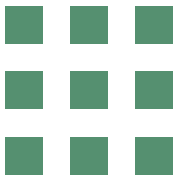


OpenFOAM

The OpenFOAM Foundation

User Guide

version 10

12th July 2022

https://openfoam.org

Author: Christopher J. Greenshields, CFD Direct Ltd.

Typeset in LaTeX.

License

THE WORK (AS DEFINED BELOW) IS PROVIDED UNDER THE TERMS OF THIS CREATIVE COMMONS PUBLIC LICENSE ("CCPL" OR "LICENSE"). THE WORK IS PROTECTED BY COPYRIGHT AND/OR OTHER APPLICABLE LAW. ANY USE OF THE WORK OTHER THAN AS AUTHORIZED UNDER THIS LICENSE OR COPYRIGHT LAW IS PROHIBITED.

BY EXERCISING ANY RIGHTS TO THE WORK PROVIDED HERE, YOU ACCEPT AND AGREE TO BE BOUND BY THE TERMS OF THIS LICENSE. TO THE EXTENT THIS LICENSE MAY BE CONSIDERED TO BE A CONTRACT, THE LICENSOR GRANTS YOU THE RIGHTS CONTAINED HERE IN CONSIDERATION OF YOUR ACCEPTANCE OF SUCH TERMS AND CONDITIONS.

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;

b. and, 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.

e. This License may not be modified without the mutual written agreement of the Licensor and You. 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

ANSYS is a registered trademark of ANSYS Inc.
CFX is a registered trademark of Ansys Inc.
CHEMKIN is a registered trademark of Reaction Design Corporation.
EnSight is a registered trademark of Computational Engineering International Ltd.
Fieldview is a registered trademark of Intelligent Light.
Fluent is a registered trademark of Ansys Inc.
GAMBIT is a registered trademark of Ansys Inc.
Icem-CFD is a registered trademark of Ansys Inc.
I-DEAS is a registered trademark of Structural Dynamics Research Corporation.
Linux is a registered trademark of Linus Torvalds.
OpenFOAM is a registered trademark of ESI Group.
ParaView is a registered trademark of Kitware.
STAR-CD is a registered trademark of CD-Adapco.
UNIX is a registered trademark of The Open Group.

Contents

Chapter 1

Introduction

This guide accompanies the release of version 10 of the Open Source Field Operation and Manipulation (OpenFOAM) C++ libraries. It provides a description of the basic operation of OpenFOAM, first through a set of tutorial exercises in chapter 2 and later by a more detailed description of the individual components that make up OpenFOAM.

OpenFOAM is a framework for developing *application* executables that use packaged functionality contained within a collection of over 100 *C++ libraries*. OpenFOAM is shipped with approximately 200 pre-built applications that fall into two categories: *solvers*, that are each designed to solve a specific problem in fluid (or continuum) mechanics; and *utilities*, that are designed to perform tasks that involve data manipulation. The solvers in OpenFOAM cover a wide range of problems in fluid dynamics, as described in chapter 3.

Users can extend the collection of solvers, utilities and libraries in OpenFOAM, using 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. The pre-processing and running of OpenFOAM cases is described

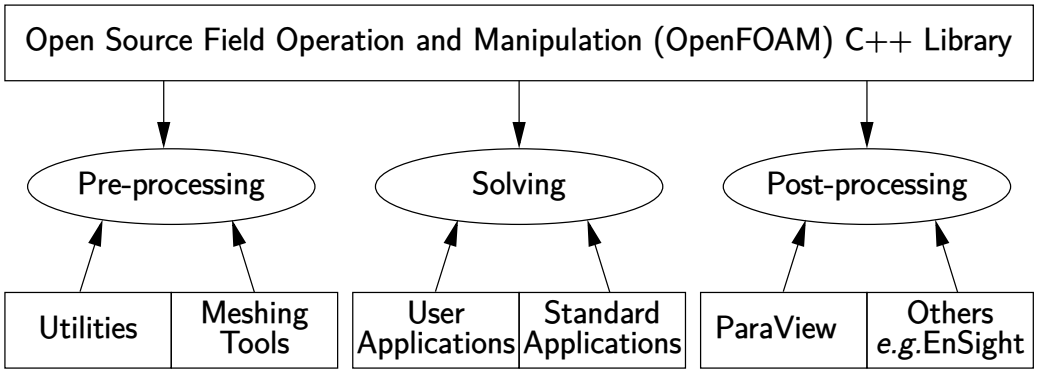


Figure 1.1: Overview of OpenFOAM structure.

in chapter 4. In chapter 5, we cover both the generation of meshes using the mesh generator supplied with OpenFOAM and conversion of mesh data generated by third-party products. Post-processing is described in chapter 6 and some aspects of physical modelling, *e.g.* transport and thermophysical modelling, are described in in chapter 7.

Chapter 2

Tutorials

In this chapter we shall describe in detail the process of setup, simulation and post-processing for some OpenFOAM test cases, with the principal aim of introducing a user to the basic procedures of running OpenFOAM. The *$FOAM_TUTORIALS* directory contains many more cases that demonstrate the use of all the solvers and many utilities supplied with OpenFOAM.

Before attempting to run the tutorials, the user must first make sure that OpenFOAM is installed correctly. Cases in the tutorials will be copied into the so-called *run* directory, an OpenFOAM project directory in the user's file system at *$HOME/OpenFOAM/-<USER>-10/run* where <*USER*> is the account login name and "10" is the OpenFOAM version number. The *run* directory is represented by the $FOAM_RUN environment variable enabling the user to check its existence conveniently by typing

```
ls $FOAM_RUN
```

If a message is returned saying no such directory exists, the user should create the directory by typing

```
mkdir -p $FOAM_RUN
```

The tutorial cases describe the use of the meshing and pre-processing utilities, case setup and running OpenFOAM solvers and post-processing using ParaView.

Copies of all tutorials are available from the *tutorials* directory of the OpenFOAM installation. The tutorials are organised into a set of directories according to the type of flow and then subdirectories according to solver. For example, all the simpleFoam cases are stored within a subdirectory *incompressible/simpleFoam*, where *incompressible* indicates the type of flow. The user can copy cases from the *tutorials* directory into their local *run* directory as needed. For example to run the pitzDaily tutorial case for the simpleFoam solver, the user can copy it to the *run* directory by typing:

```
cd $FOAM_RUN
cp -r $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily .
```

2.1 Lid-driven cavity flow

This tutorial will describe how to pre-process, run and post-process a case involving isothermal, incompressible flow in a two-dimensional square domain. The geometry is

shown in Figure 2.1 in which all the boundaries of the square are walls. The top wall moves in the x-direction at a speed of 1 m/s while the other 3 are stationary. Initially, the flow will be assumed laminar and will be solved on a uniform mesh using the icoFoam solver for laminar, isothermal, incompressible flow. During the course of the tutorial, the effect of increased mesh resolution and mesh grading towards the walls will be investigated. Finally, the flow Reynolds number will be increased and the pisoFoam solver will be used for turbulent, isothermal, incompressible flow.

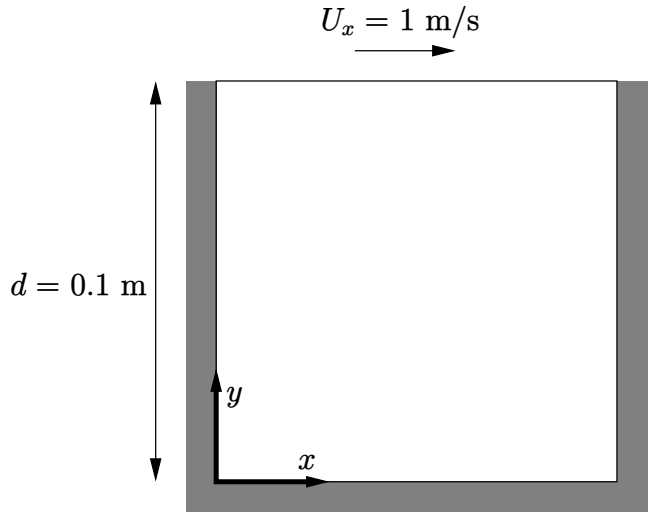


Figure 2.1: Geometry of the lid driven cavity.

2.1.1 Pre-processing

Cases are setup in OpenFOAM by editing case files. Users should select an editor of choice with which to do this, such as emacs, vi, gedit, nedit, *etc.* Editing files is possible in OpenFOAM because the I/O uses a dictionary format with keywords that convey sufficient meaning to be understood by the users.

A case being simulated involves data for mesh, fields, properties, control parameters, etc. As described in section 4.1, in OpenFOAM this data is stored in a set of files within a case directory rather than in a single case file, as in many other CFD packages. The case directory is given a suitably descriptive name. This tutorial consists of a set of cases located in *$FOAM_TUTORIALS/incompressible/icoFoam/cavity*, the first of which is simply named cavity. As a first step, the user should copy the cavity case directory to their *run* directory.

```
cd $FOAM_RUN
cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavity .
cd cavity
```

2.1.1.1 Mesh generation

OpenFOAM always operates in a 3 dimensional Cartesian coordinate system and all geometries are generated in 3 dimensions. OpenFOAM solves the case in 3 dimensions by default but can be instructed to solve in 2 dimensions by specifying a 'special' empty boundary condition on boundaries normal to the (3rd) dimension for which no solution is required.

The cavity domain consists of a square of side length $d = 0.1$ m in the x-y plane. A uniform mesh of 20 by 20 cells will be used initially. The block structure is shown in Figure 2.2. The mesh generator supplied with OpenFOAM, blockMesh, generates meshes

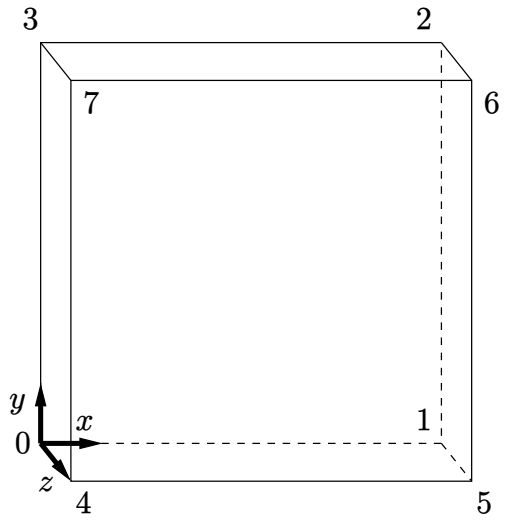


Figure 2.2: Block structure of the mesh for the cavity.

from a description specified in an input dictionary, *blockMeshDict* located in the *system* directory for a given case. The *blockMeshDict* entries for this case are as follows:

```
/*--------------------------------*- C++ -*----------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Version:  dev
     \\/     M anipulation  |
\*---------------------------------------------------------------------------*/
FoamFile
{
    format      ascii;
    class       dictionary;
    object      blockMeshDict;
}
// * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //

convertToMeters 0.1;

vertices
(
    (0 0 0)
    (1 0 0)
    (1 1 0)
    (0 1 0)
    (0 0 0.1)
    (1 0 0.1)
    (1 1 0.1)
    (0 1 0.1)
);

blocks
(
    hex (0 1 2 3 4 5 6 7) (20 20 1) simpleGrading (1 1 1)
);

boundary
(
    movingWall
    {
        type wall;
        faces
        (
            (3 7 6 2)
        );
    }
    fixedWalls
    {
```

```
47          type wall;
48          faces
49          (
50              (0 4 7 3)
51              (2 6 5 1)
52              (1 5 4 0)
53          );
54      }
55      frontAndBack
56      {
57          type empty;
58          faces
59          (
60              (0 3 2 1)
61              (4 5 6 7)
62          );
63      }
64  );
65
66
67  // ************************************************************************* //
```

The file first contains header information in the form of a banner (lines 1-7), then file information contained in a *FoamFile* sub-dictionary, delimited by curly braces (`{...}`).

For the remainder of the manual:
For the sake of clarity and to save space, file headers, including the banner and *FoamFile* sub-dictionary, will be removed from verbatim quoting of case files

The file first specifies coordinates of the block `vertices`; it then defines the `blocks` (here, only 1) from the vertex labels and the number of cells within it; and finally, it defines the boundary patches. The user is encouraged to consult section 5.3 to understand the meaning of the entries in the *blockMeshDict* file.

The mesh is generated by running blockMesh on this *blockMeshDict* file. From within the case directory, this is done, simply by typing in the terminal:

```
blockMesh
```

The running status of blockMesh is reported in the terminal window. Any mistakes in the *blockMeshDict* file are picked up by blockMesh and the resulting error message directs the user to the line in the file where the problem occurred.

2.1.1.2 Boundary and initial conditions

Once the mesh generation is complete, the user can look at this initial fields set up for this case. The case is set up to start at time $t = 0$ s, so the initial field data is stored in a *0* sub-directory of the *cavity* directory. The *0* sub-directory contains 2 files, *p* and *U*, one for each of the pressure (p) and velocity ($\mathbf{U}$) fields whose initial values and boundary conditions must be set. Let us examine file *p*:

```
16  dimensions      [0 2 -2 0 0 0 0];
17
18  internalField   uniform 0;
19
20  boundaryField
21  {
22      movingWall
23      {
24          type            zeroGradient;
25      }
26
27      fixedWalls
28      {
29          type            zeroGradient;
30      }
```

```
31
32      frontAndBack
33      {
34          type            empty;
35      }
36  }
37
38  // ************************************************************************* //
```

There are 3 principal entries in field data files:

`dimensions` specifies the dimensions of the field, here *kinematic* pressure, *i.e.* $m^2\,s^{-2}$ (see section 4.2.6 for more information);

`internalField` the internal field data which can be `uniform`, described by a single value; or `nonuniform`, where all the values of the field must be specified (see section 4.2.8 for more information);

`boundaryField` the boundary field data that includes boundary conditions and data for all the boundary patches (see section 4.2.8 for more information).

For this case cavity, the boundary consists of walls only, split into 2 patches named: (1) `fixedWalls` for the fixed sides and base of the cavity; (2) `movingWall` for the moving top of the cavity. As walls, both are given a zeroGradient boundary condition for `p`, meaning "the normal gradient of pressure is zero". The `frontAndBack` patch represents the front and back planes of the 2D case and therefore must be set as empty.

In this case, as in most we encounter, the initial fields are set to be uniform. Here the pressure is kinematic, and as an incompressible case, its absolute value is not relevant, so is set to `uniform 0` for convenience.

The user can similarly examine the velocity field in the *0/U* file. The `dimensions` are those expected for velocity, the internal field is initialised as uniform zero, which in the case of velocity must be expressed by 3 vector components, *i.e.* `uniform (0 0 0)` (see section 4.2.5 for more information).

The boundary field for velocity requires the same boundary condition for the `frontAndBack` patch. The other patches are walls: a no-slip condition is assumed on the `fixedWalls`, hence a noSlip condition. The top surface moves at a speed of 1 m/s in the x-direction so requires a fixedValue condition with `value` of `uniform (1 0 0)`.

2.1.1.3 Physical properties

The physical properties for the case are stored in dictionary files in the *constant* directory. For an icoFoam case, the only property that must be specified is the kinematic viscosity which is stored from the *physicalProperties* dictionary. The user can check that the kinematic viscosity is set correctly by opening the *physicalProperties* dictionary to view/edit its entries. The keyword for kinematic viscosity is `nu`, the phonetic label for the Greek symbol ν by which it is represented in equations. Initially this case will be run with a Reynolds number of 10, where the Reynolds number is defined as:

$$Re = \frac{d|\mathbf{U}|}{\nu} \tag{2.1}$$

where d and $|\mathbf{U}|$ are the characteristic length and velocity respectively and ν is the kinematic viscosity. Here $d = 0.1$ m, $|\mathbf{U}| = 1$ m/s, so that for $Re = 10$, $\nu = 0.01\ m^2\,s^{-1}$. The correct file entry for kinematic viscosity is thus specified below:

```
16
17  nu              [0 2 -1 0 0 0 0] 0.01;
18
19
20  // ************************************************************************* //
```

2.1.1.4 Control

Input data relating to the control of time and reading and writing of the solution data are read in from the *controlDict* dictionary. The user should view this file; as a case control file, it is located in the *system* directory.

The start/stop times and the time step for the run must be set. OpenFOAM offers great flexibility with time control which is described in full in section 4.4. In this tutorial we wish to start the run at time $t = 0$ which means that OpenFOAM needs to read field data from a directory named *0* — see section 4.1 for more information of the case file structure. Therefore we set the `startFrom` keyword to `startTime` and then specify the `startTime` keyword to be 0.

For the end time, we wish to reach the steady state solution where the flow is circulating around the cavity. As a general rule, the fluid should pass through the domain 10 times to reach steady state in laminar flow. In this case the flow does not pass through this domain as there is no inlet or outlet, so instead the end time can be set to the time taken for the lid to travel ten times across the cavity, *i.e.* 1 s; in fact, with hindsight, we discover that 0.5 s is sufficient so we shall adopt this value. To specify this end time, we must specify the `stopAt` keyword as `endTime` and then set the `endTime` keyword to 0.5.

Now we need to set the time step, represented by the keyword `deltaT`. To achieve temporal accuracy and numerical stability when running icoFoam, a Courant number of less than 1 is required. The Courant number is defined for one cell as:

$$Co = \frac{\delta t |\mathbf{U}|}{\delta x} \tag{2.2}$$

where δt is the time step, $|\mathbf{U}|$ is the magnitude of the velocity through that cell and δx is the cell size in the direction of the velocity. The flow velocity varies across the domain and we must ensure $Co < 1$ everywhere. We therefore choose δt based on the worst case: the *maximum* Co corresponding to the combined effect of a large flow velocity and small cell size. Here, the cell size is fixed across the domain so the maximum Co will occur next to the lid where the velocity approaches 1 $\mathrm{m\,s^{-1}}$. The cell size is:

$$\delta x = \frac{d}{n} = \frac{0.1}{20} = 0.005 \text{ m} \tag{2.3}$$

Therefore to achieve a Courant number less than or equal to 1 throughout the domain the time step `deltaT` must be set to less than or equal to:

$$\delta t = \frac{Co\ \delta x}{|\mathbf{U}|} = \frac{1 \times 0.005}{1} = 0.005 \text{ s} \tag{2.4}$$

As the simulation progresses we wish to write results at certain intervals of time that we can later view with a post-processing package. The `writeControl` keyword presents several options for setting the time at which the results are written; here we select the `timeStep` option which specifies that results are written every nth time step where the value n is specified under the `writeInterval` keyword. Let us decide that we wish to write our results at times 0.1, 0.2,..., 0.5 s. With a time step of 0.005 s, we therefore need to output results at every 20th time time step and so we set `writeInterval` to 20.

OpenFOAM creates a new directory *named after the current time*, *e.g.* 0.1 s, on each occasion that it writes a set of data, as discussed in full in section 4.1. In the icoFoam solver, it writes out the results for each field, `U` and `p`, into the time directories. For this case, the entries in the *controlDict* are shown below:

```
16
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  // ************************************************************************* //
```

2.1.1.5 Discretisation and linear-solver settings

The user specifies the choice of finite volume discretisation schemes in the *fvSchemes* dictionary in the *system* directory. The specification of the linear equation solvers and tolerances and other algorithm controls is made in the *fvSolution* dictionary, similarly in the *system* directory. The user is free to view these dictionaries but we do not need to discuss all their entries at this stage except for `pRefCell` and `pRefValue` in the *PISO* sub-dictionary of the *fvSolution* dictionary. In a closed incompressible system such as the cavity, pressure is relative: it is the pressure range that matters not the absolute values. In cases such as this, the solver sets a reference level by `pRefValue` in cell `pRefCell`. In this example both are set to 0. Changing either of these values will change the absolute pressure field, but not, of course, the relative pressures or velocity field.

2.1.2 Viewing the mesh

Before the case is run it is a good idea to view the mesh to check for any errors. The mesh is viewed in ParaView, the post-processing tool supplied with OpenFOAM. The ParaView post-processing is conveniently launched on OpenFOAM case data by executing the paraFoam script from within the case directory.

Any UNIX/Linux executable can be run in two ways: as a foreground process, *i.e.* one in which the shell waits until the command has finished before giving a command prompt; as a background process, which allows the shell to accept additional commands while it is still running. Since it is convenient to keep ParaView open while running other commands from the terminal, we will launch it in the background using the `&` operator by typing

```
paraFoam &
```

Alternatively, it can be launched from another directory location with an optional `-case` argument giving the case directory, *e.g.*

```
paraFoam -case $FOAM_RUN/cavity &
```

This launches the ParaView window as shown in Figure 6.1. In the Pipeline Browser, the user can see that ParaView has opened `cavity.OpenFOAM`, the module for the cavity case. **Before clicking the** Apply **button**, the user needs to select some geometry from the Mesh Parts panel. Because the case is small, it is easiest to select all the data by checking the box adjacent to the Mesh Parts panel title, which automatically checks all individual components within the respective panel. The user should then click the Apply button to load the geometry into ParaView.

The user should then scroll down to the Display panel that controls the visual representation of the selected module. Within the Display panel the user should do the following as shown in Figure 2.3:

1. in the `Coloring` section, select `Solid Color`;

2. click Edit (in `Coloring`) and select an appropriate colour *e.g.* black (for a white background);

3. select `Wireframe` from the `Representation` menu. The background colour can be set in the View (Render View) panel at the bottom of the Properties window.

Especially the first time the user starts ParaView, **it is recommended** that they manipulate the view as described in section 6.1.5. In particular, since this is a 2D case, it is recommended that Camera Parallel Projection is selected at the bottom of the View (Render View) panel.

Furthermore, **many parameters in the** Properties **window are only available by clicking the** Advanced Properties **gearwheel button at the top of the** Properties **window, next to the search box.**

2.1.3 Running an application

Like any UNIX/Linux executable, OpenFOAM applications can be run either in the foreground or background. On this occasion, we will run icoFoam in the foreground. The icoFoam solver is executed either by entering the case directory and typing

```
icoFoam
```

at the command prompt, or with the optional `-case` argument giving the case directory, *e.g.*

```
icoFoam -case $FOAM_RUN/cavity
```

The progress of the job is written to the terminal window. It tells the user the current time, maximum Courant number, initial and final residuals for all fields.

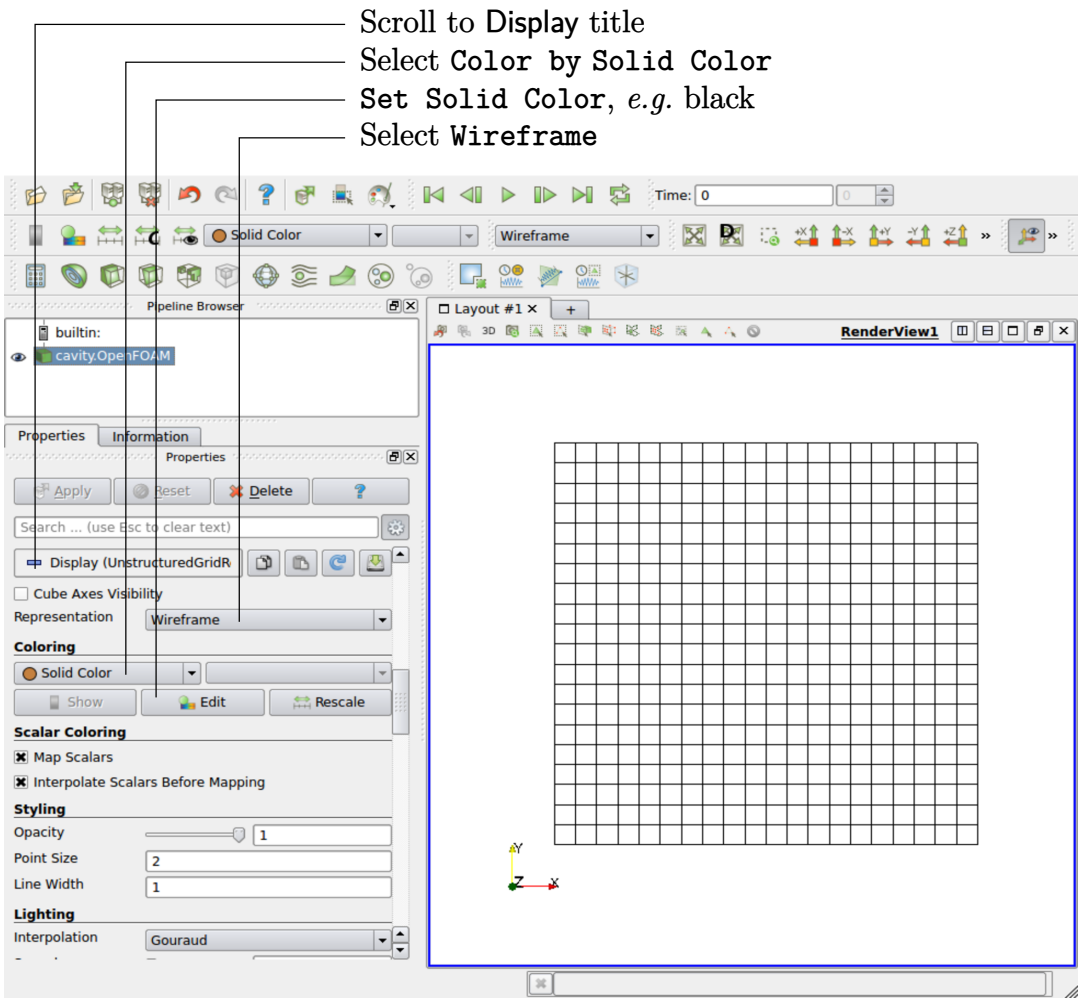


Figure 2.3: Viewing the mesh in paraFoam.

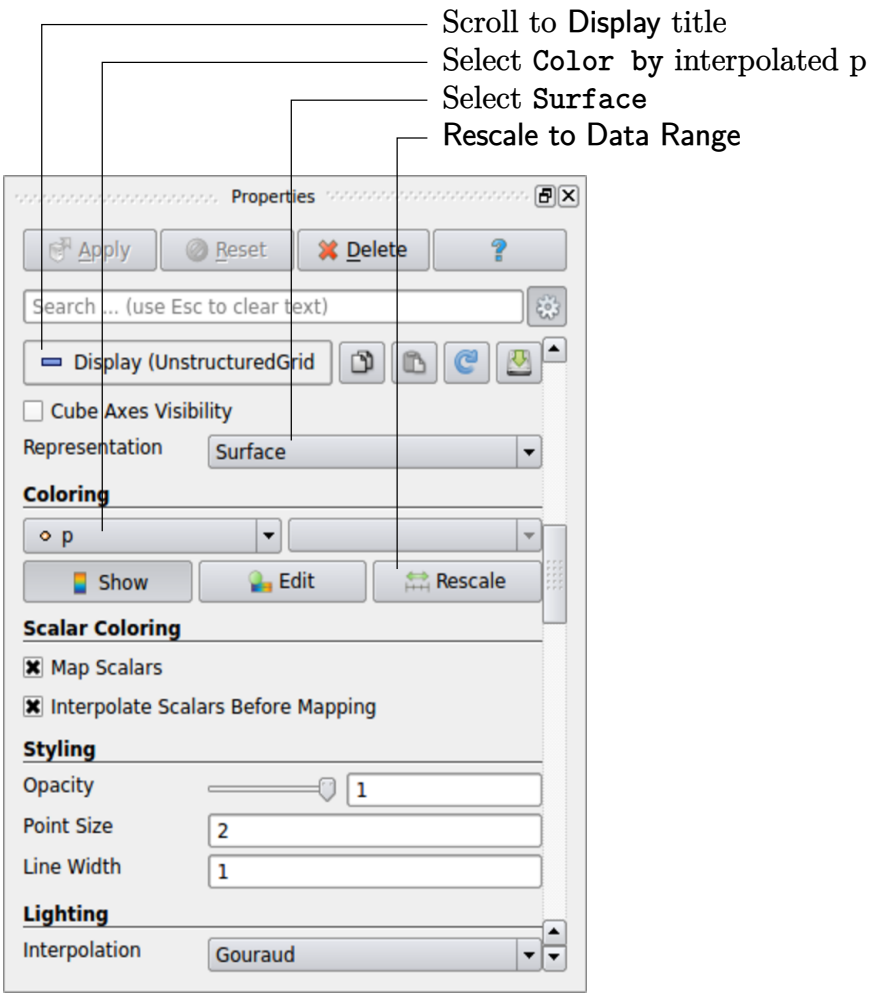


Figure 2.4: Displaying pressure contours for the cavity case.

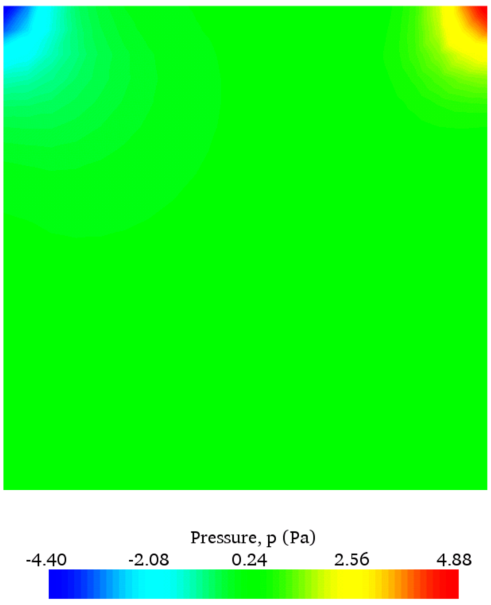


Figure 2.5: Pressures in the cavity case.

2.1.4 Post-processing

As soon as results are written to time directories, they can be viewed using paraFoam. Return to the paraFoam window and select the Properties panel for the `cavity.OpenFOAM` case module. If the correct window panels for the case module do not seem to be present at any time, please ensure that: `cavity.OpenFOAM` is highlighted in blue; eye button alongside it is switched on to show the graphics are enabled;

To prepare paraFoam to display the data of interest, we must first load the data at the required run time of 0.5 s. If the case was run while ParaView was open, the output data in time directories will not be automatically loaded within ParaView. To load the data the user should uncheck Cache Mesh and click Refresh Times at the **top** Properties window (scroll up the panel if necessary). When The time data will be loaded into ParaView.

In order to view the solution at $t = 0.5$ s, the user can use the `VCR Controls` or `Current Time Controls` to change the current time to 0.5. These are located in the toolbars at the top of the ParaView window, as shown in Figure 6.4.

2.1.4.1 Colouring surfaces

To view pressure, the user should go to the Display panel since it controls the visual representation of the selected module. To make a simple plot of pressure, the user should select the following, as described in detail in Figure 2.4:

1. select `Surface` from the `Representation` menu;
2. select ◦ p in `Coloring`
3. click the Rescale button to set the colour scale to the data range, if necessary.

The pressure field should appear as shown in Figure 2.5, with a region of low pressure at the top left of the cavity and one of high pressure at the top right of the cavity.

With the point icon (◦ p) the pressure field is interpolated across each cell to give a continuous appearance. Instead if the user selects the cell icon, p, from the `Coloring` menu, a single value for pressure will be attributed to each cell so that each cell will be denoted by a single colour with no grading.

A colour legend can be added by either by clicking the Toggle Color Legend Visibility button in the `Active Variable Controls` toolbar or the Show button in the Coloring section of the Display panel. The legend can be located in the image window by drag and drop with the mouse. The Edit button, either in the `Active Variable Controls` toolbar or in the Coloring panel of the Display panel, opens the Color Map Editor window, as shown in Figure 2.6, where the user can set a range of attributes of the colour scale and the color bar.

In particular, ParaView defaults to using a colour scale of blue to white to red rather than the more common blue to green to red (rainbow). Therefore *the first time* that the user executes ParaView, they may wish to change the colour scale. This can be done by selecting the Choose Preset button (with the heart icon) in the Color Scale Editor. The conventional color scale for CFD is Blue to Red Rainbow which is only listed if the user types the name in the Search bar or checks the gearwheel to the right of that bar.

After selecting Blue to Red Rainbow and clicking Apply and Close, the user can click the Save as Default button at the absolute bottom of the panel (file save symbol) so that ParaView will always adopt this type of colour bar.

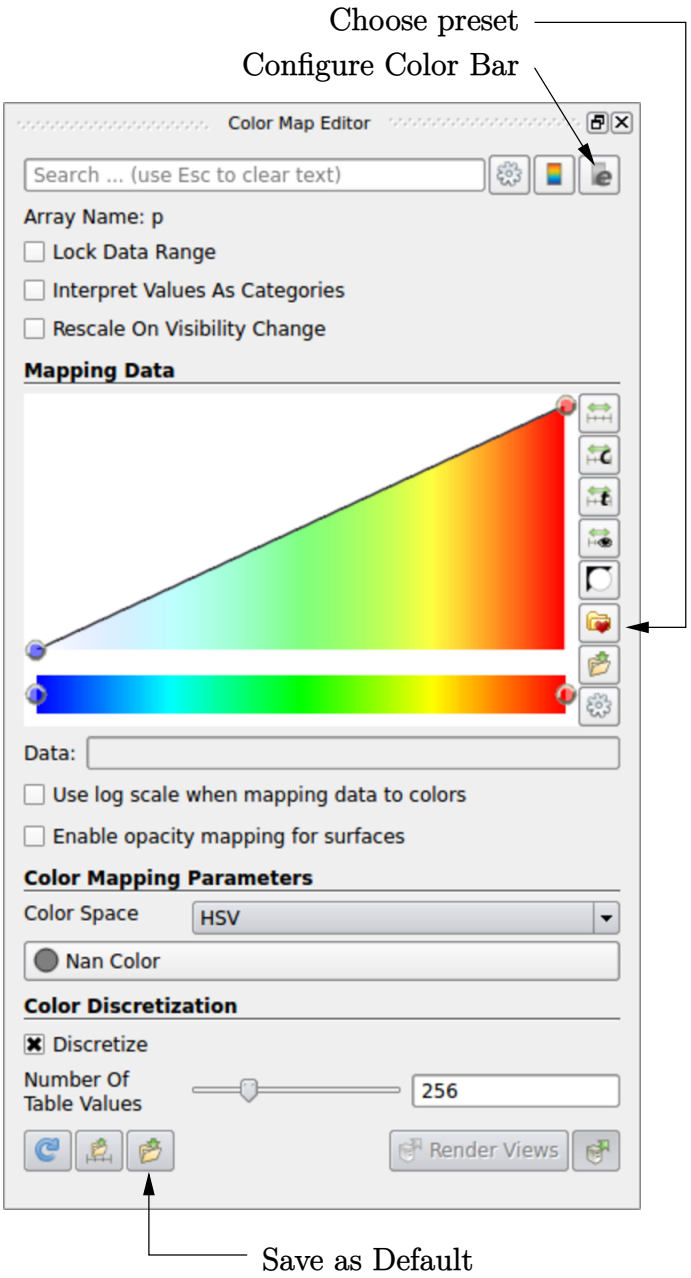


Figure 2.6: Color Map Editor.

The user can also edit the color legend properties, such as text size, font selection and numbering format for the scale, by clicking the Edit Color Legend Properties to the far right of the search bar, as shown in Figure 2.6.

2.1.4.2 Cutting plane (slice)

If the user rotates the image, by holding down the left mouse button in the image window and moving the cursor, they can see that they have now coloured the complete geometry surface by the pressure. In order to produce a genuine 2-dimensional contour plot the user should first create a cutting plane, or 'slice'. With the `cavity.OpenFOAM` module highlighted in the Pipeline Browser, the user should select the `Slice` filter from the `Filters` menu in the top menu of ParaView (accessible at the top of the screen on some systems). The `Slice` filter can be initially found in the `Common` sub-menu, but once selected, it **moves** to the `Recent` sub-menu, **disappearing** from the `Common` sub-menu. The cutting

plane should be centred at $(0.05, 0.05, 0.005)$ and its normal should be set to $(0, 0, 1)$ (click the Z Normal button).

