OpenFOAM

The Open Source CFD Toolbox

User Guide

Version v2112
20th December 2021

Typeset in LaTeX.

License

1. Definitions

a. **"Adaptation"** means a work based upon the Work, or upon the Work and other pre-existing works, such as a translation, adaptation, derivative work, arrangement of music or other alterations of a literary or artistic work, or phonogram or performance and includes cinematographic adaptations or any other form in which the Work may be recast, transformed, or adapted including in any form recognizably derived from the original, except that a work that constitutes a Collection will not be considered an Adaptation for the purpose of this License. For the avoidance of doubt, where the Work is a musical work, performance or phonogram, the synchronization of the Work in timed-relation with a moving image ("synching") will be considered an Adaptation for the purpose of this License.

b. **"Collection"** means a collection of literary or artistic works, such as encyclopedias and anthologies, or performances, phonograms or broadcasts, or other works or subject matter other than works listed in Section 1(f) below, which, by reason of the selection and arrangement of their contents, constitute intellectual creations, in which the Work is included in its entirety in unmodified form along with one or more other contributions, each constituting separate and independent works in themselves, which together are assembled into a collective whole. A work that constitutes a Collection will not be considered an Adaptation (as defined above) for the purposes of this License.

c. **"Distribute"** means to make available to the public the original and copies of the Work through sale or other transfer of ownership.

d. **"Licensor"** means the individual, individuals, entity or entities that offer(s) the Work under the terms of this License.

e. **"Original Author"** means, in the case of a literary or artistic work, the individual, individuals, entity or entities who created the Work or if no individual or entity can be identified, the publisher; and in addition (i) in the case of a performance the actors, singers, musicians, dancers, and other persons who act, sing, deliver, declaim, play in, interpret or otherwise perform literary or artistic works or expressions of folklore; (ii) in the case of a phonogram the producer being the person or legal entity who first fixes the sounds of a performance or other sounds; and, (iii) in the case of broadcasts, the organization that transmits the broadcast.

f. **"Work"** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are

assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

g. **"You"** means an individual or entity exercising rights under this License who has not previously violated the terms of this License with respect to the Work, or who has received express permission from the Licensor to exercise rights under this License despite a previous violation.

h. **"Publicly Perform"** means to perform public recitations of the Work and to communicate to the public those public recitations, by any means or process, including by wire or wireless means or public digital performances; to make available to the public Works in such a way that members of the public may access these Works from a place and at a place individually chosen by them; to perform the Work to the public by any means or process and the communication to the public of the performances of the Work, including by public digital performance; to broadcast and rebroadcast the Work by any means including signs, sounds or images.

i. **"Reproduce"** means to make copies of the Work by any means including without limitation by sound or visual recordings and the right of fixation and reproducing fixations of the Work, including storage of a protected performance or phonogram in digital form or other electronic medium.

2. Fair Dealing Rights

Nothing in this License is intended to reduce, limit, or restrict any uses free from copyright or rights arising from limitations or exceptions that are provided for in connection with the copyright protection under copyright law or other applicable laws.

3. License Grant

Subject to the terms and conditions of this License, Licensor hereby grants You a worldwide, royalty-free, non-exclusive, perpetual (for the duration of the applicable copyright) license to exercise the rights in the Work as stated below:

a. to Reproduce the Work, to incorporate the Work into one or more Collections, and to Reproduce the Work as incorporated in the Collections; and,

b. to Distribute and Publicly Perform the Work including as incorporated in Collections.

The above rights may be exercised in all media and formats whether now known or hereafter devised. The above rights include the right to make such modifications as are technically necessary to exercise the rights in other media and formats, but otherwise you have no rights to make Adaptations. Subject to 8(f), all rights not expressly granted by Licensor are hereby reserved, including but not limited to the rights set forth in Section 4(d).

4. Restrictions

The license granted in Section 3 above is expressly made subject to and limited by the following restrictions:

a. You may Distribute or Publicly Perform the Work only under the terms of this License. You must include a copy of, or the Uniform Resource Identifier (URI) for, this License with every copy of the Work You Distribute or Publicly Perform. You may not offer or impose any terms on the Work that restrict the terms of this License or the ability of the recipient of the Work to exercise the rights granted to that recipient under the terms of the License. You may not sublicense the Work. You must keep intact all notices that refer to this License and to the disclaimer of warranties with every copy of the Work You Distribute or Publicly Perform. When You Distribute or Publicly Perform the Work, You may not impose any effective technological measures on the Work that restrict the ability of a recipient of the Work from You to exercise the rights granted to that recipient under the terms of the License. This Section 4(a) applies to the Work as incorporated in a Collection, but this does not require the Collection apart from the Work itself to be made subject to the terms of this License. If You create a Collection, upon notice from any Licensor You must, to the extent practicable, remove from the Collection any credit as required by Section 4(c), as requested.

b. You may not exercise any of the rights granted to You in Section 3 above in any manner that is primarily intended for or directed toward commercial advantage or private monetary compensation. The exchange of the Work for other copyrighted works by means of digital file-sharing or otherwise shall not be considered to be intended for or directed toward commercial advantage or private monetary compensation, provided there is no payment of any monetary compensation in connection with the exchange of copyrighted works.

c. If You Distribute, or Publicly Perform the Work or Collections, You must, unless a request has been made pursuant to Section 4(a), keep intact all copyright notices for the Work and provide, reasonable to the medium or means You are utilizing: (i) the name of the Original Author (or pseudonym, if applicable) if supplied, and/or if the Original Author and/or Licensor designate another party or parties (e.g., a sponsor institute, publishing entity, journal) for attribution ("Attribution Parties") in Licensor's copyright notice, terms of service or by other reasonable means, the name of such party or parties; (ii) the title of the Work if supplied; (iii) to the extent reasonably practicable, the URI, if any, that Licensor specifies to be associated with the Work, unless such URI does not refer to the copyright notice or licensing information for the Work. The credit required by this Section 4(c) may be implemented in any reasonable manner; provided, however, that in the case of a Collection, at a minimum such credit will appear, if a credit for all contributing authors of Collection appears, then as part of these credits and in a manner at least as prominent as the credits for the other contributing authors. For the avoidance of doubt, You may only use the credit required by this Section for the purpose of attribution in the manner set out above and, by exercising Your rights under this License, You may not implicitly or explicitly assert or imply any connection with, sponsorship or endorsement by the Original Author, Licensor and/or Attribution Parties, as appropriate, of You or Your use of the Work, without the separate, express prior written permission of the Original Author, Licensor and/or Attribution Parties.

d. For the avoidance of doubt:

   i. **Non-waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme cannot be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License;

   ii. **Waivable Compulsory License Schemes**. In those jurisdictions in which the right to collect royalties through any statutory or compulsory licensing scheme can be waived, the Licensor reserves the exclusive right to collect such royalties for any exercise by You of the rights granted under this License if Your exercise of such rights is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(b) and otherwise waives the right to collect royalties through any statutory or compulsory licensing scheme; and,

iii. **Voluntary License Schemes**. The Licensor reserves the right to collect royalties, whether individually or, in the event that the Licensor is a member of a collecting society that administers voluntary licensing schemes, via that society, from any exercise by You of the rights granted under this License that is for a purpose or use which is otherwise than noncommercial as permitted under Section 4(b).

e. Except as otherwise agreed in writing by the Licensor or as may be otherwise permitted by applicable law, if You Reproduce, Distribute or Publicly Perform the Work either by itself or as part of any Collections, You must not distort, mutilate, modify or take other derogatory action in relation to the Work which would be prejudicial to the Original Author's honor or reputation.

5. Representations, Warranties and Disclaimer

UNLESS OTHERWISE MUTUALLY AGREED BY THE PARTIES IN WRITING, LICENSOR OFFERS THE WORK AS-IS AND MAKES NO REPRESENTATIONS OR WARRANTIES OF ANY KIND CONCERNING THE WORK, EXPRESS, IMPLIED, STATUTORY OR OTHERWISE, INCLUDING, WITHOUT LIMITATION, WARRANTIES OF TITLE, MERCHANTIBILITY, FITNESS FOR A PARTICULAR PURPOSE, NONINFRINGEMENT, OR THE ABSENCE OF LATENT OR OTHER DEFECTS, ACCURACY, OR THE PRESENCE OF ABSENCE OF ERRORS, WHETHER OR NOT DISCOVERABLE. SOME JURISDICTIONS DO NOT ALLOW THE EXCLUSION OF IMPLIED WARRANTIES, SO SUCH EXCLUSION MAY NOT APPLY TO YOU.

6. Limitation on Liability

EXCEPT TO THE EXTENT REQUIRED BY APPLICABLE LAW, IN NO EVENT WILL LICENSOR BE LIABLE TO YOU ON ANY LEGAL THEORY FOR ANY SPECIAL, INCIDENTAL, CONSEQUENTIAL, PUNITIVE OR EXEMPLARY DAMAGES ARISING OUT OF THIS LICENSE OR THE USE OF THE WORK, EVEN IF LICENSOR HAS BEEN ADVISED OF THE POSSIBILITY OF SUCH DAMAGES.

7. Termination

a. This License and the rights granted hereunder will terminate automatically upon any breach by You of the terms of this License. Individuals or entities who have received Collections from You under this License, however, will not have their licenses terminated provided such individuals or entities remain in full compliance with those licenses. Sections 1, 2, 5, 6, 7, and 8 will survive any termination of this License.

b. Subject to the above terms and conditions, the license granted here is perpetual (for the duration of the applicable copyright in the Work). Notwithstanding the above, Licensor reserves the right to release the Work under different license terms or to stop distributing the Work at any time; provided, however that any such election will not serve to withdraw this License (or any other license that has been, or is required to be, granted under the terms of this License), and this License will continue in full force and effect unless terminated as stated above.

8. Miscellaneous

a. Each time You Distribute or Publicly Perform the Work or a Collection, the Licensor offers to the recipient a license to the Work on the same terms and conditions as the license granted

to You under this License.

b. If any provision of this License is invalid or unenforceable under applicable law, it shall not affect the validity or enforceability of the remainder of the terms of this License, and without further action by the parties to this agreement, such provision shall be reformed to the minimum extent necessary to make such provision valid and enforceable.

c. No term or provision of this License shall be deemed waived and no breach consented to unless such waiver or consent shall be in writing and signed by the party to be charged with such waiver or consent.

d. This License constitutes the entire agreement between the parties with respect to the Work licensed here. There are no understandings, agreements or representations with respect to the Work not specified here. Licensor shall not be bound by any additional provisions that may appear in any communication from You. This License may not be modified without the mutual written agreement of the Licensor and You.

e. The rights granted under, and the subject matter referenced, in this License were drafted utilizing the terminology of the Berne Convention for the Protection of Literary and Artistic Works (as amended on September 28, 1979), the Rome Convention of 1961, the WIPO Copyright Treaty of 1996, the WIPO Performances and Phonograms Treaty of 1996 and the Universal Copyright Convention (as revised on July 24, 1971). These rights and subject matter take effect in the relevant jurisdiction in which the License terms are sought to be enforced according to the corresponding provisions of the implementation of those treaty provisions in the applicable national law. If the standard suite of rights granted under applicable copyright law includes additional rights not granted under this License, such additional rights are deemed to be included in the License; this License is not intended to restrict the license of any rights under applicable law.

Trademarks

Contents

Chapter 1

Introduction

This guide accompanies the release of version v2112 of the Open Source Field Operation and Manipulation (OpenFOAM) C++ libraries. It provides a description of the operation of OpenFOAM including case set-up, the wide-ranging functionality available, followed by running applications and post-processing the results.

OpenFOAM is first and foremost a *C++ library*, used primarily to create executables, known as *applications*. The applications fall into two categories: *solvers*, that are each designed to solve a specific problem in continuum mechanics; and *utilities*, that are designed to perform tasks that involve data manipulation. New solvers and utilities can be created by its users with some pre-requisite knowledge of the underlying method, physics and programming techniques involved.

OpenFOAM is supplied with pre- and post-processing environments. The interface to the pre- and post-processing are themselves OpenFOAM utilities, thereby ensuring consistent data handling across all environments. The overall structure of OpenFOAM is shown in Figure 1.1.

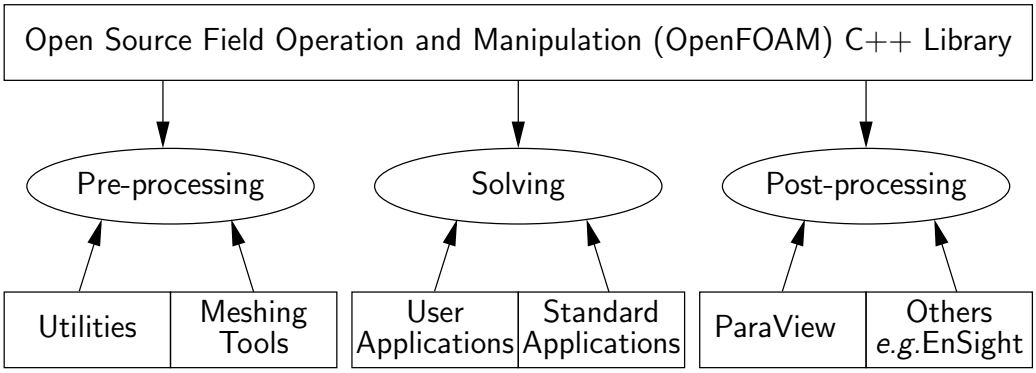


Figure 1.1: Overview of OpenFOAM structure.

The file structure of OpenFOAM cases is described in chapter 2, with examples of the syntax and file format required when specifying various input quantities.

Running OpenFOAM applications is presented in chapter 3 for serial operation and the additional steps required for parallel operation.

Mesh generation is described in chapter 4 using the mesh generators supplied with OpenFOAM and conversion of mesh data generated by third-party products.

Chapter 5 provides details of the numerous models offered by OpenFOAM, including boundary conditions, thermophysical and turbulence models.

Solving OpenFOAM cases is presented in chapter 6, including descriptions of numerical schemes, case control, and solution monitoriing.

Post-processing is described in chapter 7.

Comprehensive reference lists for the avaliable solvers, utilities, libraries and boundary conditions are available in appendix A

Chapter 2

OpenFOAM cases

This chapter deals with the file structure and organisation of OpenFOAM cases. Normally, a user would assign a name to a case, *e.g.* the tutorial case of flow in a cavity is simply named cavity. This name becomes the name of a directory in which all the case files and subdirectories are stored. The case directories themselves can be located anywhere but we recommend they are within a *run* subdirectory of the user's project directory, *i.e.* *$HOME/OpenFOAM/${USER}-v2112*.

One advantage of this is that the $FOAM_RUN environment variable is set to the directory *$HOME/OpenFOAM/${USER}-v2112/run* by default; the user can quickly move to that directory by executing a preset alias, `run`, at the command line.

It is suggested that beginners in OpenFOAM start their journey with the tutorials. Each solver has at least one tutorial which shows its use. These are located in the *$FOAM_TUTORIALS* directory, reached quickly by executing the `tut` alias at the command line. The tutorials directory structure mimics the solvers structure for easier navigation. Selected tutorials are described in the Tutorial Guide, and further community-driven content available from http://wiki.openfoam.com.

2.1 File structure of OpenFOAM cases

The basic directory structure for a OpenFOAM case, that contains the minimum set of files required to run an application, is shown in Figure 2.1 and described as follows:

A *constant* directory that contains a full description of the case mesh in a subdirectory *polyMesh* and files specifying physical properties for the application concerned, *e.g.* *transportProperties*.

A *system* directory for setting parameters associated with the solution procedure itself. It contains *at least* the following 3 files: *controlDict* where run control parameters are set including start/end time, time step and parameters for data output; *fvSchemes* where discretisation schemes used in the solution may be selected at run-time; and, *fvSolution* where the equation solvers, tolerances and other algorithm controls are set for the run.

The 'time' directories containing individual files of data for particular fields. The data can be: either, initial values and boundary conditions that the user must specify to define the problem; or, results written to file by OpenFOAM. Note that the OpenFOAM fields must always be initialised, even when the solution does not strictly require it, as in steady-state problems. The name of each time directory is based on the simulated time at which the data is written and is described fully in

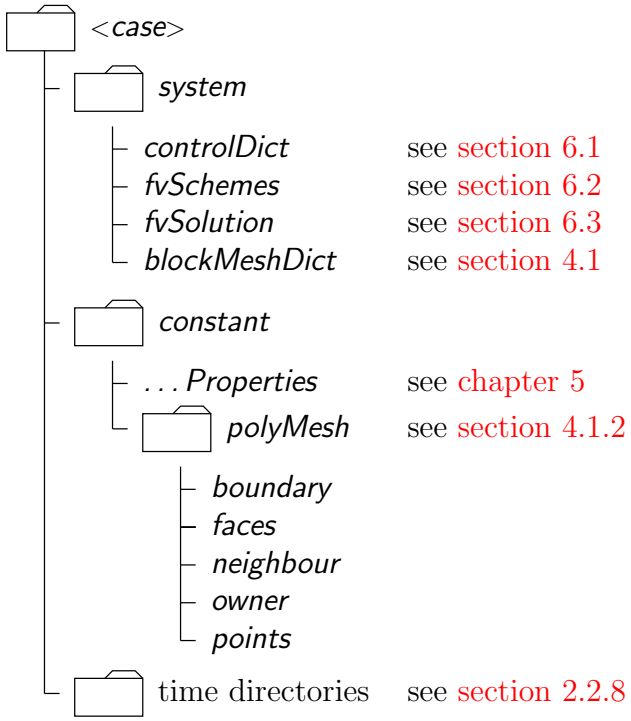


Figure 2.1: Case directory structure

section 6.1. It is sufficient to say now that since we usually start our simulations at time $t = 0$, the initial conditions are usually stored in a directory named *0* or *0.000000e+00*, depending on the name format specified. For example, in the cavity tutorial, the velocity field **U** and pressure field p are initialised from files *0/U* and *0/p* respectively.

2.2 Basic input/output file format

OpenFOAM needs to read a range of data structures such as strings, scalars, vectors, tensors, lists and fields. The input/output (I/O) format of files is designed to be extremely flexible to enable the user to modify the I/O in OpenFOAM applications as easily as possible. The I/O follows a simple set of rules that make the files extremely easy to understand, in contrast to many software packages whose file format may not only be difficult to understand intuitively but also not be published anywhere. The description of the OpenFOAM file format is described in the following sections.

2.2.1 General syntax rules

The format follows some of the general principles of C++ source code.

- Files have free form, with no particular meaning assigned to any column and no need to indicate continuation across lines.
- Lines have no particular meaning except to a `//` comment delimiter which makes OpenFOAM ignore any text that follows it until the end of line.
- A comment over multiple lines is done by enclosing the text between `/*` and `*/` delimiters.

2.2.2 Dictionaries

OpenFOAM uses *dictionaries* as the most common means of specifying data. A dictionary is an entity that contains a set of data entries that can be retrieved by the I/O by means of *keywords*. The keyword entries follow the general format

```
<keyword>  <dataEntry1> ... <dataEntryN>;
```

Most entries are single data entries of the form:

```
<keyword>  <dataEntry>;
```

Most OpenFOAM data files are themselves dictionaries containing a set of keyword entries. Dictionaries provide the means for organising entries into logical categories and can be specified hierarchically so that any dictionary can itself contain one or more dictionary entries. The format for a dictionary is to specify the dictionary name followed the entries enclosed in curly braces {} as follows

```
<dictionaryName>
{
    ... keyword entries ...
}
```

2.2.3 The data file header

All data files that are read and written by OpenFOAM begin with a dictionary named `FoamFile` containing a standard set of keyword entries, listed in Table 2.1. The table

Keyword	Description	Entry
`version`	I/O format version	`2.0`
`format`	Data format	`ascii` / `binary`
`arch`	Architecture and compilation options	"LSB;label=32;scalar=64"
`class`	OpenFOAM class constructed from the data file concerned	typically `dictionary` or a field, *e.g.*`volVectorField`
`location`	Path to the file, in `"..."`	(optional)
`object`	Filename	*e.g.*`controlDict`

Table 2.1: Header keywords entries for data files.

provides brief descriptions of each entry, which is probably sufficient for most entries with the notable exception of `class`. The `class` entry is the name of the C++ class in the OpenFOAM library that will be constructed from the data in the file. Without knowledge of the underlying code which calls the file to be read, and knowledge of the OpenFOAM classes, the user will probably be unable to surmise the `class` entry correctly. However, most data files with simple keyword entries are read into an internal dictionary class and therefore the `class` entry is `dictionary` in those cases.

The following example shows the use of keywords to provide data for a case using the types of entry described so far. The extract, from an *fvSolution* dictionary file, contains 2 dictionaries, *solvers* and *PISO*. The *solvers* dictionary contains multiple data entries for solver and tolerances for each of the pressure and velocity equations, represented by the `p` and `U` keywords respectively; the *PISO* dictionary contains algorithm controls.

```
17  solvers
18  {
19      p
20      {
21          solver          PCG;
22          preconditioner  DIC;
23          tolerance       1e-06;
24          relTol          0.05;
25      }
26
27      pFinal
28      {
29          $p;
30          relTol          0;
31      }
32
33      U
34      {
35          solver          smoothSolver;
36          smoother        symGaussSeidel;
37          tolerance       1e-05;
38          relTol          0;
39      }
40  }
41
42  PISO
43  {
44      nCorrectors     2;
45      nNonOrthogonalCorrectors 0;
46      pRefCell        0;
47      pRefValue       0;
48  }
49
50
51  // ************************************************************************* //
```

2.2.4 Lists

OpenFOAM applications contain lists, *e.g.* a list of vertex coordinates for a mesh description. Lists are commonly found in I/O and have a format of their own in which the entries are contained within round braces `( )`. There is also a choice of format preceeding the round braces:

simple the keyword is followed immediately by round braces

```
<listName>
(
    ... entries ...
);
```

numbered the keyword is followed by the number of elements `<n>` in the list

```
<listName>
<n>
(
    ... entries ...
);
```

token identifier the keyword is followed by a class name identifier Label<Type> where <Type> states what the list contains, *e.g.* for a list of scalar elements is

```
<listName>
List<scalar>
<n>        // optional
(
    ... entries ...
);
```

Note that <scalar> in List<scalar> is not a generic name but the actual text that should be entered.

The simple format is a convenient way of writing a list. The other formats allow the code to read the data faster since the size of the list can be allocated to memory in advance of reading the data. The simple format is therefore preferred for short lists, where read time is minimal, and the other formats are preferred for long lists.

2.2.5 Scalars, vectors and tensors

A scalar is a single number represented as such in a data file. A vector is a VectorSpace of rank 1 and dimension 3, and since the number of elements is always fixed to 3, the simple List format is used. Therefore a vector $(1.0, 1.1, 1.2)$ is written:

```
(1.0 1.1 1.2)
```

In OpenFOAM, a tensor is a VectorSpace of rank 2 and dimension 3 and therefore the data entries are always fixed to 9 real numbers. Therefore the identity tensor can be written:

```
(
    1 0 0
    0 1 0
    0 0 1
)
```

This example demonstrates the way in which OpenFOAM ignores the line return is so that the entry can be written over multiple lines. It is treated no differently to listing the numbers on a single line:

```
( 1 0 0 0 1 0 0 0 1 )
```

2.2.6 Dimensional units

In continuum mechanics, properties are represented in some chosen units, *e.g.* mass in kilograms (kg), volume in cubic metres (m^3), pressure in Pascals ($kg\,m^{-1}\,s^{-2}$). Algebraic operations must be performed on these properties using consistent units of measurement; in particular, addition, subtraction and equality are only physically meaningful for properties of the same dimensional units. As a safeguard against implementing a meaningless operation, OpenFOAM attaches dimensions to field data and physical properties and performs dimension checking on any tensor operation.

The I/O format for a dimensionSet is 7 scalars delimited by square brackets, *e.g.*

```
[0 2 -1 0 0 0 0]
```

where each of the values corresponds to the power of each of the base units of measurement listed in Table 2.2. The table gives the base units for the Système International (SI) and the United States Customary System (USCS) but OpenFOAM can be used with any system of units. All that is required is that the *input data is correct for the chosen set of units*. It is particularly important to recognise that OpenFOAM requires some dimensioned physical constants, *e.g.* the Universal Gas Constant R, for certain calculations, *e.g.* thermophysical modelling. These dimensioned constants are specified in

No.	Property	SI unit	USCS unit
1	Mass	kilogram (kg)	pound-mass (lbm)
2	Length	metre (m)	foot (ft)
3	Time	— — — — second (s)	— — — —
4	Temperature	Kelvin (K)	degree Rankine (°R)
5	Quantity	kilogram-mole (kgmol)	pound-mole (lbmol)
6	Current	— — — — ampere (A)	— — — —
7	Luminous intensity	— — — — candela (cd)	— — — —

Table 2.2: Base units for SI and USCS

a *DimensionedConstant* sub-dictionary of main *controlDict* file of the OpenFOAM installation (*$WM_PROJECT_DIR/etc/controlDict*). By default these constants are set in SI units. Those wishing to use the USCS or any other system of units should modify these constants to their chosen set of units accordingly.

2.2.7 Dimensioned types

Physical properties are typically specified with their associated dimensions. These entries have the format that the following example of a dimensionedScalar demonstrates:

```
nu              [0 2 -1 0 0 0 0]  1;
```

The first `nu` is the keyword; the next entry is the dimensionSet and the final entry is the scalar value.

2.2.8 Fields

Much of the I/O data in OpenFOAM are tensor fields, *e.g.* velocity, pressure data, that are read from and written into the time directories. OpenFOAM writes field data using keyword entries as described in Table 2.3.

Keyword	Description	Example
`dimensions`	Dimensions of field	`[1 1 -2 0 0 0 0]`
`internalField`	Value of internal field	`uniform (1 0 0)`
`boundaryField`	Boundary field	see file listing in section 2.2.8

Table 2.3: Main keywords used in field dictionaries.

The data begins with an entry for its `dimensions`. Following that, is the `internalField`, described in one of the following ways.

Uniform field a single value is assigned to all elements within the field, taking the form:

```
internalField uniform <entry>;
```

Nonuniform field each field element is assigned a unique value from a list, taking the following form where the token identifier form of list is recommended:

```
internalField nonuniform <List>;
```

The `boundaryField` is a dictionary containing a set of entries whose names correspond to each of the names of the boundary patches listed in the *boundary* file in the *polyMesh* directory. Each patch entry is itself a dictionary containing a list of keyword entries. The compulsory entry, `type`, describes the patch field condition specified for the field. The remaining entries correspond to the type of patch field condition selected and can typically include field data specifying initial conditions on patch faces (see Section 5.1). An example set of field dictionary entries for velocity `U` are shown below:

```
17  dimensions      [0 1 -1 0 0 0 0];
18
19  internalField   uniform (0 0 0);
20
21  boundaryField
22  {
23      movingWall
24      {
25          type            fixedValue;
26          value           uniform (1 0 0);
27      }
28
29      fixedWalls
30      {
31          type            noSlip;
32      }
33
34      frontAndBack
35      {
36          type            empty;
37      }
38  }
39
40
41  // ************************************************************************* //
```

2.2.9 Directives and macro substitutions

There is additional file syntax that offers great flexibility for the setting up of OpenFOAM case files, namely directives and macro substitutions. Directives are commands that can be contained within case files that begin with the hash (`#`) symbol. Macro substitutions begin with the dollar (`$`) symbol.

At present there are 4 directive commands available in OpenFOAM:

`#include "<fileName>"` (or `#includeIfPresent "<fileName>"` reads the file of name <*fileName*>;

`#inputMode` has the following options:

- `#default` : provide default value if entry is not already defined
- `#overwrite` : silently overwrites existing entries
- `#warn` : warn about duplicate entries
- `#error` : error if any duplicate entries occur
- `#merge` : merge sub-directories when possible (the default mode)

`#remove <keywordEntry>` removes any included keyword entry; can take a word or regular expression;

`#codeStream` followed by verbatim C++ code, compiles, loads and executes the code on-the-fly to generate the entry.

2.2.10 The #include and #inputMode directives

For example, let us say a user wishes to set an initial value of pressure once to be used as the internal field and initial value at a boundary. We could create a file, *e.g.* named *initialConditions*, which contains the following entries:

```
pressure 1e+05;
#inputMode merge
```

In order to use this pressure for both the internal and initial boundary fields, the user would simply include the following macro substitutions in the pressure field file *p*:

```
#include "initialConditions"
internalField uniform $pressure;
boundaryField
{
    patch1
    {
        type fixedValue;
        value $internalField;
    }
}
```

This is a fairly trivial example that simply demonstrates how this functionality works. However, the functionality can be used in many, more powerful ways particularly as a means of generalising case data to suit the user's needs. For example, if a user has a set of cases that require the same RAS turbulence model settings, a single file can be created with those settings which is simply included in the *turbulenceProperties* file of each case. Macro substitutions can extend well beyond a single value so that, for example, sets of boundary conditions can be predefined and called by a single macro. The extent to which such functionality can be used is almost endless.

2.2.11 The #codeStream directive

The `#codeStream` directive takes C++ code which is compiled and executed to deliver the dictionary entry. The code and compilation instructions are specified through the following keywords.

- `code`: specifies the code, called with arguments `OStream& os` and `const dictionary& dict` which the user can use in the code, *e.g.* to lookup keyword entries from within the current case dictionary (file).
- `codeInclude` (optional): specifies additional C++ `#include` statements to include OpenFOAM files.
- `codeOptions` (optional): specifies any extra compilation flags to be added to `EXE_INC` in *Make/options*.
- `codeLibs` (optional): specifies any extra compilation flags to be added to `LIB_LIBS` in *Make/options*.

Code, like any string, can be written across multiple lines by enclosing it within hash-bracket delimiters, *i.e.* `#{...#}`. Anything in between these two delimiters becomes a string with all newlines, quotes, *etc.* preserved.

An example of `#codeStream` is given below. The code in the *controlDict* file looks up dictionary entries and does a simple calculation for the write interval:

```
startTime       0;
endTime         100;
...
writeInterval   #codeStream
{
    code
    #{
        scalar start = readScalar(dict.lookup("startTime"));
        scalar end = readScalar(dict.lookup("endTime"));
        label nDumps = 5;
        os << ((end - start)/nDumps);
    #};
};
```

Chapter 3

Running applications

We should reiterate from the outset that OpenFOAM is a C++ library used primarily to create executables, known as *applications*. OpenFOAM is distributed with a large set of precompiled applications but users also have the freedom to create their own or modify existing ones. Applications are split into two main categories:

solvers that are each designed to solve a specific problem in computational continuum mechanics;

utilities that perform simple pre-and post-processing tasks, mainly involving data manipulation and algebraic calculations.

OpenFOAM is divided into a set of precompiled libraries that are dynamically linked during compilation of the solvers and utilities. Libraries such as those for physical models are supplied as source code so that users may conveniently add their own models to the libraries. This chapter gives an overview of solvers, utilities and libraries, their creation, modification, compilation and execution.

3.1 Running applications

Each application is designed to be executed from a terminal command line, typically reading and writing a set of data files associated with a particular case. The data files for a case are stored in a directory named after the case as described in section 2.1; the directory name with full path is here given the generic name <*caseDir*>.

For any application, the form of the command line entry for any can be found by simply entering the application name at the command line with the `-help` option, *e.g.* typing

```
blockMesh -help
```

returns the usage

```
Usage: blockMesh [OPTIONS]
options:
  -case <dir>       Specify case directory to use (instead of cwd)
  -dict <file>      Alternative blockMeshDict
  -merge-points     Geometric point merging instead of topological merging
                    [default for 1912 and earlier].
  -no-clean         Do not remove polyMesh/ directory or files
```

```
  -region <name>    Specify alternative mesh region
  -sets             Write cellZones as cellSets too (for processing purposes)
  -time <time>      Specify a time to write mesh to (default: constant)
  -verbose          Force verbose output. (Can be used multiple times)
  -write-vtk        Write topology as VTU file and exit
  -doc              Display documentation in browser
  -help             Display short help and exit
  -help-compat      Display compatibility options and exit
  -help-full        Display full help and exit

Block description

  For a given block, the correspondence between the ordering of
  vertex labels and face labels is shown below.
  For vertex numbering in the sequence 0 to 7 (block, centre):
    faces 0 (f0) and 1 are left and right, respectively;
    faces 2 and 3 are bottom and top;
    and faces 4 and 5 are front the back:

           7 ---- 6
      f5   |\     |\   f3
      |    | 4 ---- 5   \
      |    3 |--- 2 |    \
      |     \|     \|    f2
      f4     0 ---- 1

 Z         f0 ----- f1
 |  Y
 | /
 0 --- X
```

The arguments in square brackets, `[ ]`, are optional flags. If the application is executed from within a case directory, it will operate on that case. Alternatively, the `-case` <`caseDir`> option allows the case to be specified directly so that the application can be executed from anywhere in the filing system.

