a CFD method, once the grids are available and the basic physical models have been defined.

6.3.6.1. Target Variables

In order to monitor numerical errors, it is recommended that you define target variables. The convergence of the numerical scheme can then be checked more easily and without interpolation between different grids. You should select target variables that:

1. Are representative of the goals of the simulation.
2. Are sensitive to numerical treatment and resolution.

   This criteria should help to avoid the use of measures that are insensitive to the resolution, such as pressure-based variables in boundary layer simulations.
3. Can be computed with existing post-processing tools.
4. Can be computed inside the solver and displayed during run-time (optimal).

It is optimal if the variable can be computed during run-time and displayed as part of the convergence history. This enables you to follow the development of the target variable during the iterative process.

6.3.6.2. Minimizing Iteration Errors

A first indication of the convergence of the solution to steady state is the reduction in the residuals. Experience shows, however, that different types of flows require different levels of residual reduction. For example, it is found regularly that swirling flows can exhibit significant changes even if the residuals are reduced by more than 5 - 6 orders of magnitude. Other flows are well converged with a reduction of only 3 - 4 orders.

In addition to the residual reduction, it is therefore required to monitor the solution during convergence and to plot the pre-defined target quantities of the simulation as a function of the residual (or the iteration number). A visual observation of the solution at different levels of convergence is recommended.

It is also recommended that you monitor the global balances of conserved variables, such as mass, momentum and energy, vs. the iteration number.

Convergence is therefore monitored and ensured by the following steps:

- Reduce residuals by a pre-specified level and provide residual plots.
- Plot evolution of r.m.s. and maximum residual with iteration number.
- Report global mass balance with iteration number.
- Plot target variables as a function of iteration number or residual level.
- Report target variables as a function of r.m.s. residual (table).

It is desirable to have the target variable written out at every timestep in order to display it during the simulation run.

Depending on the numerical scheme, the recommendations may also be relevant to the iterative convergence within the timestep loop for transient simulations.

6.3.6.3. Minimizing Spatial Discretization Errors

Spatial discretization errors result from the numerical order of accuracy of the discretization scheme and from the grid spacing. It is well known that only second- and higher-order space discretization methods are able to produce high quality solutions on realistic grids. First-order methods should therefore be avoided for high quality CFD simulations.

As the order of the scheme is usually given (mostly second-order), spatial discretization errors can be influenced only by the provision of an optimal grid. It is important for the quality of the solution and the applicability of the error estimation procedures defined in *Solution Error Estimation* (p. 89), that the coarse grid already resolves the main features of the flow. This requires that the grid points are concentrated in areas of large solution variation. For the reduction of spatial discretization errors, it is also important to provide a high-quality numerical grid.

For grid convergence tests, the simulations are carried out for a minimum of three grids. The target quantities will be given as a function of the grid density. In addition, an error estimate based on the definition given in *Solution Error Estimation* (p. 89) (*Equation 6–25* (p. 91)) will be carried out. It is also recommended that you compute the quantity given by *Equation 6–27* (p. 92) to test the assumption of asymptotic convergence.

It is further recommended that the graphical comparison between the experiments and the simulations show the grid influence for some selected examples. The following information should be provided:

- Define target variable as given in *Target Variables* (p. 101).
- Provide three (or more) grids using the same topology (or for unstructured meshes, a uniform refinement over all cells).
- Compute solution on these grids:
  - Ensure convergence of the target variable in the time- or iteration domain. See *Iteration Errors* (p. 88) and *Minimizing Iteration Errors* (p. 102).
  - Compute target variables for these solutions.
- Compute and report error measure for target variable(s) based on *Equation 6–25* (p. 91).
- Plot selected variables for the different grids in one picture.
- Check if the solution is in the asymptotic range using *Equation 6–27* (p. 92).

6.3.6.4. Minimizing Time Discretization Errors

In order to reduce time integration errors for unsteady-state simulations, it is recommended that you use at least a second-order-accurate time-discretization scheme. Usually, the relevant frequencies can be estimated beforehand and the timestep can be adjusted to provide at least 10 - 20 steps for each period of the highest relevant frequency. In case of unsteadiness due to a moving front, the timestep should be chosen as a fraction of:

$$\Delta t \sim \frac{\Delta x}{U} \qquad \textbf{(6–28)}$$

with the grid spacing Δx and the front speed U.

It should be noted that under strong grid and timestep refinement, sometimes flow features are resolved that are not relevant for the simulation. An example is the (undesirable) resolution of the vortex shedding at the trailing edge of an airfoil or a turbine blade in a *RANS* (p. 92) simulation for very fine grids and timesteps. Another example is the gradual switch to a *DNS* (p. 84) for the simulation of free surface flows with a **Volume of Fluid** (VOF) method (for example, drop formation, wave excitation for free surfaces, and so on). This is a difficult situation, as it usually means that no grid/timestep converged solution exists below the *DNS* (p. 84) range, which can usually not be achieved.

In principle, the time dependency of the solution can be treated as another dimension of the problem with the definitions in *Solution Error Estimation* (p. 89). However, a four-dimensional grid study would be very demanding. It is therefore more practical to carry out the error estimation in the time domain separately from the space discretization. Under the assumption that a sufficiently fine space discretization is available, the error estimation in the time domain can be performed as a one-dimensional study.

Studies should be carried out with at least two and if possible three different timesteps for one given spatial resolution. Again, the error estimators given in *Solution Error Estimation* (p. 89) (*Equation 6–25* (p. 91)) can be used, if Δ is replaced by the timestep. The following information should be provided:

- Unsteady target variables as a function of timestep (graphical representation)
- Error estimate based on *Equation 6–25* (p. 91) for (time averaged) target variables
- Comparison with experimental data for different timesteps.

6.3.6.5. Avoiding Round-Off Errors

Round-off errors are usually not a significant problem. They can occur for high-Reynolds number flows where the boundary layer resolution can lead to very small cells near the wall. The number of digits of a single precision simulation can be insufficient for such cases. The only way to avoid round-off errors with a given CFD code is the use of a double precision version. In case of an erratic behavior of the CFD method, the use of a double precision version is recommended.

6.3.7. Handling Software Errors

Software errors can be detected by verification studies. They are based on a systematic comparison of CFD results with verified solutions (in the optimal case analytical solutions). It is the task of the software developer to ensure the functionality of the software by systematic testing.

In most cases, pre-existing software will be used. It is assumed that all CFD packages have been sufficiently tested to ensure that no software verification studies have to be carried out in the project (except for newly developed modules). In case that two CFD packages give different results for the same application, using the same physical models, the sources for these differences will have to be evaluated. In case of code errors, they will be reported to the code developers and if possible removed.

6.4. Selection and Evaluation of Experimental Data

Because of the necessity to model many of the unresolved details of technical flows, it is necessary to assess the accuracy of the CFD method with the help of experimental data. Experiments are required for the following tasks and purposes:

- Verification of model implementation
- Validation and calibration of statistical models
- Demonstration of model capabilities.

There is no philosophical difference between the different types of test cases. The same test case can be used for the different phases of model development, implementation, validation, and application, depending on the status of the model and the suitability of the data.

6.4.1. Verification Experiments

The purpose of verification tests is to ensure the correct implementation of all numerical and physical models in a CFD method. The best verification data would be the analytical solutions for simple cases that allow testing all relevant implementation aspects of a CFD code and the models implemented. As analytical solutions are not always available, simple experimental test cases are often used instead.

6.4.1.1. Description

For CFD code verification, convergence can be tested against exact analytical solutions like:

- Convection of disturbances by a given flow
- Laminar Couette flow
- Laminar channel flow.

For the verification of newly implemented models, verification can only in limited cases be based on analytical solutions. An example is the terminal rise velocity of a spherical bubble in a calm fluid.

In most other cases, simple experiments are used for the verification. It is recommended that you compute the test cases given by the model developer in the original publication of the model, or other trustworthy publications. Quite often experimental correlations can be applied, without the need for comparison with one specific experiment. For instance for turbulence model verification, the most frequently used correlations are those for flat plate boundary layers.

6.4.1.2. Requirements

The only requirement for verification data is that they allow a judgement of the correct implementation of the code and/or the models. This requires information from other sources concerning the performance of the model for the test case. Strictly speaking, it is not required that the simulations are in good agreement with the data, but that the differences between the simulations and the data are as expected.

The test suite for model verification must be diverse enough to check all aspects of the implementation. As an example, a fully developed channel flow does not allow to test the correct implementation of the convective terms of a transport equation. The test suite should also allow testing the correct interaction of the new model with existing other features of the software.

Software verification for physical models should be carried out in the same environment that the end-user has available. Testing of the new features in an expert environment might miss some of error sources, such as the *GUI* (p. 94).

Verification cases should be selected before the model is implemented. They must be considered an integral part of the model implementation.

6.4.2. Validation Experiments

The purpose of validation tests is to check the quality of a statistical model for a given flow situation. Validation tests are the only method to ensure that a new model is applicable with confidence to certain types of flows. The more validation tests a model passes with acceptable accuracy, the more generally it can be applied to industrial flows. The goal of validation tests is to minimize and quantify modeling errors. Validation cases are often called building block experiments, as they test different aspects of a CFD code and its physical models. The successful simulation of these building blocks is a prerequisite for a complex industrial flow simulation.

6.4.2.1. Description

Examples of validation cases are flows with a high degree of information required to test the different aspects of the model formulation. In an ideal case, a validation test case should be sufficiently complete to allow for an improvement of the physical models it was designed to evaluate. Increasingly, validation data are obtained from *DNS* (p. 84) studies. The main limitation here is in the low-Reynolds number and the limited physical complexity of *DNS* (p. 84) data. Typically, validation cases are geometrically simple and often based on two-dimensional or axisymmetric geometries.

6.4.2.2. Requirements

Validation cases are selected to be as close as possible to the intended application of the model. As an example, the validation of a turbulence model for a flat plate boundary layer does not ensure the applicability of the model to flows with separation (as is known from the $k-\varepsilon$ model). It is well accepted by the CFD community and by model developers that no model (turbulence, multi-phase or other) will be able to cover all applications with sufficient accuracy. This is the reason why there are always multiple models for each application. The validation cases allow the CFD user to select the most appropriate model for the intended type of application.

Test case selection requires that the main features of the CFD models that are to be tested be clearly identified. They must then be dominant in the validation case. Validation cases are often 'single physics' cases, but it will be more and more necessary to validate CFD methods for combined effects.

The requirements for validation cases are that there should be sufficient detail to be able to compute the flow unambiguously and to evaluate the performance of the CFD method for a given target application.

Completeness of information is one of the most important requirements for a validation test case. This includes all information required to run the simulation, like:

- Geometry
- Boundary conditions
- Initial conditions (for unsteady flows)
- Physical effects involved.

While the first three demands are clearly necessary to be able to set up and run the simulation, the knowledge of all physical effects taking place in the experiment is not always considered. However, it is crucial to have a clear understanding of the overall flow in order to be able to judge the quality of a test case. Typical questions are:

- Is the flow steady-state or does it have a large-scale unsteadiness?
- Is the flow two-dimensional (axisymmetric, for example)?
- Are all the relevant physical effects known (multi-phase, transition, and so on)?
- Have any corrections been applied to the data and are they appropriate?
- Was there any measurement/wind or water tunnel interference?

Completeness of information is also essential for the comparison of the simulation results with the experimental data. A validation case should have sufficient detail to identify the sources for the discrepancies between the simulations and the data. This is a vague statement and cannot always be ensured, but a validation experiment should provide more information than isolated point measurements. Profiles and distributions of variables at least in one space dimension should be available (possibly at different locations). More desirable is the availability of field data in two-dimensional measuring planes including flow visualizations.

Completeness also relates to the non-dimensionalization of the data. Frequently the information provided is not sufficient to reconstruct the data in the form required by the validation exercise.

In case that the data provided are not sufficient, the impact of the missing information has to be assessed. Most crucial is the completeness of the data required to set up the simulation. In case of missing information, the influence of this information deficit has to be assessed. Typical examples are incomplete inlet boundary conditions. While the mean flow quantities are often provided, other information required by the method, as profiles for turbulent length scales and volume fractions is frequently missing. The importance of this deficit can be estimated by experience with similar flows and by sensitivity studies during the validation exercise.

Next to the completeness of the data, their quality is of primary importance for a successful validation exercise. The quality of the data is mainly evaluated by error bounds provided by the experimentalists. Unfortunately, most experiments still do not provide this information. Moreover, even if error estimates are available, they cannot exclude systematic errors by the experimentalist.

In addition to error bounds, it is therefore desirable to have an overlap of experimental data that allow testing the consistency of the measurements. Examples are the same data from different experimental techniques. It is also a quality criterion when different experimental groups in different facilities have carried out the same experiment. Consistency can also be judged from total balances, like mass, momentum and energy conservation. Quality and consistency can frequently be checked if validation exercises have already been carried out by other CFD groups, even if they used different models.

The availability of the data has to be considered before any CFD validation is carried out. This includes questions of ownership. For most CFD code developers, data that cannot be shown publicly are much less valuable than freely available experimental results.

6.4.3. Demonstration Experiments

The purpose of a demonstration exercise is to build confidence in the ability of a CFD method to simulate complex flows. While validation studies have shown for a number of building block experiments that the physical models can cover the basic aspects of the target application, the demonstration cases test the ability of a method to predict combined effects, including geometrical complexity.

6.4.3.1. Description

For an aerodynamic study, a typical hierarchy would be:

- Verification - Flat plate
- Validation - Airfoil or wing
- Demonstration - Complete aircraft.

Similar hierarchies can be established for other industrial areas.

6.4.3.2. Requirements

Typically, the detail of the experimental data is much lower than for verification or validation cases.

Completeness of information to set up the test case is of similar importance as for validation cases and involves the same aspects as listed below:

- Geometry
- Boundary conditions
- Initial conditions (for unsteady flows)
- Physical effects involved.

Typically, the level of completeness of the data for demonstration cases is much lower than for validation cases. It is therefore even more essential to identify the missing information and to carry out sensitivity studies with respect to these data.

In terms of post-processing, demonstration cases often do not provide a high degree of detail. They are usually not appropriate to identify specific weaknesses in the physical models or the CFD codes. Typically, only the point data or global parameters, as efficiencies, are provided.

Even though the density of data is usually lower, the quality should satisfy the same criteria as for validation cases. Error estimates are desirable and so are independent measurements.

Due to the limited amount of data available, the information is usually not sufficient to carry out consistency checks.

The requirements in terms of availability/openness are usually lower than for validation cases, as the demonstration applies usually to a smaller audience. A demonstration case might be carried out for a single customer or one specific industrial sector. It has to be ensured, as in all cases, that the data can be shown to the target audience of the simulation.

Chapter 7: CFX Best Practices Guide for Cavitation

This guide is part of a series that provides advice for using CFX in specific engineering application areas. It is aimed at users with moderate or little experience using CFX for applications involving cavitation.

This guide describes *Liquid Pumps* (p. 109).

Cavitation is the formation of vapor bubbles within a liquid where flow dynamics cause the local static pressure to drop below the vapor pressure. The bubbles of vapor usually last a short time, collapsing when they encounter higher pressure. Cavitation should not be confused with boiling. When a liquid boils, thermal energy drives the local temperature to exceed the saturation temperature.

Cavitation is a concern in several application areas, including pumps, inducers, marine propellers, water turbines, and fuel injectors. One of the major problems caused by cavitation is a loss of pressure rise across a pump. Other problems include noise, vibration, and damage to metal components.

The next section discusses the effects of cavitation on the performance of liquid pumps.

7.1. Liquid Pumps

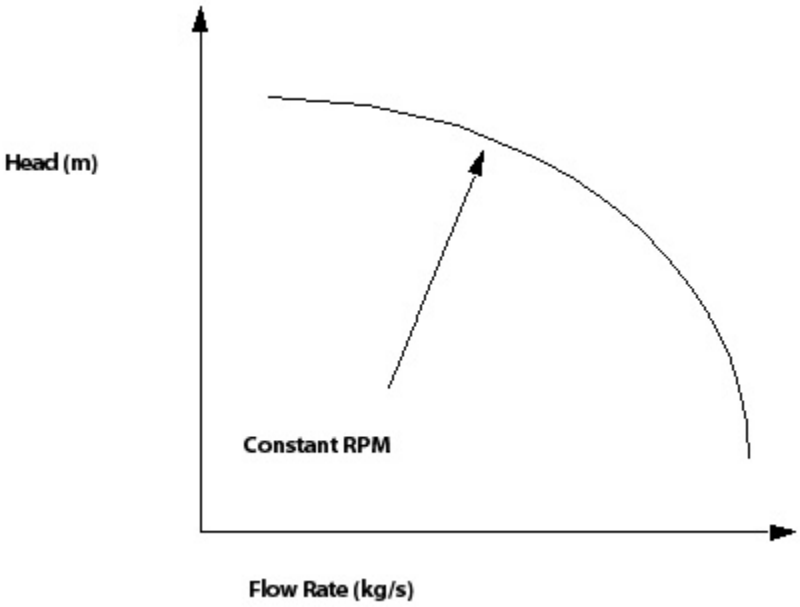
Water pumps must take in water and deliver it at a higher total pressure with an acceptable flow rate. Under certain conditions, cavitation may occur on the low pressure side of the pump, causing a loss of pressure rise and/or flow rate.

Both pump performance without cavitation and the affects of cavitation on performance will be discussed.

7.1.1. Pump Performance without Cavitation

As long as the static pressure remains sufficiently high everywhere in the system, cavitation will not occur. In this case, for a given pump RPM, the pressure rise and flow rate are directly coupled, and can be plotted in a pump performance diagram, as shown below.

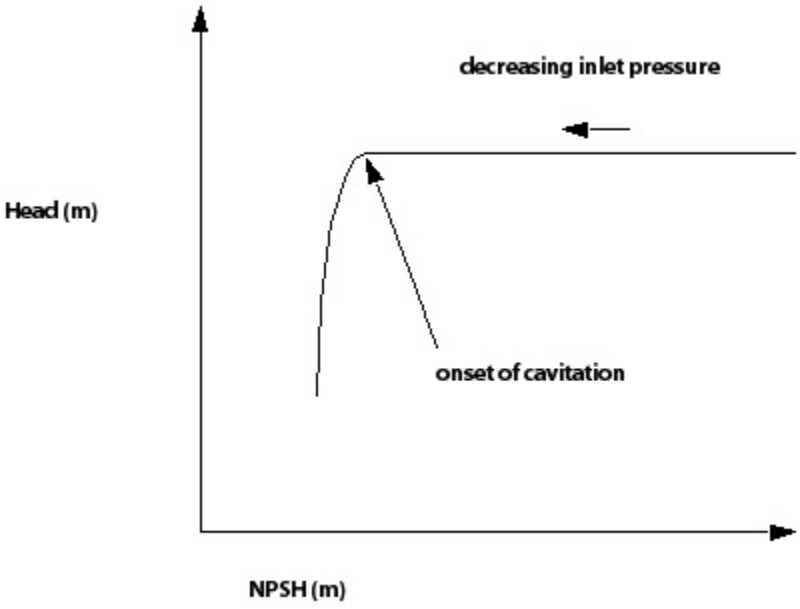
Figure 7.1 Flow Rate vs Pressure Rise for a Liquid Pump



7.1.2. Pump Performance with Cavitation

If the inlet total pressure is below the critical value for a particular flow rate and RPM, cavitation will occur causing the pressure rise to diminish. The following performance diagram shows the effect of cavitation on pressure rise.

Figure 7.2 Cavitation Performance at Constant RPM and Flow Rate



NPSH is the Net Positive Suction Head, a quantity directly related to the inlet total pressure by the relation:

$$NPSH = \left(\frac{p_{T,\text{inlet}} - p_v}{\rho g} \right) \quad \textbf{(7–1)}$$

where $p_{T,\text{inlet}}$ is the inlet total pressure, p_v is the vapor pressure, ρ is density, and g is the acceleration due to gravity. As the inlet total pressure drops, so does the NPSH value, and the amount of cavitation increases.

To generate this diagram, RPM and flow rate are fixed, and the pressure rise is measured at progressively lower inlet total pressures. For the part of the test where the inlet total pressure is sufficiently high to prevent cavitation, the pressure rise across the pump is constant, equal to the amount predicted by the first performance diagram. This results in a horizontal trend in the performance curve as the inlet total pressure is dropped. Because the pressure rise remains constant, the total pressure at the outlet drops by the same amount as at the inlet. Using CFX software, a mass flow outlet boundary condition can be specified to fix the flow rate while the inlet total pressure is varied.

When the inlet total pressure reaches a sufficiently low value, cavitation occurs. A further reduction in inlet total pressure causes more cavitation, which almost always causes a large loss of pressure rise. In rare cases, pressure rise can actually increase slightly with small amounts of cavitation. Even in such cases, however, a further increase in cavitation causes a sudden loss of pressure rise. In the lab, the pressure rise will eventually become insufficient to maintain the required flow rate. Using CFX software, the solution will eventually fail to converge. Before this point, data should be collected with a sufficient resolution (sufficiently small changes in inlet pressure) to resolve the part of the performance curve where the pressure starts to drop. The point of cavitation is often marked by the NPSH at which the pressure rise has fallen by a few percent.

7.1.3. Procedure for Plotting Performance Curve

1. Set up a simulation with cavitation turned on and pressure levels set high enough to avoid levels of cavitation that significantly affect pressure rise.

   If you have trouble getting a converged solution, try running a simulation with cavitation turned off, then use the result as an initial guess for a simulation with cavitation turned on.
2. Run the solver to obtain a solution.
3. Calculate the pressure rise across the pump and the NPSH value, then plot a point in the performance diagram.
4. Lower the pressure boundary condition by about 5% to 10%.
5. Repeat starting from step 2, using the previous solution as the initial guess, until cavitation has caused a significant loss of pump head.

7.1.4. Setup

To facilitate setting up typical domain settings for the cavitation of water, you may load a single-domain mesh, then run the template `.ccl` file:

`CFX/etc/model-templates/cavitating_water.ccl`

This file should be examined in a text editor before using it so that you understand which settings it specifies.

For the domain fluids list, specify both a liquid and a vapor for the same material. In most cases, it is sufficient to use a constant-density vapor.

Under **Fluid Models** for the domain, it is strongly recommended that you select **Homogeneous Model** under **Multiphase Options**. You do not need to select **Free Surface Model** for the purpose of simulating cavitation.

Under **Fluid Pairs** for the domain, select the fluid pair from the list and, for **Mass Transfer**, set **Option** to `Cavitation`. Select the **Rayleigh Plesset** cavitation model or a **User Defined** Cavitation Model. For the **Rayleigh Plesset** model, the **Mean Diameter** for cavitation bubbles represents the mean nucleation site diameter and must be specified. The default value of 2e-06 m is a reasonable value in most cases. The **Saturation Pressure** must be defined unless the materials you have selected are the components of a homogeneous binary mixture. In the latter case, the saturation properties will already be defined in the mixture definition, but you may still choose to override the **Saturation Pressure** by specifying it on the **Fluid Pairs** tab.

When initializing the domain, set the volume fraction of vapor to zero and the volume fraction of liquid to unity. These settings (represented by the `Automatic` setting for **Volume Fraction**) are used by default in CFX.

Set up the problem with one of the following boundary condition combinations:

1. Inlet total pressure and outlet mass flow (recommended)
2. Inlet velocity profile and outlet static pressure

The inlet boundary condition should specify that the volume fraction of vapor is zero.

Turbulence models should be chosen as usual (e.g., k-epsilon or SST). For turbulence induced cavitation, consider using the DES model.

For advection scheme, use high resolution, or a specified blend factor of unity.

If editing a material, remember that the vapor pressure is on an absolute scale; it does not depend on the specified reference pressure.

Cavitation models cannot be combined with other types of interphase mass transfer, such as thermal phase changes.

7.1.5. Convergence Tips

If performing a single solution, initially turn off cavitation, then turn on cavitation and use the first set of results as an initial guess.

If performing a series of simulations using one solution to initialize the next, solve the cases in order of decreasing pressure (i.e., approaching cavitation).

7.1.6. Post-Processing

A contour plot of volume fraction for the vapor can show where cavitation bubbles exist.

To calculate the inlet and outlet pressures, use the function calculator.

Chapter 8: CFX Best Practices Guide for Combustion

This guide is part of a series that provides advice for using CFX in specific engineering application areas. It is aimed at users with moderate or little experience using CFX for applications involving combustion.

This guide describes:

- *Gas Turbine Combustors* (p. 113)
- *Combustion Modeling in HVAC Cases* (p. 115)

8.1. Gas Turbine Combustors

Gas turbines are widely used in stationary and aircraft applications. The combustor receives the working fluid in a compressed state, burns fuel to increase its temperature, and passes it to the turbine. One of the key design goals for the combustor is to achieve a stable combustion process. Another key design goal is to minimize the emission of pollutants, particularly oxides of nitrogen.

8.1.1. Setup

8.1.1.1. Steady State vs. Transient

Most simulations are steady-state, particularly for stationary gas turbines that operate at a constant load.

8.1.1.2. Turbulence Model

The $k-\varepsilon$ turbulence model is used in many applications, but the SST model should be considered for flows with separated boundary layers, and the Reynolds Stress Model is the best choice for highly swirling flows.

8.1.1.3. Reference Pressure

Because of the high inlet pressure, a reference pressure between 4 and 20 atmospheres is common, and depends upon the type of simulation you are running.

8.1.1.4. Combustion Model

The choice of combustion model depends of whether the fuel/oxidant combination is premixed. The following table outlines some of the differences.

Premixed Combustion	Non-Premixed Combustion
Commonly used for recent stationary gas turbines in power generation.	Typically used for flight engines because it is easier to control variable operating conditions.

Premixed Combustion	Non-Premixed Combustion
Combustion Models: EDM with product limiter and/or extinction submodels, FRC/EDM combined, Partially Premixed (Turbulent Flame Speed Closure (TFC)) Note that the EDM model usually needs adjusting for premixed combustion (for example, extinction by temperature or by mixing/chemical time scales).	Combustion Models: EDM, FRC/EDM combined, Flamelet (and, for some cases, also the Premixed model).

For the preliminary analysis of high speed turbulent flow, the Eddy Dissipation combustion model is a sensible choice, but cannot simulate burning velocities or flame position.

The Laminar Flamelet model is applicable for turbulent flow with non-premixed combustion, and provides a robust solution at a low computational expense for multi-step reactions. The Flamelet model uses a chemistry library, meaning that only two additional transport equations are solved to calculate multiple species. As a result, the Flamelet model is a very good choice for modeling the formation of various pollutants during the combustion process. The Flamelet model predicts incomplete combustion to some extent (CO in exhaust gas), which helps to predict reduction in temperature (unlike EDM).

8.1.2. Reactions

During the initial analysis of a combustor, the highest values of temperature and outgoing heat flux are likely to be of primary concern. For this purpose, a single-step Eddy Dissipation reaction can be used. Such a reaction is likely to overpredict the temperature, and will not predict emissions correctly, but can provide a conservative indicator of the expected temperature levels.

Other reaction steps might then be added to the simulation to account for the formation of combustion byproducts. Each reaction step has its own separate time scale. As a result, convergence can become very difficult when a multi-step reaction contains more than about 5 steps.

8.1.3. Convergence Tips

The Equation Class Settings tab in CFX-Pre can be used to set different advection schemes and time scales for each class of equation you are solving. For multi-step Eddy Dissipation reactions, convergence can be improved by temporarily increasing the mass fraction time scale by a factor of about 5-10.

For the Eddy Dissipation Model, multistep convergence can be aided by first running a simplified single-step simulation and using the results from the run as an initial values file for a multi-step run.

You may restart a Flamelet model from a cold solution. You should avoid restarting with the Flamelet model from an EDM solution. You may restart an EDM case from a Flamelet model solution.

The High Resolution advection scheme is always recommended for combustion simulations because it is bounded and prevents over/undershoots. Care must be taken, however, to provide a mesh of sufficient quality to resolve most of the flow features. A very poor mesh will result in the scheme using a blend factor close to zero (therefore not providing a solution as accurate as expected).

For simulations which include Finite Rate Chemistry, small temperature variations can result in large changes in reaction rate. When a solution is converging, temperature values may change sufficiently

to make the solution unstable. To aid convergence, add a TEMPERATURE DAMPING CCL structure within a SOLVER CONTROL block, as follows:

```
FLOW:
  SOLVER CONTROL:
    TEMPERATURE DAMPING:
       Option = Temperature Damping
       Temperature Damping Limit = <Real number>
       Under Relaxation Factor = <Real number>
    END
  END
END
```

Depending on the location of the SOLVER CONTROL block, the temperature damping may be applied to a particular domain or phase. Set the Temperature Damping Limit to 0 so that positive damping is always applied. The Under Relaxation Factor can be set to multiply changes in temperature by a value between 0 and 1. You should try a factor of 0.2 if you are having trouble converging a solution.

8.1.4. Post-Processing

Some of the most common plots to create in CFD-Post include:

- Mass Fractions: fuel, O_2, products, intermediate species (CO), pollutants (NO)
- Turbulent Mixing Time Scale (Eddy Dissipation / Turbulent Kinetic Energy)
- Reaction Rates

  The variable "<my reaction>.Molar Reaction Rate" is available for every "Single Step" reaction (EDM, FRC or combined model).
- Plots of the turbulent Damköhler number (the ratio of the turbulent time scale to the chemical time scale)

8.2. Combustion Modeling in HVAC Cases

This section deals with the setup of combustion cases for HVAC simulations, where it is important to accurately model the combustion process. Such processes are known as "combusting fire" simulations, as opposed to "inert fire" simulations.

Using a combusting fire simulation is the most accurate way to model fires in all HVAC cases. It is particularly important in cases when the fire is under-ventilated, or when the ventilation cannot be easily predicted. The drawback is the additional computational expense involved in solving a full combustion model as part of the main solution.

8.2.1. Set Up

Most simulations are set up as transient. The choice of timestep is generally model dependent, but will usually fall into the range 0.5 s to 2 s. The Total Energy heat transfer model should be selected to fully model buoyancy.

Note

When modeling buoyancy, it is very important to correctly specify the buoyancy reference density when opening boundary conditions are used.

The RNG $k-\varepsilon$ turbulence model is a good choice for combusting flows, with either no buoyancy terms in equations or buoyancy terms in both the equations (with C3=1). The SST model is also reasonable, but may not converge well for natural convection. The SSG model is accurate, but convergence may be very slow.

The most common fuels used are hydrocarbons such as methane, diesel and petroleum. Cellulosic materials, plastics and wood are also used. The simulation will dictate the type of materials to use as a fuel.

The Eddy Dissipation Model is widely used, combined with an Additional Variable for toxins. The flamelet model is more rigorous, and is a better choice when the fire is under ventilated.

8.2.2. Convergence Tips

Convergence can be slowed if care is not taken in the setup of buoyancy and openings.

The presence of an instantaneous fuel supply is sometimes not physical, and can slow convergence. In many transient cases, the amount of fuel available can be controlled by using time-dependent expressions.

8.2.3. Post-processing

The most common parameters of interest in a combusting fire model are simulation-dependent, but will usually include one of more of the following:

- Temperature
- Products (including carbon monoxide and other toxins)
- Visibility
- Wall Temperature
- Wall convective and radiative heat fluxes

Chapter 9: CFX Best Practices Guide for HVAC

This guide discusses best practices for setting up, solving, and post processing an HVAC simulation:

This guide is part of a series that provides advice for using CFX in specific engineering application areas. It is aimed at users with moderate or little experience using CFX for HVAC[1] applications.

9.1. HVAC Simulations

HVAC studies range in scale from single components (such as a radiator) to large and complicated systems (such as a climate control system for a large building).

Physical processes that are commonly modeled in HVAC simulations include:

- Buoyancy
- Thermal radiation
- Conjugate heat transfer (CHT) between fluids and solids.

Typical HVAC systems include the following components:

- Heating/cooling units such as furnaces, heaters, air conditioners, and radiators
- Fans/pumps
- Thermostats.

9.1.1. Setting Up HVAC Simulations

This section discusses how to set up various physical processes, CFD features, and components involved in HVAC simulations.

9.1.1.1. Buoyancy

Most HVAC cases involve flow that is affected by buoyancy. Buoyancy can be activated on the **Basic Settings** tab of the **Domain** details view.

Two buoyancy models are available: Full and Boussinesq. These models are automatically selected according to the properties of the selected fluid(s).

- The Full buoyancy model is used if fluid density is a function of temperature and/or pressure (which includes all ideal gases and real fluids). In this case, a Buoyancy Reference Density must be set as the expected average density of the domain.
- The Boussinesq model is used if fluid density is not a function of temperature or pressure. In this case, a Buoyancy Reference Temperature must be set as the expected average temperature of the domain.

[1] HVAC is a reference to Heating, Ventilation (or Ventilating), and Air Conditioning. Often, it is also used as a reference to refrigeration.

When fluid properties are functions of pressure, the absolute pressure is used to evaluate them. The calculation of absolute pressure requires a Buoyancy Reference Location to be defined, preferably by specifying coordinates.

When modeling fire, it is recommended that you choose a compressible fluid because density variations will be significant. An incompressible fluid should be chosen only if density variations are small (a few percent or less).

9.1.1.2. Thermal Radiation

To set the radiation model for a fluid domain, visit the **Fluid Models** panel for that domain and set the following:

9.1.1.2.1. Thermal Radiation Model

For HVAC studies, select either **Monte Carlo** or **Discrete Transfer**. If directed radiation is to be modeled, Monte Carlo must be used.

9.1.1.2.2. Spectral Model

Select either **Gray** or **Multiband**. Spectral bands are used to discretize the spectrum and should therefore be able to adequately resolve all radiation quantities that depend on wavelength (or frequency or wave number). For HVAC, two bands will usually suffice.

9.1.1.2.3. Scattering Model

A scattering model should not be used if you are modeling clear air. The isotropic scattering model should be used if you are modeling air that contains dust or fog.

To set up radiation for a solid domain, visit the **Solid Models** panel for that domain (each solid domain must be set up separately). The only radiation model available for solid domains is Monte Carlo.

Note

If any solid domain uses the Monte Carlo radiation model (that is, if it uses radiation at all), then all fluid domains using a radiation model must use the Monte Carlo model.

The material used in a domain that transmits radiation has radiation properties that specify **Absorption Coefficient**, **Scattering Coefficient**, and the **Refractive Index**. These properties may be edited in the **Materials** details view.

Note

Radiation modeling cannot be used with Eulerian multiphase simulations.

Thermal radiation properties are specified on the **Boundary Details** panel for each boundary of a domain that transmits radiation. For opaque surfaces, the properties that must be specified are: **Emissivity** and **Diffuse Fraction**. For inlets, outlets, and openings, you may specify either the **Local Temperature** or an **External Blackbody Temperature**.

The Monte Carlo and Discrete Transfer models allow radiation sources to be specified on the **Sources** panel for any subdomain or wall boundary. For subdomains, radiation sources per unit volume are

specified; for boundaries, radiation fluxes are specified. Radiation sources may be directed or isotropic. Multiple isotropic sources and up to one directed source may be specified for any given wall boundary or subdomain.

Material properties related to radiation, thermal radiation properties for boundaries, and source strengths can be specified as expressions that depend on one or more of the built-in variables: **Wavelength in Vacuum** (or **wavelo**), **Frequency** (or **freq**), **Wavenumber in Vacuum** (or **waveno**).

A domain representing an opaque solid should not have a radiation model set. The boundaries of radiation-transmitting domains that interface with such a solid domain should be specified as opaque.

External windows of a room can be modeled as solid domains which interface with the room (air) domain; they may also be modeled as an external boundary of the room domain. In either case, the exterior boundary must be modeled as an opaque wall. A diffuse and a directed radiation source emitted from the opaque surface can be used to simulate sunlight. In order to simulate the motion of the sun, the direction vector for directed radiation can be specified by CEL expressions that depend on time (**t**). Radiation escaping through a window can be modeled by specifying a non-zero emissivity (to cause radiation absorption) and either:

- Specifying a heat transfer coefficient via a CEL expression that accounts for the thermal energy lost
- Specifying a fixed wall temperature.

When using solid domains that transmit radiation, a spectral radiation model is recommended. If a simulation contains no solid domains that transmit radiation, a gray radiation model can be used for rough calculations but a spectral model should be used for more detailed modeling.

9.1.1.3. CHT (Conjugate Heat Transfer) Domains

CHT domains are solid domains that model heat transfer. In CFX, all solid domains must model heat transfer, and are therefore CHT domains. If you do not want to model heat transfer in a particular region, do not assign the mesh for that region to any domain.

Boundaries between domains that model heat transfer have temperatures and thermal fluxes calculated automatically, and should not have thermal boundary conditions specified. External boundaries (which can represent solids that are not explicitly modeled) require the specification of a thermal boundary condition.

Boundary conditions other than thermal boundary conditions (for example, wall roughness) may be specified on the boundaries of a fluid domain that interface with a solid domain.

Sources of thermal energy and/or radiation can be added to a subdomain of a CHT domain.

9.1.1.4. Mesh Quality

Ensure that wall boundary layers have adequate mesh resolution. This is important regardless of the type of wall heat transfer: adiabatic, specified temperature, specified heat flux, or heat transfer coefficient.

The mesh resolution in a boundary layer affects the prediction of convective heat transfer and the temperature gradient near the wall. For walls without a specified temperature, the temperature gradient near the wall affects the calculated wall temperature and, consequently, the amount of radiation emitted (provided that the emissivity of the wall is non-zero).

9.1.1.5. Fans

Fans should be represented by momentum sources if they are embedded in the domain. Fans can also be represented by an inlet or outlet boundary condition or both.

9.1.1.6. Thermostats

A Thermostat can be defined using a User Fortran routine. Refer to Air Conditioning Simulation in the *CFX Tutorials* for details.

9.1.1.7. Collections of Objects

If your HVAC simulation models a large number of people/equipment/items, consider volumetric sources of heat, CO_2, and resistances.

9.2. Convergence Tips

Buoyancy and coupling between the relevant equations often make convergence difficult. Smaller timesteps may help convergence.

Chapter 10: CFX Best Practices Guide for Multiphase

This guide is part of a series that provides advice for using CFX in specific engineering application areas. It is aimed at users with moderate or little experience using CFX for applications involving multiphase flows.

In the context of CFX, a multiphase flow is a flow composed of more than one fluid. Each fluid may possess its own flow field, or all fluids may share a common flow field. Unlike multicomponent flow[1], the fluids are not mixed on a microscopic scale; rather, they are mixed on a macroscopic scale, with a discernible interface between the fluids. CFX includes a variety of multiphase models to allow the simulation of multiple fluid streams, bubbles, droplets, and free surface flows.

This guide describes:

10.1. Bubble Columns

Bubble columns are tall gas-liquid contacting vessels and are often used in processes where gas absorption is important (for example, bioreactors to dissolve oxygen in broths) and to limit the exposure of micro-organisms to excessive shear imparted by mechanically driven mixers. There are two types of bubble columns in general use: airlift reactors that use a baffle to separate the riser and downcomer regions, and other columns that do not use a baffle.

10.1.1. Setup

The choice of a steady state or transient simulation depends on the type of simulation you want to analyze. For example, an analysis using a steady-state simulation is often satisfactory for monitoring global quantities. A transient simulation can be used to observe transient effects, such as recirculation zones.

Most bubble columns use two fluids: one continuous fluid and one dispersed fluid. The $k-\varepsilon$ model is typically used in the continuous fluid, and the dispersed phase zero equation is used for the dispersed phase.

Non-drag forces become less significant with increasing size of the bubble column. For smaller columns, non-drag forces may be significant.

The Grace drag model is recommended, especially for modeling air/water.

A degassing boundary condition is generally employed at the top of the bubble column. The degassing boundary behaves as an outlet boundary to the dispersed phase, but as a wall to the continuous phase.

[1] Note that a fluid in a multiphase flow may be a multicomponent mixture.

A reasonable estimate of the time scale is given by a factor of the length of the domain divided by the velocity scale (for example, 0.5 * L/U).

10.1.2. Convergence Tips

Sometimes, physical instabilities (such as recirculation zones) can result in slow or stalled convergence. In these cases, you can still obtain an indicator of convergence for a global quantity by creating a monitor point at some point in the domain. As a result, you can determine whether values for selected global quantities (such as gas hold-up) are meaningful.

10.1.3. Post-Processing

The main design objective for bubble columns is efficient mixing, which is strongly influenced by the dispersed phase. Mixing efficiency can be measured in a number of ways. One example is to measure the gas hold-up in the riser as a function of the superficial gas velocity. This would require solving for the gas volume fraction for a number of simulations, each with a different mass flow rate of the dispersed phase at the sparger. Another option would be to use the same input parameters, this time measuring the liquid velocity in the downcomer.

10.2. Mixing Vessels

Mixing vessels are widely used in the chemical industry to optimize mixing and/or heat transfer between fluids. Mixing must be efficient, precise and repeatable to ensure optimum product quality. Quantities of interest may include mixing times, gas hold-up, power draw, local shear and strain rates, and solids distribution. The application of Computational Fluid Dynamics to address these needs results in faster and lower cost design through reduced experimentation, more reliable scale-up, and better understanding of the processes, leading to higher yields and reduced waste.