2.1.4.3 Contours

Having generated the cutting plane, contours can be created using by applying the `Contour` filter. With the `Slice` module highlighted in the Pipeline Browser, the user should select the `Contour` filter. In the Properties panel, the user should select pressure from the `Contour By` menu. Under `Isosurfaces`, the user could delete the default value with the minus button, then add a range of 10 values. The contours can be displayed with a `Wireframe` representation if the `Coloring` is solid or by a field, *e.g.* pressure.

2.1.4.4 Vector plots

Before we start to plot the vectors of the flow velocity, it may be useful to remove other modules that have been created, *e.g.* using the `Slice` and `Contour` filters described above. These can: either be deleted entirely, by highlighting the relevant module in the Pipeline Browser and clicking Delete in their respective Properties panel; or, be disabled by toggling the eye button for the relevant module in the Pipeline Browser.

We now wish to generate a vector glyph for velocity at the centre of each cell. We first need to filter the data to cell centres as described in section 6.1.7.1. With the `cavity.OpenFOAM` module highlighted in the Pipeline Browser, the user should select `Cell Centers` from the `Filter->Alphabetical` menu and then click Apply.

With these `Centers` highlighted in the Pipeline Browser, the user should then select `Glyph` from the `Filter->Common` menu. The Properties window panel should appear as shown in Figure 2.7. This outdated image omits `Orientation Array` which must be set to `U`.

We suggest popular filters are added to the `Filters->Favourites` menu. In the resulting Properties panel, the velocity field, `U`, must be selected from the `vectors` menu. The user should set the `Scale Array` for the glyphs to be `No scale array`, with Set Scale Factor set to 0.005. Under `Glyph Mode`, the user should select All Points.

On clicking Apply, the glyphs appear but, probably as a single colour, *e.g.* white. The user should colour the glyphs by velocity magnitude which, as usual, is controlled by setting `Color by U` in the Display panel. The user can also select Show Color Legend in `Edit Color Map`. The output is shown in Figure 2.8, in which uppercase Times Roman fonts are selected for the Color Legend headings and the labels are specified to 2 fixed significant figures by deselecting Automatic Label Format and entering `%-#6.2f` in the Label Format text box. The background colour is set to white in the General panel of `View Settings` as described in section 6.1.5.1.

Note that at the left and right walls, glyphs appear to indicate flow through the walls. However, it is clear that, while the flow direction is normal to the wall, its magnitude is 0. This slightly confusing situation is caused by ParaView choosing to orientate the glyphs in the x-direction when the glyph scaling `off` and the velocity magnitude is 0.

2.1.4.5 Streamline plots

Again, before the user continues to post-process in ParaView, they should disable modules such as those for the vector plot described above. We now wish to plot streamlines of velocity as described in section 6.1.8. With the `cavity.OpenFOAM` module highlighted in the Pipeline Browser, the user should then select `Stream Tracer` from the `Filter` menu

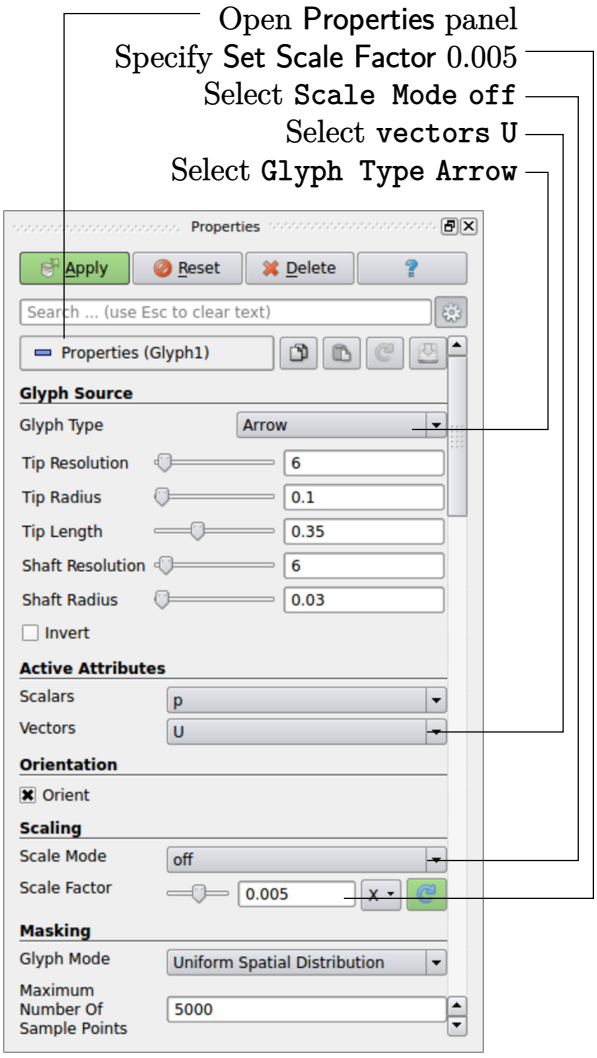


Figure 2.7: Properties panel for the `Glyph` filter.

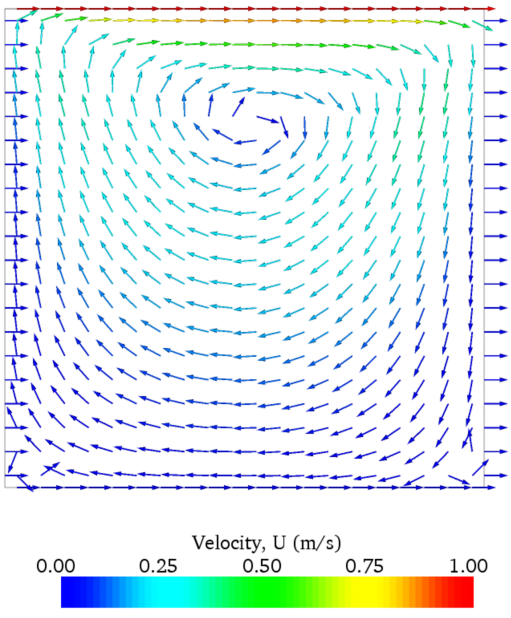


Figure 2.8: Velocities in the cavity case.

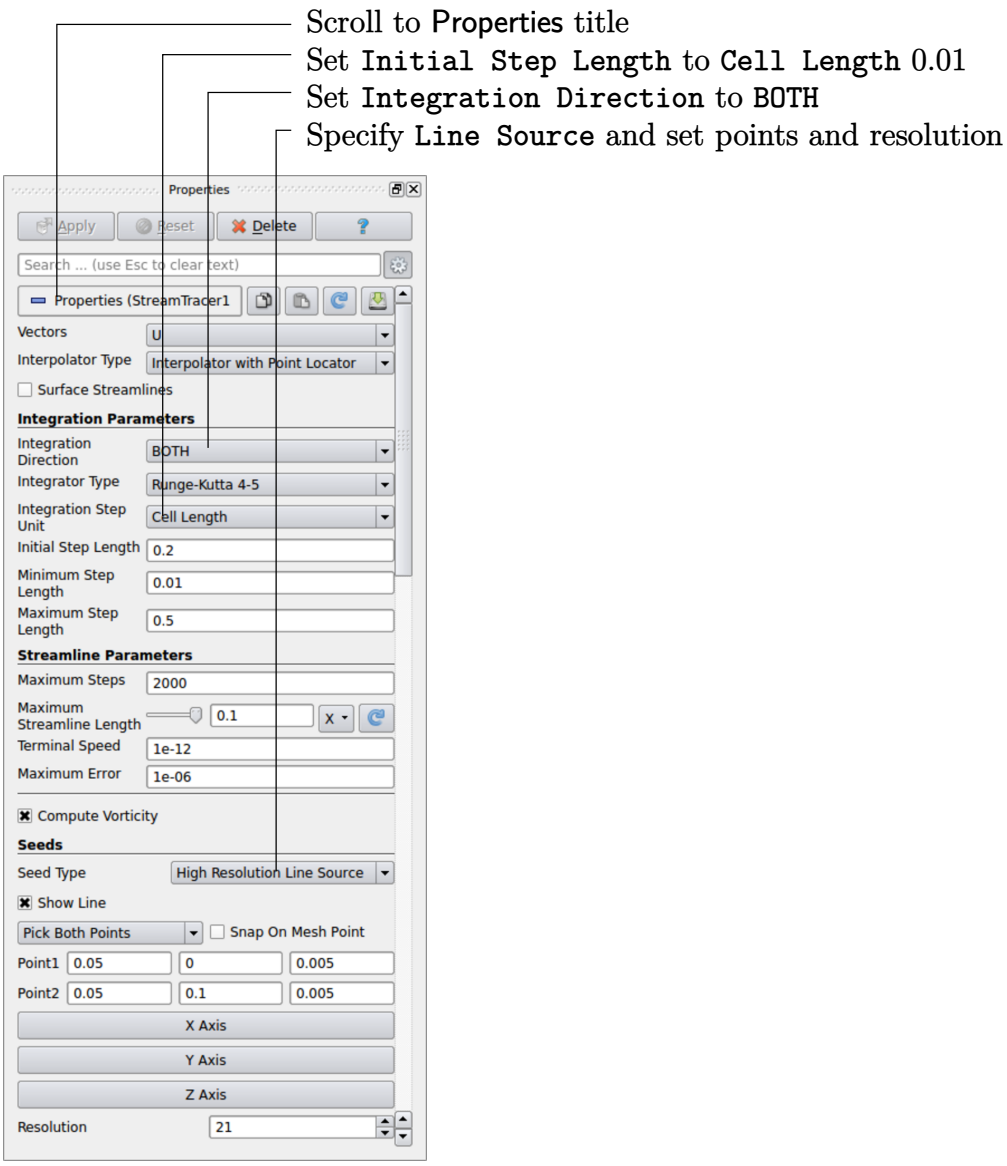


Figure 2.9: Properties panel for the `Stream Tracer` filter.

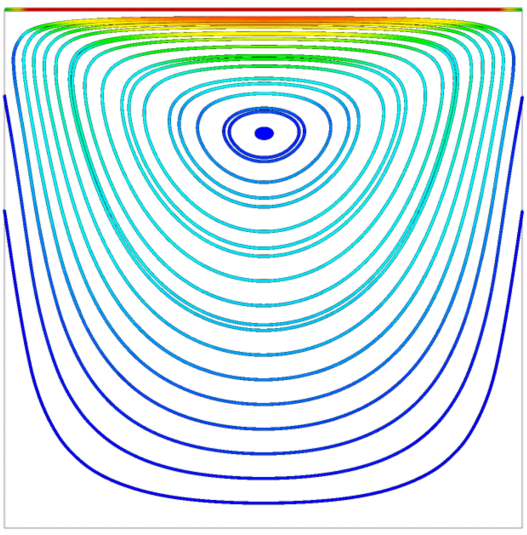


Figure 2.10: Streamlines in the cavity case.

and then click Apply. The Properties window panel should appear as shown in Figure 2.9. The Seed points should be specified along a `High Resolution Line Source` running vertically through the centre of the geometry, *i.e.* from $(0.05, 0, 0.005)$ to $(0.05, 0.1, 0.005)$. For the image in this guide we used: a point Resolution of 21; Maximum Step Length of 0.5; Initial Step Length of 0.2; and, `Integration Direction BOTH`. The `Runge-Kutta 4/5` IntegratorType was used with default parameters.

On clicking Apply the tracer is generated. The user should then select `Tube` from the `Filter` menu to produce high quality streamline images. For the image in this report, we used: Num. sides 6; Radius 0.0003; and, Radius factor 10. The streamtubes are coloured by velocity magnitude. On clicking Apply the image in Figure 2.10 should be produced.

2.1.5 Increasing the mesh resolution

The mesh resolution will now be increased by a factor of two in each direction. The results from the coarser mesh will be mapped onto the finer mesh to use as initial conditions for the problem. The solution from the finer mesh will then be compared with those from the coarser mesh.

2.1.5.1 Creating a new case using an existing case

We now wish to create a new case named cavityFine that is created from cavity. The user should therefore clone the cavity case and edit the necessary files. First the user should go to the *run* directory, by typing

```
cd $FOAM_RUN
```

Note that there is also a convenient alias, named `run`, that reproduces the command above to change directory to *$FOAM_RUN*, simply by typing `run`.

The cavityFine case can be created by making a new case directory and copying the relevant directories from the cavity case.

```
mkdir cavityFine
cp -r cavity/constant cavityFine
cp -r cavity/system cavityFine
```

The user can then prepare to run the new case by changing into the case directory.

```
cd cavityFine
```

2.1.5.2 Creating the finer mesh

We now wish to increase the number of cells in the mesh by using blockMesh. The user should open the *blockMeshDict* file in the *system* directory in an editor and edit the block specification. The blocks are specified in a list under the `blocks` keyword. The syntax of the block definitions is described fully in section 5.3.1.3; at this stage it is sufficient to know that following `hex` is first the list of vertices in the block, then a list (or vector) of numbers of cells in each direction. This was originally set to `(20 20 1)` for the cavity case. The user should now change this to `(40 40 1)` and save the file. The new refined mesh should then be created by running blockMesh as before.

2.1.5.3 Mapping the coarse mesh results onto the fine mesh

The mapFields utility maps one or more fields relating to a given geometry onto the corresponding fields for another geometry. In our example, the fields are deemed 'consistent' because the geometry and the boundary types, or conditions, of both source and target fields are identical. We use the `-consistent` command line option when executing mapFields in this example.

The field data that mapFields maps is read from the time directory specified by `startFrom` and `startTime` in the *controlDict* of the target case, *i.e.* those **into which** the results are being mapped. In this example, we wish to map the final results of the coarser mesh from case cavity onto the finer mesh of case cavityFine. Therefore, since these results are stored in the *0.5* directory of cavity, the `startTime` should be set to 0.5 s in the *controlDict* dictionary and `startFrom` should be set to `startTime`.

The case is ready to run mapFields. Typing `mapFields -help` quickly shows that mapFields requires the source case directory as an argument. We are using the `-consistent` option, so the utility is executed from within the *cavityFine* directory by

```
mapFields -consistent ../cavity
```

The utility should run with output to the terminal including:

```
Source: ".." "cavity"
Target: "." "cavityFine"

Create databases as time

Source time: 0.5
Target time: 0.5

Create meshes

Source mesh size: 400   Target mesh size: 1600

Consistently creating and mapping fields for time 0.5

    interpolating p
    interpolating U

End
```

2.1.5.4 Control adjustments

To maintain a Courant number of less that 1, as discussed in section 2.1.1.4, the time step must now be halved since the size of all cells has halved. Therefore `deltaT` should be set to 0.0025 s in the *controlDict* dictionary. Field data is currently written out at an interval of a fixed number of time steps. Here we demonstrate how to specify data output at fixed intervals of time. Under the `writeControl` keyword in *controlDict*, instead of requesting output by a fixed number of time steps with the `timeStep` entry, a fixed amount of run time can be specified between the writing of results using the `runTime` entry. In this case the user should specify output every 0.1 and therefore should set `writeInterval` to `0.1` and `writeControl` to `runTime`. Finally, since the case is starting with a solution obtained on the coarse mesh we only need to run it for a short period to achieve reasonable convergence to steady-state. Therefore the `endTime` should be set to 0.7 s. Make sure these settings are correct and then save the file.

2.1.5.5 Running the code as a background process

The user should experience running icoFoam as a background process, redirecting the terminal output to a *log* file that can be viewed later. From the *cavityFine* directory, the user should execute:

```
icoFoam > log &
cat log
```

2.1.5.6 Vector plot with the refined mesh

The user can open multiple cases simultaneously in ParaView; essentially because each new case is simply another module that appears in the Pipeline Browser. There is an inconvenience when opening a new OpenFOAM case in ParaView because it expects that case data is stored in a single file which has a file extension that enables it to establish the format. However, OpenFOAM stores case data in multiple files without an extension in the name, within a specific directory structure. The ParaView reader module works on the basis that, when opening case data in OpenFOAM format, it is passed a dummy (empty) file with the *.OpenFOAM* extension that resides in the case directory. The paraFoam script automatically creates this file — hence, the cavity case module is called `cavity.OpenFOAM`.

If the user wishes to open a second case directly from within ParaView, they need to create such a dummy file. They can do this 'by hand' or, more simply, use the paraFoam script with the option `-touch`. For the cavityFine example, that involves executing from the case directory:

```
paraFoam -touch
```

Now the cavityFine case can be loaded into ParaView by selecting `Open` from the `File` menu, and having navigated to the *cavityFine* directory, opening `cavityFine.OpenFOAM`. The user can now make a vector plot of the results from the refined mesh in ParaView. The plot can be compared with the cavity case by enabling glyph images for both case simultaneously.

2.1.5.7 Plotting graphs

The user may wish to visualise the results by extracting some scalar measure of velocity and plotting 2-dimensional graphs along lines through the domain. OpenFOAM is well equipped for this kind of data manipulation. There are numerous utilities that do specialised data manipulations, and the postProcess utility that includes a broad range of generic post-processing functionality. The functions within postProcess can be listed by typing:

```
postProcess -list
```

The `components` and `mag` functions provide useful scalar measures of velocity. When the `components` function is executed on a case, *e.g. cavity*, it reads in the velocity vector field from each time directory and, in the corresponding time directories, writes scalar fields `Ux`, `Uy` and `Uz` representing the x, y and z components of velocity.

The user can run postProcess with the `components` function on both cavity and cavityFine cases. For example, for the cavity case the user should go into the *cavity* directory and execute postProcess as follows:

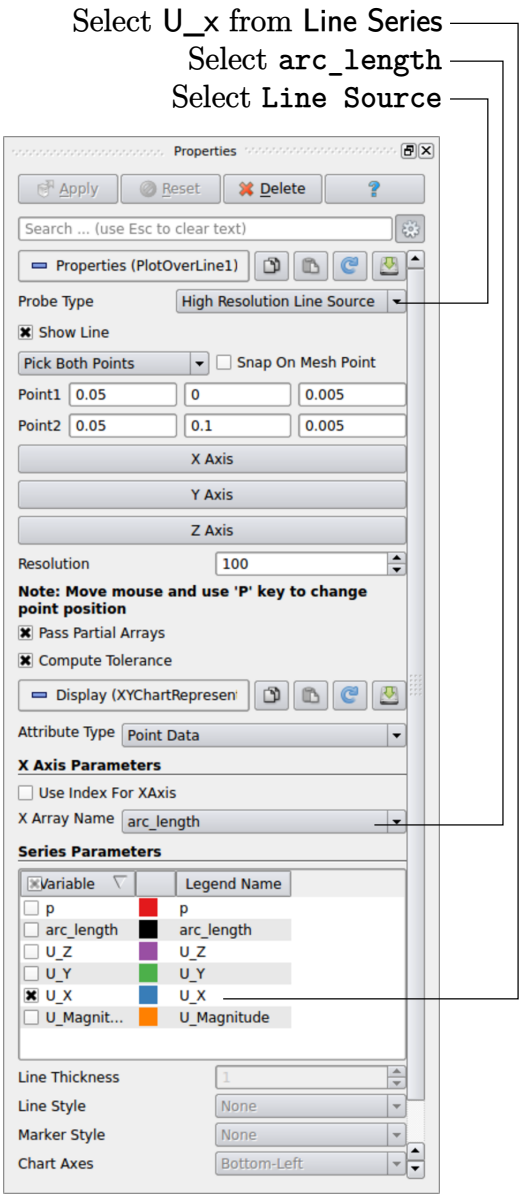


Figure 2.11: Selecting fields for graph plotting.

```
cd $FOAM_RUN/cavity
postProcess -func "components(U)"
```

The individual components can be plotted as a graph in ParaView. It is quick, convenient and has reasonably good control over labelling and formatting, so the printed output is a fairly good standard. However, to produce graphs for publication, users may prefer to write raw data and plot it with a dedicated graphing tool, such as gnuplot. To do this, we recommend using the sampling tools, described in section 6.3.2 and section 2.2.3.

Before commencing plotting, the user needs to load the newly generated `Ux`, `Uy` and `Uz` fields into ParaView. To do this, the user should click the Refresh Times at the top of the Properties panel for the `cavity.OpenFOAM` module which will cause the new fields to be loaded into ParaView and appear in the Volume Fields window. Ensure the new fields are selected and the changes are applied, *i.e.* click Apply again if necessary. *Also*, data is interpolated incorrectly at boundaries if the boundary regions are selected in the Mesh Parts panel. Therefore the user should *deselect the patches* in the Mesh Parts panel, *i.e.*

`movingWall`, `fixedWall` and `frontAndBack`, and apply the changes.

Now, in order to display a graph in ParaView the user should select the module of interest, *e.g.*`cavity.OpenFOAM` and apply the `Plot Over Line` filter from the `Filter->Data Analysis` menu. This opens up a new XY Plot window below or beside the existing 3D View window. A `PlotOverLine` module is created in which the user can specify the end points of the line in the Properties panel. In this example, the user should position the line vertically up the centre of the domain, *i.e.* from $(0.05, 0, 0.005)$ to $(0.05, 0.1, 0.005)$, in the Point1 and Point2 text boxes. The Resolution can be set to 100.

On clicking Apply, a graph is generated in the XY Plot window. In the Display panel, the user should set `Attribute Type` to `Point Data`. The `X Axis Parameters` can be set to use the `arc_length` for the the X Array Name, which displays distance from the base of the cavity on the horizontal axis of the graph.

The user can choose the fields to be displayed in the Series Parameters panel of the Display window. From the list of scalar fields to be displayed, it can be seen that the magnitude and components of vector fields are available by default, *e.g.* displayed as `U_X`, so that it was not necessary to create `Ux` using the `components` function. Nevertheless, the user should deselect all series except `Ux` (or `U_x`). A square colour box in the adjacent column to the selected series indicates the line colour. The user can edit this most easily by a double click of the mouse over that selection.

In order to format the graph, the user should modify the settings below the Line Series panel, namely `Line Color`, `Line Thickness`, `Line Style`, `Marker Style` and `Chart Axes`.

Below these parameters in the Properties window, the user can control graph annotations, *e.g.* the legend for each axis. Also, the user can set font, colour and alignment of the axes titles, and has several options for axis range and labels in linear or logarithmic scales.

Figure 2.12 is a graph produced using ParaView. The user can produce a graph however he/she wishes. For information, the graph in Figure 2.12 was produced with the options for axes of: `Standard` type of `Notation`; Specify Axis Range selected; titles in `Sans Serif 12` font. The graph is displayed as a set of points rather than a line by activating the Enable Line Series button in the Display window. Note: if this button appears to be inactive by being "greyed out", it can be made active by selecting and deselecting the sets of variables in the Line Series panel. Once the Enable Line Series button is selected, the `Line Style` and `Marker Style` can be adjusted to the user's preference.

2.1.6 Introducing mesh grading

The error in any solution will be more pronounced in regions where the form of the true solution differ widely from the form assumed in the chosen numerical schemes. For example a numerical scheme based on linear variations of variables over cells can only generate an exact solution if the true solution is itself linear in form. The error is largest in regions where the true solution deviates greatest from linear form, *i.e.* where the change in gradient is largest. Error decreases with cell size.

It is useful to have an intuitive appreciation of the form of the solution before setting up any problem. It is then possible to anticipate where the errors will be largest and to grade the mesh so that the smallest cells are in these regions. In the cavity case the large variations in velocity can be expected near a wall and so in this part of the tutorial the mesh will be graded to be smaller in this region. By using the same number of cells, greater accuracy can be achieved without a significant increase in computational cost.

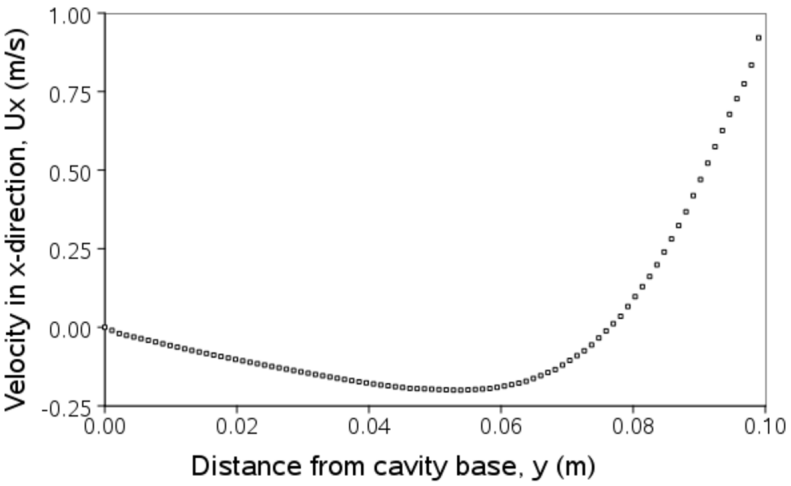


Figure 2.12: Plotting graphs in paraFoam.

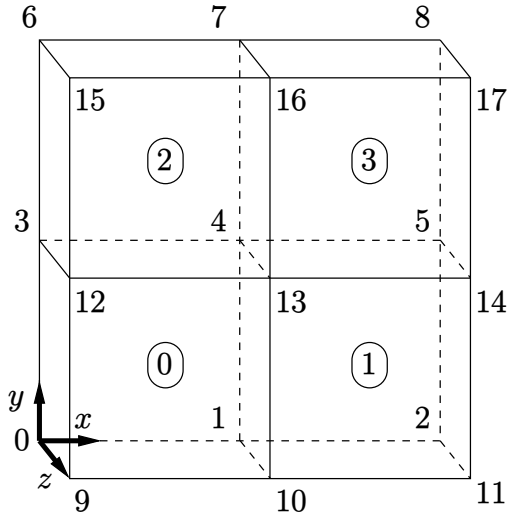


Figure 2.13: Block structure of the graded mesh for the cavity (block numbers encircled).

A mesh of 20×20 cells with grading towards the walls will be created for the lid-driven cavity problem and the results from the finer mesh of section 2.1.5.2 will then be mapped onto the graded mesh to use as an initial condition. The results from the graded mesh will be compared with those from the previous meshes. Since the changes to the *blockMeshDict* dictionary are fairly substantial, the case used for this part of the tutorial, cavityGrade, is supplied in the *$FOAM_TUTORIALS/incompressible/icoFoam/cavity* directory. The user should copy the cavityGrade case into the *run* directory, then follow the steps below.

2.1.6.1 Creating the graded mesh

The mesh now needs 4 blocks as different mesh grading is needed on the left and right and top and bottom of the domain. The block structure for this mesh is shown in Figure 2.13. The user can view the *blockMeshDict* file in the *system* subdirectory of cavityGrade; for completeness the key elements of the *blockMeshDict* file are also reproduced below. Each block now has 10 cells in the x and y directions and the ratio between largest and smallest cells is 2.

```
16  convertToMeters 0.1;
17
18  vertices
19  (
```

```
20      (0 0 0)
21      (0.5 0 0)
22      (1 0 0)
23      (0 0.5 0)
24      (0.5 0.5 0)
25      (1 0.5 0)
26      (0 1 0)
27      (0.5 1 0)
28      (1 1 0)
29      (0 0 0.1)
30      (0.5 0 0.1)
31      (1 0 0.1)
32      (0 0.5 0.1)
33      (0.5 0.5 0.1)
34      (1 0.5 0.1)
35      (0 1 0.1)
36      (0.5 1 0.1)
37      (1 1 0.1)
38  );
39
40  blocks
41  (
42      hex (0 1 4 3 9 10 13 12) (10 10 1) simpleGrading (2 2 1)
43      hex (1 2 5 4 10 11 14 13) (10 10 1) simpleGrading (0.5 2 1)
44      hex (3 4 7 6 12 13 16 15) (10 10 1) simpleGrading (2 0.5 1)
45      hex (4 5 8 7 13 14 17 16) (10 10 1) simpleGrading (0.5 0.5 1)
46  );
47
48  boundary
49  (
50      movingWall
51      {
52          type wall;
53          faces
54          (
55              (6 15 16 7)
56              (7 16 17 8)
57          );
58      }
59      fixedWalls
60      {
61          type wall;
62          faces
63          (
64              (3 12 15 6)
65              (0 9 12 3)
66              (0 1 10 9)
67              (1 2 11 10)
68              (2 5 14 11)
69              (5 8 17 14)
70          );
71      }
72      frontAndBack
73      {
74          type empty;
75          faces
76          (
77              (0 3 4 1)
78              (1 4 5 2)
79              (3 6 7 4)
80              (4 7 8 5)
81              (9 10 13 12)
82              (10 11 14 13)
83              (12 13 16 15)
84              (13 14 17 16)
85          );
86      }
87  );
88
89
90  // ************************************************************************* //
```

Once familiar with the *blockMeshDict* file for this case, the user can execute blockMesh from the command line. The graded mesh can be viewed as before using paraFoam as described in section 2.1.2.

2.1.6.2 Changing time and time step

The highest velocities and smallest cells are next to the lid, therefore the highest Courant number will be generated next to the lid, for reasons given in section 2.1.1.4. It is therefore useful to estimate the size of the cells next to the lid to calculate an appropriate time step for this case.

When a nonuniform mesh grading is used, blockMesh calculates the cell sizes using a geometric progression. Along a length l, if n cells are requested with a ratio of R between the last and first cells, the size of the smallest cell, δx_s, is given by:

$$\delta x_s = l\frac{r-1}{\alpha r - 1} \tag{2.5}$$

where r is the ratio between one cell size and the next which is given by:

$$r = R^{\frac{1}{n-1}} \tag{2.6}$$

and

$$\alpha = \begin{cases} R & \text{for } R > 1, \\ 1 - r^{-n} + r^{-1} & \text{for } R < 1. \end{cases} \tag{2.7}$$

For the cavityGrade case the number of cells in each direction in a block is 10, the ratio between largest and smallest cells is 2 and the block height and width is 0.05 m. Therefore the smallest cell length is 3.45 mm. From Equation 2.2, the time step should be less than 3.45 ms to maintain a Courant of less than 1. To ensure that results are written out at convenient time intervals, the time step `deltaT` should be reduced to 2.5 ms and the `writeInterval` set to 40 so that results are written out every 0.1 s. These settings can be viewed in the *cavityGrade/system/controlDict* file.

The `startTime` needs to be set to that of the final conditions of the case cavityFine, *i.e.* `0.7`. Since cavity and cavityFine converged well within the prescribed run time, we can set the run time for case cavityGrade to 0.1 s, *i.e.* the `endTime` should be `0.8`.

2.1.6.3 Mapping fields

First,

As in section 2.1.5.3, use mapFields to map the final results from case cavityFine onto the mesh for case cavityGrade. From the *cavityGrade* directory, execute mapFields by:

```
mapFields -consistent ../cavityFine
```

Now run icoFoam from the case directory and monitor the run time information. View the converged results for this case and compare with other results using post-processing tools described previously in section 2.1.5.6 and section 2.1.5.7.

2.1.7 Increasing the Reynolds number

The cases solved so far have had a Reynolds number of 10. This is very low and leads to a stable solution quickly with only small secondary vortices at the bottom corners of the cavity. We will now increase the Reynolds number to 100, at which point the solution takes a noticeably longer time to converge. The coarsest mesh in case cavity will be used initially. The user should clone the cavity case and name it cavityHighRe. Rather

than copying individual directories (*system*, *constant*, *etc.*) as described previously, the foamCloneCase can be used, which copies the relevant directories in one step. By default the *0* time directory is copied, but here the user can use the `-latestTime` option to copy the latest time directory, *0.5*, which can be used as the initial field data for our simulation. The example also uses the run alias as a quick way to change to the *run* directory.

```
run
foamCloneCase -latestTime cavity cavityHighRe
cd cavityHighRe
```

2.1.7.1 Pre-processing

Go into the cavityHighRe case and edit the *physicalProperties* dictionary in the *constant* directory. Since the Reynolds number is required to be increased by a factor of 10, decrease the kinematic viscosity by a factor of 10, *i.e.* to 1×10^{-3} $m^2 s^{-1}$. We now run this case by restarting from the solution at the end of the cavity case run. To do this we can use the option of setting the `startFrom` keyword to `latestTime` so that icoFoam takes as its initial data the values stored in the directory corresponding to the most recent time, *i.e.* *0.5*. The `endTime` should be set to 2 s.

2.1.7.2 Running the code

Run icoFoam for this case from the case directory and view the run time information. When running a job in the background, the following UNIX commands can be useful:

`nohup` enables a command to keep running after the user who issues the command has logged out;

`nice` changes the priority of the job in the kernel's scheduler; a "niceness" of -20 is the highest priority and 19 is the lowest priority.

This is useful, for example, if a user wishes to set a case running on a remote machine and does not wish to monitor it heavily, in which case they may wish to give it low priority on the machine. In that case the `nohup` command allows the user to log out of a remote machine he/she is running on and the job continues running, while `nice` can set the priority to 19. For our case of interest, we can execute the command in this manner as follows:

```
nohup nice -n 19 icoFoam > log &
cat log
```

In previous runs you may have noticed that icoFoam stops solving for velocity `U` quite quickly but continues solving for pressure `p` for a lot longer or until the end of the run. In practice, once icoFoam stops solving for `U` and the initial residual of `p` is less than the tolerance set in the *fvSolution* dictionary (typically 10^{-6}), the run has effectively converged and can be stopped once the field data has been written out to a time directory. For example, at convergence a sample of the *log* file from the run on the cavityHighRe case appears as follows in which the velocity has already converged after 1.395 s and initial pressure residuals are small; `No Iterations 0` indicates that the solution of `U` has stopped:

```
Time = 1.43

Courant Number mean: 0.221921 max: 0.839902
smoothSolver:  Solving for Ux, Initial residual = 8.733e-06, Final residual = 8.733e-06, No Iterations 0
smoothSolver:  Solving for Uy, Initial residual = 9.896e-06, Final residual = 9.896e-06, No Iterations 0
DICPCG:  Solving for p, Initial residual = 3.675e-06, Final residual = 8.629e-07, No Iterations 4
time step continuity errors : sum local = 6.579e-09, global = -6.667e-19, cumulative = -6.2539e-18
DICPCG:  Solving for p, Initial residual = 2.608e-06, Final residual = 7.925e-07, No Iterations 3
time step continuity errors : sum local = 6.261e-09, global = -1.029e-18, cumulative = -7.28374e-18
ExecutionTime = 0.37 s  ClockTime = 0 s

Time = 1.435

Courant Number mean: 0.221923 max: 0.839903
smoothSolver:  Solving for Ux, Initial residual = 8.539e-06, Final residual = 8.539e-06, No Iterations 0
smoothSolver:  Solving for Uy, Initial residual = 9.714e-06, Final residual = 9.714e-06, No Iterations 0
DICPCG:  Solving for p, Initial residual = 4.022e-06, Final residual = 9.896e-07, No Iterations 3
time step continuity errors : sum local = 8.151e-09, global = 5.336e-19, cumulative = -6.75012e-18
DICPCG:  Solving for p, Initial residual = 2.388e-06, Final residual = 8.445e-07, No Iterations 3
time step continuity errors : sum local = 7.487e-09, global = -4.427e-19, cumulative = -7.19283e-18
ExecutionTime = 0.37 s  ClockTime = 0 s
```

2.1.8 High Reynolds number flow

View the results in paraFoam and display the velocity vectors. The secondary vortices in the corners have increased in size. The user can then increase the Reynolds number further by decreasing the viscosity and then rerun the case. The number of vortices increases so the mesh resolution around them will need to increase in order to resolve the more complicated flow patterns. In addition, as the Reynolds number increases the time to convergence increases. The user should monitor residuals and extend the `endTime` accordingly to ensure convergence.