The `-help-full` offers additional application options. For example the simpleFoam solver offers following extra options:

```
  -fileHandler <handler>
                    Override the file handler type
  -hostRoots <((host1 dir1) .. (hostN dirN))>
                    Per-subprocess root directories for distributed running. The
                    host specification can be a regex.
  -info-switch <name=val>
                    Specify the value of a registered info switch. Default is 1
                    if the value is omitted. (Can be used multiple times)
  -lib <name>       Additional library or library list to load (can be used
                    multiple times)
  -listFunctionObjects
```

```
                  List functionObjects
  -listFvOptions  List fvOptions
  -listRegisteredSwitches
                  List switches registered for run-time modification (see
                  -listUnsetSwitches option)
  -listScalarBCs  List scalar field boundary conditions
                  (fvPatchField<scalar>)
  -listSwitches   List switches declared in libraries (see -listUnsetSwitches
                  option)
  -listTurbulenceModels
                  List turbulenceModels
  -listUnsetSwitches
                  Modifies switch listing to display values not set in
                  etc/controlDict
  -listVectorBCs  List vector field boundary conditions
                  (fvPatchField<vector>)
  -noFunctionObjects
                  Do not execute function objects
  -opt-switch <name=val>
                  Specify the value of a registered optimisation switch.
                  Default is 1 if the value is omitted. (Can be used multiple
                  times)
```

3.1.1 Command line options

Starting from OpenFOAM v1706 the set of command line options can be readily obtained by command-line completion. For example, by pressing the <TAB> key after typing *e.g.*blockMesh will prompt the list of options.

```
blockMesh <TAB> <TAB>
```

Returns:

```
blockMesh -
-case           -doc-source    -help-full     -merge-points  -sets          -write-vtk
-debug-switch   -fileHandler   -help-notes    -no-clean      -time
-dict           -help          -info-switch   -opt-switch    -verbose
-doc            -help-compat   -lib           -region        -write-obj
```

This functionality is available for for all solvers and utilities.

3.1.2 Running in the background

Like any UNIX/Linux executable, applications can be run as a background process, *i.e.* one which does not have to be completed before the user can give the shell additional commands. If the user wished to run the blockMesh example as a background process and output the case progress to a *log* file, they could enter:

```
blockMesh > log &
```

3.2 Running applications in parallel

This section describes how to run OpenFOAM in parallel on distributed processors. The method of parallel computing used by OpenFOAM is known as domain decomposition, in which the geometry and associated fields are broken into pieces and allocated to separate processors for solution. The process of parallel computation involves: decomposition of mesh and fields; running the application in parallel; and, post-processing the decomposed case as described in the following sections. The parallel running uses the public domain openMPI implementation of the standard message passing interface (MPI).

3.2.1 Decomposition of mesh and initial field data

The mesh and fields are decomposed using the decomposePar utility. The underlying aim is to break up the domain with minimal effort but in such a way to guarantee a fairly economic solution. The geometry and fields are broken up according to a set of parameters specified in a dictionary named *decomposeParDict* that must be located in the *system* directory of the case of interest. An example *decomposeParDict* dictionary can be copied from the interFoam/damBreak/damBreak tutorial if the user requires one; the dictionary entries within it are reproduced below:

```
17  numberOfSubdomains 4;
18
19  method          simple;
20
21  coeffs
22  {
23      n           (2 2 1);
24  }
25
26
27  // ************************************************************************* //
```

The user has a choice of four methods of decomposition, specified by the `method` keyword as described below.

`simple` Simple geometric decomposition in which the domain is split into pieces by direction, *e.g.* 2 pieces in the x direction, 1 in y *etc.*

`hierarchical` Hierarchical geometric decomposition which is the same as `simple` except the user specifies the order in which the directional split is done, *e.g.* first in the y-direction, then the x-direction *etc.*

`scotch` Scotch decomposition which requires no geometric input from the user and attempts to minimise the number of processor boundaries. The user can specify a weighting for the decomposition between processors, through an optional `processorWeights` keyword which can be useful on machines with differing performance between processors. There is also an optional keyword entry `strategy` that controls the decomposition strategy through a complex string supplied to Scotch. For more information, see the source code file: *$FOAM_SRC/parallel/decompose/scotchDecomp/scotchDecomp.C*

`manual` Manual decomposition, where the user directly specifies the allocation of each cell to a particular processor.

For each `method` there are a set of coefficients specified in a sub-dictionary of *decompositionDict*, named *<method>Coeffs* as shown in the dictionary listing. The full set of keyword entries in the *decomposeParDict* dictionary are explained in Table 3.1. Note a

Compulsory entries		
`numberOfSubdomains`	Total number of subdomains	N
`method`	Method of decomposition	`simple`/ `hierarchical`/ `scotch`/ `metis`/ `manual`/
`simpleCoeffs` entries		
`n`	Number of subdomains in x, y, z	$(n_x\ n_y\ n_z)$
`delta`	Cell skew factor	Typically, 10^{-3}
`hierarchicalCoeffs` entries		
`n`	Number of subdomains in x, y, z	$(n_x\ n_y\ n_z)$
`delta`	Cell skew factor	Typically, 10^{-3}
`order`	Order of decomposition	`xyz`/`xzy`/`yxz`...
`scotchCoeffs` entries		
`processorWeights` (optional)	List of weighting factors for allocation of cells to processors; `<wt1>` is the weighting factor for processor 1, *etc.*; weights are normalised so can take any range of values.	`(<wt1>...<wtN>)`
`strategy`	Decomposition strategy (optional); defaults to `"b"`	
`manualCoeffs` entries		
`dataFile`	Name of file containing data of allocation of cells to processors	`"<fileName>"`
Distributed data entries (optional) — see section 3.2.3		
`distributed`	Is the data distributed across several disks?	`yes/no`
`roots`	Root paths to case directories; `<rt1>` is the root path for node 1, *etc.*	`(<rt1>...<rtN>)`

Table 3.1: Keywords in *decompositionDict* dictionary.

change in the syntax of *coeffs* dictionary. From version 1712 users may now specify a single dictionary with a generic name *coeffs*. The earlier `<method>Coeffs` is still supported for backwards compatibility.

The decomposePar utility is executed in the normal manner by typing

```
decomposePar
```

On completion, a set of subdirectories will have been created, one for each processor, in the case directory. The directories are named *processor*N where $N = 0, 1, \ldots$ represents a processor number and contains a time directory, containing the decomposed field descriptions, and a *constant/polyMesh* directory containing the decomposed mesh description.

3.2.2 Running a decomposed case

A decomposed OpenFOAM case is run in parallel using the openMPI implementation of MPI.

openMPI can be run on a local multiprocessor machine very simply but when running on machines across a network, a file must be created that contains the host names of the machines. The file can be given any name and located at any path. In the following description we shall refer to such a file by the generic name, including full path, <*machines*>.

The <*machines*> file contains the names of the machines listed one machine per line. The names must correspond to a fully resolved hostname in the */etc/hosts* file of the machine on which the openMPI is run. The list must contain the name of the machine running the openMPI. Where a machine node contains more than one processor, the node name may be followed by the entry `cpu=`n where n is the number of processors openMPI should run on that node.

For example, let us imagine a user wishes to run openMPI from machine `aaa` on the following machines: `aaa`; `bbb`, which has 2 processors; and `ccc`. The <*machines*> would contain:

```
aaa
bbb cpu=2
ccc
```

An application is run in parallel using mpirun.

```
mpirun --hostfile <machines> -np <nProcs>
    <foamExec> <otherArgs> -parallel > log &
```

where: <`nProcs`> is the number of processors; <`foamExec`> is the executable, *e.g.*icoFoam; and, the output is redirected to a file named `log`. For example, if icoFoam is run on 4 nodes, specified in a file named *machines*, on the cavity tutorial in the $FOAM_RUN/*tutorials/incompressible/icoFoam* directory, then the following command should be executed:

```
mpirun --hostfile machines -np 4 icoFoam -parallel > log &
```

3.2.3 Distributing data across several disks

Data files may need to be distributed if, for example, if only local disks are used in order to improve performance. In this case, the user may find that the root path to the case directory may differ between machines. The paths must then be specified in the *decomposeParDict* dictionary using `distributed` and `roots` keywords. The `distributed` entry should read

```
distributed  yes;
```

and the `roots` entry is a list of root paths, <`root0`>, <`root1`>, ..., for each node

```
roots
<nRoots>
(
```

```
    "<root0>"
    "<root1>"
    ...
);
```

where <nRoots> is the number of roots.

Each of the *processor*<*N*> directories should be placed in the case directory at each of the root paths specified in the *decomposeParDict* dictionary. The *system* directory and *files* within the *constant* directory must also be present in each case directory. Note: the files in the *constant* directory are needed, but the *polyMesh* directory is not.

3.2.4 Post-processing parallel processed cases

When post-processing cases that have been run in parallel the user can:

- reconstruct the mesh and field data to recreate the complete domain and fields, which can be post-processed as normal;
- post-process each segment of decomposed domain individually; or
- use ParaView via the option `paraFoam -vtk` and select the `decomposedCase` from the GUI whereby the case will be assembled internally

3.2.4.1 Reconstructing mesh and data

After a case has been run in parallel, it can be reconstructed for post-processing. The case is reconstructed by merging the sets of time directories from each *processor*<*N*> directory into a single set of time directories. The reconstructPar utility performs such a reconstruction by executing the command:

```
reconstructPar
```

Executing reconstructPar without any additional options will process all stored time directories. Specifc times can be processed using the following options:

- `-latestTime`: latest time only
- `-time N`: time N
- `-newTimes`: any time directories that have not been processed previously

When the data is distributed across several disks, it must be first copied to the local case directory for reconstruction.

3.2.4.2 Post-processing decomposed cases

The user may post-process decomposed cases using the paraFoam post-processor, described in section 7.1. The whole simulation can be post-processed by reconstructing the case or alternatively it is possible to post-process a segment of the decomposed domain individually by simply treating the individual processor directory as a case in its own right.

Chapter 4

Mesh generation and conversion

This chapter describes all topics relating to the creation of meshes in OpenFOAM: section 4.1 gives an overview of the ways a mesh may be described in OpenFOAM; section 4.3 covers the blockMesh utility for generating simple meshes of blocks of hexahedral cells; section 4.4 covers the snappyHexMesh utility for generating complex meshes of hexahedral and split-hexahedral cells automatically from triangulated surface geometries; section 4.5 describes the options available for conversion of a mesh that has been generated by a third-party product into a format that OpenFOAM can read.

4.1 Mesh description

This section provides a specification of the way the OpenFOAM C++ classes handle a mesh. The mesh is an integral part of the numerical solution and must satisfy certain criteria to ensure a valid, and hence accurate, solution. During any run, OpenFOAM checks that the mesh satisfies a fairly stringent set of validity constraints and will cease running if the constraints are not satisfied. The consequence is that a user may experience some frustration in 'correcting' a large mesh generated by third-party mesh generators before OpenFOAM will run using it. This is unfortunate but we make no apology for OpenFOAM simply adopting good practice to ensure the mesh is valid; otherwise, the solution is flawed before the run has even begun.

By default OpenFOAM defines a mesh of arbitrary polyhedral cells in 3-D, bounded by arbitrary polygonal faces, *i.e.* the cells can have an unlimited number of faces where, for each face, there is no limit on the number of edges nor any restriction on its alignment. A mesh with this general structure is known in OpenFOAM as a polyMesh. This type of mesh offers great freedom in mesh generation and manipulation in particular when the geometry of the domain is complex or changes over time. The price of absolute mesh generality is, however, that it can be difficult to convert meshes generated using conventional tools. The OpenFOAM library therefore provides cellShape tools to manage conventional mesh formats based on sets of pre-defined cell shapes.

4.1.1 Mesh specification and validity constraints

Before describing the OpenFOAM mesh format, polyMesh, and the cellShape tools, we will first set out the validity constraints used in OpenFOAM. The conditions that a mesh must satisfy are:

4.1.1.1 Points

A point is a location in 3-D space, defined by a vector in units of metres (m). The points are compiled into a list and each point is referred to by a label, which represents its position in the list, starting from zero. *The point list cannot contain two different points at an exactly identical position nor any point that is not part at least one face.*

4.1.1.2 Faces

A face is an ordered list of points, where a point is referred to by its label. The ordering of point labels in a face is such that each two neighbouring points are connected by an edge, *i.e.* you follow points as you travel around the circumference of the face. Faces are compiled into a list and each face is referred to by its label, representing its position in the list. The direction of the face normal vector is defined by the right-hand rule, *i.e.* looking towards a face, if the numbering of the points follows an anti-clockwise path, the normal vector points towards you, as shown in Figure 4.1.

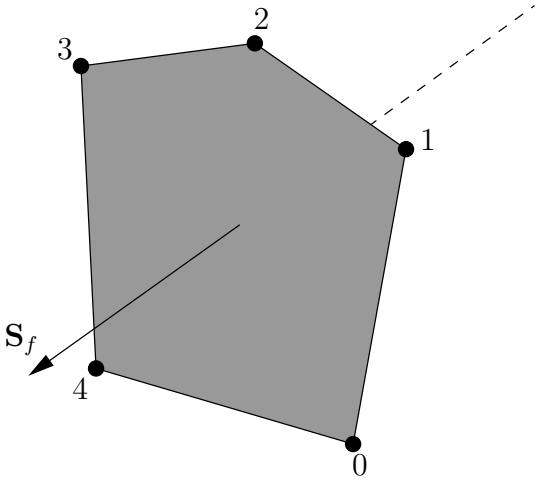


Figure 4.1: Face area vector from point numbering on the face

There are two types of face:

Internal faces Those faces that connect two cells (and it can never be more than two). For each internal face, the ordering of the point labels is such that the face normal points into the cell with the larger label, *i.e.* for cells 2 and 5, the normal points into 5;

Boundary faces Those belonging to one cell since they coincide with the boundary of the domain. A boundary face is therefore addressed by one cell(only) and a boundary patch. The ordering of the point labels is such that the face normal points outside of the computational domain.

Faces are generally expected to be convex; at the very least the face centre needs to be inside the face. Faces are allowed to be warped, i.e. not all points of the face need to be coplanar.

4.1.1.3 Cells

A cell is a list of faces in arbitrary order. Cells must have the properties listed below.

Contiguous The cells must completely cover the computational domain and must not overlap one another.

Convex Every cell must be convex and its cell centre inside the cell.

Closed Every cell must be *closed*, both geometrically and topologically where:

- geometrical closedness requires that when all face area vectors are oriented to point outwards of the cell, their sum should equal the zero vector to machine accuracy;
- topological closedness requires that all the edges in a cell are used by exactly two faces of the cell in question.

Orthogonality For all internal faces of the mesh, we define the centre-to-centre vector as that connecting the centres of the 2 cells that it adjoins oriented from the centre of the cell with smaller label to the centre of the cell with larger label. The orthogonality constraint requires that for each internal face, the angle between the face area vector, oriented as described above, and the centre-to-centre vector must always be less than 90°.

4.1.1.4 Boundary

A boundary is a list of patches, each of which is associated with a boundary condition. A patch is a list of face labels which clearly must contain only boundary faces and no internal faces. The boundary is required to be closed, *i.e.* the sum all boundary face area vectors equates to zero to machine tolerance.

4.1.2 The polyMesh description

The *constant* directory contains a full description of the case polyMesh in a subdirectory *polyMesh*. The polyMesh description is based around faces and, as already discussed, internal cells connect 2 cells and boundary faces address a cell and a boundary patch. Each face is therefore assigned an 'owner' cell and 'neighbour' cell so that the connectivity across a given face can simply be described by the owner and neighbour cell labels. In the case of boundaries, the connected cell is the owner and the neighbour is assigned the label '-1'. With this in mind, the I/O specification consists of the following files:

points a list of vectors describing the cell vertices, where the first vector in the list represents vertex 0, the second vector represents vertex 1, *etc.*;

faces a list of faces, each face being a list of indices to vertices in the points list, where again, the first entry in the list represents face 0, *etc.*;

owner a list of owner cell labels, the index of entry relating directly to the index of the face, so that the first entry in the list is the owner label for face 0, the second entry is the owner label for face 1, etc;

neighbour a list of neighbour cell labels;

boundary a list of patches, containing a dictionary entry for each patch, declared using the patch name, *e.g.*

```
movingWall
{
    type patch;
    nFaces 20;
    startFace 760;
```

}

The `startFace` is the index into the face list of the first face in the patch, and `nFaces` is the number of faces in the patch.

Note that if the user wishes to know how many cells are in their domain, there is a `note` *in the* `FoamFile` *header of the* owner *file that contains an entry for* `nCells`.

4.1.3 The cellShape tools

We shall describe the alternative cellShape tools that may be used particularly when converting some standard (simpler) mesh formats for the use with OpenFOAM library.

The vast majority of mesh generators and post-processing systems support only a fraction of the possible polyhedral cell shapes in existence. They define a mesh in terms of a limited set of 3D cell geometries, referred to as *cell shapes*. The OpenFOAM library contains definitions of these standard shapes, to enable a conversion of such a mesh into the polyMesh format described in the previous section.

The cellShape models supported by OpenFOAM are shown in Table 4.1. The shape is defined by the ordering of point labels in accordance with the numbering scheme contained in the shape model. The ordering schemes for points, faces and edges are shown in Table 4.1. The numbering of the points must not be such that the shape becomes twisted or degenerate into other geometries, *i.e.* the same point label cannot be used more that once is a single shape. Moreover it is unnecessary to use duplicate points in OpenFOAM since the available shapes in OpenFOAM cover the full set of degenerate hexahedra.

The cell description consists of two parts: the name of a cell model and the ordered list of labels. Thus, using the following list of points

```
8
(
    (0 0 0)
    (1 0 0)
    (1 1 0)
    (0 1 0)
    (0 0 0.5)
    (1 0 0.5)
    (1 1 0.5)
    (0 1 0.5)
)
```

A hexahedral cell would be written as:

```
(hex 8(0 1 2 3 4 5 6 7))
```

Here the hexahedral cell shape is declared using the keyword `hex`. Other shapes are described by the keywords listed in Table 4.1.

4.1.4 1- and 2-dimensional and axi-symmetric problems

OpenFOAM is designed as a code for 3-dimensional space and defines all meshes as such. However, 1- and 2- dimensional and axi-symmetric problems can be simulated in OpenFOAM by generating a mesh in 3 dimensions and applying special boundary conditions on any patch in the plane(s) normal to the direction(s) of interest. More specifically, 1- and 2- dimensional problems use the empty patch type and axi-symmetric problems use the wedge type. The use of both are described in section 4.2.2 and the generation of wedge geometries for axi-symmetric problems is discussed in section 4.3.3.

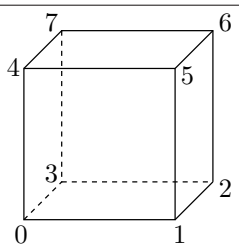
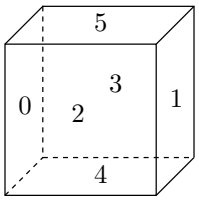
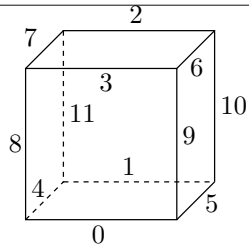
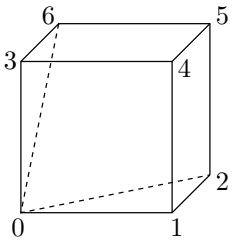
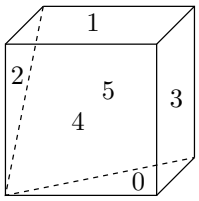
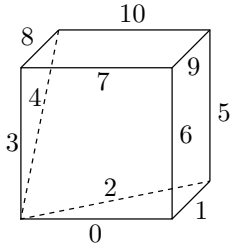
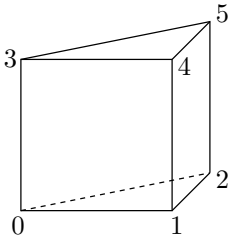
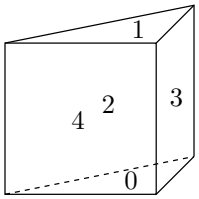
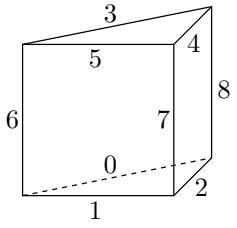
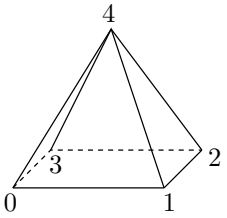
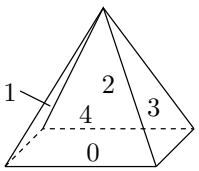
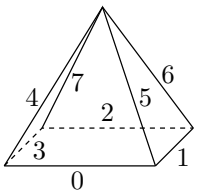
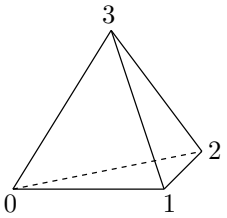
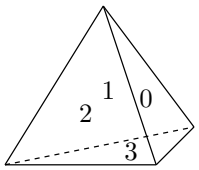
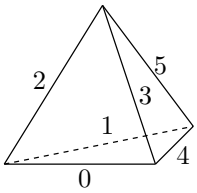
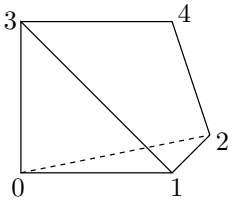
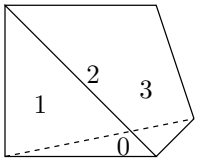
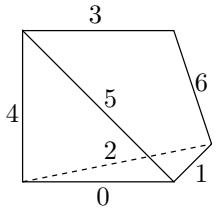
Cell type	Keyword	Vertex numbering	Face numbering	Edge numbering
Hexahedron	`hex`			
Wedge	`wedge`			
Prism	`prism`			
Pyramid	`pyr`			
Tetrahedron	`tet`			
Tet-wedge	`tetWedge`			

Table 4.1: Vertex, face and edge numbering for cellShapes.

4.2 Boundaries

In this section we discuss the way in which mesh boundaries are treated in OpenFOAM. We first need to consider that, for the purpose of applying boundary conditions, a boundary is generally broken up into a set of *patches*. One patch may include one or more enclosed areas of the boundary surface which do not necessarily need to be physically connected.

4.2.1 Specification of patch types in OpenFOAM

The patch types are specified in the mesh and field files of a OpenFOAM case. More precisely:

- the base type is specified under the `type` keyword for each patch in the *boundary* file, located in the *constant/polyMesh* directory;

An example *boundary* file is shown below for a sonicFoam case

```
17  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
18
19  6
20  (
21      inlet
22      {
23          type            patch;
24          nFaces          50;
25          startFace       10325;
26      }
27      outlet
28      {
29          type            patch;
30          nFaces          40;
31          startFace       10375;
32      }
33      bottom
34      {
35          type            symmetryPlane;
36          inGroups        1(symmetryPlane);
37          nFaces          25;
38          startFace       10415;
39      }
40      top
41      {
42          type            symmetryPlane;
43          inGroups        1(symmetryPlane);
44          nFaces          125;
45          startFace       10440;
46      }
47      obstacle
48      {
49          type            patch;
50          nFaces          110;
51          startFace       10565;
52      }
53      defaultFaces
54      {
55          type            empty;
56          inGroups        1(empty);
57          nFaces          10500;
58          startFace       10675;
59      }
60  )
61
62  // ************************************************************************* //
```

The `type` in the boundary file is `patch` for all patches except those patches that have some geometrical constraint applied to them, *i.e.* the symmetryPlane and empty patches.

4.2.2 Base types

The base and geometric types are described below; the keywords used for specifying these types in OpenFOAM are summarised in Table 4.2.

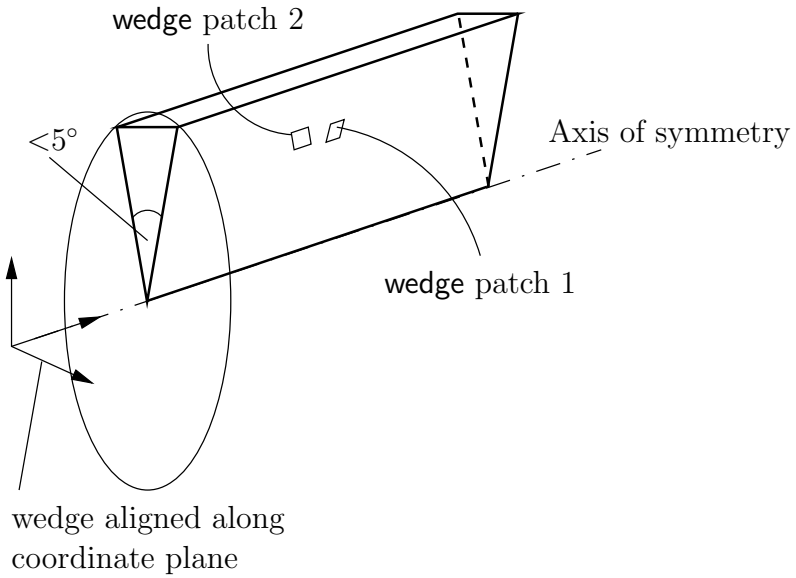


Figure 4.2: Axi-symmetric geometry using the wedge patch type.

Selection Key	Description
`patch`	generic patch
`symmetryPlane`	plane of symmetry
`empty`	front and back planes of a 2D geometry
`wedge`	wedge front and back for an axi-symmetric geometry
`cyclic`	cyclic plane
`wall`	wall — used for wall functions in turbulent flows
`processor`	inter-processor boundary

Table 4.2: Basic patch types.

patch The basic patch type for a patch condition that contains no geometric or topological information about the mesh (with the exception of wall), *e.g.* an inlet or an outlet.

wall There are instances where a patch that coincides with a wall needs to be identifiable as such, particularly where specialist modelling is applied at wall boundaries. A good example is wall turbulence modelling where a wall must be specified with a wall patch type, so that the distance from the wall of the cell centres next to the wall are stored as part of the patch.

symmetryPlane For a symmetry plane.

empty While OpenFOAM always generates geometries in 3 dimensions, it can be instructed to solve in 2 (or 1) dimensions by specifying a special empty condition on each patch whose plane is normal to the 3rd (and 2nd) dimension for which no solution is required.

wedge For 2 dimensional axi-symmetric cases, *e.g.* a cylinder, the geometry is specified as a wedge of small angle (*e.g.* $< 5°$) and 1 cell thick running along the plane of symmetry, straddling one of the coordinate planes, as shown in Figure 4.2. The axi-symmetric wedge planes must be specified as separate patches of wedge type. The details of generating wedge-shaped geometries using blockMesh are described in section 4.3.3.

cyclic Enables two patches to be treated as if they are physically connected; used for repeated geometries, *e.g.* heat exchanger tube bundles. One cyclic patch is linked to another through a `neighbourPatch` keyword in the *boundary* file. Each pair of connecting faces must have similar area to within a tolerance given by the `matchTolerance` keyword in the *boundary* file. Faces do not need to be of the same orientation.

processor If a code is being run in parallel, on a number of processors, then the mesh must be divided up so that each processor computes on roughly the same number of cells. The boundaries between the different parts of the mesh are called processor boundaries.

4.3 Mesh generation with the blockMesh utility

This section describes the mesh generation utility, blockMesh, supplied with OpenFOAM. The blockMesh utility creates parametric meshes with grading and curved edges.

The mesh is generated from a dictionary file named *blockMeshDict* located in the *constant/polyMesh* directory of a case. blockMesh reads this dictionary, generates the mesh and writes out the mesh data to *points* and *faces*, *cells* and *boundary* files in the same directory.

The principle behind blockMesh is to decompose the domain geometry into a set of 1 or more three dimensional, hexahedral blocks. Edges of the blocks can be straight lines, arcs or splines. The mesh is ostensibly specified as a number of cells in each direction of the block, sufficient information for blockMesh to generate the mesh data.

Each block of the geometry is defined by 8 vertices, one at each corner of a hexahedron. The vertices are written in a list so that each vertex can be accessed using its label, remembering that OpenFOAM always uses the C++ convention that the first element of the list has label '0'. An example block is shown in Figure 4.3 with each vertex numbered according to the list. The edge connecting vertices 1 and 5 is curved to remind the reader that curved edges can be specified in blockMesh.

It is possible to generate blocks with less than 8 vertices by collapsing one or more pairs of vertices on top of each other, as described in section 4.3.3.

Each block has a local coordinate system (x_1, x_2, x_3) that must be right-handed. A right-handed set of axes is defined such that to an observer looking down the Oz axis, with O nearest them, the arc from a point on the Ox axis to a point on the Oy axis is in a clockwise sense.

The local coordinate system is defined by the order in which the vertices are presented in the block definition according to:

- the axis origin is the first entry in the block definition, vertex 0 in our example;
- the x_1 direction is described by moving from vertex 0 to vertex 1;
- the x_2 direction is described by moving from vertex 1 to vertex 2;
- vertices 0, 1, 2, 3 define the plane $x_3 = 0$;
- vertex 4 is found by moving from vertex 0 in the x_3 direction;
- vertices 5,6 and 7 are similarly found by moving in the x_3 direction from vertices 1,2 and 3 respectively.

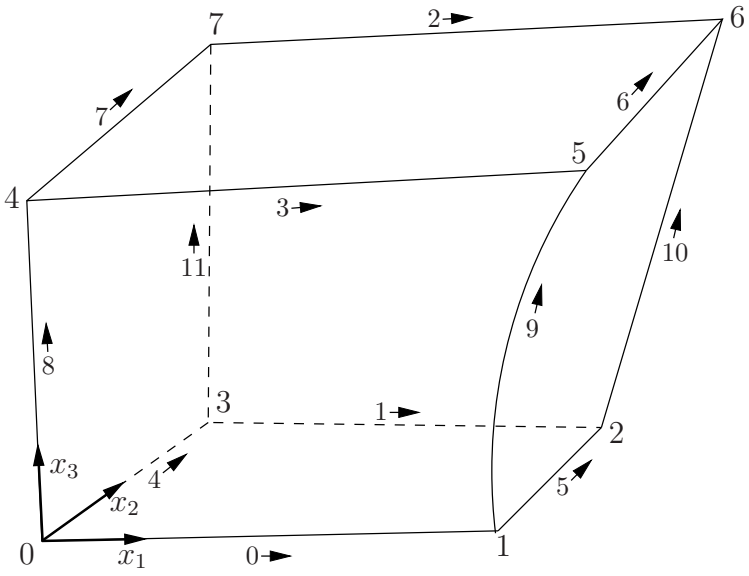


Figure 4.3: A single block

Keyword	Description	Example/selection
`scale`	Scaling factor for the vertex coordinates	`0.001` scales to mm
`vertices`	List of vertex coordinates	`(0 0 0)`
`edges`	Used to describe `arc` or `spline` edges	`arc 1 4 (0.939 0.342 -0.5)`

`block`	Ordered list of vertex labels and mesh size	`hex (0 1 2 3 4 5 6 7)`
		`(10 10 1)`
		`simpleGrading (1.0 1.0 1.0)`

`patches`	List of patches	`symmetryPlane base` `( (0 1 2 3) )`
`mergePatchPairs`	List of patches to be merged	see section 4.3.2

Table 4.3: Keywords used in *blockMeshDict*.

4.3.1 Writing a *blockMeshDict* file

The *blockMeshDict* file is a dictionary using keywords described in Table 4.3. The `scale` keyword specifies a scaling factor by which all vertex coordinates in the mesh description are multiplied. For example,

```
scale           0.001;
```

means that all coordinates are multiplied by 0.001, *i.e.* the values quoted in the *blockMeshDict* file are in mm.

4.3.1.1 The `vertices`

The vertices of the blocks of the mesh are given next as a standard list named `vertices`, *e.g.* for our example block in Figure 4.3, the vertices are:

```
vertices
(
    ( 0    0    0  )    // vertex number 0
```

```
    ( 1    0    0.1)    // vertex number 1
    ( 1.1  1    0.1)    // vertex number 2
    ( 0    1    0.1)    // vertex number 3
    (-0.1 -0.1  1  )    // vertex number 4
    ( 1.3  0    1.2)    // vertex number 5
    ( 1.4  1.1  1.3)    // vertex number 6
    ( 0    1    1.1)    // vertex number 7
);
```

4.3.1.2 The `edges`

Each edge joining 2 vertex points is assumed to be straight by default. However any edge may be specified to be curved by entries in a list named `edges`. The list is optional; if the geometry contains no curved edges, it may be omitted.

Each entry for a curved edge begins with a keyword specifying the type of curve from those listed in Table 4.4.

Keyword selection	Description	Additional entries
`arc`	Circular arc	Single interpolation point
`simpleSpline`	Spline curve	List of interpolation points
`polyLine`	Set of lines	List of interpolation points
`polySpline`	Set of splines	List of interpolation points
`line`	Straight line	—

Table 4.4: Edge types available in the *blockMeshDict* dictionary.

The keyword is then followed by the labels of the 2 vertices that the edge connects. Following that, interpolation points must be specified through which the edge passes. For a `arc`, a single interpolation point is required, which the circular arc will intersect. For `simpleSpline`, `polyLine` and `polySpline`, a list of interpolation points is required. The `line` edge is directly equivalent to the option executed by default, and requires no interpolation points. Note that there is no need to use the `line` edge but it is included for completeness. For our example block in Figure 4.3 we specify an `arc` edge connecting vertices 1 and 5 as follows through the interpolation point $(1.1, 0.0, 0.5)$:

```
edges
(
    arc 1 5 (1.1 0.0 0.5)
);
```

4.3.1.3 The `blocks`

The block definitions are contained in a list named `blocks`. Each block definition is a compound entry consisting of a list of vertex labels whose order is described in section 4.3, a vector giving the number of cells required in each direction, the type and list of cell expansion ratio in each direction.

Then the blocks are defined as follows:

```
blocks
(
```

```
    hex (0 1 2 3 4 5 6 7)    // vertex numbers
    (10 10 10)               // numbers of cells in each direction
    simpleGrading (1 2 3)    // cell expansion ratios
);
```

The definition of each block is as follows:

Vertex numbering The first entry is the shape identifier of the block, as defined in the *.OpenFOAM-v2112/cellModels* file. The shape is always `hex` since the blocks are always hexahedra. There follows a list of vertex numbers, ordered in the manner described on page U-40.

Number of cells The second entry gives the number of cells in each of the x_1 x_2 and x_3 directions for that block.