10.2.1. Setup

Mixing vessels generally use two domains. The impeller domain is a small, rotating domain which encloses the impeller. The rest of the tank is represented by a stationary domain. Different types of domain interfaces are available for the connection between the stationary and the rotating domains. The recommended types are:

- Frozen Rotor: faster but cruder
- Transient: slower (transient analysis) but much more accurate

The choice of a steady state or transient simulation is dependent on the type of interface that exists between the two domains. Where a Frozen Rotor interface is used, a steady state simulation is usually carried out. Performing a transient simulation allows you to use the transient rotor/stator frame change model to account for transient effects.

The initial guess for velocity can be set to zero in the relative frame for each domain.

10.3. Free Surface Applications

Free Surface flow refers to a multiphase situation where the fluids (commonly water and air) are separated by a distinct resolvable interface. Such flows occur, for example, around the hull of a ship, or in a breaking wave.

10.3.1. Setup

The choice of using a steady-state or transient simulation is problem-dependent. There are two models available for free surface flow: homogeneous and inhomogeneous. The homogeneous model can be used when the interface between the two phases remains well defined and none of the dispersed phase becomes entrained in the continuous phase. An example of homogeneous free surface flow is flow in an open channel. A breaking wave is one example of an inhomogeneous flow case.

The same choice of turbulence model as for single phase simulations is appropriate. When using the inhomogeneous model, you should use the homogeneous turbulence option in CFX-Pre. The Buoyancy Reference Density should be set to the density of the least dense fluid.

When setting boundary conditions, the volume fractions can be set up using step functions to set the liquid height at each boundary. An outlet boundary having supercritical flow should use the **Supercritical** option for **Mass And Momentum**. This requires that you set the relative pressure of the gas above the free surface at the outlet.

For most free surface applications, the initial conditions can use step functions to set the volume fractions of the phases as a function of height. The initial condition for pressure should be set to hydrostatic conditions for the specified volume fraction initialization and the buoyancy reference density.

The timestep for free surface flows should be based on a L/U (Length/Velocity) scale. The length scale should be a geometric length scale. The velocity scale should be the maximum of a representative flow velocity and a buoyant velocity, which is given by:

$$U_{\text{buoyant}} = \sqrt{g\,L}$$

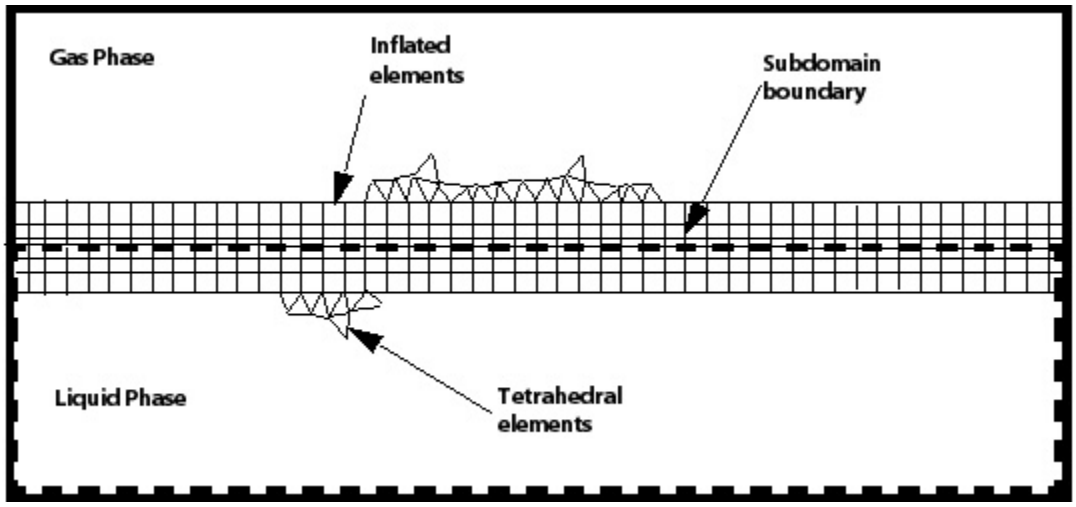
In addition, it is sometimes helpful to reduce the timestep for the volume fraction equations by an order of magnitude below that of the other equations.

10.3.2. Convergence Tips

The interface between the liquid and gas phase can sometimes become blurry. This could be due to physical properties (such as a breaking wave or sloshing in a vessel). Where the dispersed phase becomes entrained in the continuous phase, the inhomogeneous model is a better choice.

A technique to increase the mesh density in the region of a liquid-gas interface is to create a subdomain which occupies the same region as the liquid (or gas) phase, and inflate the mesh in both directions from the edge of the subdomain, as shown in *Figure 10.1* (p. 124). The inflation layers can increase the resolution in the region of the interface and enhance the results.

Figure 10.1 An exaggerated view of three inflation layers on each side of the uppermost subdomain boundary surface.



10.4. Multiphase Flow with Turbulence Dispersion Force

For Release 14, in order to reduce convergence difficulties encountered in some multiphase flow problems, the value of the expert parameter `ggi ap relaxation` is multiplied internally by 0.75. This occurs in the following situation only:

- Multiphase flow
- Nontrivial turbulence dispersion force included
- Coupled volume fraction solution algorithm.

In this situation, the default value of 1.0 is converted internally to 0.75. If you override the default with a smaller value, the new value is also multiplied internally by 0.75. This ensures that you retain some control over this parameter. The above restrictions ensure that the numerics changes are focused on a narrow range of problems and do not deteriorate convergence of other classes of problems.

Chapter 11: CFX Best Practices Guide for Turbomachinery

Turbomachinery applications can generally be divided into three main categories: gas compressors and turbines, liquid pumps and turbines, and fans and blowers. Each category is discussed in a separate section below.

This guide describes best practices for setting up simulations involving:

This guide is part of a series that provides advice for using CFX in specific engineering application areas. It is aimed at users with moderate or little experience using CFX for applications involving turbomachinery.

11.1. Gas Compressors and Turbines

This section describes:

11.1.1. Setup for Simulations of Gas Compressors and Turbines

Heat transfer and viscous work are involved, and can be modeled by using the Total Energy heat transfer model and enabling the Viscous Work Term option in CFX-Pre.

The industry-standard $k-\omega$ turbulence model is widely used, and the Shear Stress Transport model is also a good choice for these cases. When using the Shear Stress Transport model, ensure a resolution of the boundary layer of more than 10 points. For details, see The k-omega and SST Models in the *CFX-Solver Modeling Guide*.

A common boundary condition configuration is to specify the total pressure and total temperature at the inlet and the mass flow at the outlet. Other configurations are also commonly used.

A good estimate of the timestep is within the region $0.1/\omega$ to $1/\omega$, where ω is the angular velocity of the rotating domain in radians per second. Selecting an automatic timestep will result in a timestep of $0.2/\omega$.

The second-order high-resolution advection scheme is generally recommended.

11.1.2. Convergence Tips

For high speed compressors where a specified mass flow outlet boundary condition is applied, the flow can "choke" when the velocity approaches Mach 1, resulting in possible solver failure. A suggested workaround to this problem is to run the simulation using a static pressure outlet, which is more stable.

If you have trouble converging a simulation involving real gases, try to obtain a solution first using an ideal gas. Ideal gases are available in the real gas library.

If you have trouble converging a problem with many stages, you may find that solving for a reduced number of stages can give you a better understanding of the physics, and may allow you to restart a multi-stage problem with a better initial guess. You can also try ramping up boundary conditions and the RPM.

Low pressure ratio Gas compressors (1.1 or less) can be treated more like liquid pumps. For details, see *Liquid Pumps and Turbines* (p. 127).

11.1.3. Post-Processing

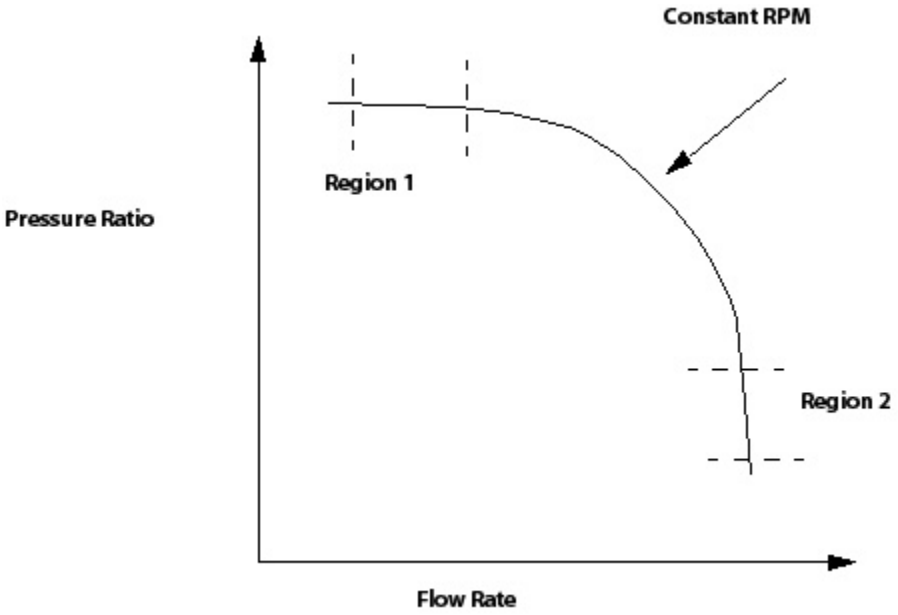
CFD-Post offers a powerful array of post-processing tools for turbomachinery applications, including turbo-specific plots and performance calculation macros. To use many of the Turbo Post tools, you must first initialize each domain by specifying the locations of turbo regions and instancing information.

The Turbo Calculator from the Turbo menu in CFD-Post allows you to perform calculations on the type of application you are modeling. The macro prints a report in HTML showing a number of calculated variables, including torque, head and flow coefficients, blade loading, and efficiency.

You can also create your own macros to customize post-processing using Power Syntax, which is based on the Perl language.

The optimal performance characteristics can be determined by creating a curve of pressure ratio versus flow rate.

Figure 11.1 Flow Rate vs Pressure Rise for a Gas Compressor



In *Figure 11.1* (p. 126), Region 1 shows an area where a large change in mass flow rate represents a small change in pressure rise. When modeling flow in this region, a mass flow rate specified outlet is better than a pressure specified outlet. Region 2 shows an area where a small change in flow rate represents a large pressure variation. This region is close to "choking", and a pressure-specified outlet is the best choice. For details, see *Convergence Tips* (p. 126).

If a total pressure inlet boundary condition is used (recommended), it will also provide a useful starting point for streamlines that are colored by total pressure during post-processing. The uniform total pressure distribution means lines will begin with a uniform color. It may be harder to visually resolve these pressure values if an inlet velocity profile is used.

CFD-Post provides a performance macro for gas compressors and turbines.

11.2. Liquid Pumps and Turbines

This section describes:

- *Setup for Simulations of Liquid Pumps and Turbines* (p. 127)
- *Convergence Tips* (p. 127)
- *Post-Processing* (p. 128)

11.2.1. Setup for Simulations of Liquid Pumps and Turbines

Heat transfer is not significant in most cases, so the heat transfer option can be set to None in CFX-Pre.

The $k-\varepsilon$ and Shear Stress Transport models are appropriate choices for modeling turbulence. When using the Shear Stress Transport model, ensure a resolution of the boundary layer of more than 10 points. For details, see The k-epsilon Model in the *CFX-Solver Modeling Guide*.

When setting boundary conditions, a total pressure specified inlet and a mass flow outlet are a recommended practice. The total pressure inlet condition is often more appropriate than the uniform velocity or massflow inlet condition for cases that assume that the machine is drawing fluid directly from a static reservoir.

As with gas compressors, a good estimate of the timestep is within the region $0.1/\omega$ to $1/\omega$, where ω is the angular velocity of the rotating domain in radians per second. Selecting an automatic timestep will result in a timestep of $0.2/\omega$.

The high-resolution advection scheme is recommended.

This document deals with obtaining solutions for cases without cavitation, but cavitation may be present. For advice on how to deal with cavitation, see *CFX Best Practices Guide for Cavitation* (p. 109).

11.2.2. Convergence Tips

When only a poor initial guess is available, it may be helpful to first run with a specified mass flow inlet and a static pressure outlet. The outlet pressure in this case is fairly arbitrary and is usually set at, or close to zero to reduce round-off error. The specification of a mass flow inlet may be more robust. However, a mass flow inlet assumes a uniform inlet velocity which may not be appropriate. Once the overall flow is established, the boundary conditions may then be changed to total pressure at the inlet and mass flow at the outlet.

11.2.3. Post-Processing

If a total pressure inlet boundary condition is used (recommended where possible), it will also provide a useful starting point for streamlines that are colored by total pressure during post-processing. The uniform total pressure distribution means lines will begin with a uniform color. It may be harder to visually resolve these pressure values if an inlet velocity profile is used.

CFD-Post provides a performance macro for liquid pumps and turbines.

11.3. Fans and Blowers

This section describes:

11.3.1. Setup for Simulations of Fans and Blowers

Fans and blowers behave like liquid pumps, and require a similar model setup. The flow is generally modeled as incompressible and isothermal. The fluid is typically air as a general fluid or as an ideal gas at a specified temperature. The $k-\varepsilon$ or SST model is used to model turbulence.

Boundary conditions, turbulence models and choice of timestep are the same as for liquid pumps and turbines.

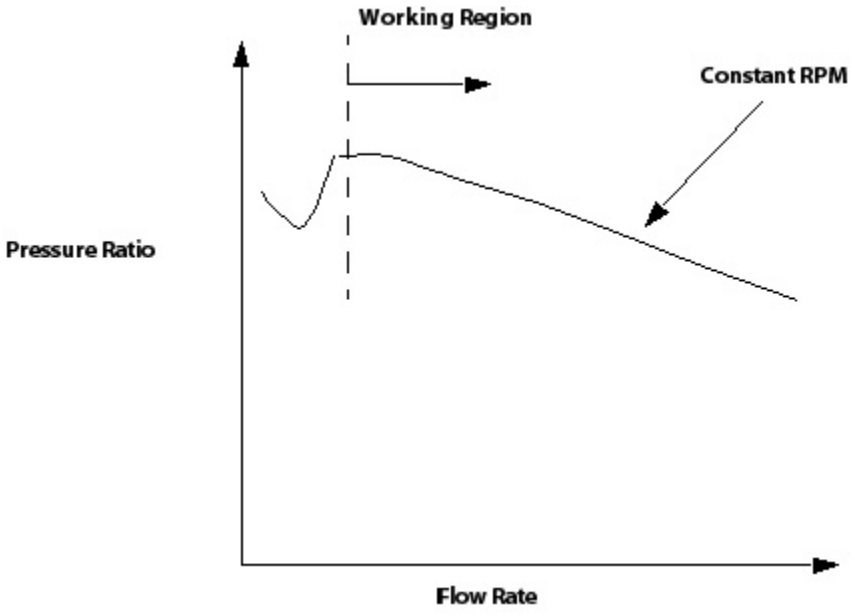
11.3.2. Convergence Tips

The use of the alternate rotation model is an important consideration when modeling fans and blowers. Where long axisymmetric inlets exist, the absolute frame velocity has less swirl than the relative frame velocity. Because the alternate rotation model solves for the absolute frame velocity, it can reduce numerical error in such inlet sections. The model may introduce errors in the exit stream if the flow is highly swirling. Hence, the length of the inlet and exit sections can be an important factor when choosing whether to implement the model. The alternate rotation model is generally recommended, especially for axial fans. In most realistic flow situations, this model reduces (or at least will not increase) numerical errors.

Air foil drag is significant and boundary layer friction is an important modeling issue for fans and blowers. A good resolution of the boundary layer, requiring a high concentration of nodes close to the blade surfaces, is therefore important. The Shear Stress Transport model can provide relatively accurate results where the boundary layer is sufficiently resolved by the mesh.

11.3.3. Post-Processing

A similar post-processing approach to pumps and turbines is also useful for fans and blowers. For details, see *Post-Processing* (p. 128). See the following figure for a plot of flow rate vs pressure rise for a blower.



11.4. Frame Change Models

When specifying domain interfaces in CFX-Pre, you must select the type of analysis that will be carried out in the solver. The choices are:

- *Frozen Rotor* (p. 129)
- *Stage* (p. 129)
- *Transient Rotor-Stator* (p. 130)

11.4.1. Frozen Rotor

The Frozen Rotor model treats the flow from one component to the next by changing the frame of reference while maintaining the relative position of the components. Usually, periodicity is used to reduce the number of components to a subset that has approximately the same pitch ratio as the full geometry. To account for differences in pitch ratio between the subset and the full geometry, the flow passing through the interface is scaled according to the net pitch ratio of the subsets.

The Frozen Rotor model must be used for non-axisymmetric flow domains, such as impeller/volute, turbine/draft tube, propeller/ship and scroll/volute cases. It can also be used for axial compressors and turbines. The Frozen Rotor model has the advantages of being robust, using less computer resources than the other frame change models, and being well suited for high blade counts. The drawbacks of the model include inadequate prediction of physics for local flow values and sensitivity of the results to the relative position of the rotor and stator for tightly coupled components.

11.4.2. Stage

The Stage model circumferentially averages the fluxes in bands and transmits the average fluxes to the downstream component. Possible applications include axial turbines, compressors and pumps, as well as fans and torque converters. The model is useful for large pitch ratios and still takes a relatively short time to solve. The model is not suitable for applications with tight coupling of components and/or significant wake interaction effects and may not accurately predict loading.

11.4.3. Transient Rotor-Stator

The Transient Rotor-Stator model takes into account all of the transient flow characteristics. A sliding interface is used to allow a smooth rotation between components. As with the Frozen Rotor model, the Transient Rotor-Stator model scales the flow from one component to the next in order to account for a non-unity net pitch ratio. This model is robust and yields high accuracy predictions of loading. The drawbacks include high computational cost and large amounts of storage required to hold the transient data.

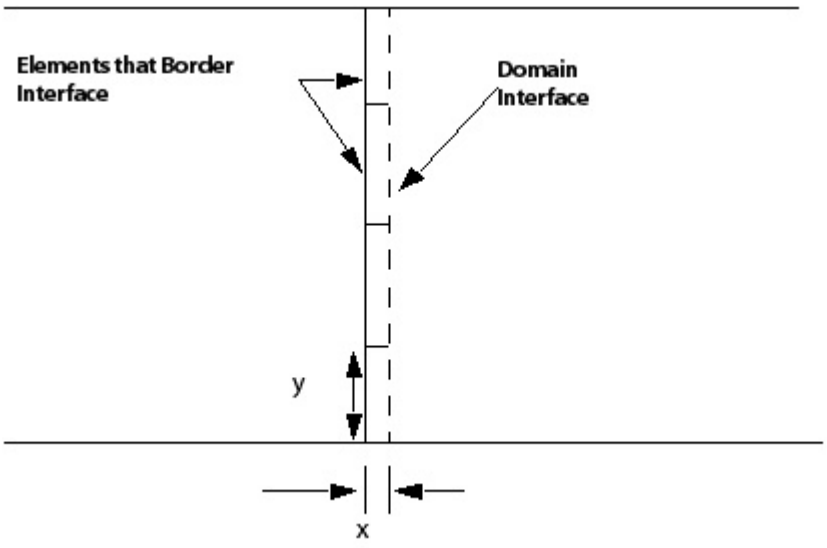
11.5. Domain Interface Setup

The setup of domain interfaces is an important consideration when defining a problem. The following section outlines some approved practices for use in turbomachinery applications.

11.5.1. General Considerations

- Domain interfaces should typically be placed midway between the rotor and stator for turbomachinery cases.
- To avoid numerical errors, the aspect ratio of elements on the domain interface should be between 0.1:1 and 10:1, as measured by x/y in *Figure 11.2* (p. 130).
- Where circular domain interfaces exist, they must be axisymmetric in shape as shown in *Figure 11.3* (p. 131).

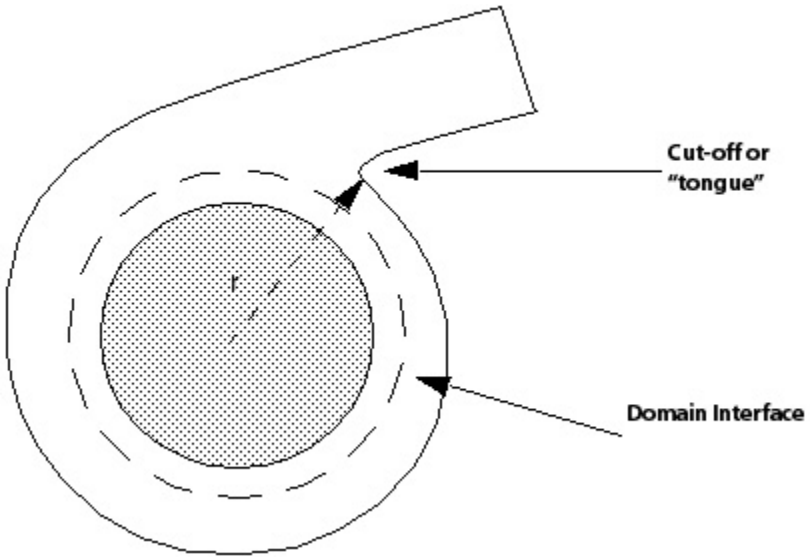
Figure 11.2 Element Aspect Ratio at Domain Interface



11.5.2. Case 1: Impeller/Volute

A basic impeller/volute sketch is shown in *Figure 11.3* (p. 131). The edge of the inner circle shows the maximum extent of the impeller blades. A good practice here is to create the domain interface halfway across the narrowest gap between the blade and volute wall. This usually occurs around the cut-off or "tongue" illustrated in the diagram.

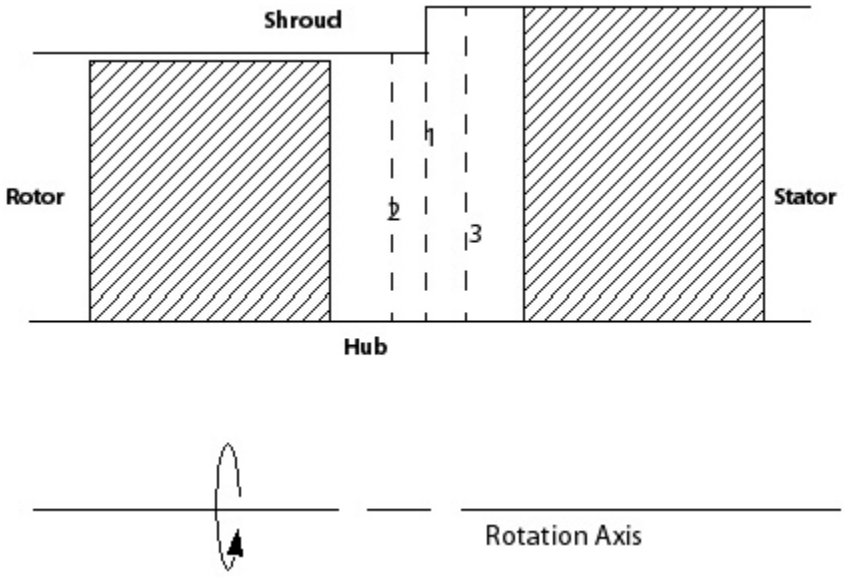
Figure 11.3 Impeller/Volute



11.5.3. Case 2: Step Change Between Rotor and Stator

For the case shown, there is a step change in the passage height between the rotor and stator. A common choice for placement of the interface would be choice 1. However, take care with this setup because the non-overlap regions of the interface should be specified as walls. A better alternative may be to use a domain interface upstream or downstream of the step change, at position 2 or position 3.

Figure 11.4 Possible Domain Interface Positions with Step Change in Passage Height

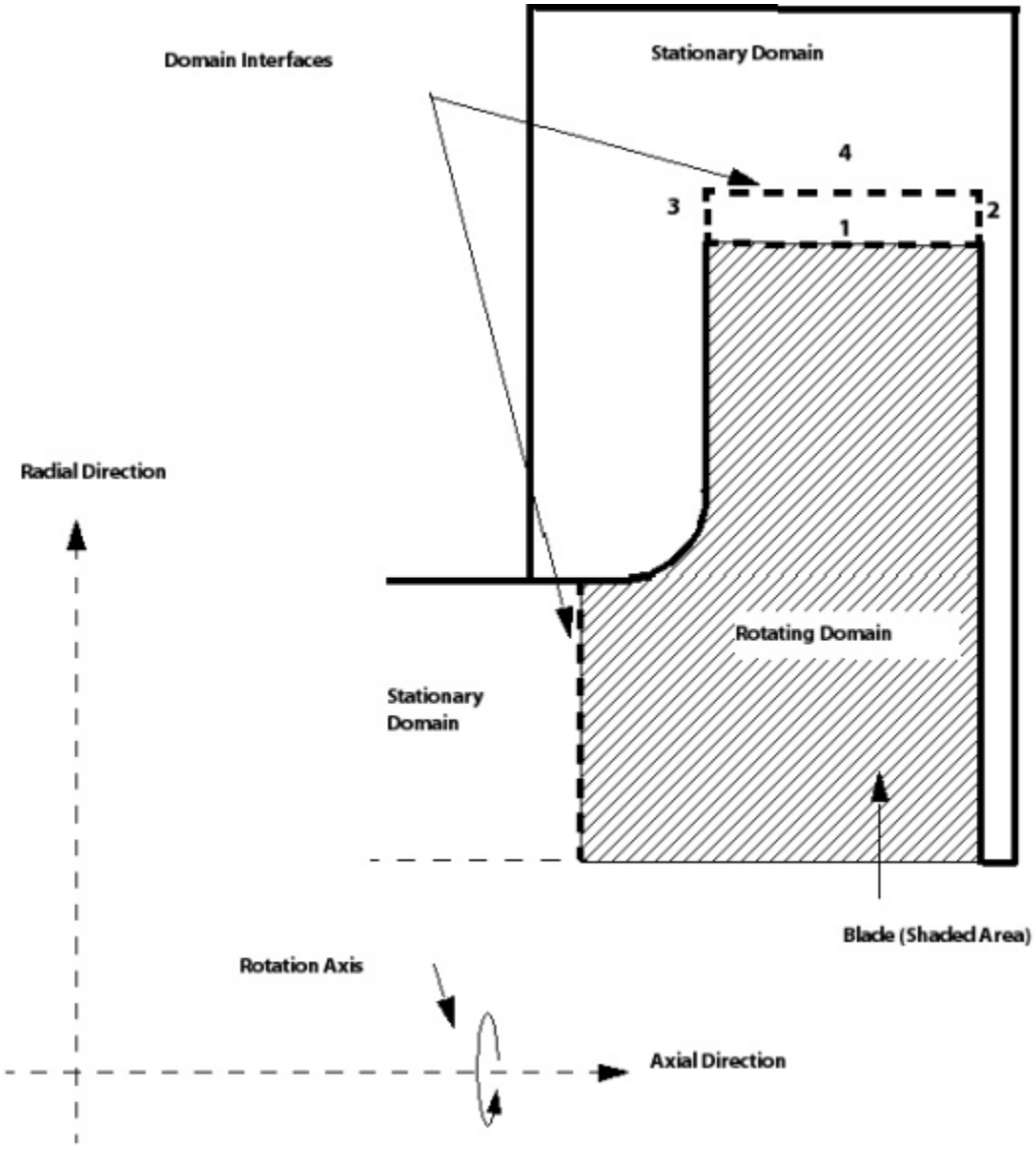


11.5.4. Case 3: Blade Passage at or Close to the Edge of a Domain

Figure 11.5 (p. 132) shows a blade which extends to the edge of the rotating domain. Although it is convenient to place a domain interface at the blade edge (1), this can result in unrealistic results (The area of the interface would be reduced on one side where the interface is displaced by the blade edge, resulting in an inaccurate pitch change calculation. Also, in the case of a stage interface, the wake would

be mixed out at the trailing edge.) A better arrangement is to extend the rotating domain away from the blade edge. Domain Interfaces can then be created at (2), (3), and (4).

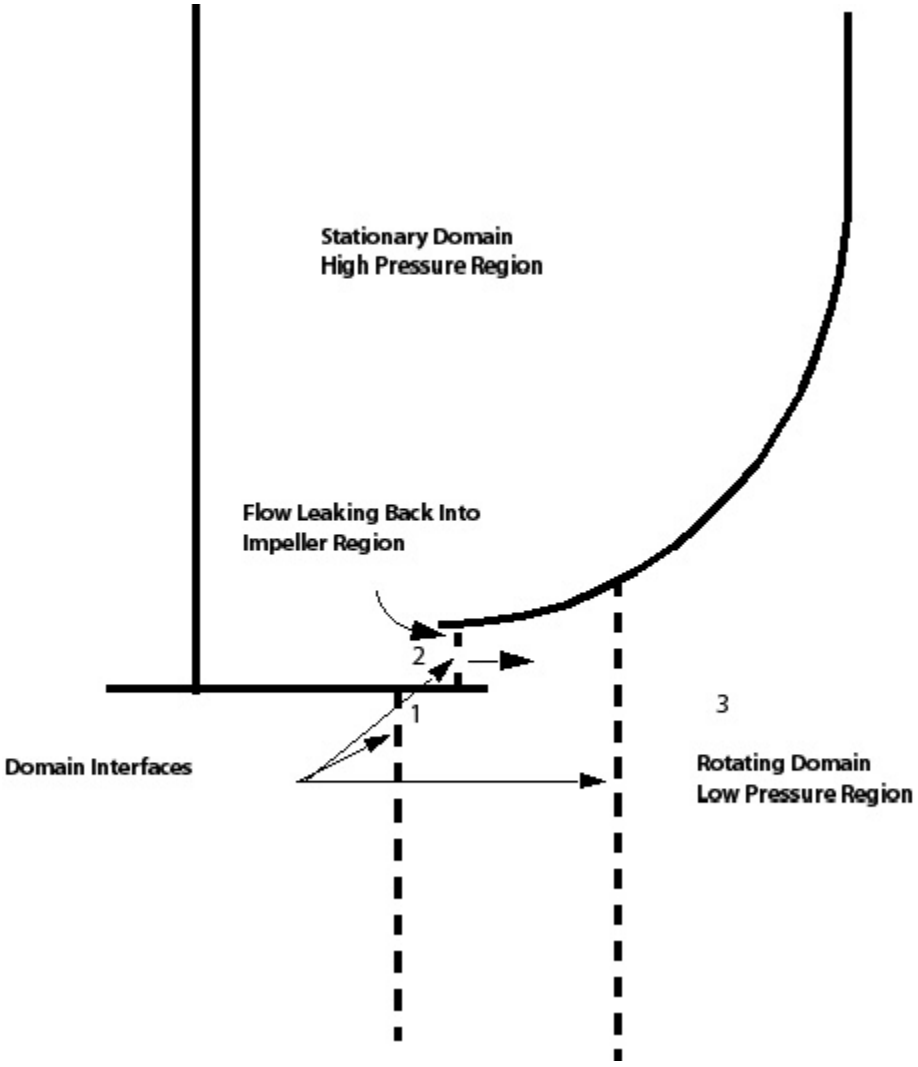
Figure 11.5 Radial Compressor



11.5.5. Case 4: Impeller Leakage

A close-up view of part of *Figure 11.5* (p. 132), which models flow leaking from a volute back into the impeller region. To model the feature, you can use two domain interfaces (at positions 1 and 2), or a single domain interface downstream of the leak (position 3).

Figure 11.6 Flow Leakage Through Gap Near Impeller Inlet



11.5.6. Case 5: Domain Interface Near Zone of Reversed Flow

Be wary of flow moving backwards across stage or frozen rotor interfaces. Because of the approximations implied by these interfaces, flow moving upstream and downstream on the same interface will lead to unphysical results. Try relocating the interface to prevent this from occurring.

As an example, *Figure 11.7* (p. 134) shows two blade rows of an axial machine with a frozen rotor interface between them. The flow moves from left to right everywhere except in a small region just downstream of the trailing edge of the first row of blades. In this case, the domain interface, shown as a dashed line, should be located to the right of this region, as shown.

Figure 11.7 Domain Interface Between Blade Rows in an Axial Machine

Chapter 12: CFX Command Language (CCL)

The CFX Command Language (CCL) is the internal communication and command language of ANSYS CFX. It is a simple language that can be used to create objects or perform actions in the post-processor. All CCL statements can be classified into one of three categories:

- Object and parameter definitions, which are described in Object Creation and Deletion.
- CCL actions, which are commands that perform a specific task (such as reading a session file) and which are described in Command Actions in the *CFD-Post User's Guide*.
- Power Syntax programming, which uses the Perl programming language to allow loops, logic, and custom macros (subroutines). Power Syntax enables you to embed Perl commands into CCL to achieve powerful quantitative post-processing. For details, see *Power Syntax in ANSYS CFX* (p. 229).

State files and session files contain object definitions in CCL. In addition, session files can also contain CCL action commands. You can view and modify the CCL in these files by using a text editor.

For more information, see Object Creation and Deletion.

12.1. CFX Command Language (CCL) Syntax

The following topics will be discussed:

12.1.1. Basic Terminology

The following is an example of a CCL object that defines an isosurface.

```
ISOSURFACE: Iso1
   Variable = Pressure
   Value = 15000 [Pa]
   Color = 1,0,0
   Transparency = 0.5
END
```

- `ISOSURFACE` is an object type
- `Iso1` is an object name
- `Variable = Pressure` is a parameter
- `Variable` is a parameter name
- `Pressure` is a parameter value
- If the object type does not need a name, it is called a singleton object. Only one object of a given singleton type can exist.

12.1.2. The Data Hierarchy

Data is entered via parameters. These are grouped into objects that are stored in a tree structure.

```
OBJECT1: object name
   name1 = value
   name2 = value
END
```

Objects and parameters may be placed in any order, provided that the information is set prior to being used further down the file. If data is set in one place and modified in another, the latter definition overrides the first.

In CFD-Post, all object definitions are only one object level deep (that is, objects contain parameters, but not other objects).

12.1.3. Simple Syntax Details

The following applies to any line that is not a Power Syntax or action line (that is, the line does not start with a `!` or `>`).

12.1.3.1. Case Sensitivity

Everything in the file is sensitive to case.

Case sensitivity is not ideal for typing in many long parameter names, but it is essential for bringing the CFX Expression Language (CEL) into CCL. This is because some names used to define CCL objects (such as `Fluids`, `Materials` and `Additional Variables`) are used to construct corresponding CEL names.

For simplicity and consistency, the following is implemented:

- Singletons and object types use upper case only.
- Parameter names, and predefined object names, are mixed case. The CFX Expression Language tries to follow the following conventions:
  1. Major words start with an upper case letter, while minor words such as prepositions and conjunctions are left in lower case (for example, `Mass Flow in`).
  2. Case is preserved for familiar names (for variables `k` or `r`), or for abbreviation RNG.

- User object names conventions can be chosen arbitrarily by you.

12.1.3.2. CCL Names Definition

Names of singletons, types of object, names of objects, and names of parameters all follow the same rules:

- In simple syntax, a CCL name must be at least one character. This first character must be alphabetic; there may be any number of subsequent characters and these can be alphabetic, numeric, space or tab.
- The effects of spaces in CCL names are:
  - Spaces appearing before or after a name are not considered to be part of the name.
  - Single spaces appearing inside a name are significant.
  - Multiple spaces and tabs appearing inside a name are treated as a single space.

12.1.3.3. Indentation

Nothing in the file is sensitive to indentation, but indentation can be used for easier reading.

12.1.3.4. End of Line Comment Character

The # character is used for this. Any text to the right of this character will be treated as comments. Any characters may be used within comments.

12.1.3.5. Continuation Character

If a line ends with the character \, the following line will be linked to the existing line. There is no restriction on the number of continuation lines.

12.1.3.6. Named Objects

A named object consists of an object type at the start of a line, followed by a : and an object name. Subsequent lines may define parameters and child objects associated with this object. The object definition is terminated by the string END on a line by itself.

Object names must be unique within the given scope, and the name must not contain an underscore.

12.1.3.7. Singleton Objects

A singleton object consists of an object type at the start of a line, followed by a :. Subsequent lines may define parameters and child objects associated with this object. The object definition is terminated by the string END on a line by itself.

The difference between a singleton object and a named object is that (after the data has been processed), a singleton can appear just once as the child of a parent object. However, there may be several instances of a named object of the same type defined with different names.

12.1.3.8. Parameters

A parameter consists of a parameter name at the start of a line followed by an = character followed by a parameter value. A parameter may belong to many different object types. For example, `U Velocity`

`= 1.0 [m/s]` may belong to an initial value object and `U Velocity = 2.0 [m/s]` may belong to a boundary condition object. Both refer to the same definition of U velocity in the rules file.

12.1.3.9. Lists

Lists are used within the context of parameter values and are comma separated.

12.1.3.10. Parameter Values

All parameter values are initially handled as data of type String, and should first of all conform to the following definition of allowed String values:

12.1.3.10.1. String

- Any characters can be used in a parameter value.
- String values or other parameter type values are normally unquoted. If any quotes are present, they are considered part of the value. Leading and trailing spaces are ignored. Internal spaces in parameter values are preserved as given, although a given application is free to subsequently assume a space condensation rule when using the data.
- The characters $ and # have a special meaning. A string beginning with $ is evaluated as a Power Syntax variable, even if it occurs within a simple syntax statement. This is useful for performing more complex Power Syntax variable manipulation, and then using the result as part of a parameter or object definition. The appearance of # anywhere in the CCL file denotes the start of a comment.
- The characters such as [,],{ and } are special only if used in conjunction with $. Following a $, such characters terminate the preceding Perl variable name.
- Other characters that might be special elsewhere in power syntax are escaped automatically when they appear in parameter values. For example, @, % and & are escaped automatically (that is, you do not need to precede these characters with the escape character \ when using them in parameter values).
- Parameter values can contain commas, but if the string is processed as a List or part of a List then the commas may be interpreted as separators (see below under List data types).

Some examples of valid parameter values using special characters in power syntax are:

```
Estimated cost = \$500
Title = Run\#1
Sys Command = "echo 'Starting up Stress solver' ; fred.exe &"
Pressure = $myArray[4]
Option = $myHash{"foo"}
Fuel = C${numberCatoms}H${numberHatoms}
```

Parameter values for data types other than String will additionally conform to one of the following definitions.

12.1.3.10.2. String List

A list of string items separated by commas. Items in a String List should *not* contain a comma unless contained between parentheses. One exception can be made if the String List to be is interpreted as a Real List (see below). Otherwise, each item in the String List follows the same rules as String data.

```
names = one, two, three, four
```

12.1.3.10.3. Integer

Sequence of digits containing no spaces or commas. If a real is specified when an integer is needed, the real is rounded to the nearest integer.

12.1.3.10.4. Integer List

List of integers, separated by commas.

12.1.3.10.5. Real

A single-precision real number that may be specified in integer, floating point, or scientific format, followed optionally by a dimension. Units use the same syntax as CEL.

Expressions are allowed to include commas inside function call argument lists.

Example usage:

```
a = 12.24
a = 1.224E01
a = 12.24 [m s^-1]
```

A real may also be specified as an expression such as:

```
a = myvel^2 + b
a = max(b,2.0)
```

12.1.3.10.6. Real List

List of reals, comma separated. Note that all items in the list must have the same dimensions. Items that are expressions may include commas inside function call argument lists, and the enclosed commas will be ignored when the list is parsed into individual items. Example usage:

```
a = 1.0 [m/s], 2.0 [m/s], 3.0 [m/s], 2.0*myvel, 4.0 [cm/s]
```

The list syntax `5*2.0` to represent 5 entries of the value 2.0 is not supported within CCL and hence within CFD-Post.

12.1.3.10.7. Logical

Several forms are acceptable: YES, TRUE, `1` or `ON` are all equivalent; `NO` or FALSE or `0` or `OFF` are all equivalent; initial letter variants Y, T, N, F are accepted (`0` is not accepted for `On`/`Off`); all case variants are accepted. Logical strings are also case insensitive (`YeS`, `nO`).

12.1.3.10.8. Logical List

List of logicals, separated by commas.

12.1.3.11. Escape Character

The \ character to be used as an escape character, for example, to allow $ or # to be used in strings.

Chapter 13: CFX Expression Language (CEL)

CFX Expression Language (CEL) is an interpreted, declarative language that has been developed to enable CFX users to enhance their simulations without recourse to writing and linking separate external Fortran routines.

You can use CEL expressions anywhere a value is required for input in ANSYS CFX.

CEL can be used to:

- Define material properties that depend on other variables.
- Specify complex boundary conditions.
- Add terms to the solved equations.

You can also monitor the value of an expression during the solution using monitor points.

Important

There is some CEL that works elsewhere in ANSYS CFX, but not in CFD-Post. Any expression created in CFX-Pre and used as a Design Exploration output parameter could potentially cause fatal errors during the Design Exploration run, so you should create all expressions for Design Exploration output parameters in CFD-Post.