The need to increase spatial and temporal resolution then becomes impractical as the flow moves into the turbulent regime, where problems of solution stability may also occur. Of course, many engineering problems have very high Reynolds numbers and it is infeasible to bear the huge cost of solving the turbulent behaviour directly. Instead Reynolds-averaged simulation (RAS) turbulence models are used to solve for the mean flow behaviour and calculate the statistics of the fluctuations. The standard $k-\varepsilon$ model with wall functions will be used in this tutorial to solve the lid-driven cavity case with a Reynolds number of 10^4. Two extra variables are solved for: k, the turbulent kinetic energy; and, ε, the turbulent dissipation rate. The additional equations and models for turbulent flow are implemented into a OpenFOAM solver called pisoFoam.

2.1.8.1 Pre-processing

Go back to the run directory and copy the cavity case in the *$FOAM_RUN/tutorials/-incompressible/pisoFoam/RAS* directory (N.B: the ***pisoFoam/RAS*** directory), renaming it cavityRAS to avoid a clash with the existing cavity tutorial. Go into the new case directory.

```
run
cp -r $FOAM_TUTORIALS/incompressible/pisoFoam/RAS/cavity cavityRAS
cd cavityRAS
```

Generate the mesh by running blockMesh as before. Mesh grading towards the wall is not necessary when using the standard $k-\varepsilon$ model with wall functions since the flow in the near wall cell is modelled, rather than having to be resolved.

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 models are specified through the turbulent viscosity field, ν_t in the *0/nut* file:

```
16
17  dimensions      [0 2 -1 0 0 0 0];
18
19  internalField   uniform 0;
20
21  boundaryField
22  {
23      movingWall
24      {
25          type            nutkWallFunction;
26          value           uniform 0;
27      }
28      fixedWalls
29      {
30          type            nutkWallFunction;
31          value           uniform 0;
32      }
33      frontAndBack
34      {
35          type            empty;
36      }
37  }
38
39
40  // ************************************************************************* //
```

This case uses standard wall functions, specified by the `nutWallFunction` type on the `movingWall` and `fixedWalls` patches. Other wall function models include the rough wall functions, specified through the `nutRoughWallFunction` keyword.

The user should now open the field files for k and ε (*0/k* and *0/epsilon*) and examine their boundary conditions. For a wall boundary condition, ε is assigned a epsilonWallFunction boundary condition and a kqRwallFunction boundary condition is assigned to k. The latter is a generic boundary condition that can be applied to any field that are of a turbulent kinetic energy type, *e.g.* k, q or Reynolds Stress R. The initial values for k and ε are set using an estimated fluctuating component of velocity $\mathbf{U}'$ and a turbulent length scale, l. k and ε are defined in terms of these parameters as follows:

$$k = \frac{1}{2}\overline{\mathbf{U}' \bullet \mathbf{U}'} \tag{2.8}$$

$$\varepsilon = \frac{C_\mu^{0.75} k^{1.5}}{l} \tag{2.9}$$

where C_μ is a constant of the $k-\varepsilon$ model equal to 0.09. For a Cartesian coordinate system, k is given by:

$$k = \frac{1}{2}({U'_x}^2 + {U'_y}^2 + {U'_z}^2) \tag{2.10}$$

where ${U'_x}^2$, ${U'_y}^2$ and ${U'_z}^2$ are the fluctuating components of velocity in the x, y and z directions respectively. Let us assume the initial turbulence is isotropic, *i.e.* ${U'_x}^2 = {U'_y}^2 = {U'_z}^2$, and equal to 5% of the lid velocity and that l, is equal to 5% of the box width, 0.1 m, then k and ε are given by:

$$U'_x = U'_y = U'_z = \frac{5}{100} 1 \text{ m s}^{-1} \tag{2.11}$$

$$\Rightarrow k = \frac{3}{2}\left(\frac{5}{100}\right)^2 \text{ m}^2\,\text{s}^{-2} = 3.75 \times 10^{-3} \text{ m}^2\,\text{s}^{-2} \tag{2.12}$$

$$\varepsilon = \frac{C_\mu^{0.75} k^{1.5}}{l} \approx 7.54 \times 10^{-3} \text{ m}^2\text{s}^{-3} \tag{2.13}$$

These form the initial conditions for k and ε. The initial conditions for $\mathbf{U}$ and p are $(0, 0, 0)$ and 0 respectively as before.

Turbulence modelling includes a range of methods, *e.g.* RAS or large-eddy simulation (LES), that are provided in OpenFOAM. The choice of turbulence modelling method is selectable at run-time through the `simulationType` keyword in *momentumTransport* dictionary (known as *turbulenceProperties* prior to OpenFOAM v8). The user can view this file in the *constant* directory:

```
16
17  simulationType  RAS;
18
19  RAS
20  {
21      model           kEpsilon;
22
23      turbulence      on;
24
25      printCoeffs     on;
26  }
27
28  // ************************************************************************* //
```

The options for `simulationType` are `laminar`, `RAS` and `LES`. With `RAS` selected in this case, the choice of RAS modelling is specified in a `RAS` subdictionary. The turbulence model is selected by the `model` entry from a long list of available models that are listed in Section 7.2.1.1. The `kEpsilon` model should be selected which is the standard $k-\varepsilon$ model; the user should also ensure that `turbulence` calculation is switched `on`.

The coefficients for each turbulence model are stored within the respective code with a set of default values. Setting the optional switch called `printCoeffs` to `on` will make the default values be printed to standard output, *i.e.* the terminal, when the model is called at run time. The coefficients are printed out as a sub-dictionary whose name is that of the model name with the word `Coeffs` appended, *e.g.* `kEpsilonCoeffs` in the case of the `kEpsilon` model. The coefficients of the model, *e.g.* `kEpsilon`, can be modified by optionally including (copying and pasting) that sub-dictionary within the `RAS` sub-dictionary and adjusting values accordingly.

The user should next set the laminar kinematic viscosity in the *physicalProperties* dictionary. To achieve a Reynolds number of 10^4, a kinematic viscosity of 10^{-5} m is required based on the Reynolds number definition given in Equation 2.1.

Finally the user should set the `startTime`, `stopTime`, `deltaT` and the `writeInterval` in the *controlDict*. Set `deltaT` to 0.005 s to satisfy the Courant number restriction and the `endTime` to 10 s.

2.1.8.2 Running the code

Execute pisoFoam by entering the case directory and typing "`pisoFoam`" in a terminal. In this case, where the viscosity is low, the boundary layer next to the moving lid is very thin and the cells next to the lid are comparatively large so the velocity at their centres are much less than the lid velocity. In fact, after $\approx$ 100 time steps it becomes apparent that the velocity in the cells adjacent to the lid reaches an upper limit of around 0.2 m s^{-1} hence the maximum Courant number does not rise much above 0.2. It is sensible to increase the solution time by increasing the time step to a level where the Courant number is much closer to 1. Therefore reset `deltaT` to 0.02 s and, on this occasion, set `startFrom` to `latestTime`. This instructs pisoFoam to read the start data from the latest time directory, *i.e. 10.0*. The `endTime` should be set to 20 s since the run converges a lot slower than the laminar case. Restart the run as before and monitor the convergence of the solution. View the results at consecutive time steps as the solution progresses to see if the solution converges to a steady-state or perhaps reaches some periodically oscillating

state. In the latter case, convergence may never occur but this does not mean the results are inaccurate.

2.1.9 Changing the case geometry

A user may wish to make changes to the geometry of a case and perform a new simulation. It may be useful to retain some or all of the original solution as the starting conditions for the new simulation. This is a little complex because the fields of the original solution are not consistent with the fields of the new case. However the mapFields utility can map fields that are inconsistent, either in terms of geometry or boundary types or both.

As an example, let us copy the cavityClipped case from the *tutorials* directory in the user's *run* directory, and change into the cavityClipped directory:

```
run
cp -r $FOAM_TUTORIALS/incompressible/icoFoam/cavity/cavityClipped .
cd cavityClipped
```

The case consists of the standard cavity geometry but with a square of length 0.04 m removed from the bottom right of the cavity, according to the *blockMeshDict* below:

```
16  convertToMeters 0.1;
17
18  vertices
19  (
20      (0 0 0)
21      (0.6 0 0)
22      (0 0.4 0)
23      (0.6 0.4 0)
24      (1 0.4 0)
25      (0 1 0)
26      (0.6 1 0)
27      (1 1 0)
28
29      (0 0 0.1)
30      (0.6 0 0.1)
31      (0 0.4 0.1)
32      (0.6 0.4 0.1)
33      (1 0.4 0.1)
34      (0 1 0.1)
35      (0.6 1 0.1)
36      (1 1 0.1)
37
38  );
39
40  blocks
41  (
42      hex (0 1 3 2 8 9 11 10) (12 8 1) simpleGrading (1 1 1)
43      hex (2 3 6 5 10 11 14 13) (12 12 1) simpleGrading (1 1 1)
44      hex (3 4 7 6 11 12 15 14) (8 12 1) simpleGrading (1 1 1)
45  );
46
47  boundary
48  (
49      lid
50      {
51          type wall;
52          faces
53          (
54              (5 13 14 6)
55              (6 14 15 7)
56          );
57      }
58      fixedWalls
59      {
60          type wall;
61          faces
62          (
63              (0 8 10 2)
64              (2 10 13 5)
65              (7 15 12 4)
66              (4 12 11 3)
```

```
67              (3 11 9 1)
68              (1 9 8 0)
69          );
70      }
71      frontAndBack
72      {
73          type empty;
74          faces
75          (
76              (0 2 3 1)
77              (2 5 6 3)
78              (3 6 7 4)
79              (8 9 11 10)
80              (10 11 14 13)
81              (11 12 15 14)
82          );
83      }
84  );
85
86
87  // ************************************************************************* //
```

Generate the mesh with blockMesh. The patches are set accordingly as in previous cavity cases. For the sake of clarity in describing the field mapping process, the upper wall patch is renamed `lid`, previously the `movingWall` patch of the original cavity.

In an inconsistent mapping, there is no guarantee that all the field data can be mapped from the source case. The remaining data must come from field files in the target case itself. Therefore field data must exist in the time directory of the target case before mapping takes place. In the cavityClipped case the mapping is set to occur at time 0.5 s, since the `startTime` is set to 0.5 s in the *controlDict*. Therefore the user needs to copy initial field data to that directory, *e.g.* from time 0:

```
cp -r 0 0.5
```

Before mapping the data, the user should view the geometry and fields at 0.5 s.

Now we wish to map the velocity and pressure fields from cavity onto the new fields of cavityClipped. Since the mapping is inconsistent, we need to edit the *mapFieldsDict* dictionary, located in the *system* directory. The dictionary contains 2 keyword entries: `patchMap` and `cuttingPatches`. The `patchMap` list contains a mapping of patches from the source fields to the target fields. It is used if the user wishes a patch in the target field to inherit values from a corresponding patch in the source field. In cavityClipped, we wish to inherit the boundary values on the `lid` patch from `movingWall` in cavity so we must set the `patchMap` as:

```
patchMap
(
    lid movingWall
);
```

The `cuttingPatches` list contains names of target patches whose values are to be mapped from the source internal field through which the target patch cuts. In this case, the `fixedWalls` patch is a noSlip condition so the internal values cannot be interpolated to the patch. Therefore the `cuttingPatches` list can simply be empty:

```
cuttingPatches
(
);
```

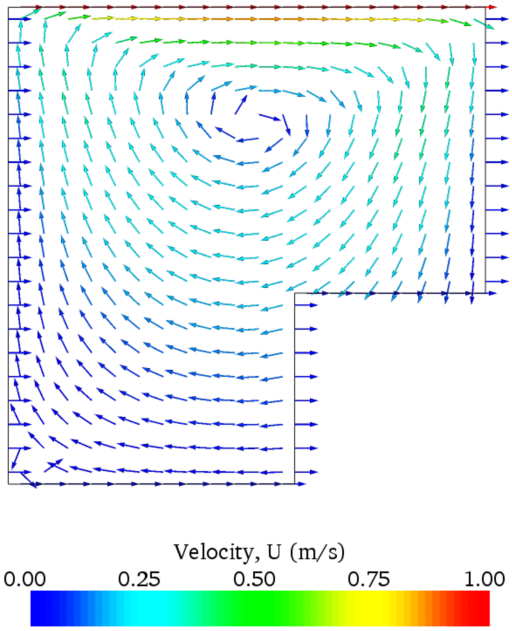


Figure 2.14: cavity solution velocity field mapped onto cavityClipped.

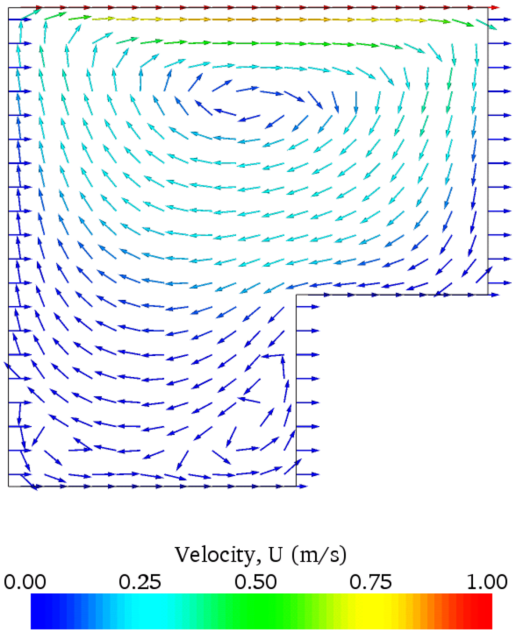


Figure 2.15: cavityClipped solution for velocity field.

If the user does wish to interpolate internal values from the source case to the `fixedWalls` patch in the target case, a fixedValue boundary condition needs to be specified on the patch, whose `value` can then be overridden during the mapping process; the `fixedWalls` patch then needs to be included in the `cuttingPatches` list.

The user should run mapFields, from within the *cavityClipped* directory:

```
mapFields ../cavity
```

The user can view the mapped field as shown in Figure 2.14. The `fixedWalls` patch has not inherited values from the source case as we expected. The user can then run the case with icoFoam.

2.1.10 Post-processing the modified geometry

Velocity glyphs can be generated for the case as normal, first at time 0.5 s and later at time 0.6 s, to compare the initial and final solutions. In addition, we provide an outline of the geometry which requires some care to generate for a 2D case. The user should select `Extract Block` from the `Filter` menu and, in the Parameter panel, highlight the patches of interest, namely the lid and fixedWalls. On clicking Apply, these items of geometry can be displayed by selecting `Wireframe` in the Display panel. Figure 2.15 displays the patches in black and shows vortices forming in the bottom corners of the modified geometry.

2.2 Stress analysis of a plate with a hole

This tutorial describes how to pre-process, run and post-process a case involving linear-elastic, steady-state stress analysis on a square plate with a circular hole at its centre. The plate dimensions are: side length 4 m and radius $R = 0.5$ m. It is loaded with a uniform traction of $\sigma = 10$ kPa over its left and right faces as shown in Figure 2.16. Two symmetry planes can be identified for this geometry and therefore the solution domain need only cover a quarter of the geometry, shown by the shaded area in Figure 2.16.

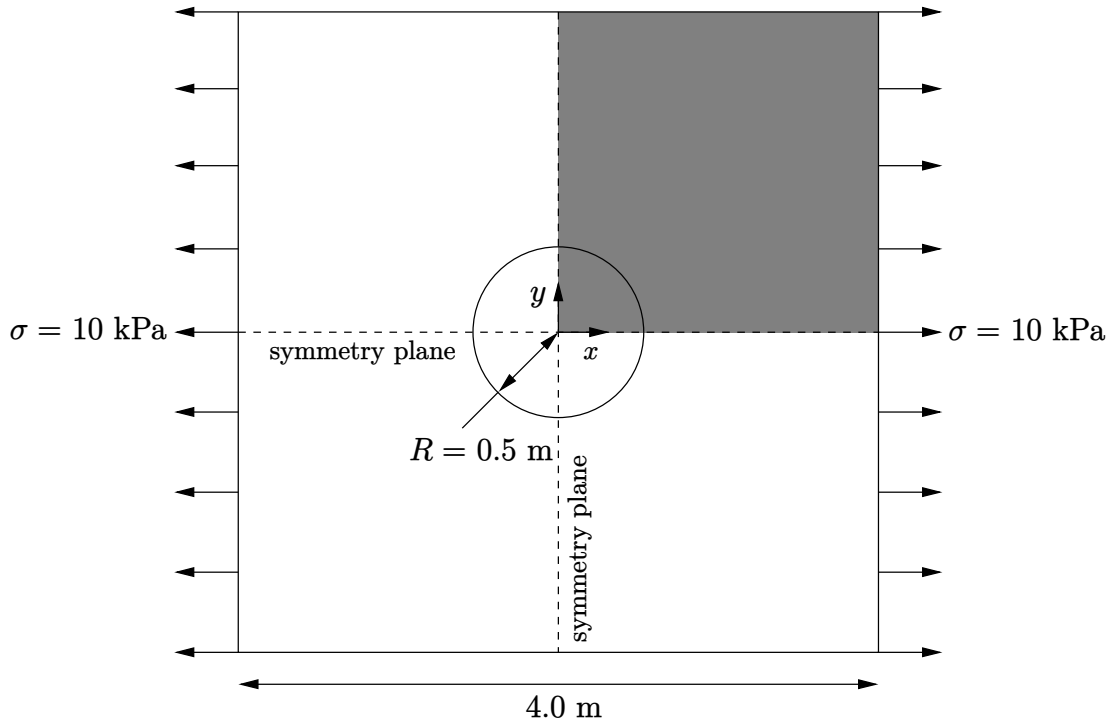


Figure 2.16: Geometry of the plate with a hole.

The problem can be approximated as 2-dimensional since the load is applied in the plane of the plate. In a Cartesian coordinate system there are two possible assumptions to take in regard to the behaviour of the structure in the third dimension: (1) the plane stress condition, in which the stress components acting out of the 2D plane are assumed to be negligible; (2) the plane strain condition, in which the strain components out of the 2D plane are assumed negligible. The plane stress condition is appropriate for solids whose third dimension is thin as in this case; the plane strain condition is applicable for solids where the third dimension is thick.

An analytical solution exists for loading of an infinitely large, thin plate with a circular

hole. The solution for the stress normal to the vertical plane of symmetry is

$$(\sigma_{xx})_{x=0} = \begin{cases} \sigma\left(1 + \dfrac{R^2}{2y^2} + \dfrac{3R^4}{2y^4}\right) & \text{for } |y| \geq R \\ 0 & \text{for } |y| < R \end{cases} \tag{2.14}$$

Results from the simulation will be compared with this solution. At the end of the tutorial, the user can: investigate the sensitivity of the solution to mesh resolution and mesh grading; and, increase the size of the plate in comparison to the hole to try to estimate the error in comparing the analytical solution for an infinite plate to the solution of this problem of a finite plate.

2.2.1 Mesh generation

The domain consists of four blocks, some of which have arc-shaped edges. The block structure for the part of the mesh in the $x-y$ plane is shown in Figure 2.17. As already mentioned in section 2.1.1.1, all geometries are generated in 3 dimensions in OpenFOAM even if the case is to be as a 2 dimensional problem. Therefore a dimension of the block in the z direction has to be chosen; here, 0.5 m is selected. It does not affect the solution since the traction boundary condition is specified as a stress rather than a force, thereby making the solution independent of the cross-sectional area.

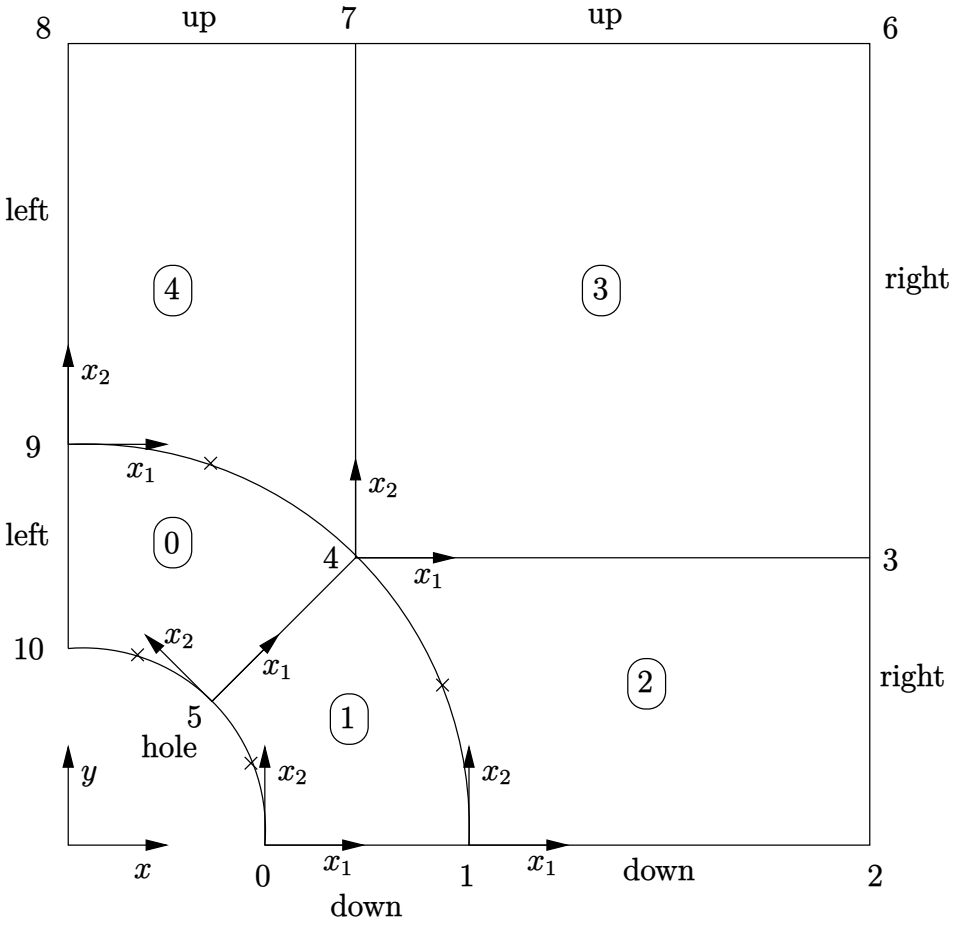


Figure 2.17: Block structure of the mesh for the plate with a hole.

The user should change to the *run* directory and copy the plateHole case into it from the *$FOAM_TUTORIALS/stressAnalysis/solidDisplacementFoam* directory. The user should

then go into the plateHole directory and open the *blockMeshDict* file in an editor, as listed below

```
16  convertToMeters 1;
17
18  vertices
19  (
20      (0.5 0 0)
21      (1 0 0)
22      (2 0 0)
23      (2 0.707107 0)
24      (0.707107 0.707107 0)
25      (0.353553 0.353553 0)
26      (2 2 0)
27      (0.707107 2 0)
28      (0 2 0)
29      (0 1 0)
30      (0 0.5 0)
31      (0.5 0 0.5)
32      (1 0 0.5)
33      (2 0 0.5)
34      (2 0.707107 0.5)
35      (0.707107 0.707107 0.5)
36      (0.353553 0.353553 0.5)
37      (2 2 0.5)
38      (0.707107 2 0.5)
39      (0 2 0.5)
40      (0 1 0.5)
41      (0 0.5 0.5)
42  );
43
44  blocks
45  (
46      hex (5 4 9 10 16 15 20 21) (10 10 1) simpleGrading (1 1 1)
47      hex (0 1 4 5 11 12 15 16) (10 10 1) simpleGrading (1 1 1)
48      hex (1 2 3 4 12 13 14 15) (20 10 1) simpleGrading (1 1 1)
49      hex (4 3 6 7 15 14 17 18) (20 20 1) simpleGrading (1 1 1)
50      hex (9 4 7 8 20 15 18 19) (10 20 1) simpleGrading (1 1 1)
51  );
52
53  edges
54  (
55      arc 0 5 (0.469846 0.17101 0)
56      arc 5 10 (0.17101 0.469846 0)
57      arc 1 4 (0.939693 0.34202 0)
58      arc 4 9 (0.34202 0.939693 0)
59      arc 11 16 (0.469846 0.17101 0.5)
60      arc 16 21 (0.17101 0.469846 0.5)
61      arc 12 15 (0.939693 0.34202 0.5)
62      arc 15 20 (0.34202 0.939693 0.5)
63  );
64
65  boundary
66  (
67      left
68      {
69          type symmetryPlane;
70          faces
71          (
72              (8 9 20 19)
73              (9 10 21 20)
74          );
75      }
76      right
77      {
78          type patch;
79          faces
80          (
81              (2 3 14 13)
82              (3 6 17 14)
83          );
84      }
85      down
86      {
87          type symmetryPlane;
88          faces
89          (
90              (0 1 12 11)
91              (1 2 13 12)
92          );
93      }
94      up
```

```
95      {
96          type patch;
97          faces
98          (
99              (7 8 19 18)
100             (6 7 18 17)
101         );
102     }
103     hole
104     {
105         type patch;
106         faces
107         (
108             (10 5 16 21)
109             (5 0 11 16)
110         );
111     }
112     frontAndBack
113     {
114         type empty;
115         faces
116         (
117             (10 9 4 5)
118             (5 4 1 0)
119             (1 4 3 2)
120             (4 7 6 3)
121             (4 9 8 7)
122             (21 16 15 20)
123             (16 11 12 15)
124             (12 13 14 15)
125             (15 14 17 18)
126             (15 18 19 20)
127         );
128     }
129 );
130
131
132 // ************************************************************************* //
```

Until now, we have only specified straight edges in the geometries of previous tutorials but here we need to specify curved edges. These are specified under the `edges` keyword entry which is a list of non-straight edges. The syntax of each list entry begins with the type of curve, including `arc`, `simpleSpline`, `polyLine` *etc.*, described further in section 5.3.1. In this example, all the edges are circular and so can be specified by the `arc` keyword entry. The following entries are the labels of the start and end vertices of the arc and a point vector through which the circular arc passes.

The blocks in this *blockMeshDict* do not all have the same orientation. As can be seen in Figure 2.17 the x_2 direction of block 0 is equivalent to the $-x_1$ direction for block 4. This means care must be taken when defining the number and distribution of cells in each block so that the cells match up at the block faces.

6 patches are defined: one for each side of the plate, one for the hole and one for the front and back planes. The `left` and `down` patches are both a symmetry plane. Since this is a *geometric* constraint, it is included in the definition of the *mesh*, rather than being purely a specification on the boundary condition of the fields. Therefore they are defined as such using a special symmetryPlane type as shown in the *blockMeshDict*.

The `frontAndBack` patch represents the plane which is ignored in a 2D case. Again this is a geometric constraint so is defined within the mesh, using the empty type as shown in the *blockMeshDict*. For further details of boundary types and geometric constraints, the user should refer to section 5.2.

The remaining patches are of the regular patch type. The mesh should be generated using blockMesh and can be viewed in paraFoam as described in section 2.1.2. It should appear as in Figure 2.18.

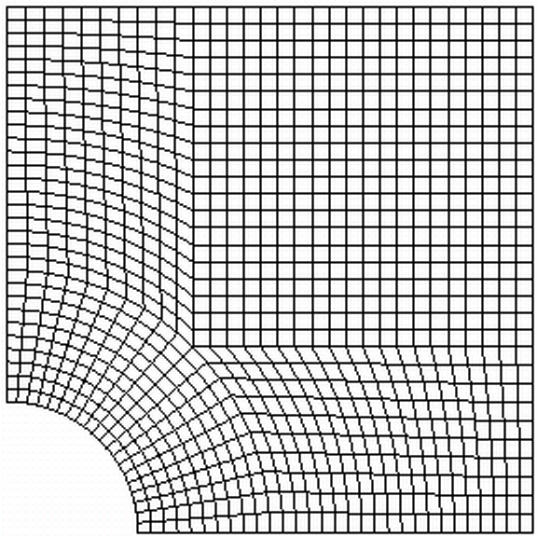


Figure 2.18: Mesh of the hole in a plate problem.

2.2.1.1 Boundary and initial conditions

Once the mesh generation is complete, the initial field with boundary conditions must be set. For a stress analysis case without thermal stresses, only displacement `D` needs to be set. The *0/D* is as follows:

```
16  dimensions      [0 1 0 0 0 0 0];
17
18  internalField   uniform (0 0 0);
19
20  boundaryField
21  {
22      left
23      {
24          type            symmetryPlane;
25      }
26      right
27      {
28          type            tractionDisplacement;
29          traction        uniform (10000 0 0);
30          pressure        uniform 0;
31          value           uniform (0 0 0);
32      }
33      down
34      {
35          type            symmetryPlane;
36      }
37      up
38      {
39          type            tractionDisplacement;
40          traction        uniform (0 0 0);
41          pressure        uniform 0;
42          value           uniform (0 0 0);
43      }
44      hole
45      {
46          type            tractionDisplacement;
47          traction        uniform (0 0 0);
48          pressure        uniform 0;
49          value           uniform (0 0 0);
50      }
51      frontAndBack
52      {
53          type            empty;
54      }
55  }
56
57  // ************************************************************************* //
```

Firstly, it can be seen that the displacement initial conditions are set to $(0, 0, 0)$ m. The

`left` and `down` patches **must** be both of symmetryPlane type since they are specified as such in the mesh description in the *constant/polyMesh/boundary* file. Similarly the `frontAndBack` patch is declared empty.

The other patches are traction boundary conditions, set by a specialist traction boundary type. The traction boundary conditions are specified by a linear combination of: (1) a boundary traction vector under keyword `traction`; (2) a pressure that produces a traction normal to the boundary surface that is defined as negative when pointing out of the surface, under keyword `pressure`. The `up` and `hole` patches are zero traction so the boundary traction and pressure are set to zero. For the `right` patch the traction should be $(1e4, 0, 0)$ Pa and the pressure should be 0 Pa.

2.2.1.2 Physical properties

The physical properties for the case are set in the *physicalProperties* dictionary in the *constant* directory, shown below:

```
16
17  rho
18  {
19      type        uniform;
20      value       7854;
21  }
22
23  nu
24  {
25      type        uniform;
26      value       0.3;
27  }
28
29  E
30  {
31      type        uniform;
32      value       2e+11;
33  }
34
35  Cp
36  {
37      type        uniform;
38      value       434;
39  }
40
41  kappa
42  {
43      type        uniform;
44      value       60.5;
45  }
46
47  alphav
48  {
49      type        uniform;
50      value       1.1e-05;
51  }
52
53  planeStress     yes;
54  thermalStress   no;
55
56
57  // ************************************************************************* //
```

The file includes mechanical properties of steel:

- Density `rho` $= 7854$ kg m^{-3}
- Young's modulus `E` $= 2 \times 10^{11}$ Pa
- Poisson's ratio `nu` $= 0.3$

The `planeStress` switch is set to `yes` to adopt the plane stress assumption in this 2D case. The solidDisplacementFoam solver may optionally solve a thermal equation that is

coupled with the momentum equation through the thermal stresses that are generated. The user specifies at run time whether OpenFOAM should solve the thermal equation by the `thermalStress` switch (currently set to `no`). The thermal properties are also specified for steel for this case, *i.e.*:

- Specific heat capacity `Cp` = 434 $\mathrm{Jkg^{-1}K^{-1}}$
- Thermal conductivity `kappa` = 60.5 $\mathrm{Wm^{-1}K^{-1}}$
- Thermal expansion coefficient `alphav` = 1.1×10^{-5} $\mathrm{K^{-1}}$

For thermal calculations, the temperature field variable `T` is present in the *0* directory.

2.2.1.3 Control

As before, the information relating to the control of the solution procedure are read in from the *controlDict* dictionary. For this case, the `startTime` is 0 s. The time step is not important since this is a steady state case; in this situation it is best to set the time step `deltaT` to 1 so it simply acts as an iteration counter for the steady-state case. The `endTime`, set to 100, then acts as a limit on the number of iterations. The `writeInterval` can be set to 20.

The *controlDict* entries are as follows:

```
16
17  application     solidDisplacementFoam;
18
19  startFrom       startTime;
20
21  startTime       0;
22
23  stopAt          endTime;
24
25  endTime         100;
26
27  deltaT          1;
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  graphFormat     raw;
46
47  runTimeModifiable true;
48
49
50  // ************************************************************************* //
```

2.2.1.4 Discretisation schemes and linear-solver control

Let us turn our attention to the *fvSchemes* dictionary. Firstly, the problem we are analysing is steady-state so the user should select `SteadyState` for the time derivatives in `timeScheme`. This essentially switches off the time derivative terms. Not all solvers, especially in fluid dynamics, work for both steady-state and transient problems but solidDisplacementFoam does work, since the base algorithm is the same for both types of simulation.

The momentum equation in linear-elastic stress analysis includes several explicit terms containing the gradient of displacement. The calculations benefit from accurate and smooth evaluation of the gradient. Normally, in the finite volume method the discretisation is based on Gauss's theorem. The Gauss method is sufficiently accurate for most purposes but, in this case, the least squares method will be used. The user should therefore open the `fvSchemes` dictionary in the *system* directory and ensure the `leastSquares` method is selected for the `grad(U)` gradient discretisation scheme in the `gradSchemes` sub-dictionary:

```
16
17  d2dt2Schemes
18  {
19      default         steadyState;
20  }
21
22  ddtSchemes
23  {
24      default         Euler;
25  }
26
27  gradSchemes
28  {
29      default         leastSquares;
30      grad(D)         leastSquares;
31      grad(T)         leastSquares;
32  }
33
34  divSchemes
35  {
36      default         none;
37      div(sigmaD)     Gauss linear;
38  }
39
40  laplacianSchemes
41  {
42      default         none;
43      laplacian(DD,D) Gauss linear corrected;
44      laplacian(kappa,T) Gauss linear corrected;
45  }
46
47  interpolationSchemes
48  {
49      default         linear;
50  }
51
52  snGradSchemes
53  {
54      default         none;
55  }
56
57  // ************************************************************************* //
```

The *fvSolution* dictionary in the *system* directory controls the linear equation solvers and algorithms used in the solution. The user should first look at the *solvers* sub-dictionary and notice that the choice of `solver` for `D` is `GAMG`. The solver `tolerance` should be set to 10^{-6} for this problem. The solver relative tolerance, denoted by `relTol`, sets the required reduction in the residuals within each iteration. It is uneconomical to set a tight (low) relative tolerance within each iteration since a lot of terms in each equation are explicit and are updated as part of the segregated iterative procedure. Therefore a reasonable value for the relative tolerance is 0.01, or possibly even higher, say 0.1, or in some cases even 0.9 (as in this case).

```
16
17  solvers
18  {
19      "(D|T)"
20      {
21          solver          GAMG;
22          tolerance       1e-06;
23          relTol          0.9;
24          smoother        GaussSeidel;
```

```
25          nCellsInCoarsestLevel 20;
26      }
27  }
28
29  stressAnalysis
30  {
31      compactNormalStress yes;
32      nCorrectors     1;
33      D               1e-06;
34  }
35
36
37  // ************************************************************************* //
```

The *fvSolution* dictionary contains a sub-dictionary, *stressAnalysis* that contains some control parameters specific to the application solver. Firstly there is `nCorrectors` which specifies the number of outer loops around the complete system of equations, including traction boundary conditions *within each time step*. Since this problem is steady-state, we are performing a set of iterations towards a converged solution with the 'time step' acting as an iteration counter. We can therefore set `nCorrectors` to 1.

The `D` keyword specifies a convergence tolerance for the outer iteration loop, *i.e.* sets a level of initial residual below which solving will cease. It should be set to the desired solver tolerance specified earlier, 10^{-6} for this problem.

2.2.2 Running the code

The user should run the code here in the background from the command line as specified below, so he/she can look at convergence information in the log file afterwards.

```
solidDisplacementFoam > log &
```

The user should check the convergence information by viewing the generated *log* file which shows the number of iterations and the initial and final residuals of the displacement in each direction being solved. The final residual should always be less than 0.9 times the initial residual as this iteration tolerance set. Once both initial residuals have dropped below the convergence tolerance of 10^{-6} the run has converged and can be stopped by killing the batch job.