Cell expansion ratios The third entry gives the cell expansion ratios for each direction in the block. The expansion ratio enables the mesh to be graded, or refined, in specified directions. The ratio is that of the width of the end cell δ_e along one edge of a block to the width of the start cell δ_s along that edge, as shown in Figure 4.4. Each of the following keywords specify one of two types of grading specification available in blockMesh.

`simpleGrading` The simple description specifies uniform expansions in the local x_1, x_2 and x_3 directions respectively with only 3 expansion ratios, *e.g.*

```
simpleGrading (1 2 3)
```

`edgeGrading` The full cell expansion description gives a ratio for each edge of the block, numbered according to the scheme shown in Figure 4.3 with the arrows representing the direction 'from first cell...to last cell' *e.g.* something like

```
edgeGrading (1 1 1 1 2 2 2 2 3 3 3 3)
```

This means the ratio of cell widths along edges 0-3 is 1, along edges 4-7 is 2 and along 8-11 is 3 and is directly equivalent to the `simpleGrading` example given above.

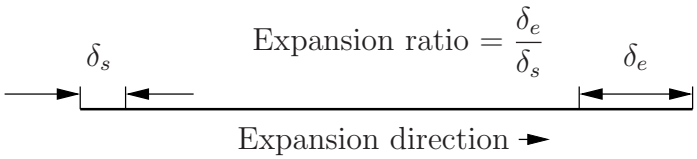


Figure 4.4: Mesh grading along a block edge

4.3.1.4 The `boundary`

The boundary of the mesh is given in a list named `boundary`. The boundary is broken into patches (regions), where each patch in the list has its name as the keyword, which is the choice of the user, although we recommend something that conveniently identifies the patch, *e.g.*`inlet`; the name is used as an identifier for setting boundary conditions in the field data files. The patch information is then contained in sub-dictionary with:

- `type`: the patch type, either a generic `patch` on which some boundary conditions are applied or a particular geometric condition, as listed in Table 4.2 and described in section 4.2.2;

- `faces`: a list of block faces that make up the patch and whose name is the choice of the user, although we recommend something that conveniently identifies the patch, *e.g.*`inlet`; the name is used as an identifier for setting boundary conditions in the field data files.

blockMesh collects faces from any boundary patch that is omitted from the `boundary` list and assigns them to a default patch named `defaultFaces` of type empty. This means that for a 2 dimensional geometry, the user has the option to omit block faces lying in the 2D plane, knowing that they will be collected into an empty patch as required.

Returning to the example block in Figure 4.3, if it has an inlet on the left face, an output on the right face and the four other faces are walls then the patches could be defined as follows:

```
boundary                // keyword
(
    inlet               // patch name
    {
        type patch;     // patch type for patch 0
        faces
        (
            (0 4 7 3); // block face in this patch
        );
    }                   // end of 0th patch definition

    outlet              // patch name
    {
        type patch;     // patch type for patch 1
        faces
        (
            (1 2 6 5)
        );
    }

    walls
    {
        type wall;
        faces
        (
            (0 1 5 4)
            (0 3 2 1)
            (3 7 6 2)
            (4 5 6 7)
        );
    }
);
```

Each block face is defined by a list of 4 vertex numbers. The order in which the vertices are given **must** be such that, looking from inside the block and starting with any vertex, the face must be traversed in a clockwise direction to define the other vertices.

When specifying a cyclic patch in blockMesh, the user must specify the name of the related cyclic patch through the `neighbourPatch` keyword. For example, a pair of cyclic patches might be specified as follows:

```
left
{
    type            cyclic;
    neighbourPatch  right;
    faces           ((0 4 7 3));
}
right
{
    type            cyclic;
    neighbourPatch  left;
    faces           ((1 5 6 2));
}
```

4.3.2 Multiple blocks

A mesh can be created using more than 1 block. In such circumstances, the mesh is created as has been described in the preceeding text; the only additional issue is the connection between blocks, in which there are two distinct possibilities:

face matching the set of faces that comprise a patch from one block are formed from *the same set of vertices* as a set of faces patch that comprise a patch from another block;

face merging a group of faces from a patch from one block are connected to another group of faces from a patch from another block, to create a new set of internal faces connecting the two blocks.

To connect two blocks with **face matching**, the two patches that form the connection should simply be ignored from the `patches` list. blockMesh then identifies that the faces do not form an external boundary and combines each collocated pair into a single internal faces that connects cells from the two blocks.

The alternative, **face merging**, requires that the block patches to be merged are first defined in the `patches` list. Each pair of patches whose faces are to be merged must then be included in an optional list named `mergePatchPairs`. The format of `mergePatchPairs` is:

```
mergePatchPairs
(
    ( <masterPatch> <slavePatch> ) // merge patch pair 0
    ( <masterPatch> <slavePatch> ) // merge patch pair 1
    ...
)
```

The pairs of patches are interpreted such that the first patch becomes the *master* and the second becomes the *slave*. The rules for merging are as follows:

- the faces of the master patch remain as originally defined, with all vertices in their original location;
- the faces of the slave patch are projected onto the master patch where there is some separation between slave and master patch;

- the location of any vertex of a slave face might be adjusted by blockMesh to eliminate any face edge that is shorter than a minimum tolerance;
- if patches overlap as shown in Figure 4.5, each face that does not merge remains as an external face of the original patch, on which boundary conditions must then be applied;
- if all the faces of a patch are merged, then the patch itself will contain no faces and is removed.

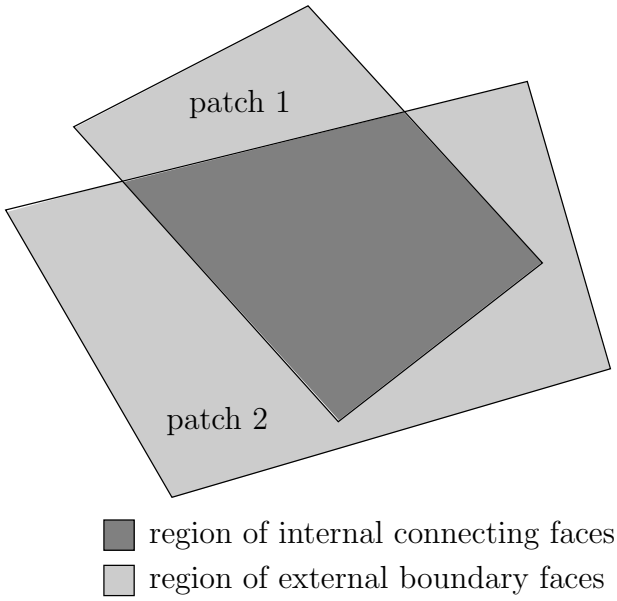


Figure 4.5: Merging overlapping patches

The consequence is that the original geometry of the slave patch will not necessarily be completely preserved during merging. Therefore in a case, say, where a cylindrical block is being connected to a larger block, it would be wise to the assign the master patch to the cylinder, so that its cylindrical shape is correctly preserved. There are some additional recommendations to ensure successful merge procedures:

- in 2 dimensional geometries, the size of the cells in the third dimension, *i.e.* out of the 2D plane, should be similar to the width/height of cells in the 2D plane;
- it is inadvisable to merge a patch twice, *i.e.* include it twice in `mergePatchPairs`;
- where a patch to be merged shares a common edge with another patch to be merged, both should be declared as a master patch.

4.3.3 Creating blocks with fewer than 8 vertices

It is possible to collapse one or more pair(s) of vertices onto each other in order to create a block with fewer than 8 vertices. The most common example of collapsing vertices is when creating a 6-sided wedge shaped block for 2-dimensional axi-symmetric cases that use the wedge patch type described in section 4.2.2. The process is best illustrated by using a simplified version of our example block shown in Figure 4.6. Let us say we wished to create a wedge shaped block by collapsing vertex 7 onto 4 and 6 onto 5. This is simply done by exchanging the vertex number 7 by 4 and 6 by 5 respectively so that the block numbering would become:

```
hex (0 1 2 3 4 5 5 4)
```

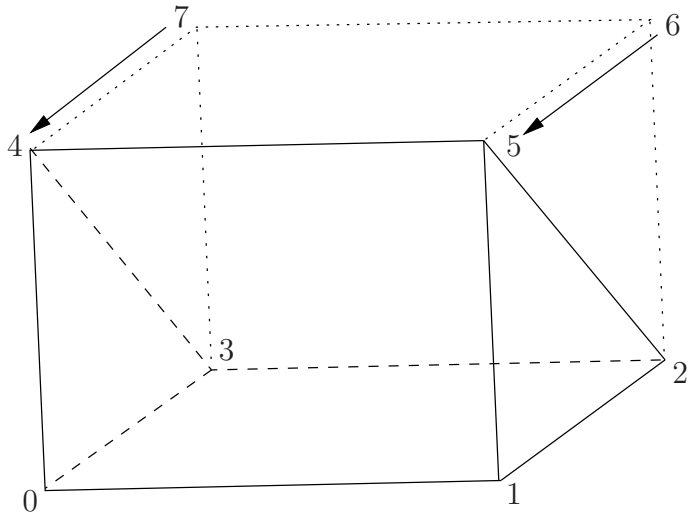


Figure 4.6: Creating a wedge shaped block with 6 vertices

The same applies to the patches with the main consideration that the block face containing the collapsed vertices, previously `(4 5 6 7)` now becomes `(4 5 5 4)`. This is a block face of zero area which creates a patch with no faces in the polyMesh, as the user can see in a *boundary* file for such a case. The patch should be specified as empty in the *blockMeshDict* and the boundary condition for any fields should consequently be empty also.

4.3.4 Running blockMesh

As described in section 3.1, the following can be executed at the command line to run blockMesh for a case in the <*case*> directory:

```
blockMesh -case <case>
```

The *blockMeshDict* file must exist in subdirectory *constant/polyMesh*.

4.4 Mesh generation with the snappyHexMesh utility

This section describes the mesh generation utility, snappyHexMesh, supplied with OpenFOAM. The snappyHexMesh utility generates 3-dimensional meshes containing hexahedra (hex) and split-hexahedra (split-hex) automatically from triangulated surface geometries in Stereolithography (STL) format. The mesh approximately conforms to the surface by iteratively refining a starting mesh and morphing the resulting split-hex mesh to the surface. An optional phase will shrink back the resulting mesh and insert cell layers. The specification of mesh refinement level is very flexible and the surface handling is robust with a pre-specified final mesh quality. It runs in parallel with a load balancing step every iteration.

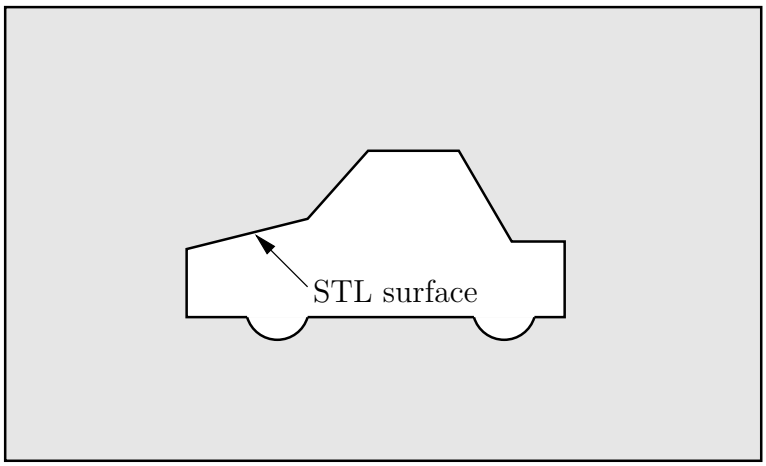


Figure 4.7: Schematic 2D meshing problem for snappyHexMesh

4.4.1 The mesh generation process of snappyHexMesh

The process of generating a mesh using snappyHexMesh will be described using the schematic in Figure 4.7. The objective is to mesh a rectangular shaped region (shaded grey in the figure) surrounding an object described by and STL surface, *e.g.* typical for an external aerodynamics simulation. Note that the schematic is 2-dimensional to make it easier to understand, even though the snappyHexMesh is a 3D meshing tool.

In order to run snappyHexMesh, the user requires the following:

- surface data files in STL format, either binary or ASCII, located in a *constant/triSurface* sub-directory of the case directory;
- a background hex mesh which defines the extent of the computational domain and a base level mesh density; typically generated using blockMesh, discussed in section 4.4.2.
- a *snappyHexMeshDict* dictionary, with appropriate entries, located in the *system* sub-directory of the case.

The *snappyHexMeshDict* dictionary includes: switches at the top level that control the various stages of the meshing process; and, individual sub-directories for each process. The entries are listed in Table 4.5. A full list of analytical shapes can be found online at https://www.openfoam.com/documentation/cpp-guide/html/guide-meshing-snappyhexmesh-geo

All the geometry used by snappyHexMesh is specified in a *geometry* sub-dictionary in the *snappyHexMeshDict* dictionary. The geometry can be specified through an STL surface or bounding geometry entities in OpenFOAM. An example is given below:

```
geometry
{
    sphere.stl // STL filename
    {
        type triSurfaceMesh;
        regions
        {
            secondSolid             // Named region in the STL file
            {
                name mySecondPatch; // User-defined patch name
            }                       // otherwise given sphere.stl_secondSolid
        }
    }

    box1x1x1  // User defined region name
    {
        type   searchableBox;       // region defined by bounding box
```

Keyword	Description	Example
castellatedMesh	Create the castellated mesh?	true
snap	Do the surface snapping stage?	true
addLayers	Add surface layers?	true
mergeTolerance	Merge tolerance as fraction of bounding box of initial mesh	1e-06
debug	Controls writing of intermediate meshes and screen printing	
	— Write final mesh only	0
	— Write intermediate meshes	1
	— Write volScalarField with cellLevel for post-processing	2
	— Write current intersections as *.obj* files	4
geometry	Sub-dictionary of all surface geometry used	
castellatedMeshControls	Sub-dictionary of controls for castellated mesh	
snapControls	Sub-dictionary of controls for surface snapping	
addLayersControls	Sub-dictionary of controls for layer addition	
meshQualityControls	Sub-dictionary of controls for mesh quality	

Table 4.5: Keywords at the top level of *snappyHexMeshDict*.

```
        min    (1.5 1 -0.5);
        max    (3.5 2 0.5);
    }

    sphere2  // User defined region name
    {
        type   searchableSphere;    // region defined by bounding sphere
        centre (1.5 1.5 1.5);
        radius 1.03;
    }
};
```

4.4.2 Creating the background hex mesh

Before snappyHexMesh is executed the user must create a background mesh of hexahedral cells that fills the entire region within by the external boundary as shown in Figure 4.8. This can be done simply using blockMesh. The following criteria must be observed when

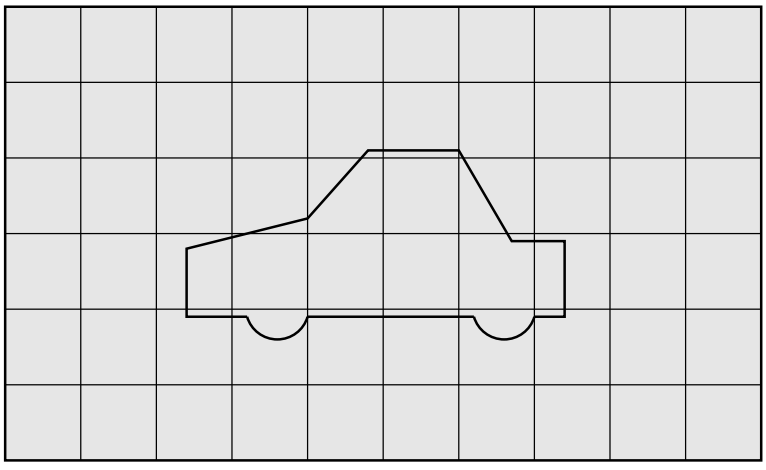


Figure 4.8: Initial mesh generation in snappyHexMesh meshing process

creating the background mesh:

- the mesh must consist purely of hexes;
- the cell aspect ratio should be approximately 1, at least near surfaces at which the subsequent snapping procedure is applied, otherwise the convergence of the snapping procedure is slow, possibly to the point of failure;
- there must be at least one intersection of a cell edge with the STL surface, *i.e.* a mesh of one cell will not work.

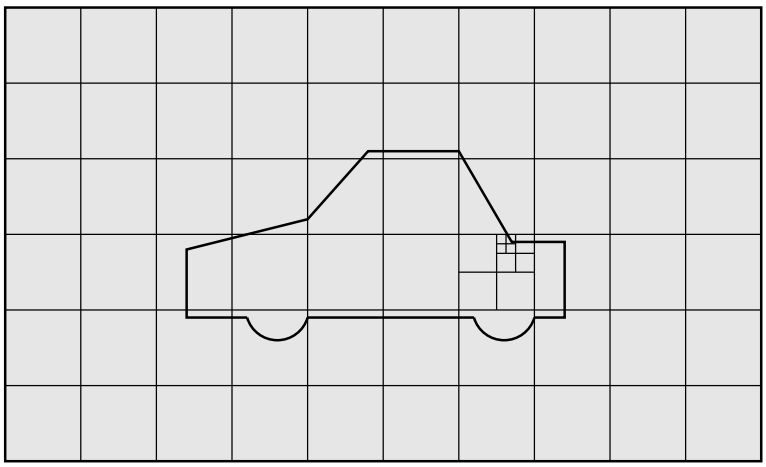


Figure 4.9: Cell splitting by feature edge in snappyHexMesh meshing process

4.4.3 Cell splitting at feature edges and surfaces

Cell splitting is performed according to the specification supplied by the user in the *castellatedMeshControls* sub-dictionary in the *snappyHexMeshDict*. The entries for *castellatedMeshControls* are presented in Table 4.6.

The splitting process begins with cells being selected according to specified edge features first within the domain as illustrated in Figure 4.9. The `features` list in the *castellatedMeshControls* sub-dictionary permits dictionary entries containing a name of an edgeMesh file and the `level` of refinement, *e.g.*:

```
features
(
    {
        file "features.eMesh"; // file containing edge mesh
        level 2;               // level of refinement
    }
);
```

The edgeMesh containing the features can be extracted from the STL geometry file using surfaceFeatureExtract, *e.g.*

```
surfaceFeatureExtract -includedAngle 150 surface.stl features
```

Following feature refinement, cells are selected for splitting in the locality of specified surfaces as illustrated in Figure 4.10. The `refinementSurfaces` dictionary in *castellatedMeshControls* requires dictionary entries for each STL surface and a default `level` specification of the minimum and maximum refinement in the form `(<min> <max>)`.

Keyword	Description	Example
locationInMesh	Location vector inside the region to be meshed *N.B.* vector must not coincide with a cell face either before or during refinement	(5 0 0)
maxLocalCells	Maximum number of cells per processor during refinement	1e+06
maxGlobalCells	Overall cell limit during refinement (*i.e.* before removal)	2e+06
minRefinementCells	If $\geq$ number of cells to be refined, surface refinement stops	0
maxLoadUnbalance	Maximum processor imbalance during refinement where a value of 0 represents a perfect balance	0.1
nCellsBetweenLevels	Number of buffer layers of cells between different levels of refinement	1
resolveFeatureAngle	Applies maximum level of refinement to cells that can see intersections whose angle exceeds this	30
allowFreeStandingZoneFaces	Allow the generation of free-standing zone faces	flase
features	List of features for refinement	
refinementSurfaces	Dictionary of surfaces for refinement	
refinementRegions	Dictionary of regions for refinement	

Table 4.6: Keywords in the *castellatedMeshControls* sub-dictionary of *snappyHexMeshDict*.

The minimum level is applied generally across the surface; the maximum level is applied to cells that can see intersections that form an angle in excess of that specified by resolveFeatureAngle.

The refinement can optionally be overridden on one or more specific region of an STL surface. The region entries are collected in a regions sub-dictionary. The keyword for each region entry is the name of the region itself and the refinement level is contained within a further sub-dictionary. An example is given below:

```
refinementSurfaces
{
    sphere.stl
    {
        level (2 2); // default (min max) refinement for whole surface
        regions
        {
            secondSolid
            {
                level (3 3); // optional refinement for secondSolid region
            }
        }
    }
}
```

4.4.4 Cell removal

Once the feature and surface splitting is complete a process of cell removal begins. Cell removal requires one or more regions enclosed entirely by a bounding surface within the domain. The region in which cells are retained are simply identified by a location vector within that region, specified by the locationInMesh keyword in *castellatedMeshControls*.

Cells are retained if, approximately speaking, 50% or more of their volume lies within the region. The remaining cells are removed accordingly as illustrated in Figure 4.11.

4.4.5 Cell splitting in specified regions

Those cells that lie within one or more specified volume regions can be further split as illustrated in Figure 4.12 by a rectangular region shown by dark shading. The `refinementRegions` sub-dictionary in *castellatedMeshControls* contains entries for refinement of the volume regions specified in the *geometry* sub-dictionary. A refinement `mode` is applied to each region which can be:

- `inside` refines inside the volume region;
- `outside` refines outside the volume region
- `distance` refines according to distance to the surface; and can accommodate different levels at multiple distances with the `levels` keyword.

For the `refinementRegions`, the refinement level is specified by the `levels` list of entries with the format `(<distance> <level>)`. In the case of `inside` and `outside` refinement, the `<distance>` is not required so is ignored (but it must be specified). Examples are shown below:

```
refinementRegions
{
    box1x1x1
    {
        mode inside;
        levels ((1.0 4));         // refinement level 4 (1.0 entry ignored)
    }

    sphere.stl
    {                             // refinement level 5 within 1.0 m
        mode distance;            // refinement level 3 within 2.0 m
        levels ((1.0 5) (2.0 3)); // levels must be ordered nearest first
    }
}
```

4.4.6 Snapping to surfaces

The next stage of the meshing process involves moving cell vertex points onto surface geometry to remove the jagged castellated surface from the mesh. The process is:

Figure 4.10: Cell splitting by surface in snappyHexMesh meshing process

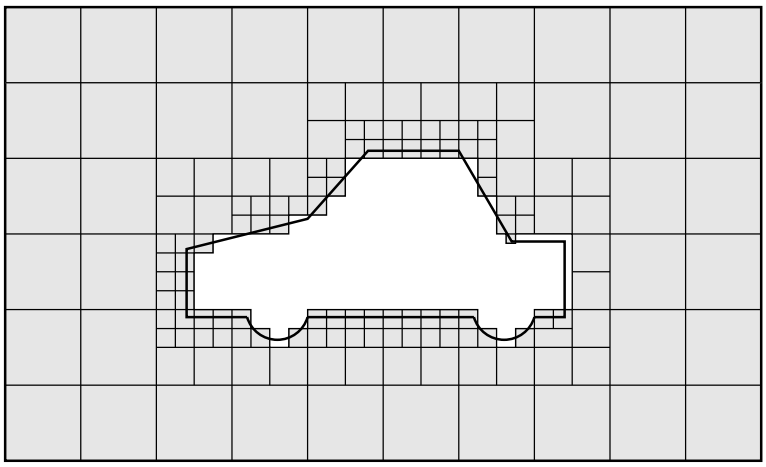


Figure 4.11: Cell removal in snappyHexMesh meshing process

1. displace the vertices in the castellated boundary onto the STL surface;
2. solve for relaxation of the internal mesh with the latest displaced boundary vertices;
3. find the vertices that cause mesh quality parameters to be violated;
4. reduce the displacement of those vertices from their initial value (at 1) and repeat from 2 until mesh quality is satisfied.

The method uses the settings in the *snapControls* sub-dictionary in *snappyHexMeshDict*, listed in Table 4.7. An example is illustrated in the schematic in Figure 4.13 (albeit with

Keyword	Description	Example
`nSmoothPatch`	Number of patch smoothing iterations before finding correspondence to surface	`3`
`tolerance`	Ratio of distance for points to be attracted by surface feature point or edge, to local maximum edge length	`4.0`
`nSolveIter`	Number of mesh displacement relaxation iterations	`30`
`nRelaxIter`	Maximum number of snapping relaxation iterations	`5`
`nFeatureSnapIter`	Number of feature edge snapping iterations	`10`
`implicitFeatureSnap`	Detect (geometric only) features by sampling the surface	false
`explicitFeatureSnap`	Use *castellatedMeshControls* features	true
`multiRegionFeatureSnap`	Detect features between multiple surfaces when using the `explicitFeatureSnap`	false

Table 4.7: Keywords in the *snapControls* dictionary of *snappyHexMeshDict*.

mesh motion that looks slightly unrealistic).

4.4.7 Mesh layers

The mesh output from the snapping stage may be suitable for the purpose, although it can produce some irregular cells along boundary surfaces. There is an optional stage of

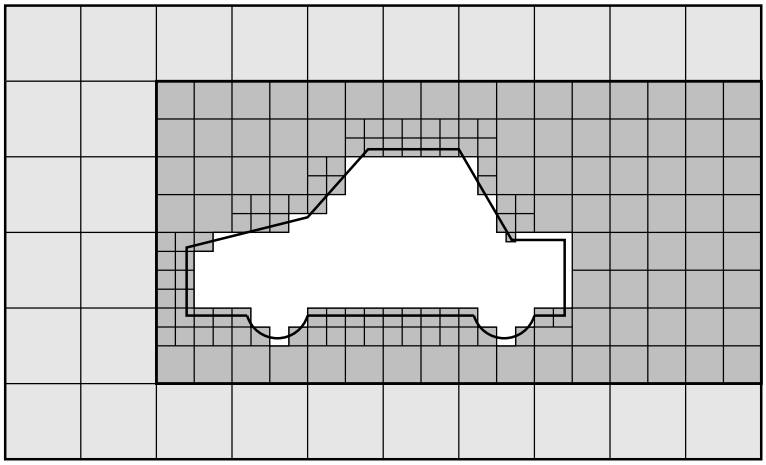


Figure 4.12: Cell splitting by region in snappyHexMesh meshing process

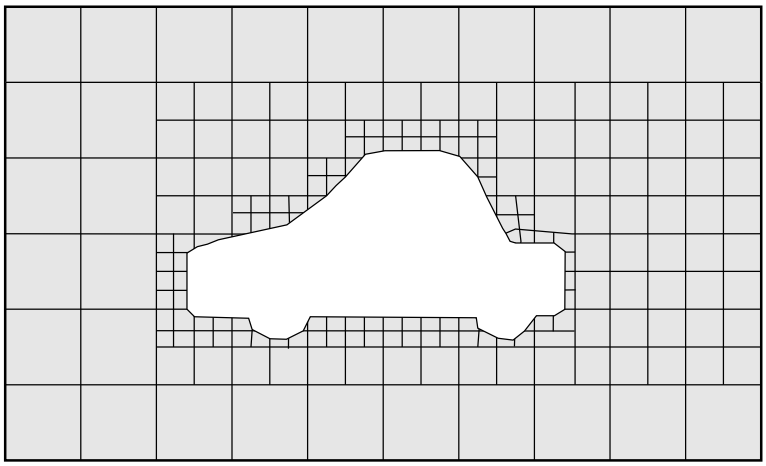


Figure 4.13: Surface snapping in snappyHexMesh meshing process

the meshing process which introduces additional layers of hexahedral cells aligned to the boundary surface as illustrated by the dark shaded cells in Figure 4.14.

The process of mesh layer addition involves shrinking the existing mesh from the boundary and inserting layers of cells, broadly as follows:

1. the mesh is projected back from the surface by a specified thickness in the direction normal to the surface;
2. solve for relaxation of the internal mesh with the latest projected boundary vertices;
3. check if validation criteria are satisfied otherwise reduce the projected thickness and return to 2; if validation cannot be satisfied for any thickness, do not insert layers;
4. if the validation criteria can be satisfied, insert mesh layers;
5. the mesh is checked again; if the checks fail, layers are removed and we return to 2.

The layer addition procedure uses the settings in the *addLayersControls* sub-dictionary in *snappyHexMeshDict*; entries are listed in Table 4.8. The `layers` sub-dictionary contains entries for each *patch* on which the layers are to be applied and the number of surface layers required. The patch name is used because the layers addition relates to the existing mesh, not the surface geometry; hence applied to a patch, not a surface region. An example `layers` entry is as follows:

```
layers
{
    sphere.stl_firstSolid
    {
        nSurfaceLayers 1;
    }
    maxY
    {
        nSurfaceLayers 1;
    }
}
```

4.4.8 Mesh quality controls

The mesh quality is controlled by the entries in the *meshQualityControls* sub-dictionary in *snappyHexMeshDict*; entries are listed in Table 4.9.

4.5 Mesh conversion

The user can generate meshes using other packages and convert them into the format that OpenFOAM uses. There are numerous mesh conversion utilities listed in Table A.2. Some of the more popular mesh converters are listed below and their use is presented in this section.

fluentMeshToFoam reads a Fluent `.msh` mesh file, working for both 2-D and 3-D cases;

star4ToFoam reads STAR-CD/PROSTAR mesh files.

gambitToFoam reads a GAMBIT `.neu` neutral file;

ansysToFoam reads an I-DEAS mesh written in ANSYS `.ans` format;

cfx4ToFoam reads a CFX mesh written in `.geo` format;

4.5.1 fluentMeshToFoam

Fluent writes mesh data to a single file with a *.msh* extension. The file must be written in ASCII format, which is not the default option in Fluent. It is possible to convert single-stream Fluent meshes, including the 2 dimensional geometries. In OpenFOAM, 2 dimensional geometries are currently treated by defining a mesh in 3 dimensions, where the front and back plane are defined as the empty boundary patch type. When reading a 2 dimensional Fluent mesh, the converter automatically extrudes the mesh in the third direction and adds the empty patch, naming it `frontAndBackPlanes`.

The following features should also be observed.

- The OpenFOAM converter will attempt to capture the Fluent boundary condition definition as much as possible; however, since there is no clear, direct correspondence between the OpenFOAM and Fluent boundary conditions, the user should check the boundary conditions before running a case.
- Creation of axi-symmetric meshes from a 2 dimensional mesh is currently not supported but can be implemented on request.
- Multiple material meshes are not permitted. If multiple fluid materials exist, they will be converted into a single OpenFOAM mesh; if a solid region is detected, the converter will attempt to filter it out.
- Fluent allows the user to define a patch which is internal to the mesh, *i.e.* consists of the faces with cells on both sides. Such patches are not allowed in OpenFOAM and the converter will attempt to filter them out.
- There is currently no support for embedded interfaces and refinement trees.

The procedure of converting a Fluent.*msh* file is first to create a new OpenFOAM case by creating the necessary directories/files: the case directory containing a *controlDict* file in a *system* subdirectory. Then at a command prompt the user should execute:

```
fluentMeshToFoam <meshFile>
```

where <`meshFile`> is the name of the *.msh* file, including the full or relative path.

4.5.2 star4ToFoam

This section describes how to convert a mesh generated on the STAR-CD code into a form that can be read by OpenFOAM mesh classes. The mesh can be generated by any of the packages supplied with STAR-CD, *i.e.*PROSTAR, SAMM, ProAM and their derivatives. The converter accepts any single-stream mesh including integral and arbitrary couple matching and all cell types are supported. The features that the converter does not support are:

- multi-stream mesh specification;
- baffles, *i.e.* zero-thickness walls inserted into the domain;
- partial boundaries, where an uncovered part of a couple match is considered to be a boundary face;
- sliding interfaces.

For multi-stream meshes, mesh conversion can be achieved by writing each individual stream as a separate mesh and reassemble them in OpenFOAM.

OpenFOAM adopts a policy of only accepting input meshes that conform to the fairly stringent validity criteria specified in section 4.1. It will simply not run using invalid meshes and cannot convert a mesh that is itself invalid. The following sections describe steps that must be taken when generating a mesh using a mesh generating package supplied with STAR-CD to ensure that it can be converted to OpenFOAM format. To avoid repetition in the remainder of the section, the mesh generation tools supplied with STAR-CD will be referred to by the collective name STAR-CD.

4.5.2.1 General advice on conversion

We strongly recommend that the user run the STAR-CD mesh checking tools before attempting a star4ToFoam conversion and, after conversion, the checkMesh utility should be run on the newly converted mesh. Alternatively, star4ToFoam may itself issue warnings containing PROSTAR commands that will enable the user to take a closer look at cells with problems. Problematic cells and matches should be checked and fixed before attempting to use the mesh with OpenFOAM. Remember that an invalid mesh will not run with OpenFOAM, but it may run in another environment that does not impose the validity criteria.

Some problems of tolerance matching can be overcome by the use of a matching tolerance in the converter. However, there is a limit to its effectiveness and an apparent need to increase the matching tolerance from its default level indicates that the original mesh suffers from inaccuracies.

4.5.2.2 Eliminating extraneous data

When mesh generation in is completed, remove any extraneous vertices and compress the cells boundary and vertex numbering, assuming that fluid cells have been created and all other cells are discarded. This is done with the following PROSTAR commands:

```
CSET NEWS FLUID
CSET INVE
```

The CSET should be empty. If this is not the case, examine the cells in CSET and adjust the model. If the cells are genuinely not desired, they can be removed using the PROSTAR command:

```
CDEL CSET
```

Similarly, vertices will need to be discarded as well:

```
CSET NEWS FLUID
VSET NEWS CSET
VSET INVE
```

Before discarding these unwanted vertices, the unwanted boundary faces have to be collected before purging:

```
CSET NEWS FLUID
VSET NEWS CSET
BSET NEWS VSET ALL
BSET INVE
```

If the BSET is not empty, the unwanted boundary faces can be deleted using:

```
BDEL BSET
```

At this time, the model should contain only the fluid cells and the supporting vertices, as well as the defined boundary faces. All boundary faces should be fully supported by the vertices of the cells, if this is not the case, carry on cleaning the geometry until everything is clean.