This chapter describes:

13.1. CEL Fundamentals

The following topics will be discussed:

13.1.1. Values and Expressions

CEL can be used to generate both values and expressions. Values are dimensional (that is, with units) or dimensionless constants. The simplest type of definition is the dimensionless value, for example:

b = 3.743

You can also specify a value with units, for example:

g = 9.81 [m s^-2]

The dimensions of the quantities of interest for CFD calculations can be written in terms of mass, length, time, temperature and angle. The concept of units is fundamental to the behavior of values and expressions.

Values can be used directly, or they can be used as part of an expression. For example, you can use an expression to add two values together:

```
<Expr_1> = <Value_1> + <Value_2>
```

In this example, you may want to predefine <Value_1> and <Value_2>, but this is not required. However, in order to add two quantities together, *they must have the same dimension*; that is, it is meaningful to add a quantity in inches to one expressed in meters, but it is not meaningful to add one expressed in kilograms to one in square feet.

Expressions can also be functions of other (predefined) expressions:

```
<Expr_2> = <Expr_1> + <Value_3>
```

Units follow the conventions in the rest of CFX, in that a calculation has a set of solution units (by default, SI units), and that any quantity can be defined either in terms of the solution units, or any other set of units with the correct form.

An expression does not have its own units string, but if it references quantities that have dimensions, these will determine the resulting units for the expression. For example, if an expression depends inversely on the square of the x coordinate, then it has *implied dimensions* of length to the power -2.

13.1.1.1. Using Locators in Expressions

A CFX simulation has physics areas and mesh areas; physics areas are boundaries while mesh areas are regions. These two types of area can occupy completely different spaces in a simulation; however, there is no requirement that area names be unique between physics and mesh. This can lead to ambiguities when you use these names in expressions.

To avoid these ambiguities, ANSYS CFX first checks to see if "@<locator>" is a physics name; if this is not found, the name is checked in the list of mesh names. Thus if "in1" is both the name of a physics area and the name of a mesh area, "@<locator>" is taken to indicate the physics area.

ANSYS CFX also has @REGION CEL syntax so that you can identify a named area as being a mesh area. Thus to identify the mesh area in1, you would use the syntax:

```
@REGION:in1
```

Note that if <locator> does not appear as a physics name or a mesh name, the expression fails.

13.1.2. CFX Expression Language Statements

The CFX Expression Language is declarative. You declare the name and definition of the expression using expression language statements. The statements must conform to a predefined syntax that is similar to Fortran mathematical statements and to C statements for logical expressions.

The statement must consist of the following:

- a number, optionally with associated units. This defines a *constant*. Constants without units are termed *dimensionless*.
- for mathematical expressions, one or more references to mathematical constants, system variables, or existing user variables, separated by + (addition), - (subtraction), * (multiplication), / (division) and ^

(exponentiation), with optional grouping of these by parentheses. The syntax rules for these expressions are the same as those for conventional arithmetic.

- for logical expressions involving relational operators, one or more references to mathematical constants or results from mathematical expressions, separated by <= (is less than or equal to), < (is less than), == (is equal to), != (is not equal to), > (is greater than) and >= (is greater than or equal to) with optional grouping of these by parentheses.
- for logical expressions involving logical operators, one or more references to logical constants or results from relational operations separated by ! (negation), && (logical AND) and || (logical OR), with optional grouping by parentheses.

13.1.2.1. Use of Constants

Constants do not need to be defined prior to being used in an expression. For example, you could choose to evaluate the expression `x + 5 [m]`. Or, you could define a constant, `b = 5 [m]` and then create an expression `x + b`.

The logical constants are `false` and `true`. Results of logical expressions are either false or true, which are evaluated as `0` and `1` (corresponding to `false` and `true`, respectively) when a numerical representation is required.

The use of constants may be of benefit in generating complicated expressions or if you have several expressions that use the same constants.

13.1.2.2. Expression Syntax

All numbers are treated as real numbers.

The precedence of mathematical operators is as follows (from highest to lowest):

- The power operator ^ as in `x^y`.
- The unary minus or negation operator - as in `-x`.
- Multiplication and division as in `x*y/z`.
- Addition and subtraction as in `x+y-z`.

The precedence of logical and relational operators is as follows (from highest to lowest):

- The negation operator ! as in `!x`.
- The relational operators involving less than or greater than (<=, <, > and >=) as in `x >= y`.
- The relational operator is equal to and is not equal to (== and !=) as in `x != y`.
- The logical AND operator (&&) as in `x && y`.
- The logical OR operator (||) as in `x || y`.

13.1.2.3. Multiple-Line Expressions

It is often useful, particularly with complex expressions, to use more than one line when creating your expression. CFX allows you to use multiple lines to generate an expression, provided each line is separated by an appropriate operator.

For example, you may have an equation, A + B/C, that consists of three complex terms, A, B, and C. In this case, you could use three lines to simplify creating the expression:

```
A +
B
/ C
```

Note that the operator may be used at the end of a line (A +) or at the beginning of a line (/ C). You do not need to enter the operator twice.

Once the expression has been created, it will appear in the Existing Definitions list box as if it were generated on a single line (A + B/C).

13.2. CEL Operators, Constants, and Expressions

The following topics are discussed:

- *CEL Operators* (p. 144)
- *Conditional if Statement* (p. 145)
- *CEL Constants* (p. 146)
- *Using Expressions* (p. 146)

13.2.1. CEL Operators

CFX provides a range of mathematical, logical and operational operators as built-in functions to help you create complex expressions using the **Expression** details view.

Table 13.1 CEL Operators

Operator	First Operand's Dimensions [x]	Second Operand's Dimensions [y]	Operands' Values (Approx)[a]	Result's Dimensions
-x	Any		Any	[x]
x+y	Any	[x]	Any	[x]
x-y	Any	[x]	Any	[x]
x*y	Any	Any	Any	[x]*[y]
x/y	Any	Any	$y \neq 0$	[x]/[y]
x^y (if y is a simple, constant, integer expression)	Any	Dimensionless	Any[b]	[x]^y
x^y (if y is any simple, constant, expression)	Any	Dimensionless	x > 0	[x]^y
x^y (if y is not simple and constant)	Dimensionless	Dimensionless	x > 0	Dimensionless

Operator	First Operand's Dimensions [x]	Second Operand's Dimensions [y]	Operands' Values (Approx)[a]	Result's Dimensions
!x	Dimensionless		false or true	Dimensionless
x <= y	Any	[x]	false or true	Dimensionless
x < y	Any	[x]	false or true	Dimensionless
x > y	Any	[x]	false or true	Dimensionless
x >= y	Any	[x]	false or true	Dimensionless
x == y	Any	[x]	false or true	Dimensionless
x != y	Any	[x]	false or true	Dimensionless
x && y	Dimensionless	Dimensionless	false or true	Dimensionless
x || y	Dimensionless	Dimensionless	false or true	Dimensionless

[a]The logical constants true and false are represented by "1" and "0" when a numerical representation is required.
[b]For y < 0, x must be non-zero.

13.2.2. Conditional if Statement

CEL supports the conditional if statement using the following syntax:

```
if( cond_expr, true_expr, false_expr )
```

where:

- `cond_expr`: is the logical expression used as the conditional test
- `true_expr`: is the mathematical expression used to determine the result if the conditional test is `true`.
- `false_expr` : is the mathematical expression used to determine the result if the conditional test is `false`.

Note

The expressions `true_expr` and `false_expr` are always evaluated independent of whether the evaluation of `cond_expr` is `true` or `false`. As a consequence, a conditional statement cannot be used to avoid division by zero as in `if( x>0, 1/x, 1.0)`. In this case, when `x=0.0`, a division by zero will still occur because the expression `1/x` is evaluated independent of whether `x>0` is satisfied or not.

13.2.3. CEL Constants

Right-click in the **Expression** details view to access the following useful constants when developing expressions:

Table 13.2 CEL Constants

Constant	Units	Description
R	J K^-1 mol^-1	Universal Gas Constant: 8.314472
avogadro	mol^-1	6.02214199E+23
boltzmann	J K^-1	1.3806503E-23
clight	m s^-1	2.99792458E+08
e	Dimensionless	Constant: 2.7182817
echarge	A s	Constant: 1.60217653E-19
epspermo		1./(clight*clight*mupermo)
g	m s^-2	Acceleration due to gravity: 9.8066502
mupermo	N A^-2	4*pi*1.E-07
pi	Dimensionless	Constant: 3.141592654
planck	J s	6.62606876E-34
stefan	W m^-2 K^-4	5.670400E-08

13.2.4. Using Expressions

The interaction with CEL consists of two phases:

- a definition phase, and,
- a use phase.

The definition phase consists of creating a set of values and expressions of valid syntax. The purpose of the **Expression** details view is to help you to do this.

13.2.4.1. Use of Offset Temperature

When using temperature values in expressions, it is generally safer to use units of [K] only. When units are used that posses an offset (for example, [C]), they are converted internally to [K]. For terms that have temperature to the power of unity, any unit conversion will include the offset between temperature scales. However, in all other cases the offset is ignored because this is usually the most appropriate behavior. You should therefore take care when specifying an expression involving non-unit powers of temperature. For example, each of the expressions below is equivalent:

```
Temperature = 30 [C]
Temperature = 303.15 [K]
Temperature = 0 [C] + 30 [K]
Temperature = 273.15 [K] + 30 [K]
```

These are only equivalent because all units are to the power of unity and units other than [K] appear no more than once in each expression. The following expression will not produce the expected result:

```
Temperature = 0 [C] + 30 [C]
```

This is equivalent to `576.30 [K]` because each value is converted to [K] and then summed. The two expression below are equivalent (as expected) because the offset in scales is ignored for non-unit powers of temperature:

```
Specific Heat = 4200 [J kg^-1 C^-1]
Specific Heat = 4200 [J kg^-1 K^-1]
```

13.3. CEL Examples

The following examples are included in this section:

- *Example: Reynolds Number Dependent Viscosity* (p. 147)
- *Example: Feedback to Control Inlet Temperature* (p. 148)

13.3.1. Example: Reynolds Number Dependent Viscosity

In this example it is assumed that some of the fluid properties, including the dynamic viscosity, are not known. However the Reynolds number, inlet velocity and a length scale are known. The flow is compressible and therefore the density is variable.

Given this information it is possible to calculate the fluid dynamic viscosity based on the Reynolds number. The Reynolds number is given by:

$$Re = \frac{\rho\, U\, L}{\mu}$$

where ρ is density, U a velocity scale, L a length scale and μ the dynamic viscosity. The velocity scale is taken as the inlet velocity, the length scale as the inlet width and the density is calculated as the average density over the inlet area.

The LIBRARY section of the CCL (CFX Command Language) file appears as follows:

```
LIBRARY :
   CEL :
      EXPRESSIONS :
         Re = 4.29E6 [  ]
         Vel = 60 [m s^-1]
         L=1.044[m]
         Visc=areaAve(density)@in*Vel*L/Re
      END
   END
   MATERIAL : Air Ideal Gas
      Option = Pure Substance
      PROPERTIES :
         Option = Ideal Gas
         Molar Mass = 2.896E1 [kg kmol^-1]
         Dynamic Viscosity = Visc
         Specific Heat Capacity = 1.E3 [J kg^-1 K^-1]
         Thermal Conductivity = 2.52E-2 [W m^-1 K^-1]
      END
   END
END
```

This shows that four CEL expressions have been created. The first three expressions define constant values that are used in the `Visc` expression. The `Visc` expression calculates the dynamic viscosity

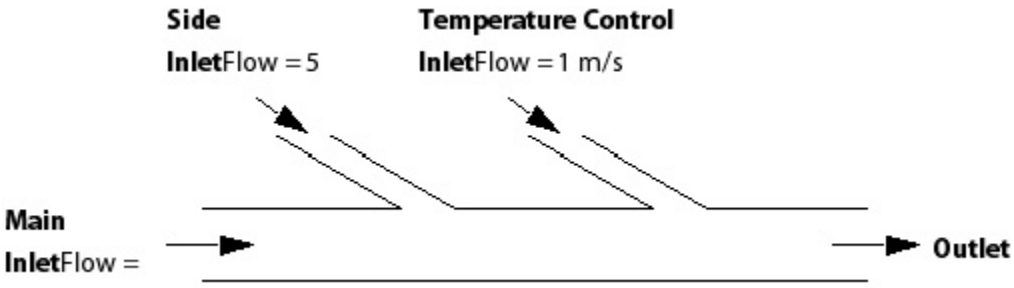
based on the equation for Reynolds number given above. Within the expression the function `areaAve(density)@in` is used to evaluate the average density at the inlet.

The `Visc` expression can now be used to replace the value of `Dynamic Viscosity` in the `MATERIAL` > `PROPERTIES` section.

13.3.2. Example: Feedback to Control Inlet Temperature

In this example a feedback loop is used to control the outlet temperature by varying the temperature at an inlet. To illustrate the example consider the geometry shown below:

Figure 13.1 Temperature Feedback Loop



Fluid from a main and a side inlet enter at temperatures of 275 K and 375 K respectively. The temperature of the fluid entering from the third inlet depends on the outlet temperature. When the outlet temperature is greater than 325 K, the fluid from the third inlet is set to 275 K. When the outlet temperature is less than 325 K, the fluid from the third inlet is set to 375 K. In addition an expression is used to set the dynamic viscosity to be a linear function of temperature.

The LIBRARY section of the `.ccl` (CFX Command Language) file appears as follows. Note that the "\" character indicates a line continuation in CCL.

```
LIBRARY:
   MATERIAL: Water at STP Modified
      Option = Pure Substance
      PROPERTIES:
         Option = General Fluid
         Density = 9.999E2 [kg m^-3]
         Dynamic Viscosity = VisT
         Specific Heat Capacity = 4.21E3 [J kg^-1 K^-1]
         Thermal Conductivity = 5.69E-1 [W m^-1 K^-1]
      END # PROPERTIES
   END # MATERIAL Water at STP Modified
   CEL:
      EXPRESSIONS:
         Tupper = 375.0 [ K ]  # Upper temp.
         Tlower = 275.0 [ K ]  # Lower temp.
         Visupper = 0.000545 [ N s m^-2 ]  # Vis. at Tupper
         Vislower = 0.0018 [ N s m^-2 ]  # Vis. at Tlower
         VisT = Vislower+(Visupper-Vislower)*(T-Tlower)/ \
            (Tupper-Tlower)
         # Vis.-Temp. relationship
         Tm=(Tupper+Tlower)/2
         Tout=areaAve(Water at STP Modified.T)@outlet
         Tcontrol=Tlower*step((Tout-Tm)/1[K]) \
            +Tupper*step((Tm-Tout)/1[K])
      END # EXPRESSIONS
   END # CEL
END # LIBRARY
```

The first four expressions, `Tupper`, `Tlower`, `Visupper` and `Vislower` are simply constant values to define temperature and viscosity values. The expression `VisT` produces a linear function for the dynamic viscosity taking a value of `Visupper` at `Tupper` and a value of `Vislower` at `Tlower`. The expression `Tm` sets the desired value of the outlet temperature. In this case it is set to a mean value of the two inlet temperatures.

`Tout` calculates the outlet temperature using the `areaAve` function.

Finally the expression `Tcontrol` is used to set the temperature of the third inlet. Two step functions are used so that the temperature is equal to `Tlower` when `Tout-Tm` is positive (that is, the outlet temperature is greater than `Tm`), and is equal to `Tupper` when `Tout-Tm` is positive.

13.3.3. Examples: Using Expressions in CFD-Post

The first example is a single-valued expression that calculates the pressure drop through a pipe. The names of inlet and outlet boundaries are "`inlet`" and "`outlet`".

Create a new expression named "`dp`":

```
dp = massFlowAve(Pressure)@inlet – massFlowAve(Pressure)@outlet
```

When you click **Apply**, the value is shown below the editor.

Tip

Alternatively, type the expression in a table cell and prefix with '=' sign. The cell displays the result when you click outside of the cell.

The second example is a variable expression that plots the pressure coefficient variation on a surface or a line:

1. Click the **Expressions** tab, then right-click in the **Expressions** area and select **New**.
2. Create these three expressions:

   ```
   RefPressure = 100000 [Pa]
   dynHead = 0.5 * areaAve(Density)@inlet * areaAve(Velocity)@inlet^2
   cpExp = (Pressure - RefPressure)/dynHead
   ```

3. Click the **Variables** tab, then right-click and select **New**.
4. Create a user variable defined by `cpExp`.
5. Select **Insert** > **Location** > **Line** and use the details view to position the line in the simulation.

   From the details view **Color** tab, plot the user variable on a surface or a line (just as you would with any other variable).

13.4. CEL Technical Details

CEL is a byte code compiled language. Compiled languages, such as Fortran, rely on a translation program to convert them into the native machine language of the host platform. Interpreted languages are of two types: the fully interpreted languages such as the UNIX C shell, and the byte code compiled languages such as CEL. With byte codes, host machines are loaded with a client program (written in a compiled language and compiled for that machine architecture) that interprets the byte stream. The advantage

of the byte code is that they can be the same on all host platforms, obviating the need for platform dependent codes.

Because the byte codes are interpreted, there is no need to re-link executable programs to perform a different calculation. Furthermore, many of the problems encountered by writing and linking in separate routines, for instance in C or Fortran, are averted, and the time taken to set up and debug complicated problems reduced considerably.

The link between CEL and the CFX-Solver is accomplished through an internal program called *genicode*. Genicode generates intermediate code from your CEL definitions and writes to a file that is then interpreted by the CFX-Solver during the solution process.

Chapter 14: Functions in ANSYS CFX

This chapter describes predefined functions in ANSYS CFX:

14.1. CEL Mathematical Functions

The following mathematical functions are available for use with all CEL expressions.

Note

In the **Function** column in the table below, [a] denotes any dimension of the first operand.

Table 14.1 Standard Mathematical CEL Functions

Function	Operand's Values	Result's Dimensions
abs([a])	Any	[a]
acos([ ])	$-1 \le x \le 1$	Radians
asin([ ])	$-1 \le x \le 1$	Radians
atan([ ])[a]	Any	Radians
atan2([a], [a])[a]	Any	Radians
besselJ([ ], [ ])[b]	$0 \le n$	Dimensionless
besselY([ ], [ ])[b]	$0 \le n$	Dimensionless
cos([radians])	Any	Dimensionless
cosh([ ])	Any	Dimensionless
exp([ ])	Any	Dimensionless
int([ ])[c]	Dimensionless	Dimensionless
loge([ ])[d]	$0 < x$	Dimensionless
log10([ ])[e]	$0 < x$	Dimensionless
min([a], [a])	Any	[a]
max([a], [a])	Any	[a]
mod([a], [a])[f]	Any	[a]
nint([ ])[g]	Dimensionless	Dimensionless
sin([radians])	Any	Dimensionless

Function	Operand's Values	Result's Dimensions
sinh([ ])	Any	Dimensionless
sqrt([a])	$0 \leq x$	[a]^0.5
step([ ]) [h]	Any	Dimensionless
tan([radians]) [i]	Any	Dimensionless
tanh([ ])	Any	Dimensionless

[a] atan does not determine the quadrant of the result, but atan2 does.
[b] The value of the first dimensionless operand n, also referred to as the order of the Bessel function, must be an integer (n=0, 1, 2, ...). The second argument is a dimensionless real number.
[c] The int() function truncates the dimensionless argument to its integer part.

Examples:

int(1) = 1

int(2.5) = 2

int(-3.1) = -3

int(-4.8) = -4

The int() function requires a dimensionless argument but will not report an error if the argument of the function has a dimension of radians or degrees.
[d] ln(x) is valid as an alias for loge(x)
[e] log(x) is valid as an alias for log10(x)
[f] mod(x, y) returns the remainder on dividing x by y; the function is not defined for y = 0.
[g] The nint function requires a dimensionless argument and is defined as:

int(x + 0.5) if x >= 0

int(x - 0.5) if x < 0

See the implementation of int( ) function in the table above.

Examples:

nint(2.6) = 3

nint(2.5) = 3

nint(2.4) = 2

nint(1) = 1

nint(-1) = -1

nint(-2.4) = -2

nint(-2.5) = -3

nint(-2.6) = -3

Note that the nint() function will not report an error if the argument of the function has a dimension of radians or degrees.
[h] step(x) is 0 for negative x, 1 for positive x and 0.5 for x=0. x must be dimensionless.
[i] tan(x) is undefined for x=nπ/2, where n=1, 3, 5, ...

14.2. Quantitative CEL Functions in ANSYS CFX

CEL expressions can incorporate specialized functions that are useful in CFD calculations. All CEL functions are described in *Quantitative Function List* (p. 156). For a description of the full CFX Expression Language, see *CFX Expression Language (CEL)* (p. 141).

Important

You must use consistent units when adding, subtracting, or comparing values.

There are some differences between CEL functions in CFX-Pre and CFX-Solver and those in CFD-Post. For details, see below.

The syntax used for calling these functions when used within CEL expressions is:

```
[<Phase_Name>.][<Component_Name>.]<Function>([<Operand>])@<Location>
```

where:

- Terms enclosed in square brackets [ ] are optional and terms in angle brackets < > should be replaced with the required entry.
- <Phase_Name>: specifies a valid name of a phase. The phase can be fluid, particle, solid, fluid pair, or polydispersed fluid. For multi-phase cases in CFX-Pre, if the phase name is not specified in the <Operand>, then the phase name associated with the domain, subdomain, domain boundary, initialization or function in which the operand is being evaluated will be used. For multi-phase cases in CFX-Pre, a discussion of the handling of the phase name when it is not used to qualify (prepended to) <Function> and/or <Operand> can be found in *CEL Functions with Multiphase Flow* (p. 155). For multi-phase cases in CFD-Post, if the phase name is not specified then the bulk quantity (if available for the CFX-Solver Results file) is used.
- <Component_Name>: specifies a valid name of a component material, size group, or reaction
- <Function>: specifies the CEL function to evaluate. See *Quantitative Function List* (p. 156). The function can be further qualified by appending _Coordinate_Direction. In CFX-Pre, if the coordinate frame is not specified (in _Coordinate_Direction) then the function will use the coordinate frame associated with the object (such as for a material, domain, subdomain, domain boundary, source point, monitor point, initialization, reference location or spark ignition object) in which it is being invoked.
- <Coordinate_Direction>: specifies a particular coordinate direction. The syntax of the coordinate direction is [x|y|z][_<Coordinate_Frame>] where the coordinate frame can be the global coordinate frame or any user defined coordinate frame. In CFD-Post, if the coordinate frame is not specified then the global frame is used. See Coordinate Frame Command in the *CFD-Post User's Guide*, for discussion of creating a coordinate frame in CFD-Post.
- <Operand>: specifies the argument of the function (if required). The operand can be either a valid mathematical CEL expression (only in CFD-Post) or specified using the following general variable syntax:

  ```
  [<Phase_Name>.][<Comp_Name>.]<Var_Name>[.<Var_Operator>][.Difference]
  ```

  where <Comp_Name>, <Var_Name>, and <Var_Operator> represent <Component_Name>, <Variable_Name>, and <Variable_Operator>, respectively.

  In CFX-Pre the operand cannot be a CEL expression or any operand qualified by <Variable_Operator>. However, you can create an Additional Variable based on any expression and then use the Additional Variable as the operand. The operand always uses the conservative values unless the Boundcon variable operator is specified (for details, see Data Acquisition Routines in the *CFX-Solver Modeling Guide*). For primitive or composite mesh regions, conservative values will be used even if the Boundcon variable operator is specified.

  The operand must be valid for the physical models being used over the entire location. For example, if the location spans fluid and solid domains, then the operand cannot be Pressure.

For some functions the operand must be left blank as in `area()@Inlet`.

In CFD-Post, difference variables created during case comparison are appended by `.Difference`.

- `<Variable_Name>`: specifies the base name of the variable. You can use the short or long form for variable names. In CFX-Pre the variable name can be further qualified by appending `_<Coordinate_Direction>`. This is useful for specifying a particular component of a vector or tensor, for example `Velocity_y_myLocalFrame`. In CFX-Pre, if the variable name corresponds to that of a component of a vector or a tensor and coordinate frame is not prescribed (as part of the coordinate direction) then the global coordinate frame is used. An exception applies for the position vector `x`, `y`, `z` (or `r`, `theta`, `z`) components, which are always local, see *Functions Involving Coordinates* (p. 155).
- `<Variable_Operator>` specifies the name of the variable operator. The variable operators are.

Long Name	Short Name
Magnitude	magnitude
Gradient	grad
Curl	curl
Laplacian	laplacian
Time Derivative	dt
Transient Minimum	Trnmin
Transient Maximum	Trnmax
Transient Std Deviation	Trnsdv
Transient RMS	Trnrms
Transient Average	Trnavg
Boundcon	hybrid

All but the `<Derived>` operator are available in CFX-Pre and the CFX-Solver. See Data Acquisition Routines in the *CFX-Solver Modeling Guide*. The `<Derived>` variable operator is available only in CFD-Post, for example `Absolute Helicity` derived for use with Vortex Cores, see Vortex Core Region in the *CFD-Post User's Guide*. In CFX-Pre the variable operator can be further qualified by appending `_<Coordinate_Direction>`.

The `Magnitude` and `Curl` operators may only be applied to vector variables. All other operators may be applied to scalar variables and vector components.

- `<Location>`: specifies the location over which the function is to be applied. The syntax of location is:

```
[Case:<Case_Name>.][REGION:]<Location_Name>
```

The case syntax `[Case:<Case_Name>.]` is only available in CFD-Post and is used when multiple cases are loaded to indicate the name of the desired case.

In CFX-Pre `[<Location_Name>]` must be a domain boundary, domain, subdomain, or, primitive or composite mesh region. If the location name of a mesh region is the same as the name of a named boundary, domain or subdomain, then the mesh location name must be prepended by `REGION:`. For primitive or composite mesh regions, conservative values will be used even if the name of the mesh region is the same as that of a named boundary.

In CFD-Post `[<Location_Name>]` can be any loaded or user-defined location (for example, a point, domain boundary, plane, mesh region etc.). The syntax `REGION:<Region Name>` can also be used in CFD-Post to refer to any mesh region. If a mesh region is present with the same name

as, for example, a domain boundary, then the mesh region is imported into CFD-Post with a `Region` suffix. For example, if there is both a domain boundary and a mesh region called `in1` in the CFX-Solver Results file, then in CFD-Post the mesh region will appear in CFD-Post as `in1 Region`. The syntax `in1` will refer to the domain boundary, and either of `in1 Region` or `REGION:in1` can be used to refer to the mesh region as desired.

Note

You cannot use a composite region that consists of a mixture of 2D and 3D regions.

Table 14.2 Examples of the Calling Syntax for an Expression

`areaAve(p)@Inlet`	This results in the area-weighted average of pressure on the boundary named `Inlet`.
`area()@REGION:myCompositeMeshRegion`	This results in the area of a 2D mesh region named `myCompositeMeshRegion`.
`areaAve(Pressure - 10000 [Pa])@outlet`	This syntax is appropriate only for CFD-Post.
`area_x()@inlet`	
`Water at RTP.force_z()@Default`	

14.3. Functions Involving Coordinates

The CEL variables `x`, `y`, `z`, `r` and `theta`, representing the local coordinates, cannot be used as the variable. However, the variables `xGlobal`, `yGlobal` and `zGlobal` can be used. For example, the following is a valid expression definition:

```
z*areaAve(xGlobal)@inlet
```

14.4. CEL Functions with Multiphase Flow

Note

These functions are available in CFX-Pre and CFX-Solver without restrictions, and in CFD-Post with the restriction that you cannot use short names.

If the function is fluid-specific, various behaviors are possible depending on the function type:

- For `massFlow` and `massFlowAve`, if the phase name is not specified for the function, then the bulk mass flows will be used. See cases 1 to 7 in the table below.
- For other fluid-specific functions:
  - if a fluid-specific operand is specified and no fluid is specified for the function, then the fluid specified for the operand will be assumed for the function as well. See case 8 in the table below.
  - if the function is specified and no fluid is specified for the operand, then the fluid specified for the function will be assumed for the operand as well. See cases 7 and 9 in the table below.

- If both the function or operand are fluid-specific, and a phase name is not given for either, the solver will stop with an error. See case 10 in the table below.

Table 14.3 CEL Multiphase Examples

Case	CEL Function - Multiphase	Behavior
1	`massFlow()@inlet`	Bulk mass flow rate through inlet
2	`Air.massFlow()@inlet`	Air mass flow rate through inlet
3	`massFlowAve(Pressure)@inlet`	Bulk mass flow averaged pressure on inlet
4	`Air.massFlowAve(Pressure)@inlet`	Air mass flow averaged pressure on inlet
5	`massFlowAve(Air.Volume Fraction)@inlet`	Bulk mass flow averaged air volume fraction on inlet
6	`Air.massFlowAve(Air.Volume Fraction)@inlet`	Air mass flow averaged air volume fraction on inlet
7	`Air.massFlowAve(Volume Fraction)@inlet`	Same as `Air.massFlowAve(Air.Volume Fraction)@ inlet`
8	`massInt(Air.Volume Fraction)@domain1`	Same as `Air.massInt(Air.Volume Fraction)@ domain1`
9	`Air.massInt(Volume Fraction)@domain1`	Same as `Air.massInt(Air.Volume Fraction)@ domain1`
10	`massFlowAve(Volume Fraction)@inlet`	Error because no fluid specified

14.5. Quantitative Function List

The available quantitative functions are outlined in the sections that follow.

In the table that follows, <Expression> in CFD-Post means any expression; however, in CFX-Pre and CFX-Solver <Expression> means "Additional Variable Expression".

The behavior of the functions in the table below depends on the type of <Location>. Typically:

- on a domain the functions use vertex values for the operand,
- on a subdomain the functions use element values for the operand,
- on a boundary the functions use conservative values for the operand unless this is overriden by the `Boundcon` variable operator in CFX-Pre,
- on user locations in CFD-Post the functions use values interpolated from nodal values.

Table 14.4 CEL Functions in CFX-Pre/CFX-Solver and in CFD-Post

Function Name and Syntax <required> [<optional>]	Operation	Availability
area( )	Area of a boundary or interface. Supports @<Location> See *area* (p. 161).	All

Function Name and Syntax <required> [<optional>]	Operation	Availability
area_x[_<Coord Frame>]( ) area_y[_<Coord Frame>]( ) area_z[_<Coord Frame>]( )	The (signed) component of the normal area vector in the local x, y or z direction. The normal area vectors are always directed out of the domain, therefore you may obtain positive or negative areas depending on the orientation of your domain and the boundary you are operating on. The area of a closed surface will always be zero. Supports @<Location>	All[a]
areaAve(<Expression>)	Area-weighted average of <Expression> on a boundary. Supports @<Location> See *areaAve* (p. 162).	All
areaAve_x[_<Coord Frame>]( ) areaAve_y[_<Coord Frame>]( ) areaAve_z[_<Coord Frame>]( )	The (signed) component of the normal area vector *weighted average* in the local x, y or z direction. The normal area vectors are always directed out of the domain, therefore you may obtain positive or negative areas depending on the orientation of your domain and the boundary you are operating on. The area of a closed surface will always be zero. Supports @<Location>	CFD-Post
areaInt(<Expression>)	Area-weighted integral of <Expression> on a boundary. The `areaInt` function projects the location onto a plane normal to the specified direction (if the direction is not set to `None`) and then performs the calculation on the projected location (the direction specification can also be `None`). The direction of the normal vectors for the location is important and will cancel out for surfaces such as closed surfaces. Supports @<Location> See *areaInt* (p. 163).	All
areaInt_x[_<Coord Frame>]( ) areaInt_y[_<Coord Frame>]( ) areaInt_z[_<Coord Frame>]( )	The (signed) component of the normal area vector *weighted integral* in the local x, y or z direction. The normal area vectors are always directed out of the domain, therefore you may obtain positive or negative areas depending on the orientation of your domain and the boundary you are operating on. The area of a closed surface will always be zero.	All

Function Name and Syntax <required> [<optional>]	Operation	Availability
	Supports @<Location>	
ave(<Expression>)	Arithmetic average of <Expression> over nodes within a domain or subdomain. Supports @<Location> See *ave* (p. 164).	All
count( )	Counts the number of evaluation points (nodes) on the named region. See *count* (p. 164).	All
countTrue(<Expression>)	Counts the number of nodes at which the logical expression evaluates to true. Supports @<Location> See *countTrue* (p. 165).	All
force( )	The magnitude of the force vector on a boundary. Supports [<Phase>.], @<Location> See *force* (p. 165).	All
forceNorm [_<Axis>[_<Coord Frame>]]( )	The length of the normalized force on a curve in the specified direction. Supports [<Phase>.], @<Location> See *forceNorm* (p. 167).	CFD-Post
force_x[_<Coord Frame>]( ) force_y[_<Coord Frame>]( ) force_z[_<Coord Frame>]( )	The (signed) component of the force vector in the local x, y or z direction. Supports [<Phase>.], @<Location>	All[a]
inside()	Similar to the `subdomain` variable, but allows a specific 2D or 3D location to be given. Supports @<Location> See *inside* (p. 167).	CFX-Pre, CFX-Solver
length()	Length of a curve. Supports @<Location> See *length* (p. 168).	CFD-Post
lengthAve(<Expression>)	Length-weighted average. Supports @<Location> See *lengthAve* (p. 168).	CFD-Post

Function Name and Syntax <required> [<optional>]	Operation	Availability
lengthInt(<Expression>)	Length-weighted integration. Supports @<Location> See *lengthInt* (p. 169).	CFD-Post
mass()	The total mass within a domain or subdomain. This is fluid-dependent. Supports @<Location> See *mass* (p. 169).	CFX-Pre, CFX-Solver
massAve(<Expression>)	Mass-weighted average of <Expression> on a domain or subdomain. Supports @<Location> See *massAve* (p. 169).	CFX-Pre, CFX-Solver
massFlow()	Mass flow through a boundary. Supports [<Phase>.], @<Location> See *massFlow* (p. 169).	All
massFlowAve(<var>)	Mass flow weighted average of <Expression> on a boundary. Supports [<Phase>.], @<Location> See *massFlowAve* (p. 171).	All
massFlowAveAbs(<var>)	Absolute mass flow weighted average of <Expression> on a boundary. Supports [<Phase>.], @<Location> See *massFlowAveAbs* (p. 171).	All
massFlowInt(<var>)	Mass flow weighted integration of <Expression> on a boundary. Supports [<Phase>.], @<Location> See *massFlowInt* (p. 173).	All
massInt(<Expression>)	The mass-weighted integration of <Expression> within a domain or subdomain. Supports @<Location> See *massInt* (p. 174).	CFX-Pre, CFX-Solver
maxVal(<Expression>)	Maximum Value of <Expression> within a domain or subdomain. Supports @<Location>	All

Function Name and Syntax <required> [<optional>]	Operation	Availability
	See *maxVal* (p. 174).	
minVal(<Expression>)	Minimum Value of <Expression> within a domain or subdomain. Supports @<Location> See *minVal* (p. 174).	All
probe(<Expression>)	Returns the value of the specified variable on the specified Point locator. Supports @<Location> See *probe* (p. 175).	All
rbstate(<rbvar>[<axis>])	Returns the position/velocity/acceleration or orientation/angular velocity/angular acceleration (or axis components of these) of a rigid body object or immersed solid that is governed by a rigid body solution. These quantities are with respect to the global coordinate frame. See *rbstate* (p. 175).	CFX-Pre, CFX-Solver
rmsAve(<Expression>)	RMS average of <Expression> within a 2D domain. Supports @<Location> See *rmsAve* (p. 176).	CFX-Pre, CFX-Solver
sum(<Expression>)	Sum of <Expression> over all domain or subdomain vertices. Supports @<Location> See *sum* (p. 176).	All
torque( )	Magnitude of the torque vector on a boundary. Supports [<Phase>.], @<Location> See *torque* (p. 177).	All
torque_x[_<Coord Frame>]() torque_y[_<Coord Frame>]() torque_z[_<Coord Frame>]()	The (signed) components of the torque vector about the local x, y, or z coordinate axis. Supports [<Phase>.], @<Location>	All[a]
volume( )	The total volume of a domain or subdomain. Supports @<Location> See *volume* (p. 177).	All

Function Name and Syntax <required> [<optional>]	Operation	Availability
volumeAve(<Expression>)	Volume-weighted average of <var> on a domain. Supports @<Location> See *volumeAve* (p. 178).	All
volumeInt(<Expression>)	Volume-weighted integration of <var> within a domain or subdomain. Supports @<Location> See *volumeInt* (p. 178).	All

[a]See the definition for [_<Coordinate_ Direction>]] in *Quantitative CEL Functions in ANSYS CFX* (p. 152)

14.5.1. area

The `area` function is used to calculate the area of a 2D locator.

```
area[_<Axis>[_<Coord Frame>] ]()@<Location>
```

where:

- `<Axis>` is x, y, or z
- `<Coord Frame>` is the coordinate frame
- `<Location>` is any 2D region (such as a boundary or interface).

An error is raised if the location specified is not a 2D object. If an axis is not specified, the total area of the location is calculated.

`area()@Isosurface1` calculates the total area of the location, and `Isosurface1.area_y()@Isosurface1` calculates the projected area of `Isosurface1` onto a plane normal to the Y-axis.

14.5.1.1. Tools > Command Editor Example

```
>calculate area, <Location>, [<Axis>]
```

The specification of an axis is optional. If an axis is not specified, the value held in the object will be used. To calculate the total area of the location, the axis specification should be left blank (that is, type a comma after the location specification).

`>calculate area, myplane` calculates the area of the locator `myplane` projected onto a plane normal to the axis specification in the CALCULATOR object.

`>calculate area, myplane,` calculates the area of the locator `myplane`. Note that adding the comma after `myplane` removes the axis specification.

14.5.1.2. Tools > Function Calculator Example

The following example will calculate the total area of the locator `Plane1`:

Function: area, **Location:** Plane1.

14.5.2. areaAve

The areaAve function calculates the area-weighted average of an expression on a 2D location. The area-weighted average of a variable is the average value of the variable on a location when the mesh element sizes are taken into account. Without the area weighting function, the average of all the nodal variable values would be biased towards variable values in regions of high mesh density.

```
areaAve[_<Axis>[_<Coord Frame>] ](<Expression>)@<Location>
```

where:

- <Axis> is x, y, or z
- <Coord Frame> is available in CFD-Post only
- <Expression> is an expression
- <Location> is any 2D region (such as a boundary or interface). An error is raised if the location specified is not a 2D object.

To calculate the pressure coefficient C_p, use:

```
(Pressure - 1[bar])/(0.5*Density*(areaAve(Velocity)@inlet)^2)
```

You can create an expression using this, and then create a user variable using the expression. The user variable can then be plotted on objects like any other variable.

Note

Projected areaAve (for example, areaAve_x) works as expected only for surfaces that do not fold in the selected direction. In extreme case, if the surface is fully closed, the projected average will result in a randomly large number, as the projected area will be zero.

14.5.2.1. Tools > Command Editor Example

```
>calculate areaAve, <Expression>, <Location>, <Axis>
```

14.5.2.2. Tools > Function Calculator Examples

- This example will calculate the average magnitude of Velocity on outlet.

  Function: areaAve, **Location:** outlet, **Variable:** Velocity.

  Note that flow direction is not considered because the magnitude of a vector quantity at each node is calculated.

- You can use the scalar components of Velocity (such as Velocity u) to include a directional sign. This example will calculate the area-weighted average value of Velocity u, with negative values of Velocity u replaced by zero. Note that this is not the average positive value because zero values will contribute to the average.

  Function: areaAve, **Location:** outlet, **Variable:** max(Velocity u, 0.0[m s^-1]).

14.5.3. arealnt

The `areaInt` function integrates a variable over the specified 2D location. To perform the integration over the total face area, select the `None` option from the **Axis** drop-down menu. If a direction is selected, the result is an integration over the projected area of each face onto a plane normal to that direction. Each point on a location has an associated area which is stored as a vector and therefore has direction. By selecting a direction in the function calculator, you are using only a single component of the vector in the area-weighting function. Because these components can be positive or negative, depending on the direction of the normal on the location, it is possible for areas to cancel out. An example of this would be on a closed surface where the projected area will always be zero (the results returned will not in general be exactly zero because the variable values differ over the closed surface). On a flat surface, the normal vectors always point in the same direction and never cancel out.

```
areaInt[_<Axis>[_<Coord Frame>] ](<Expression>)@<Location>
```

where:

- `<Axis>` is x, y, or z.

  Axis is optional; if not specified the integration is performed over the total face area. If axis is specified, then the integration is performed over the projected face area. A function description is available.
- `<Coord Frame>` is the coordinate frame.
- `<Location>` is any 2D region (such as a boundary or interface). An error is raised if the location specified is not a 2D object.

`areaInt_y_Frame2(Pressure)@boundary1` calculates the pressure force acting in the y-direction of the coordinate frame `Frame2` on the locator `boundary1`. This differs from a calculation using the force function, which calculates the total force on a wall boundary (that is, viscous forces on the boundary are included).