2.2.3 Post-processing

Post processing can be performed as in section 2.1.4. The solidDisplacementFoam solver outputs the stress field σ as a symmetric tensor field `sigma`. This is consistent with the way variables are usually represented in OpenFOAM solvers by the mathematical symbol by which they are represented; in the case of Greek symbols, the variable is named phonetically.

For post-processing individual scalar field components, σ_{xx}, σ_{xy} *etc.*, can be generated by running the postProcess utility as before in section 2.1.5.7, this time on `sigma`:

```
postProcess -func "components(sigma)"
```

Components named `sigmaxx`, `sigmaxy` *etc.* are written to time directories of the case. The σ_{xx} stresses can be viewed in paraFoam as shown in Figure 2.19.

We would like to compare the analytical solution of Equation 2.14 to our solution. We therefore must output a set of data of σ_{xx} along the left edge symmetry plane of our domain. The user may generate the required graph data using the postProcess utility

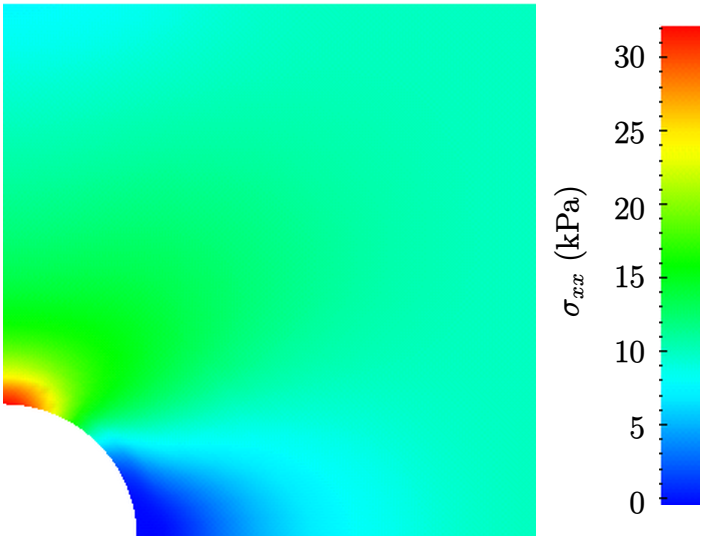


Figure 2.19: σ_{xx} stress field in the plate with hole.

with the `graphUniform` function. Unlike earlier examples of postProcess where no configuration is required, this example includes a *graphUniform* file pre-configured in the *system* directory. The sample line is set between $(0.0, 0.5, 0.25)$ and $(0.0, 2.0, 0.25)$, and the fields are specified in the `fields` list:

```
9       Writes graph data for specified fields along a line, specified by start and
10      end points. A specified number of graph points are used, distributed
11      uniformly along the line.
12
13  \*---------------------------------------------------------------------------*/
14
15  start           (0 0.5 0.25);
16  end             (0 2 0.25);
17  nPoints         100;
18
19  fields          (sigmaxx);
20
21  axis            y;
22
23  #includeEtc "caseDicts/postProcessing/graphs/graphUniform.cfg"
24
25  // ************************************************************************* //
```

The user should execute postProcessing with the `graphUniform` function:

```
postProcess -func graphUniform
```

Data is written in `raw` 2 column format into files within time subdirectories of a *postProcessing/graphUniform* directory, *e.g.* the data at $t = 100$ s is found within the file *graphUniform/100/line.xy*. If the user has GnuPlot installed they launch it (by typing `gnuplot`) and then plot both the numerical data and analytical solution as follows:

```
plot [0.5:2] [0:] "postProcessing/graphUniform/100/line.xy",
    1e4*(1+(0.125/(x**2))+(0.09375/(x**4)))
```

An example plot is shown in Figure 2.20.

2.2.4 Exercises

The user may wish to experiment with solidDisplacementFoam by trying the following exercises:

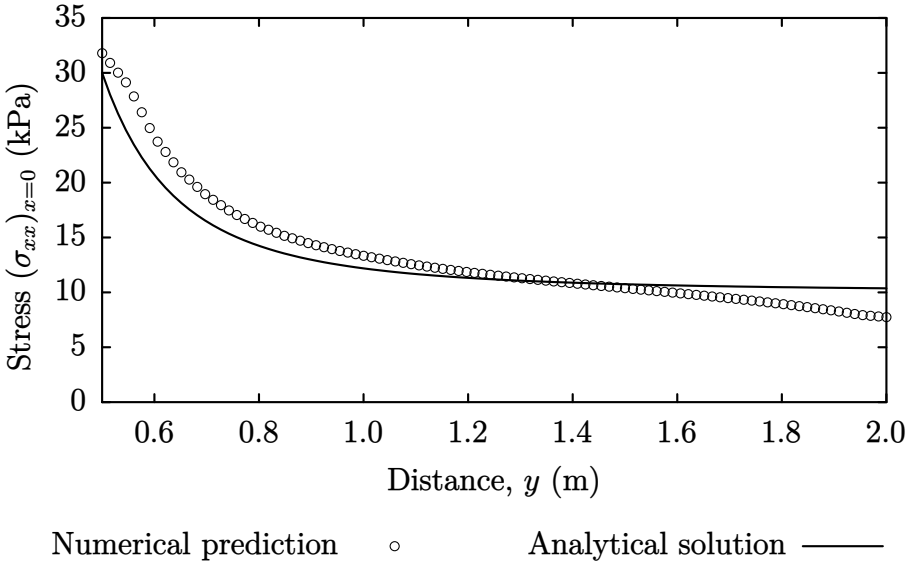


Figure 2.20: Normal stress along the vertical symmetry $(\sigma_{xx})_{x=0}$

2.2.4.1 Increasing mesh resolution

Increase the mesh resolution in each of the x and y directions. Use mapFields to map the final coarse mesh results from section 2.2.3 to the initial conditions for the fine mesh.

2.2.4.2 Introducing mesh grading

Grade the mesh so that the cells near the hole are finer than those away from the hole. Design the mesh so that the ratio of sizes between adjacent cells is no more than 1.1 and so that the ratio of cell sizes between blocks is similar to the ratios within blocks. Mesh grading is described in section 2.1.6. Again use mapFields to map the final coarse mesh results from section 2.2.3 to the initial conditions for the graded mesh. Compare the results with those from the analytical solution and previous calculations. Can this solution be improved upon using the same number of cells with a different solution?

2.2.4.3 Changing the plate size

The analytical solution is for an infinitely large plate with a finite sized hole in it. Therefore this solution is not completely accurate for a finite sized plate. To estimate the error, increase the plate size while maintaining the hole size at the same value.

2.3 Breaking of a dam

In this tutorial we shall solve a problem of a simplified dam break in 2 dimensions using the interFoam solver. The feature of the problem is a transient flow of two fluids separated by a sharp interface, or free surface. The two-phase algorithm in interFoam is based on the volume of fluid (VOF) method in which a phase transport equation is used to determine the relative volume fraction of the two phases, or phase fraction α, in each computational cell. Physical properties are calculated as weighted averages based on this fraction. The nature of the VOF method means that an interface between the phases is not explicitly computed, but rather emerges as a property of the phase fraction field. Since the phase

fraction can have any value between 0 and 1, the interface is never sharply defined, but occupies a volume around the region where a sharp interface should exist.

The test setup consists of a column of water at rest located behind a membrane on the left side of a tank. At time $t = 0$ s, the membrane is removed and the column of water collapses. During the collapse, the water impacts an obstacle at the bottom of the tank and creates a complicated flow structure, including several captured pockets of air. The geometry and the initial setup is shown in Figure 2.21.

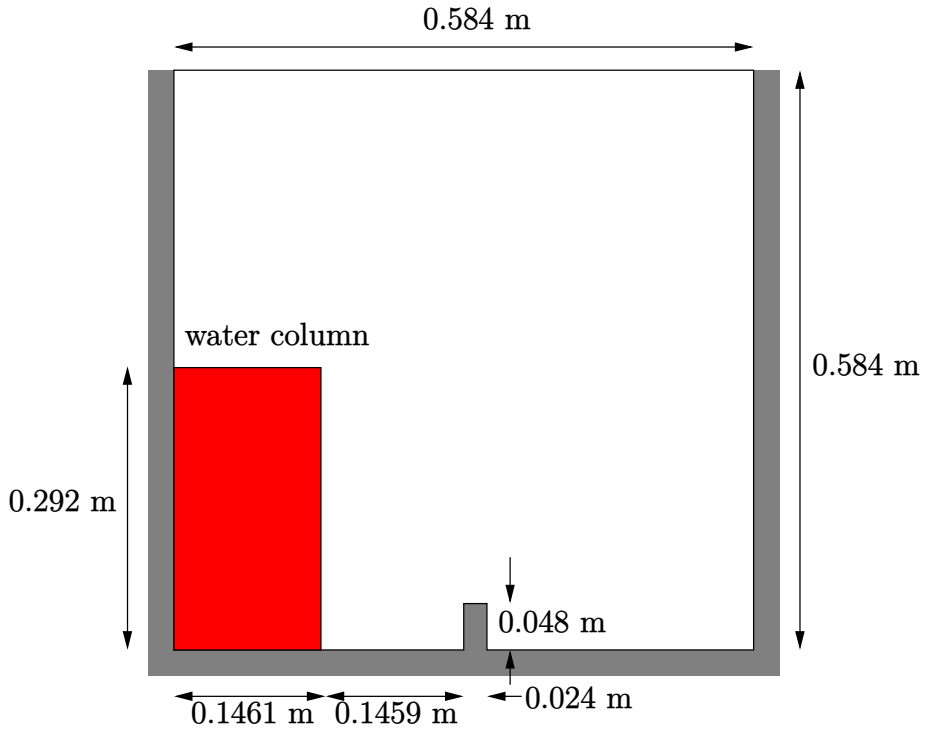


Figure 2.21: Geometry of the dam break.

2.3.1 Mesh generation

The user should go to their *run* directory and copy the *damBreak* case from the *$FOAM_TUTORIALS/multiphase/interFoam/laminar/damBreak* directory, *i.e.*

```
run
cp -r $FOAM_TUTORIALS/multiphase/interFoam/laminar/damBreak/damBreak .
```

Go into the damBreak case directory and generate the mesh running blockMesh as described previously. The damBreak mesh consist of 5 blocks; the *blockMeshDict* entries are given below.

```
16  convertToMeters 0.146;
17
18  vertices
19  (
20      (0 0 0)
21      (2 0 0)
22      (2.16438 0 0)
23      (4 0 0)
24      (0 0.32876 0)
25      (2 0.32876 0)
26      (2.16438 0.32876 0)
27      (4 0.32876 0)
28      (0 4 0)
```

```
29      (2 4 0)
30      (2.16438 4 0)
31      (4 4 0)
32      (0 0 0.1)
33      (2 0 0.1)
34      (2.16438 0 0.1)
35      (4 0 0.1)
36      (0 0.32876 0.1)
37      (2 0.32876 0.1)
38      (2.16438 0.32876 0.1)
39      (4 0.32876 0.1)
40      (0 4 0.1)
41      (2 4 0.1)
42      (2.16438 4 0.1)
43      (4 4 0.1)
44  );
45
46  blocks
47  (
48      hex (0 1 5 4 12 13 17 16) (23 8 1) simpleGrading (1 1 1)
49      hex (2 3 7 6 14 15 19 18) (19 8 1) simpleGrading (1 1 1)
50      hex (4 5 9 8 16 17 21 20) (23 42 1) simpleGrading (1 1 1)
51      hex (5 6 10 9 17 18 22 21) (4 42 1) simpleGrading (1 1 1)
52      hex (6 7 11 10 18 19 23 22) (19 42 1) simpleGrading (1 1 1)
53  );
54
55  defaultPatch
56  {
57      type empty;
58  }
59
60  boundary
61  (
62      leftWall
63      {
64          type wall;
65          faces
66          (
67              (0 12 16 4)
68              (4 16 20 8)
69          );
70      }
71      rightWall
72      {
73          type wall;
74          faces
75          (
76              (7 19 15 3)
77              (11 23 19 7)
78          );
79      }
80      lowerWall
81      {
82          type wall;
83          faces
84          (
85              (0 1 13 12)
86              (1 5 17 13)
87              (5 6 18 17)
88              (2 14 18 6)
89              (2 3 15 14)
90          );
91      }
92      atmosphere
93      {
94          type patch;
95          faces
96          (
97              (8 20 21 9)
98              (9 21 22 10)
99              (10 22 23 11)
100         );
101     }
102 );
103
104
105 // ************************************************************************* //
```

2.3.2 Boundary conditions

The user can examine the boundary geometry generated by blockMesh by viewing the *boundary* file in the *constant/polyMesh* directory. The file contains a list of 5 boundary patches: `leftWall`, `rightWall`, `lowerWall`, `atmosphere` and `defaultFaces`.

The user should notice the `type` of the patches. Firstly, the `atmosphere` is a standard patch, *i.e.* has no special attributes, merely an entity on which boundary conditions can be specified. Then, the `defaultFaces` patch is formed of **block faces that are omitted** from the `boundary` sub-dictionary in the *blockMeshDict* file. Those block faces form a patch whose properties are specified in a `defaultPatch` sub-dictionary in in the *blockMeshDict* file. In this case, the default `type` is set to empty since the patch normal is in the direction we will not solve in this 2D case.

The `leftWall`, `rightWall` and `lowerWall` patches are each a wall. Like the generic patch, the wall type contains no geometric or topological information about the mesh and only differs from the plain patch in that it identifies the patch as a wall, should an application need to know, *e.g.* to apply special wall surface modelling. For example, the interFoam solver includes modelling of surface tension and can include wall adhesion at the contact point between the interface and wall surface. Wall adhesion models can be applied through a special boundary condition on the `alpha` (α) field, *e.g.* the constantAlphaContactAngle boundary condition, which requires the user to specify a static contact angle, `theta0`.

In this tutorial we would like to ignore surface tension effects between the wall and interface. We can do this by setting the static contact angle, $\theta_0 = 90°$. However, rather than using the constantAlphaContactAngle boundary condition, the simpler zeroGradient can be applied to `alpha` on the walls.

The `top` boundary is free to the atmosphere so needs to permit both outflow and inflow according to the internal flow. We therefore use a combination of boundary conditions for pressure and velocity that does this while maintaining stability. They are:

- prghTotalPressure, applied to the pressure field, without the hydrostatic component, $p_{\rho gh}$, which is a fixedValue condition calculated from a specified total pressure `p0` and local velocity `U`;

- pressureInletOutletVelocity, applied to velocity **U**, which sets zeroGradient on all components of **U**, except where there is inflow, in which case a fixedValue condition is applied to the *tangential* component;

- inletOutlet applied to other fields, which is a zeroGradient condition when flow outwards, fixedValue when flow is inwards.

At all wall boundaries, the fixedFluxPressure boundary condition is applied to the pressure field, which adjusts the pressure gradient so that the boundary flux matches the velocity boundary condition for solvers that include body forces such as gravity and surface tension.

The `defaultFaces` patch representing the front and back planes of the 2D problem, is, as usual, an empty type.

2.3.3 Phases

The fluid phases are specified in the *phaseProperties* file in the *constant* directory as follows:

```
16
17  phases          (water air);
18
19  sigma           0.07;
20
21
22  // ************************************************************************* //
```

It lists two phases, `water` and `air`. Equations for phase fraction are solved for the phases in the list, **except the last phase** listed, *i.e.* `air` in this case. Since there are only two phases, only one phase fraction equation is solved in this case, for the water phase fraction α_{water}, specified in the file *alpha.water* in the *0* directory.

The *phaseProperties* file also contains an entry for the surface tension between the two phases, specified by the keyword `sigma` in units $\text{N}\,\text{m}^{-1}$.

2.3.4 Setting initial fields

Unlike the previous cases, we shall now specify a non-uniform initial condition for the phase fraction α_{water} where

$$\alpha_{\text{water}} = \begin{cases} 1 & \text{for the water phase} \\ 0 & \text{for the air phase} \end{cases} \tag{2.15}$$

This will be done by running the setFields utility. It requires a *setFieldsDict* dictionary, located in the *system* directory, whose entries for this case are shown below.

```
16
17  defaultFieldValues
18  (
19      volScalarFieldValue alpha.water 0
20  );
21
22  regions
23  (
24      boxToCell
25      {
26          box (0 0 -1) (0.1461 0.292 1);
27          fieldValues
28          (
29              volScalarFieldValue alpha.water 1
30          );
31      }
32  );
33
34
35  // ************************************************************************* //
```

The `defaultFieldValues` sets the default value of the fields, *i.e.* the value the field takes unless specified otherwise in the `regions` sub-dictionary. That sub-dictionary contains a list of subdictionaries containing `fieldValues` that override the defaults in a specified region. The region creates a set of points, cells or faces based on some topological constraint. Here, `boxToCell` creates a bounding box within a vector minimum and maximum to define the set of cells of the water region. The phase fraction α_{water} is defined as 1 in this region.

The setFields utility reads fields from file and, after re-calculating those fields, will write them back to file. In the damBreak tutorial, the `alpha.water` field is initially stored as a backup named `alpha.water.orig`. A field file with the `.orig` extension is read in when the actual file does not exist, so setFields will read `alpha.water.orig` but write the resulting output to `alpha.water` (or `alpha.water.gz` if compression is switched on). This way the original file is not overwritten, so can be reused.

The user should therefore execute setFields like any other utility by:

```
setFields
```

Using paraFoam, check that the initial `alpha.water` field corresponds to the desired distribution as in Figure 2.22.

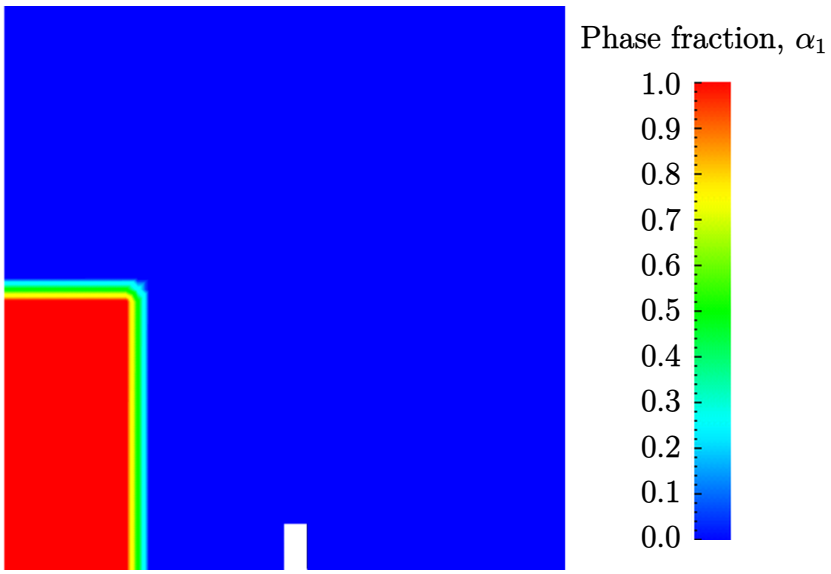


Figure 2.22: Initial conditions for phase fraction `alpha.water`.

2.3.5 Fluid properties

The physical properties for the air and water phases are specified in *physicalProperties.air* and *physicalProperties.water* files, respectively, in the *constant* directory. In each file, the viscosity model is selected by the `viscosityModel` keyword. The `constant` model is selected for both fluids, *i.e.* representing them as Newtonian fluids with a kinematic viscosity specified by the `nu` keyword in units $\mathrm{N\,m^{-1}}$. Other non-Newtonian viscosity models are described in section 7.3. The density of each fluid is specified by the keyword `rho` in units $\mathrm{kg\,m^{-3}}$. The *physicalProperties.air* file is shown below as an example:

```
16
17  viscosityModel  constant;
18
19  nu              1.48e-05;
20
21  rho             1;
22
23
24  // ************************************************************************* //
```

Gravitational acceleration is uniform across the domain and is specified in a file named *g* in the *constant* directory. Unlike a normal field file, *e.g. U* and *p*, *g* is a uniformDimensionedVectorField and so simply contains a set of `dimensions` and a `value` that represents $(0, 9.81, 0)$ $\mathrm{m\,s^{-2}}$ for this tutorial:

```
16
17  dimensions      [0 1 -2 0 0 0 0];
18  value           (0 -9.81 0);
19
20
21  // ************************************************************************* //
```

2.3.6 Turbulence modelling

As in the cavity example, the choice of turbulence modelling method is selectable at run-time through the `simulationType` keyword in *momentumTransport* dictionary. In this example, we wish to run without turbulence modelling so we set `laminar`:

```
16
17  simulationType  laminar;
18
19
20  // ************************************************************************* //
```

2.3.7 Time step control

Time step control is an important issue in transient simulation and the surface-tracking algorithm in interface-capturing solvers. The interFoam solver uses the multidimensional universal limiter for explicit solution (MULES) method, created by Henry Weller, to maintain boundedness of the phase fraction. The Courant number Co needs to be limited depending on the choice of algorithm: with the *explicit* MULES algorithm, an upper limit of $Co \approx 0.25$ for stability is typical in the region of the interface; but with *semi-implicit* MULES, specified by the `MULESCorr` keyword in the *fvSolution* file, there is really no upper limit in Co for stability, but instead the level is determined by requirements of temporal accuracy.

In general it is difficult to specify a fixed time-step to satisfy the Co criterion, so interFoam offers automatic adjustment of the time step as standard in the *controlDict*. The user should specify `adjustTimeStep` to be `on` and the the maximum Co for the phase fields, `maxAlphaCo`, and other fields, `maxCo`, to be 1.0. The upper limit on time step `maxDeltaT` can be set to a value that will not be exceeded in this simulation, *e.g.* 1.0.

By using automatic time step control, the steps themselves are never rounded to a convenient value. Consequently if we request that OpenFOAM saves results at a fixed number of time step intervals, the times at which results are saved are somewhat arbitrary. However even with automatic time step adjustment, OpenFOAM allows the user to specify that results are written at fixed times; in this case OpenFOAM forces the automatic time stepping procedure to adjust time steps so that it 'hits' on the exact times specified for write output. The user selects this with the `adjustableRunTime` option for `writeControl` in the *controlDict* dictionary. The *controlDict* dictionary entries should be:

```
16
17  application     interFoam;
18
19  startFrom       startTime;
20
21  startTime       0;
22
23  stopAt          endTime;
24
25  endTime         1;
26
27  deltaT          0.001;
28
29  writeControl    adjustableRunTime;
30
31  writeInterval   0.05;
32
33  purgeWrite      0;
34
35  writeFormat     binary;
36
37  writePrecision  6;
38
39  writeCompression off;
40
41  timeFormat      general;
```

```
42
43  timePrecision   6;
44
45  runTimeModifiable yes;
46
47  adjustTimeStep  yes;
48
49  maxCo           1;
50  maxAlphaCo      1;
51
52  maxDeltaT       1;
53
54
55  // ************************************************************************* //
```

2.3.8 Discretisation schemes

The MULES method, used by the interFoam solver, maintains boundedness of the phase fraction independently of the underlying numerical scheme, mesh structure, *etc.* The choice of schemes for convection are therfore not restricted to those that are strongly stable or bounded, *e.g.* upwind differencing.

The convection schemes settings are made in the *divSchemes* sub-dictionary of the *fvSchemes* dictionary. In this example, the convection term in the momentum equation ($\nabla \bullet (\rho \mathbf{U}\mathbf{U})$), denoted by the `div(rhoPhi,U)` keyword, uses `Gauss linearUpwind grad(U)` to produce good accuracy. Here, we have opted for best stability with $\phi = 1.0$. The $\nabla \bullet (\mathbf{U}\alpha)$ term, represented by the `div(phi,alpha)` keyword uses a bespoke `interfaceCompression` scheme where the specified coefficient is a factor that controls the compression of the interface where: 0 corresponds to no compression; 1 corresponds to conservative compression; and, anything larger than 1, relates to enhanced compression of the interface. We generally adopt a value of 1.0 which is employed in this example.

The other discretised terms use commonly employed schemes so that the *fvSchemes* dictionary entries should therefore be:

```
16
17  ddtSchemes
18  {
19      default         Euler;
20  }
21
22  gradSchemes
23  {
24      default         Gauss linear;
25  }
26
27  divSchemes
28  {
29      div(rhoPhi,U)  Gauss linearUpwind grad(U);
30      div(phi,alpha)  Gauss interfaceCompression vanLeer 1;
31      div(((rho*nuEff)*dev2(T(grad(U))))) Gauss linear;
32  }
33
34  laplacianSchemes
35  {
36      default         Gauss linear corrected;
37  }
38
39  interpolationSchemes
40  {
41      default         linear;
42  }
43
44  snGradSchemes
45  {
46      default         corrected;
47  }
48
49
50  // ************************************************************************* //
```

2.3.9 Linear-solver control

In the *fvSolution* file, the *alpha.water* sub-dictionary in *solvers* contains elements that are specific to interFoam. Of particular interest is the `nAlphaCorr` keyword which controls the number of iterations of the phase fraction equation within a solution step. The iteration is used to overcome nonlinearities in the advection which are present in this case due to the `interfaceCompression` scheme.

2.3.10 Running the code

Running of the code has been described in detail in previous tutorials. Try the following, that uses tee, a command that enables output to be written to both standard output and files:

```
cd $FOAM_RUN/damBreak
interFoam | tee log
```

The code will now be run interactively, with a copy of output stored in the *log* file.

2.3.11 Post-processing

Post-processing of the results can now be done in the usual way. The user can monitor the development of the phase fraction `alpha.water` in time, *e.g.* see Figure 2.23.

2.3.12 Running in parallel

The results from the previous example are generated using a fairly coarse mesh. We now wish to increase the mesh resolution and re-run the case. The new case will typically take a few hours to run with a single processor so, should the user have access to multiple processors, we can demonstrate the parallel processing capability of OpenFOAM.

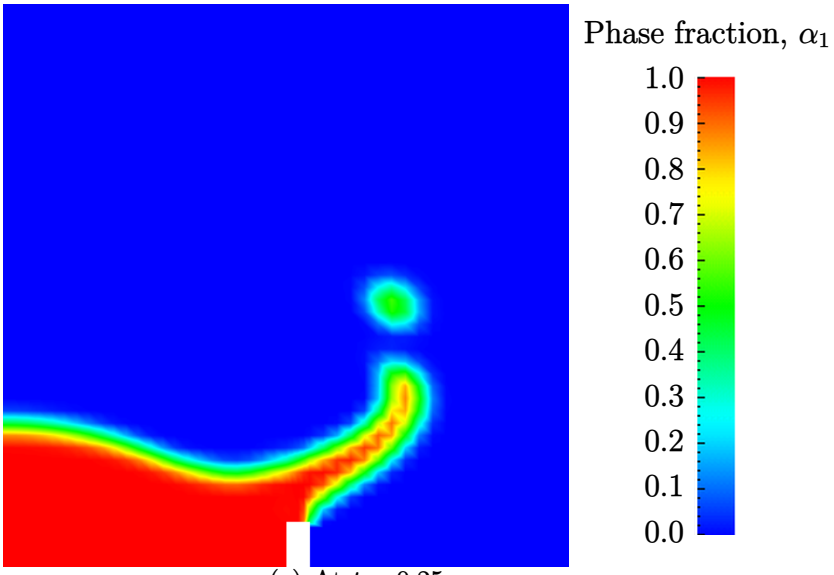
The user should first clone the damBreak case, *e.g.* by

```
run
foamCloneCase damBreak damBreakFine
```

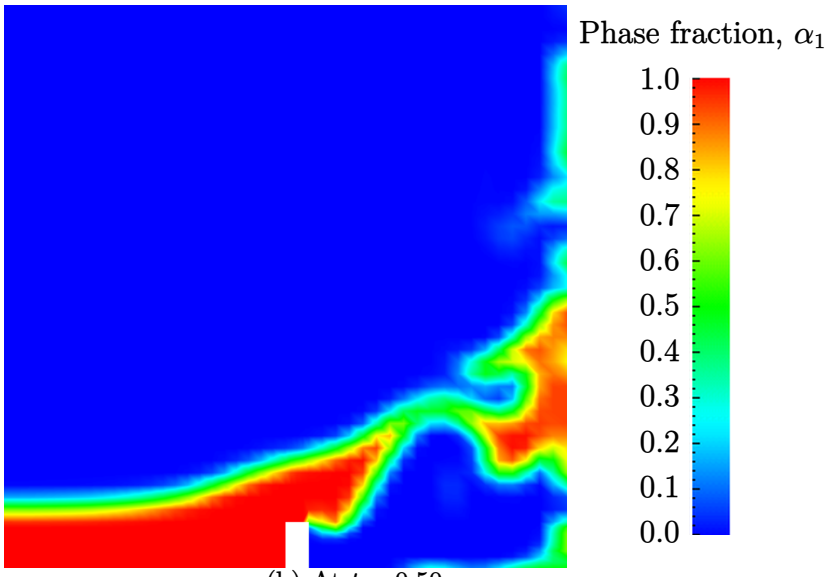
Enter the new case directory and change the `blocks` description in the *blockMeshDict* dictionary to

```
blocks
(
    hex (0 1 5 4 12 13 17 16) (46 10 1) simpleGrading (1 1 1)
    hex (2 3 7 6 14 15 19 18) (40 10 1) simpleGrading (1 1 1)
    hex (4 5 9 8 16 17 21 20) (46 76 1) simpleGrading (1 2 1)
    hex (5 6 10 9 17 18 22 21) (4 76 1) simpleGrading (1 2 1)
    hex (6 7 11 10 18 19 23 22) (40 76 1) simpleGrading (1 2 1)
);
```

Here, the entry is presented as printed from the *blockMeshDict* file; in short the user must change the mesh densities, *e.g.* the `46 10 1` entry, and some of the mesh grading entries to `1 2 1`. Once the dictionary is correct, generate the mesh by running blockMesh.



(a) At $t = 0.25$ s.



(b) At $t = 0.50$ s.

Figure 2.23: Snapshots of phase α.

As the mesh has now changed from the damBreak example, the user must re-initialise the phase field `alpha.water` in the *0* time directory since it contains a number of elements that is inconsistent with the new mesh. Note that there is no need to change the `U` and `p_rgh` fields since they are specified as `uniform` which is independent of the number of elements in the field. We wish to initialise the field with a sharp interface, *i.e.* it elements would have $\alpha = 1$ or $\alpha = 0$. Updating the field with mapFields may produce interpolated values $0 < \alpha < 1$ at the interface, so it is better to rerun the setFields utility.

The mesh size is now inconsistent with the number of elements in the *alpha.water* file in the *0* directory, so the user must delete that file so that the original *alpha.water.orig* file is used instead.

```
rm 0/alpha.water
setFields
```

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 first step required to run a parallel case is therefore to decompose the domain using the decomposePar utility. There is a dictionary associated with decomposePar named *decomposeParDict* which is located in the *system* directory of the tutorial case. Also, sample dictionaries can be found within the *etc* directory in the OpenFOAM installation, which can be copied to the case directory by running the foamGet script, *e.g.* (you do not need to do this):

```
foamGet decomposeParDict
```

The first entry is `numberOfSubdomains` which specifies the number of subdomains into which the case will be decomposed, usually corresponding to the number of processors available for the case.

In this tutorial, the `method` of decomposition should be `simple` and the corresponding `simpleCoeffs` should be edited according to the following criteria. The domain is split into pieces, or subdomains, in the x, y and z directions, the number of subdomains in each direction being given by the vector $\mathbf{n}$. As this geometry is 2 dimensional, the 3rd direction, z, cannot be split, hence n_z must equal 1. The n_x and n_y components of $\mathbf{n}$ split the domain in the x and y directions and must be specified so that the number of subdomains specified by n_x and n_y equals the specified `numberOfSubdomains`, *i.e.* $n_x n_y =$ `numberOfSubdomains`. It is beneficial to keep the number of cell faces adjoining the subdomains to a minimum so, for a square geometry, it is best to keep the split between the x and y directions should be fairly even.

For example, let us assume we wish to run on 4 processors. We would set `numberOfSubdomains` to 4 and $\mathbf{n} = (2, 2, 1)$. The user should run decomposePar with:

```
decomposePar
```

The terminal output shows that the decomposition is distributed fairly even between the processors.

The user should consult section 3.4 for details of how to run a case in parallel; in this tutorial we merely present an example of running in parallel. We use the openMPI implementation of the standard message-passing interface (MPI). As a test here, the user can run in parallel on a single node, the local host only, by typing:

```
mpirun -np 4 interFoam -parallel > log &
```

The user may run on more nodes over a network by creating a file that lists the host names of the machines on which the case is to be run as described in section 3.4.3. The case should run in the background and the user can follow its progress by monitoring the *log* file as usual.

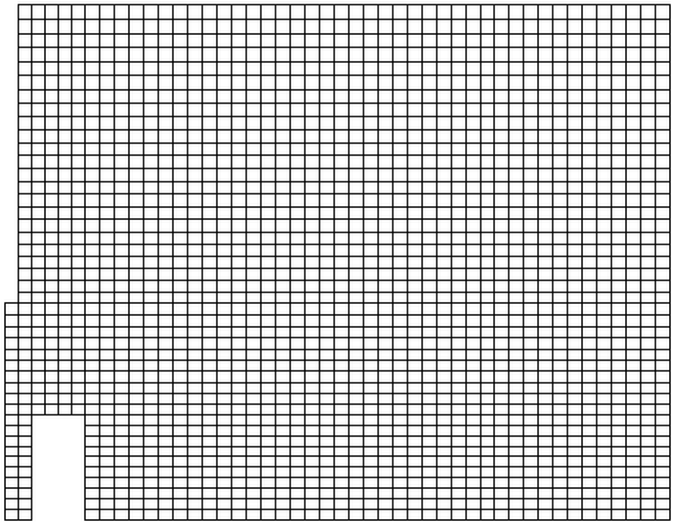


Figure 2.24: Mesh of processor 2 in parallel processed case.

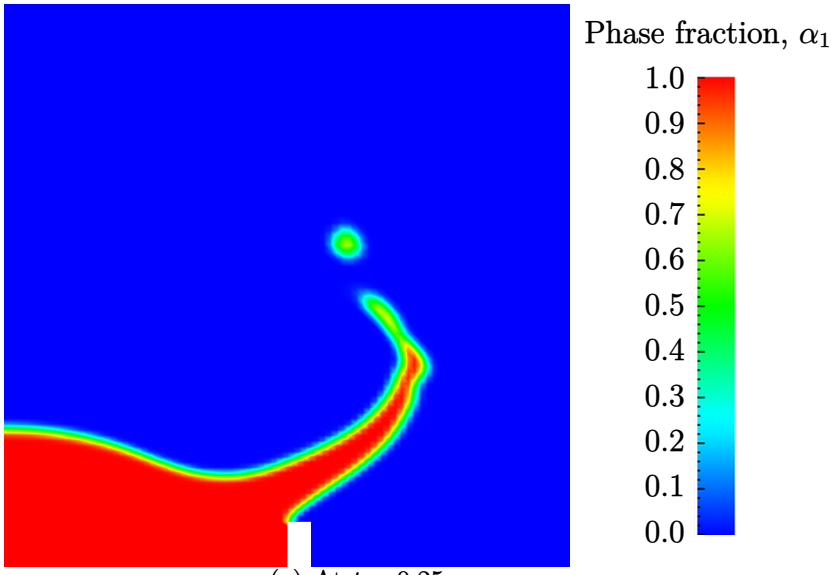
2.3.13 Post-processing a case run in parallel

Once the case has completed running, the decomposed fields and mesh can be reassembled for post-processing using the reconstructPar utility. Simply execute it from the command line. The results from the fine mesh are shown in Figure 2.25. The user can see that the resolution of interface has improved significantly compared to the coarse mesh.