4.5.2.3 Removing default boundary conditions

By default, STAR-CD assigns wall boundaries to any boundary faces not explicitly associated with a boundary region. The remaining boundary faces are collected into a `default` boundary region, with the assigned boundary type `0`. OpenFOAM deliberately does not have a concept of a `default` boundary condition for undefined boundary faces since it invites human error, *e.g.* there is no means of checking that we meant to give all the unassociated faces the default condition.

Therefore **all** boundaries for each OpenFOAM mesh must be specified for a mesh to be successfully converted. The `default` boundary needs to be transformed into a real one using the procedure described below:

1. Plot the geometry with `Wire Surface` option.

2. Define an extra boundary region with the same parameters as the `default` region 0 and add all visible faces into the new region, say 10, by selecting a zone option in the boundary tool and drawing a polygon around the entire screen draw of the model. This can be done by issuing the following commands in PROSTAR:

   ```
   RDEF 10 WALL
   BZON 10 ALL
   ```

3. We shall remove all previously defined boundary types from the set. Go through the boundary regions:

   ```
   BSET NEWS REGI 1
   BSET NEWS REGI 2
   ... 3, 4, ...
   ```

   Collect the vertices associated with the boundary set and then the boundary faces associated with the vertices (there will be twice as many of them as in the original set).

   ```
   BSET NEWS REGI 1
   VSET NEWS BSET
   BSET NEWS VSET ALL
   BSET DELE REGI 1
   REPL
   ```

   This should give the faces of boundary Region 10 which have been defined on top of boundary Region 1. Delete them with `BDEL BSET`. Repeat these for all regions.

4.5.2.4 Renumbering the model

Renumber and check the model using the commands:

```
CSET NEW FLUID
CCOM CSET

VSET NEWS CSET
VSET INVE (Should be empty!)
VSET INVE
VCOM VSET

BSET NEWS VSET ALL
BSET INVE (Should be empty also!)
BSET INVE
BCOM BSET

CHECK ALL
GEOM
```

Internal PROSTAR checking is performed by the last two commands, which may reveal some other unforeseeable error(s). Also, take note of the scaling factor because PROSTAR only applies the factor for STAR-CD and not the geometry. If the factor is not 1, use the scalePoints utility in OpenFOAM.

4.5.2.5 Writing out the mesh data

Once the mesh is completed, place all the integral matches of the model into the couple type 1. All other types will be used to indicate arbitrary matches.

```
CPSET NEWS TYPE INTEGRAL
CPMOD CPSET 1
```

The components of the computational grid must then be written to their own files. This is done using PROSTAR for boundaries by issuing the command

```
BWRITE
```

by default, this writes to a *.bnd* file. For cells, the command

```
CWRITE
```

outputs the cells to a *.cel* file and for vertices, the command

```
VWRITE
```

outputs to file a *.vrt* file. The current default setting writes the files in ASCII format. If couples are present, an additional couple file with the extension *.cpl* needs to be written out by typing:

```
CPWRITE
```

After outputting to the three files, exit PROSTAR or close the files. Look through the panels and take note of all STAR-CD sub-models, material and fluid properties used – the material properties and mathematical model will need to be set up by creating and editing OpenFOAM dictionary files.

The procedure of converting the PROSTAR files is first to create a new OpenFOAM case by creating the necessary directories. The PROSTAR files must be stored within the same directory.

4.5.2.6 Converting the mesh to OpenFOAM format

The translator utility star4ToFoam can now be run to create the boundaries, cells and points files necessary for a OpenFOAM run:

```
star4ToFoam <meshFilePrefix>
```

where <meshFilePrefix> is the name of the prefix of the mesh files, including the full or relative path. After the utility has finished running, OpenFOAM boundary types should be specified by editing the *boundary* file by hand.

4.5.3 gambitToFoam

GAMBIT writes mesh data to a single file with a *.neu* extension. The procedure of converting a GAMBIT*.neu* file is first to create a new OpenFOAM case, then at a command prompt, the user should execute:

```
gambitToFoam <meshFile>
```

where <`meshFile`> is the name of the *.neu* file, including the full or relative path.

The GAMBIT file format does not provide information about type of the boundary patch, *e.g.* wall, symmetry plane, cyclic. Therefore all the patches have been created as type patch. Please reset after mesh conversion as necessary.

4.5.4 ansysToFoam

OpenFOAM can convert a mesh generated by I-DEAS but written out in ANSYS format as a *.ans* file. The procedure of converting the *.ans* file is first to create a new OpenFOAM case, then at a command prompt, the user should execute:

```
ansysToFoam <meshFile>
```

where <`meshFile`> is the name of the *.ans* file, including the full or relative path.

Note, the ideasUnvToFoam utility for can convert *.unv* files written by I-DEAS.

4.5.5 cfx4ToFoam

CFX writes mesh data to a single file with a *.geo* extension. The mesh format in CFX is block-structured, *i.e.* the mesh is specified as a set of blocks with glueing information and the vertex locations. OpenFOAM will convert the mesh and capture the CFX boundary condition as best as possible. The 3 dimensional 'patch' definition in CFX, containing information about the porous, solid regions *etc.* is ignored with all regions being converted into a single OpenFOAM mesh. CFX supports the concept of a 'default' patch, where

each external face without a defined boundary condition is treated as a wall. These faces are collected by the converter and put into a `defaultFaces` patch in the OpenFOAM mesh and given the type wall; of course, the patch type can be subsequently changed.

Like, OpenFOAM 2 dimensional geometries in CFX are created as 3 dimensional meshes of 1 cell thickness. If a user wishes to run a 2 dimensional case on a mesh created by CFX, the boundary condition on the front and back planes should be set to empty; the user should ensure that the boundary conditions on all other faces in the plane of the calculation are set correctly. Currently there is no facility for creating an axi-symmetric geometry from a 2 dimensional CFX mesh.

The procedure of converting a CFX.*geo* file is first to create a new OpenFOAM case, then at a command prompt, the user should execute:

```
cfx4ToFoam <meshFile>
```

where <`meshFile`> is the name of the *.geo* file, including the full or relative path.

4.6 Mapping fields between different geometries

The mapFields utility maps one or more fields relating to a given geometry onto the corresponding fields for another geometry. It is completely generalised in so much as there does not need to be any similarity between the geometries to which the fields relate. However, for cases where the geometries are consistent, mapFields can be executed with a special option that simplifies the mapping process.

For our discussion of mapFields we need to define a few terms. First, we say that the data is mapped from the *source* to the *target*. The fields are deemed *consistent* if the geometry *and* boundary types, or conditions, of both source and target fields are identical. The field data that mapFields maps are those fields within the time directory specified by `startFrom`/`startTime` in the *controlDict* of the target case. The data is read from the equivalent time directory of the source case and mapped onto the equivalent time directory of the target case.

4.6.1 Mapping consistent fields

A mapping of consistent fields is simply performed by executing mapFields on the (target) case using the `-consistent` command line option as follows:

```
mapFields <source dir> -consistent
```

4.6.2 Mapping inconsistent fields

When the fields are not consistent, as shown in Figure 4.15, mapFields requires a *mapFieldsDict* dictionary in the *system* directory of the target case. The following rules apply to the mapping:

- the field data is mapped from source to target wherever possible, *i.e.* in our example all the field data within the target geometry is mapped from the source, except those in the shaded region which remain unaltered;
- the patch field data is left unaltered unless specified otherwise in the *mapFieldsDict* dictionary.

The *mapFieldsDict* dictionary contain two lists that specify mapping of patch data. The first list is `patchMap` that specifies mapping of data between pairs of source and target patches that are geometrically coincident, as shown in Figure 4.15. The list contains each pair of names of source and target patch. The second list is `cuttingPatches` that contains names of target patches whose values are to be mapped from the source internal field through which the target patch cuts. In the situation where the target patch only cuts through part of the source internal field, *e.g.* bottom left target patch in our example, those values within the internal field are mapped and those outside remain unchanged. An example *mapFieldsDict* dictionary is shown below:

```
17  patchMap        (lid movingWall);
18
19  cuttingPatches  ();
20
21
22  // ************************************************************************* //
```

```
mapFields <source dir>
```

4.6.3 Mapping parallel cases

If either or both of the source and target cases are decomposed for running in parallel, additional options must be supplied when executing mapFields:

`-parallelSource` if the source case is decomposed for parallel running;

`-parallelTarget` if the target case is decomposed for parallel running.

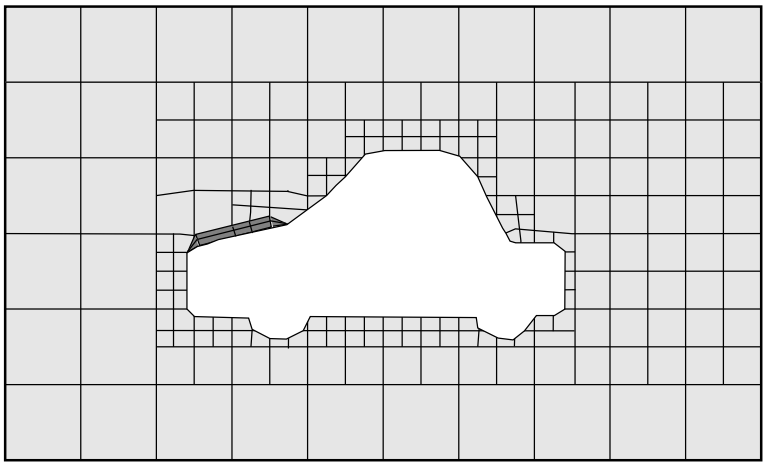


Figure 4.14: Layer addition in snappyHexMesh meshing process

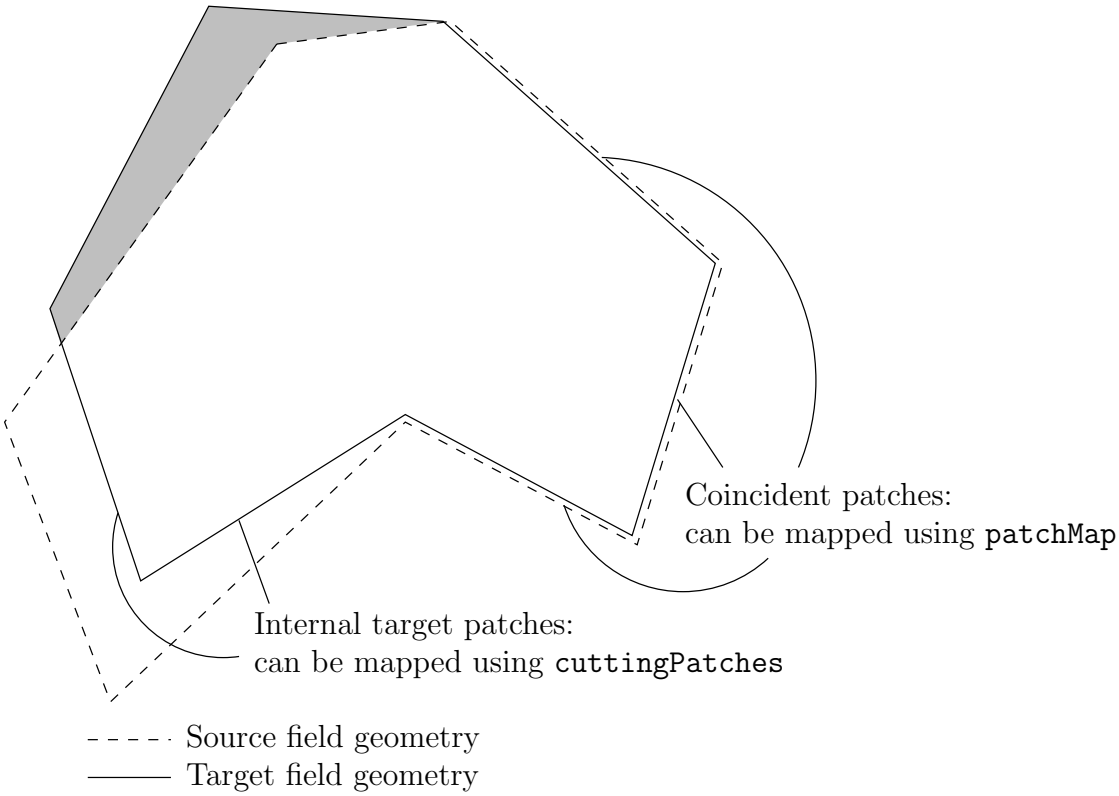


Figure 4.15: Mapping inconsistent fields

Keyword	Description	Example
`layers`	Dictionary of layers	
`relativeSizes`	Are layer thicknesses relative to undistorted cell size outside layer or absolute?	`true/false`
`expansionRatio`	Expansion factor for layer mesh	`1.0`
`finalLayerThickness`	Thickness of layer furthest from the wall, either relative or absolute according to the `relativeSizes` entry	`1`
`firstLayerThickness`	Thickness of layer closest to the wall, either relative or absolute according to	`0.3`
`thickness`	Overall thickness of all layers	`0.3`
`minThickness`	Minimum overall thickness of all layers, below which surface is not extruded	`0.1`
`nGrow`	Number of layers of connected faces that are not grown if points are not extruded; helps convergence of layer addition close to features	`1`
`featureAngle`	Angle above which surface is not extruded	`60`
`maxFaceThickness-Ratio`	Face thickness ratio above which surface is not extruded, useful for warped cells	`0.5`
`nSmoothSurfaceNormals`	Number of smoothing iterations of surface normals	`1`
`nSmoothThickness`	Smooth layer thickness over surface patches	`10`
`minMedialAxisAngle`	Angle used to pick up medial axis points	`90`
`maxThicknessTo-MedialRatio`	Reduce layer growth where ratio thickness to medial distance is large	`0.3`
`maxThicknessTo-MedialRatio`	Reduce layer growth where ratio thickness to medial distance is large	`0.3`
`nSmoothNormals`	Number of smoothing iterations of interior mesh movement direction	`3`
`nRelaxIter`	Maximum number of snapping relaxation iterations	`5`
`nBufferCellsNo-Extrude`	Create buffer region for new layer terminations	`0`
`nLayerIter`	Overall max number of layer addition iterations	`50`
`nRelaxedIter`	Max number of iterations after which the controls in the *relaxed* sub dictionary of `meshQuality` are used	`20`

Table 4.8: Keywords in the *addLayersControls* sub-dictionary of *snappyHexMeshDict*.

Keyword	Description	Example
`maxNonOrtho`	Maximum non-orthogonality allowed; `180` disables	`65`
`maxBoundarySkewness`	Max boundary face skewness allowed; `<0` disables	`20`
`maxInternalSkewness`	Max internal face skewness allowed; `<0` disables	`4`
`maxConcave`	Max concaveness allowed; `180` disables	`80`
`minFlatness`	Ratio of minimum projected area to actual area; `-1` disables	`0.5`
`minVol`	Minimum pyramid volume; large negative number, *e.g.*`-1e30` disables	`1e-13`
`minTetQuality`	Minimum quality of the tetrahedron formed by the face-centre and variable base point minimum decomposition triangles and the cell centre; set to very negative number, *e.g.*`-1e30` to disable	`1e-13`
`minArea`	Minimum face area; `<0` disables	`-1`
`minTwist`	Minimum face twist; `<-1` disables	`0.05`
`minDeterminant`	Minimum normalised cell determinant; `1` = hex; ≤ 0 illegal cell	`0.001`
`minFaceWeight`	`0`→`0.5`	`0.05`
`minVolRatio`	`0`→`1.0`	`0.01`
`minTriangleTwist`	`>0` for Fluent compatability	`-1`
`nSmoothScale`	Number of error distribution iterations	`4`
`errorReduction`	Amount to scale back displacement at error points	`0.75`
`relaxed`	Sub-dictionary that can include modified values for the above keyword entries to be used when `nRelaxedIter` is exceeded in the layer addition process	`relaxed { ... }`

Table 4.9: Keywords in the *meshQualityControls* sub-dictionary of *snappyHexMeshDict*.

Chapter 5

Models and physical properties

OpenFOAM includes a large range of solvers each designed for a specific class of problem. The equations and algorithms differ from one solver to another so that the selection of a solver involves the user making some initial choices on the modelling for their particular case. The choice of solver typically involves scanning through their descriptions in Table A.1 to find the one suitable for the case. It ultimately determines many of the parameters and physical properties required to define the case but leaves the user with some modelling options that can be specified at runtime through the entries in dictionary files in the *constant* directory of a case. This chapter deals with many of the more common models and associated properties that may be specified at runtime.

5.1 Boundary Conditions

Setting appropriate boundary conditions is vital for a successful simulation. Ill-posed boundary conditions will lead to physically incorrect predictions, and in many cases solver failure. Users must specify the boundary conditions for each solved field. The tutorials provided with OpenFOAM show examples of good practice in terms of selection and application for various cases.

Boundary conditions are organised into categories for easier navigation, comprising:

basic basic types

- `fixedValue`
- `fixedGradient`
- `mixed`
- ...

constraint geometrical constraints

- `symmetry`
- `wedge`
- `empty`
- `cyclic`
- ...

derived specialised conditions

- `fixedProfile`: to specify a profile of a variable

- `swirlFlowRateInletVelocity`: to specify velocity inlet for a swirling flow providing flow rate
- `inletOutlet`: outlet condition with handling of reverse flow
- `codedFixedValue`: fixed value set by user coding
- ...

In all there are more than 70 boundary conditions. The list of all available boundary conditions divided into categories based of the use can be found in section A.4

An example pressure field file is shown below

```
17  dimensions      [1 -1 -2 0 0 0 0];
18
19  internalField   uniform 1;
20
21  boundaryField
22  {
23      inlet
24      {
25          type            fixedValue;
26          value           uniform 1;
27      }
28
29      outlet
30      {
31          type            waveTransmissive;
32          field           p;
33          psi             thermo:psi;
34          gamma           1.4;
35          fieldInf        1;
36          lInf            3;
37          value           uniform 1;
38      }
39
40      bottom
41      {
42          type            symmetryPlane;
43      }
44
45      top
46      {
47          type            symmetryPlane;
48      }
49
50      obstacle
51      {
52          type            zeroGradient;
53      }
54
55      defaultFaces
56      {
57          type            empty;
58      }
59  }
60
61
62  // ************************************************************************* //
```

5.2 Thermophysical models

Thermophysical models are used to describe cases where the thermal energy, compressibility or mass transfer is important.

OpenFOAM allows thermophysical properties to be constant, or functions of temperature, pressure and composition. Thermal energy can be described in form of enthalpy or internal energy. The $p - v - T$ relation can be described with various equations of state or as isobaric system.

The *thermophysicalProperties* dictionary is read by any solver that uses the thermophysical model library. A thermophysical model is constructed in OpenFOAM as a pressure-temperature $p - T$ system from which other properties are computed. There

is one compulsory dictionary entry called `thermoType` which specifies the complete thermophysical model that is used in the simulation. The thermophysical modelling starts with a layer that defines the basic equation of state and then adds further layers for the thermodynamic, transport and mixture modelling, as listed in Table 5.1.

Equation of State — equationOfState	
icoPolynomial	Incompressible polynomial equation of state, *e.g.* for liquids
perfectGas	Perfect gas equation of state

Basic thermophysical properties — thermo	
eConstThermo	Constant specific heat c_p model with evaluation of internal energy e and entropy s
hConstThermo	Constant specific heat c_p model with evaluation of enthalpy h and entropy s
hPolynomialThermo	c_p evaluated by a function with coefficients from polynomials, from which h, s are evaluated
janafThermo	c_p evaluated by a function with coefficients from JANAF thermodynamic tables, from which h, s are evaluated

Derived thermophysical properties — specieThermo	
specieThermo	Thermophysical properties of species, derived from c_p, h and/or s

Transport properties — transport	
constTransport	Constant transport properties
polynomialTransport	Polynomial based temperature-dependent transport properties
sutherlandTransport	Sutherland's formula for temperature-dependent transport properties

Mixture properties — mixture	
pureMixture	General thermophysical model calculation for passive gas mixtures
homogeneousMixture	Combustion mixture based on normalised fuel mass fraction b
inhomogeneousMixture	Combustion mixture based on b and total fuel mass fraction f_t
veryInhomogeneousMixture	Combustion mixture based on b, f_t and unburnt fuel mass fraction f_u
dieselMixture	Combustion mixture based on f_t and f_u
basicMultiComponentMixture	Basic mixture based on multiple components
multiComponentMixture	Derived mixture based on multiple components
reactingMixture	Combustion mixture using thermodynamics and reaction schemes
egrMixture	Exhaust gas recirculation mixture

Thermophysical model — thermoModel	
hePsiThermo	General thermophysical model calculation based on enthalpy h or internal energy e, and compressibility ψ

Continued on next page

Continued from previous page

heRhoThermo	General thermophysical model calculation based on enthalpy h or internal energy e, and density ρ
hePsiMixtureThermo	Calculates enthalpy for combustion mixture based on enthalpy h or internal energy e, and ψ
heRhoMixtureThermo	Calculates enthalpy for combustion mixture based on enthalpy h or internal energy e, and ρ
heheuMixtureThermo	Calculates enthalpy h or internal energy e for unburnt u gas and combustion mixture

Table 5.1: Layers of thermophysical modelling.

Various combinations are available as 'packages', specified using, *e.g.*

```
17  thermoType
18  {
19      type            heRhoThermo;
20      mixture         pureMixture;
21      transport       const;
22      thermo          hConst;
23      equationOfState perfectGas;
24      specie          specie;
25      energy          sensibleEnthalpy;
26  }
27
28  mixture
29  {
30      specie
31      {
32          molWeight       28.96;
33      }
34      thermodynamics
35      {
36          Cp              1004.4;
37          Hf              0;
38      }
39      transport
40      {
41          mu              1.831e-05;
42          Pr              0.705;
43      }
44  }
45
46
47  // ************************************************************************* //
```

Only certain combinations are predefined. One method to identify the possible combinations from Table 5.1 is to use a nonexistent setting for one of the entries, *e.g.*`banana` and execute the solver. OpenFOAM will issue an error message and list all possible combinations to the terminal.

5.2.1 Thermophysical property data

The basic thermophysical properties are specified for each species from input data. Data entries must contain the name of the specie as the keyword, *e.g.* `O2`, `H2O`, `mixture`, followed by sub-dictionaries of coefficients, including:

`specie` containing *i.e.* number of moles, `nMoles`, of the specie, and molecular weight, `molWeight` in units of g/mol;

`thermo` containing coefficients for the chosen thermodynamic model (see below);

`transport` containing coefficients for the chosen transport model (see below).

The thermodynamic coefficients are ostensibly concerned with evaluating the specific heat c_p from which other properties are derived. The current thermo models are described as follows:

hConstThermo assumes a constant c_p and a heat of fusion H_f which is simply specified by a two values c_p H_f, given by keywords `Cp` and `Hf`.

eConstThermo assumes a constant c_v and a heat of fusion H_f which is simply specified by a two values c_v H_f, given by keywords `Cv` and `Hf`.

janafThermo calculates c_p as a function of temperature T from a set of coefficients taken from JANAF tables of thermodynamics. The ordered list of coefficients is given in Table 5.2. The function is valid between a lower and upper limit in temperature T_l and T_h respectively. Two sets of coefficients are specified, the first set for temperatures above a common temperature T_c (and below T_h, the second for temperatures below T_c (and above T_l). The function relating c_p to temperature is:

$$c_p = R((((a_4T + a_3)T + a_2)T + a_1)T + a_0) \tag{5.1}$$

In addition, there are constants of integration, a_5 and a_6, both at high and low temperature, used to evaluating h and s respectively.

hPolynomialThermo calculates C_p as a function of temperature by a polynomial of any order. The following case provides an example of its use: $FOAM_TUTORIALS/-lagrangian/porousExplicitSourceReactingParcelFoam/filter

Description	Entry	Keyword
Lower temperature limit	T_l (K)	`Tlow`
Upper temperature limit	T_h (K)	`Thigh`
Common temperature	T_c (K)	`Tcommon`
High temperature coefficients	$a_0 \dots a_4$	`highCpCoeffs (a0 a1 a2 a3 a4...`
High temperature enthalpy offset	a_5	`a5...`
High temperature entropy offset	a_6	`a6)`
Low temperature coefficients	$a_0 \dots a_4$	`lowCpCoeffs (a0 a1 a2 a3 a4...`
Low temperature enthalpy offset	a_5	`a5...`
Low temperature entropy offset	a_6	`a6)`

Table 5.2: JANAF thermodynamics coefficients.

The transport coefficients are used to to evaluate dynamic viscosity μ, thermal conductivity κ and laminar thermal conductivity (for enthalpy equation) α. The current transport models are described as follows:

constTransport assumes a constant μ and Prandtl number $Pr = c_p\mu/\kappa$ which is simply specified by a two keywords, `mu` and `Pr`, respectively.

sutherlandTransport calculates μ as a function of temperature T from a Sutherland coefficient A_s and Sutherland temperature T_s, specified by keywords `As` and `Ts`; μ is calculated according to:

$$\mu = \frac{A_s\sqrt{T}}{1 + T_s/T} \tag{5.2}$$

polynomialTransport calculates μ and κ as a function of temperature T from a polynomial of any order.

The following is an example entry for a specie named `fuel` modelled using sutherlandTransport and janafThermo:

```
fuel
{
    specie
    {
        nMoles       1;
        molWeight    16.0428;
    }
    thermodynamics
    {
        Tlow         200;
        Thigh        6000;
        Tcommon      1000;
        highCpCoeffs (1.63543 0.0100844 -3.36924e-06 5.34973e-10
                      -3.15528e-14 -10005.6 9.9937);
        lowCpCoeffs  (5.14988 -0.013671 4.91801e-05 -4.84744e-08
                      1.66694e-11 -10246.6 -4.64132);
    }
    transport
    {
        As           1.67212e-06;
        Ts           170.672;
    }
}
```

The following is an example entry for a specie named `air` modelled using constTransport and hConstThermo:

```
air
{
    specie
    {
        nMoles          1;
        molWeight       28.96;
    }
    thermodynamics
    {
        Cp              1004.5;
        Hf              2.544e+06;
    }
    transport
    {
        mu              1.8e-05;
        Pr              0.7;
    }
}
```

5.3 Turbulence models

The *turbulenceProperties* dictionary is read by any solver that includes turbulence modelling. Within that file is the `simulationType` keyword that controls the type of turbulence modelling to be used, either:

`laminar` uses no turbulence models;

`RAS` uses Reynolds-averaged stress (RAS) modelling;

`LES` uses large-eddy simulation (LES) or detached-eddy simulation (DES) modelling.

If `RAS` is selected, the choice of RAS modelling is specified in a *RAS* subdictionary. The RAS turbulence model is selected by the `RASModel` entry from a long list of available models that are listed in Table A.5. Similarly, if `LES` is selected, the choice of LES modelling is specified in a *LES* subdictionary and the LES turbulence model is selected by the `LESModel` entry. Note that `DES` models are defined as a subset of the available LES models.

The entries required in the *RAS* subdictionary are listed in Table 5.3 and those for the *LES* subdictionary are listed in Table 5.4.

`RASModel`	Name of RAS turbulence model
`turbulence`	Switch to turn turbulence modelling on/off
`printCoeffs`	Switch to print model coeffs to terminal at simulation startup
`<RASModel>Coeffs`	Optional dictionary of coefficients for the respective `RASModel`

Table 5.3: Keyword entries in the *RAS* dictionary.

`LESModel`	Name of LES model
`delta`	Name of delta δ model
`<LESModel>Coeffs`	Dictionary of coefficients for the respective `LESModel`
`<delta>Coeffs`	Dictionary of coefficients for each `delta` model

Table 5.4: Keyword entries in the *LES* dictionary.

The incompressible and compressible RAS turbulence models, isochoric and anisochoric LES models and delta models are all named and described in Table A.5. Examples of their use can be found in the *$FOAM_TUTORIALS*.

5.3.1 Model coefficients

The coefficients for the RAS turbulence models are given default values in their respective source code. If the user wishes to override these default values, then they can do so by adding a sub-dictionary entry to the *RAS* dictionary, whose keyword name is that of the model with `Coeffs` appended, *e.g.* `kEpsilonCoeffs` for the `kEpsilon` model. If the `printCoeffs` switch is `on` an example of the relevant `...Coeffs` dictionary is printed to standard output when the model is created at the beginning of a run. The user can simply copy this into the *RAS* dictionary and edit the entries as required.

5.3.2 Wall functions

A range of wall function models is available in OpenFOAM that are applied as boundary conditions on individual patches. This enables different wall function models to be applied to different wall regions. The choice of wall function model is specified through ν_t in the *0/nut* file. For example, a *0/nut* file:

```
17  dimensions      [0 2 -1 0 0 0 0];
18
19  internalField   uniform 0;
20
21  boundaryField
22  {
23      "(movingWall|fixedWalls)"
24      {
25          type            nutkWallFunction;
26          value           uniform 0;
27      }
28
29      frontAndBack
30      {
31          type            empty;
32      }
33  }
34
35
36  // ************************************************************************* //
```

There are a number of wall function models available in the release, *e.g.* `nutkWallFunction`, `nutUWallFunction`, `nutUSpaldingWallFunction`. The user can consult the relevant directories for a full list of wall function models:

```
find $FOAM_SRC/TurbulenceModels -name wallFunctions
```

Within each wall function boundary condition the user can over-ride default settings for E, κ and C_μ through optional `E`, `kappa` and `Cmu` keyword entries.

Having selected the particular wall functions on various patches in the *nut*/*mut* file, the user should select epsilonWallFunction on corresponding patches in the *epsilon* field and kqRwallFunction on corresponding patches in the turbulent fields *k*, *q* and *R*.

Further details on implementation and usage are available in the Extended Code Guide.

Chapter 6

Solving

This chapter describes how to solve and manage OpenFOAM cases, including options to control the time and output behaviour, numerical schemes, solvers, and how to monitor solution progress.

6.1 Time and data input/output control

The OpenFOAM solvers begin all runs by setting up a database. The database controls I/O and, since output of data is usually requested at intervals of time during the run, time is an inextricable part of the database. The *controlDict* dictionary sets input parameters *essential* for the creation of the database. The keyword entries in *controlDict* are listed in Table 6.1. Only the time control and `writeInterval` entries are truly compulsory, with the database taking default values indicated by † in Table 6.1 for any of the optional entries that are omitted.

Time control	
`startFrom`	Controls the start time of the simulation.
- `firstTime`	Earliest time step from the set of time directories.
- `startTime`	Time specified by the `startTime` keyword entry.
- `latestTime`	Most recent time step from the set of time directories.
`startTime`	Start time for the simulation with `startFrom startTime;`
`stopAt`	Controls the end time of the simulation.
- `endTime`	Time specified by the `endTime` keyword entry.
- `writeNow`	Stops simulation on completion of current time step and writes data.
- `noWriteNow`	Stops simulation on completion of current time step and does not write out data.
- `nextWrite`	Stops simulation on completion of next scheduled write time, specified by `writeControl`.
`endTime`	End time for the simulation when `stopAt endTime;` is specified.
`deltaT`	Time step of the simulation.
Time step control	

Continued on next page

Continued from previous page

`adjustTimeStep`	`yes`/`no`† to adjust time step according to maximum Courant number in transient simulation.
`maxCo`	Maximum Courant number allowed.
`maxDeltaT`	Maximum time step allowed in transient simulation.

Data writing

`writeControl`	Controls the timing of write output to file.
- `timeStep`†	Writes data every `writeInterval` time steps.
- `runTime`	Writes data every `writeInterval` seconds of simulated time.
- `adjustableRunTime`	Writes data every `writeInterval` seconds of simulated time, adjusting the time steps to coincide with the `writeInterval` if necessary — used in cases with automatic time step adjustment.
- `cpuTime`	Writes data every `writeInterval` seconds of CPU time.
- `clockTime`	Writes data out every `writeInterval` seconds of real time.
`writeInterval`	Scalar used in conjunction with `writeControl` described above.
`purgeWrite`	Integer representing a limit on the number of time directories that are stored by overwriting time directories on a cyclic basis. Example of $t_0 = 5$s, $\Delta t = 1$s and `purgeWrite 2;`: data written into 2 directories, *6* and *7*, before returning to write the data at 8 s in *6*, data at 9 s into *7*, *etc.* *To disable the time directory limit, specify* `purgeWrite 0;`† For steady-state solutions, results from previous iterations can be continuously overwritten by specifying `purgeWrite 1;`
`writeFormat`	Specifies the format of the data files.
- `ascii`†	ASCII format, written to `writePrecision` significant figures.
- `binary`	Binary format.
`writePrecision`	Integer used in conjunction with `writeFormat` described above, 6† by default
`writeCompression`	Specifies the compression of the data files.
- `uncompressed`	No compression.†
- `compressed`	`gzip` compression.
`timeFormat`	Choice of format of the naming of the time directories.
- `fixed`	$\pm m.dddddd$ where the number of ds is set by `timePrecision`.
- `scientific`	$\pm m.ddddde\pm xx$ where the number of ds is set by `timePrecision`.
- `general`†	Specifies `scientific` format if the exponent is less than -4 or greater than or equal to that specified by `timePrecision`.
`timePrecision`	Integer used in conjunction with `timeFormat` described above, 6† by default
`graphFormat`	Format for graph data written by an application.

Continued on next page

Continued from previous page

- `raw`†	Raw ASCII format in columns.
- `gnuplot`	Data in gnuplot format.
- `xmgr`	Data in Grace/xmgr format.
- `jplot`	Data in jPlot format.

Data reading

`runTimeModifiable`	`yes`†/`no` switch for whether dictionaries, *e.g.controlDict*, are re-read by OpenFOAM at the beginning of each time step.