14.5.3.1. Tools > Command Editor Example

```
>calculate areaInt, <Expression>, <Location>, [<Axis>]
```

Axis is optional. If it is not specified, the value held in the object will be used. To perform the integration over the total face area, the axis specification should be blank (that is, type a comma after the location name). A function description is available in *arealnt* (p. 163).

14.5.3.2. Tools > Function Calculator Examples

- This example integrates `Pressure` over `Plane 1`. The returned result is the total pressure force acting on `Plane 1`. The magnitude of each area vector is used and so the direction of the vectors is not considered.

  Function: `areaInt`, **Location:** `Plane 1`, **Variable:** `Pressure`, **Direction:** `None`
- This example integrates `Pressure` over the projected area of `Plane 1` onto a plane normal to the X-axis. The result is the pressure force acting in the X-direction on `Plane 1`. This differs slightly from using the force function to calculate the X-directional force on `Plane 1`. The force function includes forces due to the advection of momentum when calculating the force on an internal arbitrary plane or a non-wall boundary (inlets, etc.).

  Function: arealnt, **Location:** `Plane 1`, **Variable:** `Pressure`, **Direction:** `Global X`.

14.5.4. ave

The ave function calculates the arithmetic average (the mean value) of a variable or expression on the specified location. This is simply the sum of the values at each node on the location divided by the number of nodes. Results will be biased towards areas of high nodal density on the location. To obtain a mesh independent result, you should use the lengthAve, areaAve, volumeAve or massFlowAve functions.

```
ave(<var|Expression>)@<Location>
```

where:

- <var|Expression> is a variable or a logical expression
- <Location> is any 3D region (such as a domain or subdomain).

The ave function can be used on point, 1D, 2D, and 3D locations.

ave(Yplus)@Default calculates the mean Yplus values from each node on the default walls.

14.5.4.1. Tools > Command Editor Example

```
>calculate ave, <var|Expression>, <Location>
```

Note

To obtain a mesh-independent result, you should use the lengthAve, areaAve, volumeAve or massFlowAve functions.

The average of a vector value is calculated as an average of its magnitudes, not the magnitude of component averages. As an example, for velocity:

$$|v|_{\text{ave}} = \frac{|v_1| + |v_2|}{2} \qquad \textbf{(14–1)}$$

where

$$|v_i| = \sqrt{\left(v_{x,i}^2 + v_{y,i}^2 + v_{z,i}^2\right)} \qquad \textbf{(14–2)}$$

14.5.4.2. Tools > Function Calculator Example

This example calculates the mean temperature at all nodes in the selected domain.

Function: ave, **Location:** MainDomain, **Variable:** Temperature.

14.5.5. count

The count function returns the number of nodes on the specified location.

```
count()@<Location>
```

where:

- <Location> is valid for point, 1D, 2D, and 3D locations.

count()@Polyline1 returns the number of points on the specified polyline locator.

14.5.5.1. Tools > Command Editor Example

```
>calculate count, <Location>
```

14.5.5.2. Tools > Function Calculator Example

This example returns the number of nodes in the specified domain.

Function: count, **Location:** MainDomain.

14.5.6. countTrue

The countTrue function returns the number of mesh nodes on the specified region that evaluate to "true", where true means greater than or equal to 0.5. The countTrue function is valid for 1D, 2D, and 3D locations.

```
countTrue(<Expression>)@<Location>
```

where <Expression> is:

- In CFD-Post, an expression that contains the logical operators =, >, <, <=, or >=.
- In CFX-Solver, an Additional Variable that you define. For example:

```
TemperatureLE = Temperature > 300[K]
```

countTrue(TemperatureLE)@Polyline1 returns the number of nodes on the specified polyline locator that evaluate to true.

14.5.6.1. Tools > Command Editor Examples

In CFD-Post:

```
>calculate countTrue(Temperature > 300[K]), Domain1
```

In CFX-Solver:

```
>calculate countTrue(TemperatureLE), Domain1
```

14.5.6.2. Tools > Function Calculator Example

This example returns the number of nodes that evaluate to "true" in the specified domain.

Function: countTrue, **Location:** MainDomain, **Expression:** Temperature > 300[K].

14.5.7. force

This function returns the force exerted by the fluid on the specified 2D locator in the specified direction.

```
[<Phase>.]force[_<Axis>[_<Coord Frame>] ]()@<Location>
```

where:

- `[<Phase>.]` is an optional prefix that is not required for single-phase flows. For details, see *CEL Functions with Multiphase Flow* (p. 155).
- `<Axis>` is x, y, or z
- `<Coord Frame>` is the coordinate frame
- `<Location>` is any 2D region (such as a boundary or interface).

Force calculations on boundaries require additional momentum flow data.

`Water at RTP.force_x()@wall1` returns the total force in the x-direction acting on `wall1` due to the fluid `Water at RTP`.

The force on a boundary is calculated using momentum flow data from the results file, if it is available. The result can be positive or negative, indicating the direction of the force. For non-boundary locators, an approximate force is always calculated.

CFD-Post calculates the approximate force as follows:

- If the locator is a wall boundary, the force is equal to the pressure force.
- For all other locators, the force is equal to the pressure force plus the mass flow force (due to the advection of momentum).
- In all cases, if wall shear data exists in the results file, the viscous force is added to the calculated force.

The `force` function enables you to select the fluids to use when performing your calculation. The result returned is the force on the locator due to that fluid/those fluids. Because the pressure force is the same at each node irrespective of the choice of fluids, the only difference is in the viscous forces (on wall boundaries) or the mass flow forces.

It is important to note that forces arising as a result of the reference pressure are not included in the force calculation. You can include reference pressure effects in the force calculation in the CFX-Solver by setting the expert parameter `include pref in forces = t`

When performing transient simulations in rotating domains, forces may be reported in one of two reference frames. The first is a reference frame attached to the rotating domain. The second is a reference frame that does not rotate with the domain, having an orientation fixed by the initial orientation. The forces reported by the CFX Solver are always given with respect to the second choice. The forces reported by CFD-Post may be given in either reference frame, depending on the context:

- If the forces are calculated from boundary flows available in the Results file (see Output Boundary Flows Check Box in the *CFX-Pre User's Guide*), then the forces are calculated with respect to the second choice
- If the forces are not calculated from boundary flows available in the Results file, they are calculated by CFD-Post with respect to the first choice if the domain is rotated by CFD-Post (according to the Angular Shift for Transient Rotating Domains option), and with respect to the second choice otherwise.

14.5.7.1. Tools > Command Editor Example

```
>calculate force, <Location>, <Axis>, [<Phase>]
```

14.5.7.2. Tools > Function Calculator Examples

- This calculates the total force on the default wall boundaries in the x-direction. Pressure and viscous forces are included.

  Function: `force`, **Location:** `Default`, **Direction:** `Global X`, **Phase:** `All Fluids`.

- This calculates the forces on inlet1 due to pressure and the advection of momentum.

  Function: force, **Location:** inlet1, **Direction:** `Global X`, **Phase:** `Water at RTP`.

14.5.8. forceNorm

Returns the per unit width force on a line in the direction of the specified axis. It is available only for a polyline created by intersecting a locator on a boundary. Momentum data must also be available. The magnitude of the value returned can be thought of as the force in the specified direction on a polyline, if the polyline were 2D with a width of one unit.

```
[<Phase>.]forceNorm[_<Axis>[_<Coord Frame>] ]()@<Location>
```

where:

- `[<Phase>.]` is an optional prefix that is not required for single-phase flows. For details, see *CEL Functions with Multiphase Flow* (p. 155).
- `<Axis>` is x, y, or z
- `<Coord Frame>` is available in CFD-Post only
- `<Location>` is any 1D location. An error will be raised if the location specified is not one-dimensional.

`forceNorm_y()@Polyline1` calculates the per unit width force in the y-direction on the selected polyline.

14.5.8.1. Tools > Command Editor Example

```
>calculate forceNorm, <Location>, <Axis>, [<Phase>]
```

14.5.8.2. Tools > Function Calculator Example

The result from this calculation is force per unit width on `Polyline1` in the x-direction.

Function: `forceNorm`, **Location:** `Polyline1`, **Direction:** `Global X`, **Phase:** `All Fluids`.

14.5.9. inside

The `inside` CEL function is essentially a step function variable, defined to be unity within a subdomain and zero elsewhere. This is useful for describing different initial values or fluid properties in different regions of the domain. It is similar to the CEL subdomain variable, but allows a specific 2D or 3D location to be given. For example, `273 [K] * inside()@Subdomain 1` has a value of 273 [K] at points in Subdomain 1 and 0 [K] elsewhere. The location does not need to be a subdomain, but can be any 2D or 3D named sub-region of the physical location on which the expression is evaluated. For immersed solids simulations, the location can also be a specific immersed solid domain, and the inside function will be updated automatically at the beginning of each time step.

```
inside()@<Location>
```

where:

- `<Location>` is any 2D or 3D named sub-region of the physical location on which the expression is evaluated.
- `<Location>` can also be an immersed solid domain on which the expression is evaluated dynamically.

Note

The `inside` CEL function is not available in CFD-Post.

14.5.9.1. Tools > Command Editor Example

```
>calculate inside, <Location>
```

14.5.10. length

Computes the length of the specified line as the sum of the distances between the points making up the line.

```
length()@<Location>
```

where:

- `<Location>` is any 1D location. Specifying a 2D location will not produce an error; the sum of the edge lengths from the elements in the locator will be returned.

`length()@Polyline1` returns the length of the polyline.

14.5.10.1. Tools > Command Editor Example

```
>calculate length, <Location>
```

Note

While using this function in Power Syntax, the leading character is capitalized to avoid confusion with the Perl internal command "length".

14.5.10.2. Tools > Function Calculator Example

This example calculates the length of a polyline.

Function: `length`, **Location:** `Polyline1`.

14.5.11. lengthAve

Computes the length-based average of the variable on the specified line. This is the 1D equivalent of the `areaAve` function. The results is independent of the nodal distribution along the line because a weighting function assigns a higher weighting to areas of sparse nodal density.

```
lengthAve(<Expression>)@<Location>
```

where:

- `<Expression>` is an expression
- `<Location>` is any 1D or 2D location.

`lengthAve(T)@Polyline1` calculates the average temperature on `Polyline1` weighted by the distance between each point (T is the system variable for temperature).

14.5.11.1. Tools > Command Editor Example

```
>calculate lengthAve, <Expression>, <Location>
```

14.5.11.2. Tools > Function Calculator Example

This calculates the average velocity on the location `Polyline1` using a length-based weighting function to account for the distribution of points along the line.

Function: `lengthAve`, **Location:** `Polyline1`, **Variable:** `Velocity`.

14.5.12. lengthInt

Computes the length-based integral of the variable on the specified line. This is the 1D equivalent of the areaInt function.

```
lengthInt(<Expression>)@<Location>
```

where:

- `<Expression>` is an expression
- `<Location>` is any 1D location.

14.5.12.1. Tools > Command Editor Example

```
>calculate lengthInt, <Expression>, <Location>.
```

14.5.13. mass

```
mass()@<Location>
```

where:

- `<Location>` is any 3D region (such as a domain or subdomain).

14.5.13.1. Tools > Command Editor Example

```
>calculate mass, <Location>.
```

14.5.14. massAve

```
massAve(<var>)@<Location>
```

where:

- `<var>` is a variable
- `<Location>` is any 3D region (such as a domain or subdomain).

14.5.14.1. Tools > Command Editor Example

```
>calculate massAve, <var>, <Location>.
```

14.5.15. massFlow

Computes the mass flow through the specified 2D location.

```
[<Phase>.]massFlow()@<Location>
```

where:

- `[<Phase>.]` is an optional prefix that is not required for single-phase flows. For details, see *CEL Functions with Multiphase Flow* (p. 155).
- `<Location>` is any fluid surfaces (such as Inlets, Outlets, Openings and fluid-fluid interfaces).

`Air at STP.massFlow()@DegassingOutlet` calculates the mass flow of `Air at STP` through the selected location.

For boundary locators:

- The mass flow is calculated using mass flow data from the results file, if it is available[1]. Otherwise, an approximate mass flow is calculated.
- For multiphase cases, the mass flow through a boundary on a GGI interface evaluated in CFD-Post is an approximation to the 'exact' mass flow evaluated by the solver. This approximation vanishes as the mesh is refined or as the volume fraction on the interface becomes uniform.

For non-boundary locators (that is, internal locators):

- If the locator is an edge based locator (such as a slice plane or isosurface), the domain mass flow data from the results file will be used.
- In all other cases, an approximate mass flow is calculated.

The `massFlow` function enables you to select the fluids to use when performing your calculation. The result returned is the mass flow of the selected fluids through the locator.

If a user specifies a boundary mass source, this mass source is not included in the massFlow calculator. This will affect all massFlow related calculations, such as massFlowInt().

14.5.15.1. Mass Flow Sign Convention

The mass flow through a surface is defined by $-\rho\mathbf{V}\cdot\mathbf{n}$ where $\mathbf{V}$ is the velocity vector and $\mathbf{n}$ is the surface normal vector. By convention, the surface normal at a domain boundary is directed out of the domain. Therefore, the mass flow is positive at an inlet boundary with the velocity directed into the domain. For planes and surfaces that cut through a domain, the normal of the plane or surface is determined by from the right-hand rule and the manner in which the plane or surface is constructed. For example, the surface normal for a Z-X plane has the same sense and direction as the Y-axis.

14.5.15.2. Tools > Command Editor Example

```
>calculate massFlow, <Location>, [<Phase>]
```

14.5.15.3. Tools > Function Calculator Example

This calculates the mass flow for all fluids in the domains through the location outlet2:

Function: `massFlow`, **Location:** `outlet2`, **Phase:** `All Fluids`.

[1]The availability depends on the setting for **Output Boundary Flows** for the file (see Output Boundary Flows Check Box in the *CFX-Pre User's Guide* for details).

14.5.16. massFlowAve

Computes the average of a variable/expression on the specified 2D location. The `massFlowAve` function allows you to select the fluids to use when performing your calculation. The result returned is the average variable value, evaluated according to the formula:

$$\text{massFlowAve}\ (\Phi) = \frac{\Sigma\ (m\ \Phi)}{\Sigma\ m} \qquad \textbf{(14–3)}$$

where Φ represents the variable/expression being averaged and m represents the local mass flow (net local mass flow if more than one fluid is selected). Each summation term is evaluated on, and corresponds to, a node on the 2D locator. The mass flow for each term is derived from summing contributions from the surrounding solver integration points. As a result, the denominator evaluates to the conservative net mass flow through the 2D locator.

In cases where there is significant flow, but little or no net flow through the 2D locator (as can happen with recirculation), the denominator of the averaging formula becomes small, and the resulting average value may become adversely affected. In such cases, the `massFlowAveAbs` (see *massFlowAveAbs* (p. 171)) function is a viable alternative to the `massFlowAve` function.

```
[<Phase>.]massFlowAve(<var|Expression>)@<Location>
```

where:

- `[<Phase>.]` is an optional prefix that is not required for single-phase flows. For details, see *CEL Functions with Multiphase Flow* (p. 155).
- `<var|Expression>` is a variable or expression
- `<Location>` is any fluid surfaces (such as Inlets, Outlets, Openings and fluid-fluid interfaces). An error is raised if the location specified is not 2D.

`massFlowAve(Density)@Plane1` calculates the average density on `Plane1` weighted by the mass flow at each point on the location.

See the *Advanced Mass Flow Considerations* (p. 172) and *Mass Flow Technical Note* (p. 172) sections under *massFlowAveAbs* (p. 171) for more information.

14.5.16.1. Tools > Command Editor Example

```
>calculate massFlowAve, <var|Expression>, <Location>, [<Phase>]
```

14.5.16.2. Tools > Function Calculator Example

This example calculates the average velocity on `Plane1` weighted by the mass flow for all fluids assigned to each point on `Plane1`:

Function: `massFlowAve`, **Location:** `Plane1`, **Variable:** `Velocity`, **Phase:** `All Fluids`

14.5.17. massFlowAveAbs

This function is similar to the `massFlowAve` function (see *massFlowAve* (p. 171)), except that each local mass flow value used in the averaging formula has the absolute function applied. That is:

$$\text{massFlowAveAbs}\ (\Phi) = \frac{\Sigma\ (|\ m\ |\ \Phi)}{\Sigma\ |\ m\ |} \quad \textbf{(14–4)}$$

```
[<Phase>.]massFlowAveAbs(<var|Expression>)@<Location>
```

where:

- `[<Phase>.]` is an optional prefix that is not required for single-phase flows. For details, see *CEL Functions with Multiphase Flow* (p. 155).
- `<var|Expression>` is a variable or expression
- `<Location>` is any fluid surfaces (such as Inlets, Outlets, Openings and fluid-fluid interfaces). An error is raised if the location specified is not 2D.

`massFlowAve(Density)@Plane1` calculates the average density on `Plane1` weighted by the mass flow at each point on the location.

In cases where there is significant flow, but little or no net flow through the 2D locator (as can happen with recirculation), the `massFlowAveAbs` function is a viable alternative to the `massFlowAve` function (see *massFlowAve* (p. 171)).

14.5.18. Advanced Mass Flow Considerations

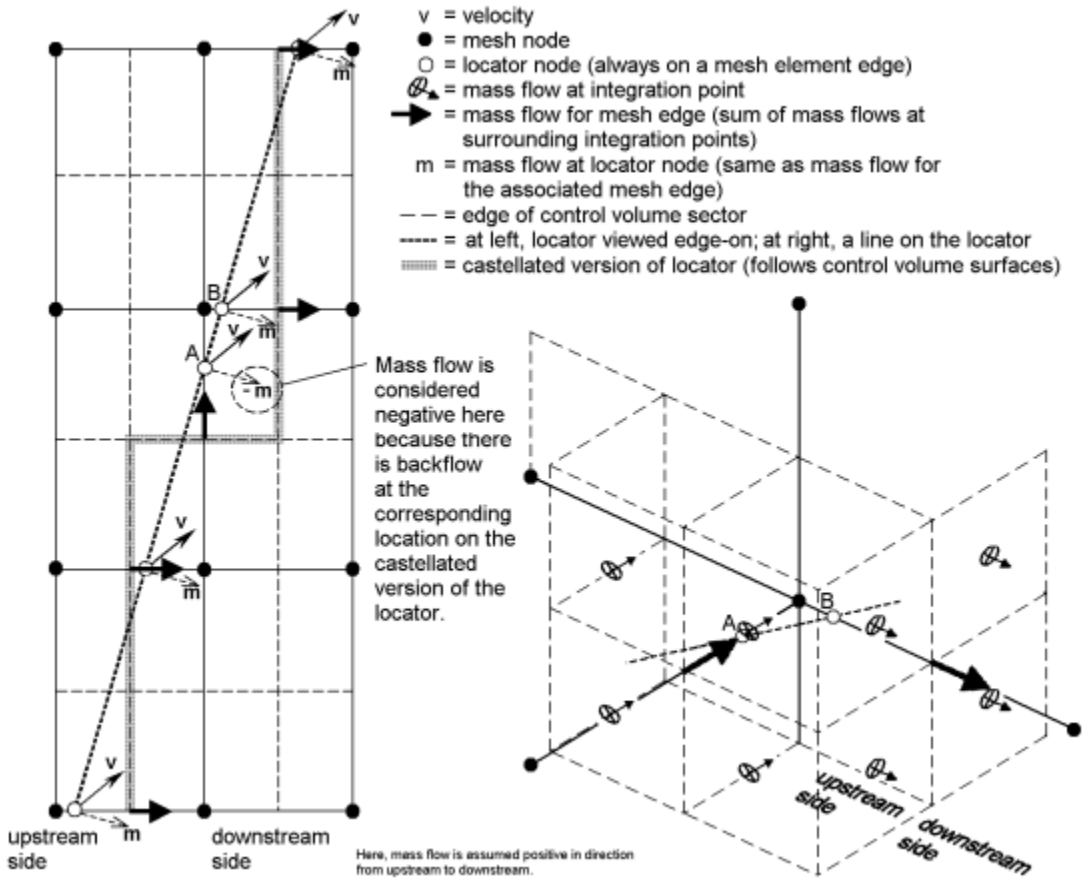
Note that the `massFlowAveAbs` and `massFlowAve` functions provide the same result, and that the denominator evaluates to the net mass flow through the 2D locator, only when all of the flow passes through the 2D locator in the same general direction (in other words, when there is no backflow). If there is any backflow through the 2D locator, the denominator in the function for `massFlowAveAbs` evaluates to a value of greater magnitude than the conservative net mass flow through the 2D locator (although this is not necessarily harmful to the resulting average value).

The values of variables other than mass flow are stored at the mesh nodes and are applied to the locator nodes by linear interpolation. For the mass flow variable, CFD-Post uses the integration point mass flow data if it is available; otherwise, it will approximate mass flow values based on mesh node values of velocity (and density, if available).

14.5.19. Mass Flow Technical Note

When integration point mass flow data is stored, backflow through the 2D locator may occur as an artifact of how the mass flow data is applied to the locator nodes, even though there may be no actual backflow (as evidenced by a vector plot on the locator). The figure below illustrates how this may occur.

Figure 14.1 Backflow



In order to visualize this type of backflow through a locator, try making a contour plot of the variable `Mass Flow`, setting a user defined **Range** from 0 to 1 and the **# of Contours** to 3. This will produce a contour plot with two color bands: one for each general flow direction. This visualization technique works because the method of applying integration-point mass-flow data to locator nodes is the same for all uses of the mass flow variable involving a 2D locator (contour plots, massFlowAve, massFlowAveAbs, etc.).

14.5.20. massFlowInt

Integrates a variable over the specified 2D location. A weighting function is applied to the variable value at each point based on the mass flow assigned to that point. You can also specify the fluid(s) used to calculate the mass flow at each locator point.

```
[<Phase>.]massFlowInt(<var|Expression>)@<Location>
```

where:

- `[<Phase>.]` is an optional prefix that is not required for single-phase flows. For details, see *CEL Functions with Multiphase Flow* (p. 155).
- `<var|Expression>` is a variable or expression
- `<Location>` is any fluid surfaces (such as Inlets, Outlets, Openings and fluid-fluid interfaces). An error is raised if the location specified is not 2D.

14.5.20.1. Tools > Command Editor Example

```
>calculate massFlowInt, <var|Expression>, <Location>, [<Phase>]
```

14.5.20.2. Tools > Function Calculator Example

This example integrates pressure over `Plane1`. The result is the pressure force acting on Plane1 weighted by the mass flow assigned to each point on `Plane1`:

Function: `massFlowInt`, **Location:** `Plane1`, **Variable:** `Pressure`, **Phase:** `All Fluids`

14.5.21. massInt

The mass-weighted integration of a variable within a domain or subdomain.

```
massInt(<var|Expression>)@<Location>
```

where:

- `<var>` is a variable
- `<Location>` is any 3D region (such as a domain or subdomain)

14.5.21.1. Tools > Command Editor Example

```
>calculate massInt, <var>, <Location>
```

14.5.22. maxVal

Returns the maximum value of the specified variable on the specified locator. You should create a User Variable if you want to find the maximum value of an expression.

```
maxVal(<var|Expression>)@<Location>
```

where:

- `<var|Expression>` is a variable or expression
- `<Location>` in CFX-Solver is any 2D or 3D region (such as a domain or subdomain); in CFD-Post, Point and 1D, 2D, and 3D locators can be specified.

14.5.22.1. Tools > Command Editor Example

```
>calculate maxVal, <var|Expression>, <Location>
```

14.5.22.2. Tools > Function Calculator Example

This will return the maximum Yplus value on the default wall boundaries:

Function: `maxVal`, **Location:** `Default`, **Variable:** `Yplus`

14.5.23. minVal

Returns the minimum value of the specified variable on the specified locator. You should create a User Variable if you want to find the minimum value of an expression.

```
minVal(<var|Expression>)@<Location>
```

where:

- <var|Expression> is a variable or expression
- <Location> in CFX-Solver is any 2D or 3D region (such as a domain or subdomain); in CFD-Post, Point and 1D, 2D, and 3D locators can be specified.

14.5.23.1. Tools > Command Editor Example

```
>calculate minVal, <var|Expression>, <Location>
```

14.5.23.2. Tools > Function Calculator Example

These settings will return the minimum temperature in the domain:

Function: minVal, **Location:** MainDomain, **Variable:** Temperature

14.5.24. probe

Returns the value of the specified variable on the specified Point object.

```
probe(<var|Expression>)@<Location>
```

where:

- <var|Expression> is a variable or expression
- <Location> is any point object (such as a Source Point or Cartesian Monitor Point).

Important

This calculation should be performed only for point locators described by single points. Incorrect solutions will be produced for multiple point locators.

14.5.24.1. Tools > Command Editor Example

```
>calculate probe, <Expression>, <Location>
```

14.5.24.2. Tools > Function Calculator Example

This example returns the density value at Point1:

Function: probe, **Location:** Point1, **Variable:** Density

14.5.25. rbstate

Returns the value of the specified rigid body state variable, or axis component thereof, on the specified:

- Rigid Body object (for a standard rigid body definition), or
- Immersed Solid domain that is governed by a rigid body solution.

The rigid body state variables are:

- Position
- Linear Velocity

- Linear Acceleration
- Euler Angle
- Angular Velocity
- Angular Acceleration

Syntax:

```
rbstate(<rbvar>[<Axis>])@<Location>
```

where:

- <rbvar> is a rigid body state variable.
- <Axis> is X, Y, or Z.

  For the Euler Angle rigid body state variable, an axis must be specified. For all of the other rigid body state variables, the axis is optional. If you do not specify an axis, the magnitude of the vector is returned. For example, if the variable is Position and you do not specify an axis, the distance from the origin will be returned.
- <Location> is any rigid body object or any immersed solid domain that is governed by a rigid body solution.

Results are given with respect to the global coordinate frame (Coord 0).

14.5.25.1. Expressions Details View Example

```
rbstate(Linear Velocity Z)@Buoy
```

14.5.26. rmsAve

Returns the RMS average of the specified variable within a domain.

```
rmsAve(<var>)@<Location>
```

where:

- <var> is a variable
- <Location> is any 2D region (such as a domain or subdomain).

14.5.26.1. Tools > Command Editor Example

```
>calculate rmsAve, <var>, <Location>
```

14.5.27. sum

Computes the sum of the specified variable values at each point on the specified location.

```
sum(<var|Expression>)@<Location>
```

where:

- <var|Expression> is a variable or expression
- <Location> in CFX-Solver is any 3D region (such as a domain or subdomain); in CFD-Post, Point and 1D, 2D, and 3D locators can be specified.

14.5.27.1. Tools > Command Editor Example

```
>calculate sum, <var|Expression>, <Location>
```

14.5.27.2. Tools > Function Calculator Example

This example returns the sum of the finite volumes assigned to each node in the location `SubDomain1`. In this case, this sums to the volume of the subdomain:

Function: `sum`, **Location:** `SubDomain1`, **Variable:** `Volume of Finite Volume`

14.5.28. torque

Returns the torque on a 2D locator about the specified axis. The force calculated during evaluation of the torque function has the same behavior as the force function. For details, see *force* (p. 165). You can select the fluids involved in the calculation.

```
[<Phase>.]torque_[<Axis>[_<Coord Frame>] ]()@<Location>
```

where:

- `[<Phase>.]` is an optional prefix that is not required for single-phase flows. For details, see *CEL Functions with Multiphase Flow* (p. 155).
- `<Axis>` is x, y, or z
- `<Coord Frame>`
- `<Location>` is any 2D region (such as a wall). If the location specified is not 2D, an error is raised.

14.5.28.1. Tools > Command Editor Example

```
>calculate torque, <Location>, <Axis>, [<Phase>]
```

14.5.28.2. Tools > Function Calculator Example

This example calculates the torque on `Plane1` about the z-axis due to all fluids in the domain.

Function: `torque`, **Location:** `Plane1`, **Axis:** `Global Z`, **Phase:** `All Fluids`

14.5.29. volume

Calculates the volume of a 3D location.

```
volume()@<Location>
```

where:

- `<Location>` is any 3D region (such as a domain or subdomain). An error is raised if the location specified is not a 3D object. For details, see *volume* (p. 177).

14.5.29.1. Tools > Command Editor Example

```
>calculate volume, <Location>
```

14.5.29.2. Tools > Function Calculator Example

This example returns the sum of the volumes of each mesh element included in the location `Volume1`.

Function: volume, **Location:** Volume1

14.5.30. volumeAve

Calculates the volume-weighted average of an expression on a 3D location. This is the 3D equivalent of the areaAve function. The volume-weighted average of a variable is the average value of the variable on a location weighted by the volume assigned to each point on a location. Without the volume weighting function, the average of all the nodal variable values would be biased towards values in regions of high mesh density. The following example demonstrates use of the function.

```
volumeAve(<var|Expression>)@<Location>
```

where:

- <var|Expression> is a variable or expression
- <Location> is any 3D region (such as a domain or subdomain).

14.5.30.1. Tools > Command Editor Example

```
>calculate volumeAve, <var|Expression>, <Location>
```

14.5.30.2. Tools > Function Calculator Example

This example calculates the volume-weighted average value of density in the region enclosed by the location Volume1:

Function: volumeAve, **Location:** Volume1, **Variable:** Density

14.5.31. volumeInt

Integrates the specified variable over the volume location. This is the 3D equivalent of the areaInt function.

```
volumeInt(<var|Expression>)@<Location>
```

where:

- <var|Expression> is a variable or expression
- <Location> is any 3D region (such as a domain or subdomain). An error is raised if the location specified is not a 3D object.

For example, volumeInt(Density)@StaticMixer will calculate the total fluid mass in the domain StaticMixer.

Note

Because the Density variable represents the average density during the timestep rather than the density at the end of the timestep, the volumeInt(Density) does not accurately give the mass of fluid at the end of a timestep. Use the mass() function instead.

14.5.31.1. Tools > Command Editor Example

```
>calculate volumeInt, <var|Expression>, <Location>
```

14.5.31.2. Tools > Function Calculator Example

This example calculates the integral of density (the total mass) in `Volume1`.

Function: `volumeInt`, **Location:** `Volume1`, **Variable:** `Density`

Chapter 15: Variables in ANSYS CFX

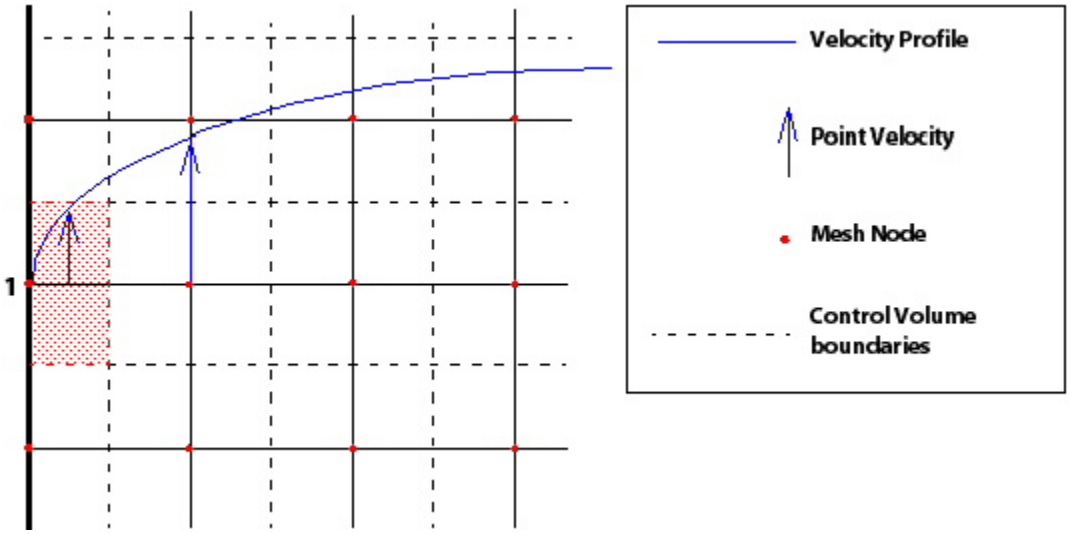
This chapter describes the variables available in ANSYS CFX:

15.1. Hybrid and Conservative Variable Values

The CFX-Solver calculates the solution to your CFD problem using polyhedral finite volumes surrounding the vertices of the underlying mesh elements (hexahedrons, tetrahedrons, prisms, pyramids). Analytical solutions to the Navier-Stokes equations exist for only the simplest of flows under ideal conditions. To obtain solutions for real flows, a numerical approach must be adopted whereby the equations are replaced by algebraic approximations which may be solved using a numerical method.

The solution values on the boundary vertices, called *conservative values*, are the values obtained from solving the conservation equations for the boundary control volumes. These values are not necessarily the same as the specified boundary condition values, although the specified boundary value is used to close boundary fluxes for the boundary control volume. For example, on a no-slip wall, the wall velocity is used to compute the viscous force for the boundary face of the boundary control volume, but the resulting control volume equation solution will not necessarily be the wall velocity. The conservative values are representative of the boundary control volume, not the boundary itself. For visualization purposes, it is often useful to view the specified boundary condition value for the boundary vertices rather than the conservative values. This is especially true when the value of a conservative solution variable (such as pressure or temperature, for instance) is specified at a particular boundary condition. The specified boundary values are called *hybrid values*. CFD-Post uses hybrid values by default for most variables. Hybrid values are obtained by overwriting the conservative results on the boundary nodes produced by the CFX-Solver with values based on the specified boundary conditions. This ensures, for example, that the velocity is displayed as zero on no-slip walls. For quantitative calculations, the conservative values should normally be used because they are consistent with the discrete solutions obtained by the solver. If you want to use these values in CFD-Post, you can select them from the **Variables Editor** dialog box as described above. By default, CFD-Post uses conservative values when the **Calculate** command is used.

The difference between hybrid and conservative values at wall boundaries can be demonstrated using the following figure:



Using velocity as an example, the velocity value calculated at a mesh node is based upon the 'average' in the control volume surrounding that node. For calculation purposes, the entire control volume is then assumed to possess that velocity. At a boundary node, its surrounding control volume includes an area in the bulk of the fluid (this area is highlighted around the boundary node marked **1**). Hence, the conservative velocity calculated at the wall node is not zero, but an 'average' over the control volume adjacent to the boundary. At a wall boundary node the difference between conservative and hybrid values can be illustrated by considering the case of the mass flow rate through the wall-adjacent control volume. If a zero velocity was enforced at the boundary node, then this would produce zero mass flow through the control volume, which is clearly not correct.

15.1.1. Solid-Fluid Interface Variable Values

15.1.1.1. Conservative Values at 1:1 Interface

At a solid-fluid 1:1 interface, duplicate nodes exist. The conservative value for the solid-side node is the variable values averaged over the half on the control volume that lies inside the solid. The conservative value for the fluid-side node is the variable values averaged over the half of the control volume that lies in the fluid.

Consider the example of heat transfer from a hot solid to a cool fluid when advection dominates within the fluid. If you create a plot across the solid-fluid interface using conservative values of temperature, then you will see a sharp change in temperature across the interface. This is because values are interpolated from the interface into the bulk of the solid domain using the value for the solid-side node at the interface, while values are interpolated from the interface into the bulk of the fluid domain using the value for the fluid-side node at the interface. This results in a temperature discontinuity at the interface.

15.1.1.2. Hybrid Values at 1:1 Interface

When creating plots using hybrid variable values (the default in CFD-Post), the 1:1 interface is single valued and takes the solid-side conservative value. You can therefore expect to see the same plot within the solid, but the temperature profile between the interface and the first node in the fluid interpolates between the solid-side interface value and the first fluid node value. In this case, a discontinuity does not exist because all nodes are single valued.

Conservative values should be used for all quantitative calculations.

15.1.1.3. Conservative Values on a GGI Interface

At a GGI interface, the CFX Solver calculates both fluid-side and solid-side temperatures based on heat flux conservation. These values are representative of the temperature within the half-control volumes around the vertices on the interface. The fluid-side and solid-side temperatures are generally not equal. As a result, a plot of conservative values of temperature will generally show a discontinuity across a GGI interface.

15.1.1.4. Hybrid Values on a GGI Interface

At a GGI interface, the CFX Solver calculates a "surface temperature" based on a flux-conservation equation for the 'control surfaces' that lie between the fluid side and the solid side. The surface temperature is usually between the fluid-side and solid-side temperatures. Hybrid values of temperature on a GGI interface are set equal to the surface temperature. As a result, there is no discontinuity in hybrid values of temperature across a GGI interface.

15.2. List of Field Variables

This section contains a list of field variables that you may have defined in CFX-Pre or that are available for viewing in CFD-Post and exporting to other files. Many variables are relevant only for specific physical models.

The information given in this section includes:

- **Long Variable Name**: The name that you see in the user interface.
- **Short Variable Name**: The name that must be used in CEL expressions.
- **Units**: The default units for the variable. An empty entry [ ] indicates a dimensionless variable.

Note

The entries in the Units columns are SI but could as easily be any other system of units.

- In the **Availability** column:
  - A number represents the user level (1 indicates that the variable appears in default lists, 2 and 3 indicate that the variable appears in extended lists that you see when you click 

). This number is useful when using the CFX Export facility. For details, see File Export Utility in the *CFX-Solver Manager User's Guide*. Note that the CFX-Solver may sometimes override the user-level setting depending on the physics of the problem. In these cases, the User Level may be different from that shown in the tables that follow.
  - **Boundary (B)**: A **B** in this column indicates that the variable contains only non-zero values on the boundary of the model. See Boundary-Value-Only Variables in the *CFD-Post User's Guide* for more details.

    Boundary-Value-Only Variables in the *CFD-Post User's Guide* describes the useful things that you can do with variables that are defined only on the boundaries of the model.
  - A indicates the variable is available for mesh adaption
  - C indicates the variable is available in CEL
  - DT indicates the variable is available for data transfer to ANSYS

  - M indicates the variable is available for monitoring
  - P indicates the variable is available for particle user-routine argument lists
  - PR indicates the variable is available for particle results
  - R indicates the variable is available to be output to the results, transient results, and backup files
  - RA indicates the variable is available for radiation results
  - TS indicates the variable is available for transient statistics
- **Definition**: Defines the variable.

This is not a complete list of variables. Information on obtaining details on all variables is available in RULES and VARIABLES Files in the *CFX-Solver Manager User's Guide*.

Note

Variables with names shown in **bold text** are not output to CFD-Post. However, some of these variables can be output to CFD-Post by selecting them from the **Extra Output Variables List** on the **Results** tab of the **Solver** > **Output Control** details view of CFX-Pre.

15.2.1. Common Variables Relevant for Most CFD Calculations

The following table contains a list of variables (with both long and short variable names) that can be used when working with CFD calculations. For an explanation of the column headings, see *List of Field Variables* (p. 183).

Long Variable Name	Short Variable Name	Units	Availability	Definition
Density	density	[kg m^-3]	1 A, C, M, P, R, TS	For **Fixed** and **Variable Composition Mixture**, the density is determined by a mass fraction weighted harmonic average: $\frac{Y_A}{\rho_A} + \frac{Y_B}{\rho_B} + \ldots + \frac{Y_N}{\rho_N} = \frac{1}{\rho_{mix}}$
Dynamic Viscosity	viscosity	[kg m^-1 s^-1]	2 A, C, M, P, R, TS	Dynamic viscosity (μ), also called *absolute viscosity*, is a measure of the resistance of a fluid to shearing forces, and appears in the momentum equations. Using an expression to set the dynamic viscosity is possible. For details, see Non-Newtonian Flow in the *CFX-Solver Modeling Guide*.

Long Variable Name	Short Variable Name	Units	Availability	Definition
Velocity[a]	vel	[m s^-1]	1 A, C, M, P, R, TS	Velocity vector.
Velocity u Velocity v Velocity w	u v w	[m s^-1]	1 A, C, M, P, R, TS	Components of velocity.
Pressure	p	[kg m^-1 s^-2]	1 A, C, M, P, R, TS	Both `Pressure` and `Total Pressure` are measured relative to the reference pressure that you specified on the **Domains** panel in CFX-Pre. Additionally, `Pressure` is the total normal stress, which means that when using the k-e turbulence model, `Pressure` is the thermodynamic pressure plus the turbulent normal stress. `Static Pressure` is the thermodynamic pressure, in most cases this is the same as `Pressure`. For details, see Modified Pressure in the *CFX-Solver Theory Guide*.
Static Pressure	pstat	[kg m^-1 s^-2]	3	CFX solves for the relative `Static Pressure` (thermodynamic pressure) p_{stat} in the flow field, and is related to `Absolute Pressure` $p_{\text{abs}} = p_{\text{stat}} + p_{\text{ref}}$.
Total Pressure	ptot	[kg m^-1 s^-2]	2 A, C, M, P, R, TS	The total pressure, p_{tot}, is defined as the pressure that would exist at a point if the fluid was brought instantaneously to rest such that the dynamic energy of the flow converted to

Long Variable Name	Short Variable Name	Units	Availability	Definition
				pressure without losses. The following three sections describe how total pressure is computed for a pure component material with constant density, ideal gas equation of state and a general equation of state (CEL expression or RGP table). For details, see Scalable Wall Functions in the *CFX-Solver Theory Guide*.
Wall Shear	wall shear	Pa	3,**B**	For details, see Scalable Wall Functions in the *CFX-Solver Theory Guide*.
Volume of Finite Volume			3 C, DT, R, TS	Volume of finite volume. For details, see Discretization of the Governing Equations in the *CFX-Solver Theory Guide*.
X coordinate	x	[m]	2 C	Cartesian coordinate components.
Y coordinate	y	[m]	2 C	
Z coordinate	z	[m]	2 C	
Kinematic Diffusivity	visckin		2 C, M, P, R, TS	*Kinematic diffusivity* describes how rapidly a scalar quantity would move through the fluid in the absence of convection. For convection-dominated flows, the kinematic diffusivity can have little effect because convection processes dominate over diffusion processes.
Shear Strain Rate	sstrnr	[s^-1]	2 A, C, M, R, TS	For details see Non-Newtonian Flow in the *CFX-Solver Modeling Guide*.