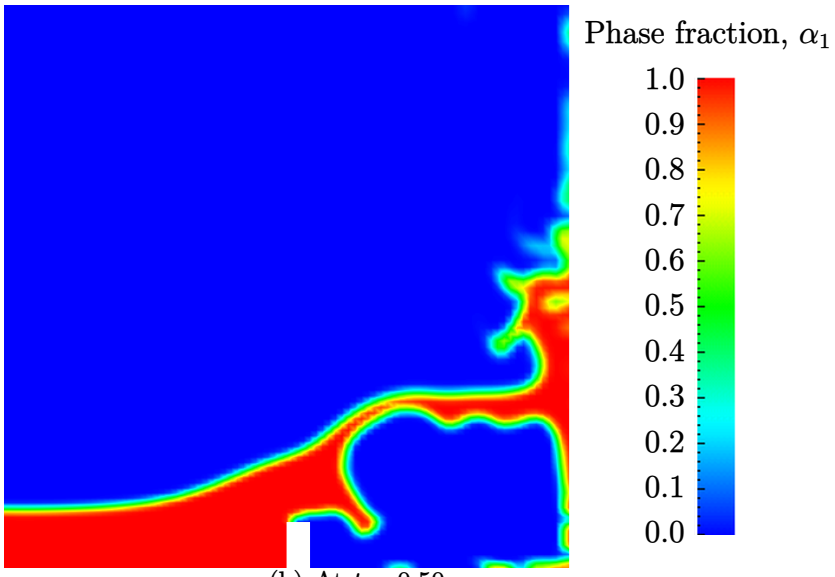
The user may also post-process an individual region of the decomposed domain individually by simply treating the individual processor directory as a case in its own right. For example if the user starts paraFoam by

```
paraFoam -case processor1
```

then processor1 will appear as a case module in ParaView. Figure 2.24 shows the mesh from processor 1 following the decomposition of the domain using the `simple` method.



(a) At $t = 0.25$ s.



(b) At $t = 0.50$ s.

Figure 2.25: Snapshots of phase α with refined mesh.

Chapter 3

Applications and libraries

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 The programming language of OpenFOAM

In order to understand the way in which the OpenFOAM library works, some background knowledge of C++, the base language of OpenFOAM, is required; the necessary information will be presented in this chapter. Before doing so, it is worthwhile addressing the concept of language in general terms to explain some of the ideas behind object-oriented programming and our choice of C++ as the main programming language of OpenFOAM.

3.1.1 Language in general

The success of verbal language and mathematics is based on efficiency, especially in expressing abstract concepts. For example, in fluid flow, we use the term "velocity field", which has meaning without any reference to the nature of the flow or any specific velocity data. The term encapsulates the idea of movement with direction and magnitude and relates to other physical properties. In mathematics, we can represent velocity field by a single symbol, *e.g.* $\mathbf{U}$, and express certain concepts using symbols, *e.g.* "the field of velocity magnitude" by $|\mathbf{U}|$. The advantage of mathematics over verbal language is its greater efficiency, making it possible to express complex concepts with extreme clarity.

The problems that we wish to solve in continuum mechanics are not presented in terms of intrinsic entities, or types, known to a computer, *e.g.* bits, bytes, integers. They

are usually presented first in verbal language, then as partial differential equations in 3 dimensions of space and time. The equations contain the following concepts: scalars, vectors, tensors, and fields thereof; tensor algebra; tensor calculus; dimensional units. The solution to these equations involves discretisation procedures, matrices, solvers, and solution algorithms.

3.1.2 Object-orientation and C++

Progamming languages that are object-oriented, such as C++, provide the mechanism — *classes* — to declare types and associated operations that are part of the verbal and mathematical languages used in science and engineering. Our velocity field introduced earlier can be represented in programming code by the symbol `U` and "the field of velocity magnitude" can be `mag(U)`. The velocity is a vector field for which there should exist, in an object-oriented code, a vectorField class. The velocity field `U` would then be an instance, or *object*, of the vectorField class; hence the term object-oriented.

The clarity of having objects in programming that represent physical objects and abstract entities should not be underestimated. The class structure concentrates code development to contained regions of the code, *i.e.* the classes themselves, thereby making the code easier to manage. New classes can be derived or inherit properties from other classes, *e.g.* the vectorField can be derived from a vector class and a Field class. C++ provides the mechanism of *template classes* such that the template class Field<Type> can represent a field of any <Type>, *e.g.*scalar, vector, tensor. The general features of the template class are passed on to any class created from the template. Templating and inheritance reduce duplication of code and create class hierarchies that impose an overall structure on the code.

3.1.3 Equation representation

A central theme of the OpenFOAM design is that the solver applications, written using the OpenFOAM classes, have a syntax that closely resembles the partial differential equations being solved. For example the equation

$$\frac{\partial \rho \mathbf{U}}{\partial t} + \nabla \bullet \phi \mathbf{U} - \nabla \bullet \mu \nabla \mathbf{U} = -\nabla p$$

is represented by the code

```
solve
(
    fvm::ddt(rho, U)
  + fvm::div(phi, U)
  - fvm::laplacian(mu, U)
    ==
  - fvc::grad(p)
);
```

This and other requirements demand that the principal programming language of OpenFOAM has object-oriented features such as inheritance, template classes, virtual functions and operator overloading. These features are not available in many languages that purport to be object-orientated but actually have very limited object-orientated capability, such

as FORTRAN-90. C++, however, possesses all these features while having the additional advantage that it is widely used with a standard specification so that reliable compilers are available that produce efficient executables. It is therefore the primary language of OpenFOAM.

3.1.4 Solver codes

Solver codes are largely procedural since they are a close representation of solution algorithms and equations, which are themselves procedural in nature. Users do not need a deep knowledge of object-orientation and C++ programming to write a solver but should know the principles behind object-orientation and classes, and to have a basic knowledge of some C++ code syntax. An understanding of the underlying equations, models and solution method and algorithms is far more important.

There is often little need for a user to immerse themselves in the code of any of the OpenFOAM classes. The essence of object-orientation is that the user should not have to go through the code of each class they use; merely the knowledge of the class' existence and its functionality are sufficient to use the class. A description of each class, its functions *etc.* is supplied with the OpenFOAM distribution in HTML documentation generated with Doxygen at `https://cpp.openfoam.org`

3.2 Compiling applications and libraries

Compilation is an integral part of application development that requires careful management since every piece of code requires its own set instructions to access dependent components of the OpenFOAM library. In UNIX/Linux systems these instructions are often organised and delivered to the compiler using the standard UNIXmake utility. OpenFOAM uses its own wmake compilation script that is based on make but is considerably more versatile and easier to use (wmake can be used on any code, not only the OpenFOAM library). To understand the compilation process, we first need to explain certain aspects of C++ and its file structure, shown schematically in Figure 3.1. A class is defined through a set of instructions such as object construction, data storage and class member functions. The file that defines these functions — the class *definition* — takes a *.C* extension, *e.g.* a class nc would be written in the file *nc.C*. This file can be compiled independently of other code into a binary executable library file known as a shared object library with the *.so* file extension, *i.e.nc.so*. When compiling a piece of code, say *newApp.C*, that uses the nc class, *nc.C* need not be recompiled, rather *newApp.C* calls the *nc.so* library at runtime. This is known as *dynamic linking*.

3.2.1 Header *.H* files

As a means of checking errors, the piece of code being compiled must know that the classes it uses and the operations they perform actually exist. Therefore each class requires a class *declaration*, contained in a header file with a *.H* file extension, *e.g. nc.H*, that includes the names of the class and its functions. This file is included at the beginning of any piece of code using the class, using the `#include` directive described below, including the class declaration code itself. Any piece of *.C* code can resource any number of classes and must begin by including all the *.H* files required to declare these classes. Those classes in turn can resource other classes and so also begin by including the relevant *.H* files. By searching recursively down the class hierarchy we can produce a complete list of header

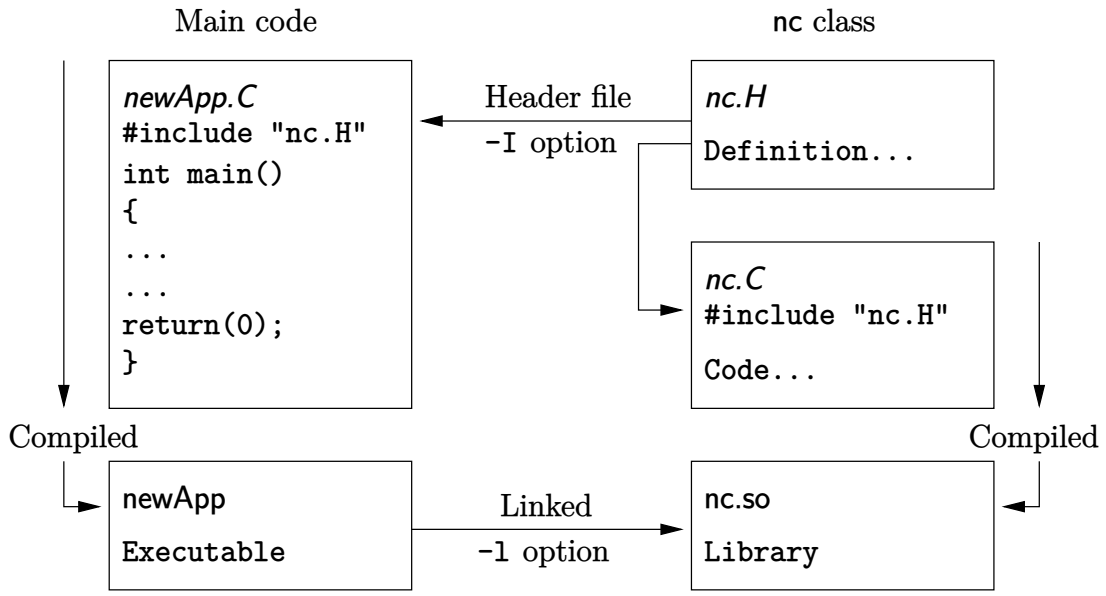


Figure 3.1: Header files, source files, compilation and linking

files for all the classes on which the top level *.C* code ultimately depends; these *.H* files are known as the *dependencies*. With a dependency list, a compiler can check whether the source files have been updated since their last compilation and selectively compile only those that need to be.

Header files are included in the code using the `# include` directive, *e.g.*

```
# include "otherHeader.H";
```

This causes the compiler to suspend reading from the current file, to read the included file. This mechanism allows any self-contained piece of code to be put into a header file and included at the relevant location in the main code in order to improve code readability. For example, in most OpenFOAM applications the code for creating fields and reading field input data is included in a file *createFields.H* which is called at the beginning of the code. In this way, header files are not solely used as class declarations.

It is wmake that performs the task of maintaining file dependency lists amongst other functions listed below.

- Automatic generation and maintenance of file dependency lists, *i.e.* lists of files which are included in the source files and hence on which they depend.
- Multi-platform compilation and linkage, handled through appropriate directory structure.
- Multi-language compilation and linkage, *e.g.* C, C++, Java.
- Multi-option compilation and linkage, *e.g.* debug, optimised, parallel and profiling.
- Support for source code generation programs, *e.g.* lex, yacc, IDL, MOC.
- Simple syntax for source file lists.
- Automatic creation of source file lists for new codes.
- Simple handling of multiple shared or static libraries.

- Extensible to new machine types.
- Extremely portable, works on any machine with: `make`; `sh`, `ksh` or `csh`; `lex`, `cc`.

3.2.2 Compiling with wmake

OpenFOAM applications are organised using a standard convention that the source code of each application is placed in a directory whose name is that of the application. The top level source file then takes the application name with the *.C* extension. For example, the source code for an application called newApp would reside is a directory *newApp* and the top level file would be *newApp.C* as shown in Figure 3.2. wmake then requires the

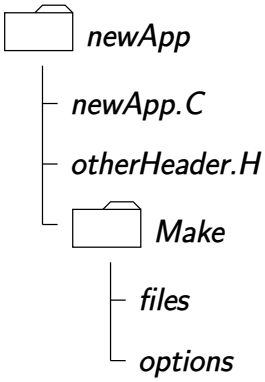


Figure 3.2: Directory structure for an application

directory must contain a *Make* subdirectory containing 2 files, *options* and *files*, that are described in the following sections.

3.2.2.1 Including headers

The compiler searches for the included header files in the following order, specified with the `-I` option in wmake:

1. the *$WM_PROJECT_DIR/src/OpenFOAM/lnInclude* directory;
2. a local *lnInclude* directory, *i.e.newApp/lnInclude*;
3. the local directory, *i.e.newApp*;
4. platform dependent paths set in files in the *$WM_PROJECT_DIR/wmake/rules/-$WM_ARCH/* directory, *e.g./usr/X11/include* and *$(MPICH_ARCH_PATH)/include*;
5. other directories specified explicitly in the *Make/options* file with the `-I` option.

The *Make/options* file contains the full directory paths to locate header files using the syntax:

```
EXE_INC = \
    -I<directoryPath1> \
    -I<directoryPath2> \
    ...                \
    -I<directoryPathN>
```

Notice first that the directory names are preceeded by the `-I` flag and that the syntax uses the `\` to continue the `EXE_INC` across several lines, with no `\` after the final entry.

3.2.2.2 Linking to libraries

The compiler links to shared object library files in the following directory **paths**, specified with the `-L` option in wmake:

1. the *$FOAM_LIBBIN* directory;
2. platform dependent paths set in files in the *$WM_DIR/rules/$WM_ARCH/* directory, *e.g./usr/X11/lib* and *$(MPICH_ARCH_PATH)/lib*;
3. other directories specified in the *Make/options* file.

The actual library **files** to be linked must be specified using the `-l` option and removing the `lib` prefix and `.so` extension from the library file name, *e.g.* libnew.so is included with the flag `-lnew`. By default, wmake loads the following libraries:

1. the libOpenFOAM.so library from the *$FOAM_LIBBIN* directory;
2. platform dependent libraries specified in set in files in the *$WM_DIR/rules/$WM_ARCH/* directory, *e.g.* libm.so from */usr/X11/lib* and liblam.so from *$(LAM_ARCH_PATH)/lib*;
3. other libraries specified in the *Make/options* file.

The *Make/options* file contains the full directory paths and library names using the syntax:

```
EXE_LIBS = \
    -L<libraryPath> \
    -l<library1>     \
    -l<library2>     \
    ...              \
    -l<libraryN>
```

To summarise: the directory paths are preceeded by the `-L` flag, the library names are preceeded by the `-l` flag.

3.2.2.3 Source files to be compiled

The compiler requires a list of *.C* source files that must be compiled. The list must contain the main *.C* file but also any other source files that are created for the specific application but are not included in a class library. For example, users may create a new class or some new functionality to an existing class for a particular application. The full list of *.C* source files must be included in the *Make/files* file. For many applications the list only includes the name of the main *.C* file, *e.g. newApp.C* in the case of our earlier example.

The *Make/files* file also includes a full path and name of the compiled executable, specified by the `EXE =` syntax. Standard convention stipulates the name is that of the application, *i.e.*newApp in our example. The OpenFOAM release offers two useful choices for path: standard release applications are stored in *$FOAM_APPBIN*; applications developed by the user are stored in *$FOAM_USER_APPBIN*.

If the user is developing their own applications, we recommend they create an *applications* subdirectory in their *$WM_PROJECT_USER_DIR* directory containing the source code for personal OpenFOAM applications. As with standard applications, the source code for each OpenFOAM application should be stored within its own directory. The only difference between a user application and one from the standard release is that the *Make/files* file should specify that the user's executables are written into their *$FOAM_USER_APPBIN* directory. The *Make/files* file for our example would appear as follows:

```
newApp.C

EXE = $(FOAM_USER_APPBIN)/newApp
```

3.2.2.4 Running wmake

The wmake script is generally executed by typing:

```
wmake <optionalDirectory>
```

The <optionalDirectory> is the directory path of the application that is being compiled. Typically, wmake is executed from within the directory of the application being compiled, in which case <optionalDirectory> can be omitted.

3.2.2.5 wmake environment variables

For information, the general environment variable settings used by wmake are listed below:

- $WM_PROJECT_INST_DIR: full path to the installation directory, *e.g.$HOME/-OpenFOAM*.
- $WM_PROJECT: name of the project being compiled, *i.e.* `OpenFOAM`.
- $WM_PROJECT_VERSION: version of the project being compiled, *i.e.* `10`.
- $WM_PROJECT_DIR: full path to the main directory of the OpenFOAM release, *e.g. $HOME/OpenFOAM/OpenFOAM-10*.
- $WM_PROJECT_USER_DIR: full path to the equivalent directory for customised developments in the user account*e.g. $HOME/OpenFOAM/${USER}-10*.
- $WM_THIRD_PARTY_DIR: full path to the directory of *ThirdParty* software, *e.g. $HOME/OpenFOAM/ThirdParty-10*.

The environment variable settings for the compilation with wmake are listed below:

- $WM_ARCH: machine architecture, *e.g.* `linux`, `linux64`, `linuxArm64`, `linuxARM7`, `linuxPPC64`, `linuxPPC64le`.
- $WM_ARCH_OPTION: 32 or 64 bit architecture.
- $WM_COMPILER: compiler being used, *e.g.* `Gcc` = gcc, `Clang` = LLVM Clang
- $WM_COMPILE_OPTION: compilation option, `Debug` = debugging, `Opt` = optimised.
- $WM_COMPILER_TYPE: choice of compiler, `system`, or `ThirdParty`, *i.e.* compiled in *ThirdParty* directory.
- $WM_DIR: full path of the *wmake* directory.
- $WM_LABEL_SIZE: 32 or 64 bit size for labels (integers).

- $WM_LABEL_OPTION: `Int32` or `Int64` compilation of labels.
- $WM_LINK_LANGUAGE: compiler used to link libraries and executables `c++`.
- $WM_MPLIB: parallel communications library, `SYSTEMOPENMPI` = system version of openMPI, alternatives include `OPENMPI`, `SYSTEMMPI`, `MPICH`, `MPICH-GM`, `HPMPI`, `MPI`, `QSMPI`, `INTELMPI` and `SGIMPI`.
- $WM_OPTIONS, *e.g.* `linuxGccDPInt64Opt`, formed by combining $WM_ARCH, $WM_COMPILER, $WM_PRECISION_OPTION, $WM_LABEL_OPTION, and $WM_COMPILE_OPTION.
- $WM_PRECISION_OPTION: floating point precision of the compiled binares, `SP` = single precision, `DP` = double precision.

3.2.3 Removing dependency lists: wclean

On execution, wmake builds a dependency list file with a *.dep* file extension, *e.g. newApp.C.dep* in our example, in a *$WM_OPTIONS* sub-directory of the *Make* directory, *e.g. Make/linuxGccDPInt64Opt*. If the user wishes to remove these files, *e.g.* after making code changes, the user can run the wclean script by typing:

```
wclean <optionalDirectory>
```

Again, the <`optionalDirectory`> is a path to the directory of the application that is being compiled. Typically, wclean is executed from within the directory of the application, in which case the path can be omitted.

3.2.4 Compiling libraries

When compiling a library, there are 2 critical differences in the configuration of the file in the *Make* directory:

- in the *files* file, `EXE =` is replaced by `LIB =` and the target directory for the compiled entity changes from `$FOAM_APPBIN` to `$FOAM_LIBBIN` (and an equivalent `$FOAM_USER_LIBBIN` directory);
- in the *options* file, `EXE_LIBS =` is replaced by `LIB_LIBS =` to indicate libraries linked to library being compiled.

When wmake is executed it additionally creates a directory named *lnInclude* that contains soft links to all the files in the library. The *lnInclude* directory is deleted by the wclean script when cleaning library source code.

3.2.5 Compilation example: the pisoFoam application

The source code for application pisoFoam is in the *$FOAM_APP/solvers/incompressible/pisoFoam* directory and the top level source file is named *pisoFoam.C*. The *pisoFoam.C* source code is:

```
/*---------------------------------------------------------------------------*\
  =========                 |
  \\      /  F ield         | OpenFOAM: The Open Source CFD Toolbox
   \\    /   O peration     | Website:  https://openfoam.org
    \\  /    A nd           | Copyright (C) 2011-2021 OpenFOAM Foundation
     \\/     M anipulation  |
-------------------------------------------------------------------------------
License
    This file is part of OpenFOAM.

    OpenFOAM is free software: you can redistribute it and/or modify it
    under the terms of the GNU General Public License as published by
    the Free Software Foundation, either version 3 of the License, or
    (at your option) any later version.

    OpenFOAM is distributed in the hope that it will be useful, but WITHOUT
    ANY WARRANTY; without even the implied warranty of MERCHANTABILITY or
    FITNESS FOR A PARTICULAR PURPOSE.  See the GNU General Public License
    for more details.

    You should have received a copy of the GNU General Public License
    along with OpenFOAM.  If not, see <http://www.gnu.org/licenses/>.

Application
    pisoFoam

Description
    Transient solver for incompressible, turbulent flow, using the PISO
    algorithm.

    Sub-models include:
    - turbulence modelling, i.e. laminar, RAS or LES
    - run-time selectable MRF and finite volume options, e.g. explicit porosity

\*---------------------------------------------------------------------------*/

#include "fvCFD.H"
#include "viscosityModel.H"
#include "incompressibleMomentumTransportModels.H"
#include "pisoControl.H"
#include "pressureReference.H"
#include "fvModels.H"
#include "fvConstraints.H"

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

int main(int argc, char *argv[])
{
    #include "postProcess.H"

    #include "setRootCaseLists.H"
    #include "createTime.H"
    #include "createMesh.H"
    #include "createControl.H"
    #include "createFields.H"
    #include "initContinuityErrs.H"

    turbulence->validate();

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

    Info<< "\nStarting time loop\n" << endl;

    while (runTime.loop())
    {
        Info<< "Time = " << runTime.userTimeName() << nl << endl;

        #include "CourantNo.H"

        // Pressure-velocity PISO corrector
        {
            fvModels.correct();

            #include "UEqn.H"

            // --- PISO loop
            while (piso.correct())
            {
                #include "pEqn.H"
            }
        }

```

```
83          viscosity->correct();
84          turbulence->correct();
85
86          runTime.write();
87
88          Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
89              << "  ClockTime = " << runTime.elapsedClockTime() << " s"
90              << nl << endl;
91      }
92
93      Info<< "End\n" << endl;
94
95      return 0;
96  }
97
98
99  // ************************************************************************* //
```

The code begins with a brief description of the application contained within comments over 1 line (`//`) and multiple lines (`/*...*/`). Following that, the code contains several `# include` statements, *e.g.* `# include "fvCFD.H"`, which causes the compiler to suspend reading from the current file, *pisoFoam.C* to read the *fvCFD.H*.

pisoFoam resources the turbulence and transport model libraries and therefore requires the necessary header files, specified by the `EXE_INC = -I...` option, and links to the libraries with the `EXE_LIBS = -l...` option. The *Make/options* therefore contains the following:

```
1   EXE_INC = \
2       -I$(LIB_SRC)/MomentumTransportModels/momentumTransportModels/lnInclude \
3       -I$(LIB_SRC)/MomentumTransportModels/incompressible/lnInclude \
4       -I$(LIB_SRC)/physicalProperties/lnInclude \
5       -I$(LIB_SRC)/finiteVolume/lnInclude \
6       -I$(LIB_SRC)/meshTools/lnInclude \
7       -I$(LIB_SRC)/sampling/lnInclude
8
9   EXE_LIBS = \
10      -lmomentumTransportModels \
11      -lincompressibleMomentumTransportModels \
12      -lphysicalProperties \
13      -lfiniteVolume \
14      -lmeshTools \
15      -lfvModels \
16      -lfvConstraints \
17      -lsampling
```

pisoFoam contains only the *pisoFoam.C* source and the executable is written to the *$FOAM_APPBIN* directory as all standard applications are. The *Make/files* therefore contains:

```
1   pisoFoam.C
2
3   EXE = $(FOAM_APPBIN)/pisoFoam
```

Following the recommendations of section 3.2.2.3, the user can compile a separate version of pisoFoam into their local *$FOAM_USER_DIR* directory by the following:

- copying the pisoFoam source code to a local directory, *e.g.* *$FOAM_RUN*;

  ```
  cd $FOAM_RUN
  cp -r $FOAM_SOLVERS/incompressible/pisoFoam .
  cd pisoFoam
  ```

- editing the *Make/files* file as follows;

  ```
  1   pisoFoam.C
  2
  3   EXE = $(FOAM_USER_APPBIN)/pisoFoam
  ```

- executing wmake.

```
wmake
```

The code should compile and produce a message similar to the following

```
Making dependency list for source file pisoFoam.C
g++ -std=c++0x -m32...
...
-o ... platforms/linuxGccDPInt64Opt/bin/pisoFoam
```

The user can now try recompiling and will receive a message similar to the following to say that the executable is up to date and compiling is not necessary:

```
make: `../bin/pisoFoam' is up to date.
```

The user can compile the application from scratch by removing the dependency list with

```
wclean
```

and running wmake.

3.2.6 Debug messaging and optimisation switches

OpenFOAM provides a system of messaging that is written during runtime, most of which are to help debugging problems encountered during running of a OpenFOAM case. The switches are listed in the *$WM_PROJECT_DIR/etc/controlDict* file; should the user wish to change the settings they should make a copy to their *$HOME* directory, *i.e.* *$HOME/.OpenFOAM/10/controlDict* file (see section 4.3 for more information). The list of possible switches is extensive, relating to a class or range of functionality, and can be switched on by their inclusion in the *controlDict* file, and by being set to `1`. For example, OpenFOAM can perform the checking of dimensional units in all calculations by setting the `dimensionSet` switch to `1`.

A small number of switches control messaging at three levels, `0`, `1` and `2`, most notably the overall `level` switch and `lduMatrix` which provides messaging for solver convergence during a run.

There are some switches that control certain operational and optimisation issues. Of particular importance is `fileModificationSkew`. OpenFOAM scans the write time of data files to check for modification. When running over a NFS with some disparity in the clock settings on different machines, field data files appear to be modified ahead of time. This can cause a problem if OpenFOAM views the files as newly modified and attempting to re-read this data. The `fileModificationSkew` keyword is the time in seconds that OpenFOAM will subtract from the file write time when assessing whether the file has been newly modified. The main optimisation switches are listed below:

- `fileModificationSkew`: a time in seconds that should be set higher than the maximum delay in NFS updates and clock difference for running OpenFOAM over a NFS.

- `fileModificationChecking`: method of checking whether files have been modified during a simulation, either reading the `timeStamp` or using `inotify`; versions that read only master-node data also exist, termed `timeStampMaster` and `inotifyMaster`.

- `commsType`: parallel communications type, `nonBlocking`, `scheduled` or `blocking`.
- `floatTransfer`: if `1`, will compact numbers to `float` precision before transfer; default is `0`.
- `nProcsSimpleSum`: optimises the global sum for parallel processing, by setting the number of processors above which a hierarchical sum is performed rather than a linear sum.

3.2.7 Linking user-defined libraries to applications

The situation may arise that a user creates a new library, say new, and wishes the features within that library to be available across a range of applications. For example, the user may create a new boundary condition, compiled into new, that would need to be recognised by a range of solver applications, pre- and post-processing utilities, mesh tools, *etc.* Under normal circumstances, the user would need to recompile every application with the new linked to it.

Instead there is a simple mechanism to link one or more shared object libraries dynamically at run-time in OpenFOAM. Simply add the optional keyword entry `libs` to the *controlDict* file for a case and enter the full names of the libraries within a list (as quoted string entries). For example, if a user wished to link the libraries new1 and new2 at run-time, they would simply need to add the following to the case *controlDict* file:

```
libs
(
    "libnew1.so"
    "libnew2.so"
);
```

3.3 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 4.1; 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:
  -blockTopology    write block edges and centres as .obj files
  -case <dir>       specify alternate case directory, default is the
                    cwd
  -dict <file>      read control dictionary from specified location
```

```
-fileHandler <handler>
                  override the fileHandler
-libs <(lib1 .. libN)>
                  pre-load libraries
-noClean          keep the existing files in the polyMesh
-noFunctionObjects
                  do not execute functionObjects
-region <name>    specify alternative mesh region
-srcDoc           display source code in browser
-doc              display application documentation in browser
-help             print the usage
```

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.

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 &
```

3.4 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) by default, although other libraries can be used.

3.4.1 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 an 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 is available from the interFoam/damBreak tutorial if the user requires one; the dictionary entries within it are reproduced below:

```
16
17  numberOfSubdomains 4;
18
19  method          simple;
20
21  simpleCoeffs
22  {
23      n               (2 2 1);
24  }
25
```

```
26  hierarchicalCoeffs
27  {
28      n               (1 1 1);
29      order           xyz;
30  }
31
32  manualCoeffs
33  {
34      dataFile        "";
35  }
36
37  distributed     no;
38
39  roots           ( );
40
41
42  // ************************************************************************* //
```

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 below:

- `numberOfSubdomains`: total number of subdomains N.
- `method`: method of decomposition, `simple`, `hierarchical`, `scotch`, `manual`.
- `n`: for `simple` and `hierarchical`, number of subdomains in x, y, z (`(`n_x n_y n_z`)`)
- `order`: order of `hierarchical` decomposition, `xyz`/`xzy`/`yxz`...
- `processorWeights` option for `scotch`: list of weighting factors (`(`<`wt1`>`...`<`wtN`>`)`) 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.
- `dataFile`: for `manual` decomposition, name of file containing data of allocation of cells to processors.
- `distributed`: `yes`/`no` switch indicating whether data is distributed across several disks.

- `roots`: list of root paths to case directories (<`rt1`>...<`rtN`>), where <`rt1`> is the root path for node 1, *etc.*

The decomposePar utility is executed in the normal manner by typing

```
decomposePar
```

3.4.2 File input/output in parallel

Using standard file input/output completion, a set of subdirectories will have been created, one for each processor, in the case directory. The directories are named *processorN* 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.

While this file structure is well-organised, for large parallel cases, it generates a large number of files. In very large simulations, users can experience problems including hitting limits on the number of open files imposed by the operating system.

As an alternative, the **collated** file format was introduced in OpenFOAM in which the data for each decomposed field (and mesh) is collated into a single file that is written (and read) on the master processor. The files are stored in a single directory named *processors*.

The file writing can be threaded allowing the simulation to continue running while the data is being written to file — see below for details. NFS (Network File System) is not needed when using the collated format and, additionally, there is a `masterUncollated` option to write data with the original `uncollated` format without NFS.

The controls for the file handling are in the `OptimisationSwitches` of the global *etc/controlDict* file:

```
OptimisationSwitches
{
    ...

    //- Parallel IO file handler
    // uncollated (default), collated or masterUncollated
    fileHandler uncollated;

    //- collated: thread buffer size for queued file writes.
    // If set to 0 or not sufficient for the file size threading is not used.
    // Default: 2e9
    maxThreadFileBufferSize 2e9;

    //- masterUncollated: non-blocking buffer size.
    // If the file exceeds this buffer size scheduled transfer is used.
    // Default: 2e9
    maxMasterFileBufferSize 2e9;
}
```

3.4.2.1 Selecting the file handler

The `fileHandler` can be set for a specific simulation by:

- over-riding the global `OptimisationSwitches {fileHandler ...;}` in the case *controlDict* file;
- using the `-fileHandler` command line argument to the solver;
- setting the $FOAM_FILEHANDLER environment variable.

3.4.2.2 Updating exisiting files

A foamFormatConvert utility allows users to convert files between the collated and uncollated formats, e.g.

```
mpirun -np 2 foamFormatConvert -parallel -fileHandler uncollated
```

An example case demonstrating the file handling methods is provided in:
$FOAM_TUTORIALS/heatTransfer/buoyantFoam/iglooWithFridges

3.4.2.3 Threading support

Collated file handling runs faster with threading, especially on large cases. But it requires threading support to be enabled in the underlying MPI. Without it, the simulation will "hang" or crash. For openMPI, threading support is not set by default prior to version 2, but is generally switched on from version 2 onwards. The user can check whether openMPI is compiled with threading support by the following command:

```
ompi_info -c | grep -oE "MPI_THREAD_MULTIPLE[^,]*"
```

When using the collated file handling, memory is allocated for the data in the thread. `maxThreadFileBufferSize` sets the maximum size of memory that is allocated in bytes. If the data exceeds this size, the write does not use threading.

Note: if threading is **not enabled** in the MPI, it must be disabled for collated file handling by setting in the global *etc/controlDict* file:

```
maxThreadFileBufferSize   0;
```

When using the `masterUncollated` file handling, non-blocking MPI communication requires a sufficiently large memory buffer on the master node. `maxMasterFileBufferSize` sets the maximum size of the buffer. If the data exceeds this size, the system uses scheduled communication.