Run-time loadable functionality

`libs`	List of additional libraries (on $LD_LIBRARY_PATH) to be loaded at run-time, *e.g.*`( "libUser1.so" "libUser2.so" )`
`functions`	List of functions, *e.g.* `probes` to be loaded at run-time; see examples in *$FOAM_TUTORIALS*

† denotes default entry if associated keyword is omitted.

Table 6.1: Keyword entries in the *controlDict* dictionary.

Example entries from a *controlDict* dictionary are given below:

```
17  application     icoFoam;
18
19  startFrom       startTime;
20
21  startTime       0;
22
23  stopAt          endTime;
24
25  endTime         0.5;
26
27  deltaT          0.005;
28
29  writeControl    timeStep;
30
31  writeInterval   20;
32
33  purgeWrite      0;
34
35  writeFormat     ascii;
36
37  writePrecision  6;
38
39  writeCompression off;
40
41  timeFormat      general;
42
43  timePrecision   6;
44
45  runTimeModifiable true;
46
47
48  // ************************************************************************* //
```

6.2 Numerical schemes

The *fvSchemes* dictionary in the *system* directory sets the numerical schemes for terms, such as derivatives in equations, that appear in applications being run. This section describes how to specify the schemes in the *fvSchemes* dictionary.

The terms that must typically be assigned a numerical scheme in *fvSchemes* range from derivatives, *e.g.* gradient ∇, and interpolations of values from one set of points to another.

The aim in OpenFOAM is to offer an unrestricted choice to the user. For example, while linear interpolation is effective in many cases, OpenFOAM offers complete freedom to choose from a wide selection of interpolation schemes for all interpolation terms.

The derivative terms further exemplify this freedom of choice. The user first has a choice of discretisation practice where standard Gaussian finite volume integration is the common choice. Gaussian integration is based on summing values on cell faces, which must be interpolated from cell centres. The user again has a completely free choice of interpolation scheme, with certain schemes being specifically designed for particular derivative terms, especially the convection divergence $\nabla \bullet$ terms.

The set of terms, for which numerical schemes must be specified, are subdivided within the *fvSchemes* dictionary into the categories listed in Table 6.2. Each keyword in Table 6.2 is the name of a sub-dictionary which contains terms of a particular type, *e.g.*`gradSchemes` contains all the gradient derivative terms such as `grad(p)` (which represents ∇p). Further examples can be seen in the extract from an *fvSchemes* dictionary below:

Keyword	Category of mathematical terms
`interpolationSchemes`	Point-to-point interpolations of values
`snGradSchemes`	Component of gradient normal to a cell face
`gradSchemes`	Gradient ∇
`divSchemes`	Divergence $\nabla \bullet$
`laplacianSchemes`	Laplacian ∇^2
`timeScheme`	First and second time derivatives $\partial/\partial t, \partial^2/\partial^2 t$

Table 6.2: Main keywords used in *fvSchemes*.

```
17  ddtSchemes
18  {
19      default         Euler;
20  }
21
22  gradSchemes
23  {
24      default         Gauss linear;
25      grad(p)         Gauss linear;
26  }
27
28  divSchemes
29  {
30      default         none;
31      div(phi,U)      Gauss linear;
32  }
33
34  laplacianSchemes
35  {
36      default         Gauss linear orthogonal;
37  }
38
39  interpolationSchemes
40  {
41      default         linear;
42  }
43
44  snGradSchemes
45  {
46      default         orthogonal;
47  }
48
49
50  // ************************************************************************* //
```

The example shows that the *fvSchemes* dictionary comprises ... *Schemes* sub-dictionaries containing keyword entries for each term specified within, including: a `default` entry; other entries whose names correspond to a word identifier for the particular term specified, *e.g.*`grad(p)` for ∇p

If a `default` scheme is specified in a particular ... *Schemes* sub-dictionary, it is assigned to all of the terms to which the sub-dictionary refers, *e.g.* specifying a `default` in *gradSchemes* sets the scheme for all gradient terms in the application, *e.g.* ∇p, $\nabla \mathbf{U}$. When a `default` is specified, it is not necessary to specify each specific term itself in that sub-dictionary, *i.e.* the entries for `grad(p)`, `grad(U)` in this example. However, if any of these terms are included, the specified scheme overrides the `default` scheme for that term.

Alternatively the user may insist on no `default` scheme by the `none` entry. In this instance the user is obliged to specify all terms in that sub-dictionary individually. Setting `default` to `none` may appear superfluous since `default` can be overridden. However, specifying `none` forces the user to specify all terms individually which can be useful to remind the user which terms are actually present in the application.

The following sections describe the choice of schemes for each of the categories of terms in Table 6.2.

6.2.1 Interpolation schemes

The *interpolationSchemes* sub-dictionary contains terms that are interpolations of values typically from cell centres to face centres. A *selection* of interpolation schemes in OpenFOAM are listed in Table 6.3, being divided into 4 categories: 1 category of general schemes; and, 3 categories of schemes used primarily in conjunction with Gaussian discretisation of convection (divergence) terms in fluid flow, described in section 6.2.5. It is *highly unlikely* that the user would adopt any of the convection-specific schemes for general field interpolations in the *interpolationSchemes* sub-dictionary, but, as valid interpolation schemes, they are described here rather than in section 6.2.5. Note that additional schemes such as `UMIST` are available in OpenFOAM but only those schemes that are generally recommended are listed in Table 6.3.

A general scheme is simply specified by quoting the keyword and entry, *e.g.* a `linear` scheme is specified as `default` by:

```
default linear;
```

The convection-specific schemes calculate the interpolation based on the flux of the flow velocity. The specification of these schemes requires the name of the flux field on which the interpolation is based; in most OpenFOAM applications this is `phi`, the name commonly adopted for the surfaceScalarField velocity flux ϕ. The 3 categories of convection-specific schemes are referred to in this text as: general convection; normalised variable (NV); and, total variation diminishing (TVD). With the exception of the `blended` scheme, the general convection and TVD schemes are specified by the scheme and flux, *e.g.* an `upwind` scheme based on a flux `phi` is specified as `default` by:

```
default upwind phi;
```

Some TVD/NVD schemes require a coefficient $\psi, 0 \leq \psi \leq 1$ where $\psi = 1$ corresponds to TVD conformance, usually giving best convergence and $\psi = 0$ corresponds to best accuracy. Running with $\psi = 1$ is generally recommended. A `limitedLinear` scheme based on a flux `phi` with $\psi = 1.0$ is specified as `default` by:

```
default limitedLinear phi 1.0;
```

6.2.1.1 Schemes for strictly bounded scalar fields

There are enhanced versions of some of the limited schemes for scalars that need to be strictly bounded. To bound between user-specified limits, the scheme name should be prepended by the word `limited` and followed by the lower and upper limits respectively. For example, to bound the `vanLeer` scheme strictly between -2 and 3, the user would specify:

```
default limitedVanLeer -2.0 3.0;
```

There are specialised versions of these schemes for scalar fields that are commonly bounded between 0 and 1. These are selected by adding `01` to the name of the scheme. For example, to bound the `vanLeer` scheme strictly between 0 and 1, the user would specify:

```
default vanLeer01;
```

Strictly bounded versions are available for the following schemes: `limitedLinear`, `vanLeer`, `Gamma`, `limitedCubic`, `MUSCL` and `SuperBee`.

6.2.1.2 Schemes for vector fields

There are improved versions of some of the limited schemes for vector fields in which the limiter is formulated to take into account the direction of the field. These schemes are selected by adding `V` to the name of the general scheme, *e.g.*`limitedLinearV` for `limitedLinear`. 'V' versions are available for the following schemes: `limitedLinearV`, `vanLeerV`, `GammaV`, `limitedCubicV` and `SFCDV`.

Centred schemes	
`linear`	Linear interpolation (central differencing)
`cubicCorrection`	Cubic scheme
`midPoint`	Linear interpolation with symmetric weighting
Upwinded convection schemes	
`upwind`	Upwind differencing
`linearUpwind`	Linear upwind differencing
`skewLinear`	Linear with skewness correction
`filteredLinear2`	Linear with filtering for high-frequency ringing
TVD schemes	
`limitedLinear`	limited linear differencing
`vanLeer`	van Leer limiter
`MUSCL`	MUSCL limiter
`limitedCubic`	Cubic limiter
NVD schemes	
`SFCD`	Self-filtered central differencing
`Gamma` ψ	Gamma differencing

Table 6.3: Interpolation schemes.

6.2.2 Surface normal gradient schemes

The *snGradSchemes* sub-dictionary contains surface normal gradient terms. A surface normal gradient is evaluated at a cell face; it is the component, normal to the face, of the gradient of values at the centres of the 2 cells that the face connects. A surface normal gradient may be specified in its own right and is also required to evaluate a Laplacian term using Gaussian integration.

The available schemes are listed in Table 6.4 and are specified by simply quoting the keyword and entry, with the exception of `limited` which requires a coefficient $\psi, 0 \leq \psi \leq 1$ where

$$\psi = \begin{cases} 0 & \text{corresponds to } \texttt{uncorrected}, \\ 0.333 & \text{non-orthogonal correction} \leq 0.5 \times \text{orthogonal part}, \\ 0.5 & \text{non-orthogonal correction} \leq \text{orthogonal part}, \\ 1 & \text{corresponds to } \texttt{corrected}. \end{cases} \tag{6.1}$$

A `limited` scheme with $\psi = 0.5$ is therefore specified as `default` by:

```
default limited 0.5;
```

Scheme	Description
`corrected`	Explicit non-orthogonal correction
`uncorrected`	No non-orthogonal correction
`limited` ψ	Limited non-orthogonal correction
`bounded`	Bounded correction for positive scalars
`fourth`	Fourth order

Table 6.4: Surface normal gradient schemes.

6.2.3 Gradient schemes

The *gradSchemes* sub-dictionary contains gradient terms. The discretisation scheme for each term can be selected from those listed in Table 6.5.

Discretisation scheme	Description
`Gauss <interpolationScheme>`	Second order, Gaussian integration
`leastSquares`	Second order, least squares
`fourth`	Fourth order, least squares
`cellLimited <gradScheme>`	Cell limited version of one of the above schemes
`faceLimited <gradScheme>`	Face limited version of one of the above schemes

Table 6.5: Discretisation schemes available in *gradSchemes*.

The discretisation scheme is sufficient to specify the scheme completely in the cases of `leastSquares` and `fourth`, *e.g.*

```
grad(p) leastSquares;
```

The `Gauss` keyword specifies the standard finite volume discretisation of Gaussian integration which requires the interpolation of values from cell centres to face centres. Therefore, the `Gauss` entry must be followed by the choice of interpolation scheme from Table 6.3. It would be extremely unusual to select anything other than general interpolation schemes and in most cases the `linear` scheme is an effective choice, *e.g.*

```
grad(p) Gauss linear;
```

Limited versions of any of the 3 base gradient schemes — `Gauss`, `leastSquares` and `fourth` — can be selected by preceding the discretisation scheme by `cellLimited` (or `faceLimited`), *e.g.* a cell limited Gauss scheme

```
grad(p) cellLimited Gauss linear 1;
```

6.2.4 Laplacian schemes

The *laplacianSchemes* sub-dictionary contains Laplacian terms. Let us discuss the syntax of the entry in reference to a typical Laplacian term found in fluid dynamics, $\nabla \bullet (\nu \nabla \mathbf{U})$, given the word identifier `laplacian(nu,U)`. The `Gauss` scheme is the only choice of discretisation and requires a selection of both an interpolation scheme for the diffusion coefficient, *i.e.* ν in our example, and a surface normal gradient scheme, *i.e.* $\nabla \mathbf{U}$. To summarise, the entries required are:

```
Gauss <interpolationScheme> <snGradScheme>
```

The interpolation scheme is selected from Table 6.3, the typical choices being from the general schemes and, in most cases, `linear`. The surface normal gradient scheme is selected from Table 6.4; the choice of scheme determines numerical behaviour as described in Table 6.6. A typical entry for our example Laplacian term would be:

```
laplacian(nu,U) Gauss linear corrected;
```

Scheme	Numerical behaviour
`corrected`	Unbounded, second order, conservative
`uncorrected`	Bounded, first order, non-conservative
`limited` ψ	Blend of `corrected` and `uncorrected`
`bounded`	First order for bounded scalars
`fourth`	Unbounded, fourth order, conservative

Table 6.6: Behaviour of surface normal schemes used in *laplacianSchemes*.

6.2.5 Divergence schemes

The *divSchemes* sub-dictionary contains divergence terms. Let us discuss the syntax of the entry in reference to a typical convection term found in fluid dynamics $\nabla \bullet (\rho \mathbf{U}\mathbf{U})$, which in OpenFOAM applications is commonly given the identifier `div(phi,U)`, where `phi` refers to the flux $\phi = \rho \mathbf{U}$.

The `Gauss` scheme is the only choice of discretisation and requires a selection of the interpolation scheme for the dependent field, *i.e.* $\mathbf{U}$ in our example. To summarise, the entries required are:

```
Gauss <interpolationScheme>
```

The interpolation scheme is selected from the full range of schemes in Table 6.3, both general and convection-specific. The choice critically determines numerical behaviour as described in Table 6.7. The syntax here for specifying convection-specific interpolation schemes *does not include the flux* as it is already known for the particular term, *i.e.* for `div(phi,U)`, we know the flux is `phi` so specifying it in the interpolation scheme would only invite an inconsistency. Specification of upwind interpolation in our example would therefore be:

```
div(phi,U) Gauss upwind;
```

Scheme	Numerical behaviour
`linear`	Second order, unbounded
`skewLinear`	Second order, (more) unbounded, skewness correction
`cubicCorrected`	Fourth order, unbounded
`upwind`	First order, bounded
`linearUpwind`	First/second order, bounded
`QUICK`	First/second order, bounded
TVD schemes	First/second order, bounded
`SFCD`	Second order, bounded
NVD schemes	First/second order, bounded

Table 6.7: Behaviour of interpolation schemes used in *divSchemes*.

6.2.6 Time schemes

The first time derivative $(\partial/\partial t)$ terms are specified in the *ddtSchemes* sub-dictionary. The discretisation scheme for each term can be selected from those listed in Table 6.8.

There is an off-centering coefficient ψ with the `CrankNicholson` scheme that blends it with the `Euler` scheme. A coefficient of $\psi = 1$ corresponds to pure `CrankNicholson` and and $\psi = 0$ corresponds to pure `Euler`. The blending coefficient can help to improve stability in cases where pure `CrankNicholson` are unstable.

Scheme	Description
`Euler`	First order, bounded, implicit
`localEuler`	Local-time step, first order, bounded, implicit
`CrankNicholson` ψ	Second order, bounded, implicit
`backward`	Second order, implicit
`steadyState`	Does not solve for time derivatives

Table 6.8: Discretisation schemes available in *ddtSchemes*.

When specifying a time scheme it must be noted that an application designed for transient problems will not necessarily run as steady-state and visa versa. For example the solution will not converge if `steadyState` is specified when running icoFoam, the transient, laminar incompressible flow code; rather, simpleFoam should be used for steady-state, incompressible flow.

Any second time derivative ($\partial^2/\partial t^2$) terms are specified in the *d2dt2Schemes* sub-dictionary. Only the `Euler` scheme is available for *d2dt2Schemes*.

6.3 Solution and algorithm control

The equation solvers, tolerances and algorithms are controlled from the *fvSolution* dictionary in the *system* directory. Below is an example set of entries from the *fvSolution* dictionary required for the icoFoam solver.

```
17  solvers
18  {
19      p
20      {
21          solver          PCG;
22          preconditioner  DIC;
23          tolerance       1e-06;
24          relTol          0.05;
25      }
26
27      pFinal
28      {
29          $p;
30          relTol          0;
31      }
32
33      U
34      {
35          solver          smoothSolver;
36          smoother        symGaussSeidel;
37          tolerance       1e-05;
38          relTol          0;
39      }
40  }
41
42  PISO
43  {
44      nCorrectors     2;
45      nNonOrthogonalCorrectors 0;
46      pRefCell        0;
47      pRefValue       0;
48  }
49
50
51  // ************************************************************************* //
```

fvSolution contains a set of subdictionaries that are specific to the solver being run. However, there is a small set of standard subdictionaries that cover most of those used by the standard solvers. These subdictionaries include *solvers*, *relaxationFactors*, *PISO* and *SIMPLE* which are described in the remainder of this section.

6.3.1 Linear solver control

The first sub-dictionary in our example, and one that appears in all solver applications, is `solvers`. It specifies each linear-solver that is used for each discretised equation; it is emphasised that the term *linear*-solver refers to the method of number-crunching to solve the set of linear equations, as opposed to *application* solver which describes the set of equations and algorithms to solve a particular problem. The term 'linear-solver' is abbreviated to 'solver' in much of the following discussion; we hope the context of the term avoids any ambiguity.

The syntax for each entry within *solvers* uses a keyword that is the word relating to the variable being solved in the particular equation. For example, icoFoam solves equations for velocity **U** and pressure p, hence the entries for `U` and `p`. The keyword is followed by a dictionary containing the type of solver and the parameters that the solver uses. The solver is selected through the `solver` keyword from the choice in OpenFOAM, listed

Solver	Keyword
Preconditioned (bi-)conjugate gradient	`PCG`/`PBiCG`†
Stabilized Preconditioned (bi-)conjugate gradient (recommended over PCG/PBiCG)	`PBiCGStab`
Solver using a smoother	`smoothSolver`
Generalised geometric-algebraic multi-grid	`GAMG`
Diagonal solver for explicit systems	`diagonal`

†`PCG` for symmetric matrices, `PBiCG` for asymmetric

Table 6.9: Linear solvers.

in Table 6.9. The parameters, including `tolerance`, `relTol`, `preconditioner`, *etc.* are described in following sections.

The solvers distinguish between symmetric matrices and asymmetric matrices. The symmetry of the matrix depends on the structure of the equation being solved and, while the user may be able to determine this, it is not essential since OpenFOAM will produce an error message to advise the user if an inappropriate solver has been selected, *e.g.*

```
--> FOAM FATAL IO ERROR : Unknown asymmetric matrix solver PCG
Valid asymmetric matrix solvers are :
3
(
PBiCG
PBiCGStab
smoothSolver
GAMG
)
```

6.3.1.1 Solution tolerances

The sparse matrix solvers are iterative, *i.e.* they are based on reducing the equation residual over a succession of solutions. The residual is ostensibly a measure of the error in the solution so that the smaller it is, the more accurate the solution. More precisely, the residual is evaluated by substituting the current solution into the equation and taking the magnitude of the difference between the left and right hand sides; it is also normalised in to make it independent of the scale of problem being analysed.

Before solving an equation for a particular field, the initial residual is evaluated based on the current values of the field. After each solver iteration the residual is re-evaluated. The solver stops if *either* of the following conditions are reached:

- the residual falls below the *solver tolerance*, `tolerance`;
- the ratio of current to initial residuals falls below the *solver relative tolerance*, `relTol`;
- the number of iterations exceeds a *maximum number of iterations*, `maxIter`;

The solver tolerance should represent the level at which the residual is small enough that the solution can be deemed sufficiently accurate. The solver relative tolerance limits the relative improvement from initial to final solution. In transient simulations, it is usual to set the solver relative tolerance to 0 to force the solution to converge to the solver tolerance in each time step. The tolerances, `tolerance` and `relTol` must be specified in the dictionaries for all solvers; `maxIter` is optional.

6.3.1.2 Preconditioned conjugate gradient solvers

There are a range of options for preconditioning of matrices in the conjugate gradient solvers, represented by the `preconditioner` keyword in the solver dictionary. The preconditioners are listed in Table 6.10.

Preconditioner	Keyword
Diagonal incomplete-Cholesky (symmetric)	`DIC`
Faster diagonal incomplete-Cholesky (`DIC` with caching)	`FDIC`
Diagonal incomplete-LU (asymmetric)	`DILU`
Diagonal	`diagonal`
Geometric-algebraic multi-grid	`GAMG`
No preconditioning	`none`

Table 6.10: Preconditioner options.

6.3.1.3 Smooth solvers

The solvers that use a smoother require the smoother to be specified. The smoother options are listed in Table 6.11. Generally `GaussSeidel` is the most reliable option, but for bad matrices `DIC` can offer better convergence. In some cases, additional post-smoothing using `GaussSeidel` is further beneficial, *i.e.* the method denoted as `DICGaussSeidel`

Smoother	Keyword
Gauss-Seidel	`GaussSeidel`
Diagonal incomplete-Cholesky (symmetric)	`DIC`
Diagonal incomplete-Cholesky with Gauss-Seidel (symmetric)	`DICGaussSeidel`

Table 6.11: Smoother options.

The user must also specify the number of sweeps, by the `nSweeps` keyword, before the residual is recalculated, following the tolerance parameters.

6.3.1.4 Geometric-algebraic multi-grid solvers

The generalised method of geometric-algebraic multi-grid (GAMG) uses the principle of: generating a quick solution on a mesh with a small number of cells; mapping this solution onto a finer mesh; using it as an initial guess to obtain an accurate solution on the fine mesh. GAMG is faster than standard methods when the increase in speed by solving first on coarser meshes outweighs the additional costs of mesh refinement and mapping of field data. In practice, GAMG starts with the mesh specified by the user and coarsens/refines the mesh in stages. The user is only required to specify an approximate mesh size at the most coarse level in terms of the number of cells `nCoarsestCells`.

The agglomeration of cells is performed by the algorithm specified by the `agglomerator` keyword. Presently we recommend the `faceAreaPair` method. It is worth noting there is an `MGridGen` option that requires an additional entry specifying the shared object library for `MGridGen`:

```
geometricGamgAgglomerationLibs ("libMGridGenGamgAgglomeration.so");
```

In the experience of OpenCFD, the MGridGen method offers no obvious benefit over the `faceAreaPair` method. For all methods, agglomeration can be optionally cached by the `cacheAgglomeration` switch.

Smoothing is specified by the `smoother` as described in section 6.3.1.3. The number of sweeps used by the smoother at different levels of mesh density are specified by the `nPreSweeps`, `nPostSweeps` and `nFinestSweeps` keywords. The `nPreSweeps` entry is used as the algorithm is coarsening the mesh, `nPostSweeps` is used as the algorithm is refining, and `nFinestSweeps` is used when the solution is at its finest level.

The `mergeLevels` keyword controls the speed at which coarsening or refinement levels is performed. It is often best to do so only at one level at a time, *i.e.* set `mergeLevels 1`. In some cases, particularly for simple meshes, the solution can be safely speeded up by coarsening/refining two levels at a time, *i.e.* setting `mergeLevels 2`.

6.3.2 Solution under-relaxation

A second sub-dictionary of *fvSolution* that is often used in OpenFOAM is *relaxationFactors* which controls under-relaxation, a technique used for improving stability of a computation, particularly in solving steady-state problems. Under-relaxation works by limiting the amount which a variable changes from one iteration to the next, either by modifying the solution matrix and source prior to solving for a field or by modifying the field directly. An under-relaxation factor $\alpha, 0 < \alpha \leq 1$ specifies the amount of under-relaxation, ranging from none at all for $\alpha = 1$ and increasing in strength as $\alpha \to 0$. The limiting case where $\alpha = 0$ represents a solution which does not change at all with successive iterations. An optimum choice of α is one that is small enough to ensure stable computation but large enough to move the iterative process forward quickly; values of α as high as 0.9 can ensure stability in some cases and anything much below, say, 0.2 are prohibitively restrictive in slowing the iterative process.

OpenFOAM includes two variants of the SIMPLE algorithm, standard SIMPLE and its consistent formulation, SIMPLEC. By default SIMPLE is used. To use SIMPLEC, the switch

```
consistent      yes;
```

must be set in the *SIMPLE* subdirectory of the *fvSolution* dictionary The SIMPLEC formulation for the pressure-velocity coupling method needs only a small amount of under-relaxation for velocity and other transport equations. There is no need to use any relaxation on pressure. This results typically in more robust solution and faster convergence.

The user can specify the relaxation factor for a particular field by specifying first the word associated with the field, then the factor. The user can view the relaxation factors used in a tutorial example of simpleFoam for incompressible, laminar, steady-state flows.

```
17  solvers
18  {
19      p
20      {
21          solver          GAMG;
22          tolerance       1e-06;
23          relTol          0.1;
24          smoother        GaussSeidel;
25      }
26
27      "(U|k|epsilon|omega|f|v2)"
28      {
29          solver          smoothSolver;
30          smoother        symGaussSeidel;
31          tolerance       1e-05;
32          relTol          0.1;
33      }
```

```
34  }
35
36  SIMPLE
37  {
38      nNonOrthogonalCorrectors 0;
39      consistent      yes;
40
41      residualControl
42      {
43          p               1e-2;
44          U               1e-3;
45          "(k|epsilon|omega|f|v2)" 1e-3;
46      }
47  }
48
49  relaxationFactors
50  {
51      equations
52      {
53          U               0.9; // 0.9 is more stable but 0.95 more convergent
54          ".*"            0.9; // 0.9 is more stable but 0.95 more convergent
55      }
56  }
57
58
59  // ************************************************************************* //
```

6.3.3 PISO and SIMPLE algorithms

Most fluid dynamics solver applications in OpenFOAM use the pressure-implicit split-operator (PISO) or semi-implicit method for pressure-linked equations (SIMPLE) algorithms. These algorithms are iterative procedures for solving equations for velocity and pressure, PISO being used for transient problems and SIMPLE for steady-state.

Both algorithms are based on evaluating some initial solutions and then correcting them. SIMPLE only makes 1 correction whereas PISO requires more than 1, but typically not more than 4. The user must therefore specify the number of correctors in the PISO dictionary by the `nCorrectors` keyword as shown in the example on page U-84.

An additional correction to account for mesh non-orthogonality is available in both SIMPLE and PISO in the standard OpenFOAM solver applications. A mesh is orthogonal if, for each face within it, the face normal is parallel to the vector between the centres of the cells that the face connects, *e.g.* a mesh of hexahedral cells whose faces are aligned with a Cartesian coordinate system. The number of non-orthogonal correctors is specified by the `nNonOrthogonalCorrectors` keyword as shown in the examples above and on page U-84. The number of non-orthogonal correctors should correspond to the mesh for the case being solved, *i.e.* 0 for an orthogonal mesh and increasing with the degree of non-orthogonality up to, say, 20 for the most non-orthogonal meshes.

6.3.3.1 Pressure referencing

In a closed incompressible system, pressure is relative: it is the pressure range that matters not the absolute values. In these cases, the solver sets a reference level of `pRefValue` in cell `pRefCell` where `p` is the name of the pressure solution variable. Where the pressure is `p_rgh`, the names are `p_rhgRefValue` and `p_rhgRefCell` respectively. These entries are generally stored in the *PISO*/*SIMPLE* sub-dictionary and are used by those solvers that require them when the case demands it. If omitted, the solver will not run, but give a message to alert the user to the problem.

6.3.4 Other parameters

The *fvSolutions* dictionaries in the majority of standard OpenFOAM solver applications contain no other entries than those described so far in this section. However, in general

the *fvSolution* dictionary may contain any parameters to control the solvers, algorithms, or in fact anything. For a given solver, the user can look at the source code to find the parameters required. Ultimately, if any parameter or sub-dictionary is missing when an solver is run, it will terminate, printing a detailed error message. The user can then add missing parameters accordingly.

6.4 Monitoring and managing jobs

This section is concerned primarily with successful running of OpenFOAM jobs and extends on the basic execution of solvers described in section 3.1. When a solver is executed, it reports the status of equation solution to standard output, *i.e.* the screen, if the `level` debug switch is set to 1 or 2 (default) in *DebugSwitches* in the *$WM_PROJECT_DIR/etc/-controlDict* file. An example from the beginning of the solution of the cavity tutorial is shown below where it can be seen that, for each equation that is solved, a report line is written with the solver name, the variable that is solved, its initial and final residuals and number of iterations.

```
    Starting time loop

Time = 0.005

Max Courant Number = 0
BICCG:  Solving for Ux, Initial residual = 1, Final residual = 2.96338e-06, No Iterations 8
ICCG:  Solving for p, Initial residual = 1, Final residual = 4.9336e-07, No Iterations 35
time step continuity errors : sum local = 3.29376e-09, global = -6.41065e-20, cumulative = -6.41065e-20
ICCG:  Solving for p, Initial residual = 0.47484, Final residual = 5.41068e-07, No Iterations 34
time step continuity errors : sum local = 6.60947e-09, global = -6.22619e-19, cumulative = -6.86725e-19
ExecutionTime = 0.14 s


Time = 0.01

Max Courant Number = 0.585722
BICCG:  Solving for Ux, Initial residual = 0.148584, Final residual = 7.15711e-06, No Iterations 6
BICCG:  Solving for Uy, Initial residual = 0.256618, Final residual = 8.94127e-06, No Iterations 6
ICCG:  Solving for p, Initial residual = 0.37146, Final residual = 6.67464e-07, No Iterations 33
time step continuity errors : sum local = 6.34431e-09, global = 1.20603e-19, cumulative = -5.66122e-19
ICCG:  Solving for p, Initial residual = 0.271556, Final residual = 3.69316e-07, No Iterations 33
time step continuity errors : sum local = 3.96176e-09, global = 6.9814e-20, cumulative = -4.96308e-19
ExecutionTime = 0.16 s


Time = 0.015

Max Courant Number = 0.758267
BICCG:  Solving for Ux, Initial residual = 0.0448679, Final residual = 2.42301e-06, No Iterations 6
BICCG:  Solving for Uy, Initial residual = 0.0782042, Final residual = 1.47009e-06, No Iterations 7
ICCG:  Solving for p, Initial residual = 0.107474, Final residual = 4.8362e-07, No Iterations 32
time step continuity errors : sum local = 3.99028e-09, global = -5.69762e-19, cumulative = -1.06607e-18
ICCG:  Solving for p, Initial residual = 0.0806771, Final residual = 9.47171e-07, No Iterations 31
time step continuity errors : sum local = 7.92176e-09, global = 1.07533e-19, cumulative = -9.58537e-19
ExecutionTime = 0.19 s
```

6.4.1 The foamJob script for running jobs

The user may be happy to monitor the residuals, iterations, Courant number *etc.* as report data passes across the screen. Alternatively, the user can redirect the report to a log file which will improve the speed of the computation. The foamJob script provides useful options for this purpose with the following executing the specified <`solver`> as a background process and redirecting the output to a file named *log*:

```
foamJob <solver>
```

For further options the user should execute `foamJob -help`. The user may monitor the *log* file whenever they wish, using the UNIXtail command, typically with the `-f` 'follow' option which appends the new data as the *log* file grows:

```
tail -f log
```

6.4.2 The foamLog script for monitoring jobs

There are limitations to monitoring a job by reading the log file, in particular it is difficult to extract trends over a long period of time. The foamLog script is therefore provided to extract data of residuals, iterations, Courant number *etc.* from a log file and present it in a set of files that can be plotted graphically. The script is executed by:

```
foamLog <logFile>
```

The files are stored in a subdirectory of the case directory named *logs*. Each file has the name <*var*>_<*subIter*> where <*var*> is the name of the variable specified in the log file and <*subIter*> is the iteration number within the time step. Those variables that are solved for, the initial residual takes the variable name <*var*> and final residual takes <*var*>*FinalRes*. By default, the files are presented in two-column format of time and the extracted values.

For example, in the cavity tutorial we may wish to observe the initial residual of the `Ux` equation to see whether the solution is converging to a steady-state. In that case, we would plot the data from the *logs/Ux_0* file as shown in Figure 6.1. It can be seen here that the residual falls monotonically until it reaches the convergence tolerance of 10^{-5}.

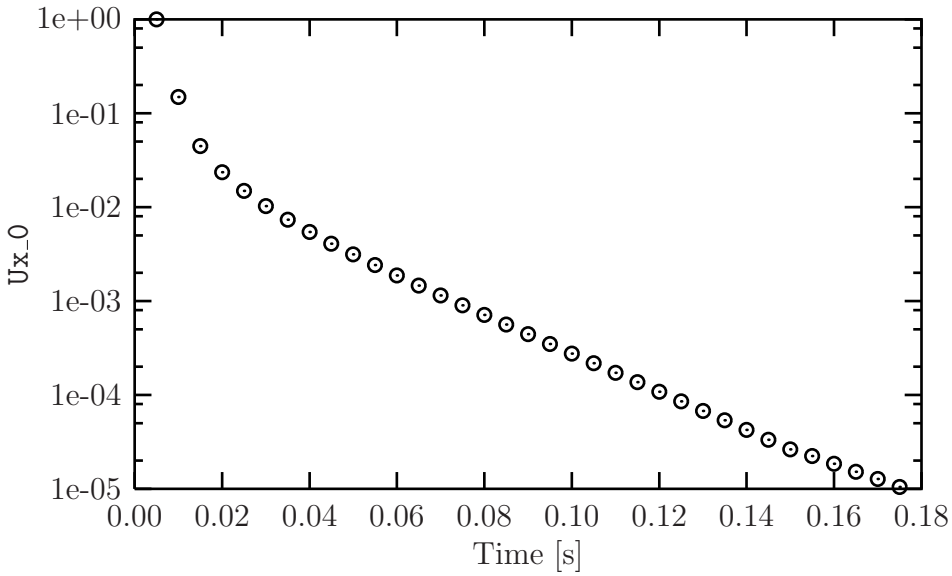


Figure 6.1: Initial residual of `Ux` in the cavity tutorial

foamLog generates files for everything it feasibly can from the *log* file. In the cavity tutorial example, this includes:

- the Courant number, `Courant_0`;

- `Ux` equation initial and final residuals, `Ux_0` and `UxFinalRes_0`, and iterations, `UxIters_0` (and equivalent `Uy` data);
- cumulative, global and local continuity errors after each of the 2 `p` equations, `contCumulative_0`, `contGlobal_0`, `contLocal_0` and `contCumulative_1`, `contGlobal_1`, `contLocal_1`;
- residuals and iterations from the 2 `p` equations `p_0`, `pFinalRes_0`, `pIters_0` and `p_1`, `pFinalRes_1`, `pIters_1`;
- and execution time, `executionTime`.