Long Variable Name	Short Variable Name	Units	Availability	Definition
Specific Heat Capacity at Constant Pressure	Cp	[m^2 s^-2 K^-1]	2 A, C, M, R, TS	For details, see Specific Heat Capacity in the *CFX-Solver Modeling Guide*.
Specific Heat Capacity at Constant Volume	**Cv**	[m^2 s^-2 K^-1]	2 A, C, M, P, R, TS	
Thermal Conductivity	cond	[kg m s^-3 K^-1]	2 A, C, M, R, TS	Thermal conductivity, λ, is the property of a fluid that characterizes its ability to transfer heat by conduction. For details, see Thermal Conductivity in the *CFX-Solver Modeling Guide*.
Temperature	T	[K]	1 A, C, DT, M, P, R, TS	The static temperature, T_{stat}, is the thermodynamic temperature, and depends on the internal energy of the fluid. In CFX, depending on the heat transfer model you select, the flow solver calculates either total or static enthalpy (corresponding to the total or thermal energy equations). For details, see Static Temperature in the *CFX-Solver Theory Guide*.
Total Temperature	Ttot	[K]	1 A, C, M, P, R, TS	The total temperature is derived from the concept of total enthalpy and is computed exactly the same way as static temperature, except that total enthalpy is used in the property relationships. For details, see Total Temperature in the *CFX-Solver Theory Guide*.

Long Variable Name	Short Variable Name	Units	Availability	Definition
Wall Heat Flux	Qwall	[W m^-2]	2,B C, DT, R, TS	A heat flux is specified across the wall boundary. A positive value indicates heat flux into the domain. For multiphase cases, when the bulk heat flux into both phases is set, this option is labeled Wall Heat Flux instead of Heat Flux. When set on a per fluid basis, this option is labelled Heat Flux.
Wall Heat Transfer Coefficient	htc	[W m^-2 K^-1]	2,B C, R, TS	For details, see Heat Transfer Coefficient and Wall Heat Transfer Coefficient in the *CFX-Solver Modeling Guide*.
Total Enthalpy	htot	[m^2 s^-2]	A, C, M, R, TS	h_{tot} For details, see Transport Equations in the *CFX-Solver Theory Guide*.
Static Enthalpy	enthalpy	[m^2 s^-2]	2 A, C, M, P, R, TS	For details, see Static Enthalpy in the *CFX-Solver Theory Guide*.

[a]When a rotating frame of reference is used, all variables in the CFX-5 results file are relative to the rotating frame, unless specified as a `Stn Frame` variable.

15.2.2. Variables Relevant for Turbulent Flows

The following table contains a list of variables (with both long and short variable names) that can be used when working with turbulent flows. For an explanation of the column headings, see *List of Field Variables* (p. 183).

A **B** in the **Type** column indicates that the variable contains only non-zero values on the boundary of the model.

Long Variable Name	Short Variable Name	Units	Availability	Definition
Blending Function for DES model	desbf	[ ]	2 C, M, R, TS	Controls blending between RANS and LES regimes for the DES model

Long Variable Name	Short Variable Name	Units	Availability	Definition
Turbulence Kinetic Energy	ke	[m^2 s^-2]	1 A, C, M, P, R, TS	For details, see The k-epsilon Model in the *CFX-Solver Modeling Guide*.
Turbulence Eddy Dissipation	ed	[m^2 s^-3]	1 A, C, M, P, R, TS	The rate at which the velocity fluctuations dissipate. For details, see The k-epsilon Model in the *CFX-Solver Modeling Guide*.
Turbulent Eddy Frequency	tef	[s^-1]	1 A, C, M, P, R, TS	
Eddy Viscosity	eddy viscosity	[kg m^-1 s-1]	2 A, C, M, P, R, TS	The "eddy viscosity model" proposes that turbulence consists of small eddies that are continuously forming and dissipating, and in which the Reynolds stresses are assumed to be proportional to mean velocity gradients. For details, see Eddy Viscosity Turbulence Models in the *CFX-Solver Theory Guide*.
Reynolds Stress	rs	[m^2 s^-2]	2 A, C, M, P, R, TS	This is a tensor quantity with six components. For details, see Statistical Reynolds Stresses in the *CFX-Solver Modeling Guide* and Reynolds Stress Turbulence Models in the *CFX-Solver Theory Guide* in the ANSYS CFX documentation.
Statistical Reynolds Stress uu	rsstat uu	[m^2 s^-2]	3 M, R	In LES runs, Reynolds Stress components are automatically generated using running statistics of the instantaneous, transient velocity field. For details, see Statistical Reynolds Stresses in the *CFX-Solver Modeling Guide*.
Statistical Reynolds Stress vv	rsstat vv	[m^2 s^-2]	3 M, R	

Long Variable Name	Short Variable Name	Units	Availability	Definition
Statistical Reynolds Stress ww	rsstatww	[m^2 s^-2]	3 M, R	
Statistical Reynolds Stress uv	rsstatuv	[m^2 s^-2]	3 M, R	
Statistical Reynolds Stress uw	rsstatuw	[m^2 s^-2]	3 M, R	
Statistical Reynolds Stress vw	rsstatvw	[m^2 s^-2]	3 M, R	
Velocity Correlation uu	uu	[m^2 s^-2]	3 C, M, R	For details, see Statistical Reynolds Stresses in the *CFX-Solver Modeling Guide*.
Velocity Correlation vv	vv	[m^2 s^-2]	3 C, M, R	
Velocity Correlation ww	ww	[m^2 s^-2]	3 C, M, R	
Velocity Correlation uv	uv	[m^2 s^-2]	3 C, M, R	
Velocity Correlation uw	uw	[m^2 s^-2]	3 C, M, R	
Velocity Correlation vw	vw	[m^2 s^-2]	3 C, M, R	
Yplus	yplusstd	[ ]	2,**B** C, R, TS	A variable based on the distance from the wall to the first node and the wall shear stress. For details, see Solver Yplus and Yplus in the *CFX-Solver Modeling Guide*.

Long Variable Name	Short Variable Name	Units	Availability	Definition
Solver Yplus	yplus	[ ]	2,**B** C, R, TS	A deprecated internal variable. For details, see Solver Yplus and Yplus in the *CFX-Solver Modeling Guide*.

15.2.3. Variables Relevant for Buoyant Flow

The following table contains a list of variables (with both long and short variable names) that can be used when working with buoyant flows. For an explanation of the column headings, see *List of Field Variables* (p. 183).

Long Variable Name	Short Variable Name	Units	Availability	Definition
Thermal Expansivity	beta	[K^ -1]	2 C	For details, see Buoyancy in the *CFX-Solver Modeling Guide*.

15.2.4. Variables Relevant for Compressible Flow

The following table contains a list of variables (with both long and short variable names) that can be used when working with compressible flows.

Long Variable Name	Short Variable Name	Units	Availability	Definition
Isobaric Compressibility	compisoP	[K^-1]	2 C, M, R	$-\frac{1}{\rho}\frac{\partial \rho}{\partial T}\Big\vert_{p}$
Isothermal Compressibility	compisoT	[m s^2 kg^-1]	2 C, M, R	Defines the rate of change of the system volume with pressure. $\frac{1}{\rho}\frac{\partial \rho}{\partial p}\Big\vert_{T}$
Mach Number	Mach	[ ]	1 A, C, M, R, TS	For details, see List of Symbols in the *CFX-Solver Theory Guide*.

Long Variable Name	Short Variable Name	Units	Availability	Definition
Shock Indicator	shock indicator	[ ]	2 A, C, M, R, TS	The variable takes a value of 0 away from a shock and a value of 1 in the vicinity of a shock.
Isentropic Compressibility	compisoS	[m s^2 kg^-1]	2 C, M, R	The extent to which a material reduces its volume when it is subjected to compressive stresses at a constant value of entropy. $\left(\frac{1}{\rho}\right)\left(\frac{\partial \rho}{\partial p}\right)_s$

15.2.5. Variables Relevant for Particle Tracking

The following table contains a list of variables (with both long and short variable names) that can be used when working with compressible flows.

Long Variable Name	Short Variable Name	Units	User Level	Definition
Latent Heat	lheat	[ ]	2 C, R, M	User-specified latent heat for phase pairs involving a particle phase.
Particle Momentum Source	ptmomsrc	[ ]	2 A, C, M, P, R	Momentum source from particle phase to continuous phase.
Particle Diameter	particle diameter	[ ]	3 A, C, M, R	Diameter of a particle phase.

15.2.6. Variables Relevant for Calculations with a Rotating Frame of Reference

The following table contains a list of variables (with both long and short variable names) that can be used when working with a rotating frame of reference. For an explanation of the column headings, see *List of Field Variables* (p. 183).

Long Variable Name	Short Variable Name	Units	Availability	Definition
Total Pressure in Stn Frame	ptot-stn	[kg m^-1 s^-2]	2 A, C, M, P, R, TS	The velocity in the rotating frame of reference is defined as: $\boldsymbol{U}_{\mathrm{rel}} = \boldsymbol{U}_{\mathrm{stn}} - \boldsymbol{\omega} \times \boldsymbol{R}$ where ω is the angular velocity, $\boldsymbol{R}$ is the local radius vector, and $\boldsymbol{U}_{\mathrm{stn}}$ is velocity in the stationary frame of reference. For details, see Rotating Frame Quantities in the *CFX-Solver Theory Guide*.
Total Temperature in Stn Frame	Ttot-stn	[K]	2 A, C, DT, M, P, R, TS	
Total Enthalpy in Stn Frame	htot-stn	[kg m^2 s^-2]	2 A, C, M, R, TS	
Mach Number in Stn Frame	Mach-stn	[ ]	1 A, C, M, R, TS	
Velocity in Stn Frame	velstn	[m s^-1]	1 A, C, M, R, TS	

15.2.7. Variables Relevant for Parallel Calculations

The following table contains a list of variables (with both long and short variable names) that can be used when working with parallel calculations. For an explanation of the column headings, see *List of Field Variables* (p. 183).

Long Variable Name	Short Variable Name	Units	Availability	Definition
Real Partition Number		[ ]	2 C, M, R	The partition that the node was in for the parallel run.

15.2.8. Variables Relevant for Multicomponent Calculations

The following table contains a list of variables (with both long and short variable names) that can be used when working with multicomponent calculations. For an explanation of the column headings, see *List of Field Variables* (p. 183).

Long Variable Name	Short Variable Name	Units	Availability	Definition
Mass Fraction	mf	[ ]	1 A, C, M, P, R, TS	The fraction of a component in a multicomponent fluid by mass.
Mass Concentration	mconc	[kg m^-3]	2 A, C, M, P, R, TS	The concentration of a component.

15.2.9. Variables Relevant for Multiphase Calculations

The following table contains a list of variables (with both long and short variable names) that can be used when working with multiphase calculations. For an explanation of the column headings, see *List of Field Variables* (p. 183).

Long Variable Name	Short Variable Name	Units	Availability	Definition
Interfacial Area Density	area density	[m^-1]	3 C	Interface area per unit volume for Eulerian multiphase fluid pairs.
Interphase Mass Transfer Rate	ipmt rate	[ ]	3 C	Interface mass transfer rate for Eulerian multiphase fluid pairs.
Mean Particle Diameter	mean particle diameter	[m]	3 A, C, M	Mean particle diameter for an Eulerian dispersed phase (including Polydispersed Fluids).
Volume Fraction	vf	[ ]	1 A, C, M, P, R, TS	For details, see Volume Fraction in the *CFX-Solver Modeling Guide*.
Conservative Volume Fraction	vfc	[ ]	2 A, C, M, R, TS	For details, see Volume Fraction in the *CFX-Solver Modeling Guide*.

Long Variable Name	Short Variable Name	Units	Availability	Definition
Drift Velocity	drift velocity	[ ]	2 C, M, R, TS	Velocity of an algebraic slip component relative to the mixture.
Slip Reynolds Number	slip Re	[ ]	3 C	Reynolds number for Eulerian multiphase fluid pairs.
Slip Velocity	slipvel	[ ]	1 C, M, R, TS	Velocity of an algebraic slip component relative to the continuous component.
Surface Tension Coefficient	surface tension coefficient	[N m^-1]	2 C	Surface tension coefficient between fluids in a fluid pair.
Unclipped Interfacial Area Density	unclipped area density	[m^-1]	3 C	Similar to area density, but values are not clipped to be non-zero.
Superficial Velocity	volflx	[m s^-1]	1 A, C, M, R, TS	The Fluid.Volume Fraction multiplied by the Fluid.Velocity.

15.2.10. Variables Relevant for Radiation Calculations

The following table contains a list of variables (with both long and short variable names) that can be used when working with radiation calculations. For an explanation of the column headings, see *List of Field Variables* (p. 183).

A **B** in the **Type** column indicates that the variable contains only non-zero values on the boundary of the model.

Long Variable Name	Short Variable Name	Units	Availability	Definition
Wall Radiative Heat Flux	Qrad	[W m^-2]	2,**B** DT, R, TS	Wall Radiative Heat Flux represents the net radiative energy flux leaving the boundary. It is computed as the difference between the radiative emission and the incoming radiative flux (Wall Irradiation Flux).
Wall Heat Flux	Qwall	[W m^-2]	2,**B** C, DT, R, TS	Wall Heat Flux is sum of the Wall Radiative Heat Flux and the Wall Convective Heat Flux. For an adiabatic wall, the sum should be zero.
Wall Irradiation Flux	irrad	[W m^-2]	2,**B** C, DT, R, TS	Wall Irradiation Flux represents the incoming radiative flux. It is computed as the solid angle integral of the incoming Radiative Intensity over a hemisphere on the boundary. For simulations using the multiband model, the Wall Irradiation Flux for each spectral band is also available for post-processing.

15.2.11. Variables for Total Enthalpies, Temperatures, and Pressures

The following table lists the names of the various total enthalpies, temperatures, and pressures when visualizing results in CFD-Post or for use in CEL expressions. For an explanation of the column headings, see *List of Field Variables* (p. 183).

Long Variable Name	Short Variable Name	Units	Availability	Definition
Total Enthalpy	htot	[m^2 s^-2]	A, C, M, R, TS	h_{tot} For details, see Transport Equations in the *CFX-Solver Theory Guide*.
Rothalpy	rothalpy	[m^2 s^-2]	A, C, M, R, TS	I
Total Enthalpy in Stn Frame	htotstn	[m^2 s^-2]	A, C, M, R, TS	$h_{\mathrm{tot,stn}}$
Total Temperature in Rel Frame	Ttotrel	[K]	A, C, DT, M, P, R, TS	$T_{\mathrm{tot,rel}}$

Long Variable Name	Short Variable Name	Units	Availability	Definition
Total Temperature	Ttot	[K]	A, C, DT, M, P, R, TS	T_{tot}
Total Temperature in Stn Frame	Ttotstn	[K]	A, C, DT, M, P, R, TS	$T_{\text{tot,stn}}$
Total Pressure in Rel Frame	ptotrel	[kg m^-1 s^-2]	A, C, M, P, R, TS	$P_{\text{tot,rel}}$
Total Pressure	ptot	[kg m^-1 s^-2]	A, C, M, P, R, TS	P_{tot}
Total Pressure in Stn Frame	ptotstn	[kg m^-1 s^-2]	A, C, M, P, R, TS	$P_{\text{tot,stn}}$

15.2.12. Variables and Predefined Expressions Available in CEL Expressions

The following is a table of the more common variables and predefined expressions that are available for use with CEL when defining expressions. To view a complete list, open the **Expressions** workspace. For an explanation of the column headings, see *List of Field Variables* (p. 183).

Many variables and expressions have a long and a short form (for example, *Pressure* or *p*).

Additional Variables and expressions are available in CFD-Post. For details, see CFX Expression Language (CEL) in CFD-Post in the *CFD-Post User's Guide*.

Table 15.1 Common CEL Single-Value Variables and Predefined Expressions

Long Variable Name	Short Variable Name	Units	Availability	Definition
Accumulated Coupling Step	acplgstep	[ ]	2 C	These single-value variables enable access to timestep, timestep interval, and iteration number in CEL expressions. They may be useful in setting parameters such as the Physical Timescale via CEL expressions. For details, see *Timestep, Timestep Interval, and Iteration Number Variables* (p. 207).
Accumulated Iteration Number	aitern	[ ]	2 C	
Accumulated Time Step	atstep	[ ]	2 C	

Long Variable Name	Short Variable Name	Units	Availability	Definition
Current Iteration Number	citern	[ ]	2 C	
Current Stagger Iteration	cstagger	[ ]	2 C	
Current Time Step	ctstep	[ ]	2 C	
Sequence Step	sstep	[ ]	2 C	
Time Step Size	dtstep	[s]	2 C	
Time	t	[s]	2 C	

Note

Variables with names shown in **bold text** in the tables that follow are not output to CFD-Post. However, some of these variables can be output to CFD-Post by selecting them from the **Extra Output Variables List** on the **Results** tab of the **Solver** > **Output Control** details view in CFX-Pre.

Table 15.2 Common CEL Field Variables and Predefined Expressions

Long Variable Name	Short Variable Name	Units	Availability	Definition
Axial Distance	aaxis	[m]	2 C	Axial spatial location measured along the locally-defined axis from the origin of the latter. When the locally-defined axis happens to be the z-axis, z and aaxis are identical.
Absorption Coefficient	absorp	[m^-1]	1 C, M, R, TS	The property of a medium that describes the amount of absorption of thermal radiation per unit path length

Long Variable Name	Short Variable Name	Units	Availability	Definition
				within the medium. It can be interpreted as the inverse of the mean free path that a photon will travel before being absorbed (if the absorption coefficient does not vary along the path).
Boundary Distance	bnd distance	[m]	2 A, C, M, R, TS	
Boundary Scale	**bnd scale**	[m^-2]	3 C, M, R, TS	
Contact Area Fraction	**af**	[ ]	3 M	
[AV name]	[AV name]			Additional Variable name
Thermal Expansivity	beta	[K^-1]	2 C	
Effective Density	**deneff**	[kg m^-3]	3 A, C, M, R, TS	
Density	density	[kg m^-3]	2 A, C, M, P, R, TS	
Turbulence Eddy Dissipation	ed	[m^2 s^-3]	1 A, C, M, P, R, TS	
Eddy Viscosity	eddy viscosity	[kg m^-1 s^-1]	1 A, C, M, P, R, TS	
Emissivity	emis	[ ]	1 C	

Long Variable Name	Short Variable Name	Units	Availability	Definition
Extinction Coefficient	**extinct**	[m^-1]	1 C	
Initial Cartesian Coordinates	initcartcrd	[m]	2 C	The position of each node as it was at the start of the simulation (that is, the current position with Total Mesh Displacement subtracted). The individual components are referred to as "Initial X", "Initial Y" and "Initial Z".
Turbulence Kinetic Energy	ke	[m^2 s^-2]	1 A, C, M, P, R, TS	
Mach Number	Mach	[ ]	1 A, C, M, R, TS	
Mach Number in Stn Frame	Machstn	[ ]	1 A, C, M, R, TS	Mach Number in Stationary Frame
Mass Concentration	mconc	[m^-3 kg]	2 A, C, M, P, R, TS	Mass concentration of a component
Mass Fraction	mf	[ ]	1 A, C, M, P, R, TS	
Conservative Mass Fraction	**mfc**	[ ]	2 A, C, M, R, TS	
Mean Particle Diameter	mean particle diameter	[m]	3 C, P	
Mesh Displacement	meshdisp	[m]	3 C, M, R, TS	The displacement relative to the previous mesh

Long Variable Name	Short Variable Name	Units	Availability	Definition
Mesh Expansion Factor	mesh exp fact	[ ]	2 C, M, R, TS	Ratio of largest to smallest sector volumes for each control volume.
Mesh Initialisation Time	meshinittime	[s]	2 C	Simulation time at which the mesh was last re-initialized (most often due to interpolation that occurs as part of remeshing)
Mixture Fraction	mixfrc	[ ]	1 A, C, M, R, TS	Mixture Fraction Mean
Mixture Model Length Scale	**mixture length scale**	[m]	3 M	
Mixture Fraction Variance	mixvar	[ ]	1 A, C, M, R, TS	
Molar Concentration	molconc	[m^-3 mol]	2 A, C, M, P, R, TS	
Molar Fraction	molf	[ ]	2 A, C, M, P, R, TS	
Molar Mass	**mw**	[kg mol^-1]	3 C, P	
Orthogonality Angle	orthangle	[rad]	2 C, M, R, TS	A measure of the average mesh orthogonality angle
Orthogonality Angle Minimum	orthanglemin	[rad]	2 C, M, R, TS	A measure of the worst mesh orthogonality angle
Orthogonality Factor	orthfact		2 C, M, R, TS	A non-dimensional measure of the average mesh orthogonality

Long Variable Name	**Short Variable Name**	**Units**	**Availability**	**Definition**
Orthogonality Factor Minimum	orthfactmin		2 C, M, R, TS	A measure of the worst mesh orthogonality angle
Pressure	p	[kg m^-1 s^-2]	1 A, C, M, P, R, TS	
Absolute Pressure	pabs	[kg m^-1 s^-2]	2 A, C, M, R, TS	
Reference Pressure	pref	[kg m^-1 s^-2]	2 C	The `Reference Pressure` is the absolute pressure datum from which all other pressure values are taken. All relative pressure specifications in CFX are relative to the `Reference Pressure`. For details, see Setting a Reference Pressure in the *CFX-Solver Modeling Guide*.
Distance from local z axis	**r**	[m]	2 C	Radial spatial location. $r = \sqrt{x^2 + y^2}$. For details, see *CEL Variables r and theta* (p. 206).
Radius	raxis	[m]	2 C	Radial spatial location measured normal to the locally-defined axis. When the locally-defined axis happens to be the z-axis, r and raxis are identical.
Radiative Emission	rademis	[kg s^-3]	1 RA	
Incident Radiation	radinc	[kg s^-3]	1 C, DT, M, R, TS	

Long Variable Name	Short Variable Name	Units	Availability	Definition
Radiation Intensity	radint	[kg s^-3]	1 A, C, M, P, R, TS	Radiative Emission. This is written to the results file for Monte Carlo simulations as `Radiation Intensity.Normalized Std Deviation`.
Refractive Index	refrac	[ ]	1 C, R, TS	A non-dimensional parameter defined as the ratio of the speed of light in a vacuum to that in a material.
Non dimensional radius	**rNoDim**	[ ]	2 C	Non-dimensional radius (available only when a rotating domain exists). For details, see *CEL Variable rNoDim* (p. 207).
Reynolds Stress	rs uu, rs vv, rs ww, rs uv, rs uw, rs vw	[m^2 s^-2]	2 A, C, M, P, R, TS	The six Reynolds Stress components
Statistical Reynolds Stress	rsstat uu, rsstat vv, rsstat ww, rsstat uv, rsstat uw, rsstat vw	[m^2 s^-2]	3 M, R	The six Statistical Reynolds Stress components
Scattering Coefficient	scatter	[m^-1]	1 C, M, R, TS	The property of a medium that describes the amount of scattering of thermal radiation per unit path length for propagation in the medium. It can be interpreted as the inverse of the mean free path that a photon will travel before undergoing scattering (if the scattering coefficient does not vary along the path).

Long Variable Name	Short Variable Name	Units	Availability	Definition
Soot Mass Fraction	sootmf	[ ]	1 A, C, M, R, TS	
Soot Nuclei Specific Concentration	sootncl	[m^-3]	1 A, C, M, R, TS	
Specific Volume	**specvol**	[m^3 kg^-1]	3 A, C, M, R, TS	
Local Speed of Sound	**speedofsound**	[m s^-1]	2 C, M, R, TS	
Subdomain	**subdomain**	[ ]	2 C	Subdomain variable (1.0 in subdomain, 0.0 elsewhere). For details, see *CEL Variable "subdomain" and CEL Function "inside"* (p. 207).
inside() @<Locations>	**inside() @<Locations>**			inside variable (1.0 in subdomain, 0.0 elsewhere). For details, see *CEL Variable "subdomain" and CEL Function "inside"* (p. 207).
Theta	taxis	[rad]	2 C	taxis is the angular spatial location measured around the locally-defined axis, when the latter is defined by the Coordinate Axis option. When the locally defined axis is the z(/x/y)-axis, taxis is measured from the x(/y/z)-axis, positive direction as per right-hand rule.
Turbulence Eddy Frequency	tef	[s^-1]	1 A, C, M, P, R, TS	

Long Variable Name	**Short Variable Name**	**Units**	**Availability**	**Definition**
Angle around local z axis	**theta**	[rad]	2 C	Angle, arctan(y/x). For details, see *CEL Variables r and theta* (p. 206).
Total Mesh Displacement	mesh-disptot	[m]	1 C, DT, M, R, TS	The total displacement relative to the initial mesh
Velocity u Velocity v Velocity w	u v w	[m s^-1]	1 A, C, M, P, R, TS	Velocity in the x, y, and z coordinate directions
Velocity in Stn Frame u Velocity in Stn Frame v Velocity in Stn Frame w	velstn u velstn v velstn w	[m s^-1]	1 A, C, M, R, TS	Velocity in Stationary Frame in the x, y, and z coordinate directions
Volume Fraction	vf	[ ]	1 A, C, M, P, R, TS	
Conservative Volume Fraction	vfc	[ ]	2 A, C, M, R, TS	The variable `<fluid>.Conservative Volume Fraction` should not usually be used for post-processing.
Kinematic Viscosity	**visckin**	[m^2 s^-1]	2 A, C, M, P, R, TS	
Wall Distance	wall dis-tance	[m]	2 A, C, M, P, R, TS	
Wall Scale	**wall scale**	[m^2]	3 M, R, TS	

15.2.12.1. System Variable Prefixes

In order to distinguish system variables of the different components and fluids in your CFX model, prefixes are used. For example, if carbon dioxide is a material used in the fluid `air`, then some of the system variables that you might expect to see are:

- `air.density` - the density of air
- `air.viscosity` - the viscosity of air
- `air.carbondioxide.mf` - the mass fraction of carbon dioxide in air
- `air | water.surface tension coefficient` – the surface tension coefficient between air and water
- `air | water.area density` – the interfacial area density between air and water.

In a single phase simulation the fluid prefix may be omitted.

For multiphase cases, a fluid prefix indicates either a specific fluid, or a specific fluid pair. The absence of a prefix indicates a bulk or fluid independent variable, such as pressure.

For porous solids, those variables that exist in the solid are prefixed by the name of the solid phase.

15.2.12.2. CEL Variables r and theta

`r` is defined as the normal distance from the third axis with respect to the reference coordinate frame. `theta` is defined as the angular rotation about the third axis with respect to the reference coordinate frame. For details, see Coordinate Frames in the *CFX-Solver Modeling Guide*.

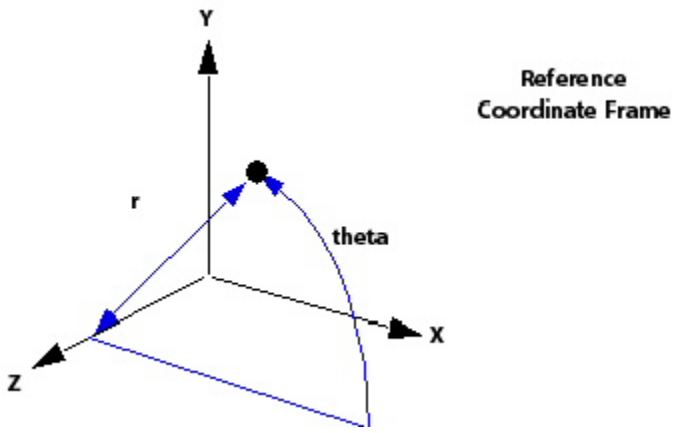
The variables `Radius` and `theta` are available only when the rotational axis has been defined. The rotational axis can either be defined in the results file or in CFD-Post through the **Initialization** panel in the **Turbo** workspace.

Note

`theta` is expressed in radians and will have values between $-\pi$ and π.

`r` and `theta` are particularly useful for describing radial distributions, for instance the velocity profile at the inlet to a pipe.

Figure 15.1 r and theta with Respect to the Reference Coordinate Frame



15.2.12.3. CEL Variable rNoDim

`rNoDim` is a dimensionless system variable that can be useful for rotating machinery applications. It is a ratio of radii, defined to be zero at the minimum radius and unity at the maximum radius, so that in general:

$$rNoDim = \frac{R - R_{\min}}{R_{\max} - R_{\min}}$$

where R is the radius of any point in the domain from the axis of rotation. `rNoDim` is only available for domains defined with a rotating frame of reference.

15.2.12.4. CEL Variable "subdomain" and CEL Function "inside"

`subdomain` is essentially a step function variable, defined to be unity within a subdomain and zero elsewhere. This is useful for describing different initial values or fluid properties in different regions of the domain. It works in all subdomains but cannot be applied to specific subdomains (for example, an expression for temperature in a subdomain could be `373*subdomain [K]`).

The `inside` CEL function can be used in a similar way to the `subdomain` variable, but allows a specific 2D or 3D location to be given. For example, `273 [K] * inside()@Subdomain 1` has a value of 273 [K] at points in `Subdomain 1` and 0 [K] elsewhere. Furthermore, the location can be any 2D or 3D named sub-region of the physical location on which the expression is evaluated. The location can also be an immersed solid domain.

15.2.12.5. Timestep, Timestep Interval, and Iteration Number Variables

These variables allow access to timestep, timestep interval, and iteration number in CEL expressions. They may be useful in setting parameters such as the Physical Timescale via CEL expressions.

In CFD-Post, `sstep` is the 'global' sequence time step. It is equivalent to the **Step** value in the Timestep Selector in the *CFD-Post User's Guide*.

15.2.12.5.1. Steady-State Runs

In steady-state runs, only `aitern` (or, equivalently `atstep`) and `citern` (or, equivalently `ctstep`) are of use. `citern` gives the outer iteration number of the current run. The outer iteration number begins at 1 for each run, irrespective of whether it is a restarted run. `aitern` gives the accumulated outer iteration number, which accumulates across a restarted run.

15.2.12.5.2. Transient Runs

In transient runs, `atstep` and `ctstep` are used for the accumulated and current timestep numbers of the outer timestep loop. `citern` gives the current coefficient loop number within the current timestep. Thus, `citern` will cycle between 1 and n for each timestep during a transient run, where n is the number of coefficient loops. `aitern` is equivalent to `citern` for transient runs.

15.2.12.5.3. ANSYS Multi-field Runs

For ANSYS Multi-field runs, `cstagger` and `acplgstep` are also available. `cstagger` gives the current stagger iteration, which will cycle between 1 and n for each coupling step of the run. `acplgstep` gives the accumulated coupling step. This gives the multi-field timestep number or "coupling step" number

for the run, and accumulates across a restarted run. For transient ANSYS Multi-field runs where the CFX timestep is the same as the multi-field timestep, `acplgstep` is equivalent to `atstep`.

15.2.12.6. Expression Names

Your CEL expression name can be any name that does not conflict with the name of a CFX system variable, mathematical function, or an existing CEL expression. The RULES and VARIABLES files provide information on valid options, variables, and dependencies. Both files are located in <CFXROOT>/etc/ and can be viewed in any text editor.

15.2.12.7. Scalar Expressions

A *scalar expression* is a real valued expression using predefined variables, user variables, and literal constants (for example, 1.0). Note that literal constants have to be of the same dimension. Scalar expressions can include the operators + - * / and ^ and several of the mathematical functions found in standard Fortran (for example, sin() and exp()).

An expression's value is a real value and has specified dimensions (except where it is dimensionless - but this is also a valid dimension setting).

For example, if *t* is time and *L* is a length then the result of *L*/*t* has the same dimensions as speed.

The + and - operators are only valid between expressions with the same dimensions and result in an expression of those dimensions.

The * and / operators combine the dimensions of their operands in the usual fashion. *X*^*I*, where *I* is an integer, results in an expression whose dimensions are those of *X* to the power *I*. The trigonometric functions all work in terms of an angle in radians and a dimensionless ratio.

15.2.12.8. Expression Properties

There are three properties of expressions:

- An expression is a simple expression if the only operations are +, -, *, / and there are no functions used in the expression.
- An expression is a constant expression if all the numbers in the expression are explicit (that is, they do not depend on values from the solver).
- An expression is an integer expression if all the numbers in the expression are integers and the result of each function or operation is an integer.

For example, (3+5)/2 is a simple, constant, integer expression. However, 2*(1/2) is not a constant integer expression because the result of 1/2 is 0.5, not an integer. Also 3.*4 is not a constant integer expression because 3 is not an integer. Moreover, 2^3 is not a simple, constant, integer expression because ^ is not in the list (+, -, *, /).

Expressions are evaluated at runtime and in single precision floating point arithmetic.

15.2.12.9. Available and Unavailable Variables

CFX System Variables and user-defined expressions will be available or unavailable depending on the simulation you are performing and the expressions you want to create. In some circumstances, System Variables are logically unavailable; for instance, time (`t`) is not available for steady-state simulations. In others, the availability of a System Variable is not allowed for physical model reasons. For example,

density can be a function of pressure (p), temperature (T) and location (x, y, z), but no other system variables.

Information on how to find dependencies for all parameters is available in the RULES and VARIABLES files. Both files are located in <CFXROOT>/etc/ and can be viewed in any text editor.

The expression definition can depend on any system variable. If, however, that expression depends on a system variable that is unavailable for a particular context, then that expression will also be unavailable.

15.3. Particle Variables Generated by the Solver

This section describes the following types of particle variables that you may have defined in CFX-Pre or that are available for viewing in CFD-Post and exporting to other files. Many variables are relevant only for specific physical models.

- *Particle Track Variables* (p. 210)
- *Particle Field Variables* (p. 212)
- *Particle Boundary Vertex Variables* (p. 217)

Some variables are defined only on the boundaries of the model. When using these variables in CFD-Post, there are a limited number of useful things that you can do with these. For details, see Boundary-Value-Only Variables in the *CFD-Post User's Guide*.

The following information is given for particle variables described in this section:

- **Long Variable Name**: The name that you see in the user interface.
- **Short Variable Name**: The name that must be used in CEL expressions.
- **Units**: The default units for the variable. An empty entry [ ] indicates a dimensionless variable.

  Note

  The entries in the Units columns are SI but could as easily be any other system of units.

- **Type (User Level, Boundary)**

  User Level: This number is useful when using the CFX Export facility. For details, see File Export Utility in the *CFX-Solver Manager User's Guide*. Note that the CFX-Solver may sometimes override the user-level setting depending on the physics of the problem. In these cases, the User Level may be different from that shown in the table below.

  Boundary (B): A **B** in this column indicates that the variable contains only non-zero values on the boundary of the model. See Boundary-Value-Only Variables in the *CFD-Post User's Guide* for more details.

This section does not cover the complete list of variables. For information on obtaining details on all variables, see RULES and VARIABLES Files in the *CFX-Solver Manager User's Guide*.

Note

Variables with names shown in **bold text** are not output to CFD-Post. However, some of these variables can be output to CFD-Post by selecting them from the **Extra Output Variables List** on the **Results** tab of the **Solver** > **Output Control** details view of CFX-Pre.

15.3.1. Particle Track Variables

Particle track variables are particle variables that are defined directly on each track. These variables are defined on the particle positions for which track information is written to the results file. Direct access to the particle track variables outside of CFD-Post is only possible if the raw track file is kept after a particle run.

Particle track variables can only be used in two ways: to color particle tracks in CFD-Post, and to be used as input to Particle User Fortran. Particle track variables can be exported from CFD-Post along the particle tracks.

Note

Particle track variables are not available for use in CEL expressions and general User Fortran, and they also cannot be monitored during a simulation.

For Particle User Fortran, additional track variables can be specified in the argument list for the user routine, which are not available in CFD-Post:

Long Variable Name	Short Variable Name	Units	Description	Availability
<Particle Type>.Mean Particle Diameter	mean particle diameter	[m]	Particle diameter	3 PR
<Particle Type>.Particle Number Rate	particle number rate	[s^-1]	Particle number rate	3 PR
<Particle Type>.Particle Time	pttime	[s]	Simulation time	2 PR
<Particle Type>.Particle Traveling Distance	ptdist	[m]	Distance along the particle track measured from the injection point	2 PR
<Particle Type>.Particle Traveling Time		[s]	Time measured from the time of injection of the particle. For steady-state simulations only, this time is identical to <Particle Type>.Particle Time.	2 PR

Long Variable Name	Short Variable Name	Units	Description	Availability
<Particle Type>.Temperature	T	[K]	Particle temperature	1 PR
<Particle Type>.Total Particle Mass	ptmasst	[kg]	Particle total mass	2 PR
<Particle Type>.Velocity		[m/s]	Particle velocity	1 PR
<Particle Type>.Velocity u <Particle Type>.Velocity v <Particle Type>.Velocity w	u v w	[m/s]	Particle velocity components in x, y, and z-direction	1 PR

Long Variable Name	Short Variable Name	Units	Availability
Particle Eotvos Number	pteo	[ ]	2 PR
Particle Morton Number	ptmo	[ ]	2 PR
Particle Nusselt Number	ptnu	[ ]	2 PR
Particle Ohnesorge Number	pton	[ ]	2 PR
Particle Reynolds Number	ptre	[ ]	2 PR
Particle Weber Number [a]	ptwe	[ ]	2 PR
Particle Slip Velocity	ptslipvel	[m s^-1]	2 PR
Particle Position	ptpos	[m]	2 PR

Long Variable Name	Short Variable Name	Units	Availability
Particle Impact Angle [b]	particle impact angle	[radian]	3 PR

[a]Note: The factors affecting the Weber number are particle diameter, particle slip velocity, fluid density, and surface tension.
[b]Note: The impact angle is measured from the wall.

15.3.2. Droplet Breakup Variable

Long Variable Name	Units	Description
<Particle Type>.Particle Weber Number	[-]	Particle Weber number along track

$$We = \rho_f V_{slip}{}^2 \frac{D_p}{\sigma}$$

where

$\rho_f = \text{fluid density}$

$V_{slip} = \text{slip velocity}$

$D_p = \text{particle diameter}$

$\sigma = \text{surface tension coefficient}$

15.3.3. Multi-component Particle Variable

Long Variable Name	Units	Description
<Particle Type>.<Particle Component>.Mass Fraction		Fraction of mass of a particular particle component

15.3.4. Particle Field Variables

Particle field variables are particle variables that are defined at the vertices of the fluid calculation. In contrast to track variables, these variables can be used in the same way as "standard" Eulerian variables. This means that particle field variables are available for use in CEL expressions and User Fortran, they can be monitored during a simulation, and are available for general post-processing in CFD-Post. Additionally, particle field variables can be used in the same way as particle track variables as input to particle User Fortran and for coloring tracks. When used for coloring tracks, the field variables have to be interpolated onto the tracks, and so this operation will be slower than coloring with a track variable.

The following particle variables are available as field variables:

15.3.4.1. Particle Sources into the Coupled Fluid Phase

For fully-coupled particle simulations involving energy, momentum and mass transfer to the fluid phase, the following variables are written to the results file:

Long Variable Name	Short Variable Name	Units	Availability
Particle Energy Source	ptenysrc	[W m^-3]	2 A, C, M, P, R
Particle Energy Source Coefficient	ptenysrcc	[W m^-3 K^-1]	2 A, C, M, P, R
Particle Momentum Source	ptmomsrc	[kg m^-2 s^-2]	2 A, C, M, P, R
Particle Momentum Source Coefficient	ptmomsrcc	[kg m^-3 s^-1]	2 A, C, M, P, R
Total Particle Mass Source	ptmassrctot	[kg s^-1 m^-3]	2 A, C, M, P, R
Total Particle Mass Source Coefficient	ptmassrcctot	[kg s^-1 m^-3]	2 A, C, M, P, R
For multi-component mass transfer, the following Additional Variables are available [a]:			
Particle Mass Source	ptmassrc	[kg s^-1 m^-3]	2 A, C, M, P, R
Particle Mass Source Coefficient	ptmassrcc	[kg s^-1 m^-3]	2 A, C, M, P, R

[a]The variables for multi-component take the following form: <Particle Type>.<Particle Component>.<Variable Name>

Particle source terms are accumulated along the path of a particle through a control volume and stored at the corresponding vertex. A smoothing procedure can be applied to the particle source terms, which may help with convergence or grid independence. For details, see Particle Source Smoothing in the *CFX-Solver Modeling Guide*.