3.4.3 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 &
```

3.4.4 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 *processorN* 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.

3.4.5 Post-processing parallel processed cases

When post-processing cases that have been run in parallel the user has two options:

- reconstruction of the mesh and field data to recreate the complete domain and fields, which can be post-processed as normal;
- post-processing each segment of decomposed domain individually.

3.4.5.1 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 *processorN* directory into a single set of time directories. The reconstructPar utility performs such a reconstruction by executing the command:

```
reconstructPar
```

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

3.4.5.2 Post-processing decomposed cases

The user may post-process decomposed cases using the paraFoam post-processor, described in section 6.1. 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.

3.5 Standard solvers

The solvers with the OpenFOAM distribution are in the *$FOAM_SOLVERS* directory, reached quickly by typing `sol` at the command line. This directory is further subdivided into several directories by category of continuum mechanics, *e.g.* incompressible flow, combustion and solid body stress analysis. Each solver is given a name that is reasonably descriptive, *e.g.*icoFoam solves incompressible, laminar flow. The current list of solvers distributed with OpenFOAM is given in the following Sections.

3.5.1 'Basic' CFD codes

laplacianFoam Solves a simple Laplace equation, e.g. for thermal diffusion in a solid.

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.

scalarTransportFoam Solves the steady or transient transport equation for a passive scalar.

3.5.2 Incompressible flow

adjointShapeOptimisationFoam 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, for use in a simulation.

icoFoam Transient solver for incompressible, laminar flow of Newtonian fluids.

pimpleFoam Transient solver for incompressible, turbulent flow of Newtonian fluids, with optional mesh motion and mesh topology changes.

pisoFoam Transient solver for incompressible, turbulent flow, using the PISO algorithm.

porousSimpleFoam Steady-state solver for incompressible, turbulent flow with implicit or explicit porosity treatment and support for multiple reference frames (MRF).

shallowWaterFoam Transient solver for inviscid shallow-water equations with rotation.

simpleFoam Steady-state solver for incompressible, turbulent flow, using the SIMPLE algorithm.

SRFPimpleFoam Large time-step transient solver for incompressible, turbulent flow in a single rotating frame.

SRFSimpleFoam Steady-state solver for incompressible, turbulent flow of non-Newtonian fluids in a single rotating frame.

3.5.3 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.

rhoPimpleFoam Transient solver for turbulent flow of compressible fluids for HVAC and similar applications, with optional mesh motion and mesh topology changes.

rhoPorousSimpleFoam Steady-state solver for turbulent flow of compressible fluids, with implicit or explicit porosity treatment and optional sources.

rhoSimpleFoam Steady-state solver for turbulent flow of compressible fluids.

3.5.4 Multiphase flow

cavitatingFoam Transient cavitation code 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 2 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.

compressibleMultiphaseInterFoam Solver for n compressible, non-isothermal immiscible fluids using a VOF (volume of fluid) phase-fraction based interface capturing approach.

driftFluxFoam Solver for 2 incompressible fluids using the mixture approach with the drift-flux approximation for relative motion of the phases.

interFoam Solver for 2 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.

multiphaseEulerFoam 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 momentun, heat and mass 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, with optional mesh motion and mesh topology changes.

twoLiquidMixingFoam Solver for mixing 2 incompressible fluids.

3.5.5 Direct numerical simulation (DNS)

dnsFoam Direct numerical simulation solver for boxes of isotropic turbulence.

3.5.6 Combustion

buoyantReactingFoam Solver for combustion with chemical reactions using a density based thermodynamics package with enhanced buoyancy treatment.

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.

PDRFoam Solver for compressible premixed/partially-premixed combustion with turbulence modelling.

reactingFoam Solver for combustion with chemical reactions.

XiEngineFoam Solver for internal combustion engines.

XiFoam Solver for compressible premixed/partially-premixed combustion with turbulence modelling.

3.5.7 Heat transfer and buoyancy-driven flows

buoyantFoam Solver for steady or transient buoyant, turbulent flow of compressible fluids for ventilation and heat-transfer, with optional mesh motion and mesh topology changes.

chtMultiRegionFoam Solver for steady or transient fluid flow and solid heat conduction, with conjugate heat transfer between regions, buoyancy effects, turbulence, reactions and radiation modelling.

thermoFoam Solver for energy transport and thermodynamics on a frozen flow field.

3.5.8 Particle-tracking flows

denseParticleFoam Transient solver for the coupled transport of particle clouds including the effect of the volume fraction of particles on the continuous phase, with optional mesh motion and mesh topology changes.

particleFoam Transient solver for the passive transport of a single kinematic particle cloud, with optional mesh motion and mesh topology changes.

rhoParticleFoam Transient solver for the passive transport of a particle cloud.

3.5.9 Discrete methods

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

mdEquilibrationFoam Solver to equilibrate and/or precondition molecular dynamics systems.

mdFoam Molecular dynamics solver for fluid dynamics.

3.5.10 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.

3.5.11 Stress analysis of solids

solidDisplacementFoam Transient solver of linear-elastic, small-strain deformation of a solid body, with optional thermal diffusion and thermal stresses.

solidEquilibriumDisplacementFoam Steady-state solver of linear-elastic, small-strain deformation of a solid body, with optional thermal diffusion and thermal stresses.

3.5.12 Finance

financialFoam Solves the Black-Scholes equation to price commodities.

3.6 Standard utilities

The utilities with the OpenFOAM distribution are in the *$FOAM_UTILITIES* directory. The names are reasonably descriptive, *e.g.* ideasToFoam converts mesh data from the format written by I-DEAS to the OpenFOAM format. The descriptions of current utilities distributed with OpenFOAM are given in the following Sections.

3.6.1 Pre-processing

applyBoundaryLayer Apply a simplified boundary-layer model to the velocity and turbulence fields based on the 1/7th power-law.

boxTurb Makes a box of turbulence which conforms to a given energy spectrum and is divergence free.

changeDictionary Utility to change dictionary entries, e.g. can be used to change the patch type in the field and `polyMesh`/boundary files.

createExternalCoupledPatchGeometry Application to generate the patch geometry (points and faces) for use with the `externalCoupled` boundary condition.

dsmcInitialise Initialise a case for `dsmcFoam` by reading the initialisation dictionary system/-`dsmcInitialise`.

engineSwirl Generates a swirling flow for engine calulations.

faceAgglomerate Agglomerate boundary faces using the pairPatchAgglomeration algorithm. It writes a map from the fine to coarse grid.

foamSetupCHT Sets up a multi-region case using template files for material properties, field and system files.

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. Parallel and non-parallel cases are handled without the need to reconstruct them first.

mdInitialise Initialises fields for a molecular dynamics (MD) simulation.

setAtmBoundaryLayer Applies atmospheric boundary layer models to the entire domain for case initialisation.

setFields Set values on a selected set of cells/patchfaces through a dictionary.

setWaves Applies wave models to the entire domain for case initialisation using level sets for second-order accuracy.

viewFactorsGen View factors are calculated based on a face agglomeration array (`finalAgglom` generated by `faceAgglomerate` utility).

3.6.2 Mesh generation

blockMesh A multi-block mesh generator.

extrudeMesh Extrude mesh from existing patch (by default outwards facing normals; optional flips faces) or from patch read from file.

extrude2DMesh Takes 2D mesh (all faces 2 points only, no front and back faces) and creates a 3D mesh by extruding with specified thickness.

extrudeToRegionMesh Extrude `faceZones` (internal or boundary faces) or `faceSets` (boundary faces only) into a separate mesh (as a different region).

foamyHexMesh Conformal Voronoi automatic mesh generator

foamyQuadMesh Conformal-Voronoi 2D extruding automatic mesher with grid or read initial points and point position relaxation with optional "squarification".

snappyHexMesh Automatic split hex mesher. Refines and snaps to surface.

3.6.3 Mesh conversion

ansysToFoam Converts an ANSYS input mesh file, exported from I-DEAS, to OpenFOAM format.

ccm26ToFoam Reads CCM files as written by Prostar/ccm using ccm 2.6

cfx4ToFoam Converts a CFX 4 mesh to OpenFOAM format.

datToFoam Reads in a `datToFoam` mesh file and outputs a points file. Used in conjunction with `blockMesh`.

fluent3DMeshToFoam Converts a Fluent mesh to OpenFOAM format.

fluentMeshToFoam Converts a Fluent mesh to OpenFOAM format including multiple region and region boundary handling.

foamMeshToFluent Writes out the OpenFOAM mesh in Fluent mesh format.

foamToStarMesh Reads an OpenFOAM mesh and writes a pro-STAR (v4) bnd/cel/vrt format.

foamToSurface Reads an OpenFOAM mesh and writes the boundaries in a surface format.

gambitToFoam Converts a GAMBIT mesh to OpenFOAM format.

gmshToFoam Reads .msh file as written by Gmsh.

ideasUnvToFoam I-Deas unv format mesh conversion.

kivaToFoam Converts a KIVA3v grid to OpenFOAM format.

mshToFoam Converts .msh file generated by the Adventure system.

netgenNeutralToFoam Converts neutral file format as written by Netgen v4.4.

plot3dToFoam Plot3d mesh (ascii/formatted format) converter.

sammToFoam Converts a Star-CD (v3) SAMM mesh to OpenFOAM format.

star3ToFoam Converts a Star-CD (v3) pro-STAR mesh into OpenFOAM format.

star4ToFoam Converts a Star-CD (v4) pro-STAR mesh into OpenFOAM format.

tetgenToFoam Converts .ele and .node and .face files, written by tetgen.

vtkUnstructuredToFoam Converts ascii .vtk (legacy format) file generated by vtk/paraview.

writeMeshObj For mesh debugging: writes mesh as three separate OBJ files which can be viewed with e.g. javaview.

3.6.4 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`.

createNonConformalCouples Utility to create non-conformal couples between non-coupled patches.

createPatch Utility to create patches out of selected boundary faces. Faces come 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 Picks up cells with cell centre 'inside' of surface. Requires surface to be closed and singly connected.

mergeBaffles Detects faces that share points (baffles) and merges them into internal faces.

mergeMeshes Merges two meshes.

mirrorMesh Mirrors a mesh around a given plane.

moveMesh Mesh motion and topological mesh changes utility.

objToVTK Read obj line (not surface!) file and convert into vtk.

orientFaceZone Corrects orientation of `faceZone`.

polyDualMesh Calculates the dual of a `polyMesh`. Adheres 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 n1 to direction n2.

setsToZones Add `pointZones`, `faceZones` or `cellZones` to the mesh from similar named `pointSets`, `faceSets` or `cellSets`.

singleCellMesh Reads all fields and maps them to a mesh with all internal faces removed (`singleCellFvMesh`) which gets written to region `singleCell`.

splitBaffles Detects faces that share points (baffles) and duplicates the points to separate them.

splitMesh Splits mesh by making internal faces external. Uses `attachDetach`.

splitMeshRegions Splits mesh into multiple regions.

stitchMesh Stitches a mesh.

subsetMesh Selects a section of mesh based on a `cellSet`.

topoSet Operates on `cellSets`/`faceSets`/`pointSets` through a dictionary.

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.

3.6.5 Other mesh tools

autoRefineMesh Utility to refine cells near to a surface.

collapseEdges Collapses short edges and combines edges that are in line.

combinePatchFaces Checks for multiple patch faces on 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 Manipulates mesh elements.

PDRMesh Mesh and field preparation utility for PDR type simulations.

refineHexMesh Refines a hex mesh by 2x2x2 cell splitting.

refinementLevel Tries to figure out what the refinement level is on refined Cartesian meshes. Run BEFORE snapping.

refineWallLayer Utility to refine cells next to patches.

removeFaces Utility to remove faces (combines cells on both sides).

selectCells Select cells in relation to surface.

splitCells Utility to split cells with flat faces.

3.6.6 Post-processing

engineCompRatio Calculate the geometric compression ratio. Note that if you have valves and/or extra volumes it will not work, since it calculates the volume at BDC and TCD.

noise Utility to perform noise analysis of pressure data using the `noiseFFT` library.

particleTracks Generates a VTK file of particle tracks for cases that were computed using a tracked-parcel-type cloud.

pdfPlot Generates a graph of a probability distribution function.

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.

steadyParticleTracks Generates a VTK file of particle tracks for cases that were computed using a steady-state cloud NOTE: case must be re-constructed (if running in parallel) before use

temporalInterpolate Interpolate fields between time-steps e.g. for animation.

3.6.7 Post-processing data converters

foamDataToFluent Translates OpenFOAM data to Fluent format.

foamToEnsight Translates OpenFOAM data to EnSight format.

foamToEnsightParts Translates OpenFOAM data to Ensight format. An Ensight part is created for each `cellZone` and patch.

foamToGMV Translates foam output to GMV readable files.

foamToTetDualMesh Converts `polyMesh` results to `tetDualMesh`.

foamToVTK Legacy VTK file format writer.

smapToFoam Translates a STAR-CD SMAP data file into OpenFOAM field format.

3.6.8 Surface mesh (e.g. OBJ/STL) tools

surfaceAdd Add two surfaces. Does geometric merge on points. Does not check for overlapping/intersecting triangles.

surfaceAutoPatch Patches surface according to feature angle. Like `autoPatch`.

surfaceBooleanFeatures Generates the *extendedFeatureEdgeMesh* for the interface between a boolean operation on two surfaces. Assumes that the orientation of the surfaces is correct.

surfaceCheck Checks geometric and topological quality of a surface.

surfaceClean Removes baffles - collapses small edges, removing triangles. - converts sliver triangles into split edges by projecting point onto base of triangle.

surfaceCoarsen Surface coarsening using bunnylod:

surfaceConvert Converts from one surface mesh format to another.

surfaceFeatureConvert Convert between `edgeMesh` formats.

surfaceFeatures Identifies features in a surface geometry and writes them to file, based on control parameters specified by the user.

surfaceFind Finds nearest face and vertex.

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`. Inertia can either be of the solid body or of a thin shell.

surfaceLambdaMuSmooth Smooths a surface using lambda/mu smoothing.

surfaceMeshConvert Converts between surface formats with optional scaling or transformations (rotate/translate) on a `coordinateSystem`.

surfaceMeshConvertTesting Converts from one surface mesh format to another, but primarily used for testing functionality.

surfaceMeshExport Export from `surfMesh` to various third-party surface formats with optional scaling or transformations (rotate/translate) on a `coordinateSystem`.

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.

surfaceMeshTriangulate Extracts surface from a `polyMesh`. Depending on output surface format triangulates faces.

surfaceOrient Set normal consistent with respect to a user provided 'outside' point. If the `-inside` option is used the point is considered inside.

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). Neighbouring triangles which are not marked for refinement get split in half ('green' refinement).

surfaceSplitByPatch Writes regions of `triSurface` to separate files.

surfaceSplitByTopology Strips any baffle parts of a surface. A baffle region is one which is reached by walking from an open edge, and stopping when a multiply connected edge is reached.

surfaceSplitNonManifolds Takes multiply connected surface and tries to split surface at multiply connected edges by duplicating points. Introduces concept of - `borderEdge`. Edge with 4 faces connected to it. - `borderPoint`. Point connected to exactly 2 `borderEdges`. - `borderLine`. Connected list of `borderEdges`.

surfaceSubset A surface analysis tool which sub-sets the `triSurface` to choose only a part of interest. Based on `subsetMesh`.

surfaceToPatch Reads surface and applies surface regioning to a mesh. Uses `boundaryMesh` to do the hard work.

surfaceTransformPoints Transform (scale/rotate) a surface. Like `transformPoints` but for surfaces.

3.6.9 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.

3.6.10 Thermophysical-related utilities

adiabaticFlameT Calculates the adiabatic flame temperature for a given fuel over a range of unburnt temperatures and equivalence ratios.

chemkinToFoam Converts CHEMKINIII thermodynamics and reaction data files into OpenFOAM format.

equilibriumCO Calculates the equilibrium level of carbon monoxide.

equilibriumFlameT Calculates the equilibrium flame temperature for a given fuel and pressure for a range of unburnt gas temperatures and equivalence ratios; the effects of dissociation on O2, H2O and CO2 are included.

mixtureAdiabaticFlameT Calculates the adiabatic flame temperature for a given mixture at a given temperature.

3.6.11 Miscellaneous utilities

foamDictionary Interrogates and manipulates dictionaries.

foamFormatConvert Converts all IOobjects associated with a case into the format specified in the `controlDict`.

foamListTimes List times using `timeSelector`.

patchSummary Writes fields and boundary condition info for each patch at each requested time instance.

Chapter 4

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}-10* as described at the beginning of chapter 2. One advantage of this is that the $FOAM_RUN environment variable is set to *$HOME/OpenFOAM/${USER}-10/run* by default; the user can quickly move to that directory by executing a preset alias, `run`, at the command line.

The tutorial cases that accompany the OpenFOAM distribution provide useful examples of the case directory structures. The tutorials are located in the *$FOAM_TUTORIALS* directory, reached quickly by executing the `tut` alias at the command line. Users can view tutorial examples at their leisure while reading this chapter.

4.1 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 4.1 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.* *physicalProperties*.

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, *e.g.* velocity and pressure. 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 section 4.4. It is sufficient to say now that since we usually start

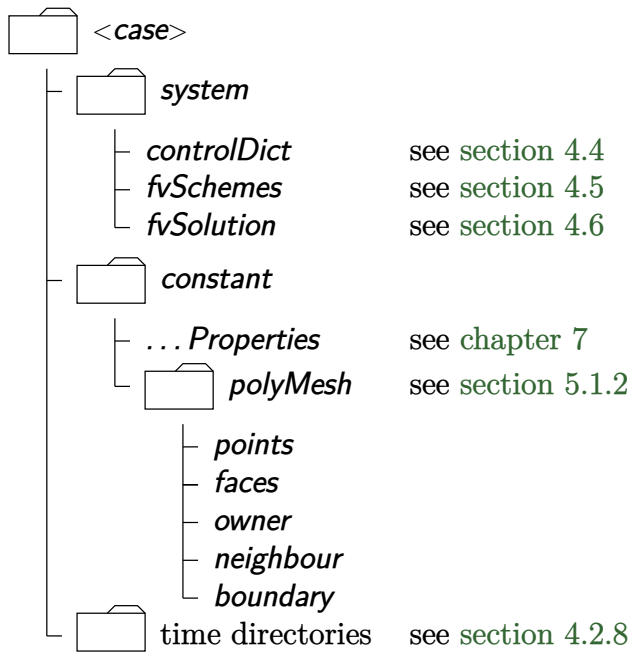


Figure 4.1: Case directory structure

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.

4.2 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. The OpenFOAM file format is described in the following sections.

4.2.1 General syntax rules

The format follows some 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.

4.2.2 Dictionaries

OpenFOAM uses *dictionaries* as the most common means of specifying data. A dictionary is an entity that contains 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 by keyword entries enclosed in curly braces `{}` as follows.

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

4.2.3 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 below:

- `version`: I/O format version, optional, defaults to `2.0`
- `format`: data format, `ascii` or `binary`
- `class`: class relating to the data, either `dictionary` or a field, *e.g.* `volVectorField`
- `object`: filename, *e.g.* `controlDict` (mandatory, but not used)
- `location`: path to the file (optional)

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.

```
16
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  // ************************************************************************* //
```

4.2.4 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:

- the keyword is followed immediately by round braces

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

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

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

- 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.

4.2.5 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 )
```

4.2.6 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]
```

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

Table 4.1: Base units for SI and USCS

where each of the values corresponds to the power of each of the base units of measurement listed in Table 4.1. 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 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.

4.2.7 Dimensioned types

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

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

The first `nu` is the keyword; the second `nu` is the word name stored in class word, usually chosen to be the same as the keyword; the next entry is the dimensionSet and the final entry is the scalar value.

The majority of dimensioned keyword lookups set a default for the word name which can therefore be omitted from the entry, so the more common syntax is:

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

4.2.8 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 4.2.

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 4.2.8

Table 4.2: 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 mandatory 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. A selection of patch field conditions available in OpenFOAM are listed in section 5.2.1, section 5.2.2 and section 5.2.3, with a description and the data that must be specified with it. Example field dictionary entries for velocity `U` are shown below:

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

4.2.9 Macro expansion

OpenFOAM dictionary files include a macro syntax to allow convenient configuration of case files. The syntax uses the dollar (`$`) symbol in front of a keyword to expand the data associated with the keyword. For example the value set for keyword `a` below, 10, is expanded in the following line, so that the value of `b` is also 10.

```
a 10;
b $a;
```

Variables can be accessed within different levels of sub-dictionaries, or scope. Scoping is performed using a '`/`' (dot) syntax, illustrated by the following example, where `b` is set to the value of `a`, specified in a sub-dictionary called `subdict`.

```
subdict
{
    a 10;
}
b $subdict/a;
```

There are further syntax rules for macro expansions:

- to traverse up one level of sub-dictionary, use the '`..`' (double-dot) prefix, see below;
- to traverse up two levels use '`...`' (triple-dot) prefix, *etc.*;

- to traverse to the top level dictionary use the ':' (colon) prefix (most useful), see below;
- for multiple levels of macro substitution, each specified with the '$' dollar syntax, '{}' brackets are required to protect the expansion, see below.

```
a 10;
b a;
c ${${b}}; // returns 10, since $b returns "a", and $a returns 10

subdict
{
    b $..a; // double-dot takes scope up 1 level, then "a" is available

    subsubdict
    {
        c $:a; // colon takes scope to top level, then "a" is available
    }
}
```

4.2.10 Including files

There is additional file syntax that provides further flexibility for setting up of OpenFOAM case files, namely directives. Directives are commands that can be contained within case files that begin with the hash (#) symbol. The first set of directive commands are those for reading a data file from within another data file. 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;
```

In order to use this pressure for both the internal and initial boundary fields, the user would simply include the *initialConditions* file using the `#include` directive, then use macro expansions for the `pressure` keyword, as follows.

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

The file include directives are as follows:

- `#include "<path>/<fileName>"`: reads the file of name <*fileName*> from an absolute or relative directory path <*path*>;

- `#includeIfPresent "<path>/<fileName>"`: reads the file if it exists;
- `#includeEtc "<path>/<fileName>"`: reads the file of name <*fileName*> from the directory path <*path*>, relative to the *$FOAM_ETC* directory;
- `#includeFunc <fileName>`: reads the file of name <*fileName*>, searched from the case *system* directory, followed by the *$FOAM_ETC* directory;
- `#remove <keywordEntry>`: removes any included keyword entry; can take a word or regular expression;

4.2.11 Environment variables

OpenFOAM recognises the use of environment variables in input files. For example, the $FOAM_RUN environment variable can be used to identify the *run* directory, as described in the introduction to Chapter 2. This could be used to include a file, *e.g.* by

```
#include "$FOAM_RUN/pitzDaily/0/U"
```

In addition to environment variables like $FOAM_RUN, set within the operating system, OpenFOAM recognises a number of "internal" environment variables, including the following.

- $FOAM_CASE: the path and directory of the running case.
- $FOAM_CASENAME: the directory *name* of the running case.
- $FOAM_APPLICATION: the name of the running application.

4.2.12 Regular expressions

When running an application, data is initialised by looking up keywords from dictionaries. The user can either provide an entry with a keyword that directly matches the one being looked up, or can provide a POSIX regular expression that matches the keyword, specified inside double-quotations (`"..."`). Regular expressions have an extensive syntax for various matches of text patterns but they are typically only used in the following ways in OpenFOAM input files.

- `"inlet.*"` matches any word beginning `inlet`..., including `inlet` itself, because '`.`' denotes "any character" and '`*`' denotes "repeated any number of times, including 0 times".
- `"(inlet|output)"` matches `inlet` and `outlet` because `()` specified an expression grouping and `|` is an OR operator.

4.2.13 Keyword ordering

The order in which keywords are listed does not matter, except when *the same keyword is specified multiple times.* Where the same keyword is duplicated, the last instance is used. The most common example of a duplicate keyword occurs when a keyword is included from the file or expanded from a macro, and then overridden. The example below demonstrates this, where `pFinal` adopts all the keyword entries, including `relTol 0.05` in the `p` sub-dictionary by the macro expansion `$p`, then overrides the `relTol` entry.

```
p
{
    solver          PCG;
    preconditioner  DIC;
    tolerance       1e-6;
    relTol          0.05;
}
pFinal
{
    $p;
    relTol          0;
}
```

Where a data lookup matches both a keyword and a regular expression, the keyword match takes precedence irrespective of the order of the entries.

4.2.14 Inline calculations and code

There are two further directives that enable calculations from within input files: `#calc`, for simple calculations; `#codeStream`, for more complex calculations.

The pipeCyclic tutorial in *$FOAM_TUTORIALS/incompressible/simpleFoam* demonstrates the `#calc` directive through its blockMesh configuration in *blockMeshDict*:

```
//- Half angle of wedge in degrees
halfAngle 45.0;

//- Radius of pipe [m]
radius 0.5;

radHalfAngle    #calc "degToRad($halfAngle)";
y               #calc "$radius*sin($radHalfAngle)";
z               #calc "$radius*cos($radHalfAngle)";
```

The file contains several calculations that calculate vertex ordinates, *e.g.* `y`, `z`, *etc.*, from geometry dimensions, *e.g.* `radius`. Calculations include standard C++ functions including unit conversions, *e.g.* `degToRad`, and trigonometric functions, *e.g.* `sin`.

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, where the code in the calculates moment of inertia of a box shaped geometry.

```
momentOfInertia #codeStream
{
    codeInclude
    #{
        #include "diagTensor.H"
    #};

    code
    #{
        scalar sqrLx = sqr($Lx);
        scalar sqrLy = sqr($Ly);
        scalar sqrLz = sqr($Lz);
        os  <<
            $mass
           *diagTensor(sqrLy + sqrLz, sqrLx + sqrLz, sqrLx + sqrLy)/12.0;
    #};
};
```

4.2.15 Conditionals

Input files support two conditional directives: `#if...#else...#endif`; and, `#ifEq... #else... #endif`. The `#if` conditional reads a switch that can be generated by a `#calc` directive, *e.g.*:

```
angle 65;

laplacianSchemes
{
#if #calc "${angle} < 75"
    default  Gauss linear corrected;
#else
    default  Gauss linear limited corrected 0.5;
#endif
}
```

The `#ifEq` compares a word or string, and executes based on a match, *e.g.*:

```
ddtSchemes
{
#ifeq ${FOAM_APPLICATION} simpleFoam
    default         steadyState;
#else
    default         Euler;
#endif
}
```

4.3 Global controls

OpenFOAM includes a large number of global parameters that are configured by default in a file named *controlDict*. This is the so-called "global" *controlDict* file, as opposed to a case *controlDict* file that is described in the following section.

The global *controlDict* file can be found in the installation within a directory named *etc*, represented by the environment variable $FOAM_ETC. The file contains sub-dictionaries for the following items.

- `Documentation`: for opening documentation in a web browser.
- `InfoSwitches`: controls information printed to standard output, *i.e.* the terminal.
- `OptimisationSwitches`: for parallel communication and I/O, see section 3.4.2.
- `DebugSwitches`: messaging switches to help debug code failures, as described in section 3.2.6.
- `DimensionedConstants`: defines fundamental physical constants, *e.g.* Boltzmann's Constant.
- `DimensionSets`: defines a notation for dimensional units, *e.g.* `kg`.

4.3.1 Overriding global controls

The *values* of the `DimensionedConstants` depend on the unit system being adopted, *i.e.* the International System of Units (SI units), or US Customary system (USCS), based on English units (pounds, feet, *etc.*). The default system is naturally SI, but some users may wish to override this with USCS units, either globally or for a specific case. The system is set through the `unitSet` keyword, *i.e.*

```
DimensionedConstants
{
    unitSet  SI; // USCS
}
```

While a user could modify this setting in the *etc/controlDict* file in the installation, it is better practice to use a file in their user directory. OpenFOAM provides a set of directory locations, where global configuration files can be included, which it looks up in an order of precedence. To list the locations, simply run the following command.

```
foamEtcFile -list
```

The listed locations include a local *$HOME/.OpenFOAM* directory and follow a descending order of precedence, *i.e.* the last location listed (*etc*) is lowest precedence.

If a user therefore wished to work permanently in USCS units, they could maintain a *controlDict* file in their *$HOME/.OpenFOAM* directory that includes the following entry.

```
DimensionedConstants
{
    unitSet  USCS;
}
```

OpenFOAM would read the `unitSet` entry from this file, but read all other *controlDict* keyword entries from the global *controlDict* file.

Alternatively, if a user wished to work on a *single case* in USCS units, they could add the same entry into the *controlDict* file in the *system* directory for their *case*. This file is discussed in the next section.

4.4 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 the following sections. Only the time control and `writeInterval` entries are mandatory, with the database using default values for any of the optional entries that are omitted. Example entries from a *controlDict* dictionary are given below:

```
16
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  // ************************************************************************* //
```

4.4.1 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.

4.4.2 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. For example, if the simulations starts at $t = 5$s and $\Delta t = 1$s, then with `purgeWrite 2;`, data is first written into 2 directories, *6* and *7*, then when *8* is written, *6* is deleted, and so on so that only 2 new results directories exists at any time. *To disable the purging, specify* `purgeWrite 0;` (default).

`writeFormat` Specifies the format of the data files.

- `ascii` (default): ASCII format, written to `writePrecision` significant figures.
- `binary`: binary format.

`writePrecision` Integer used in conjunction with `writeFormat` described above, 6 by default.

`writeCompression` Switch to specify whether files are compressed with gzip when written: `on/off` (`yes/no`, `true/false`)

`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`: $\pm m.dddddde\pm xx$ where the number of ds is set by `timePrecision`.
- `general` (default): 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.

- `raw` (default): Raw ASCII format in columns.
- `gnuplot`: Data in gnuplot format.
- `xmgr`: Data in Grace/xmgr format.
- `jplot`: Data in jPlot format.

4.4.3 Other settings

`adjustTimeStep` Switch used by some solvers to adjust the time step during the simulation, usually according to `maxCo`.

`maxCo` Maximum Courant number, *e.g.* `0.5`

`runTimeModifiable` Switch for whether dictionaries, *e.g.controlDict*, are re-read during a simulation at the beginning of each time step, allowing the user to modify parameters during a simulation.

`libs` List of additional libraries (existing on $LD_LIBRARY_PATH) to be loaded at run-time, *e.g.* `("libNew1.so" "libNew2.so")`

`functions` Dictionary of functions, *e.g.* `probes` to be loaded at run-time; see examples in *$FOAM_TUTORIALS*

4.5 Numerical schemes

The *fvSchemes* dictionary in the *system* directory sets the numerical schemes for terms, such as derivatives in equations, that are calculated during a simulation. 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 ∇, to interpolations of values from one set of points to another. The aim in OpenFOAM is to offer an unrestricted choice to the user, starting with the choice of discretisation practice which is generally standard Gaussian finite volume integration. Gaussian integration is based on summing values on cell faces, which must be interpolated from cell centres. The user has a wide range of options for interpolation scheme, with certain schemes being specifically designed for particular derivative terms, especially the advection divergence $\nabla \bullet$ terms.

The set of terms, for which numerical schemes must be specified, are subdivided within the *fvSchemes* dictionary into the categories below.

- `timeScheme`: first and second time derivatives, *e.g.* $\partial/\partial t, \partial^2/\partial^2 t$
- `gradSchemes`: gradient ∇
- `divSchemes`: divergence $\nabla \bullet$
- `laplacianSchemes`: Laplacian ∇^2
- `interpolationSchemes`: cell to face interpolations of values.
- `snGradSchemes`: component of gradient normal to a cell face.
- `wallDist`: distance to wall calculation, where required.

Each keyword in represents 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:

```
16
17  ddtSchemes
18  {
19      default         Euler;
20  }
21
22  gradSchemes
23  {
24      default         Gauss linear;
25  }
26
27  divSchemes
28  {
29      default             none;
30
31      div(phi,U)          Gauss linearUpwind grad(U);
32      div(phi,k)          Gauss upwind;
33      div(phi,epsilon)    Gauss upwind;
34      div(phi,R)          Gauss upwind;
35      div(R)              Gauss linear;
36      div(phi,nuTilda)    Gauss upwind;
37
38      div((nuEff*dev2(T(grad(U))))) Gauss linear;
39  }
40
41  laplacianSchemes
42  {
43      default         Gauss linear corrected;
44  }
45
46  interpolationSchemes
47  {
48      default         linear;
49  }
50
51  snGradSchemes
52  {
53      default         corrected;
54  }
55
56
57  // ************************************************************************* //
```

The example shows that the *fvSchemes* dictionary contains 6 ... *Schemes* subdictionaries 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 can specify that no `default` scheme by the `none` entry, as in the `divSchemes` in the example above. 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.

OpenFOAM includes a vast number of discretisation schemes, from which only a few are typically recommended for real-world, engineering applications. The user can get help with scheme selection by interrogating the tutorial cases for example scheme settings. They should look at the schemes used in relevant cases, *e.g.* for running a large-eddy simulation (LES), look at schemes used in tutorials running LES. Additionally, foamSearch provides a useful tool to get a quick list of schemes used in all the tutorials. For example, to print all the `default` entries for `ddtSchemes` for cases in the *$FOAM_*-

TUTORIALS directory, the user can type:

```
foamSearch $FOAM_TUTORIALS fvSchemes ddtSchemes/default
```

which prints:

```
default         backward;
default         CrankNicolson 0.9;
default         Euler;
default         localEuler;
default         none;
default         steadyState;
```

The schemes listed using foamSearch are described in the following sections.

4.5.1 Time schemes

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

- `steadyState`: sets time derivatives to zero.
- `Euler`: transient, first order implicit, bounded.
- `backward`: transient, second order implicit, potentially unbounded.
- `CrankNicolson`: transient, second order implicit, bounded; requires an off-centering coefficient ψ where:

  $$\psi = \begin{cases} 1 & \text{corresponds to pure } \texttt{CrankNicolson}, \\ 0 & \text{corresponds to } \texttt{Euler}; \end{cases}$$

  generally $\psi = 0.9$ is used to bound/stabilise the scheme for practical engineering problems.
- `localEuler`: pseudo transient for accelerating a solution to steady-state using local-time stepping; first order implicit.

Solvers are generally configured to simulate either transient or steady-state. Changing the time scheme from one which is steady-state to transient, or visa versa, does not affect the fundamental nature of the solver and so fails to achieve its purpose, yielding a nonsensical solution.

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*.

4.5.2 Gradient schemes

The *gradSchemes* sub-dictionary contains gradient terms. The `default` discretisation scheme that is primarily used for gradient terms is:

```
default         Gauss linear;
```

The `Gauss` entry specifies the standard finite volume discretisation of Gaussian integration which requires the interpolation of values from cell centres to face centres. The interpolation scheme is then given by the `linear` entry, meaning linear interpolation or central differencing.

In some tutorials cases, particular involving poorer quality meshes, the discretisation of specific gradient terms is overridden to improve boundedness and stability. The terms that are overridden in those cases are the velocity gradient

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

and, less frequently, the gradient of turbulence fields, *e.g.*

```
grad(k)         cellLimited Gauss linear 1;
grad(epsilon)   cellLimited Gauss linear 1;
```

They use the `cellLimited` scheme which limits the gradient such that when cell values are extrapolated to faces using the calculated gradient, the face values do not fall outside the bounds of values in surrounding cells. A limiting coefficient is specified after the underlying scheme for which 1 guarantees boundedness and 0 applies no limiting; 1 is invariably used.

Other schemes that are rarely used are as follows.

- `leastSquares`: a second-order, least squares distance calculation using all neighbour cells.
- `Gauss cubic`: third-order scheme that appears in the dnsFoam simulation on a regular mesh.

4.5.3 Divergence schemes

The *divSchemes* sub-dictionary contains divergence terms, *i.e.* terms of the form $\nabla \bullet \ldots$, excluding Laplacian terms (of the form $\nabla \bullet (\Gamma \nabla \ldots)$). This includes both advection terms, *e.g.* $\nabla \bullet (\mathbf{U}k)$, where velocity $\mathbf{U}$ provides the advective flux, and other terms, that are often diffusive in nature, *e.g.* $\nabla \bullet \nu(\nabla \mathbf{U})^{\mathrm{T}}$.

The fact that terms that are fundamentally different reside in one sub-dictionary means that the `default` scheme in generally set to `none` in *divSchemes*. The non-advective terms then generally use the `Gauss` integration with `linear` interpolation, *e.g.*

```
div(U)          Gauss linear;
```

The treatment of advective terms is one of the major challenges in CFD numerics and so the options are more extensive. The keyword identifier for the advective terms are usually of the form `div(phi,...)`, where `phi` generally denotes the (volumetric) flux of velocity on the cell faces for constant-density flows and the mass flux for compressible flows, *e.g.* `div(phi,U)` for the advection of velocity, `div(phi,e)` for the advection of internal energy, `div(phi,k)` for turbulent kinetic energy, *etc.* For advection of velocity, the user can run the foamSearch script to extract the `div(phi,U)` keyword from all tutorials.

```
foamSearch $FOAM_TUTORIALS fvSchemes "divSchemes/div(phi,U)"
```

The schemes are all based on `Gauss` integration, using the flux `phi` and the advected field being interpolated to the cell faces by one of a selection of schemes, *e.g.* `linear`, `linearUpwind`, *etc.* There is a `bounded` variant of the discretisation, discussed later.

Ignoring 'V'-schemes (with keywords ending "`V`"), and rarely-used schemes such as `Gauss cubic` and `vanLeerV`, the interpolation schemes used in the tutorials are as follows.

- `linear`: second order, unbounded.
- `linearUpwind`: second order, upwind-biased, unbounded (but much less so than `linear`), that requires discretisation of the velocity gradient to be specified.
- `LUST`: blended 75% `linear`/ 25%`linearUpwind` scheme, that requires discretisation of the velocity gradient to be specified.
- `limitedLinear`: `linear` scheme that limits towards `upwind` in regions of rapidly changing gradient; requires a coefficient, where 1 is strongest limiting, tending towards `linear` as the coefficient tends to 0.
- `upwind`: first-order bounded, generally too inaccurate to be recommended.