Chapter 7

Post-processing

This chapter describes options for post-processing with OpenFOAM. OpenFOAM is supplied with a post-processing utility paraFoam that uses ParaView, an open source visualisation application described in section 7.1.

Other methods of post-processing using third party products are offered, including EnSight, Fieldview and the post-processing supplied with Fluent.

7.1 paraFoam

Post-processing OpenFOAM cases with ParaView is supported in several ways:

- visualise the OpenFOAM *blockMeshDict* file within ParaView using the PVblockMeshReader module supplied with OpenFOAM.
- read the OpenFOAM data within ParaView using the native ParaView reader for OpenFOAM.
- read the OpenFOAM data within ParaView using the PVFoamReader module supplied with OpenFOAM.
- convert OpenFOAM data to VTK format with the foamToVTK utility.
- generate VTK format during the simulation with the vtkWrite function object. This functionality largely mirrors that of the foamToVTK utility.
- generate output during the simulation by using VTK via the runTimePostProcessing function object.

The main post-processing tool provided with OpenFOAM is ParaView, an open-source visualization application. The most recent ParaView version at the time of release was 5.4.0, which is also the recommended version. Further details about ParaView can be found at `http://www.paraview.org` and further documentation is available at `http://www.kitware.com/products/books/paraview.html`.

7.1.1 Overview of paraFoam

paraFoam is strictly a script that launches ParaView, by default using the reader module supplied with OpenFOAM. The term paraFoam may thus sometimes be used synonymously for the OpenFOAM reader module itself. Like any OpenFOAM utility, paraFoam can be executed from within the case directory or with the `-case` option with the case path as an argument, *e.g.*:

```
paraFoam -case <caseDir>
```

The paraFoam script can conveniently be used to select the PVFoamReader (this is the default behaviour), to select the PVblockMeshReader (using the -block option), or to select the native ParaView reader (using the `-vtk` option).

7.1.1.1 Recommendations

- The PVblockMeshReader module (paraFoam with `-block`) is currently the only option visualising the *blockMeshDict*.
- The native ParaView reader for OpenFOAM (paraFoam with `-vtk`) is generally the preferred method for general visualisation. It manages decomposed cases and can be used in parallel. However, it understandably does not cope well with complex dictionary input. For such cases, the best recourse is often to skip the *0* initial conditions directory. OpenCFD Ltd.intends to continue participating in the further development of this reader, with several improvements already have been integrated into the ParaView 5.3.0 and 5.4.0 versions.
- The PVFoamReader module (paraFoam default) supplied with OpenFOAM, which predates the native reader, continues to receive active development effort. However, it must be noted the reader *only manages serial or reconstructed cases*, but does support some features not yet found in native reader (*e.g.*, display of patch names, display of fields per cell-zone...). The primary focus of this reader module is now shifted to support things that the native reader does not, and to serve as a platform for testing new ideas and features.

7.1.2 Converting to VTK format

As an alternative, the OpenFOAM data can be also be converted into VTK format using the foamToVTK utility or during the simulation with the the vtkWrite function object that largely mirrors the functionality of the foamToVTK utility. The converted data can be post-processed in ParaView or any other program supporting VTK format.

Both the foamToVTK utility and the vtkWrite function object support legacy and xml VTK formats.

See the tutorials/incompressible/simpleFoam/windAroundBuildings for an example.

7.1.3 Overview of ParaView

After ParaView is launched and opens, the window shown in Figure 7.1 is displayed. The case is controlled from the left panel, which contains the following:

Pipeline Browser lists the *modules* opened in ParaView, where the selected modules are highlighted in blue and the graphics for the given module can be enabled/disabled by clicking the eye button alongside;

Properties **panel** contains the input selections for the case, such as times, regions and fields;

Display **panel** controls the visual representation of the selected module, *e.g.* colours;

Information **panel** gives case statistics such as mesh geometry and size.

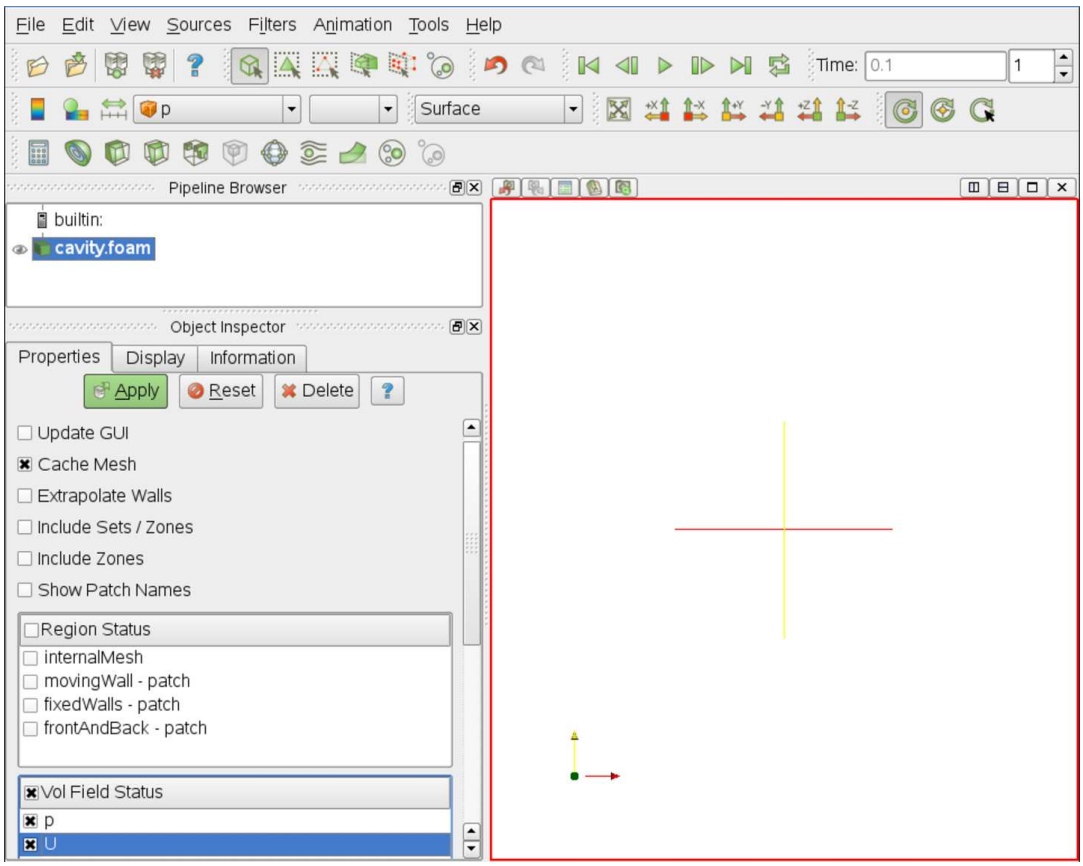


Figure 7.1: The paraFoam window

ParaView operates a tree-based structure in which data can be filtered from the top-level case module to create sets of sub-modules. For example, a contour plot of, say, pressure could be a sub-module of the case module which contains all the pressure data. The strength of ParaView is that the user can create a number of sub-modules and display whichever ones they feel to create the desired image or animation. For example, they may add some solid geometry, mesh and velocity vectors, to a contour plot of pressure, switching any of the items on and off as necessary.

The general operation of the system is based on the user making a selection and then clicking the green Apply button in the Properties panel. The additional buttons are: the Reset button which is used to reset the GUI if necessary; and, the Delete button that will delete the active module.

7.1.4 The Properties panel

The Properties panel for the case module contains the settings for time step, regions and fields. The controls are described in Figure 7.2. It is particularly worth noting that in the current reader module, data in all time directories are loaded into ParaView (in the reader module for ParaView 4.4.0, a set of check boxes controlled the time that were displayed). In the current reader module, the buttons in the `Current Time Controls` and `VCR Controls` toolbars select the time data to be displayed, as shown is section 7.1.6.

As with any operation in paraFoam, the user must click Apply after making any changes to any selections. The Apply button is highlighted in green to alert the user if changes have been made but not accepted. This method of operation has the advantage of allowing the user to make a number of selections before accepting them, which is particularly useful in large cases where data processing is best kept to a minimum.

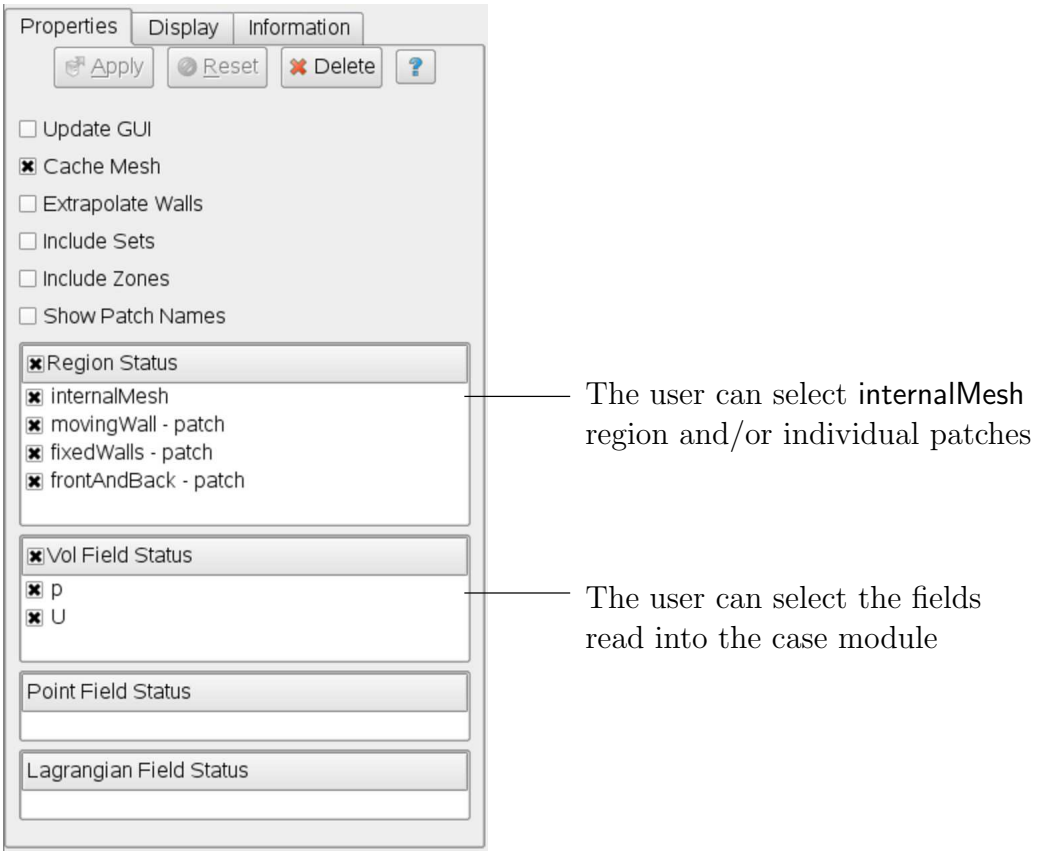


Figure 7.2: The Properties panel for the case module

There are occasions when the case data changes on file and ParaView needs to load the changes, *e.g.* when field data is written into new time directories. To load the changes, the user should check the Update GUI button at the top of the Properties panel and then apply the changes.

7.1.5 The Display panel

The Display panel contains the settings for visualising the data for a given case module. The following points are particularly important:

- the data range may not be automatically updated to the max/min limits of a field, so the user should take care to select Rescale to Data Range at appropriate intervals, in particular after loading the initial case module;

- clicking the Edit Color Map button, brings up a window in which there are two panels:

  1. The Color Scale panel in which the colours within the scale can be chosen. The standard blue to red colour scale for CFD can be selected by clicking Choose Preset and selecting `Blue to Red Rainbox HSV`.

  2. The Color Legend panel has a toggle switch for a colour bar legend and contains settings for the layout of the legend, *e.g.* font.

- the underlying mesh can be represented by selecting `Wireframe` in the `Representation` menu of the Style panel;

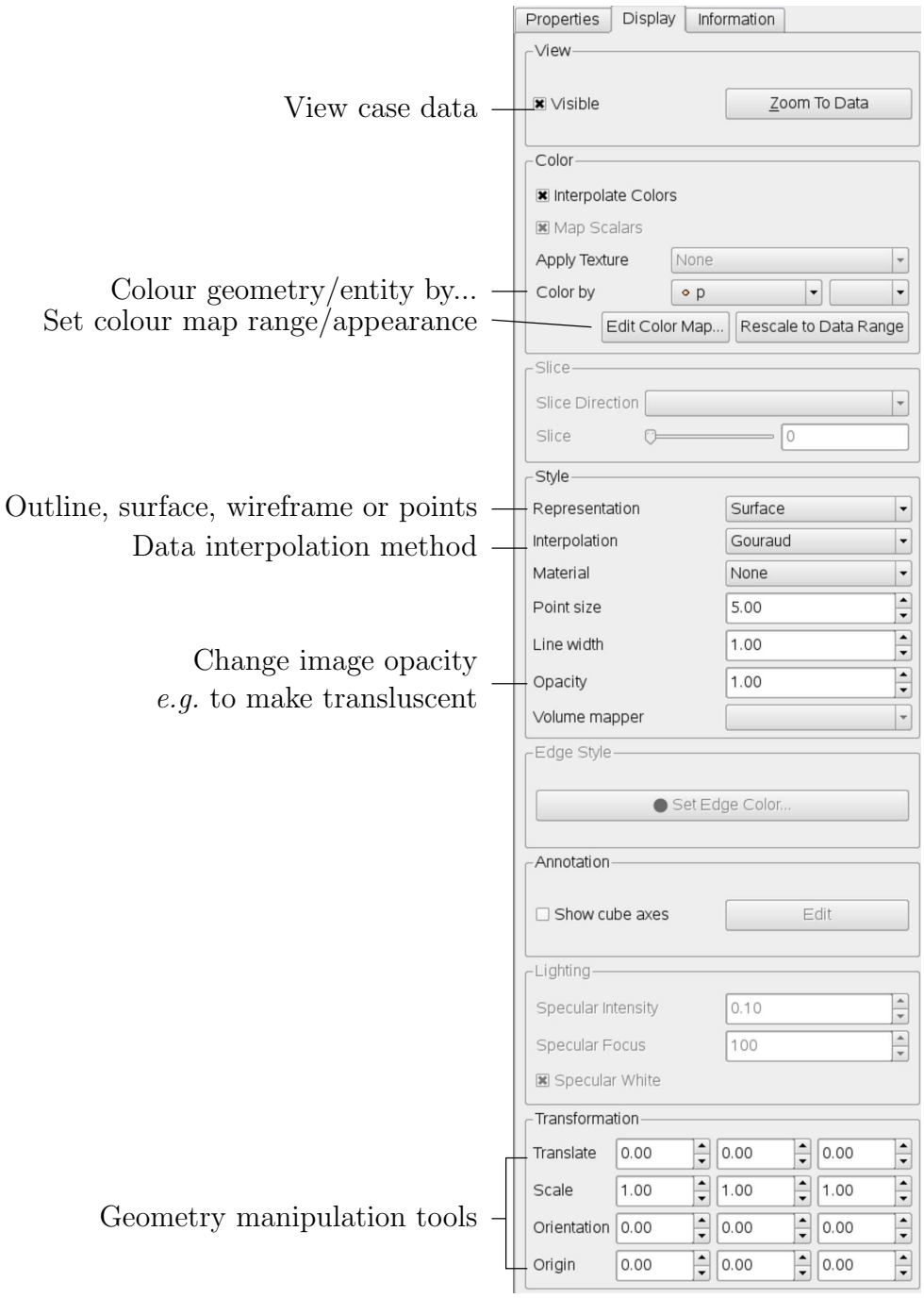


Figure 7.3: The Display panel

- the geometry, *e.g.* a mesh (if `Wireframe` is selected), can be visualised as a single colour by selecting `Solid Color` from the `Color By` menu and specifying the colour in the Set Ambient Color window;
- the image can be made translucent by editing the value in the Opacity text box (1 = solid, 0 = invisible) in the Style panel.

7.1.6 The button toolbars

ParaView duplicates functionality from pull-down menus at the top of the main window and the major panels, within the toolbars below the main pull-down menus. The displayed toolbars can be selected from `Toolbars` in the main `View` menu. The default layout with all toolbars is shown in Figure 7.4 with each toolbar labelled. The function of many of

the buttons is clear from their icon and, with tooltips enabled in the `Help` menu, the user is given a concise description of the function of any button.

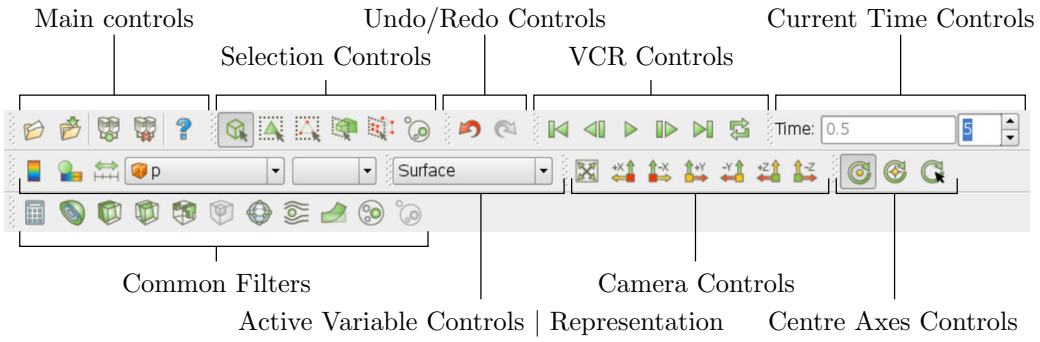


Figure 7.4: Toolbars in ParaView

7.1.7 Manipulating the view

This section describes operations for setting and manipulating the view of objects in paraFoam.

7.1.7.1 View settings

The `View Settings` are selected from the `Edit` menu, which opens a View Settings (Render View) window with a table of 3 items: General, Lights and Annotation. The General panel includes the following items which are **often worth setting at startup**:

- the background colour, where white is often a preferred choice for printed material, is set by choosing background from the down-arrow button next to Choose Color button, then selecting the color by clicking on the Choose Color button;
- Use parallel projection which is the usual choice for CFD, especially for 2D cases.

The Lights panel contains detailed lighting controls within the Light Kit panel. A separate Headlight panel controls the direct lighting of the image. Checking the Headlight button with white light colour of strength 1 seems to help produce images with strong bright colours, e.g. with an isosurface.

The Annotation panel includes options for including annotations in the image. The Orientation Axes feature controls an axes icon in the image window, e.g. to set the colour of the axes labels x, y and z.

7.1.7.2 General settings

The general `Settings` are selected from the `Edit` menu, which opens a general Options window with General, Colors, Animations, Charts and Render View menu items.

The General panel controls some default behaviour of ParaView. In particular, there is an Auto Accept button that enables ParaView to accept changes automatically without clicking the green Apply button in the Properties window. For larger cases, this option is generally not recommended: the user does not generally want the image to be re-rendered between each of a number of changes he/she selects, but be able to apply a number of changes to be re-rendered in their entirety once.

The Render View panel contains 3 sub-items: General, Camera and Server. The General panel includes the level of detail (LOD) which controls the rendering of the image while it

is being manipulated, *e.g.* translated, resized, rotated; lowering the levels set by the sliders, allows cases with large numbers of cells to be re-rendered quickly during manipulation.

The Camera panel includes control settings for 3D and 2D movements. This presents the user with a map of rotation, translate and zoom controls using the mouse in combination with Shift- and Control-keys. The map can be edited to suit by the user.

7.1.8 Contour plots

A contour plot is created by selecting `Contour` from the `Filter` menu at the top menu bar. The filter acts on a given module so that, if the module is the 3D case module itself, the contours will be a set of 2D surfaces that represent a constant value, *i.e.* isosurfaces. The Properties panel for contours contains an Isosurfaces list that the user can edit, most conveniently by the New Range window. The chosen scalar field is selected from a pull down menu.

7.1.8.1 Introducing a cutting plane

Very often a user will wish to create a contour plot across a plane rather than producing isosurfaces. To do so, the user must first use the `Slice` filter to create the cutting plane, on which the contours can be plotted. The `Slice` filter allows the user to specify a cutting `Plane`, `Box` or `Sphere` in the `Slice Type` menu by a center and normal/radius respectively. The user can manipulate the cutting plane like any other using the mouse.

The user can then run the `Contour` filter on the cut plane to generate contour lines.

7.1.9 Vector plots

Vector plots are created using the `Glyph` filter. The filter reads the field selected in `Vectors` and offers a range of `Glyph Types` for which the `Arrow` provides a clear vector plot images. Each glyph has a selection of graphical controls in a panel which the user can manipulate to best effect.

The remainder of the Properties panel contains mainly the `Scale Mode` menu for the glyphs. The most common options are `Scale Mode` are: `Vector`, where the glyph length is proportional to the vector magnitude; and, `Off` where each glyph is the same length. The Set Scale Factor parameter controls the base length of the glyphs.

7.1.9.1 Plotting at cell centres

Vectors are by default plotted on cell vertices but, very often, we wish to plot data at cell centres. This is done by first applying the `Cell Centers` filter to the case module, and then applying the `Glyph` filter to the resulting cell centre data.

7.1.10 Streamlines

Streamlines are created by first creating tracer lines using the `Stream Tracer` filter. The tracer Seed panel specifies a distribution of tracer points over a `Line Source` or `Point Cloud`. The user can view the tracer source, *e.g.* the line, but it is displayed in white, so they may need to change the background colour in order to see it.

The distance the tracer travels and the length of steps the tracer takes are specified in the text boxes in the main Stream Tracer panel. The process of achieving desired tracer lines is largely one of trial and error in which the tracer lines obviously appear smoother as the step length is reduced but with the penalty of a longer calculation time.

Once the tracer lines have been created, the `Tubes` filter can be applied to the *Tracer* module to produce high quality images. The tubes follow each tracer line and are not strictly cylindrical but have a fixed number of sides and given radius. When the number of sides is set above, say, 10, the tubes do however appear cylindrical, but again this adds a computational cost.

7.1.11 Image output

The simplest way to output an image to file from ParaView is to select `Save Screenshot` from the `File` menu. On selection, a window appears in which the user can select the resolution for the image to save. There is a button that, when clicked, locks the aspect ratio, so if the user changes the resolution in one direction, the resolution is adjusted in the other direction automatically. After selecting the pixel resolution, the image can be saved. To achieve high quality output, the user might try setting the pixel resolution to 1000 or more in the x-direction so that when the image is scaled to a typical size of a figure in an A4 or US letter document, perhaps in a PDF document, the resolution is sharp.

7.1.12 Animation output

To create an animation, the user should first select `Save Animation` from the `File` menu. A dialogue window appears in which the user can specify a number of things including the image resolution. The user should specify the resolution as required. The other noteworthy setting is number of frames per timestep. While this would intuitively be set to 1, it can be set to a larger number in order to introduce more frames into the animation artificially. This technique can be particularly useful to produce a slower animation because some movie players have limited speed control, particularly over `mpeg` movies.

On clicking the Save Animation button, another window appears in which the user specifies a file name *root* and file format for a set of images. On clicking OK, the set of files will be saved according to the naming convention "<fileRoot>_<imageNo>.<fileExt>", *e.g.* the third image of a series with the file root "`animation`", saved in `jpg` format would be named "`animation_0002.jpg`" (<imageNo> starts at 0000).

Once the set of images are saved the user can convert them into a movie using their software of choice. The convert utility in the ImageMagick package can do this from the command line, *e.g.* by

```
convert animation*jpg movie.mpg
```

When creating an `mpg` movie it can be worth increasing the default quality setting, *e.g.* with `-quality 90%`, to reduce the graininess that can occur with the default setting.

7.2 Post-processing with Fluent

It is possible to use Fluent as a post-processor for the cases run in OpenFOAM. Two converters are supplied for the purpose: foamMeshToFluent which converts the OpenFOAM mesh into Fluent format and writes it out as a *.msh* file; and, foamDataToFluent converts the OpenFOAM results data into a *.dat* file readable by Fluent. foamMeshToFluent is executed in the usual manner. The resulting mesh is written out in a *fluentInterface* subdirectory of the case directory, *i.e.*<*caseName*>*/fluentInterface/*<*caseName*>*.msh*

foamDataToFluent converts the OpenFOAM data results into the Fluent format. The conversion is controlled by two files. First, the *controlDict* dictionary specifies `startTime`, giving the set of results to be converted. If you want to convert the latest result, `startFrom` can be set to `latestTime`. The second file which specifies the translation is the *foamDataToFluentDict* dictionary, located in the *constant* directory. An example *foamDataToFluentDict* dictionary is given below:

```
/*--------------------------------*- C++ -*----------------------------------*\
| =========                 |                                                 |
| \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox           |
|  \\    /   O peration     | Version:  v2112                                 |
|   \\  /    A nd           | Website:  www.openfoam.com                      |
|    \\/     M anipulation  |                                                 |
\*---------------------------------------------------------------------------*/
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      foamDataToFluentDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

p               1;

U               2;

T               3;

h               4;

k               5;

epsilon         6;

alpha1          150;


// ************************************************************************* //
```

The dictionary contains entries of the form

```
<fieldName> <fluentUnitNumber>
```

The <fluentUnitNumber> is a label used by the Fluent post-processor that only recognises a fixed set of fields. The basic set of <fluentUnitNumber> numbers are quoted in Table 7.1. The dictionary must contain all the entries the user requires to post-process,

Fluent name	Unit number	Common OpenFOAM name
PRESSURE	1	p
MOMENTUM	2	U
TEMPERATURE	3	T
ENTHALPY	4	h
TKE	5	k
TED	6	epsilon
SPECIES	7	—
G	8	—
XF_RF_DATA_VOF	150	gamma
TOTAL_PRESSURE	192	—
TOTAL_TEMPERATURE	193	—

Table 7.1: Fluent unit numbers for post-processing.

e.g. in our example we have entries for pressure p and velocity U. The list of default entries described in Table 7.1. The user can run foamDataToFluent like any utility.

To view the results using Fluent, go to the *fluentInterface* subdirectory of the case directory and start a 3 dimensional version of Fluent with

```
fluent 3d
```

The mesh and data files can be loaded in and the results visualised. The mesh is read by selecting `Read Case` from the `File` menu. Support items should be selected to read certain data types, *e.g.* to read turbulence data for `k` and `epsilon`, the user would select `k-epsilon` from the `Define->Models->Viscous` menu. The data can then be read by selecting `Read Data` from the `File` menu.

A note of caution: users MUST NOT try to use an original Fluent mesh file that has been converted to OpenFOAM format in conjunction with the OpenFOAM solution that has been converted to Fluent format since the alignment of zone numbering cannot be guaranteed.

7.3 Post-processing with EnSight

Post-processing OpenFOAM cases with EnSight is supported in several ways:

- convert OpenFOAM data to EnSight format with the foamToEnsight utility (serial or parallel);
- generate data in EnSight format during the simulation with the ensightWrite function object (serial or parallel). This functionality largely mirrors that of the foamToEnsight utility;
- convert the OpenFOAM data to EnSight format with the foamToEnsightParts utility. This is serial only, but supports separate parts for each cell-zone;
- read the OpenFOAM data within EnSight using the native EnSight capabilities (available in EnSight 9.2.2 and 10 - https://www.ensight.com/openfoam/);
- read the OpenFOAM data within EnSight using the ensightFoamReader module supplied with OpenFOAM.

7.3.1 Converting data to EnSight format

The foamToEnsight and foamToEnsightParts convert data from OpenFOAM to EnSight file format and are executed as normal OpenFOAM applications. The foamToEnsight normally creates a directory named *EnSight* in the case directory and *deletes any existing EnSight directory*. The foamToEnsightParts normally creates a directory named *Ensight* in the case directory, but does not delete an existing directory. In both cases, the output directory can be adjusted using the `-name` option. The principal difference between the two utilities is that while foamToEnsight runs in serial and parallel, it only writes the internal mesh as a single EnSight part. By contrast, The foamToEnsightParts utility writes each cell-zone as a separate EnSight part, which can make post-processing in EnSight easier and faster, but *only runs in serial*. The operation of both utilities and their are similar. They read the OpenFOAM data for the specified times and write corresponding EnSight data files and a case file with the details of the data names and directory layout. The *data* subdirectory contains all data and transient geometry. For non-moving cases, the `geometry` file will be located within the top-level directory.

The *data* directory contains a series of numbered sub-directories that contain the converted data at different times. For documentation and scripting purposes, these numbered sub-directories each contain plain text *time* file with index and time value. The converted EnSight data are stored with their original OpenFOAM names, *e.g.*`T` for temperature, etc.

7.3.1.1 Loading converted data in EnSight

Once converted, the data can be read into EnSight:

1. from the EnSight GUI, the user should select `Data (Reader)` from the `File` menu;
2. the appropriate *EnSight_Case* file should be highlighted in the Files box;
3. the `Format` selector should be set to `Case`, the EnSight default setting;
4. the user should click (Set) Case and Okay.

7.3.2 Converting data during the simulation

If desired, most of the foamToEnsight functionality (currently no output for Lagrangian fields) can be harnessed during the simulation by using the ensightWrite function object (serial or parallel). See the tutorials/incompressible/simpleFoam/motorBike for an example.

7.3.3 The ensightFoamReader reader module

Since there was historically no native means of loading OpenFOAM data within EnSight, the ensightFoamReader reader module has been provided. This user-defined reader provides the capability to employ a user-defined module to read data from a format other than the standard EnSight format. OpenFOAM includes its own reader module ensightFoamReader that is compiled into a library named libuserd-foam. This library must be available to EnSight, i.e. it must be able to locate it on the filing system.

7.3.3.1 Configuration of EnSight for the reader module

It is necessary to set some environment variables to use the EnSight reader module. The settings are made in the *ensight* file in the *$WM_PROJECT_DIR/etc/config.sh* or *$WM_PROJECT_DIR/etc/config.csh* directories The environment variables associated with EnSight are prefixed by $CEI_ or $ENSIGHT9_ and listed in Table 7.2. With a standard user setup, only $CEI_HOME may need to be set manually, to the path of the EnSight installation.

7.3.3.2 Using the reader module

The main difficulty in using the EnSight reader lies in the fact that EnSight expects that a case is defined by the contents of a particular file, rather than a directory as used by OpenFOAM. Therefore, in the following instructions for the using the reader below, the user should pay particular attention to the details of case selection, since EnSight does not permit selection by directory name.

1. from the EnSight GUI, the user should select `Data (Reader)` from the `File` menu;
2. The user should now be able to select the `OpenFOAM` from the `Format` menu; if not, there is a problem with the configuration described above.

Environment variable	Description and options
$CEI_HOME	Path where EnSight is installed, eg */usr/local/ensight*, added to the system path by default
$CEI_ARCH	Machine architecture, from a choice of names corresponding to the machine directory names in *$CEI_HOME/ensight74/machines*; default settings include `linux_2.4` and `sgi_6.5_n32`
$ENSIGHT7_READER	Path that EnSight searches for the user defined libuserd-foam reader library, set by default to `$FOAM_LIBBIN`
$ENSIGHT7_INPUT	Set by default to `dummy`

Table 7.2: Environment variable settings for EnSight.

3. The user should find their case directory from the File Selection window, highlight one of top 2 entries in the Directories box ending in `/.` or `/..` and click (Set) Geometry.

4. The path field should now contain an entry for the case. The (Set) Geometry text box should contain a '/'.

5. The user may now click Okay and EnSight will begin reading the data.

6. When the data is read, a new Data Part Loader window will appear, asking which part(s) are to be read. The user should select Load all.

7. When the mesh is displayed in the EnSight window the user should close the Data Part Loader window, since some features of EnSight will not work with this window open.

7.4 Sampling data

OpenFOAM provides a set of sampling function objects to sample field data, either through a 1D line for plotting on graphs or a 2D plane and 3D surfaces for displaying as images. Each sampling tool is specified in a dictionary either in the main *functions* dictionary of the *controlDict* file, or separate files in the case *system* directory. The data can be written in a range of formats including well-known graphing packages such as Grace/xmgr, gnuplot and jPlot.

The plateHole tutorial case in the *$FOAM_TUTORIALS/stressAnalysis/solidDisplacementFoam* directory also contains an example for 1D line sampling:

```
17      setConfig
18      {
19          axis    y;
20      }
21
22      // Must be last entry
23      #includeEtc "caseDicts/postProcessing/graphs/graph.cfg"
24  }
25
26
27  // ************************************************************************* //
```

The dictionary contains the following entries:

`interpolationScheme` the scheme of data interpolation;

Keyword	Options	Description
`interpolation-`	`cell`	Cell-centre value assumed constant over cell
`Scheme`	`cellPoint`	Linear weighted interpolation using cell values
	`cellPointFace`	Mixed linear weighted / cell-face interpolation
	`pointMVC`	Point values only (Mean Value Coordinates)
	`cellPatchConstrained`	As `cell` but uses face value on boundary faces
`setFormat`	`raw`	Raw ASCII data in columns
	`gnuplot`	Data in gnuplot format
	`xmgr`	Data in Grace/xmgr format
	`jplot`	Data in jPlot format
	`vtk`	Data in VTK format
	`ensight`	Data in EnSight format
	`csv`	Data in CSV format
`surfaceFormat`	`null`	Suppresses output
	`foamFile`	*points*, *faces*, *values* file
	`dx`	DX scalar or vector format
	`vtk`	VTK ASCII format
	`raw`	xyz values for use with *e.g.*gnuplotsplot
	`stl`	ASCII STL; just surface, no values
	`ensight`	EnSight surface format
	`boundaryData`	A form that can be used with timeVaryingMapped bound
	`starcd`	Nastran surface format
	`nastran`	
`fields`	List of fields to be sampled, *e.g.* for velocity `U`:	
	`U`	Writes all components of **U**
`sets`	List of 1D sets subdictionaries — see Table 7.4	
`surfaces`	List of 2D surfaces subdictionaries — see Table 7.5 and Table 7.6	

Table 7.3: keyword entries for *sampleDict*.