15.3.4.2. Particle Radiation Variables

Long Variable Name	Short Variable Name	Units	Availability
Particle Radiative Emission	ptremiss	[W m^-3]	2 A, C, M, P, R

Long Variable Name	Short Variable Name	Units	Availability
Particle Absorption Coefficient	ptabscoef	[m^-1]	2 A, C, M, P, R

Particles can also interact with the radiation field and either emit or absorb radiation.

15.3.4.3. Particle Vertex Variables

By default, particle vertex variables are not written to the results file, except for the `Averaged Volume Fraction`. The other vertex variables can be written to the results file if they are selected from the **Extra Output Variables List** in the **Output Control** section of CFX-Pre or if they are used in a monitor point, CEL expression or in (Particle) User Fortran.

The following particle variables are available:

Long Variable Name	Short Variable Name	Units	Availability
Averaged Velocity	averaged vel	[m s^-1]	1 A, C, M, P, PR, R
Averaged Volume Fraction	vfpt	[ ]	1 A, C, M, P, PR, R
Averaged Temperature	averaged temperature	[K]	1 A, C, M, P, PR, R
Averaged Mass Fraction [a]	averaged mf	[ ]	1 A, C, M, P, PR, R
Averaged Particle Time	averaged pt-time	[s]	2 A, C, M, P, PR, R
Averaged Mean Particle Diameter (D43)	averaged mean particle diameter	[m]	2 A, C, M, P, PR, R
Averaged Arithmetic Mean Particle Diameter (D10)	averaged arithmetic mean particle diameter	[m]	2 A, C, M, P, PR, R
Averaged Surface Mean Particle Diameter (D20)	averaged surface mean	[m]	2

Long Variable Name	Short Variable Name	Units	Availability
	particle diameter		A, C, M, P, PR, R
Averaged Volume Mean Particle Diameter (D30)	averaged volume mean particle diameter	[m]	2 A, C, M, P, PR, R
Averaged Sauter Mean Particle Diameter (D32)	averaged sauter mean particle diameter	[m]	2 A, C, M, P, PR, R
Averaged Mass Mean Particle Diameter (D43)	averaged mass mean particle diameter	[m]	2 A, C, M, P, PR, R
Averaged Particle Number Rate	averaged particle number rate	[s^-1]	2 A, C, M, P, PR, R
For simulations with the particle wall film model activated, the following additional vertex variables are available:			
Averaged Volume Fraction Wall	vfptw	[ ]	1 A, C, M, P, PR, R
Averaged Film Temperature	averaged film temperature	[K]	1 A, C, M, P, PR, R

[a]This variable takes the following form: <Particle Type>.<Particle Component>.<Variable Name>

The following are the formulae for particle vertex fields' size distributions:

Arithmetic Mean Diameter	$d_{10} = \frac{\sum n_i d_i}{\sum n_i}$
Surface Mean Diameter	$d_{20} = \sqrt{\frac{\sum n_i d_i^2}{\sum n_i}}$
Volume Mean Diameter	$d_{30} = \sqrt[3]{\frac{\sum n_i d_i^3}{\sum n_i}}$

Sauter Mean Diameter	$d_{32} = \frac{\sum n_i d_i^3}{\sum n_i d_i^2}$
Mass Mean Diameter	$d_{32} = \frac{\sum n_i d_i^4}{\sum n_i d_i^3}$

15.3.4.3.1. Variable Calculations

Particle vertex variables are calculated using the following averaging procedure:

$$\overline{\Phi}_P = \frac{\sum \left(\Delta t \; m_P \; \dot{N}_P \; \Phi_P \right)}{\sum \left(\Delta t \; m_P \; \dot{N}_P \right)} \qquad \textbf{(15–1)}$$

With:

- Σ: Sum over all particles and time steps in a control volume
- $\Delta \; t$: Particle integration time step
- $\dot{N}_P$: Particle number rate
- m_P: Particle mass
- Φ: Particle quantity

Slightly different averaging procedures apply to particle temperature and particle mass fractions:

Averaged Particle Temperature

$$\overline{\Phi}_P = \frac{\sum \left(\Delta t \; m_P \; \dot{N}_P \; c_{P,P} \; T_P \right)}{\sum \left(\Delta t \; m_P \; \dot{N}_P \; c_{P,P} \right)} \qquad \textbf{(15–2)}$$

With:

Averaged Mass Fraction

- $c_{P,P}$: Particle specific heat capacity
- T_P: Particle temperature

$$\overline{\Phi}_P = \frac{\sum \left(\Delta t \; m_{c,P} \; \dot{N}_P \right)}{\sum \left(\Delta t \; m_P \; \dot{N}_P \right)} \qquad \textbf{(15–3)}$$

With:

- $m_{c,P}$: Mass of species c in the particle

Due to the discrete nature of particles, vertex variables may show an unsmooth spatial distribution, which may lead to robustness problems. To reduce possible problems a smoothing option is available. For details, see Vertex Variable Smoothing in the *CFX-Solver Modeling Guide*.

15.3.4.4. Particle Boundary Vertex Variables

Particle-boundary vertex variables are particle variables that are defined on the vertices of domain boundaries. They are normalized with the face area of the corresponding boundary control volume.

You can use these variables to color boundaries and to compute average or integrated values of the corresponding particle quantities.

You cannot use these variables in CEL expressions or User Fortran, and you cannot monitor them during a simulation.

Long Variable Name	Units	Availability
Available at inlet, outlet, openings and interfaces:		
Mass Flow Density	[kg m^-2 s^-1]	2 **B**, R
Momentum Flow Density	[kg m^-1 s^-2]	2 **B**, R
Energy Flow Density	[kg s^-3]	2 **B**, R
Available at walls only:		
Wall Stress	[kg m^-1 s^-2]	2 **B**, R
Wall Mass Flow Density	[kg m^-2 s^-1]	2 **B**, R
Erosion Rate Density	[kg m^-2 s^-1]	2 **B**, R
Available in transient runs:		
Time Integrated Mass Flow Density	[kg m^-2]	2 **B**, R
Time Integrated Momentum Flow Density	[kg m^-1 s^-1]	
Time Integrated Energy Flow Density	[kg s^-2]	2 **B**, R
Time Integrated Wall Mass Flow Density	[kg m^-2]	2 **B**, R
Time Integrated Erosion Rate Density	[kg m^-2]	2

Long Variable Name	Units	Availability
		B, R

15.3.4.5. Particle RMS Variables

For some applications, it may be necessary to not only provide the mean values of particle quantities, but also their standard deviation in the form of particle RMS variables. Similar to particle vertex variables, these variables are also defined at the vertices of the fluid calculation. Particle RMS variables are available for use in CEL expressions and User Fortran; they can be monitored during a simulation, and are available for general post-processing in CFD-Post. Additionally, particle RMS variables can be used in the same way as particle track variables as input to particle User Fortran and for coloring tracks.

By default, particle RMS variables are not written to the results file; unless, they have been explicitly requested by the user (selected from the **Extra Output Variables List** in the **Output Control** section of CFX-Pre, usage in a CEL expression or in User Fortran) or if the stochastic particle collision model is used in a simulation.

The following particle variables are available as field variables, particularly useful for simulations that use the stochastic particle collision model:

Long Variable Name	Short Variable Name	Units	Availability
RMS Velocity	rms velocity	[m s^-1]	1 A, C, M, P, PR, R
RMS Temperature	rms temperature	[K]	1 A, C, M, P, PR, R
RMS Mean Particle Diameter	rms mean particle diameter	[m]	3 A, C, M, P, PR, R
RMS Particle Number Rate	rms particle number rate	[s^-1]	3 A, C, M, P, PR, R

15.3.4.5.1. Variable Calculations

Particle RMS variables are calculated using the following procedure:

$$\Phi = \overline{\Phi} + \Phi''$$
$$\Phi_{rms} = \sqrt{\overline{\Phi''^2}} = \sqrt{\overline{(\Phi - \overline{\Phi})^2}} = \sqrt{\overline{\Phi^2} - \overline{\Phi}^2} \qquad \textbf{(15–4)}$$

With:

- Φ: Instantaneous particle quantity

- $\overline{\Phi}$: Average particle quantity
- Φ'': Fluctuating particle quantity
- $\overline{\Phi^2}$: Average of square of particle quantity
- $\overline{\Phi}^2$: Square of average of particle quantity

A smoothing option, as available for particle vertex variables, is available for particle RMS variables. For details, see Vertex Variable Smoothing in the *CFX-Solver Modeling Guide*.

15.4. Miscellaneous Variables

Variable names in **bold** are not output to CFD-Post.

In the **Availability** column:

- A number represents the user level (1 indicates that the variable appears in default lists, 2 and 3 indicate that the variable appears in extended lists that you see when you click ...)
- A indicates the variable is available for mesh adaption
- C indicates the variable is available in CEL
- DT indicates the variable is available for data transfer to ANSYS
- M indicates the variable is available for monitoring
- P indicates the variable is available for particle user routine argument lists
- PR indicates the variable is available for particle results
- R indicates the variable is available to be output to the results, transient results, and backup files
- TS indicates the variable is available for transient statistics

Long Variable Name	Short Variable Name	Units	Availability	Definition
Aspect Ratio	aspect ratio	[ ]	2 C, M, R, TS	
Autoignition	autoignition	[ ]	1 A, C, M, R, TS	
Boundary Scale	bnd scale	[ ]	3 C, M, R, TS	Similar to wall scale, this variable is used for controlling mesh stiffness near boundaries for moving mesh problems.
Burnt Absolute Temperature	burnt Tabs	[K]	2 A, C, M, R, TS	

Long Variable Name	Short Variable Name	Units	Availability	Definition
Burnt Density	burnt density	[kg m^-3]	2 A, C, M, R, TS	
Clipped Pressure	pclip	[Pa]	1 M, R, TS	Negative absolute values clipped for cavitation
Conservative Size Fraction	sfc	[ ]	2 A, C, M, R, TS	
Courant Number	courant	[ ]	2 C, M, R, TS	
Cumulative Size Fraction	csf	[ ]	2 A, C, M, R, TS	
Current Density	jcur		1 C, M, R, TS	
Dynamic Diffusivity	diffdyn		2 C, M, P, R, TS	
Electric Field	elec		1 C, M, R, TS	
Electric Potential	epot		1 C, M, R, TS	
Electrical Conductivity	conelec		3 C, M, R, TS	
Electrical Permittivity	permelec		3 C, M, R, TS	
Electromagnetic Force Density	bfemag		3 R	
Equivalence Ratio	equivratio	[ ]	2 A, C, M, R, TS	
External Magnetic Induction	bmagext	[ ]	1 M, R, TS	External magnetic induction field specified by the user.
First Blending Function for	sstbf1	[ ]	3 C, M, R, TS	

Long Variable Name	Short Variable Name	Units	Availability	Definition
BSL and SST model				
Second Blending Function for SST model	sstbf2	[ ]	3 C, M, R, TS	
Flame Surface Density	fsd	[m^-1]	1 A, C, M, R, TS	Combustion with flame surface density models.
Specific Flame Surface Density	spfsd		2 A, C, M, R, TS	Combustion with flame surface density models.
Frequency	freq		3 C	
Fuel Tracer	trfuel	[ ]	1 A, C, M, R, TS	Residual material model or exhaust gas recirculation (EGR)
Granular Temperature	grantemp	[m^2 s^-2]	1 A, C, M, R, TS	
Group I Index	groupi	[ ]	2 C	
Group J Index	groupj	[ ]	2 C	
Group I Diameter	diami		2 C	
Group J Diameter	diamj		2 C	
Group I Mass	massi		2 C	
Group J Mass	massj		2 C	
Group I Lower Mass	massi lower		2 C	
Group J Lower Mass	massj lower		2	

Long Variable Name	Short Variable Name	Units	Availability	Definition
			C	
Group I Upper Mass	massi upper		2 C	
Group J Upper Mass	massj upper		2 C	
Ignition Delay Elapsed Fraction	ignfrc	[ ]	2 A, C, M, R, TS	
Ignition Delay Time	tigndelay	[s]	2 A, C, M, R, TS	
Particle Integration Timestep	particle integration timestep	[s]	3 P	
Isentropic Compressibility	compisoS	[m s^2 kg^-1]	2 C, M, R	$\left(\frac{1}{\rho}\right)\ \left(\frac{\partial \rho}{\partial p}\right)_s$
Isentropic Compression Efficiency	icompeff	[ ]	2 C, M, R, TS	
Isentropic Expansion Efficiency	iexpeff	[ ]	2 C, M, R, TS	
Isentropic Total Enthalpy	htotisen		2 C, M, R, TS	
Isentropic Static Enthalpy	enthisen		2 C, M, R, TS	
Isobaric Compressibility	compisoP	[K^-1]	2 C, M, R	$-\frac{1}{\rho}\frac{\partial \rho}{\partial T}\Big\vert_p$
Isothermal Compressibility	compisoT	[m s^2 kg^-1]	2 C, M, R	$\frac{1}{\rho}\frac{\partial \rho}{\partial p}\Big\vert_T$
LES Dynamic Model Coefficient	dynmc	[ ]	1 A, C, M, P, R, TS	
Laminar Burning Velocity	velburnlam	[m s^-1]	2 A, C, R, TS	

Long Variable Name	Short Variable Name	Units	Availability	Definition
Lighthill Stress	lighthill stress tensor		2 A, C, M, R, TS	
Magnetic Induction	bmag		1 C, M, R, TS	
Magnetic Field	hmag		2 C, M, R, TS	
Magnetic Vector Potential	bpot		1 C, M, R, TS	
Magnetic Permeability	permmag		3 C, M, R, TS	
External Magnetic Induction	bmagext		1 C, M, R, TS	
Mass Flux	mfflux		2 R	
Mesh Diffusivity	diffmesh	[m^2 s^-1]	2 C, M, R, TS	
Normal Area	normarea	[ ]	2 C	Normal area vectors.
Total Force Density	forcetden		3 DT	
Total Pressure in Rel Frame	ptotrel		2 A, C, M, P, R, TS	Based on relative frame total enthalpy.
Turbulent Burning Velocity	velburnturb	[m s^-1]	2 A, C, R, TS	
Mesh Velocity	meshvel		1 C, M, R, TS	
Mixture Fraction Scalar Dissipation Rate	mixsclds	[s^-1]	3 A, C, M, R, TS	
Molar Reaction Rate	reacrate		2 C, R, TS	

Long Variable Name	Short Variable Name	Units	Availability	Definition
Nonclipped Absolute Pressure	pabsnc		3 A, C, M, R, TS	Nonclipped absolute pressure for cavitation source. This is written to the `.res` file for all cases that have cavitation.
Nonclipped Density	densitync	[kg m^-3]	2 C	Nonclipped density for cavitation source
Normal Vector	normal	[ ]	2 C	
Orthogonality Factor Minimum	orthfactmin	[ ]	2 C, M, R, TS	
Orthogonality Factor	orthfact	[ ]	2 C, M, R, TS	
Orthogonality Angle Minimum	orthanglemin		2 C, M, R, TS	
Orthogonality Angle	orthangle		2 C, M, R, TS	
Particle Laplace Number	ptla	[ ]	2 P	
Particle Turbulent Stokes Number	ptstt	[ ]	2 P	
Polytropic Compression Efficiency	pcompeff	[ ]	2 C, M, R, TS	
Polytropic Expansion Efficiency	pexpeff	[ ]	2 C, M, R, TS	
Polytropic Total Enthalpy	htotpoly		2 C, M, R, TS	
Polytropic Static Enthalpy	enthpoly		2 C, M, R, TS	

Long Variable Name	Short Variable Name	Units	Availability	Definition
Reaction Progress	reacprog	[ ]	1 A, C, M, R, TS	For premixed or partially premixed combustion.
Weighted Reaction Progress	wreacprog	[ ]	2 A, C, M, R, TS	For premixed or partially premixed combustion.
Weighted Reaction Progress Source	wreacprogsrc		3 A, C, R, TS	For premixed or partially premixed combustion.
Residual Products Mass Fraction	mfresid	[ ]	1 A, C, M, R, TS	Residual material model or exhaust gas recirculation (EGR)
Residual Products Molar Fraction	molfresid	[ ]	2 A, C, M, R, TS	Residual material model or exhaust gas recirculation (EGR)
Restitution Coefficient	restitution coefficient	[ ]	3 C, M, R, TS	
Rotation Velocity	rotvel		2 C, R, TS	
Rotational Energy	rotenergy		2 C, R, TS	
Shear Velocity	ustar		2 C	
Size Fraction	sf	[ ]	1 A, C, M, R, TS	
Solid Bulk Viscosity	solid bulk viscosity	[kg m^-1 s^-1]	3 C, M, R, TS	
Solid Pressure	solid pressure	[Pa]	3 A, C, M, R, TS	
Solid Pressure Gradient	solid pressure gradient	[ ]	3 C, M, R, TS	
Solid Shear Viscosity	solid shear viscosity	[kg m^-1 s^-1]	3 C, M, R, TS	

Long Variable Name	Short Variable Name	Units	Availability	Definition
Static Entropy	entropy		3 A, C, M, P, R, TS	
Temperature Variance	Tvar		1 A, C, M, R, TS	
Time This Run	trun		2 C	
Total Boundary Displacement	bnddisptot		1 C, DT, M, R, TS	
Total Density	dentot	[kg m^-3]	2 A, C, M, R	Total Density is the density evaluated at the Total Temperature and Total Pressure.
Total Density in Stn Frame	dentotstn	[kg m^-3]	2 A, C, M, R	
Total Density in Rel Frame	dentotrel	[kg m^-3]	2 A, C, M, R	
Total Force	forcet		3 DT	
Unburnt Absolute Temperature	unburnt Tabs	[K]	2 A, C, M, R, TS	
Unburnt Density	unburnt density	[kg m^-3]	2 A, C, M, R, TS	
Unburnt Thermal Conductivity	unburnt cond	[W m^-1 K^-1]	2 A, C, M, R, TS	
Unburnt Specific Heat Capacity at Constant Pressure	unburnt Cp	[J kg^-1 K^-1]	2 A, C, M, R, TS	
Volume Porosity	volpor	[ ]	2 C, M, R, TS	
Volume of Finite Volumes	volcvol		3 C, R, TS	

Long Variable Name	Short Variable Name	Units	Availability	Definition
Vorticity	vorticity		2 A, C, M, R, TS	Note that Vorticity is the same as Velocity.Curl.
Vorticity in Stn Frame	vortstn		2 A, C, M, R, TS	
Wall External Heat Transfer Coefficient	htco		2 R, TS	
Wall Adjacent Temperature	tnw	[K]	2 C, DT, R, TS	
Wall Distance	wall distance	[m]	2 A, C, M, P, R, TS	
Wall External Temperature	tnwo	[K]	2 DT, R, TS	User-specified external wall temperature for heat transfer coefficient boundary conditions.
Wall Film Thickness	film thickness	[m]	2 C, R	
Wall Heat Transfer Coefficient	htc		2 C, R, TS	
Wall Heat Flow	QwallFlow		3 C, DT, R, TS	
Wall Normal Velocity	nwallvel		2 C, R, TS	
Wall Scale	wall scale		3 R, M, TS	
Wavelength in Vacuum	wavelo		3 C	
Wavenumber in Vacuum	waveno		3 C	
Normalized Droplet Number	spdropn	[m^-3]	2 C, M, R, TS	

Long Variable Name	Short Variable Name	Units	Availability	Definition
Droplet Number	spdrop		1 C, M, R, TS	
Dynamic Bulk Viscosity	dynamic bulk viscosity		1 A, C, M, R, TS	
Total MUSIG Volume Fraction	vft	[ ]	2 A, C, M, R, TS	
Smoothed Volume Fraction	vfs	[ ]	2 A, C, M, R, TS	
Temperature Superheating	Tsuperheat		3 C	Temperature above saturation
Temperature Subcooling	Tsubcool		3 C	Temperature below saturation

Chapter 16: Power Syntax in ANSYS CFX

Programming constructs can be used within CCL for advanced usage. Rather than invent a new language, CCL takes advantage of the full range of capabilities and resources from an existing programming language, Perl. Perl statements can be embedded in between lines of simple syntax, providing capabilities such as loops, logic, and much, much more with any CCL input file.

Lines of Power Syntax are identified in a CCL file by an exclamation mark (!) at the start of each line. In between Perl lines, simple syntax lines may refer to Perl variables and lists.

A wide range of additional functionality is made available to expert users with the use of Power Syntax including:

- Loops
- Logic and control structures
- Lists and arrays
- Subroutines with argument handling (useful for defining commonly re-used plots and procedures)
- Basic I/O processing
- System functions
- Many other procedures (Object programming, World Wide Web access, simple embedded graphical user interfaces).

Any of the above may be included in a CCL input file or CFD-Post Session file.

Important

You should be wary when entering certain expressions because Power Syntax uses Perl mathematical operators. For example, in CEL, 2^2 is represented as 2^2, but in Perl, it would be written 2**2. If you are unsure about the validity of an operator, you should check a Perl reference guide.

There are many good reference books on Perl. Two examples are *Learning Perl* (ISBN 1-56592-042-2) and *Programming Perl* (ISBN 1-56592-149-6) from the O'Reilly series.

This chapter describes:

16.1. Examples of Power Syntax

The following are some examples in which the versatility of power syntax is demonstrated. They become steadily more complex in the later examples.

Some additional, more complex, examples of Power Syntax subroutines can be found by viewing the session files used for the **Macro Calculator**. These are located in `CFX/etc/`. You can execute these subroutines from the **Command Editor** dialog box the same as calling any other Power Syntax subroutine. The required argument format is:

```
!cpPolar(<"BoundaryList">, <"SliceNormalAxis">,
   <"SlicePosition">, <"PlotAxis">, <"InletLocation">,
   <"ReferencePressure">)
!compressorPerform(<"InletLocation">, <"OutletLocation">,
   <"BladeLocation">, <"MachineAxis">, <"RotationalSpeed">,
   <"TipRadius">, <"NumBlades">, <"FluidGamma">)
```

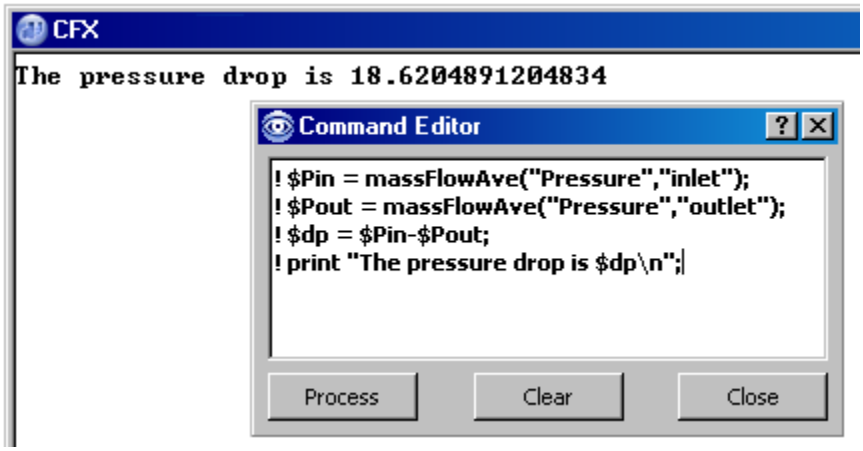
These subroutines are loaded when CFD-Post is launched, so you do not need to execute the session files before using the functions.

Additional information on these macro functions is available in Gas Compressor Performance Macro and Cp Polar Plot Macro.

All arguments passed to subroutines should be enclosed in quotations, for example `Plane 1` must be passed as "`Plane 1`" and `Eddy Viscosity` should be entered as "`Eddy Viscosity`". Any legal CFX Command Language characters that are illegal in Perl need to be enclosed in quotation marks.

16.1.1. Example 1: Print the Value of the Pressure Drop Through a Pipe

```
! $Pin = massFlowAve("Pressure","inlet");
! $Pout = massFlowAve("Pressure","outlet");
! $dp = $Pin-$Pout;
! print "The pressure drop is $dp\n";
```



Note

Function-specific Perl subroutines do not allow phase-specific evaluations; that is, you can get only bulk results (such as mass flow for all phases). A workaround is to use "evaluate" subroutine, which evaluates any CEL expression.

For example instead of

```
! $val = massFlow("Inlet", "Water")   # Does NOT work
```

you need to use:

```
! ($val, $units) = evaluate( "Water.massFlow()\@Inlet");
```

16.1.2. Example 2: Using a for Loop

This example demonstrates using Power Syntax that wraps a `for` loop around some CCL Object definitions to repetitively change the visibility on the outer boundaries.

```
# Make the outer boundaries gradually transparent in
# the specified number of steps.
!$numsteps = 10;
!for ($i=0; $i < $numsteps; $i++) {
   ! $trans = ($i+1)/$numsteps;
   BOUNDARY:in
      Visibility = 1
      Transparency = $trans
   END
   BOUNDARY:out
      Visibility = 1
      Transparency = $trans
   END
   BOUNDARY:Default
      Visibility = 1
      Transparency = $trans
   END
!}
```

The first line of Power Syntax simply defines a scalar variable called `numsteps`. Scalar variables (that is, simple single-valued variables) begin with a `$` symbol in Perl. The next line defines a `for` loop that increments the variable `i` up to `numsteps`. Next, you determine the fraction you are along in the loop and assign it to the variable `trans`. The object definitions then use `trans` to set their transparency and then repeat. Note how Perl variables can be directly embedded into the object definitions. The final line of Power Syntax (`!}`) closes the `for` loop.

Note

Function-specific Perl subroutines do not allow phase-specific evaluations; that is, you can get only bulk results (such as mass flow for all phases). A workaround is to use "evaluate" subroutine, which evaluates any CEL expression.

For example instead of

```
! $val = massFlow("Inlet", "Water")   # Does NOT work
```

you need to use:

```
! ($val, $units) = evaluate( "Water.massFlow()\@Inlet");
```

16.1.3. Example 3: Creating a Simple Subroutine

The following example defines a simple subroutine to make two planes at specified locations. The subroutine will be used in the next example.

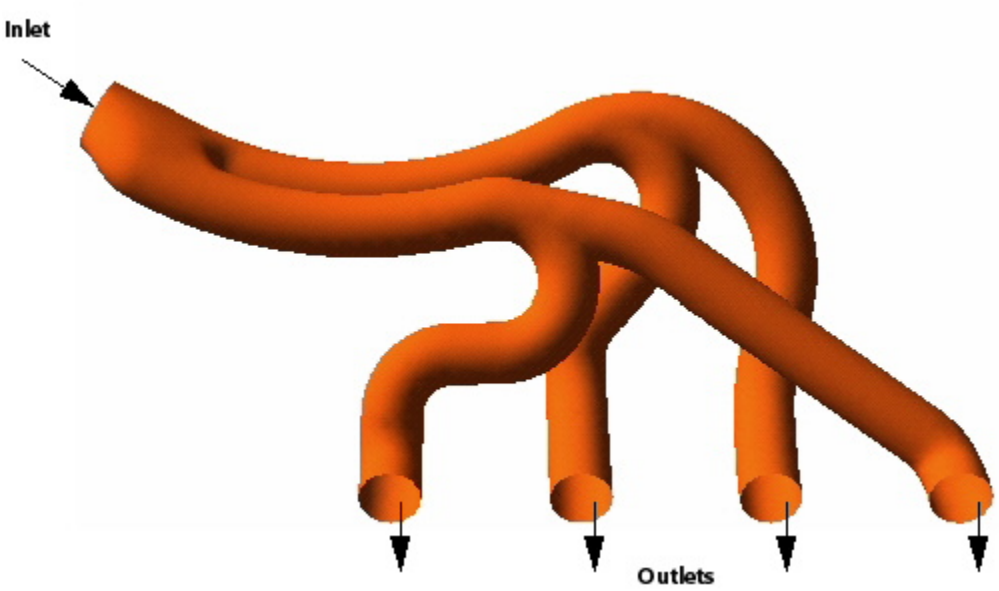
```
!sub makePlanes {
   PLANE:plane1
      Option = Point and Normal
      Point = 0.09,0,-0.03
      Normal = 1,0,0
      Draw Lines = On
      Line Color = 1,0,0
      Color Mode = Variable
      Color Variable = Pressure
      Range = Local
   END
   PLANE:plane2
```

```
      Option = Point and Normal
      Point = 0.08,-0.038,-0.0474
      Normal = 1,0,0
      Draw Faces = Off
      Draw Lines = On
      Line Color = 0,1,0
   END
!}
```

Although this subroutine is designed for use with the next example, you can execute it on its own by typing `!makePlanes();` in the **Command Editor** dialog box.

16.1.4. Example 4: Creating a Complex Quantitative Subroutine

This example is a complex quantitative subroutine that takes slices through the manifold geometry, as shown below, compares the mass flow through the two sides of the initial branch, and computes the pressure drop through to the four exit locations.




```
! sub manifoldCalcs{
# call the previously defined subroutine (Example 3) make the
# upstream and downstream cutting planes
! makePlanes();
#
# Bound the two planes so they each just cut one side of the branch.
PLANE:plane1
Plane Bound = Circular
Bound Radius = 0.025
END
PLANE:plane2
Plane Bound = Circular
Bound Radius = 0.025
END
# Calculate mass flow through each using the predefined
# 'evaluate' Power Syntax subroutine and output the results
! ($mass1, $mfunits) = evaluate( "massFlow()\@plane1" );
! ($mass2) = evaluate( "massFlow()\@plane2" );
! $sum = $mass1+$mass2;
! print "Mass flow through branch 1 = $mass1 [$mfunits]\n";
! print "Mass flow through branch 2 = $mass2 [$mfunits]\n";
! print "Total = $sum [$mfunits]\n";
# Now calculate pressure drops and mass flows through the exits
# calculate the average pressure at the inlet
!($Pin, $punits) = evaluate( "massFlowAve(Pressure)\@in1" );
```

```
# Set-up an array that holds the approximate X location of each
# of the 4 exits. We then loop over the array to move the outlet
# plane and re-do the pressure drop calculation at each exit.
! @Xlocs = (0.15,0.25,0.35,0.45);
! $sum = 0;
! for ($i=0;$i<4;$i++) {
PLANE:outlet
   Option = Point and Normal
   Normal = 0,-1,-1
   Point = $Xlocs[$i],-0.06,-0.2
   Plane Bound = Circular
   Bound Radius = 0.05
END
! ($Pout, $punits) =  evaluate( "massFlowAve(Pressure)\@outlet" );
! ($massFl) = evaluate( "massFlow()\@outlet" );
! $sum += $massFl;
! $Dp = $Pin-$Pout;
! $ii = $i+1;
! print "At outlet \#$ii: Dp=$Dp [$punits], Mass Flow=$massFl [$mfunits]\n";
! } # end loop
! print "Total Mass Flow = $sum  [$mfunits]\n";
!} # end subroutine
```

After processing these commands to define the subroutine, you can execute it, in the same way as any other subroutine, by typing `!manifoldCalcs();` in the **Command Editor** dialog box.

16.2. Predefined Power Syntax Subroutines

CFD-Post provides predefined subroutines that add Power Syntax functionality. You can view a list of these subroutines by entering `!showSubs();` in the **Command Editor** dialog box. The list is printed to the console window. The list shows all currently loaded subroutines, so it will include any custom subroutines that you have processed in the **Command Editor** dialog box.

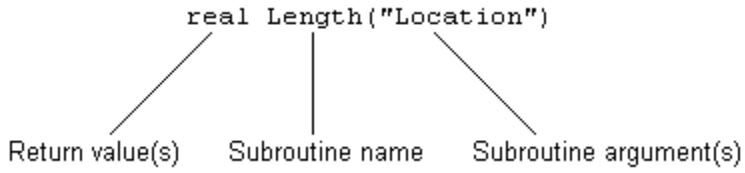
These subroutines provide access to the quantitative functionality of CFD-Post. Most of these routines provide results in a single return value. For example, if the Perl variable `$verbose = 1`, then the result is also printed to the screen. Information on the calculations performed by the subroutines is available. For details, see Function Selection.

The following sections describe these predefined subroutines:

- *Power Syntax Subroutine Descriptions* (p. 233)
- *Power Syntax Usage* (p. 234)
- *Power Syntax Subroutines* (p. 234)

16.2.1. Power Syntax Subroutine Descriptions

In the next section, each subroutine will appear in the following format:



Each of the subroutines contains an argument list (in brackets, separated by commas). If any argument contains more than one word (for example, `Plane 1`), it must be within quotes. You should enclose all arguments within quotes to avoid making possible syntax errors.

Each subroutine is preceded by its return value(s). For example:

```
real, string evaluate("Expression", "Locator")
```

will return two values, a real number and a string.

The return values will always be in the solution units of the CFX-Solver results file, even if you have changed the display units in the **Edit** menu. This means that if you have a plot of temperature in degrees C on `Plane 1`, the area averaged value of temperature on `Plane 1` returned by the `areaAve` command will still be in degrees K.

16.2.2. Power Syntax Usage

All lines of power syntax must have an exclamation mark as the first character so that they are not treated as CCL statements. The statements must also end with a semicolon. The following is an example:

```
! $lengthVal = Length("Plane 1");
! print $lengthVal;
```

Some subroutines return more than one value. To store return values for a subroutine that returns two variables (such as the `evaluate` function), you could use the following:

```
! ($value, $units) = evaluate("Expression 1");
! print "The value of Expression 1 is $value, and the units are $units";
```

16.2.3. Power Syntax Subroutines

16.2.3.1. area(Location, Axis)

```
real area("Location", "Axis")
```

Returns the area of a 2D locator. For details, see *area* (p. 161).

16.2.3.2. areaAve(Variable, Location, Axis)

```
real areaAve("Variable", "Location", "Axis")
```

Returns the area-weighted average of the variable at a 2D locator. For details, see *areaAve* (p. 162).

16.2.3.3. areaInt(Variable, Location, Axis)

```
real areaInt("Variable", "Location", "Axis")
```

Returns the result of the variable integrated over the 2D location. For details, see *areaInt* (p. 163).

16.2.3.4. ave(Variable, Location)

```
real ave("Variable", "Location")
```

Returns the arithmetic average of the variable at a location. For details, see *ave* (p. 164).

16.2.3.5. calcTurboVariables()

```
void calcTurboVariables()
```

Calculates all 'extra' turbo variables. (Works only in turbo mode.)

16.2.3.6. calculate(function,...)

```
real calculate(function,...)
```

Evaluates the named function with the supplied argument list, and returns the float result. The function name is a required argument, which can be followed by a variable length list of arguments.

16.2.3.7. calculateUnits(function,...)

```
string,string calculateUnits(function,...)
```

Evaluates the named function with the supplied argument list, and returns the value and units.

16.2.3.8. collectTurboInfo()

This is an internal subroutine that is used only to initialize report templates.

16.2.3.9. comfortFactors()

This is an internal subroutine that is used only to initialize report templates.

16.2.3.10. compressorPerform(Location, Location, Location, Var, Args)

This is a special macro; for details, see Gas Compressor Performance Macro. For example:

```
compressorPerform("Inlet", "Outlet", "Blade", "X", 600, 0.03, 10, 1.2)
```

16.2.3.11. compressorPerformTurbo()

This is an internal subroutine that is used only to initialize report templates.

16.2.3.12. copyFile(FromPath, ToPath)

```
void copyFile("FromPath", "ToPath")
```

A utility function for copying files.

16.2.3.13. count(Location)

```
real count("Location")
```

Returns the number of nodes on the location. For details, see *count* (p. 164).

16.2.3.14. countTrue(Expression, Location)

```
real countTrue("Expression", "Location")
```

Returns the number of mesh nodes on the specified region that evaluate to "true", where true means greater than or equal to 0.5. `"Expression"` should contain one of the logical operators =, >, <, <=, or >=. The countTrue function is valid for 1D, 2D, and 3D locations. For details, see *countTrue* (p. 165).

16.2.3.15. cpPolar(Location, Var, Arg, Var, Location, Arg)

This is a special macro; for details, see Cp Polar Plot Macro. For example:

```
cpPolar("Plane 1", "Y", 0.3, "X", "Inlet", 10000)
```

16.2.3.16. evaluate(Expression)

```
real,string evaluate("Expression")
```

Returns the value of the expression and the units. Only one expression can be evaluated each time the subroutine is executed. The main advantage of using `evaluate` is that it takes any CEL expression. This means that you do not have to learn any other quantitative power syntax routines described in this section. Also, evaluate will return the result units in addition to the value.

An example is:

```
evaluate("areaAve(Velocity v)\@Location 1")
```

In this case, another subroutine is evaluated. The `evaluate` command takes an any expression as the argument, or more precisely, any expression that resolves to a quantity. This means that you cannot use:

```
"2*Pressure"
```

but you can use:

```
"2*minVal(Pressure)\@locator 1"
```

or

```
"100 [m]"
```

This is simply an alternative way of typing:

```
! $myVal = 2 * minVal("Pressure", "Location");
```

The reason that the @ is escaped calling `evaluate()` is to avoid Perl treating it as a special character.

16.2.3.17. evaluateInPreferred(Expression)

```
real,string evaluateInPreferred("Expression")
```

Returns the value of the expression in your *preferred units*. Preferred units are the units of the data that CFD-Post uses when information is displayed to you and are the default units when you enter information (as contrasted with units of the data that are stored in results files). Use the **Edit** > **Options** > **Common** > **Units** dialog box to set your preferred units.

16.2.3.18. exprExists(Expression)

```
bool exprExists("Expression")
```

Returns true if an expression with this name exists; false otherwise.

16.2.3.19. fanNoiseDefault()

This is an internal subroutine that is used only to initialize report templates.

16.2.3.20. fanNoise()

This is an internal subroutine that is used only to initialize report templates.

16.2.3.21. force(Location, Axis)

```
real force("Location", "Axis")
```

Returns the `force` on a 2D locator. For details, see *force* (p. 165).

16.2.3.22. forceNorm(Location, Axis)

```
real forceNorm("Location", "Axis")
```

Returns the per unit width force on a line in the direction of the specified axis. It is available only for a polyline created by intersecting a locator on a boundary. For details, see *forceNorm* (p. 167).

16.2.3.23. getBladeForceExpr()

This is an internal subroutine that is used only to initialize report templates.

16.2.3.24. getBladeTorqueExpr()

This is an internal subroutine that is used only to initialize report templates.

16.2.3.25. getCCLState()

This is an internal debugging call.

16.2.3.26. getChildrenByCategory(Category)

```
string getChildrenByCategory("Category")
```

Returns the children of an object that belong to the specified category in a comma-separated list. Each object type (for example, a PLANE) can have multiple categories associated with it such as "geometry", "surface", and so on). Categories are specified in *<CFXROOT>*`/etc/CFXPostRules.ccl`.

For example, to get a comma-separated list of all surfaces in a state at the top level (that is, not sub-objects of other objects):

```
! $surfaces = getChildrenByCategory("/", "surface" );
```

Use '`split ","`' to convert the string into an array of strings.

16.2.3.27. getChildren(Object Name, Child Type)

```
string getChildren("Object Name", "Child Type")
```

Returns the children of an object in a comma-separated list. If `Child Type` is not an empty string, this subroutine return only children of the specified type.

16.2.3.28. getExprOnLocators()

This is an internal subroutine that is used only to initialize report templates.

16.2.3.29. getExprString(Expression)

```
string getExprString("Expression")
```

Returns the value and the units of the expression in the form “value units”. For example: “100 m”.

16.2.3.30. getExprVal(Expression)

```
real getExprVal("Expression")
```

Returns only the "value" portion of the expression (units are not included).

16.2.3.31. getObjectName(Object Path)

```
string getObjectName("Object Path")
```

Extracts the name of an object from its full path. For example:

```
!string = getObjectName("/USER SURFACE:User Surface 1")
```

returns "User Surface 1". This is the form needed for evaluating a CEL expression.

16.2.3.32. getParameterInfo(Object Name, Parameter Name, Info Type)

```
string getParameterInfo("Object Name", "Parameter Name", "Info Type")
```

Returns the requested information for a parameter of an object. `Object Name` returns the name or path of an object; "/" or an empty string specifies the root.`Parameter Name` returns the name of the parameter. `Info Type` returns the type of data requested; this can be one of "type", "value", "default value", or "allowed values". For example:

```
! $info = getParameterInfo("/USER DEFINED/POINT:Point 1", "Symbol Size", "default value");
! print "getParameterInfo returned=$info\n";
```

prints:

```
getParameterInfo returned=2.5
```

16.2.3.33. getParameters(Object Name)

```
string getParameters("Object Name")
```

Returns the parameters of an object in a comma-separated list. Use '`split ","`' to convert the string into an array of strings.

16.2.3.34. getTempDirectory()

```
string getTempDirectory()
```

Returns the temporary directory path.

16.2.3.35. getType(Object Name)

```
string getType("Object Name")
```

Returns the object type.

16.2.3.36. getValue(Object Name, Parameter Name)

```
string getValue("Object Name", "Parameter Name")
```

Takes a CCL object and parameter name and returns the value of the parameter.

Returns the value stored in `Parameter Name`.