Example syntax for these schemes is as follows.

```
div(phi,U)      Gauss linear;
div(phi,U)      Gauss linearUpwind grad(U);
div(phi,U)      Gauss LUST grad(U);
div(phi,U)      Gauss LUST unlimitedGrad(U);
div(phi,U)      Gauss limitedLinear 1;
div(phi,U)      Gauss upwind;
```

'V'-schemes are specialised versions of schemes designed for vector fields. They differ from conventional schemes by calculating a single limiter which is applied to all components of the vectors, rather than calculating separate limiters for each component of the vector. The 'V'-schemes' single limiter is calculated based on the direction of most rapidly changing gradient, resulting in the strongest limiter being calculated which is most stable but arguably less accurate. Example syntax is as follows.

```
div(phi,U)      Gauss limitedLinearV 1;
div(phi,U)      Gauss linearUpwindV grad(U);
```

The **`bounded`** variants of schemes relate to the treatment of the material time derivative which can be expressed in terms of a spatial time derivative and convection, *e.g.* for field e in incompressible flow

$$\frac{De}{Dt} = \frac{\partial e}{\partial t} + \mathbf{U} \bullet \nabla e = \frac{\partial e}{\partial t} + \nabla \bullet (\mathbf{U}e) - (\nabla \bullet \mathbf{U})e \tag{4.1}$$

For numerical solution of incompressible flows, $\nabla \bullet \mathbf{U} = 0$ at convergence, at which point the third term on the right hand side is zero. Before convergence is reached, however, $\nabla \bullet \mathbf{U} \neq 0$ and in some circumstances, particularly steady-state simulations, it is better to include the third term within a numerical solution because it helps maintain boundedness of the solution variable and promotes better convergence. The `bounded` variant of the `Gauss` scheme provides this, automatically including the discretisation of the third-term with the advection term. Example syntax is as follows, as seen in *fvSchemes* files for steady-state cases, *e.g.* for the simpleFoam tutorials

```
div(phi,U)      bounded Gauss limitedLinearV 1;
div(phi,U)      bounded Gauss linearUpwindV grad(U);
```

The schemes used for advection of scalar fields are similar to those for advection of velocity, although in general there is greater emphasis placed on boundedness than accuracy when selecting the schemes. For example, a search for schemes for advection of internal energy (`e`) reveals the following.

```
foamSearch $FOAM_TUTORIALS fvSchemes "divSchemes/div(phi,e)"

div(phi,e)      bounded Gauss upwind;
div(phi,e)      Gauss limitedLinear 1;
div(phi,e)      Gauss linearUpwind limited;
div(phi,e)      Gauss LUST grad(e);
div(phi,e)      Gauss upwind;
```

In comparison with advection of velocity, there are no cases set up to use `linear` or `linearUpwind`. Instead the `limitedLinear` and `upwind` schemes are commonly used, with the additional appearance of `vanLeer`, another limited scheme, with less strong limiting than `limitedLinear`.

There are specialised versions of the limited schemes for scalar fields that are commonly bounded between 0 and 1, *e.g.* the laminar flame speed regress variable b. A search for the discretisation used for advection in the laminar flame transport equation yields:

```
div(phiSt,b)    Gauss limitedLinear01 1;
```

The underlying scheme is `limitedLinear`, specialised for stronger bounding between 0 and 1 by adding `01` to the name of the scheme.

The `multivariateSelection` mechanism also exists for grouping multiple equation terms together, and applying the same limiters on all terms, using the strongest limiter calculated for all terms. A good example of this is in a set of mass transport equations for fluid species, where it is good practice to apply the same discretisation to all equations for consistency. The example below comes from the smallPoolFire3D tutorial in *$FOAM_TUTORIALS/combustion/fireFoam/les*, in which the equation for enthalpy h is included with the specie mass transport equations in the calculation of a single limiter.

```
div(phi,Yi_h)   Gauss multivariateSelection
{
    O2 limitedLinear01 1;
    CH4 limitedLinear01 1;
    N2 limitedLinear01 1;
    H2O limitedLinear01 1;
    CO2 limitedLinear01 1;
    h limitedLinear 1 ;
}
```

4.5.4 Surface normal gradient schemes

It is worth explaining the *snGradSchemes* sub-dictionary that contains surface normal gradient terms, before discussion of *laplacianSchemes*, because they are required to evaluate a Laplacian term using Gaussian integration. 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 search for the `default` scheme for *snGradSchemes* reveals the following entries.

```
default         corrected;
default         limited corrected 0.33;
default         limited corrected 0.5;
default         orthogonal;
default         uncorrected;
```

The basis of the gradient calculation at a face is to subtract the value at the cell centre on one side of the face from the value in the centre on the other side and divide by the distance. The calculation is second-order accurate for the gradient *normal to the face* if the vector connecting the cell centres is orthogonal to the face, *i.e.* they are at right-angles. This is the `orthogonal` scheme.

Orthogonality requires a regular mesh, typically aligned with the Cartesian co-ordinate system, which does not normally occur in meshes for real world, engineering geometries. Therefore, to maintain second-order accuracy, an explicit non-orthogonal correction can be added to the orthogonal component, known as the `corrected` scheme. The correction increases in size as the non-orthogonality, the angle α between the cell-cell vector and face normal vector, increases.

As α tends towards 90°, *e.g.* beyond 70°, the explicit correction can be so large to cause a solution to go unstable. The solution can be stabilised by applying the `limited` scheme to the correction which requires a coefficient $\psi, 0 \leq \psi \leq 1$ where

$$\psi = \begin{cases} 0 & \text{corresponds to } \texttt{uncorrected}, \\ 0.333 & \text{non-orthogonal correction} \leq 0.5 \times \text{orthogonal part}, \\ 0.5 & \text{non-orthogonal correction} \leq \text{orthogonal part}, \\ 1 & \text{corresponds to } \texttt{corrected}. \end{cases} \tag{4.2}$$

Typically, *psi* is chosen to be 0.33 or 0.5, where 0.33 offers greater stability and 0.5 greater accuracy.

The corrected scheme applies under-relaxation in which the implicit orthogonal calculation is increased by $cos^{-1}\alpha$, with an equivalent boost within the non-orthogonal correction. The `uncorrected` scheme is equivalent to the `corrected` scheme, without the non-orthogonal correction, so includes is like `orthogonal` but with the $cos^{-1}\alpha$ under-relaxation.

Generally the `uncorrected` and `orthogonal` schemes are only recommended for meshes with very low non-orthogonality (*e.g.* maximum 5°). The `corrected` scheme is generally recommended, but for maximum non-orthogonality above 70°, `limited` may be required. At non-orthogonality above 80°, convergence is generally hard to achieve.

4.5.5 Laplacian schemes

The *laplacianSchemes* sub-dictionary contains Laplacian terms. A typical Laplacian term is $\nabla \bullet (\nu \nabla \mathbf{U})$, the diffusion term in the momentum equations, which corresponds to the

keyword `laplacian(nu,U)` in *laplacianSchemes*. 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 user can search for the `default` scheme for *laplacianSchemes* in all the cases in the *$FOAM_TUTORIALS* directory.

```
foamSearch $FOAM_TUTORIALS fvSchemes laplacianSchemes/default
```

It reveals the following entries.

```
default         Gauss linear corrected;
default         Gauss linear limited corrected 0.33;
default         Gauss linear limited corrected 0.5;
default         Gauss linear orthogonal;
default         Gauss linear uncorrected;
```

In all cases, the `linear` interpolation scheme is used for interpolation of the diffusivity. The cases uses the same array of `snGradSchemes` based on level on non-orthogonality, as described in section 4.5.4.

4.5.6 Interpolation schemes

The *interpolationSchemes* sub-dictionary contains terms that are interpolations of values typically from cell centres to face centres, primarily used in the interpolation of velocity to face centres for the calculation of flux `phi`. There are numerous interpolation schemes in OpenFOAM, but a search for the `default` scheme in all the tutorial cases reveals that `linear` interpolation is used in almost every case, except for 2-3 unusual cases, *e.g.* DNS on a regular mesh, stress analysis, where `cubic` interpolation is used.

4.6 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.

```
16
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, described in the remainder of this section that includes: *solvers*; *relaxationFactors*; and, *PISO*, *SIMPLE* or *PIMPLE*.

4.6.1 Linear solver control

The first sub-dictionary in our example is `solvers`. It specifies each linear-solver that is used for each discretised equation; here, the term *linear*-solver refers to the method of number-crunching to solve a matrix equation, as opposed to an *application* solver, such as simpleFoam which describes the entire set of equations and algorithms to solve a particular problem. The term 'linear-solver' is abbreviated to 'solver' in much of what follows; hopefully the context of the term avoids any ambiguity.

The syntax for each entry within *solvers* starts with a keyword that is of 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 relates to a sub-dictionary containing the type of solver and the parameters that the solver uses. The solver is selected through the `solver` keyword from the options listed below. The parameters, including `tolerance`, `relTol`, `preconditioner`, *etc.* are described in following sections.

- `PCG`/`PBiCGStab`: Stabilised preconditioned (bi-)conjugate gradient, for both symmetric and asymmetric matrices.
- `PCG`/`PBiCG`: preconditioned (bi-)conjugate gradient, with `PCG` for symmetric matrices, `PBiCG` for asymmetric matrices.
- `smoothSolver`: solver that uses a smoother.
- `GAMG`: generalised geometric-algebraic multi-grid.
- `diagonal`: diagonal solver for explicit systems.

The solvers distinguish between symmetric matrices and asymmetric matrices. The symmetry of the matrix depends on the terms of the equation being solved, *e.g.* time derivatives and Laplacian terms form coefficients of a symmetric matrix, whereas an advective derivative introduces asymmetry. If the user specifies a symmetric solver for an asymmetric matrix, or vice versa, an error message will be written to advise the user accordingly, *e.g.*

```
--> FOAM FATAL IO ERROR : Unknown asymmetric matrix solver PCG
Valid asymmetric matrix solvers are :
```

```
3
(
PBiCG
smoothSolver
GAMG
)
```

4.6.1.1 Solution tolerances

The matrices are sparse, meaning they predominately include coefficients of 0, in segregated, decoupled, finite volume numerics. Consequently, the solvers are generally iterative, *i.e.* they are based on reducing the equation residual over successive 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 to make it independent of the scale of the 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 *any one* 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 and defaults to a value of 1000.

Equations are very often solved multiple times within one solution step, or time step. For example, when using the PISO algorithm, a pressure equation is solved according to the number specified by `nCorrectors`, as described in section 4.6.3. Where this occurs, the solver is very often set up to use different settings when solving the particular equation for the final time, specified by a keyword that adds `Final` to the field name. For example, in the cavity tutorial in section 2.1, the solver settings for pressure are as follows.

```
p
{
    solver          PCG;
    preconditioner  DIC;
    tolerance       1e-06;
    relTol          0.05;
}

pFinal
{
```

```
        $p;
        relTol          0;
    }
```

If the case is specified to solve pressure 4 times within one time step, then the first 3 solutions would use the settings for `p` with `relTol` of 0.05, so that the cost of solving each equation is relatively low. Only when the equation is solved the final (4th) time, it solves to a residual level specified by `tolerance` (since `relTol` is 0, effectively deactivating it) for greater accuracy, but at greater cost.

4.6.1.2 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, listed below. Note that the `DIC/DILU` preconditioners are exclusively specified in the tutorials in OpenFOAM.

- `DIC/DILU`: diagonal incomplete-Cholesky (symmetric) and incomplete-LU (asymmetric)
- `FDIC`: faster diagonal incomplete-Cholesky (`DIC` with caching, symmetric)
- `diagonal`: diagonal preconditioning.
- `GAMG`: geometric-algebraic multi-grid.
- `none`: no preconditioning.

4.6.1.3 Smooth solvers

The solvers that use a smoother require the choice of smoother to be specified. The smoother options are listed below. The `symGaussSeidel` and `GaussSeidel` smoothers are preferred in the tutorials.

- `GaussSeidel`: Gauss-Seidel.
- `symGaussSeidel`: symmetric Gauss-Seidel.
- `DIC/DILU`: diagonal incomplete-Cholesky (symmetric), incomplete-LU (asymmetric).
- `DICGaussSeidel`: diagonal incomplete-Cholesky/LU with Gauss-Seidel (symmetric/-asymmetric).

When using the smooth solvers, the user can optionally specify the number of sweeps, by the `nSweeps` keyword, before the residual is recalculated. Without setting it, it reverts to a default value of 1.

4.6.1.4 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

The agglomeration of cells is performed by the method specified by the `agglomerator` keyword. The tutorials all use the default `faceAreaPair` method. The agglomeration can be controlled using the following optional entries, most of which default in the tutorials.

- `cacheAgglomeration`: switch specifying caching of the agglomeration strategy (default `true`).
- `nCellsInCoarsestLevel`: approximate mesh size at the most coarse level in terms of the number of cells (default 10).
- `directSolveCoarset`: use a direct solver at the coarsest level (default `false`).
- `mergeLevels`: keyword controls the speed at which coarsening or refinement is performed; the default is 1, which is safest, but for simple meshes, the solution speed can be increased by coarsening/refining 2 levels at a time, *i.e.* setting `mergeLevels 2`.

Smoothing is specified by the `smoother` as described in section 4.6.1.3. The number of sweeps used by the smoother at different levels of mesh density are specified by the following optional entries.

- `nPreSweeps`: number of sweeps as the algorithm is coarsening (default 0).
- `preSweepsLevelMultiplier`: multiplier for the number of sweeps between each coarsening level (default 1).
- `maxPreSweeps`: maximum number of sweeps as the algorithm is coarsening (default 4).
- `nPostSweeps`: number of sweeps as the algorithm is refining (default 2).
- `postSweepsLevelMultiplier`: multiplier for the number of sweeps between each refinement level (default 1).
- `maxPostSweeps`: maximum number of sweeps as the algorithm is refining (default 4).
- `nFinestSweeps`: number of sweeps at finest level (default 2).

4.6.2 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, as described below.

- No specified α: no under-relaxation.
- $\alpha = 1$: guaranteed matrix diagonal equality/dominance.
- α decreases, under-relaxation increases.
- $\alpha = 0$: solution does not change 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.

Relaxation factors for under-relaxation of fields are specified within a *field* sub-dictionary; relaxation factors for equation under-relaxation are within a *equations* sub-dictionary. An example is shown below from tutorial example of simpleFoam, showing typical settings for an incompressible steady-state solver. The factors are specified for pressure `p`, pressure `U`, and turbulent fields grouped using a regular expression.

```
54  relaxationFactors
55  {
56      fields
57      {
58          p               0.3;
59      }
60      equations
61      {
62          U               0.7;
63          "(k|omega|epsilon).*" 0.7;
64      }
65  }
66
67  // ************************************************************************* //
```

Another example for pimpleFoam, a transient incompressible solver, just uses under-relaxation to ensure matrix diagonal equality, typical of transient simulations.

```
60  relaxationFactors
61  {
62      equations
63      {
64          ".*"            1;
65      }
66  }
67
68
69  // ************************************************************************* //
```

4.6.3 PISO, SIMPLE and PIMPLE algorithms

Most fluid dynamics solver applications in OpenFOAM use either the pressure-implicit split-operator (PISO), the semi-implicit method for pressure-linked equations (SIMPLE) algorithms, or a combined PIMPLE algorithm. These algorithms are iterative procedures

for coupling equations for momentum and mass conservation, PISO and PIMPLE being used for transient problems and SIMPLE for steady-state.

Within in time, or solution, step, both algorithms solve a pressure equation, to enforce mass conservation, with an explicit correction to velocity to satisfy momentum conservation. They optionally begin each step by solving the momentum equation — the so-called momentum predictor.

While all the algorithms solve the same governing equations (albeit in different forms), the algorithms principally differ in how they loop over the equations. The looping is controlled by input parameters that are listed below. They are set in a dictionary named after the algorithm, *i.e.* *SIMPLE*, *PISO* or *PIMPLE*.

- `nCorrectors`: used by PISO, and PIMPLE, sets the number of times the algorithm solves the pressure equation and momentum corrector in each step; typically set to 2 or 3.

- `nNonOrthogonalCcorrectors`: used by all algorithms, specifies repeated solutions of the pressure equation, used to update the explicit non-orthogonal correction, described in section 4.5.4, of the Laplacian term $\nabla \bullet ((1/A)\nabla p)$; typically set to 0 (particularly for steady-state) or 1.

- `nOuterCorrectors`: used by PIMPLE, it enables looping over the entire system of equations within on time step, representing the total number of times the system is solved; must be ≥ 1 and is typically set to 1, replicating the PISO algorithm.

- `momentumPredictor`: switch that controls solving of the momentum predictor; typically set to `off` for some flows, including low Reynolds number and multiphase.

4.6.4 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`. These entries are generally stored in the *SIMPLE*, *PISO* or *PIMPLE* sub-dictionary and are used by those solvers that require them when the case demands it.

4.6.5 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. 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.

4.7 Case management tools

There are a set of applications and scripts that help with managing case files and help the user find and set keyword data entries in case files. The tools are described in the following sections.

4.7.1 File management scripts

The following tools help manage case files.

foamListTimes lists the time directories for a case, omitting the *0* directory by default; the `-rm` option deletes the listed time directories, so that a case can be cleaned of time directories with results by the following command.

```
foamListTimes -rm
```

foamCloneCase creates a new case, by copying the *0*, *system* and *constant* directories from an existing case; executed simply by the following command, where *oldCase* refers to an existing case directory.

```
foamCloneCase oldCase newCase
```

4.7.2 foamDictionary and foamSearch

The foamDictionary utility offer several options for writing, editing and adding keyword entries in case files. The utility is executed with an OpenFOAM case dictionary file as an argument, *e.g.* from within a case directory on the *fvSchemes* file.

```
foamDictionary system/fvSchemes
```

Without options, the utility lists all the keyword entries in the file, *e.g.* as follows for the *fvSchemes* file in the pitzDaily tutorial case for simpleFoam.

```
{
    FoamFile
    {
        version         2;
        format          ascii;
        class           dictionary;
        location        "system";
        object          fvSchemes;
    }
    ddtSchemes
    {
        default         steadyState;
    }
    gradSchemes
    {
        default         Gauss linear;
    }
    divSchemes
    {
        default         none;
        div(phi,U)      bounded Gauss linearUpwind grad(U);
        div(phi,k)      bounded Gauss limitedLinear 1;
        div(phi,epsilon) bounded Gauss limitedLinear 1;
        div(phi,omega)  bounded Gauss limitedLinear 1;
        div(phi,v2)     bounded Gauss limitedLinear 1;
        div((nuEff*dev2(T(grad(U))))) Gauss linear;
        div(nonlinearStress) Gauss linear;
    }
    laplacianSchemes
    {
        default         Gauss linear corrected;
    }
    interpolationSchemes
    {
        default         linear;
    }
    snGradSchemes
    {
        default         corrected;
```

```
    }
    wallDist
    {
        method          meshWave;
    }
}
```

The `-entry` option allows the user to print the entry for a particular keyword, *e.g.* `divSchemes` in the example below

```
foamDictionary -entry divSchemes system/fvSchemes
```

The example clearly extracts the `divSchemes` dictionary.

```
divSchemes
{
    default         none;
    div(phi,U)      bounded Gauss linearUpwind grad(U);
    div(phi,k)      bounded Gauss limitedLinear 1;
    div(phi,epsilon) bounded Gauss limitedLinear 1;
    div(phi,omega)  bounded Gauss limitedLinear 1;
    div(phi,v2)     bounded Gauss limitedLinear 1;
    div((nuEff*dev2(T(grad(U))))) Gauss linear;
    div(nonlinearStress) Gauss linear;
}
```

The "/" syntax allows access to keywords with levels of sub-dictionary. For example, the `div(phi,U)` keyword can be accessed within the `divSchemes` sub-dictionary by the following command.

```
foamDictionary -entry "divSchemes/div(phi,U)" system/fvSchemes
```

The example returns the single `divSchemes/div(phi,U)` entry.

```
div(phi,U)      bounded Gauss linearUpwind grad(U);
```

The `-value` option causes only the entry to be written.

```
foamDictionary -entry "divSchemes/div(phi,U)" -value system/fvSchemes
```

The example removes the keyword and terminating semicolon, leaving just the data.

```
bounded Gauss linearUpwind grad(U)
```

The `-keywords` option causes only the keywords to be written.

```
foamDictionary -entry divSchemes -keywords system/fvSchemes
```

The example produces a list of keywords inside the `divSchemes` dictionary.

```
default
div(phi,U)
div(phi,k)
div(phi,epsilon)
div(phi,omega)
div(phi,v2)
div((nuEff*dev2(T(grad(U)))))
div(nonlinearStress)
```

The example removes the keyword and terminating semicolon, leaving just the data.

```
bounded Gauss linearUpwind grad(U)
```

foamDictionary can set entries with the `-set` option. If the user wishes to change the `div(phi,U)` to the upwind scheme, they can enter the following.

```
foamDictionary -entry "divSchemes.div(phi,U)" \
    -set "bounded Gauss upwind" system/fvSchemes
```

An alternative "`=`" syntax can be used with the `-set` option which is particularly useful when modifying multiple entries:

```
foamDictionary -set "startFrom=startTime, startTime=0" system/controlDict
```

foamDictionary can add entries with the `-add` option. If the user wishes to add an entry named `turbulence` to `divSchemes` with the upwind scheme, they can enter the following.

```
foamDictionary -entry "divSchemes.turbulence" \
   -add "bounded Gauss upwind" system/fvSchemes
```

The foamSearch script, demonstrated extensively in section 4.5, uses foamDictionary functionality to extract and sort keyword entries from all files of a specified name in a specified dictionary. The `-c` option counts the number of entries of each type, *e.g.* the user could searche for the choice of `solver` for the `p` equation in all the *fvSolution* files in the tutorials.

```
foamSearch -c $FOAM_TUTORIALS fvSolution solvers/p/solver
```

The search shows `GAMG` to be the most common choice in all the tutorials.

```
 59 solver          GAMG;
  3 solver          PBiCG;
 18 solver          PCG;
  5 solver          smoothSolver;
```

4.7.3 The foamGet script

The foamGet script copies configuration files into a case quickly and conveniently. The user must be inside a case directory to run the script or identify the case directory with the `-case` option. Its operation can be described using an example case, *e.g.* the pitzDaily case which can be obtained as follows:

```
run
cp -r $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily .
cd pitzDaily
```

The mesh is generated for the case by going into the case directory and running blockMesh:

```
cd pitzDaily
blockMesh
```

The user might decide before running the simulation to configure some automatic post-processing as described in Section 6.2. For example, the user can list the pre-configured function objects by the following command:

```
postProcess -list
```

From the output, the user could select the `patchFlowRate` function to monitor the flow rate at the outlet patch. The *patchFlowRate* configuration file can be copied into the *system* directory using foamGet:

```
foamGet patchFlowRate
```

In order to monitor the flow through the `outlet` patch, the `patch` entry in *patchFlowRate* file should be set as follows:

```
patch    outlet;
```

The *patchFlowRate* configuration is then included in the case by adding to the `functions` sub-dictionary in the *controlDict* file:

```
functions
{
    ...
    #includeFunc writeObjects(kEpsilon:G) // existing entry
    #includeFunc patchFlowRate
}
```

4.7.4 The foamInfo script

The foamInfo script provides quick information and examples relating to a subject that the user specifies. The subject can relate to models (including boundary conditions and packaged function objects), applications and scripts. For example, it prints information about the simpleFoam solver by typing the following:

```
foamInfo simpleFoam
```

Information for the flowRateInletVelocity boundary condition can similarly be obtained by typing the following:

```
foamInfo flowRateInletVelocity
```

The output includes: the location of the source code header file for this boundary condition; the description and usage details from the header file; and, a list of example cases that use the boundary condition.

The example usage for volumetric flow rate can be copied to replace the inlet boundary condition in the pitzDaily example from Section 4.7.3. The volumetric flow rate, equivalent to a uniform flow speed of 10 m/s, is 2.54×10^{-4} m^3/s so the modified `inlet` patch entry in the *U* file in the *0* directory should be:

```
inlet
{
    type                flowRateInletVelocity;
    volumetricFlowRate  2.54e-4;
    extrapolateProfile  yes;
    value               uniform (0 0 0);
}
```

The simpleFoam solver can then be run. The solution at convergence (around 280 steps), visualised in ParaView, shows a nonuniform velocity profile at the inlet, due to the `extrapolateProfile` being switched on. The flow rate at the outlet, from the function object set up in Section 4.7.3, is written to a *surfaceFieldValue.dat* file in the *postProcessing/patchFlowRate/0* directory. The value converges towards the inlet flow rate.

Chapter 5

Mesh generation and conversion

This chapter describes all topics relating to the creation of meshes in OpenFOAM: section 5.1 gives an overview of the ways a mesh may be described in OpenFOAM; section 5.3 covers the blockMesh utility for generating simple meshes of blocks of hexahedral cells; section 5.4 covers the snappyHexMesh utility for generating complex meshes of hexahedral and split-hexahedral cells automatically from triangulated surface geometries; section 5.5 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.

5.1 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.

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.

5.1.1 Mesh specification and validity constraints

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

5.1.1.1 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 any point that is not part at least one face.*

5.1.1.2 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 5.1.

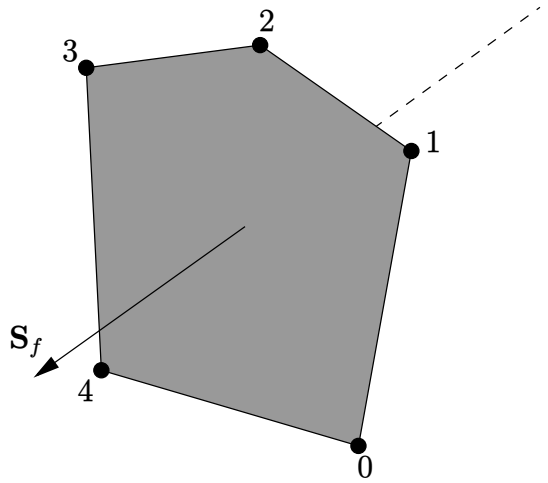


Figure 5.1: Face area vector from point numbering on the face

There are two types of face, described below.

- *Internal faces*, which 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*, which belong 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.

Note that faces can be warped, i.e. the points of the face may not necessarily lie on a plane.

5.1.1.3 Cells

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

- The cells must be *contiguous*,*i.e.* completely cover the computational domain and must not overlap one another.
- Every cell must be *closed geometrically*, such that when all face area vectors are oriented to point outwards of the cell, their sum should equal the zero vector to machine accuracy;
- Every cell must be *closed topologically* such that all the edges in a cell are used by exactly two faces of the cell in question.

5.1.1.4 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.

5.1.2 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 faces 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.

5.1.3 Cell shapes

While OpenFOAM supports any shapes of cell, other tools and software generally do not. Therefore conversion of meshes to and from OpenFOAM format often requires the use of defined cell shapes, such as tetrahedra, hexahedra, *etc.* The functionality is available in a cellShape class that uses a set of shapes defined in a *cellModels* file in the *$FOAM_ETC* directory.

Cells of a given shape are then defined by the ordering of point labels in accordance with the numbering scheme contained in the shape model. For reference, the ordering schemes for points, faces and edges are shown in Table 5.1. Any cell description then consists of two parts: the name of a cell model and the ordered list of labels. Thus, using the following list of points

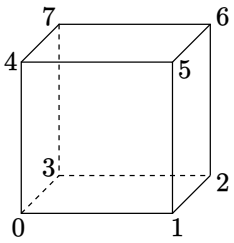
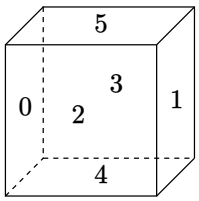
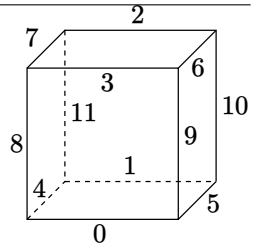
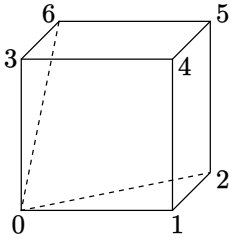
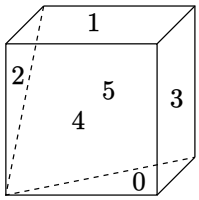
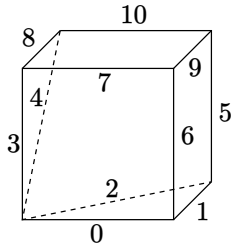
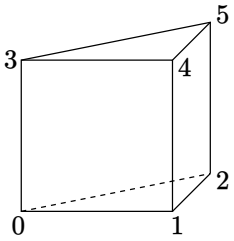
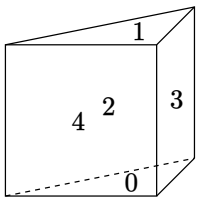
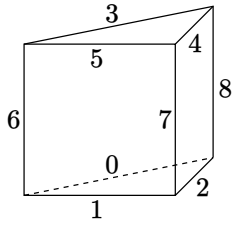
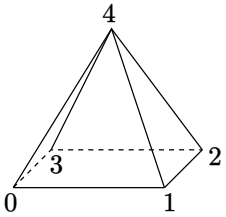
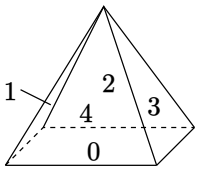
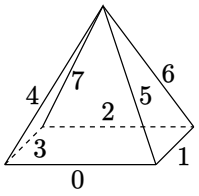
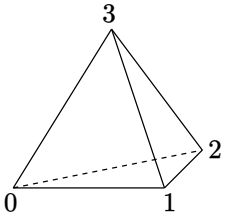
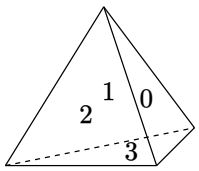
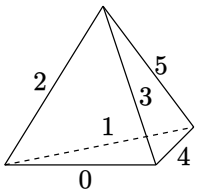
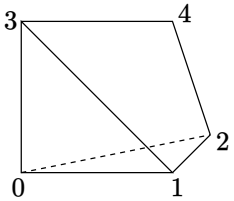
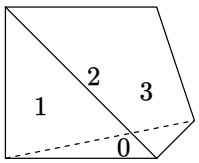
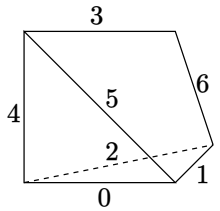
Cell type	Keyword	Vertex numbering	Face numbering	Edge numbering
Hexahedron	`hex`			
Wedge	`wedge`			
Prism	`prism`			
Pyramid	`pyr`			
Tetrahedron	`tet`			
Tet-wedge	`tetWedge`			

Table 5.1: Vertex, face and edge numbering for cellShapes.

```
(
    (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 follows using the keyword `hex`.

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

This forms the basis for the input syntax for the blockMesh mesh generator, described in section 5.3.

5.1.4 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 5.2.2 and the generation of wedge geometries for axi-symmetric problems is discussed in section 5.3.5.

5.2 Boundaries

In this section we discuss the way in which boundaries are treated in OpenFOAM. The subject of boundaries is quite complex because their role in modelling is not simply that of a geometric entity but an integral part of the solution and numerics through boundary conditions or inter-boundary 'connections'. A discussion of boundaries sits uncomfortably between a discussion on meshes, fields, discretisation, computational processing *etc.*

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. A `type` is assigned to every patch as part of the mesh description, as part of the ***boundary*** file described in section 5.1.2. It describes the type of patch in terms of geometry or a data 'communication link'. An example ***boundary*** file is shown below for a rhoPimpleFoam case. A `type` entry is clearly included for every patch (`inlet`, `outlet`, *etc.*), with types assigned that include `patch`, `symmetryPlane` and `empty`.

```
16  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
17
18  6
19  (
20      inlet
21      {
22          type            patch;
23          nFaces          50;
24          startFace       10325;
25      }
```

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

The user can scan the tutorials for mesh generation configuration files, *e.g.* *blockMeshDict* for blockMesh (see section 5.3) and *snappyHexMeshDict* for snappyHexMesh (see section 5.4, for examples of different types being used. The following example provides documentation and lists cases that use the symmetryPlane condition.

```
foamInfo -a symmetryPlane
```

The next example searches for *snappyHexMeshDict* files that specify the wall condition.

```
find $FOAM_TUTORIALS -name snappyHexMeshDict | \
    xargs grep -El "type[\t ]*wall"
```

5.2.1 Geometric (constraint) patch types

The main geometric types available in OpenFOAM are summarised below. This is not a complete list; for all types see *$FOAM_SRC/finiteVolume/fields/fvPatchFields/constraint*.

- patch: generic type containing no geometric or topological information about the mesh, *e.g.* used for an inlet or an outlet.
- wall: for patch that coincides with a solid wall, required for some physical modelling, *e.g.* wall functions in turbulence modelling.
- symmetryPlane: for a planar patch which is a symmetry plane.
- symmetry: for any (non-planar) patch which uses the symmetry plane (slip) condition.

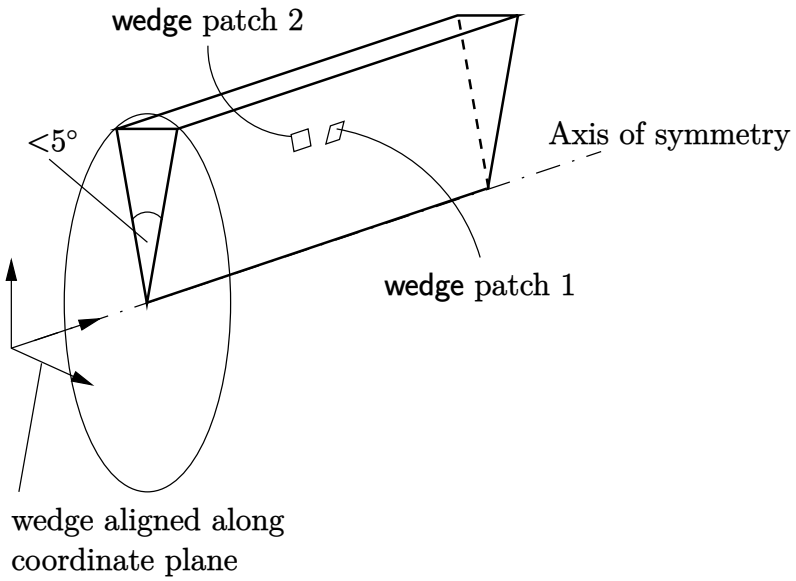


Figure 5.2: Axi-symmetric geometry using the wedge patch type.

- empty: for solutions in in 2 (or 1) dimensions (2D/1D), the type used 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.* 1°) and 1 cell thick, running along the centre line, straddling one of the coordinate planes, as shown in Figure 5.2; the axi-symmetric wedge planes must be specified as separate patches of wedge type.
- cyclic: enables two patches to be treated as if they are physically connected; used for repeated geometries; 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.
- cyclicAMI: like cyclic, but for 2 patches whose faces are non matching; used for sliding interface in rotating geometry cases.
- processor: the type that describes inter-processor boundaries for meshes decomposed for parallel running.

5.2.2 Basic boundary conditions

Boundary conditions are specified in field files, *e.g.* *p*, *U*, in time directories as described in section 4.2.8. An example pressure field file, *p*, is shown below for the rhoPimpleFoam case corresponding to the *boundary* file presented in section 5.2.1.

```
16  dimensions      [1 -1 -2 0 0 0 0];
17
18  internalField   uniform 1;
19
20  boundaryField
21  {
22      inlet
23      {
24          type            fixedValue;
25          value           uniform 1;
26      }
27
```

```
28      outlet
29      {
30          type            waveTransmissive;
31          field           p;
32          psi             thermo:psi;
33          gamma           1.4;
34          fieldInf        1;
35          lInf            3;
36          value           uniform 1;
37      }
38
39      bottom
40      {
41          type            symmetryPlane;
42      }
43
44      top
45      {
46          type            symmetryPlane;
47      }
48
49      obstacle
50      {
51          type            zeroGradient;
52      }
53
54      defaultFaces
55      {
56          type            empty;
57      }
58  }
59
60  // ************************************************************************* //
```

Every patch includes a `type` entry that specifies the type of boundary condition. They range from a basic `fixedValue` condition applied to the `inlet`, to a complex `waveTransmissive` condition applied to the `outlet`. The patches with non-generic types, *e.g.* symmetryPlane, defined in *boundary*, use consistent boundary condition types in the *p* file.

The main basic boundary condition types available in OpenFOAM are summarised below using a patch field named $\mathbf{Q}$. This is not a complete list; for all types see *$FOAM_SRC/finiteVolume/fields/fvPatchFields/basic*.