`sets` the locations within the domain that the fields are line-sampled (1D).

`surfaces` the locations within the domain that the fields are surface-sampled (2D).

`setFormat` the format of line data output;

`surfaceFormat` the format of surface data output;

`fields` the fields to be sampled;

The `interpolationScheme` includes `cellPoint` and `cellPointFace` options in which each polyhedral cell is decomposed into tetrahedra and the sample values are interpolated from values at the tetrahedra vertices. With `cellPoint`, the tetrahedra vertices include the polyhedron cell centre and 3 face vertices. The vertex coincident with the cell centre inherits the cell centre field value and the other vertices take values interpolated from cell centres. With `cellPointFace`, one of the tetrahedra vertices is also coincident with a face centre, which inherits field values by conventional interpolation schemes using values at the centres of cells that the face intersects.

The `setFormat` entry for line sampling includes a raw data format and formats for gnuplot, Grace/xmgr and jPlot graph drawing packages. The data are written into a *sets*

directory within the case directory. The directory is split into a set of time directories and the data files are contained therein. Each data file is given a name containing the field name, the sample set name, and an extension relating to the output format, including *.xy* for raw data, *.agr* for Grace/xmgr and *.dat* for jPlot. The gnuplot format has the data in raw form with an additional commands file, with *.gplt* extension, for generating the graph. *Note that any existing sets directory is deleted when* sample *is run.*

The `surfaceFormat` entry for surface sampling includes a raw data format and formats for gnuplot, Grace/xmgr and jPlot graph drawing packages. The data are written into a *surfaces* directory within the case directory. The directory is split into time directories and files are written much as with line sampling.

The `fields` list contains the fields that the user wishes to sample. The sample utility can parse the following restricted set of functions to enable the user to manipulate vector and tensor fields, *e.g.* for `U`:

`U.component(`n`)` writes the nth component of the vector/tensor, $n = 0, 1 \ldots$;

`mag(U)` writes the magnitude of the vector/tensor.

The `sets` list contains sub-dictionaries of locations where the data is to be sampled. The sub-dictionary is named according to the name of the set and contains a set of entries, also listed in Table 7.4, that describes the locations where the data is to be sampled. For example, a `uniform` sampling provides a uniform distribution of `nPoints` sample locations along a line specified by a `start` and `end` point. All sample sets are also given: a `type`; and, means of specifying the length ordinate on a graph by the `axis` keyword.

The `surfaces` list contains sub-dictionaries of locations where the data is to be sampled. The sub-dictionary is named according to the name of the surface and contains a set of entries beginning with the `type`: either a `plane`, defined by point and normal direction, with additional sub-dictionary entries specified in Table 7.5; or, a `patch`, coinciding with an existing boundary patch, with additional sub-dictionary entries specified in Table 7.6.

Sampling type	Sample locations	Required entries: name	axis	start	end	nPoints	points
`uniform`	Uniformly distributed points on a line	•	•	•	•	•	
`face`	Intersection of specified line and cell faces	•	•	•	•		
`midPoint`	Midpoint between line-face intersections	•	•	•	•		
`midPointAndFace`	Combination of `midPoint` and `face`	•	•	•	•		
`cloud`	Specified points	•	•				•
`patchCloud`	Sample nearest points on selected patches	•	•				•
`patchSeed`	Randomly sample on selected patches	•	•				•
`polyLine`	Specified points (uses particle tracking)	•	•				•
`triSurfaceMeshPointSet`	Sample points on a triangulated surface	•	•				•

Entries	Description	Options	
`type`	Sampling type	see list above	
`axis`	Output of sample location	`x`	x ordinate
		`y`	y ordinate
		`z`	z ordinate
		`xyz`	xyz coordinates
		`distance`	distance from point 0
`start`	Start point of sample line	*e.g.*`(0.0 0.0 0.0)`	
`end`	End point of sample line	*e.g.*`(0.0 2.0 0.0)`	
`nPoints`	Number of sampling points	*e.g.*`200`	
`points`	List of sampling points		

Table 7.4: Entries within `sets` sub-dictionaries.

Keyword	Description	Options
`basePoint`	Point on plane	*e.g.*`(0 0 0)`
`normalVector`	Normal vector to plane	*e.g.*`(1 0 0)`
`interpolate`	Interpolate data?	`true`/`false`
`triangulate`	Triangulate surface? (optional)	`true`/`false`

Table 7.5: Entries for a `plane` in `surfaces` sub-dictionaries.

Keyword	Description	Options
`patchName`	Name of patch	*e.g.*`movingWall`
`interpolate`	Interpolate data?	`true`/`false`
`triangulate`	Triangulate surface? (optional)	`true`/`false`

Table 7.6: Entries for a `patch` in `surfaces` sub-dictionaries.

Appendix A

Reference

A.1 Standard solvers

OpenFOAM does not have a generic solver applicable to all cases. Instead, users must choose a specific solver for a class of problems to solve. The solvers with the OpenFOAM distribution are in the *$FOAM_SOLVERS* directory, reached quickly by typing app at the command line. This directory is further subdivided into several directories by category of continuum mechanics, *e.g.* incompressible flow, heat transfer, multiphase, lagrangian, combustion. Each solver is given a name that is descriptive. For some, mainly incompressible solvers, it reflects the algorithm, *e.g.*simpleFoam using the SIMPLE algorithm, pimpleFoam using the PIMPLE algorithm. More often the name reflects the physical models or type of problem it is designed to solve, *e.g.*shallowWaterFoam, sonicFoam, cavitatingFoam. The current list of solvers distributed with OpenFOAM is given in Table A.1.

'Basic' CFD codes	
laplacianFoam	Laplace equation solver for a scalar quantity
overLaplacianDyMFoam	Laplace equation solver for a scalar quantity
potentialFoam	Potential flow solver which solves for the velocity potential, to calculate the flux-field, from which the velocity field is obtained by reconstructing the flux
overPotentialFoam	Potential flow solver which solves for the velocity potential, to calculate the flux-field, from which the velocity field is obtained by reconstructing the flux
scalarTransportFoam	Passive scalar transport equation solver
Incompressible flow	
adjointOptimisationFoam	An automated adjoint-based optimisation loop. Supports multiple types of optimisation (shape, topology etc)

Continued on next page

Continued from previous page

adjointShape-OptimizationFoam	Steady-state solver for incompressible, turbulent flow of non-Newtonian fluids with optimisation of duct shape by applying "blockage" in regions causing pressure loss as estimated using an adjoint formulation
boundaryFoam	Steady-state solver for incompressible, 1D turbulent flow, typically to generate boundary layer conditions at an inlet
icoFoam	Transient solver for incompressible, laminar flow of Newtonian fluids
nonNewtonianIcoFoam	Transient solver for incompressible, laminar flow of non-Newtonian fluids
pimpleFoam	Transient solver for incompressible, turbulent flow of Newtonian fluids on a moving mesh
overPimpleDyMFoam	Transient solver for incompressible flow of Newtonian fluids on a moving mesh using the PIMPLE (merged PISO-SIMPLE) algorithm
SRFPimpleFoam	Large time-step transient solver for incompressible flow in a single rotating frame
pisoFoam	Transient solver for incompressible, turbulent flow, using the PISO algorithm
shallowWaterFoam	Transient solver for inviscid shallow-water equations with rotation
simpleFoam	Steady-state solver for incompressible, turbulent flows
overSimpleFoam	Steady-state solver for incompressible flows with turbulence modelling
porousSimpleFoam	Steady-state solver for incompressible, turbulent flow with implicit or explicit porosity treatment and support for multiple reference frames (MRF)
SRFSimpleFoam	Steady-state solver for incompressible, turbulent flow of non-Newtonian fluids in a single rotating frame

Compressible flow

rhoCentralFoam	Density-based compressible flow solver based on central-upwind schemes of Kurganov and Tadmor with support for mesh-motion and topology changes

Continued on next page

Continued from previous page

rhoPimpleAdiabatic-Foam	Transient solver for laminar or turbulent flow of weakly compressible fluids for low Mach number aeroacoustic applications
rhoPimpleFoam	Transient solver for turbulent flow of compressible fluids for HVAC and similar applications, with optional mesh motion and mesh topology changes
overRhoPimpleDy-MFoam	Transient solver for laminar or turbulent flow of compressible fluids for HVAC and similar applications
rhoSimpleFoam	Steady-state solver for compressible turbulent flow
overRhoSimpleFoam	Overset steady-state solver for compressible turbulent flow
rhoPorousSimpleFoam	Steady-state solver for compressible turbulent flow, with implicit or explicit porosity treatment and optional sources
sonicFoam	Transient solver for trans-sonic/supersonic, turbulent flow of a compressible gas
sonicDyMFoam	Transient solver for trans-sonic/supersonic, turbulent flow of a compressible gas, with optional mesh motion and mesh topology changes
sonicLiquidFoam	Transient solver for trans-sonic/supersonic, laminar flow of a compressible liquid

Multiphase flow

cavitatingFoam	Transient cavitation solver based on the homogeneous equilibrium model from which the compressibility of the liquid/vapour 'mixture' is obtained
cavitatingDyMFoam	Transient cavitation solver based on the homogeneous equilibrium model from which the compressibility of the liquid/vapour 'mixture' is obtained, with optional mesh motion and mesh topology changes
compressibleInterFoam	Solver for two compressible, non-isothermal immiscible fluids using a VOF (volume of fluid) phase-fraction based interface capturing approach
compressibleInterDy-MFoam	Solver for two compressible, non-isothermal immiscible fluids using a VOF (volume of fluid) phase-fraction based interface capturing approach, with optional mesh motion and mesh topology changes including adaptive re-meshing

Continued on next page

Continued from previous page

compressibleInterFilmFoam	Solver for two compressible, non-isothermal immiscible fluids using a VOF (volume of fluid) phase-fraction based interface capturing approach
compressibleInterIsoFoam	Solver derived from interFoam for two compressible, immiscible fluids using the isoAdvector phase-fraction based interface capturing approach, with optional mesh motion and mesh topology changes including adaptive re-meshing
overCompressibleInterDyMFoam	Solver for two compressible, non-isothermal, immiscible fluids using VOF (i.e. volume of fluid) phase-fraction based interface capturing approach
compressibleMultiphaseInterFoam	Solver for N compressible, non-isothermal immiscible fluids using a VOF (volume of fluid) phase-fraction based interface capturing approach
driftFluxFoam	Solver for two incompressible fluids using the mixture approach with the drift-flux approximation for relative motion of the phases
icoReactingMultiphaseInterFoam	Solver for N incompressible, non-isothermal immiscible fluids with phase-change. Uses a VOF (volume of fluid) phase-fraction based interface capturing approach
interCondensatingEvaporatingFoam	Solver for two incompressible, non-isothermal immiscible fluids with phase-change (evaporation-condensation) between a fluid and its vapour. Uses a VOF (volume of fluid) phase-fraction based interface capturing approach
interFoam	Solver for two incompressible, isothermal immiscible fluids using a VOF (volume of fluid) phase-fraction based interface capturing approach, with optional mesh motion and mesh topology changes including adaptive re-meshing
interMixingFoam	Solver for 3 incompressible fluids, two of which are miscible, using a VOF method to capture the interface, with optional mesh motion and mesh topology changes including adaptive re-meshing
overInterDyMFoam	Solver for two incompressible, isothermal immiscible fluids using a VOF (volume of fluid) phase-fraction based interface capturing approach, with optional mesh motion and mesh topology changes including adaptive re-meshing

Continued on next page

Continued from previous page

interIsoFoam	Solver derived from interFoam for two incompressible, isothermal immiscible fluids using the isoAdvector phase-fraction based interface capturing approach, with optional mesh motion and mesh topology changes including adaptive re-meshing
interPhaseChangeFoam	Solver for two incompressible, isothermal immiscible fluids with phase-change (e.g. cavitation). Uses VOF (volume of fluid) phase-fraction based interface capturing
interPhaseChangeDyMFoam	Solver for two incompressible, isothermal immiscible fluids with phase-change (e.g. cavitation). Uses VOF (volume of fluid) phase-fraction based interface capturing, with optional mesh motion and mesh topology changes including adaptive re-meshing
overInterPhaseChangeDyMFoam	Solver for two incompressible, isothermal, immiscible fluids with phase-change (e.g. cavitation) using VOF (i.e. volume of fluid) phase-fraction based interface capturing, with optional dynamic mesh motion (including overset) and mesh topology changes including adaptive re-meshing
MPPICInterFoam	Solver for two incompressible, isothermal immiscible fluids using a VOF (volume of fluid) phase-fraction based interface capturing approach. The momentum and other fluid properties are of the "mixture" and a single momentum equation is solved
multiphaseEulerFoam	Solver for a system of many compressible fluid phases including heat-transfer
multiphaseInterFoam	Solver for N incompressible fluids which captures the interfaces and includes surface-tension and contact-angle effects for each phase, with optional mesh motion and mesh topology changes
potentialFreeSurfaceFoam	Incompressible Navier-Stokes solver with inclusion of a wave height field to enable single-phase free-surface approximations
potentialFreeSurfaceDyMFoam	Incompressible Navier-Stokes solver with inclusion of a wave height field to enable single-phase free-surface approximations, with optional mesh motion and mesh topology changes

Continued on next page

Continued from previous page

reactingMultiphase-EulerFoam	Solver for a system of any number of compressible fluid phases with a common pressure, but otherwise separate properties. The type of phase model is run time selectable and can optionally represent multiple species and in-phase reactions. The phase system is also run time selectable and can optionally represent different types of momentum, heat and mass transfer
reactingTwoPhase-EulerFoam	Solver for a system of two compressible fluid phases with a common pressure, but otherwise separate properties. The type of phase model is run time selectable and can optionally represent multiple species and in-phase reactions. The phase system is also run time selectable and can optionally represent different types of momentum, heat and mass transfer
twoLiquidMixingFoam	Solver for mixing two incompressible fluids
twoPhaseEulerFoam	Solver for a system of two compressible fluid phases with one dispersed phase. Eg, gas bubbles in a liquid including heat-transfer

Direct numerical simulation (DNS)

dnsFoam	Direct numerical simulation solver for boxes of isotropic turbulence

Combustion

chemFoam	Solver for chemistry problems, designed for use on single cell cases to provide comparison against other chemistry solvers, that uses a single cell mesh, and fields created from the initial conditions
coldEngineFoam	Solver for cold-flow in internal combustion engines
fireFoam	Transient solver for fires and turbulent diffusion flames with reacting particle clouds, surface film and pyrolysis modelling
PDRFoam	Solver for compressible premixed/partially-premixed combustion with turbulence modelling
reactingFoam	Solver for combustion with chemical reactions
rhoReactingBuoyant-Foam	Solver for combustion with chemical reactions using a density-based thermodynamics package with enhanced buoyancy treatment

Continued on next page

Continued from previous page

rhoReactingFoam	Solver for combustion with chemical reactions using density-based thermodynamics package
XiFoam	Solver for compressible premixed/partially-premixed combustion with turbulence modelling
XiDyMFoam	Solver for compressible premixed/partially-premixed combustion with turbulence modelling
XiEngineFoam	Solver for internal combustion engines

Heat transfer and buoyancy-driven flows

buoyantBoussinesqPimpleFoam	Transient solver for buoyant, turbulent flow of incompressible fluids, with optional mesh motion and mesh topology changes
buoyantBoussinesqSimpleFoam	Steady-state solver for buoyant, turbulent flow of incompressible fluids
buoyantPimpleFoam	Transient solver for buoyant, turbulent flow of compressible fluids for ventilation and heat-transfer, with optional mesh motion and mesh topology changes
overBuoyantPimpleDyMFoam	Transient solver for buoyant, turbulent flow of compressible fluids for ventilation and heat-transfer with overset feature
buoyantSimpleFoam	Steady-state solver for buoyant, turbulent flow of compressible fluids, including radiation, for ventilation and heat-transfer
chtMultiRegionFoam	Transient solver for buoyant, turbulent fluid flow and solid heat conduction with conjugate heat transfer between solid and fluid regions
chtMultiRegionSimpleFoam	Steady-state solver for buoyant, turbulent fluid flow and solid heat conduction with conjugate heat transfer between solid and fluid regions
chtMultiRegionTwoPhaseEulerFoam	Transient solver for buoyant, turbulent fluid flow and solid heat conduction with conjugate heat transfer between solid and fluid regions
solidFoam	Solver for energy transport and thermodynamics on a solid
thermoFoam	Solver for energy transport and thermodynamics on a frozen flow field

Particle-tracking flows

Continued on next page

Continued from previous page

coalChemistryFoam	Transient solver for compressible, turbulent flow, with coal and limestone particle clouds, an energy source, and combustion
DPMFoam	Transient solver for the coupled transport of a single kinematic particle cloud including the effect of the volume fraction of particles on the continuous phase
DPMDyMFoam	Transient solver for the coupled transport of a single kinematic particle cloud including the effect of the volume fraction of particles on the continuous phase, with optional mesh motion and mesh topology changes
MPPICDyMFoam	Transient solver for the coupled transport of a single kinematic particle cloud including the effect of the volume fraction of particles on the continuous phase. Multi-Phase Particle In Cell (MPPIC) modeling is used to represent collisions without resolving particle-particle interactions, with optional mesh motion and mesh topology changes
MPPICFoam	Transient solver for the coupled transport of a single kinematic particle cloud including the effect of the volume fraction of particles on the continuous phase. Multi-Phase Particle In Cell (MPPIC) modeling is used to represent collisions without resolving particle-particle interactions
icoUncoupledKinematicParcelFoam	Transient solver for the passive transport of a single kinematic particle cloud
icoUncoupledKinematicParcelDyMFoam	Transient solver for the passive transport of a single kinematic particle cloud, with optional mesh motion and mesh topology changes
kinematicParcelFoam	Transient solver for incompressible, turbulent flow with kinematic, particle cloud, and surface film modelling
reactingParcelFoam	Transient solver for compressible, turbulent flow with a reacting, multiphase particle cloud, and surface film modelling
reactingHeterogenousParcelFoam	Transient solver for the coupled transport of a single kinematic particle cloud including the effect of the volume fraction of particles on the continuous phase. Multi-Phase Particle In Cell (MPPIC) modeling is used to represent collisions without resolving particle-particle interactions
simpleReactingParcelFoam	Steady-state solver for compressible, turbulent flow with reacting, multiphase particle clouds and optional sources/constraints

Continued on next page

Continued from previous page

simpleCoalParcelFoam	Steady-state solver for compressible, turbulent flow with coal particle clouds and optional sources/constraints
sprayFoam	Transient solver for compressible, turbulent flow with a spray particle cloud
engineFoam	Transient solver for compressible, turbulent engine flow with a spray particle cloud
simpleSprayFoam	Steady state solver for compressible, turbulent flow with a spray particle cloud and optional sources/constraints
sprayDyMFoam	Transient solver for compressible, turbulent flow with a spray particle cloud, with optional mesh motion and mesh topology changes
uncoupledKinematic-ParcelFoam	Transient solver for the passive transport of a particle cloud
uncoupledKinematic-ParcelDyMFoam	Transient solver for the passive transport of a particle cloud

Molecular dynamics methods

mdEquilibrationFoam	Solver to equilibrate and/or precondition molecular dynamics systems
mdFoam	Molecular dynamics solver for fluid dynamics

Direct simulation Monte Carlo methods

dsmcFoam	Direct simulation Monte Carlo (DSMC) solver for transient, multi-species flows

Electromagnetics

electrostaticFoam	Solver for electrostatics
magneticFoam	Solver for the magnetic field generated by permanent magnets
mhdFoam	Solver for magnetohydrodynamics (MHD): incompressible, laminar flow of a conducting fluid under the influence of a magnetic field

Stress analysis of solids

solidDisplacement-Foam	Transient segregated finite-volume solver of linear-elastic, small-strain deformation of a solid body, with optional thermal diffusion and thermal stresses

Continued on next page

Continued from previous page

solidEquilibriumDisplacementFoam	Steady-state segregated finite-volume solver of linear-elastic, small-strain deformation of a solid body, with optional thermal diffusion and thermal stresses
Finance	
financialFoam	Solves the Black-Scholes equation to price commodities

Table A.1: Standard solvers.

A.2 Standard utilities

The utilities with the OpenFOAM distribution are in the *$FOAM_UTILITIES* directory, reached quickly by typing `util` at the command line. Again the names are reasonably descriptive, *e.g.*ideasToFoam converts mesh data from the format written by I-DEAS to the OpenFOAM format. The current list of utilities distributed with OpenFOAM is given in Table A.2.

Pre-processing	
applyBoundaryLayer	Apply a simplified boundary-layer model to the velocity and turbulence fields based on the 1/7th power-law
boxTurb	Create a box of divergence-free turbulence conforming to a given energy spectrum
changeDictionary	Utility to change dictionary entries, e.g. can be used to change the patch type in the field and *polyMesh/boundary* files
createBoxTurb	Create a box of isotropic turbulence based on a user-specified energy spectrum
createExternalCoupledPatchGeometry	Generate the patch geometry (points and faces) for use with the externalCoupled functionObject
createZeroDirectory	Creates a zero directory with fields appropriate for the chosen solver and turbulence model. Operates on both single and multi-region cases
dsmcInitialise	Initialise a case for dsmcFoam by reading the initialisation dictionary *system/dsmcInitialise*
engineSwirl	Generate a swirl flow for engine calculations
faceAgglomerate	Agglomerate boundary faces using the pairPatchAgglomeration algorithm

Continued on next page

Continued from previous page

foamUpgradeCyclics	Tool to upgrade mesh and fields for split cyclics
mapFields	Maps volume fields from one mesh to another, reading and interpolating all fields present in the time directory of both cases
mapFieldsPar	Maps volume fields from one mesh to another, reading and interpolating all fields present in the time directory of both cases
mdInitialise	Initialises fields for a molecular dynamics (MD) simulation
PDRsetFields	Preparation of fields for PDRFoam
setAlphaField	Uses cutCellIso to create a volume fraction field from either a cylinder, a sphere or a plane
setExprBoundaryFields	Set boundary values using an expression
setExprFields	Set values on a selected set of cells/patch-faces via a dictionary
setFields	Set values on a selected set of cells/patch-faces via a dictionary
viewFactorsGen	View factors are calculated based on a face agglomeration array (finalAgglom generated by faceAgglomerate utility)
wallFunctionTable	Generates a table suitable for use by tabulated wall functions

Mesh generation

blockMesh	A multi-block mesh generator
extrude2DMesh	Create a 3D mesh by extruding a 2D mesh with specified thickness. For the 2D mesh, all faces are 2 points only, no front and back faces
foamyHexMesh	Conformal Voronoi automatic mesh generator
foamyHexMesh-BackgroundMesh	Writes out background mesh as constructed by foamyHexMesh and constructs distanceSurface
foamyHexMesh-SurfaceSimplify	Simplifies surfaces by resampling

Continued on next page

Continued from previous page

foamyQuadMesh	Conformal-Voronoi 2D extruding automatic mesher with grid or read initial points and point position relaxation with optional "squarification"
PDRblockMesh	A specialized single-block mesh generator for a rectilinear mesh in x-y-z
snappyHexMesh	Automatic split hex mesher. Refines and snaps to surface

Mesh conversion

ansysToFoam	Convert an ANSYS input mesh file (exported from I-DEAS) to OpenFOAM format
ccmToFoam	Reads CCM files as written by PROSTAR/STARCCM and writes an OPENOpenFOAM polyMesh
foamToCcm	Translates OPENOpenFOAM mesh and/or results to CCM format
cfx4ToFoam	Convert a CFX 4 mesh to OpenFOAM format
datToFoam	Reads in a datToFoam mesh file and outputs a points file. Used in conjunction with blockMesh
fireToFoam	Convert AVL/FIRE polyhedral mesh to OpenFOAM format
fluent3DMeshToFoam	Convert a Fluent mesh to OpenFOAM format
fluentMeshToFoam	Convert a Fluent mesh to OpenFOAM format, including multiple region and region boundary handling
foamMeshToFluent	Write an OpenFOAM mesh in Fluent mesh format
foamToFireMesh	Write an OpenFOAM mesh in AVL/FIRE fpma format
foamToStarMesh	Write an OpenFOAM mesh in STARCD/PROSTAR (v4) bnd/cel/vrt format
foamToSurface	Extract boundaries from an OpenFOAM mesh and write in a surface format
gambitToFoam	Convert a GAMBIT mesh to OpenFOAM format
gmshToFoam	Reads .msh file as written by Gmsh
ideasUnvToFoam	I-Deas unv format mesh conversion
kivaToFoam	Convert a KIVA grid to OpenFOAM

Continued on next page

Continued from previous page

mshToFoam	Convert .msh file generated by the Adventure system
netgenNeutralToFoam	Convert a neutral file format (Netgen v4.4) to OpenFOAM
plot3dToFoam	Plot3d mesh (ascii/formatted format) converter
star4ToFoam	Convert a STARCD/PROSTAR (v4) mesh into OpenFOAM format
tetgenToFoam	Convert tetgen .ele and .node and .face files to an OpenFOAM mesh
vtkUnstructuredToFoam	Convert legacy VTK file (ascii) containing an unstructured grid to an OpenFOAM mesh without boundary information
writeMeshObj	For mesh debugging: writes mesh as three separate OBJ files

Mesh manipulation

attachMesh	Attach topologically detached mesh using prescribed mesh modifiers
autoPatch	Divides external faces into patches based on (user supplied) feature angle
checkMesh	Checks validity of a mesh
createBaffles	Makes internal faces into boundary faces. Does not duplicate points, unlike mergeOrSplitBaffles
createPatch	Create patches out of selected boundary faces, which are either from existing patches or from a faceSet
deformedGeom	Deforms a polyMesh using a displacement field `U` and a scaling factor supplied as an argument
flattenMesh	Flattens the front and back planes of a 2D cartesian mesh
insideCells	Create a cellSet for cells with their centres 'inside' the defined surface. Requires surface to be closed and singly connected
mergeMeshes	Merges two meshes
mergeOrSplitBaffles	Detects boundary faces that share points (baffles). Either merges them or duplicate the points
mirrorMesh	Mirrors a mesh around a given plane

Continued on next page

Continued from previous page

moveDynamicMesh	Mesh motion and topological mesh changes utility
moveEngineMesh	Solver for moving meshes for engine calculations
moveMesh	A solver utility for moving meshes
objToVTK	Read obj line (not surface) file and convert into legacy VTK file
orientFaceZone	Corrects the orientation of faceZone
polyDualMesh	Creates the dual of a polyMesh, adhering to all the feature and patch edges
refineMesh	Utility to refine cells in multiple directions
renumberMesh	Renumbers the cell list in order to reduce the bandwidth, reading and renumbering all fields from all the time directories
rotateMesh	Rotates the mesh and fields from the direction n_1 to direction n_2
setSet	Manipulate a cell/face/point Set or Zone interactively
setsToZones	Add pointZones/faceZones/cellZones to the mesh from similar named pointSets/faceSets/cellSets
singleCellMesh	Reads all fields and maps them to a mesh with all internal faces removed (singleCellFvMesh) which gets written to region "singleCell"
splitMesh	Splits mesh by making internal faces external. Uses attach-Detach
splitMeshRegions	Splits mesh into multiple regions
stitchMesh	'Stitches' a mesh
subsetMesh	Create a mesh subset for a particular region of interest based on a cellSet or cellZone
topoSet	Operates on cellSets/faceSets/pointSets through a dictionary, normally *system/topoSetDict*
transformPoints	Transforms the mesh points in the polyMesh directory according to the translate, rotate and scale options
zipUpMesh	Reads in a mesh with hanging vertices and 'zips' up the cells to guarantee that all polyhedral cells of valid shape are closed

Continued on next page

Continued from previous page

Other mesh tools

collapseEdges	Collapses short edges and combines edges that are in line
combinePatchFaces	Checks for multiple patch faces on the same cell and combines them. Multiple patch faces can result from e.g. removal of refined neighbouring cells, leaving 4 exposed faces with same owner
modifyMesh	Manipulate mesh elements
PDRMesh	Mesh and field preparation utility for PDR type simulations
refineHexMesh	Refine a hex mesh by 2x2x2 cell splitting for the specified cellSet
refinementLevel	Attempt to determine the refinement levels of a refined cartesian mesh. Run *before* snapping
refineWallLayer	Refine cells next to specified patches
removeFaces	Remove faces specified in faceSet by combining cells on both sides
selectCells	Select cells in relation to surface
snappyRefineMesh	Refine cells near to a surface
splitCells	Utility to split cells with flat faces

Finite area

checkFaMesh	Check a finiteArea mesh
makeFaMesh	A mesh generator for finiteArea mesh. When called in parallel, it will also try to act like decomposePar, create procAddressing and decompose serial finite-area fields

Post-processing

noise	Perform noise analysis of pressure data
postProcess	Execute the set of functionObjects specified in the selected dictionary (which defaults to *system/controlDict*) or on the command-line for the selected set of times on the selected set of fields

Continued on next page

Continued from previous page

Post-processing graphics

Post-processing data converters

foamDataToFluent	Translate OpenFOAM data to Fluent format
foamToEnsight	Translate OpenFOAM data to EnSight format. An Ensight part is created for cellZones (unzoned cells are "internalMesh") and patches
foamToGMV	Translate OpenFOAM output to GMV readable files
foamToTetDualMesh	Convert polyMesh results to tetDualMesh
foamToVTK	General OpenFOAM to VTK file writer
smapToFoam	Translate a STARCD SMAP data file into OpenFOAM field format

Post-processing Lagrangian simulation

particleTracks	Generate particle tracks for cases that were computed using a tracked-parcel-type cloud
steadyParticleTracks	Generate a legacy VTK file of particle tracks for cases that were computed using a steady-state cloud

Post-processing lumped-mass simulation

lumpedPointForces	Extract force/moment information from simulation results that use the lumped points movement description
lumpedPointMovement	This utility can be used to produce VTK files to visualize the response points/rotations and the corresponding movement of the building surfaces
lumpedPointZones	Produce a VTK PolyData file lumpedPointZones.vtp in which the segmentation of the pressure integration zones can be visualized for diagnostic purposes. Does not use external coupling

Miscellaneous post-processing

engineCompRatio	Calculate the engine geometric compression ratio
pdfPlot	Generate a graph of a probability distribution function
postChannel	Post-process data from channel flow calculations

Continued on next page

Continued from previous page

profilingSummary	Collects information from profiling files in the processor sub-directories and summarizes the number of calls and time spent as max/avg/min values. If the values are identical for all processes, only a single value is written
temporalInterpolate	Interpolate fields between time-steps e.g. for animation

Noise post-processing

noise	Perform noise analysis of pressure data

Post-processing utility

postProcess	Execute the set of functionObjects specified in the selected dictionary (which defaults to *system/controlDict*) or on the command-line for the selected set of times on the selected set of fields

Surface mesh (e.g. STL) tools

surfaceAdd	Add two surfaces. Does geometric merge on points. Does not check for overlapping/intersecting triangles
surfaceBoolean-Features	Generates the extendedFeatureEdgeMesh for the interface between a boolean operation on two surfaces
surfaceCheck	Check geometric and topological quality of a surface
surfaceClean	Utility to clean surfaces
surfaceCoarsen	Surface coarsening using 'bunnylod'
surfaceConvert	Converts from one surface mesh format to another
surfaceFeatureConvert	Convert between edgeMesh formats
surfaceFeatureExtract	Extracts and writes surface features to file. All but the basic feature extraction is a work-in-progress
surfaceFind	Finds nearest face and vertex. Uses a zero origin unless otherwise specified
surfaceHookUp	Find close open edges and stitches the surface along them
surfaceInertia	Calculates the inertia tensor, principal axes and moments of a command line specified triSurface
surfaceInflate	Inflates surface. WIP. Checks for overlaps and locally lowers inflation distance

Continued on next page

Continued from previous page

surfaceLambdaMuSmooth	Smooth a surface using lambda/mu smoothing
surfaceMeshConvert	Convert between surface formats with optional scaling or transformations (rotate/translate) on a coordinateSystem
surfaceMeshExport	Export from surfMesh to various third-party surface formats with optional scaling or transformations (rotate/translate) on a coordinateSystem
surfaceMeshExtract	Extract patch or faceZone surfaces from a polyMesh. Depending on output surface format triangulates faces
surfaceMeshImport	Import from various third-party surface formats into surfMesh with optional scaling or transformations (rotate/translate) on a coordinateSystem
surfaceMeshInfo	Miscellaneous information about surface meshes. `To` simplify parsing of the output, the normal banner information is suppressed
surfaceOrient	Set normal consistent with respect to a user provided 'outside' point. If the -inside option is used the point is considered inside
surfacePatch	Patches (regionises) a surface using a user-selectable method
surfacePointMerge	Merges points on surface if they are within absolute distance. Since absolute distance use with care!
surfaceRedistributePar	(Re)distribution of triSurface. Either takes an undecomposed surface or an already decomposed surface and redistributes it so that each processor has all triangles that overlap its mesh
surfaceRefineRedGreen	Refine by splitting all three edges of triangle ('red' refinement)
surfaceSplitByPatch	Writes surface regions to separate files
surfaceSplitByTopology	Strips any baffle parts of a surface
surfaceSplitNonManifolds	Takes multiply connected surface and tries to split surface at multiply connected edges by duplicating points
surfaceSubset	A surface analysis tool that subsets the triSurface to choose a region of interest. Based on subsetMesh

Continued on next page

Continued from previous page

surfaceToPatch	Reads surface and applies surface regioning to a mesh. Uses boundaryMesh to do the hard work
surfaceTransform-Points	Transform (scale/rotate) a surface. Like transformPoints but for surfaces

Parallel processing

decomposePar	Automatically decomposes a mesh and fields of a case for parallel execution of OpenFOAM
reconstructPar	Reconstructs fields of a case that is decomposed for parallel execution of OpenFOAM
reconstructParMesh	Reconstructs a mesh using geometric information only
redistributePar	Redistributes existing decomposed mesh and fields according to the current settings in the decomposeParDict file

Thermophysical-related utilities

adiabaticFlameT	Calculate adiabatic flame temperature for a given fuel over a range of unburnt temperatures and equivalence ratios
chemkinToFoam	Convert CHEMKIN 3 thermodynamics and reaction data files into OpenFOAM format
equilibriumCO	Calculate the equilibrium level of carbon monoxide
equilibriumFlameT	Calculate the equilibrium flame temperature for a given fuel and pressure for a range of unburnt gas temperatures and equivalence ratios. Includes the effects of dissociation on O_2, H_2O and CO_2
mixtureAdiabatic-FlameT	Calculate adiabatic flame temperature for a given mixture at a given temperature

Miscellaneous utilities

foamDictionary	Interrogate and manipulate dictionaries
foamFormatConvert	Converts all `IOobject`s associated with a case into the format specified in the *controlDict*
foamHasLibrary	Test if given libraries can be loaded
foamHelp	Top level wrapper utility around foam help utilities
foamListRegions	List regions from constant/regionProperties

Continued on next page

Continued from previous page

foamListTimes	List times using the timeSelector, or use to remove selected time directories
foamRestoreFields	Adjust (restore) field names by removing the ending. The fields are selected automatically or can be specified as optional command arguments
addr2line	A simple, partial emulation of addr2line utility for Mac-OS
patchSummary	Write field and boundary condition info for each patch at each requested time instance

Table A.2: Standard utilities.