16.2.3.36.1. Example

1. Create a text object called **Text 1**.
2. In the **Text String** box, enter `Here is a text string`.

3. Click **Apply** to create the text object.
4. In the **Command Editor** dialog box, enter the following:

```
!string = getValue( "/TEXT:Text 1/TEXT ITEM: Text Item 1", "Text String");
! print $string;
```

5. Click **Process**, and the string will be printed to your terminal window.

The same procedure can be carried out for any object.

16.2.3.37. getViewArea()

```
string,string getViewArea()
```

Calculates the area of the scene projected in the view direction. Returns the area and the units in an array of strings.

16.2.3.38. isCategory(Object Name, Category)

```
bool isCategory("Object Name", "Category")
```

A return of 1 indicates that the object matches the passed category; 0 otherwise. Categories are specified in <*CFXROOT*>/etc/CFXPostRules.ccl.

For example, the following prints "Plane 1 is a surface":

```
! if( isCategory( "Plane 1", "surface" )) {
!   print "Plane 1 is a surface\n";
! }
```

16.2.3.39. Length(Location)

```
real Length("Location")
```

Returns the length of a line locator. For details, see *length* (p. 168).

Note

While using this function in Power Syntax the leading character is capitalized to avoid confusion with the Perl internal command “length.”

16.2.3.40. lengthAve(Variable, Location)

```
real lengthAve("Variable", "Location")
```

Returns the length-based average of the variable on the line locator. For details, see *lengthAve* (p. 168).

16.2.3.41. lengthInt(Variable, Location)

```
real lengthInt("Variable", "Location")
```

Returns the length-based integral of the variable on the line locator. For details, see *lengthInt* (p. 169).

16.2.3.42. liquidTurbPerformTurbo()

This is an internal subroutine that is used only to initialize report templates.

16.2.3.43. liquidTurbPerform()

This is an internal subroutine that is used only to initialize report templates.

16.2.3.44. massFlow(Location)

```
real massFlow("Location")
```

Returns the mass flow through the 2D locator. For details, see *massFlow* (p. 169).

16.2.3.45. massFlowAve(Variable, Location)

```
real massFlowAve("Variable","Location")
```

Returns the average value of the variable, weighted by mass flow, through the 2D locator. For details, see *massFlowAve* (p. 171).

16.2.3.46. massFlowAveAbs(Variable, Location)

```
real massFlowAveAbs("Variable","Location")
```

Returns the average value of the variable, weighted by absolute mass flow, through the 2D locator. For details, see *massFlowAveAbs* (p. 171).

16.2.3.47. massFlowInt(Variable, Location)

```
real massFlowInt("Variable", "Location")
```

Returns the integral of the variable, weighted by mass flow, over the 2D locator. For details, see *massFlowInt* (p. 173).

16.2.3.48. maxVal(Variable, Location)

```
real maxVal("Variable", "Location")
```

Returns the maximum value of the variable at the location. For details, see *maxVal* (p. 174).

16.2.3.49. minVal(Variable, Location)

```
real minVal("Variable", "Location")
```

Returns the minimum value of the variable at the location. For details, see *minVal* (p. 174).

16.2.3.50. objectExists(Object Name)

```
bool objectExists("Object Name")
```

A return of 1 indicates that the object exists; 0 otherwise.

16.2.3.51. probe(Variable, Location)

```
real probe("Variable", "Location")
```

Important

This calculation should only be performed for point locators described by single points. Incorrect solutions will be produced for multiple point locators.

Returns the value of the variable at the point locator. For details, see *probe* (p. 175).

16.2.3.52. pumpPerform()

This is an internal subroutine that is used only to initialize report templates.

16.2.3.53. pumpPerformTurbo()

This is an internal subroutine that is used only to initialize report templates.

16.2.3.54. range(Variable, Location)

```
real,real range("Variable", "Location")
```

Returns the minimum and maximum values of the variable at the location.

16.2.3.55. reportError(String)

```
void reportError("String")
```

Pops up an error dialog box.

16.2.3.56. reportWarning(String)

```
void reportWarning("String")
```

Pops up a warning dialog box.

16.2.3.57. showPkgs()

```
void showPkgs()
```

Prints to the console a list of packages available which may contain other variables or subroutines in Power Syntax.

16.2.3.58. showSubs(packageName)

```
void showSubs("packageName")
```

Prints to the console a list of the subroutines available in the specified package. If no package is specified, CFD-Post is used by default.

16.2.3.59. showVars(packageName)

```
void showVars("packageName")
```

Prints to the console a list of the Power Syntax variables and their current value defined in the specified package. If no package is specified, CFD-Post is used by default.

16.2.3.60. spawnAsyncProcess(command, arguments)

```
bool spawnAsyncProcess("command", "arguments")
```

Spawns a forked process. For example:

```
! spawnAsyncProcess("dir", "c:/");
```

Displays the contents of c:\ in the console window.

16.2.3.61. sum(Variable, Location)

```
real sum("Variable", "Location")
```

Returns the sum of the variable values at each point on the locator. For details, see *sum* (p. 176).

16.2.3.62. torque(Location, Axis)

```
real torque("Location", "Axis")
```

Returns the computed value of torque at the 2D locator about the specified axis. For details, see *torque* (p. 177).

16.2.3.63. turbinePerform()

This is an internal subroutine that is used only to initialize report templates.

16.2.3.64. turbinePerformTurbo()

This is an internal subroutine that is used only to initialize report templates.

16.2.3.65. verboseOn()

```
bool verboseOn()
```

Returns 1 or 0 depending if the Perl variable `$verbose` is set to 1.

16.2.3.66. volume(Location)

```
real volume("Location")
```

Returns the volume of a 3D locator. For details, see *volume* (p. 177).

16.2.3.67. volumeAve(Variable, Location)

```
real volumeAve("Variable", "Location")
```

Returns the average value of a variable over the 3D locator. For details, see *volumeAve* (p. 178).

16.2.3.68. volumeInt(Variable, Location)

```
real volumeInt("Variable", "Location")
```

Returns the integral of a variable over the 3D locator. For details, see *volumeInt* (p. 178).

Chapter 17: Bibliography

This bibliography contains entries referenced in the CFX documentation.

17.1. References 1-20

1

Hutchinson, B.R. and Raithby, G.D.,

"A Multigrid method Based on the Additive Correction Strategy", Numerical Heat Transfer, Vol. 9, pp. 511-537, 1986.

2

Rhie, C.M. and Chow, W.L.,

"A Numerical Study of the Turbulent Flow Past an Isolated Airfoil with Trailing Edge Separation",

AIAA Paper 82-0998, 1982

3

Raw, M.J.,

"A Coupled Algebraic Multigrid Method for the 3D Navier-Stokes Equations",

10th GAMM Seminar, Kiel, 1994.

4

Launder, B.E., Reece, G.J. and Rodi, W.,

"Progress in the developments of a Reynolds-stress turbulence closure",

J. Fluid Mechanics, Vol. 68, pp.537-566, 1975.

5

Speziale, C.G., Sarkar, S. and Gatski, T.B.,

"Modelling the pressure-strain correlation of turbulence: an invariant dynamical systems approach",

J. Fluid Mechanics, Vol. 277, pp. 245-272, 1991.

6

Schiller, L. and Naumann, A.,

VDI Zeits, 77, p. 318, 1933.

7

Hughmark, G.A.,

AIChE J., 13 p. 1219, 1967.

8

Modest, M.,

"Radiative Heat Transfer", Second Edition

Academic Press, 2003.

9

Menter, F.R.,

"Two-equation eddy-viscosity turbulence models for engineering applications",

AIAA-Journal., 32(8), pp. 1598 - 1605, 1994.

10

Grotjans, H. and Menter, F.R.,

"Wall functions for general application CFD codes",

In K.D.Papailiou et al., editor, ECCOMAS 98 Proceedings of the Fourth European Computational Fluid Dynamics Conference, pp. 1112-1117. John Wiley & Sons, 1998.

11

Wilcox, D.C.,

"Multiscale model for turbulent flows",

In AIAA 24th Aerospace Sciences Meeting. American Institute of Aeronautics and Astronautics, 1986.

12

Menter, F.R.,

"Multiscale model for turbulent flows",

In 24th Fluid Dynamics Conference. American Institute of Aeronautics and Astronautics, 1993.

13

Launder, B.E. and Spalding, D.B.,

"The numerical computation of turbulent flows",

Comp Meth Appl Mech Eng, 3:269-289, 1974.

14

White, F.M.,

"Viscous Fluid Flow", Second Edition,

McGraw-Hill, 1991.

15

Kader, B.A.,

"Temperature and concentration profiles in fully turbulent boundary layers",

International Journal of Heat and Mass Transfer, 24(9):1541-1544, 1981.

16

Huang, P.G., Bradshaw, P. and Coakley, T.J.,

"Skin friction and velocity profile family for compressible turbulent boundary layers",

American Institute of Aeronautics and Astronautics Journal, 31(9):1600-1604, 1993.

17

Bouillard, J.X, Lyczkowski, R.W.and Gidaspow, D.,

"Porosity Distribution in a Fluidised Bed with an Immersed Obstacle",

AIChE J., **35**, 908-922, 1989.

18

Gidaspow, D.,

"Multiphase Flow and Fluidisation", Academic Press, 1994.

19

Ishii, M. and Zuber, N.,

"Drag Coefficient and Relative Velocity in Bubbly, Droplet or Particulate Flows",

AIChE J., **25**, 843-855, 1979.

20

Lopez de Bertodano, M.,

"Turbulent Bubbly Flow in a Triangular Duct",

Ph.D. Thesis, Rensselaer Polytechnic Institute, Troy New York, 1991.

17.2. References 21-40

21

Lopez de Bertodano, M.,

"Two Fluid Model for Two-Phase Turbulent Jet",

Nucl. Eng. Des. **179**, 65-74, 1998.

22

Sato, Y. and Sekoguchi, K.,

"Liquid Velocity Distribution in Two-Phase Bubbly Flow",

Int. J. Multiphase Flow, **2**, p.79, 1975.

23

Siegel, R and J.R. Howell,

"Thermal Radiation Heat Transfer",

ISBN 0-89116-506-1.

24

Goldstein, M. and J.R. Howell,

"Boundary Conditions for the Diffusion Solution of Coupled Conduction-Radiation Problems",

NASA Technical Note, NASA TN D-4618.

25

Raw, M.J.,

"Robustness of Coupled Algebraic Multigrid for the Navier-Stokes Equations",

AIAA 96-0297, 34th Aerospace and Sciences Meeting & Exhibit, January 15-18 1996, Reno, NV.

26

Kee, R. J., Rupley, F. M. and Miller, J. A.,

"Chemkin -II: A Fortran Chemical Kinetics Package for the Analysis of Gas-Phase Chemical Kinetics",

Sandia National Laboratories Report, SAND89-8009,(1991).

27

Brackbill, J.U, Kothe, D.B. and Zemach, C.,

"A Continuum Method for Modelling Surface Tension",

Journal of Computational Physics 100:335-354, 1992.

28

Barth, T.J., and Jesperson, D.C,

"The Design and Application of Upwind Schemes on Unstructured Meshes",

AIAA Paper 89-0366, 1989.

29

Bird, R.B., Stewart, W.E. and Lightfoot, E.N.,

"Transport Phenomena",

John Wiley & Sons, Inc., 1960.

30

Wilcox, D.C.,

"Turbulence Modelling for CFD",

DCW Industries, 2000, La Canada, CA 91011, p. 314.

31

Launder, B.E., Tselepidakis, D. P., Younis, B. A.,

"A second-moment closure study of rotating channel flow",

J. Fluid Mech., Vol. 183, pp. 63-75, 1987.

32

Menter, F. R.,

"Eddy Viscosity Transport Equations and their Relation to the $k-\varepsilon$ Model",

NASA Technical Memorandum 108854, November 1994.

33

Menter, F. R.,

"Eddy Viscosity Transport Equations and their Relation to the $k-\varepsilon$ Model",

ASME J. Fluids Engineering, Vol. 119, pp. 876-884, 1997.

34

Smagorinsky, J.,

"General Circulation Experiments with the Primitive Equations",

Month. Weath. Rev. Vol. 93, pp. 99-165, 1963.

35

Clift, R., Grace, J.R., Weber, M.E.,

"Bubbles, Drops and Particles",

Academic Press, New York, U.S.A., 1978.

36

Liang, L., Michaelides, E. E.,

"The magnitude of Basset forces in unsteady multiphase flow computations",

Journal of Fluids Engineering, Vol. 114, pp. 417-419, 1992.

37

Peters, N.,

"Turbulent Combustion",

Cambridge monographs on mechanics, Cambridge University Press, 2000.

38

Zimont, V.L., Polifke, W., Bettelini, M. and Weisenstein, W.,

"An efficient Computational Model for Premixed Turbulent Combustion at High Reynolds Numbers Based on a Turbulent Flame Speed Closure",

J. Engineering for Gas Turbines and Power (Transactions of the ASME), Vol. 120, pp. 526-532, 1998.

39

Hinze, J. O.,

"Turbulence",

McGraw-Hill, New York, U.S.A., 1975.

40

Zimont, V.L.,

"Gas Premixed Combustion at High Turbulence. Turbulent Flame Closure Combustion Model",

Proceedings of the Mediterranean Combustion Symposium, Instituto di Richerche sulla Combustione - CNR, Italy, pp. 1155-1165, 1999.

17.3. References 41-60

41

Zimont, V.L., Biagioli, F. and Syed, Khawar,

"Modelling turbulent premixed combustion in the intermediate steady propagation regime",

Progress in Computational Fluid Dynamics, Vol. 1, pp. 14-28, 2001.

42

Linan, A.,

"On the internal structure of laminar diffusion flames",

Technical note, Inst. nac. de tec. aeron., Esteban Terradas, Madrid, Spain, 1961.

43

Warnatz, J., Mass, U. and Dibble, R. W.,

"Combustion",

Springer, Verlag, 1996, pp.219-221.

44

Magnussen, B. F.,

"The Eddy Dissipation Concept for Turbulent Combustion Modelling. Its Physical and Practical Implications",

Presented at the First Topic Oriented Technical Meeting, International Flame Research Foundation, IJmuiden, The Netherlands, Oct. 1989.

45

Tesner, P. A., Snegirova, T. D., and Knorre, V. G.,

"Kinetics of Dispersed Carbon Formation",

Combustion and Flame, Vol. 17, pp. 253-260, 1971.

46

Magnussen, B. F., and Hjertager, B. H.,

"On Mathematical Modeling of Turbulent Combustion with Special Emphasis on Soot Formation and Combustion",

Sixteenth Symp. (Int.) on Combustion, The Combustion Institute, p 719, 1976.

47

Vukalovich, M. P.,

"Thermodynamic Properties of Water and Steam",

Mashgis, Moscow, 6th ed., 1958.

48

Hottel, H.C. and Sarofim, A.F.,

"Radiative transfer",

McGraw-Hill, New York 1967.

49

Hadvig, S.,

"Gas emissivity and absorptivity",

J. Inst. Fuel, 43, pp. 129-135., 1970.

50

Leckner, B.,

"Spectral and total emissivity of water vapour and carbon dioxide",

Comb. Flame, **19**, pp. 33-48., 1972.

51

Taylor, P.B. and Foster, P.J.,

"The total emissivities of luminous and non-luminous flames",

Int. J. Heat Mass Transfer, **17**, pp. 1591-1605., 1974.

52

Beer, J.M., Foster, P.J. and Siddall, R.G.,

"Calculation methods of radiative heat transfer",

HTFS Design Report No. 22, AEA Technology (Commercial)., 1971.

53

Prakash, C.,

"Two phase model for binary liquid-solid phase change",

Parts I and II, Numerical Heat Transfer, B 15, p. 171.

54

CFX Limited, Waterloo, Ontario, Canada,

CFX-TASCflow Theory Documentation,

Section 4.1.2, Version 2.12, 2002.

55

Menter, F. R. and Kuntz, M.,

"Development and Application of a Zonal DES Turbulence Model for CFX-5",

CFX-Validation Report, CFX-VAL17/0503.

56

Menter, F.R., Kuntz, M.,

"Adaptation of Eddy-Viscosity Turbulence Models to Unsteady Separated Flow Behind Vehicles",

Proc. Conf. The Aerodynamics of Heavy Vehicles: Trucks, Busses and Trains, Asilomar, Ca, 2002.

57

Spalart, P.R, Jou, W.-H., Strelets, M. and Allmaras, S.R.,

"Comments on the feasibility of LES for wings, and on a hybrid RANS/LES approach",

1st AFOSR Int. Conf. On DNS/LES, Aug.4-8, 1997, Ruston, LA. In Advances in DNS/LES, C. Liu & Z. Liu Eds., Greyden Press, Colombus, OH.

58

Strelets, M.,

"Detached Eddy Simulation of Massively Separated Flows",

AIAA Paper 2001-0879, 39th Aerospace Sciences Meeting and Exhibit, Reno, NV, 2001.

59

Ishii, M.,

"One-dimensional drift-flux model and constitutive equations for relative motion between phases in various two-phase flow regimes",

Argonne National Laboratory ANL-77-47, 1977.

60

Manninen, M. and Tavassalo, V.,

"On the Mixture Models for Multiphase Flow",

VTT Publications, 1996.

17.4. References 61-80

61

Luo, S.M., and Svendsen, H.,

"Theoretical Model for Drop and Bubble Breakup in Turbulent Dispersions",

AIChE Journal 42, 1225 -1233.

62

Prince, M. and Blanch, H.,

"Bubble Coalescence and Break-Up in Air-Sparged Bubble Columns",

AIChE Journal 36, 1485-1499.

63

Hutchings, I.M.,

"Mechanical and metallurgical aspects of the erosion of metals",

Proc. Conf. on Corrosion-Erosion of Coal Conversion System Materials, NACE (1979) 393.

64

Dosanjh, S., and Humphrey, J.A.C.,

"The influence of turbulence C on erosion by a particle laden fluid jet, Wear",

V.102, 1985, pp. 309-330.

65

Aungier, R.H.,

"Centrifugal Compressors: A strategy for Aerodynamic Design and Analysis",

ASME Press, New York, 2000.

66

Westbrook, C.K., Dryer, F.L.,

"Simplified Reaction Mechanisms for the Oxidation of Hydrocarbon Fuels in Flames",

Combustion Science and Technology Vol. 27, pp. 31-43, 1981.

67

Faeth, G. M.,

"Mixing, transport and combustion in sprays",

Process Energy Combustion Science, Vol. 13, pp. 293-345, 1987.

68

Mijnbeek, G.,

"Bubble column, airlift reactors and other reactor designs", Operational Modes of Bioreactors, Chapter 4,

Butterworth and Heinemann, 1992.

69

Bello, R. A., Robinson, C. W., and Moo-Young, M.,

Canadian Journal of Chemical Engineering, Vol. 62, pp. 573. Chemical Institute of Canada and Canadian Society for Chemical Engineering, 1984.

70

García-Calvo, E. and Letón, P.,

"Prediction of gas hold-up and liquid velocity in airlift reactors using two-phase flow friction coefficients",

Journal of Chemical Technology & Biotechnology, Vol. 67, pp. 388-396,

Wiley Interscience, 1996.

71

Maneri, C. C. and Mendelson, H. D.,

American Institute of Chemical Engineers Journal, Vol. 14, p. 295. American Institute of Chemical Engineers, 1968.

72

Baker, J. L. L. and Chao, B. T.,

American Institute of Chemical Engineers Journal, Vol. 11, p. 268. American Institute of Chemical Engineers, 1965.

73

Hughmark, G. A.,

Industrial Engineering and Chemical Process Design and Development, Vol. 6, p. 218. 1967.

74

S. Lo, R. Bagatin and M. Masi.,

"The Development of a CFD Analysis and Design Tool for Air-lift Reactors",

Proceedings of the SAIChE 2000 Conference, Secunda, South Africa, 2000.

75

Ranz, W.E. and Marshall, W.R.,

Chem. Eng. Prog. 48(3), p. 141, 1952.

76

Bardina, J.E., Huang, P.G. and Coakley, T.J.,

"Turbulence Modeling Validation Testing and Development",

NASA Technical Memorandum 110446, 1997. (See also Bardina, J.E., Huang, P.G. and Coakley, T.J., "Turbulence Modeling Validation", AIAA Paper 97-2121.)

77

H. Schlichting.,

"Boundary Layer Theory",

McGraw-Hill, 1979.

78

Badzioch, S., and Hawksley, P.G.W.,

"Kinetics of thermal decomposition of pulverised coal particles, Industrial Engineering Chemistry Process Design and Development, 9 p. 521", 1997.

79

S.J. Ubhayakar, D.B. Stickler, C.W. Von Rosenburg, and R.E. Gannon;

"Rapid devolatilization of pulverised coal in hot combustion gases",

16th Symposium (International) on Combustion, The Combustion Institute, p. 426, 1976.

80

Wall, T.F., Phelan, W.J., and Bartz, S.,

"The prediction of scaling of burnout in swirled pulverised coal flames",

InternationalFlame Research Foundation Report F388/a/3 IJmuiden, TheNetherlands, 1976.

17.5. References 81-100

81

Sutherland, W.,

"The Viscosity of Gases and Molecular Force",

Phil. Mag. 5:507-531, 1893.

82

Hirschfelder, J.O., Taylor, and M.H., Bird, R.B.,

"Molecular Theory of Gases and Liquids",

Wiley, New York, 1954.

83

Chung, T.H., Lee, L.L., and Starling, K.E.,

"Applications of Kinetic Gas Theories and Multiparameter Correlation for Prediction of Dilute Gas Viscosity and Thermal Conductivity",

Ind. Eng. Chem. Fundam.

23:8, 1984.

84

Poling, B.E., Prausnitz, J.M., and O'Connell, J.P.,

"The Properties of Gases and Liquids",

McGraw-Hill, New York, 2001.

85

Redlich, O., and Kwong, J.N.S.,

"On the Thermodynamics of Solutions. V. An Equation of State. Fugacities of Gaseous Solutions.",

Chem Rev 44:233, 1949.

86

Saffman, P. G.,

"The lift on a small sphere in a slow shear flow",

J. Fluid Mech., 22, p. 385, 1965.

87

Mei, R. and Klausner, J. F.,

"Shear lift force on spherical bubbles",

Int. J. Heat and Fluid Flow, 15, p. 62, 1994.

88

Antal, S. P., Lahey, R. T., and Flaherty, J. E.,

"Analysis of phase distribution in fully developed laminar bubbly two-phase flow",

Int. J. Multiphase Flow, 7, pp. 635-652, 1991.

89

Krepper, E., and Prasser, H-M.,

"Measurements and CFX Simulations of a bubbly flow in a vertical pipe",

in AMIFESF Workshop, Computing Methods in Two-Phase Flow, 2000.

90

Burns, A.D.B., Frank, Th., Hamill, I., and Shi, J-M.,

"The Favre Averaged Drag Model for Turbulent Dispersion in Eulerian Multi-Phase Flows",

5th International Conference on Multiphase Flow, ICMF-2004, Yokohama, Japan.

91

Moraga, J.F., Larreteguy, A.E., Drew, D.A., and Lahey, R.T.,

"Assessment of turbulent dispersion models for bubbly flows in the low Stokes number limit",

Int. J. Multiphase Flow, 29, p. 655, 2003.

92

CIBSE Guide A: Environmental Design

CIBSE, U.K., 1999.

93

ISO 7730-1984(E), Moderate thermal environments - Determination of the PMV and PPD indices and specification of the conditions for thermal comfort

1984.

94

Yamada, T. and R.D. Gunn,

J. Chem Eng. Data, 18, p. 234, 1973.

95

Pitzer, K.S., D.Z. Lippmann, R.F. Curl, C.M. Huggins, and D.E. Petersen,

J. Am. Chem. Soc., 77: 3433 (1955).

96

Aungier, R.H.,

"A Fast, Accurate Real Gas Equation of State for Fluid Dynamic Analysis Applications",

Journal of Fluids Engineering, Vol. 117, pp. 277-281, 1995.

97

H. Enwald, E. Peirano and A. E. Almstedt,

"Eulerian Two-Phase Flow Theory Applied to Fluidisation",

Int. J. Multiphase Flow, 22 Suppl., pp. 21-66,1996.

98

J. Ding and D. Gidaspow,

"A Bubbling Fluidisation Model using Theory of Granular Flow",

AIChEJ. 36, pp. 523-538, 1990.

99

C.K.K. Lun, S.B. Savage, D.J. Jeffery, and N. Chepurniy,

"Kinetic Theories for Granular Flow: Inelastic Particles in Couette Flow and Slightly Inelastic Particles in a General Flow Field",

J. Fluid Mech., 140, pp. 223-256, 1984.

100

C.K.K. Lun, and S.B. Savage,

"The Effects of an Impact Velocity Dependent Coefficient of Restitution on Stresses Developed by Sheared Granular Materials",

Acta Mechanica., 63, pp. 15-44, 1986.

17.6. References 101-120

101

Menter, F.R., Langtry, R.B., Likki, S.R., Suzen, Y.B., Huang, P.G., and Völker, S.,

"A Correlation based Transition Model using Local Variables Part 1- Model Formulation",

ASME-GT2004-53452, ASME TURBO EXPO 2004, Vienna, Austria.

102

Langtry, R.B., Menter, F.R., Likki, S.R., Suzen, Y.B., Huang, P.G., and Völker, S.,

"A Correlation based Transition Model using Local Variables Part 2 - Test Cases and Industrial Applications",

ASME-GT2004-53454, ASME TURBO EXPO 2004, Vienna, Austria.

103

Langtry, R.B., Menter, F.R.,

"Transition Modeling for General CFD Applications in Aeronautics",

AIAA paper 2005-522, 2005.

104

Mayle, R.E.,

"The Role of Laminar-Turbulent Transition in Gas Turbine Engines",

ASME Journal of Turbomachinery, Vol. 113, pp. 509-537, 1991.

105

R. Schmehl,

"Advanced Modelling of Droplet Deformation and Breakup for CFD Analysis of Mixture Preparation",

ILASS-Europe 2002, 2002.

106

Miller A. and Gidaspow D,

AIChE Journal, Vol. 38, No. 11, p. 1811, 1992.

107

F.X. Tanner,

"Liquid Jet Atomization and Droplet Breakup Modeling of Non-Evaporating Diesel Fuel Sprays",

SAE Technical Paper Series, 970050, 1997.

108

B. Liu, D. Mather and R.D. Reitz,

"Effects of Drop Drag and Breakup on Fuel Sprays",

SAE Technical Paper 930072, 1993.

109

L.P. Hsiang and G.M. Faeth,

"Near-Limit Drop Deformation and Secondary Breakup",

International Journal of Multiphase Flow, Vol. 18, No. 5, pp. 635-652, 1992.

110

S. C. Kuensberg, S.-C., Kong and R. D. Reitz,

"Modelling the Effects of Injector Nozzle Geometry on Diesel Sprays",

SAE Paper 1999-01-0912, 1999.

111

H. Hiroyasu and T. Kadota,

"Fuel droplet size distribution in diesel combustion chamber",

SAE Technical Paper, 740715, 1974.

112

R. Schmehl, G. Maier and S. Wittig,

"CFD Analysis of Fuel Atomization, Secondary Droplet Breakup and Spray Dispersion in the Premix Duct of a LPP Combustor",

Proc. of 8th Int. Conf. on Liquid Atomization and Spray Systems, Pasadena, CA, USA, 2000.

113

Schlichting, H., and Gersten, K.,

"Grenzschicht-Theorie".

9. Auflage, Springer-Verlag Berlin, Heidelberg, New York, 1997

114

W.H. Nurick,

"Orifice Cavitation and Its Effect on Spray Mixing",

Journal of Fluids Engineering, Vol. 98, pp. 681-687, 1976.

115

R.D. Reitz and R. Diwakar,

"Structure of High-Pressure Fuel Sprays",

SAE Technical Paper, 870598, 1987.

116

P.J.O'Rourke and A.A. Amsden,

"The TAB Method for Numerical Calculation of Spray Droplet Breakup",

SAE Technical Paper 872089, 1987.

117

M. Pilch and C.A. Erdman,

"Use of Breakup Time Data and Velocity History Data to Predict the Maximum Size of Stable Fragments for Acceleration-Induced Breakup of a Liquid Drop",

Int. J. Multiphase Flow, Vol. 13, No. 6, pp. 741-757, 1987.

118

S.V. Patankar,

"Numerical Heat Transfer and Fluid Flow",

Hemisphere Publishing Corp., 1980.

119

S. Majumdar,

"Role of Underrelaxation in Momentum Interpolation for Calculation of Flow with Non-staggered Grids",

Numerical Heat Transfer 13:125-132.

120

C. Baumgarten, H. Lettmann and G.P. Merker,

"Modelling of Primary and Secondary Break-Up Processes in High Pressure Diesel Sprays",

Paper No. 7, CIMAC Congress, Kyoto 2004.

17.7. References 121-140

121

T. Iijima and T. Takeno,

"Effects of pressure and temperature on burning velocity",

Combust. Flame, Vol. 65, pp. 35-43, 1986.

122

B. Lewis and G. v. Elbe,

"Combustion, Flames and Explosions of Gases",

3rd Edition, Academic Press, London, 1987.

123

B. E. Milton and J.C. Keck,

"Laminar burning velocities in stoichiometric hydrogen and hydrogen-hydrocarbon gas mixtures",

Combust. Flame, Vol. 58, pp. 13-22, 1984.

124

M. Metghalchi and J.C. Keck,

"Burning Velocities of Mixtures of Air with Methanol, iso-octane and indolene at High Pressure and Temperature",

Combust. Flame, Vol. 48, pp. 191-210, 1982.

125

W. Wagner, and A. Kruse,

"The Industrial Standard IAPWS-IF97: Properties of Water and Steam",

Springer, Berlin, 1998.

126

Senecal, P.K., Schmidt, D.P., Nouar, I., Rutland, C.J., Reitz, R.D. and Corradin, M.L.,

"Modeling High-Speed Viscous Liquid Sheet Atomization",

International Journal of Multiphase Flow, 25, pp. 1073-1097, 1999.

127

Han, Z., Perrish, S., Farrell, P.V. and Reitz R.D.,

"Modeling Atomization Processes of Pressure-Swirl Hollow-Cone Fuel Sprays",

Atomization and Sprays, Vol. 7, pp. 663-684, Nov.-Dec. 1997.

128

Kato, M., Launder, B.E.,

"The modelling of turbulent flow around stationary and vibrating square cylinders",

Ninth Symposium on "Turbulent Shear Flows", Kyoto, Japan, August 16-18, 1993.

129

Menter, F. R.,

"Zonal two equation $k-\omega$ turbulence models for aerodynamic flows",

AIAA Paper 93-2906, 1993.

130

Menter F. R. and Egorov, Y.,

"Re-visiting the turbulent scale equation",

Proc. IUTAM Symposium; One hundred years of boundary layer research, Göttingen, 2004.

131

Menter, F.R. and Egorov, Y.,

"A Scale-Adaptive Simulation Model using Two-Equation Models",

AIAA paper 2005-1095, Reno/NV, 2005.

132

Menter, F. R, Kuntz, M., Bender R.,

"A scale-adaptive simulation model for turbulent flow predictions",

AIAA Paper 2003-0767, 2003.

133

Menter. F. R., Kuntz, M. and Durand. L.,

"Adaptation of eddy viscosity turbulence models to unsteady separated flow behind vehicles",

The Aerodynamics Of Heavy Vehicles: Trucks, Buses And Trains, Monterey, Dec.2-6, 2002.

134

Rotta, J. C.,

"Turbulente Strömungen",

Teubner Verlag, Stuttgart, 1972.

135

Spalart P. R.,

"Young-Person's Guide to Detached-Eddy Simulation Grids",

NASA/CR-2001-211032, 2001.

136

Wilcox, D. C.,

"Turbulence Modelling for CFD",

DWC Industries, La Cañada, 1993.

137

Squires, K.,

"Detached eddy simulation: Current status and future perspectives",

Proc. DLES-5 Conference, München, 2004.

138

Jovic, S., Driver, D. M.,

"Backward-facing step measurement at low Reynolds number, Re_h=5000",

NASA TM 108807, 1994.

139

Roache, P. J.,

“Verification and Validation in Computational Science and Engineering”,

Hermosa publishers, Albuquerque, New Mexico, 1998.

140

Casey, M. and Wintergerste W.,

"Best Practice Guidelines",

ERCOFTAC Special Interest Group on Quality and Trust in Industrial CFD, Report, 2000.

17.8. References 141-160

141

Ferziger, J. H. and Peric, M.,

"Computational methods for fluid dynamics",

Springer, Berlin.

142

Menter, F. R. and Esch T.,

"Elements of Industrial Heat Transfer Predictions",

16th Brazilian Congress of Mechanical Engineering (COBEM), Nov. 2001, Uberlandia, Brazil.

143

Menter, F.,

"CFD Best Practice Guidelines for CFD Code Validation for Reactor Safety Applications",

Evaluation of Computational Fluid Dynamic Methods for Reactor Safety Analysis (ECORA), European Commission, 5th EURATOM FRAMEWORK PROGRAMME, 1998-2002.

144

Menter F. R. and Egorov, Y.,

"Turbulence Models based on the Length-Scale Equation",

Fourth International Symposium on Turbulent Shear Flow Phenomena, Williamsburg, 2005 - Paper TSFP4-268, 2005.

145

Menter F. R. and Egorov, Y.,

"SAS Turbulence Modelling of Technical Flows",

DLES 6 - 6th ERCOFTAC Workshop on Direct and Large Eddy Simulation September, Poitiers, 2005.

146

Spalart P. R.,

"Strategies for turbulence modelling and simulations",

Int. J. Heat Fluid Flow, 21, pp. 252-263, 2000.

147

A. D. Gosman and E. Ioannides,

"Aspects of computer simulation of liquid fuelled combustors",

AIAA Paper, No. 81-0323,1981.

148

F. Bakir, R. Rey, A.G. Gerber, T. Belamri and B. Hutchinson,

"Numerical and Experimental Investigations of the Cavitating Behavior of an Inducer",

Int J Rotating Machinery, Vol. 10, pp. 15-25, 2004.

149

Frank, Th.,

"Parallele Algorithmen für die numerische Simulation dreidimensionaler, disperser Mehrphasenströmungen und deren Anwendung in der Verfahrenstechnik",

Habilitationsschrift, Shaker Verlag Aachen, pp. 1-329, 2002.

150

Hussmann, B. et al.,

"A stochastic particle-particle collision model for dense gas-particle flows implemented in the Lagrangian solver of ANSYS CFX and its validation",

6th International Conference on Multiphase Flows, ICMF 2007, Leipzig, Germany, 2007.

151

Oesterlé, B., Petitjean, A.,

"Simulations of particle-to-particle interactions in gas-solid flows",

Int. J. Multiphase Flow, Vol. 19(1), pp. 199-211, 1993.

152

Sommerfeld, M.,

"Modellierung und numerische Berechnung von partikelbeladenen Strömungen mit Hilfe des Euler-Lagrange-Verfahrens",

Habilitationsschrift, Shaker Verlag Aachen, 1996.

153

Lavieville, J., Deutsch, E. and Simonin, O.,

"Large eddy simulations of interactions between colliding particles and a homogeneous isotropic turbulence field",

ASME FED Vol. 228, pp. 359-369, 1995.

154

Sommerfeld, M.,

"Validation of a stochastic Lagrangian modeling approach for inter-particle collision in homogeneous isotropic turbulence",

Int. J. Multiphase Flow, Vol. 27, pp. 1829 – 1858, 2001.

155

Huh, K.Y., Lee, E.,

"Diesel Spray Atomization Models Considering Nozzle Exit Turbulence Conditions",

Atomization and Sprays, Vol. 8, pp. 453-469, 1998.

156

Chryssakis, C.A., Assanis, D.N.,

"A Secondary atomization Model for Liquid Droplet Deformations and Breakup under High Weber Number Conditions",

ILASS Americas, 18th Annual Conference on Liquid Atomization and Spray Systems, Irvine, CA, 2005.

157

Peng, D.Y. and Robinson, D.B.,

"A New Two-Constant Equation of State",

Ind. Eng. Chem. Fundam., Vol. 15, No. 1, pp. 59 – 64, 1976.

158

Chung, T.H., M. Ajlan, L.L. Lee and K.E. Starling,

"Generalized Multiparameter Correlation for Nonpolar and Polar Fluid Transport Properties",

Ind. Eng. Chem. Res., Vol. 27, pp. 671 – 679, 1988.

159

Kurul, N. and Podowski, M. Z.,

"On the modeling of multidimensional effects in boiling channels",

ANS Proc. 27th National Heat Transfer Conference, Minneapolis, MN, July 28-31, 1991.

160

Kocamustafaogullari, G. and Ishii, M.,

"Interfacial area and nucleation site density in boiling systems",

Int. J. Heat Mass Transfer, 26 p. 1377, 1983.

17.9. References 161-180

161

Podowski, M. Z., Alajbegovic, A., Kurul, N., Drew, D.A. and Lahey, R. T.,

"Mechanistic modelling of CHF in forced-convection sub-cooled boiling",

Int. Conference on Convective Flow and Pool Boiling, Irsee, Germany, 1997a.

162

Podowski, R. M., Drew, D.A., Lahey, R. T. and Podowski, M. Z.,

"A mechanistic model of the ebullition cycle in forced-convection sub-cooled boiling",

NURETH-8, Kyoto, Japan, 1997b.

163

Egorov, Y. and Menter, F.,

"Experimental implementation of the RPI boiling model in CFX-5.6",

Technical Report ANSYS / TR-04-10., 2004.

164

Lemmert, M. and Chawla, J. M.,

"Influence of flow velocity on surface boiling heat transfer coefficient",

Heat Transfer and Boiling (Eds. E. Hahne and U. Grigull), Academic Press, 1977.

165

Tolubinski, V. I. and Kostanchuk, D. M.,

"Vapour bubbles growth rate and heat transfer intensity at subcooled water boiling",

4th. International Heat Transfer Conference, Paris, France, 1970.

166

Cole, R.,

"A photographic study of pool boiling in the region of CHF",

AIChEJ, 6 pp. 533-542, 1960.

167

Mikic, B. B. and Rohsenow, W. M.,

"A new correlation of pool boiling data including the fact of heating surface characteristics",

ASME J. Heat Transfer, 91 pp. 245-250, 1969.

168

Del Valle, V. H. and Kenning, D. B. R.,

"Subcooled flow boiling at high heat flux",

Int. J. Heat Mass Transfer, 28 p. 1907, 1985.

169

Ceumern-Lindenstjerna, W. C.,

"Bubble Departure and Release Frequencies During Nucleate Pool Boiling of Water and Aqueous NaCl Solutions",

Heat Transfer in Boiling, Academic Press and Hemisphere, 1977.

170

Saffman, P. G.,

Corrigendum to: "The lift on a small sphere in a slow shear flow",

J. Fluid Mech., 31, p. 624, 1968

171

Legendre, D. and Magnaudet, J.,

"The lift force on a spherical bubble in a viscous linear shear flow",

J. Fluid Mech., 368, pp. 81–126, 1998.

172

Tomiyama, A.,

"Struggle with computational bubble dynamics",

ICMF'98, 3rd Int. Conf. Multiphase Flow, Lyon, France, pp. 1-18, June 8-12, 1998.

173

Frank, Th., Shi, J. M. and Burns, A. D.,

"Validation of Eulerian Multiphase Flow Models for Nuclear Safety Applications",

3rd International Symposium on Two-Phase Flow Modelling and Experimentation, Pisa, Italy, 22-24, Sept. 2004.

174

Wellek, R. M., Agrawal, A. K. and Skelland, A. H. P.,

"Shapes of liquid drops moving in liquid media",

AIChE J, 12, pp. 854-862, 1966.

175

G. Elsässer,

"Experimentelle Untersuchung und numerische Modellierung der freien Kraftstoffstrahlausbreitung und Wandinteraktion unter motorischen Randbedingungen",

Dissertation, Logos Verlag, Berlin, 2001

176

C. Bai and A.D. Gosman,

"Prediction of spray wall impingement in reciprocating engines",

ILASS-Europe, July 1999

177

Frank, Th., Zwart, P. J., Krepper, E., Prasser, H. -M. and Lucas,

"Validation of CFD models for mono- and polydisperse air-water two-phase flows in pipes"

J. Nuclear Engineering & Design, Vol. 238, pp. 647–659, March 2008.

178

Lighthill, M. J.,

"On sound generated aerodynamically. I. General theory"

Proc. R. Soc. Series A, Vol. 211, p. 564, 1952.

179

Lighthill, M. J.,

"On sound generated aerodynamically. II. Turbulence as a source of sound"

Proc. R. Soc. Series A, Vol. 222, 1954.

180

Ffowcs-Williams, J. E. and Hawkings, D. L.,

"Theory relating to the noise of rotating machinery"

J. Sound Vib., Vol. 10, pp. 10-21, 1969.

17.10. References 181-200

181

Wen, C. Y. and Yu, Y. H.,

"Mechanics of Fluidization"

Chem. Eng. Prog. Symp. Ser. 62, pp. 100-111, 1966.

182

Choi, C. R. and Huh, K. Y.,

"Development and validation of a coherent flamelet model for a spark-ignited turbulent premixed flame in a closed vessel,"

Combustion & Flame Vol. 114, No. 3-4, 336-348, 1998.

183

A. M. Douaud, P.Eyzat,

"Four-Octane-Number Method for Predicting the Anti-Knock Behavior of Fuels and Engines",

SAE Technical Paper 780080, SAE, 1978.

184

M. P. Halstead, L. J. Kirsch, C. P. Quinn,

"The Autoignition of Hydrocarbon Fuels at High Temperatures and Pressures – Fitting of a Mathematical Model",

Combustion and Flame, Vol. 30, pp. 45-60, 1977.

185

H. O. Hardenberg, F.W. Hase,

"An Empirical Formula for Computing the Pressure Rise Delay of a Fuel from Its Cetane Number and from the Relevant Parameters of Direct-Injection Diesel Engines",

SAE Technical Paper 790493, SAE, 1979.

186

Meneveau, C., and Poinsot, T.,

"Stretching and quenching of flamelets in premixed turbulent combustion",

Combustion and Flame, 86:311-332, 1991.

187

T. Poinsot, D. Veynante,

"Theoretical and Numerical Combustion",

Edwards, 2001.

188

Wallin, S. and Johansson A.,

"A complete explicit algebraic Reynolds stress model for incompressible and compressible flows",

Journal of Fluid Mechanics, 403, pp. 89-132, 2000.

189

Wallin, S., and Johansson A.,

"Modelling streamline curvature effects in explicit algebraic Reynolds stress turbulence models",

International journal of Heat and Fluid Flow, 23(5), pp. 721-730, 2002.

190

Hellsten, A.,

"New advanced $k-\omega$ turbulence model for high-lift aerodynamics",

AIAA Paper 2004-1120, Reno, Nevada, 2004.

191

Spalart, P.R., and Shur, M.

"On the sensitization of turbulence models to rotation and curvature",

Aerospace Sci. Tech., 1(5), pp. 297-302, 1997.

192

Smirnov, P.E., and Menter, F.R.

"Sensitization of the SST turbulence model to rotation and curvature by applying the Spalart-Shur correction term",

ASME Paper GT 2008-50480, Berlin, Germany, 2008.

193

Coleman, H.W., Hodge, B.K., Taylor, R.P.,

"A Re-Evaluation of Schlichting's Surface Roughness Experiment",

Journal of Fluids Engineering, Vol. 106, 1984.

194

Lechner, R., and Menter, F.,

"Development of a rough wall boundary condition for ω-based turbulence models",

Technical Report ANSYS / TR-04-04, 2004.

195

Pimenta, M.M., Moffat, R.J. and Kays, W.M.,

" The Turbulent Boundary Layer: An Experimental Study of the Transport of Momentum and Heat with the Effect of Roughness",

Interim Report Stanford University, CA, 1975.

196

Launder, B.E.,

"Second-moment closure: present … and future".

Int. J. Heat and Fluid Flow, Vol. 10, No. 4, pp. 282-300, 1989.

197

Egorov, Y., and Menter, F.,

"Development and Application of SST-SAS Turbulence Model in the DESIDER Project",

Second Symposium on Hybrid RANS-LES Methods, Corfu, Greece, 2007.

198

Germano, M., Piomelli, U., Moin, P., Cabot, W.H.,

"A Dynamic Subgrid-Scale Eddy Viscosity Model",

Phys. Fluids A 3 (7), pp. 1760-1765, 1991.

199

Lilly, D.K.,

"A Proposed Modification of the Germano Subgrid-Scale Closure Method",

Phys. Fluids A 4 (3), pp. 633-635, 1992.

200

Nicoud, F., Ducros, F.,

"Subgrid-Scale Stress Modelling Based on the Square of the Velocity Gradient Tensor",

Flow, Turbulence and Combustion, 62, pp. 183-200, 1999.

17.11. References 201 –

201

Abramzon, B. and Sirignano, W.A.,

"Droplet Vaporization Model for Spray Combustion Calculations"

Int. J. Heat Mass Transfer, 32, pp. 1605–1618, 1989.

202

Sazhin, Sergei S.,

"Advanced Models of Fuel Droplet Heating and Evaporation"

Progress in Energy and Combustion Science, 32, pp. 162–214, 2006.

203

Hughes, T.J.R.,

"The Finite Element Method"

Prentice-Hall, Englewood Cliffs, N.J., 1987.

204

Simo, J.C. and Wong, K.K.,

"Unconditionally Stable Algorithms for Rigid Body Dynamics that exactly Preserves Energy and Momentum"

Int. J. Num. Methods in Eng., 31, pp. 19-52, 1991.

205

Hughes, Peter C.,

"Spacecraft Attitude Dynamic"

Dover, 2004.

206

Etkin, Bernard.,

"The Dynamics of Atmospheric Flight"

John Wiley & Sons, 1972.

207

Conaire, Marcus Ó, Curran, Henry J., Simmie, John M., Pitz, William J., Westbrook, Charles K.,

"A Comprehensive Modeling Study of Hydrogen Oxidation",

International Journal of Chemical Kinetics,

Volume 36, Issue 11, pp. 603-622, 2004.

208

Frank, Th.

"Numerische Simulation der feststoffbeladenen Gasströmung im horizontalen Kanal unter Berücksichtigung von Wandrauhigkeiten"

Ph.D. Thesis, Techn. University Bergakademie Freiberg, Germany, 1992

209

Matsumoto, S., Saito, S., Maeda, S.

"Simulation of Gas-Solid Two-Phase Flow in Horizontal Pipe"

Journal of Chemical Engineering of Japan

Vol. 9, No. 1, pp. 23–28, 1976

210

Tsuji, Y., Oshima, T., Morikawa, Y.

"Numerical Simulation of Pneumatic Conveying in a Horizontal Pipe"

KONA Powder Science and Technology in Japan

No. 3, pp. 38–51, 1985

211

Frank, Th.

"Parallele Algorithmen für die numerische Simulation dreidimensionaler, disperser Mehrphasenströmungen und deren Anwendung in der Verfahrenstechnik"

Doctorial Dissertation, Shaker Verlag, 2002

212

Sommerfeld, M.

"Numerical Simulation of the Particle Dispersion in Turbulent Flow: the Importance of Particle Lift Forces and Particle/Wall Collision Models"

ASME Symposium on Numerical Methods for Multiphase Flows, Toronto, Canada

pp. 1–8, 1990

213

B. P. Leonard,

"The ULTIMATE conservative difference scheme applied to unsteady one-dimensional advection,"

Comp. Methods Appl. Mech. Eng.,

88:17–74, 1991

214

Jasak, H.; Weller, H.G., Gosman, A.D.

"High resolution NVD differencing scheme for arbitrarily unstructured meshes,"

Int. J. Numer. Meth. Fluids, 1999,

pp. 413 – 449.

215

Erdos, J.A.

"Numerical Solution of Periodic Transonic Flow through a Fan Stage"

AIAA Journal, 1977.

pp. 1559-1568.

216

Gerolymos G.A.

"Analysis and Application of Chorochronic Periodicity in Turbomachinery Rotor/Stator Interaction Computations"

Journal of Propulsion and Power, 2002.

pp. 1139-1152.

217

Giles, M.

"Calculation of Unsteady Wake/Rotor Interaction"

J. Propulsion, 1988.

pp. 356-362.

218

He, L.

"An Euler Solution for Unsteady Flows Around Oscillating Blades"

Transactions of the ASME, 1990.

pp. 714-722.

219

Apsley, D.D. and Leschziner, M.A.

"A new low-reynolds-number nonlinear two-equation turbulence model for complex flows"

International Journal of Heat and Fluid Flow, 19, pp. 209-222, 1998

Glossary

Symbols

<CFDPOSTROOT>	The directory in which CFD-Post is installed; for example: `C:\Program Files\ANSYS Inc\v140\CFD-Post\`
<CFXROOT>	The directory in which CFX is installed; for example: `C:\Program Files\ANSYS Inc\v140\CFX\`

A

absolute pressure	The summation of solver pressure, reference pressure, and hydrostatic pressure (if a buoyant flow) in the cavitation model. The absolute pressure is clipped to be no less than the vapor pressure of the fluid. It is used by the solver to calculate pressure-dependent properties (such as density for compressible flow).
absorption coefficient	A property of a medium that measures the amount of thermal radiation absorbed per unit length within the medium.
adaption criteria	The criteria that are used to determine where mesh adaption takes place.
adaption level	The degree that a mesh element has been refined during adaption. Each mesh element has an adaption level. Each time an element is split into smaller elements, the new elements have an adaption level that is one greater than the "parent" element. The maximum number of adaption levels is controlled to prevent over-refinement.
adaption step	One loop of the adapt-solve cycle in the mesh adaption process.
Additional Variable	A non-reacting, scalar component. Additional Variables are used to model the distribution of passive materials in the flow, such as smoke in air or dye in water. Additional Variables are typically specified as concentrations.
adiabatic	The description of any system in which heat is prevented from crossing the boundary of the system. You can set adiabatic boundary conditions for heat transfer simulations in ANSYS CFX or in FLUENT.
Advancing Front and Inflation (AFI)	The default meshing mode in CFX. The AFI mesher consists of a triangular surface/tetrahedral volume mesh generator that uses the advancing front method to discretize first the surface and then the volume into an unstructured (irregular) mesh. Inflation can be applied to selected surfaces to produce prismatic elements from the triangular surface mesh, which combine with the tetrahedra to form a hybrid mesh.

all domains

In immersed-solids cases in CFD-Post, "all domains" refers to all of the domains in the case *excluding* the immersed solid. This is done for backwards compatibility.

Generally speaking, only the wireframe needs to keep track of both "all domains" *and* the immersed solid.

ASM (Algebraic Slip Model)

A mathematical form in which geometry may be represented, known as parametric cubic.

aspect ratio

Also known as normalized shape ratio. A measure of how close to a regular tetrahedron any tetrahedron is. The aspect ratio is 1 for a regular tetrahedron, but gets smaller the flatter the tetrahedron gets. Used for judging how good a mesh is.

B

backup file

An intermediate CFX-Solver Results file that can be manually generated during the course of a solution from the CFX-Solver Manager interface by using the **Backup** action button. Backup files should be generated if you suspect your solution may be diverging and want to retain the intermediate solution from which you can do a restart.

batch mode

A way to run some components of ANSYS CFX without needing to open windows to control the process. When running in batch mode, a Viewer is not provided and you cannot enter commands at a command prompt. Commands are issued via a CFD-Post session file (`*.cse`), the name of which is specified when executing the command to start batch mode. The session file can be created using a text editor, or, more easily, by recording a session while running in line-interface or user-interface mode.

blend factor

A setting that controls the degree of first/second order blending for the advection terms in discrete finite volume equations.

body

A collection of surfaces that completely and unambiguously enclose a finite volume. Modelers that create so-called B-Rep models create "bodies." This term was coined to distinguish between the tri-parametric entities, known herein as solids, and the shell-like representations produced by most CAD systems.

boundary

A surface or edge that limits the extent of a space. A boundary can be internal (the surface of a submerged porous material) or external (the surface of an airfoil).

boundary condition

Physical conditions at the edges of a region of interest that you must specify in order to completely describe a simulation.

buoyant flow

Flow that is driven wholly or partially by differences in fluid density. For fluids where density is not a function of temperature, pressure, or Additional Variables, the Boussinesq approximation is employed.

If density is a function of one of these, then the Full Buoyancy model is employed.

C

CEL (CFX Expression Language)
: A high level language used within CFX to develop expressions for use in your simulations. CEL can be used to apply user-defined fluid property dependencies, boundary conditions, and initial values. Expressions can be developed within CFX using the Expression Editor.

CFD (Computational Fluid Dynamics)
: The science of predicting fluid flow, heat transfer, mass transfer (as in perspiration or dissolution), phase change (as in freezing or boiling), chemical reaction (as in combustion), mechanical movement (as in fan rotation), stress or deformation of related solid structures (such as a mast bending in the wind), and related phenomena by solving the mathematical equations that govern these processes using a numerical algorithm on a computer.

CFX-Solver Input file
: A file that contains the specification for the whole simulation, including the geometry, surface mesh, boundary conditions, fluid properties, solver parameters and any initial values. It is created by CFX and used as input to CFX-Solver.

CHT (Conjugate Heat Transfer)
: Heat transfer in a conducting solid.

clipping plane
: A plane that is defined through the geometry of a model, in front of which no geometry is drawn. This enables you to see parts of the geometry that would normally be hidden.

command actions
: Command actions are:

  - Statements in session files
  - Commands entered into the **Tools** > **Command Editor** dialog box
  - Commands entered in Line Interface mode.

  All such actions must be preceded with the > symbol. These commands force CFD-Post to undertake specific tasks, usually related to the input and output of data from the system. See also *Power Syntax* (p. 288).

component
: A substance containing one or more materials in a fixed composition. The properties of a component are calculated from the mass fractions of the constituent materials and are based on the materials forming an ideal mixture.

compressible flow
: Flow in which the fluid volume changes in response to pressure change. Compressible flow effects can be taken into consideration when the Mach number (M) approaches approximately 0.2.

computational mesh
: A collection of points representing the flow field where the equations of fluid motion (and temperature, if relevant) are calculated.

control volume
: The volume surrounding each node, defined by segments of the faces of the elements associated with each node. The equations of fluid flow are solved over each control volume.

convergence
: A state of a solution that occurs when the change in residual values from one iteration to the next are below defined limits.

corrected boundary node values
: Node values obtained by taking the results produced by CFX-Solver (called "conservative values") and overwriting the results on the boundary nodes with the specified boundary conditions.

  The values of some variables on the boundary nodes (that is, on the edges of the geometry) are not precisely equal to the specified boundary conditions when CFX-Solver finishes its calculations. For instance, the value of velocity on a node on the wall will not be precisely zero, and the value of temperature on an inlet may not be precisely the specified inlet temperature. For visualization purposes, it can be more helpful if the nodes at the boundary do contain the specified boundary conditions and so "corrected boundary node values" are used. Corrected boundary node values are obtained by taking the results produced by CFX-Solver (called "conservative values") and overwriting the results on the boundary nodes with the specified boundary conditions. This will ensure the velocity is display as zero on no-slip walls and equal to the specified inlet velocity on the inlet, for example.

coupled solver
: A solver in which all of the hydrodynamic equations are solved simultaneously as a single system. The advantages of a coupled solver are that it is faster than a traditional solver and fewer iterations are required to obtain a converged solution. CFX-Solver is an example of a coupled solver.

curve
: A general vector valued function of a single parametric variable. In CFX, a line is also a curve. By default, curves are displayed in yellow in ANSYS CFX.

D

default boundary condition
: The boundary condition that is applied to all surfaces that have no boundary condition explicitly set. Normally, this is set to the No Slip Adiabatic Wall boundary condition, although you can change the type of default boundary condition in CFX.

Detached Eddy Simulation (DES)
: A model that covers the boundary layer by a RANS model and switches to a LES model in detached regions.

Direct Numerical Simulation (DNS)
: A CFD simulation in which the Navier-Stokes equations are solved without any turbulence model.

discretization
: The equations of fluid flow cannot be solved directly. Discretization is the process by which the differential equations are converted into

a system of algebraic equations, which relate the value of a variable in a control volume to the value in neighboring control volumes.

domain

Regions of fluid flow and/or heat transfer in CFX are called domains. Fluid domains define a region of fluid flow, while solid domains are regions occupied by conducting solids in which volumetric sources of energy can be specified. The domain requires three specifications:

- The region defining the flow or conducting solid. A domain is formed from one or more 3D primitives that constrain the region occupied by the fluid and/or conducting solids.
- The physical nature of the flow. This determines the modeling of specific features such as heat transfer or buoyancy.
- The properties of the materials in the region.

There can be many domains per model, with each domain defined by separate 3D primitives. Multidomain problems may be created from a single mesh if it contains multiple 3D primitives or is from multiple meshes.

dynamic viscosity

Dynamic viscosity, also called absolute viscosity, is a measure of the resistance of a fluid to shearing forces.

dynamical time

For advection dominated flows, this is an approximate timescale for the flow to move through the Domain. Setting the *physical time step* (p. 288) size to this value (or a fraction of it) can promote faster convergence.

E

eddy viscosity model

A turbulence model based on the assumption that Reynolds stresses are proportional to mean velocity gradients and that the Reynolds stress contribution can be described by the addition of a turbulent component of viscosity. An example of an eddy viscosity model is the k-ϵ model.

edge

The edge entity describes the topological relationships for a curve. Adjacent faces share at least one edge.

emissivity

A property of an object that describes how much radiation it emits as compared to that of a black body at the same temperature.

expansion factor

The rate of growth of volume elements away from curved surfaces and the rate of growth of surface elements away from curved boundaries. Expansion factor is also used to specify the rate of mesh coarsening from a mesh control.

expression editor

An interactive, form-driven facility within CFX for developing expressions.

external flow

A flow field that is located outside of your geometry.

F

face — "Face" can have several meanings:

- A solid face is a surface that exists as part of a solid. It is also known as an implicit surface.
- An element face is one side of a mesh element.
- A boundary face is an element face that exists on the exterior boundary of the domain.
- Surfaces composed of edges that are connected to each other.

FLEXlm — The program that administers ANSYS licensing.

flow boundaries — The surfaces bounding the flow field.

flow region — A volumetric space containing a fluid. Depending on the flow characteristics, you may have a single, uninterrupted flow region, or several flow regions, each exhibiting different characteristics.

flow symmetry — Flow where the conditions of the flow entering and leaving one half of a geometry are the same as the conditions of the flow entering and leaving the other half of the geometry.

fluid — A substance that tends to flow and assumes the shape of its domain, such as a gas in a duct or a liquid in a container.

free edges — Element edges belonging to only one element.

G

gas or liquid surface — A type of boundary that exhibits no friction and fluid cannot move through it. Also called a symmetry boundary.

general fluid — A fluid whose properties may be generally prescribed in ANSYS CFX or FLUENT. Density and specific heat capacity for general fluids may depend on pressure, temperature, and any Additional Variables.

global model tolerance — The minimum distance between two geometry entities below which CFX considers them to be coincident. The default setting of global model tolerance, defined in the template database, is normally .005 in whichever geometry units you are working.

geometric symmetry — The state of a geometry where each half is a mirror of the other.

group — A named collection of geometric and mesh entities that can be posted for display in viewports. The group's definition includes:

- Group name
- Group status (current/not current)
- Group display attributes (modified under Display menu)

- A list of the geometric and mesh entities that are members of the group.

H

hexahedral element
: A mesh element with the same topology as a hexahedron, with six faces and eight vertices.

home directory
: The directory on all Linux systems and some Windows NT systems where each user stores all of their files, and where various setup files are stored.

  However, on some Windows NT systems, users do not have an equivalent to the Linux home directory. In this case, the ANSYS CFX setup file `cfx5rc` can be placed in `c:\winnt\profiles\<user>\Application Data\ANSYS CFX\<release>`, where <*user*> is the user name on the machine. Other files can be put into a directory set by the variable **HOME**.

I

ideal gas
: A fluid whose properties obey the ideal gas law. The density is automatically computed using this relationship and a specified molecular weight.

IGES (Initial Graphics Exchange Specification) file
: An ANSI standard formatted file used to exchange data among most commercial CAD systems. IGES files can be imported into CFX.

implicit geometry
: Geometry that exists as part of some other entity. For example, the edges of a surface are implicit curves.

import mesh
: A meshing mode that enables import of volume meshes generated in one of a number of external CFD packages. The volume mesh can contain hexahedral, tetrahedral, prismatic, and pyramidal element types.

inactive region
: A fluid or porous region where flow and (if relevant) temperatures are not being calculated, or a solid region where temperatures are not being calculated. By default, inactive regions are hidden from view in the graphics window.

incompressible flow
: Flow in which the density is constant throughout the domain.

incremental adaption
: The method of mesh adaption used by CFX where an existing mesh is modified to meet specified criteria. Incremental adaption is much faster than remeshing; however, the mesh quality is limited by that of the initial mesh.

inertial resistance coefficients
: Mathematical terms used to define porous media resistance.

initial guess
: The values of dependent variables at the start of a steady state simulation. These can set explicitly, read from an existing solution, or given default values.

initial values
: The values of dependent variables at the initial time of a transient simulation. These can be either set explicitly, or read from an existing solution.

inlet boundary condition
: A *boundary condition* (p. 278) for which the quantity of fluid flowing into the flow domain is specified, for example, by setting the fluid velocity or mass flow rate.

instancing
: The process of copying an object and applying a positional transform to each of the copies. For example, a row of turbine blades can be visualized by applying instancing to a single blade.

interior boundary
: A boundary that enables flow to enter and exit. These types of boundaries are useful to separate two distinct fluid regions from each other, or to separate a porous region from a fluid region, when you still want flow to occur between the two regions.

internal flow
: Flow through the interior of your geometry, such as flow through a pipe.

interpolation
: The process of transferring a solution from a results file containing one mesh onto a second file containing a different mesh.

isentropic
: The description of a process where there is no heat transfer and entropy is held constant.

isosurface
: A surface of constant value for a given variable.

  A three-dimensional surface that defines a single magnitude of a flow variable such as temperature, pressure, velocity, and so on.

Isovolume
: A locator that consists of a collection of volume elements, all of which take a value of a variable greater than a user-specified value.

J

JPEG file
: A common graphics file type that is supported by CFD-Post output options.

K

k-epsilon turbulence model
: A *turbulence model* (p. 293) based on the concept that turbulence consists of small eddies that are continuously forming and dissipating. The k-epsilon turbulence model solves two additional transport equations: one for turbulence generation (k), and one for turbulence dissipation (epsilon).

kinematic diffusivity

A function of the fluid medium that describes how rapidly an Additional Variable would move through the fluid in the absence of convection.

L

laminar flow

Flow that is dominated by viscous forces in the fluid, and characterized by low Reynolds Number.

A flow field is laminar when the velocity distributions at various points downstream of the fluid entrance are consistent with each other and the fluid particles move in a parallel fashion to each other. The velocity distributions are effectively layers of fluid moving at different velocities relative to each other.

Large Eddy Simulation Model (LES)

The *Large Eddy Simulation model* decomposes flow variables into large and small scale parts. This model solves for large-scale fluctuating motions and uses "sub-grid" scale turbulence models for the small-scale motion.

legend

A color key for any colored plot.

line interface mode

A mode in which you type the commands that would otherwise be issued by the user interface. A viewer is provided that shows the geometry and the objects created on the command line. Line interface mode differs from entering commands in the Command Editor dialog box in that line interface action commands are not preceded by a > symbol. Aside from that difference, all commands that work for the Command Editor dialog box will also work in line interface mode, providing the correct syntax is used.

locator

A place or object upon which a plot can be drawn. Examples are planes and points.

M

MAlt key (Meta key)

The MAlt key (or Meta key) is used to keyboard select menu items with the use of mnemonics (the underscored letter in each menu label). By simultaneously pressing the MAlt key and a mnemonic is an alternative to using the mouse to click a menu title. The MAlt key is different for different brands of keyboards. Some examples of MAlt keys include the " " key for Sun Model Type 4 keyboards, the "Compose Character" key for Tektronix keyboards, and the **Alt** key on most keyboards for most Windows-based systems.

mass fraction

The ration of the mass of a fluid component to the total mass of the fluid. Values for mass fraction range from 0 to 1.

material

A substance with specified properties, such as density and viscosity.

meridional

A term used in FLUENT documentation that is equivalent to the ANSYS CFX term *constant streamwise location*.

mesh
: A collection of points representing the flow field where the equations of fluid motion (and temperature, if relevant) are calculated.

mesh adaption
: The process by which, once or more during a run, the mesh is selectively refined at various locations, depending on criteria that you can specify. As the solution is calculated, the mesh can automatically be refined in locations where solution variables are changed rapidly, in order to resolve the features of the flow in these regions.

  There are two general methods for performing mesh adaption. *Incremental adaption* takes an existing mesh and modifies it to meet the adaption criteria. The alternative is *remeshing*, in which the whole geometry is remeshed at every adaption step according to the adaption criteria. In CFX, incremental adaption is used because this is much faster; however, this imposes the limitation that the resulting mesh quality is limited by the quality of the initial mesh.

mesh control
: A refinement of the surface and volume mesh in specific regions of the model. Mesh controls can take the form of a point, line, or triangle.

meshing mode
: The method you use to create your mesh of nodes and elements required for analysis. There are two main meshing modes:

  - *Advancing Front and Inflation (AFI)* (p. 277)
  - *import mesh* (p. 283)

minimal results file
: A file that contains only the results for selected variables, and no mesh. It can be created only for transient calculations. It is useful when you are only interested in particular variables and want to minimize the size of the results for the transient calculation.

multicomponent fluid
: A fluid consisting of more than one component. The components are assumed to be mixed at the molecular level, though the proportions of each component may vary in space or time. The properties of a multicomponent fluid are dependent on the proportion of constituent components.

N

Navier-Stokes equations
: The fundamental equations of fluid flow and heat transfer, solved by CFX-Solver. They are partial differential equations.

new model preferences
: Preferential settings for your model that define the *meshing mode* (p. 286), the geometry units, and the *global model tolerance* (p. 282).

node allocation parameter
: A parameter that is used in *mesh adaption* (p. 286) to determine how many nodes are added to the mesh in each *adaption step* (p. 277).

non-clipped absolute pressure
: The summation of solver pressure, reference pressure, and hydrostatic pressure (if a buoyant flow). This pressure, used by the solver to calculate cavitation sources, can be negative or positive.

non-Newtonian fluid
: A fluid that does not follow a simple linear relationship between shear stress and shear strain.

normal
: The direction perpendicular to the surface of a mesh element or geometry. The positive direction is determined by the cross-product of the local parametric directions in the surface.

O

open area
: The area in a porous region that is open to flow.

OpenGL
: A graphics display system that is used on a number of different types of computer operating systems.

outlet
: A boundary condition where the fluid is constrained to flow only out of the domain.

outline plot
: A plot showing the outline of the geometry. By setting the edge angle to 0, the surface mesh can be displayed over the whole geometry.

output file
: A text file produced by CFX-Solver that details the history of a run. It is important to browse the output file when a run is finished to determine whether the run has converged, and whether a restart is necessary.

P

parallel runs
: Separate solutions of sections (partitions) of your CFD model, run on more than one processor.

parametric equation
: Any set of equations that express the coordinates of the points of a curve as functions of one parameter, or express the coordinates of the points of a surface as functions of two parameters, or express the coordinates of the points of a solid as functions of three parameters.

parametric solids
: Six-sided solids parameterized in three normalized directions. Parametric solids are colored blue ANSYS CFX.

parametric surfaces
: Four sided surfaces parameterized in two normalized directions. Parametric surfaces are colored green ANSYS CFX.

Particle-Particle Collision Model (LPTM-PPCM)
: A model in ANSYS CFX that takes inter-particle collisions and their effects on the particle and gas phase into consideration.

periodic pair boundary condition
: A boundary condition where the values on the first surface specified are mapped to the second surface. The mapping can be done either by a translation or a rotation (if a rotating frame of reference is used).

physical time step	The time represented in each iteration of the solution.
pick list	The list processor interprets the contents of all selected data boxes. All selected data boxes in CFX expect character strings as input. The character strings may be supplied by the graphics system when you select an entity from a viewport, or you can type or paste in the string directly. The character strings are called "pick lists."
plot	Any means of viewing the results in CFD-Post. Types of plots include vectors, streamlines, and contour plots.
point	An ordered *n*-tuple, where *n* is the number of dimensions of the space in which the point resides.
point probes	Points placed at specific locations in a computational domain where data can be analyzed.
polyline	A locator that consists of user-defined points.
post-processor	The component used to analyze and present the results of the simulation. For ANSYS CFX, the post-processor is CFD-Post.
Power Syntax	The CFX Command Language (CCL) is the internal communication and command language of CFD-Post. It is a simple language that can be used to create objects or perform actions in the post-processor. Power Syntax enables you to embed Perl commands into CCL to achieve powerful quantitative post-processing. Power Syntax programming uses the Perl programming language to allow loops, logic, and custom macros (subroutines). Lines of Power Syntax are identified in a `.ccl` file by an exclamation mark (!) at the start of each line. In between Perl lines, simple syntax lines may refer to Perl variables and lists. For details, see *Power Syntax in ANSYS CFX* (p. 229).
pre-processor	The component used to create the input for the solver. For ANSYS CFX, the pre-processor is CFX-Pre.
pressure	In the cavitation model, pressure is the same as solver pressure, but clipped such that the absolute pressure is non-negative. It is used for post-processing only.
prism or prismatic element	A 3D mesh element shaped like a triangular prism (with six vertices). Sometimes known as a wedge element.
PVM (Parallel Virtual Machine)	The environment that controls parallel processes.
PVMHosts file	The database file containing information about where ANSYS CFX, and consequently PVM, have been installed on each PVM node. It is consulted when the Parallel Virtual Machine is started to determine where PVM is located on each slave node.
pyramid element	A 3D mesh element that has five vertices.

R

reference coordinate frame

The coordinate frame in which the principal directions of X or Y or Z are taken. X is taken in the local X of that frame, and so on. If the coordinate frame is a non-rectangular coordinate frame, then the principal axes 1, 2, and 3 will be used to define the X, Y, and Z directions, respectively. The default is CFX global system (Coord 0).

For domains, boundary conditions, and initial values, the reference coordinate frame is always treated as Cartesian, irrespective of coordinate frame type.

region

An area comprised of a fluid, a solid material, or a porous material.

residuals

The change in the value of certain variables from one iteration to the next.

The discretized *Navier-Stokes equations* (p. 286) are solved iteratively. The residual for each equation gives a measure of how far the latest solution is from the solution in the previous iteration. A solution is considered to be converged when the residuals are below a certain value.

CFX-Solver writes the residuals to the *output file* (p. 287) so that they can be reviewed. FLUENT allows residuals to be plotted during the solution process.

results file (CFX-Solver Results file)

A file produced by CFX-Solver that contains the full definition of the simulation as well as the values of all variables throughout the flow domain and the history of the run including *residuals* (p. 289). A CFX-Solver Results file can be used as input to CFD-Post or as an input file to CFX-Solver, in order to perform a restart.

Reynolds averaged Navier-Stokes (RANS) equations

Time-averaged equations of fluid motion that are primarily used with turbulent flows.

Reynolds stress

The stress added to fluid flow due to the random fluctuations in fluid momentum in turbulent flows. When the *Navier-Stokes equations* (p. 286) are derived for time averaged turbulent flow to take into account the effect of these fluctuations in velocity, the resulting equations have six stress terms that do not appear in the laminar flow equations. These are known as Reynolds stresses.

Reynolds stress turbulence model

A model that solves transport equations for the individual Reynolds stress components. It is particularly appropriate where strong flow curvature, swirl, and separation are present. Reynolds stress models in general tend to be less numerically robust than eddy viscosity models such as the *k-epsilon turbulence model* (p. 284).

RNG k-epsilon turbulence model

An alternative to the standard *k-epsilon turbulence model* (p. 284). It is based on renormalization group analysis of the Navier-Stokes equations. The transport equations for turbulence generation and dissipation are the same as those for the standard k-epsilon model,

but the model constants differ, and the constant $C\epsilon 1$ is replaced by the function $C\epsilon 1RNG$.

Rotating Frame of Reference (RFR)
: A coordinate system that rotates. ANSYS CFX and FLUENT can solve for fluid flow in a geometry that is rotating around an axis at a fixed angular velocity.

run
: A process that requires the specification of the CFX-Solver input file (and an initial values file, if necessary), and produces an output file and a results file (if successful).

S

Sampling Plane
: A locator that is planar and consists of equally-spaced points.

scalar variable
: A variable that has only magnitude and not direction. Examples are temperature, pressure, speed (the magnitude of the velocity vector), and any component of a vector quantity.

Scale Adaptive Simulation (SAS) model
: A shear stress transport model used primarily for unsteady CFD simulations, where steady-state simulations are not of sufficient accuracy and do not properly describe the true nature of the physical phenomena. Cases that may benefit from using the SAS-SST model include:

  - Unsteady flow behind a car or in the strong mixing behind blades and baffles inside stirred chemical reactors
  - Unsteady cavitation inside a vortex core (fuel injection system) or a fluid-structure interaction (unsteady forces on bridges, wings, and so on).

  For these problems and others, the SAS-SST model provides a more accurate solution than URANS models, where steady-state simulations are not of sufficient accuracy and do not properly describe the true nature of the physical phenomena.

Second Moment Closure models
: Models that use seven transport equations for the independent Reynolds stresses and one length (or related) scale; other models use two equations for the two main turbulent scales.

session file (CFX)
: A file that contains the records of all the actions in each interactive CFX session. It has the extension `.ses`.

Shear Stress Transport (SST)
: A $k-\omega$ based SST model that accounts for the transport of the turbulent shear stress and gives highly accurate predictions of the onset and the amount of flow separation under adverse pressure gradients.

singleton (CCL object)
: A singleton object that consists of an object type at the start of a line, followed by a : (colon). Subsequent lines may define parameters and child objects associated with this object. The object definition is terminated by the string END on a line by itself. The singleton object for a session file is declared like this:

```
SESSION:
     Session Filename = <filename>.cse
END
```

The difference between a singleton object and a named object is that after the data has been processed, a singleton can appear just once as the child of a parent object. However, there may be several instances of a named object of the same type defined with different names.

slice plane

A locator that is planar, and that consists of all the points that intersect the plane and the mesh edges.

solid

A material that does not flow when a force or stress is applied to it.

The general class of vector valued functions of three parametric variables.

solid sub-domain

A region of the fluid domain that is occupied by a conducting solid. ANSYS CFX can model heat transfer in such a solid; this is known as *CHT (Conjugate Heat Transfer)* (p. 279).

solver

The component that solves the CFD problem, producing the required results.

solver pressure

The pressure calculated by solving conservative equations; it can be negative or positive. In the `.out` file it is called Pressure.

spanwise coordinate

A term used in FLUENT documentation that is equivalent to the ANSYS CFX term *constant span*.

specific heat

The ratio of the amount of heat energy supplied to a substance to its corresponding change in temperature.

specific heat capacity

The amount of heat energy required to raise the temperature of a fixed mass of a fluid by 1 K at constant pressure.

speed of sound

The velocity at which small amplitude pressure waves propagate through a fluid.

sphere volume

A locator that consists of a collection of volume elements that are contained in or intersect a user-defined sphere.

state files

Files produced by CFD-Post that contain CCL commands. They differ from session files in that only a snapshot of the current state is saved to a file. You can also write your own state files using any text editor.

STP (Standard Temperature and Pressure)

Defined as 0°C (273.15 K) and 1 atm (1.013x105 Pa).

steady-state simulation

A simulation that is carried out to determine the flow after it has settled to a steady state. Note that, even with time constant boundary conditions, some flows do not have a steady-state solution.

stream plot

A plot that shows the streamlines of a flow. Stream plots can be shown as lines, tubes, or ribbons.

streamline — The path that a small, neutrally-buoyant particle would take through the flow domain, assuming the displayed solution to be steady state.

subdomains — Regions comprising a solid or set of solids, within the region of bounding solids for a fluid domain, that allow the prescription of momentum and energy sources. They can be used to model regions of flow resistance and heat source.

subsonic flow — The movement of a fluid at a speed less than the speed of sound.

surface plot — A plot that colors a surface according to the values of a variable. Additionally, you can choose to display contours.

symmetry-plane boundary condition — A boundary condition where all variables except velocity are mathematically symmetric and there can be no diffusion or flow across the boundary. Velocity parallel to the boundary is also symmetric and velocity normal to the boundary is zero.

T

template fluid — One of a list of standard fluids with predefined properties that you can use 'as is', or use as a template to create a fluid with your own properties.

thermal conductivity — The property of a fluid that characterizes its ability to transfer heat by conduction.

A property of a substance that indicates its ability to transfer thermal energy between adjacent portions of the substance.

thermal expansivity — The property of a fluid that describes how a fluid expands as the result of an increase in temperature. Also known as the coefficient of thermal expansion, β.

theta — The angular coordinate measured about the axis of rotation following the right-hand rule. When looking along the positive direction of the axis of rotation, theta is increasing in the clockwise direction. Note that the theta coordinate in CFD-Post does not increase over 360°, even for spiral geometries that wrap to more than 360°.

topology — The shape, node, edge, and face numbering of an element.

tracers — Particles that follow a flow pathline. Used in viewing CFD results in order to visualize the mechanics of the fluid flow.

transitions — Portions of a mesh that are the result of meshing geometry with two opposing edges that have different mesh seeds. This produces an irregular mesh.

turbulence intensity — The ratio of the root-mean-square of the velocity fluctuations to the mean flow velocity.

A turbulence intensity of 1% or less is generally considered low and turbulence intensities greater than 10% are considered

high. Ideally, you will have a good estimate of the turbulence intensity at the inlet boundary from external, measured data. For example, if you are simulating a wind-tunnel experiment, the turbulence intensity in the free stream is usually available from the tunnel characteristics. In modern low-turbulence wind tunnels, the free-stream turbulence intensity may be as low as 0.05%.

For internal flows, the turbulence intensity at the inlets is totally dependent on the upstream history of the flow. If the flow upstream is under-developed and undisturbed, you can use a low turbulence intensity. If the flow is fully developed, the turbulence intensity may be as high as a few percent.

turbulence length scale

A physical quantity related to the size of the large eddies that contain the energy in turbulent flows.

In fully-developed duct flows, the turbulence length scale is restricted by the size of the duct because the turbulent eddies cannot be larger than the duct. An approximate relationship can be made between the turbulence length scale and the physical size of the duct that, while not ideal, can be applied to most situations.

If the turbulence derives its characteristic length from an obstacle in the flow, such as a perforated plate, it is more appropriate to base the turbulence length scale on the characteristic length of the obstacle rather than on the duct size.

turbulence model

A model that predicts *turbulent flow* (p. 293). The available turbulence models in ANSYS CFX are:

- *k-epsilon turbulence model* (p. 284)
- *RNG k-epsilon turbulence model* (p. 289)
- *Reynolds stress turbulence model* (p. 289)
- *zero equation turbulence model* (p. 295)

Turbulence models allow a steady state representation of (inherently unsteady) turbulent flow to be obtained.

turbulent

A flow field that is irregular and chaotic look. In turbulent flow, a fluid particle's velocity changes dramatically at any given point in the flow field, in time, direction, and magnitude, making computational analysis of the flow more challenging.

turbulent flow

Flow that is randomly unsteady over time. A characteristic of turbulent flow is chaotic fluctuations in the local velocity.

V

variable

A quantity such as temperature or velocity for which results have been calculated in a CFD calculation.

See also *Additional Variable* (p. 277).

vector plot

A plot that shows the direction of the flow at points in space, using arrows. Optionally, the size of the arrows may show the magnitude of the velocity of the flow at that point. The vectors may also be colored according to the value of any variable.

verification

A check of the model for validity and correctness.

viewer area

The area of ANSYS CFX that contains the 3D Viewer, **Table Viewer**, **Chart Viewer**, **Comment Viewer**, and **Report Viewer**, which you access from tabs at the bottom of the area.

viewport (CFX)

An assigned, named, graphics window definition, stored in the CFX database, that can be used to display selected portions of a model's geometry, finite elements, and analysis results. The viewport's definition includes:

- The viewport name
- The status of the viewport (posted or unposted; current or not current)
- Viewport display attributes
- A definition of the current view
- A current group
- A list of the posted groups for display
- A graphics environment accessed from Display, Preference, and Group menus that is common to all viewports.

There are the following types of CFX viewports:

current viewport
The viewport currently being displayed. The following actions can be performed only on the current viewport:

- Changing the view by using the **View** menu or mouse.
- Posting titles and annotations by using the **Display** menu.

posted viewport
A viewport that has been selected for display.

target viewport
A viewport selected for a viewport modify action. Any viewport (including the current viewport) can be selected as the target viewport.

viscosity

The ratio of the tangential frictional force per unit area to the velocity gradient perpendicular to the flow direction.

viscous resistance coefficients

A term to define porous media resistance.

Volume of Fluid (VOF) method

A technique for tracking a fluid-fluid interface as it changes its topology.

W

wall

A generic term describing a stationary boundary through which flow cannot pass.

workspace area

The area of CFX-Pre and CFD-Post that contains the **Outline**, **Variables**, **Expressions**, **Calculators**, and **Turbo** workspaces, which you access from the tabs at the top of the area. Each workspace has a tree view at the top and an editor at the bottom (which is often called the details view).

See also "CFD-Post Graphical Interface".

Y

y+ (YPLUS)

A non-dimensional parameter used to determine a specific distance from a wall through the boundary layer to the center of the element at a wall boundary.

Z

zero equation turbulence model

A simple model that accounts for turbulence by using an algebraic equation to calculate turbulence viscosity. This model is useful for obtaining quick, robust solutions for use as initial fields for simulations using more sophisticated turbulence models.

Index

Symbols

A

B

C

D

E

F

G

H

I

K

L

M

N

O

P

Q

R

S

T

U

V

W

Y

Z