A.3 Standard libraries

The libraries with the OpenFOAM distribution are in the *$FOAM_LIB/$WM_OPTIONS* directory, reached quickly by typing `lib` at the command line. Again, the names are prefixed by `lib` and reasonably descriptive, *e.g.* incompressibleTransportModels contains the library of incompressible transport models. The library source code is typically located in the *$FOAM_SRC* directory, easily reached by typing `src` in the command line. Other libraries devoted to specific physical models for specific solvers may be located separately with the solver source code. For ease of presentation, the libraries are separated into two types:

General libraries those that provide general classes and associated functions listed in Table A.3;

Model libraries those that specify models used in computational continuum mechanics, listed in Table A.4, Table A.5 and Table A.6.

Library of basic OpenFOAM tools — OpenFOAM	
algorithms	Algorithms
containers	Container classes
db	Database classes
dimensionedTypes	dimensioned<Type> class and derivatives
dimensionSet	dimensionSet class
fields	Field classes
global	Global settings
graph	graph class
interpolations	Interpolation schemes
matrices	Matrix classes
memory	Memory management tools
meshes	Mesh classes
primitives	Primitive classes

Continued on next page

Continued from previous page

Finite volume method library — finiteVolume

cfdTools	CFD tools
fields	Volume, surface and patch field classes; includes boundary conditions
finiteVolume	Finite volume discretisation
fvMatrices	Matrices for finite volume solution
fvMesh	Meshes for finite volume discretisation
interpolation	Field interpolation and mapping
surfaceMesh	Mesh surface data for finite volume discretisation
volMesh	Mesh volume (cell) data for finite volume discretisation

Post-processing libraries

sampling	Tools for sampling point, line and surface field data
fieldFunctionObjects	Field function objects including field averaging, min/max, *etc.*
forces	Tools for post-processing force/lift/drag data with function objects
runTimePostProcessing	Image generation and manipulation using function objects
lagrangianFunctionObjects	Lagrangian cloud-based function objects
solverFunctionObjects	Solver-based function objects, *e.g.* adding scalar transport
utilityFunctionObjects	Utility function objects

Solution and mesh manipulation libraries

snappyMesh	Library of functionality for the snappyHexMesh utility
blockMesh	Library of functionality for the blockMesh utility
dynamicMesh	For solving systems with moving meshes
dynamicFvMesh	Library for a finite volume mesh that can move and undergo topological changes
edgeMesh	For handling edge-based mesh descriptions
fvMotionSolvers	Finite volume mesh motion solvers
ODE	Solvers for ordinary differential equations
meshTools	Tools for handling a OpenFOAM mesh
surfMesh	Library for handling surface meshes of different formats
triSurface	For handling standard triangulated surface-based mesh descriptions
topoChangerFvMesh	Topological changes functionality (largely redundant)

Lagrangian particle tracking libraries

basic	Basic Lagrangian, or particle-tracking, solution scheme
coalCombustion	Coal dust combustion modelling
distributionModels	Particle distribution function modelling
dsmc	Direct simulation Monte Carlo method modelling
intermediate	Particle-tracking kinematics, thermodynamics, multispecies reactions, particle forces, *etc.*
molecule	Molecule classes for molecular dynamics
molecularMeasurements	For making measurements in molecular dynamics
potential	Intermolecular potentials for molecular dynamics
solidParticle	Solid particle implementation
spray	Liquid spray and injection modelling

Continued on next page

Continued from previous page

Miscellaneous libraries

conversion	Tools for mesh and data conversions
decompositionMethods	Tools for domain decomposition
engine	Tools for engine calculations
fileFormats	Core routines for reading/writing data in some third-party formats
genericFvPatchField	A generic patch field
MGridGenGAMG-Agglomeration	Library for cell agglomeration using the `MGridGen` algorithm
pairPatchAgglom-eration	Primitive pair patch agglomeration method
OSspecific	Operating system specific functions
randomProcesses	Tools for analysing and generating random processes

Parallel libraries

distributed	Tools for searching and IO on distributed surfaces
reconstruct	Mesh/field reconstruction library
scotchDecomp	Scotch domain decomposition library
ptsotchDecomp	PTScotch domain decomposition library

Table A.3: Shared object libraries for general use.

Basic thermophysical models — basicThermophysicalModels

hePsiThermo	General thermophysical model calculation based on enthalpy h or internal energy e, and compressibility ψ
heRhoThermo	General thermophysical model calculation based on enthalpy h or internal energy e, and density ρ
pureMixture	General thermophysical model calculation for passive gas mixtures

Reaction models — reactionThermophysicalModels

hePsiMixtureThermo	Calculates enthalpy for combustion mixture based on enthalpy h or internal energy e, and ψ
heRhoMixtureThermo	Calculates enthalpy for combustion mixture based on enthalpy h or internal energy e, and ρ
heheuMixtureThermo	Calculates enthalpy h or internal energy e for unburnt u gas and combustion mixture
homogeneousMixture	Combustion mixture based on normalised fuel mass fraction b
inhomogeneousMixture	Combustion mixture based on b and total fuel mass fraction f_t
veryInhomogeneousMixture	Combustion mixture based on b, f_t and unburnt fuel mass fraction f_u
dieselMixture	Combustion mixture based on f_t and f_u

Continued on next page

Continued from previous page

basicMultiComponent-Mixture	Basic mixture based on multiple components
multiComponentMixture	Derived mixture based on multiple components
reactingMixture	Combustion mixture using thermodynamics and reaction schemes
egrMixture	Exhaust gas recirculation mixture

Radiation models — radiationModels

fvDOM	Finite volume discrete ordinate method
P1	P1 model
solarLoad	Solar load radiation model
viewFactor	View factor radiation model

Laminar flame speed models — laminarFlameSpeedModels

constLaminarFlameSpeed	Constant laminar flame speed
GuldersLaminarFlameSpeed	Gulder's laminar flame speed model
GuldersEGRLaminar-FlameSpeed	Gulder's laminar flame speed model with exhaust gas recirculation modelling

Barotropic compressibility models — barotropicCompressibilityModels

linear	Linear compressibility model
Chung	Chung compressibility model
Wallis	Wallis compressibility model

Thermophysical properties of gaseous species — specie

icoPolynomial	Incompressible polynomial equation of state, *e.g.* for liquids
perfectGas	Perfect gas equation of state
eConstThermo	Constant specific heat c_p model with evaluation of internal energy e and entropy s
hConstThermo	Constant specific heat c_p model with evaluation of enthalpy h and entropy s
hPolynomialThermo	c_p evaluated by a function with coefficients from polynomials, from which h, s are evaluated
janafThermo	c_p evaluated by a function with coefficients from JANAF thermodynamic tables, from which h, s are evaluated
specieThermo	Thermophysical properties of species, derived from c_p, h and/or s
constTransport	Constant transport properties
polynomialTransport	Polynomial based temperature-dependent transport properties
sutherlandTransport	Sutherland's formula for temperature-dependent transport properties

Functions/tables of thermophysical properties — thermophysicalFunctions

NSRDSfunctions	National Standard Reference Data System (NSRDS) - American Institute of Chemical Engineers (AICHE) data compilation tables

Continued on next page

Continued from previous page

APIfunctions	American Petroleum Institute (API) function for vapour mass diffusivity

Chemistry model — chemistryModel

chemistryModel	Chemical reaction model
chemistrySolver	Chemical reaction solver

Other libraries

liquidProperties	Thermophysical properties of liquids
liquidMixtureProperties	Thermophysical properties of liquid mixtures
basicSolidThermo	Thermophysical models of solids
solid	Thermodynamics of solid species
SLGThermo	Thermodynamic package for solids, liquids and gases
solidProperties	Thermophysical properties of solids
solidMixtureProperties	Thermophysical properties of solid mixtures
thermalPorousZone	Porous zone definition based on cell zones that includes terms for energy equations

Table A.4: Libraries of thermophysical models.

RAS turbulence models — RASModels

laminar	Dummy turbulence model for laminar flow
kEpsilon	Standard $k-\varepsilon$ model
kOmega	$k-\omega$ model
kOmegaSST	$k-\omega-SST$ model
kOmegaSSTLM	Langtry-Menter 4-equation transitional SST model
kOmegaSSTSAS	$k-\omega-SST-SAS$ model
LaunderSharmaKE	Launder-Sharma low-Re $k-\varepsilon$ model
LRR	Launder-Reece-Rodi RSTM
realizableKE	Realizable $k-\varepsilon$ model
RNGkEpsilon	$RNG-k-\varepsilon$ model
SpalartAllmaras	Spalart-Allmaras 1-eqn mixing-length model
SSG	Speziale, Sarkar and Gatski Reynolds-stress model
v2f	$v2-f$ model

Large-eddy simulation (LES) filters — LESfilters

laplaceFilter	Laplace filters
simpleFilter	Simple filter
anisotropicFilter	Anisotropic filter

Large-eddy simulation deltas — LESdeltas

PrandtlDelta	Prandtl delta
cubeRootVolDelta	Cube root of cell volume delta
maxDeltaxyz	Maximum of x, y and z; for structured hex cells only
smoothDelta	Smoothing of delta

LES turbulence models — LESModels

Continued on next page

Continued from previous page

DeardorffDiffStress	Differential SGS Stress model
dynamicKEqn	Dynamic one equation eddy-viscosity
dynamicLagrangian	Dynamic SGS model with Lagrangian averaging
kEqn	One equation eddy-viscosity model
Smagorinsky	Smagorinsky SGS model
WALE	Wall-adapting local eddy-viscosity (WALE) model

DES turbulence models — DESModels

kOmegaSSTDES	$k-omega-SST$ delayed eddy simulation (DES) model
kOmegaSSTDDES	$k-omega-SST$ delayed detached eddy simulation (DDES) model
kOmegaSSTIDDES	$k-omega-SST$ improved delayed detached eddy simulation (DDES) model
SpalartAllmarasDES	Spalart-Allmaras delayed eddy simulation (DES) model
SpalartAllmarasDDES	Spalart-Allmaras delayed detached eddy simulation (DDES) model
SpalartAllmarasIDDES	Spalart-Allmaras improved delayed detached eddy simulation (DDES) model

Table A.5: Libraries of RAS, DES and LES turbulence models.

Transport models for incompressible fluids — incompressibleTransportModels

Newtonian	Linear viscous fluid model
CrossPowerLaw	Cross Power law nonlinear viscous model
BirdCarreau	Bird-Carreau nonlinear viscous model
HerschelBulkley	Herschel-Bulkley nonlinear viscous model
powerLaw	Power-law nonlinear viscous model
interfaceProperties	Models for the interface, *e.g.* contact angle, in multiphase simulations

Miscellaneous transport modelling libraries

interfaceProperties	Calculation of interface properties
twoPhaseInterfaceProperties	Two phase interface properties models, including boundary conditions
surfaceFilmModels	Surface film models

Table A.6: Shared object libraries of transport models.

A.4 Standard boundary conditions

basic

Continued on next page

Continued from previous page

calculated	This boundary condition is not designed to be evaluated; it is assmued that the value is assigned via field assignment, and not via a call to e.g. updateCoeffs or evaluate
fixedValue	This boundary condition supplies a fixed value constraint, and is the base class for a number of other boundary conditions
fixedGradient	This boundary condition supplies a fixed gradient condition, such that the patch values are calculated using:
zeroGradient	This boundary condition applies a zero-gradient condition from the patch internal field onto the patch faces
mixed	This boundary condition provides a base class for 'mixed' type boundary conditions, i.e. conditions that mix fixed value and patch-normal gradient conditions
directionMixed	Base class for direction-mixed boundary conditions
extrapolatedCalculated	This boundary condition applies a zero-gradient condition from the patch internal field onto the patch faces when evaluated but may also be assigned. snGrad returns the patch gradient evaluated from the current internal and patch field values rather than returning zero

Table A.7: basic boundary conditions.

constraint	
cyclic	This boundary condition enforces a cyclic condition between a pair of boundaries
cyclicACMI	This boundary condition enforces a cyclic condition between a pair of boundaries, whereby communication between the patches is performed using an arbitrarily coupled mesh interface (ACMI) interpolation
cyclicAMI	This boundary condition enforces a cyclic condition between a pair of boundaries, whereby communication between the patches is performed using an arbitrary mesh interface (AMI) interpolation
cyclicSlip	This boundary condition is a light wrapper around the cyclicFvPatchField condition, providing no new functionality

Continued on next page

Continued from previous page

empty	This boundary condition provides an 'empty' condition for reduced dimensions cases, i.e. 1- and 2-D geometries. Apply this condition to patches whose normal is aligned to geometric directions that do not constitue solution directions
jumpCyclic	This boundary condition provides a base class for coupled-cyclic conditions with a specified 'jump' (or offset) between the values
jumpCyclicAMI	This boundary condition provides a base class that enforces a cyclic condition with a specified 'jump' (or offset) between a pair of boundaries, whereby communication between the patches is performed using an arbitrary mesh interface (AMI) interpolation
nonuniformTransform-Cyclic	This boundary condition enforces a cyclic condition between a pair of boundaries, incorporating a non-uniform transformation
processor	This boundary condition enables processor communication across patches
processorCyclic	This boundary condition enables processor communication across cyclic patches
symmetry	This boundary condition enforces a symmetry constraint
symmetryPlane	This boundary condition enforces a symmetryPlane constraint
wedge	This boundary condition is similar to the cyclic condition, except that it is applied to 2-D geometries

Table A.8: constraint boundary conditions.

Inlet	
cylindricalInletVelocity	This boundary condition describes an inlet vector boundary condition in cylindrical coordinates given a central axis, central point, rpm, axial and radial velocity
fanPressure	This boundary condition can be applied to assign either a pressure inlet or outlet total pressure condition for a fan
fixedFluxExtrapolated-Pressure	This boundary condition sets the pressure gradient to the provided value such that the flux on the boundary is that specified by the velocity boundary condition

Continued on next page

Continued from previous page

fixedFluxPressure	This boundary condition sets the pressure gradient to the provided value such that the flux on the boundary is that specified by the velocity boundary condition
fixedMean	This boundary condition extrapolates field to the patch using the near-cell values and adjusts the distribution to match the specified, optionally time-varying, mean value
fixedMeanOutletInlet	This boundary condition extrapolates field to the patch using the near-cell values and adjusts the distribution to match the specified, optionally time-varying, mean value. This extrapolated field is applied as a fixedValue for outflow faces but zeroGradient is applied to inflow faces
fixedNormalInlet-OutletVelocity	This velocity inlet/outlet boundary condition combines a fixed normal component obtained from the "normalVelocity" patchField supplied with a fixed or zero-gradiented tangential component
fixedPressure-CompressibleDensity	This boundary condition calculates a (liquid) compressible density as a function of pressure and fluid properties:
flowRateInletVelocity	Velocity inlet boundary condition either correcting the extrapolated velocity or creating a uniform velocity field normal to the patch adjusted to match the specified flow rate
freestream	This boundary condition provides a free-stream condition. It is a 'mixed' condition derived from the inletOutlet condition, whereby the mode of operation switches between fixed (free stream) value and zero gradient based on the sign of the flux
freestreamPressure	This boundary condition provides a free-stream condition for pressure
freestreamVelocity	This boundary condition provides a free-stream condition for velocity
mappedFlowRate	Describes a volumetric/mass flow normal vector boundary condition by its magnitude as an integral over its area
mappedVelocityFlux-FixedValue	This boundary condition maps the velocity and flux from a neighbour patch to this patch
outletInlet	This boundary condition provides a generic inflow condition, with specified outflow for the case of reverse flow

Continued on next page

Continued from previous page

outletMappedUniformInlet	The outletMappedUniformInlet is an inlet boundary condition that - averages the patch field of <Type >over a specified "outlet" patch and uniformly applies the averaged value over a specified inlet patch. - optionally, the averaged value can be scaled and/or offset by a pair of specified values
plenumPressure	This boundary condition provides a plenum pressure inlet condition. This condition creates a zero-dimensional model of an enclosed volume of gas upstream of the inlet. The pressure that the boundary condition exerts on the inlet boundary is dependent on the thermodynamic state of the upstream volume. The upstream plenum density and temperature are time-stepped along with the rest of the simulation, and momentum is neglected. The plenum is supplied with a user specified mass flow and temperature
pressureDirectedInletOutletVelocity	This velocity inlet/outlet boundary condition is applied to velocity boundaries where the pressure is specified. A zero-gradient condition is applied for outflow (as defined by the flux); for inflow, the velocity is obtained from the flux with the specified inlet direction
pressureDirectedInletVelocity	This velocity inlet boundary condition is applied to patches where the pressure is specified. The inflow velocity is obtained from the flux with the specified inlet direction" direction
pressureInletOutletParSlipVelocity	This velocity inlet/outlet boundary condition for pressure boundary where the pressure is specified. A zero-gradient is applied for outflow (as defined by the flux); for inflow, the velocity is obtained from the flux with the specified inlet direction
pressureInletOutletVelocity	This velocity inlet/outlet boundary condition is applied to velocity boundaries where the pressure is specified. A zero-gradient condition is applied for outflow (as defined by the flux); for inflow, the velocity is obtained from the patch-face normal component of the internal-cell value
pressureInletUniformVelocity	This velocity inlet boundary condition is applied to patches where the pressure is specified. The uniform inflow velocity is obtained by averaging the flux over the patch, and then applying it in the direction normal to the patch faces
pressureInletVelocity	This velocity inlet boundary condition is applied to patches where the pressure is specified. The inflow velocity is obtained from the flux with a direction normal to the patch faces

Continued on next page

Continued from previous page

pressureNormalInlet-OutletVelocity	This velocity inlet/outlet boundary condition is applied to patches where the pressure is specified. A zero-gradient condition is applied for outflow (as defined by the flux); for inflow, the velocity is obtained from the flux with a direction normal to the patch faces
pressurePermeable-AlphaInletOutlet-Velocity	The pressurePermeableAlphaInletOutletVelocity is a velocity inlet-outlet boundary condition which can be applied to velocity boundaries for multiphase flows when the pressure boundary condition is specified
pressurePIDControl-InletVelocity	This boundary condition tries to generate an inlet velocity that maintains a specified pressure drop between two face zones downstream. The zones should fully span a duct through which all the inlet flow passes
rotatingPressureInlet-OutletVelocity	This velocity inlet/outlet boundary condition is applied to patches in a rotating frame where the pressure is specified. A zero-gradient is applied for outflow (as defined by the flux); for inflow, the velocity is obtained from the flux with a direction normal to the patch faces
rotatingTotalPressure	This boundary condition provides a total pressure condition for patches in a rotating frame
supersonicFreestream	This boundary condition provides a supersonic free-stream condition
surfaceNormalFixed-Value	This boundary condition provides a surface-normal vector boundary condition by its magnitude
swirlFlowRateInlet-Velocity	This boundary condition provides a volumetric- OR mass-flow normal vector boundary condition by its magnitude as an integral over its area with a swirl component determined by the angular speed, given in revolutions per minute (RPM)
swirlInletVelocity	This boundary condition describes an inlet vector boundary condition in swirl coordinates given a central axis, central point, axial, radial and tangential velocity profiles
syringePressure	This boundary condition provides a pressure condition, obtained from a zero-D model of the cylinder of a syringe
timeVaryingMapped-FixedValue	This boundary conditions interpolates the values from a set of supplied points in space and time
totalPressure	This boundary condition provides a total pressure condition. Four variants are possible:

Continued on next page

Continued from previous page

totalTemperature	This boundary condition provides a total temperature condition
turbulentDFSEMInlet	The turbulentDFSEMInlet is a synthesised-eddy based velocity inlet boundary condition to generate synthetic turbulence-alike time-series from a given set of turbulence statistics for LES and hybrid RANS-LES computations
turbulentDigitalFilterInlet	Digital-filter based boundary condition for velocity, i.e. `U`, to generate synthetic turbulence-alike time-series for LES and DES turbulent flow computations from input turbulence statistics
turbulentInlet	This boundary condition produces spatiotemporal-variant field by summing a set of pseudo-random numbers and a given spatiotemporal-invariant mean field. The field can be any type, e.g. scalarField. At a single point and time, all components are summed by the same random number, e.g. velocity components (u, v, w) are summed by the same random number, p; thus, output is (u+p, v+p, w+p)
turbulentIntensityKineticEnergyInlet	This boundary condition provides a turbulent kinetic energy condition, based on user-supplied turbulence intensity, defined as a fraction of the mean velocity:
uniformNormalFixedValue	This boundary condition provides a uniform surface-normal vector boundary condition by its magnitude
uniformTotalPressure	This boundary condition provides a time-varying form of the uniform total pressure boundary condition Foam::totalPressureFvPatchField
variableHeightFlowRateInletVelocity	This boundary condition provides a velocity boundary condition for multphase flow based on a user-specified volumetric flow rate
variableHeightFlowRate	This boundary condition provides a phase fraction condition based on the local flow conditions, whereby the values are constrained to lay between user-specified upper and lower bounds. The behaviour is described by:
waveSurfacePressure	This is a pressure boundary condition, whose value is calculated as the hydrostatic pressure based on a given displacement:

Table A.9: Inlet boundary conditions.

Outlet	
acousticWave-Transmissive	This boundary condition provides a wave transmissive outflow condition, based on solving DDt(W, field) = 0 at the boundary W is the wave velocity and field is the field to which this boundary condition is applied. The wave speed is input in the BC
advective	This boundary condition provides an advective outflow condition, based on solving DDt(W, field) = 0 at the boundary where W is the wave velocity and field is the field to which this boundary condition is applied
fanPressure	This boundary condition can be applied to assign either a pressure inlet or outlet total pressure condition for a fan
fixedNormalInlet-OutletVelocity	This velocity inlet/outlet boundary condition combines a fixed normal component obtained from the "normalVelocity" patchField supplied with a fixed or zero-gradiented tangential component
flowRateOutletVelocity	Velocity outlet boundary condition which corrects the extrapolated velocity to match the specified flow rate
fluxCorrectedVelocity	This boundary condition provides a velocity outlet boundary condition for patches where the pressure is specified. The outflow velocity is obtained by "zeroGradient" and then corrected from the flux:
freestream	This boundary condition provides a free-stream condition. It is a 'mixed' condition derived from the inletOutlet condition, whereby the mode of operation switches between fixed (free stream) value and zero gradient based on the sign of the flux
freestreamPressure	This boundary condition provides a free-stream condition for pressure
freestreamVelocity	This boundary condition provides a free-stream condition for velocity
inletOutlet	This boundary condition provides a generic outflow condition, with specified inflow for the case of return flow
inletOutletTotal-Temperature	This boundary condition provides an outflow condition for total temperature for use with supersonic cases, where a user-specified value is applied in the case of reverse flow

Continued on next page

Continued from previous page

matchedFlowRate-OutletVelocity	Velocity outlet boundary condition which corrects the extrapolated velocity to match the flow rate of the specified corresponding inlet patch
outletPhaseMean-Velocity	This boundary condition adjusts the velocity for the given phase to achieve the specified mean thus causing the phase-fraction to adjust according to the mass flow rate
pressureDirectedInlet-OutletVelocity	This velocity inlet/outlet boundary condition is applied to velocity boundaries where the pressure is specified. A zero-gradient condition is applied for outflow (as defined by the flux); for inflow, the velocity is obtained from the flux with the specified inlet direction
pressureInletOutletPar-SlipVelocity	This velocity inlet/outlet boundary condition for pressure boundary where the pressure is specified. A zero-gradient is applied for outflow (as defined by the flux); for inflow, the velocity is obtained from the flux with the specified inlet direction
pressureInletOutlet-Velocity	This velocity inlet/outlet boundary condition is applied to velocity boundaries where the pressure is specified. A zero-gradient condition is applied for outflow (as defined by the flux); for inflow, the velocity is obtained from the patch-face normal component of the internal-cell value
pressureNormalInlet-OutletVelocity	This velocity inlet/outlet boundary condition is applied to patches where the pressure is specified. A zero-gradient condition is applied for outflow (as defined by the flux); for inflow, the velocity is obtained from the flux with a direction normal to the patch faces
pressurePermeable-AlphaInletOutlet-Velocity	The pressurePermeableAlphaInletOutletVelocity is a velocity inlet-outlet boundary condition which can be applied to velocity boundaries for multiphase flows when the pressure boundary condition is specified
rotatingPressureInlet-OutletVelocity	This velocity inlet/outlet boundary condition is applied to patches in a rotating frame where the pressure is specified. A zero-gradient is applied for outflow (as defined by the flux); for inflow, the velocity is obtained from the flux with a direction normal to the patch faces
rotatingTotalPressure	This boundary condition provides a total pressure condition for patches in a rotating frame
supersonicFreestream	This boundary condition provides a supersonic free-stream condition

Continued on next page

Continued from previous page

totalPressure	This boundary condition provides a total pressure condition. Four variants are possible:
totalTemperature	This boundary condition provides a total temperature condition
uniformInletOutlet	Variant of inletOutlet boundary condition with uniform inletValue
uniformTotalPressure	This boundary condition provides a time-varying form of the uniform total pressure boundary condition Foam::totalPressureFvPatchField
waveTransmissive	This boundary condition provides a wave transmissive outflow condition, based on solving DDt(W, field) = 0 at the boundary W is the wave velocity and field is the field to which this boundary condition is applied

Table A.10: Outlet boundary conditions.

Wall	
fixedFluxExtrapolatedPressure	This boundary condition sets the pressure gradient to the provided value such that the flux on the boundary is that specified by the velocity boundary condition
fixedFluxPressure	This boundary condition sets the pressure gradient to the provided value such that the flux on the boundary is that specified by the velocity boundary condition
fixedNormalSlip	This boundary condition sets the patch-normal component to the field (vector or tensor) to the patch-normal component of a user specified field. The tangential component is treated as slip, i.e. copied from the internal field
movingWallVelocity	This boundary condition provides a velocity condition for cases with moving walls
noSlip	This boundary condition fixes the velocity to zero at walls
partialSlip	This boundary condition provides a partial slip condition. The amount of slip is controlled by a user-supplied field
rotatingWallVelocity	This boundary condition provides a rotational velocity condition
slip	This boundary condition provides a slip constraint

Continued on next page

Continued from previous page

translatingWallVelocity	This boundary condition provides a velocity condition for translational motion on walls

Table A.11: Wall boundary conditions.

Coupled	
activeBaffleVelocity	This velocity boundary condition simulates the opening of a baffle due to local flow conditions, by merging the behaviours of wall and cyclic conditions. The baffle joins two mesh regions, where the open fraction determines the interpolation weights applied to each cyclic- and neighbour-patch contribution
activePressureForce-BaffleVelocity	This boundary condition is applied to the flow velocity, to simulate the opening or closure of a baffle due to area averaged pressure or force delta, between both sides of the baffle. This is achieved by merging the behaviours of wall and cyclic baffles
fan	This boundary condition provides a jump condition, using the cyclic condition as a base
fixedJumpAMI	This boundary condition provides a jump condition, across non-conformal cyclic path-pairs, employing an arbitraryMesh-Interface (AMI)
fixedJump	This boundary condition provides a jump condition, using the cyclic condition as a base
mappedField	This boundary condition provides a self-contained version of the mapped condition. It does not use information on the patch; instead it holds the data locally
mappedFixedInternal-Value	This boundary condition maps the boundary and internal values of a neighbour patch field to the boundary and internal values of *this
mappedFixedPushed-InternalValue	This boundary condition maps the boundary values of a neighbour patch field to the boundary and internal cell values of *this
mappedFixedValue	This boundary condition maps the value at a set of cells or patch faces back to *this
mappedFlowRate	Describes a volumetric/mass flow normal vector boundary condition by its magnitude as an integral over its area

Continued on next page

Continued from previous page

mappedMixed	This boundary condition maps the value at a set of cells or patch faces back to *this
mappedVelocityFlux-FixedValue	This boundary condition maps the velocity and flux from a neighbour patch to this patch
swirlFanVelocity	This boundary condition provides a jump condition for `U` across a cyclic pressure jump condition and applies a transformation to `U`
timeVaryingMapped-FixedValue	This boundary conditions interpolates the values from a set of supplied points in space and time
uniformJumpAMI	This boundary condition provides a jump condition, using the cyclicAMI condition as a base. The jump is specified as a time-varying uniform value across the patch
uniformJump	This boundary condition provides a jump condition, using the cyclic condition as a base. The jump is specified as a time-varying uniform value across the patch

Table A.12: Coupled boundary conditions.

Generic	
codedFixedValue	Constructs on-the-fly a new boundary condition (derived from fixedValueFvPatchField) which is then used to evaluate
codedMixed	Constructs on-the-fly a new boundary condition (derived from mixedFvPatchField) which is then used to evaluate
electrostaticDeposition	The electrostaticDeposition is a boundary condition to calculate electric potential (`V`) on a given boundary based on film thickness (`h`) and film resistance (`R`) fields which are updated based on a given patch-normal current density field (`jn`), Coulombic efficiency and film resistivity
fixedInternalValueFv-PatchField	This boundary condition provides a mechanism to set boundary (cell) values directly into a matrix, i.e. to set a constraint condition. Default behaviour is to act as a zero gradient condition
fixedNormalSlip	This boundary condition sets the patch-normal component to the field (vector or tensor) to the patch-normal component of a user specified field. The tangential component is treated as slip, i.e. copied from the internal field

Continued on next page

Continued from previous page

fixedProfile	This boundary condition provides a fixed value profile condition
interfaceCompression	Applies interface-compression to the phase-fraction distribution at the patch by setting the phase-fraction to 0 if it is below 0.5, otherwise to 1
mappedField	This boundary condition provides a self-contained version of the mapped condition. It does not use information on the patch; instead it holds the data locally
mappedFixedInternalValue	This boundary condition maps the boundary and internal values of a neighbour patch field to the boundary and internal values of *this
mappedFixedPushedInternalValue	This boundary condition maps the boundary values of a neighbour patch field to the boundary and internal cell values of *this
mappedFixedValue	This boundary condition maps the value at a set of cells or patch faces back to *this
mappedMixed	This boundary condition maps the value at a set of cells or patch faces back to *this
partialSlip	This boundary condition provides a partial slip condition. The amount of slip is controlled by a user-supplied field
phaseHydrostaticPressure	This boundary condition provides a phase-based hydrostatic pressure condition, calculated as:
prghPressure	This boundary condition provides static pressure condition for `p_rgh`, calculated as:
prghTotalHydrostaticPressure	This boundary condition provides static pressure condition for `p_rgh`, calculated as:
prghTotalPressure	This boundary condition provides static pressure condition for `p_rgh`, calculated as:
rotatingWallVelocity	This boundary condition provides a rotational velocity condition
scaledFixedValue	This condition applies a scalar multiplier to the value of another boundary condition
slip	This boundary condition provides a slip constraint

Continued on next page

Continued from previous page

surfaceNormalFixed-Value	This boundary condition provides a surface-normal vector boundary condition by its magnitude
translatingWallVelocity	This boundary condition provides a velocity condition for translational motion on walls
uniformDensity-HydrostaticPressure	This boundary condition provides a hydrostatic pressure condition, calculated as:
uniformFixedGradient	This boundary condition provides a uniform fixed gradient condition
uniformFixedValue	This boundary condition provides a uniform fixed value condition
uniformNormalFixed-Value	This boundary condition provides a uniform surface-normal vector boundary condition by its magnitude

Table A.13: Generic boundary conditions.

Index

Symbols

Numbers

A

B

C

D

E

F

G

H

I

J

K

L

M

N

O

P

Q

R

S

T

U

V

W

X

Y

Z