- fixedValue: value of $\mathbf{Q}$ is specified by `value`.
- fixedGradient: normal gradient of $\mathbf{Q}$ ($\partial\mathbf{Q}/\partial n$) is specified by `gradient`.
- zeroGradient: normal gradient of $\mathbf{Q}$ is zero.
- calculated: patch field $\mathbf{Q}$ calculated from other patch fields.
- mixed: mixed fixedValue/ fixedGradient condition depending on `valueFraction` ($0 \leq$ `valueFraction` ≤ 1) where

$$\texttt{valueFraction} = \begin{cases} 1 & \text{corresponds to } \mathbf{Q} = \texttt{refValue}, \\ 0 & \text{corresponds to } \partial\mathbf{Q}/\partial n = \texttt{refGradient}. \end{cases} \tag{5.1}$$

- directionMixed: mixed condition with tensorial `valueFraction`, to allow different conditions in normal and tangential directions of a vector patch field, *e.g.* fixedValue in the tangential direction, zeroGradient in the normal direction.

5.2.3 Derived types

There are numerous more complex boundary conditions derived from the basic conditions. For example, many complex conditions are derived from fixedValue, where the value is

calculated by a function of other patch fields, time, geometric information, *etc.* Some other conditions derived from mixed/directionMixed switch between fixedValue and fixedGradient (usually zeroGradient).

There are a number of ways the user can list the available boundary conditions in OpenFOAM, with the -listScalarBCs and -listVectorBCs utility being the quickest. The boundary conditions for scalar fields and vector fields, respectively, can be listed for a given solver, *e.g.* simpleFoam, as follows.

```
simpleFoam -listScalarBCs -listVectorBCs
```

These produce long lists which the user can scan through. If the user wants more information of a particular condition, they can run the foamInfo script which provides a description of the boundary condition and lists example cases where it is used. For example, for the totalPressure boundary condition, run the following.

```
foamInfo totalPressure
```

In the following sections we will highlight some particular important, commonly used boundary conditions.

5.2.3.1 The inlet/outlet condition

The inletOutlet condition is one derived from mixed, which switches between zeroGradient when the fluid flows out of the domain at a patch face, and fixedValue, when the fluid is flowing into the domain. For inflow, the inlet value is specified by an `inletValue` entry. A good example of its use can be seen in the damBreak tutorial, where it is applied to the phase fraction on the upper `atmosphere` boundary. Where there is outflow, the condition is well posed, where there is inflow, the phase fraction is fixed with a value of 0, corresponding to 100% air.

```
16  dimensions      [0 0 0 0 0 0 0];
17
18  internalField   uniform 0;
19
20  boundaryField
21  {
22      leftWall
23      {
24          type            zeroGradient;
25      }
26
27      rightWall
28      {
29          type            zeroGradient;
30      }
31
32      lowerWall
33      {
34          type            zeroGradient;
35      }
36
37      atmosphere
38      {
39          type            inletOutlet;
40          inletValue      uniform 0;
41          value           uniform 0;
42      }
43
44      defaultFaces
45      {
46          type            empty;
47      }
48  }
49
50  // ************************************************************************* //
```

5.2.3.2 Entrainment boundary conditions

The combination of the totalPressure condition on pressure and pressureInletOutletVelocity on velocity is extremely common for patches where some inflow occurs and the inlet flow velocity is not known. The conditions are used on the `atmosphere` boundary in the damBreak tutorial, inlet conditions where only pressure is known, outlets where flow reversal may occur, and where flow in entrained, *e.g.* on boundaries surrounding a jet through a nozzle.

The idea behind this combination is that the condition is a standard combination in the case of outflow, but for inflow the normal velocity is allowed to find its own value. Under these circumstances, a rapid rise in velocity presents a risk of instability, but the rise is moderated by the reduction of inlet pressure, and hence driving pressure gradient, as the inflow velocity increases.

The totalPressure condition specifies:

$$p = \begin{cases} p_0 & \text{for outflow} \\ p_0 - \frac{1}{2}\rho|\mathbf{U}^2| & \text{for inflow (dynamic pressure, subsonic)} \end{cases} \tag{5.2}$$

where the user specifies p_0 through the p0 keyword. Solver applications which include buoyancy effects, though a gravitational force $\rho\mathbf{g}$ (per unit volume) source term, tend to solve for a pressure field $p_{\rho gh} = p - \rho|\mathbf{g}|\Delta h$, where the hydrostatic component is subtracted based on a height Δh above some reference. For such solvers, *e.g.* interFoam, an equivalent prghTotalPressure condition is applied which specifies:

$$p_{\rho gh} = \begin{cases} p_0 & \text{for outflow} \\ p_0 - \rho|\mathbf{g}|\Delta h - \frac{1}{2}\rho|\mathbf{U}^2| & \text{for inflow (dynamic pressure, subsonic)} \end{cases} \tag{5.3}$$

The pressureInletOutletVelocity condition specifies zeroGradient at all times, except on the tangential component which is set to fixedValue for inflow, with the tangentialVelocity defaulting to 0.

The specification of these boundary conditions in the *U* and *p_rgh* files, in the damBreak case, are shown below.

```
16
17  dimensions      [0 1 -1 0 0 0 0];
18
19  internalField   uniform (0 0 0);
20
21  boundaryField
22  {
23      leftWall
24      {
25          type            noSlip;
26      }
27      rightWall
28      {
29          type            noSlip;
30      }
31      lowerWall
32      {
33          type            noSlip;
34      }
35      atmosphere
36      {
37          type            pressureInletOutletVelocity;
38          value           uniform (0 0 0);
39      }
40      defaultFaces
41      {
42          type            empty;
43      }
44  }
```

```
45
46
47  // ************************************************************************* //

16  dimensions      [1 -1 -2 0 0 0 0];
17
18  internalField   uniform 0;
19
20  boundaryField
21  {
22      leftWall
23      {
24          type            fixedFluxPressure;
25          value           uniform 0;
26      }
27
28      rightWall
29      {
30          type            fixedFluxPressure;
31          value           uniform 0;
32      }
33
34      lowerWall
35      {
36          type            fixedFluxPressure;
37          value           uniform 0;
38      }
39
40      atmosphere
41      {
42          type            prghTotalPressure;
43          p0              uniform 0;
44      }
45
46      defaultFaces
47      {
48          type            empty;
49      }
50  }
51
52  // ************************************************************************* //
```

5.2.3.3 Fixed flux pressure

In the above example, it can be seen that all the wall boundaries use a boundary condition named fixedFluxPressure. This boundary condition is used for pressure in situations where zeroGradient is generally used, but where body forces such as gravity and surface tension are present in the solution equations. The condition adjusts the gradient accordingly.

5.2.3.4 Time-varying boundary conditions

There are several boundary conditions for which some input parameters are specified by a function of time (using Function1 functionality) class. They can be searched by the following command.

```
find $FOAM_SRC/finiteVolume/fields/fvPatchFields -type f -name "*.H" |\
    xargs grep -l Function1 | xargs dirname | sort
```

They include conditions such as uniformFixedValue, which is a fixedValue condition which applies a single value which is a function of time through a `uniformValue` keyword entry.

The Function1 is specified by a keyword following the `uniformValue` entry, followed by parameters that relate to the particular function. The Function1 options are list below.

- `constant`: constant value.
- `table`: inline list of `(time value)` pairs; interpolates values linearly between times.

- `tableFile`: as above, but with data supplied in a separate file.
- `square`: square-wave function.
- `squarePulse`: single square pulse.
- `sine`: sine function.
- `one` and `zero`: constant one and zero values.
- `polynomial`: polynomial function using a list `(coeff exponent)` pairs.
- `coded`: function specified by user coding.
- `scale`: scales a given `value` function by a scalar `scale` function; both entries can be themselves `Function1`; `scale` function is often a ramp function (below), with `value` controlling the ramp value.
- `linearRamp`, `quadraticRamp`, `halfCosineRamp`, `quarterCosineRamp` and `quarterSineRamp`: monotonic ramp functions which ramp from 0 to 1 over specified duration.
- `reverseRamp`: reverses the values of a ramp function, e.g. from 1 to 0.

Examples or a time-varying inlet for a scalar are shown below.

```
inlet
{
    type         uniformFixedValue;
    uniformValue constant 2;
}

inlet
{
    type         uniformFixedValue;
    uniformValue table ((0 0) (10 2));
}

inlet
{
    type         uniformFixedValue;
    uniformValue polynomial ((1 0) (2 2)); // = 1*t^0 + 2*t^2
}

inlet
{
    type         uniformFixedValue;
    uniformValue
    {
        type             tableFile;
        format           csv;
        nHeaderLine      4;              // number of header lines
        refColumn        0;              // time column index
        componentColumns (1);            // data column index
        separator        ",";            // optional (defaults to ",")
        mergeSeparators  no;             // merge multiple separators
        file             "dataTable.csv";
   }
}

inlet
{
    type         uniformFixedValue;
    uniformValue
    {
        type             square;
        frequency        10;
```

```
        amplitude       1;
        scale           2;  // Scale factor for wave
        level           1;  // Offset
    }
}

inlet
{
    type         uniformFixedValue;
    uniformValue
    {
        type            sine;
        frequency       10;
        amplitude       1;
        scale           2;  // Scale factor for wave
        level           1;  // Offset
    }
}

input  // ramp from 0 -> 2, from t = 0 -> 0.4
{
    type         uniformFixedValue;
    uniformValue
    {
        type            scale;
        scale           linearRamp;
        start           0;
        duration        0.4;
        value           2;
    }
}

input  // ramp from 2 -> 0, from t = 0 -> 0.4
{
    type         uniformFixedValue;
    uniformValue
    {
        type            scale;
        scale           reverseRamp;
        ramp            linearRamp;
        start           0;
        duration        0.4;
        value           2;
    }
}

inlet  // pulse with value 2, from t = 0 -> 0.4
{
    type         uniformFixedValue;
    uniformValue
    {
        type            scale;
        scale           squarePulse
        start           0;
        duration        0.4;
        value           2;
    }
}

inlet
{
    type            uniformFixedValue;
    uniformValue    coded;
    name            pulse;
    codeInclude
    #{
        #include "mathematicalConstants.H"
    #};

    code
    #{
        return scalar
        (
            0.5*(1 - cos(constant::mathematical::twoPi*min(x/0.3, 1)))
```

```
        );
    #};
}
```

5.3 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 *system* (or *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 5.3 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 5.3.5.

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.

5.3.1 Writing a blockMeshDict file

The *blockMeshDict* file is a dictionary using keywords described below.

- `convertToMeters`: scaling factor for the vertex coordinates, *e.g.* `0.001` scales to mm.
- `vertices`: list of vertex coordinates, see section 5.3.1.1.

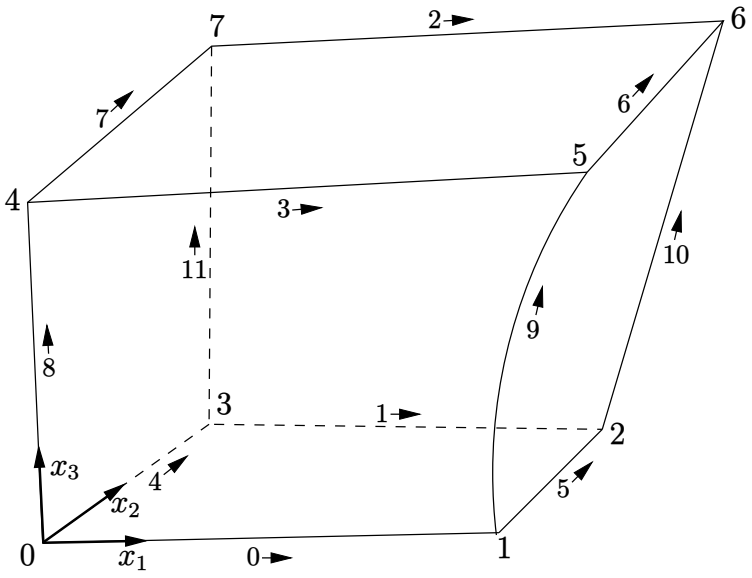


Figure 5.3: A single block

Keyword	Description	Example/selection
`convertToMeters`	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 5.3.2

Table 5.2: Keywords used in *blockMeshDict*.

- `edges`: used to describe curved geometry, see section 5.3.1.2.
- `block`: ordered list of vertex labels and mesh size, see section 5.3.1.3.
- `boundary`: sub-dictionary of boundary patches, see section 5.3.1.5.
- `mergePatchPairs`: list of patches to be merged, see section 5.3.2.

The `convertToMeters` keyword specifies a scaling factor by which all vertex coordinates in the mesh description are multiplied. For example,

```
convertToMeters     0.001;
```

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

5.3.1.1 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 5.3, 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
);
```

5.3.1.2 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 below.

- `arc`: a circular arc with a single interpolation point or angle + axis (see below).
- `spline`: spline curve using a list of interpolation points
- `polyLine`: a set of lines with list of interpolation points
- `BSpline`: a B-spline curve with list of interpolation points
- `line`: a straight line, the default which requires no edge specification.

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 an `arc`, either of the following is required: a single interpolation point, which the circular arc will intersect; or an angle and rotation axis for the arc. For `spline`, `polyLine` and `BSpline`, a list of interpolation points is required. For our example block in Figure 5.3 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)
);
```

For the angle and axis specification of an arc, the syntax is of the form:

```
edges
(
    arc 1 5 25 (0 1 0) // 25 degrees, y-normal
);
```

5.3.1.3 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 5.3, 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. The first entry is the shape identifier of the block, as defined in the *$FOAM_ETC/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-148.

The second entry gives the number of cells in each of the x_1 x_2 and x_3 directions for that block. 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 5.4.

Each of the following keywords specify one of two types of grading specification available in blockMesh.

- `simpleGrading`: The simple option 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 5.3 with the arrows representing the direction from first cell... to last cell *e.g.*
  `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.

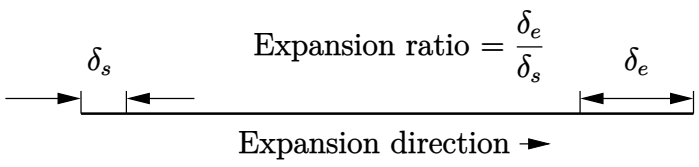


Figure 5.4: Mesh grading along a block edge

5.3.1.4 Multi-grading of a block

Using a single expansion ratio to describe mesh grading permits only "one-way" grading within a mesh block. In some cases, it reduces complexity and effort to be able to control

grading within separate divisions of a single block, rather than have to define several blocks with one grading per block. For example, to mesh a channel with two opposing walls and grade the mesh towards the walls requires three regions: two with grading to the wall with one in the middle without grading.

OpenFOAM v2.4+ includes multi-grading functionality that can divide a block in an given direction and apply different grading within each division. This multi-grading is specified by replacing any single value expansion ratio in the grading specification of the block, *e.g.* "1", "2", "3" in

```
blocks
(
    hex (0 1 2 3 4 5 6 7) (100 300 100)
    simpleGrading (1 2 3);
);
```

We will present multi-grading for the following example:

- split the block into 3 divisions in the y-direction, representing 20%, 60% and 20% of the block length;
- include 30% of the total cells in the y-direction (300) in *each* divisions 1 and 3 and the remaining 40% in division 2;
- apply 1:4 expansion in divisions 1 and 3, and zero expansion in division 2.

We can specify this by replacing the y-direction expansion ratio "2" in the example above with the following:

```
blocks
(
    hex (0 1 2 3 4 5 6 7) (100 300 100)
    simpleGrading
    (
        1                   // x-direction expansion ratio
        (
            (0.2 0.3 4)     // 20% y-dir, 30% cells, expansion = 4
            (0.6 0.4 1)     // 60% y-dir, 40% cells, expansion = 1
            (0.2 0.3 0.25) // 20% y-dir, 30% cells, expansion = 0.25 (1/4)
        )
        3                   // z-direction expansion ratio
    )
);
```

Both the fraction of the block and the fraction of the cells are normalized automatically. They can be specified as percentages, fractions, absolute lengths, *etc.* and do not need to sum to 100, 1, *etc.* The example above can be specified using percentages, *e.g.*

```
blocks
(
    hex (0 1 2 3 4 5 6 7) (100 300 100)
    simpleGrading
```

```
    (
        1
        (
            (20 30 4)   // 20%, 30%...
            (60 40 1)
            (20 30 0.25)
        )
        3
    )
);
```

5.3.1.5 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 section 5.2.1;
- `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 5.3, 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));
    }
```

5.3.2 Multiple blocks

A mesh can be created using more than 1 block. In such circumstances, the mesh is created as described in the preceeding text. The only additional issue is the connection between blocks. Firstly, if a face of one block also belongs to another block, the block face will not form an external patch but instead a set of internal faces of the cells in the resulting mesh.

Alternatively if the user wishes to combine block faces which do not exactly match one another, *i.e.* through shared vertices, they can first include the block faces within the `patches` list. Each pair of patches whose faces are to be merged can 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
    ...
)
```

See for example $FOAM_TUTORIALS/multiphase/multiphaseEulerFoam/RAS/LBend. 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 5.5, 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.

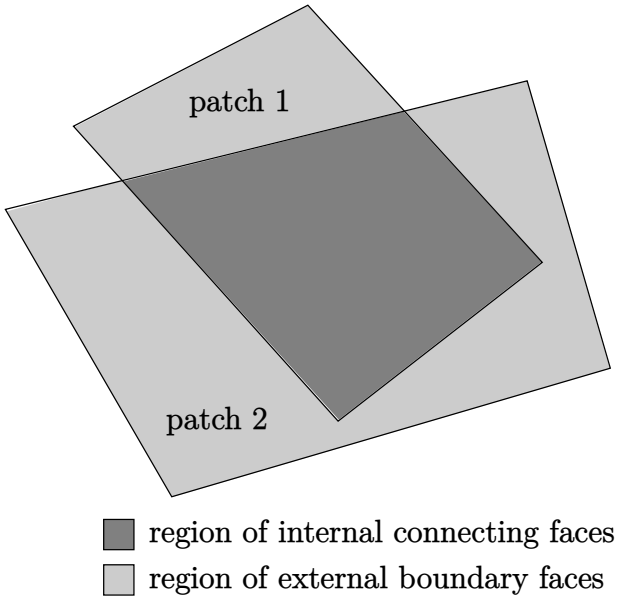


Figure 5.5: Merging overlapping patches

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.

5.3.3 Projection of vertices, edges and faces

blockMesh can be configured to create body fitted meshes using projection of vertices, edges and/or faces onto specified geometry. The functionality can be used to mesh spherical and cylindrical geometries such as pipes and vessels conveniently. The user can specify within the *blockMeshDict* file within an optional `geometry` dictionary with the same format as used in the *snappyHexMeshDict* file. For example to specify a cylinder using the built in geometric type the user could configure with the following:

```
geometry
{
    cylinder
    {
        type searchableCylinder;
        point1 (0 -4 0);
        point2 (0 4 0);
        radius 0.7;
    }
};
```

The user can then project vertices, edges and/or faces onto the cylinder surface with the `project` keyword using example syntax shown below:

```
vertices
(
    project (-1 -0.1 -1) (cylinder)
    project ( 1 -0.1 -1) (cylinder)
    ...
);

edges
(
    project 0 1 (cylinder)
    ...
);

faces
(
    project (0 4 7 3) cylinder
    ...
);
```

The use of this functionality is demonstrated in tutorials which can be located by searching for the `project` keyword in all the *blockMeshDict* files by:

```
find $FOAM_TUTORIALS -name blockMeshDict | xargs grep -l project
```

5.3.4 Naming vertices, edges, faces and blocks

Vertices, edges, faces and blocks can be named in the configuration of a *blockMeshDict* file, which can make it easier to manage more complex examples. It is done simply using the `name` keyword. The following syntax shows naming using the example for projection in the previous subsection:

```
vertices
(
    name v0 project (-1 -0.1 -1) (cylinder)
    name v1 project ( 1 -0.1 -1) (cylinder)
    ...
);

edges
(
    project v0 v1 (cylinder)
    ...
);
```

When a name is provided for a given entity, it can be used to replace the index. In the example about, rather than specify the edge using vertex indices `0` and `1`, the names `v0` and `v1` are used.

5.3.5 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 5.2.2. The process is best illustrated by using a simplified version of our example block shown in Figure 5.6. 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)
```

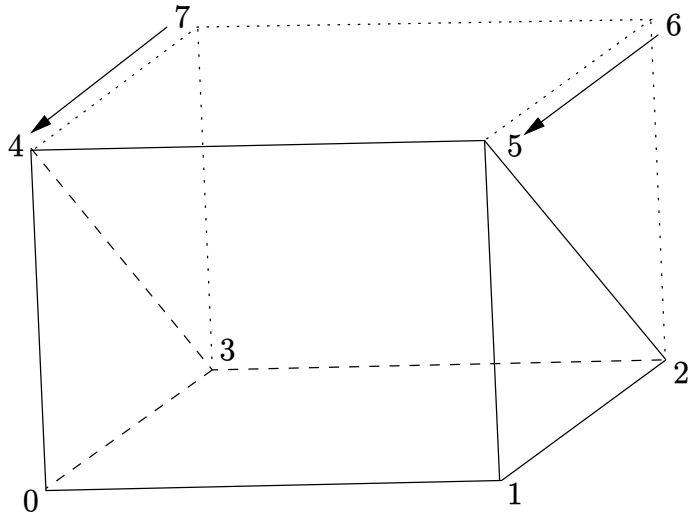


Figure 5.6: Creating a wedge shaped block with 6 vertices

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.

5.3.6 Running blockMesh

As described in section 3.3, 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 the *system* (or *constant/polyMesh*) directory.

5.4 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, or tri-surfaces, in Stereolithography (STL) or Wavefront Object (OBJ) 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.

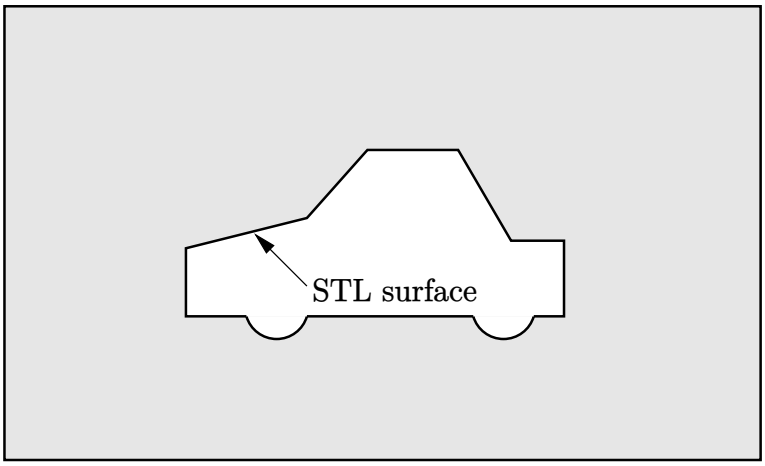


Figure 5.7: Schematic 2D meshing problem for snappyHexMesh

5.4.1 The mesh generation process of snappyHexMesh

The process of generating a mesh using snappyHexMesh will be described using the schematic in Figure 5.7. The objective is to mesh a rectangular shaped region (shaded grey in the figure) surrounding an object described by a tri-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:

- one or more tri-surface files 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 5.4.2.

- 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 below.

- `castellatedMesh`: to switch on creation of the castellated mesh.
- `snap`: to switch on surface snapping stage.
- `addLayers`: to switch on surface layer insertion.
- `mergeTolerance`: merge tolerance as fraction of bounding box of initial mesh.
- `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.

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

```
geometry
{
    sphere1     // User defined region name
    {
        type    triSurfaceMesh;
        file    "sphere1.obj";  // surface geometry OBJ file
        regions
        {
            secondSolid             // Named region in the OBJ file
            {
                name mySecondPatch; // User-defined patch name
            }                       // otherwise given sphere1_secondSolid
        }
    }

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

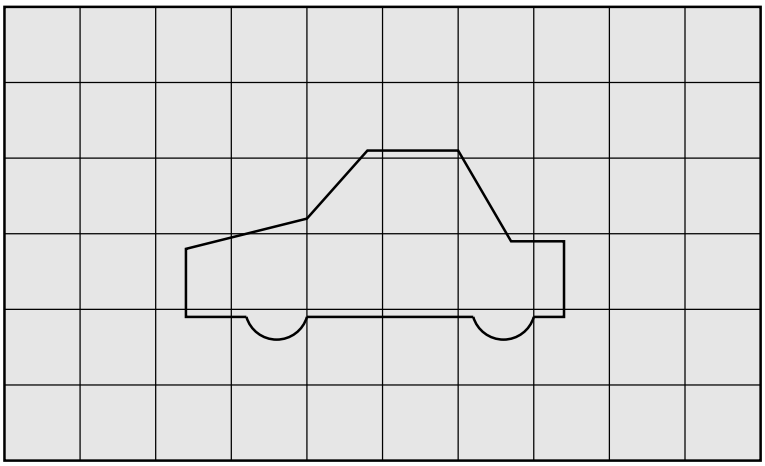


Figure 5.8: Initial mesh generation in snappyHexMesh meshing process

5.4.2 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 5.8. This can be done simply using blockMesh. The following criteria must be observed when 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 tri-surface, *i.e.* a mesh of one cell will not work.

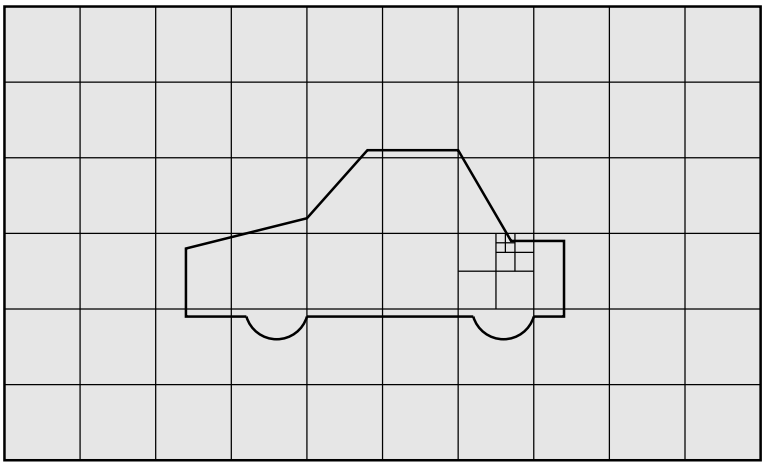


Figure 5.9: Cell splitting by feature edge in snappyHexMesh meshing process

5.4.3 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 below.

- `insidePoint`: location vector inside the region to be meshed; vector must not coincide with a cell face either before or during refinement.
- `maxLocalCells`: max number of cells per processor during refinement.
- `maxGlobalCells`: overall cell limit during refinement (*i.e.* before removal).
- `minRefinementCells`: if `minRefinementCells` $\geq$ number of cells to be refined, surface refinement stops.
- `nCellsBetweenLevels`: number of buffer layers of cells between successive levels of refinement (typically set to 3).
- `resolveFeatureAngle`: applies maximum level of refinement to cells that can see intersections whose angle exceeds `resolveFeatureAngle` (typically set to 30).
- `features`: list of features for refinement.
- `refinementSurfaces`: dictionary of surfaces for refinement.
- `refinementRegions`: dictionary of regions for refinement.

The splitting process begins with cells being selected according to specified edge features first within the domain as illustrated in Figure 5.9. 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 tri-surface file using the surfaceFeatures utility which specifies the tri-surface and controls such as included angle through a *surfaceFeaturesDict* configuration file, examples of which can be found in several tutorials and at *$FOAM_ETC/caseDicts/surface/surfaceFeaturesDict*. The utility is simply run by executing the following in a terminal

```
surfaceFeatures
```

Following feature refinement, cells are selected for splitting in the locality of specified surfaces as illustrated in Figure 5.10. 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>)`. 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
{
    sphere1
    {
        level (2 2); // default (min max) refinement for whole surface
        regions
        {
            secondSolid
            {
                level (3 3); // optional refinement for secondSolid region
            }
        }
    }
}
```

5.4.4 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 `insidePoint` 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 5.11.

5.4.5 Cell splitting in specified regions

Those cells that lie within one or more specified volume regions can be further split as illustrated in Figure 5.12 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,

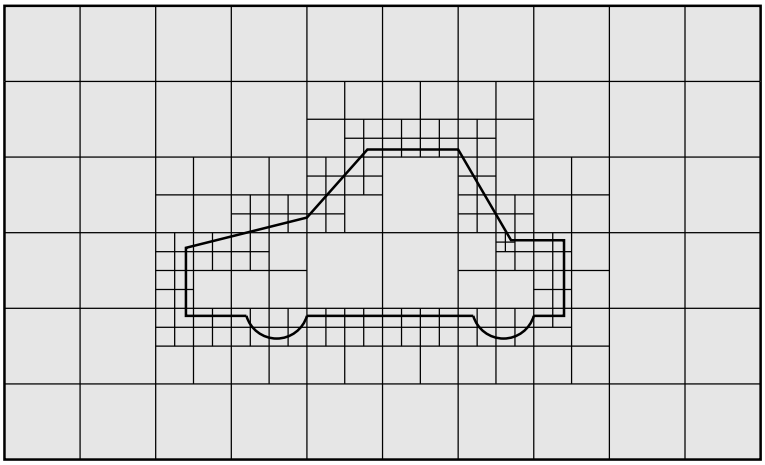


Figure 5.10: Cell splitting by surface in snappyHexMesh meshing process

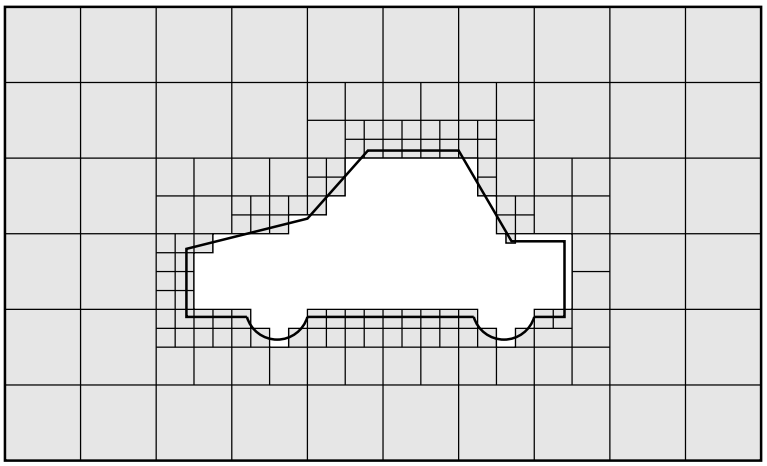


Figure 5.11: Cell removal in snappyHexMesh meshing process

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)
    }

    sphere1
    {                             // 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
    }
}
```

5.4.6 Cell splitting based on local span

Refinement of cells can also be specified to guarantee a specified number of cells across the span between opposing surfaces. This refinement option can ensure that there are sufficient cells to resolve the flow in a region of the domain, e.g. across a narrow pipe. The method requires closeness data to be provided on the surface geometry. The closeness can be calculated by the surfaceFeatures utility with the following entry in the *surfaceFeaturesDict* file:

```
surfaces
(
    "pipeWall.obj"
);

closeness
{
    pointCloseness          yes;
}
```

This writes closeness data to a file named *pipeWall.closeness.internalPointCloseness* into the *constant/triSurface* directory. The closeness is then be used for span-based refinement by the addition of an entry in the `refinementRegions` sub-dictionary in *snappyHexMeshDict*, *e.g.*:

```
refinementRegions
{
    pipeWall
```

```
    {
        mode insideSpan;
        levels ((1000 2));
        cellsAcrossSpan 40;
    }
}
```

The example shows a refinement region inside the `pipeWall` surface in which a maximum 2 levels of refinement is guaranteed within a specified distance of 1000 from the wall. The span-based refinement, specified by the `insideSpan` mode, enables the user to guarantee at least 40 `cellsAcrossSpan`, *i.e.* across the pipe diameter.

5.4.7 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:

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 below.

- `nSmoothPatch`: number of patch smoothing iterations before finding correspondence to surface (typically 3).
- `tolerance`: ratio of distance for points to be attracted by surface feature point or edge, to local maximum edge length (typically 2.0).
- `nSolveIter`: number of mesh displacement relaxation iterations (typically 30-100).
- `nRelaxIter`: maximum number of snapping relaxation iterations (typically 5).

An example is illustrated in the schematic in Figure 5.13 (albeit with mesh motion that looks slightly unrealistic).

5.4.8 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 the meshing process which introduces additional layers of hexahedral cells aligned to the boundary surface as illustrated by the dark shaded cells in Figure 5.14.

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;

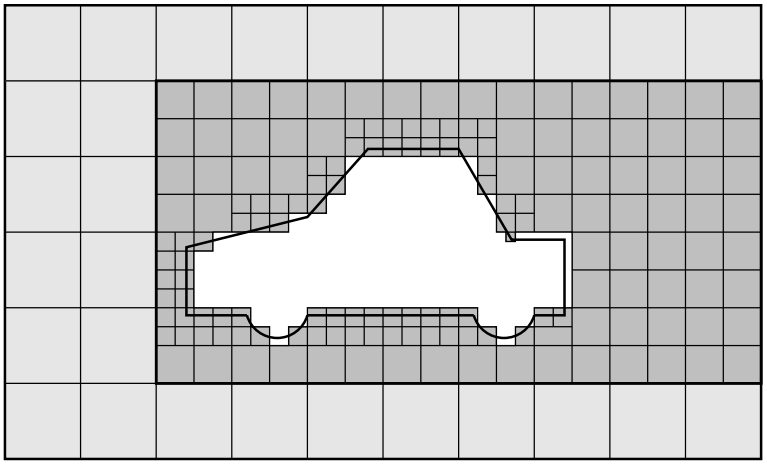


Figure 5.12: Cell splitting by region in snappyHexMesh meshing process

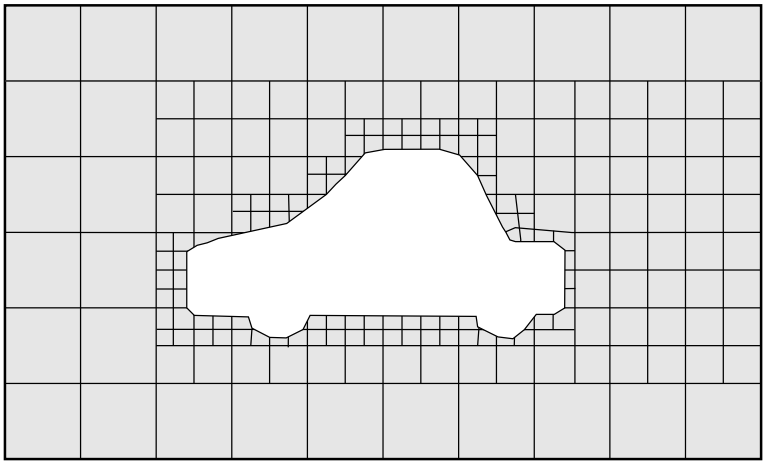


Figure 5.13: Surface snapping in snappyHexMesh meshing process

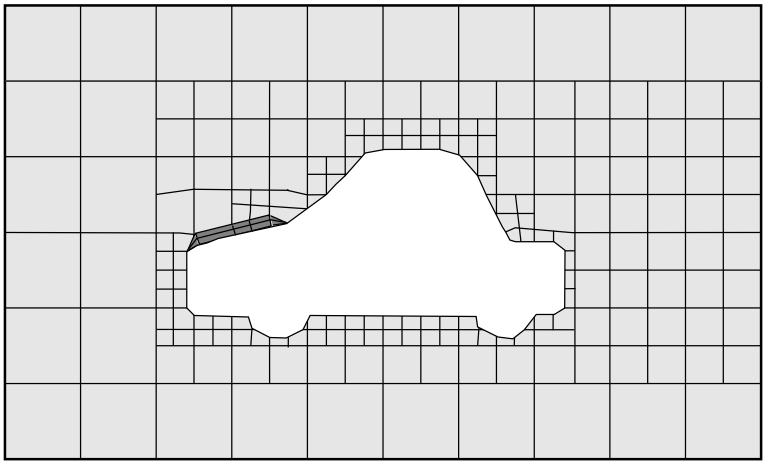


Figure 5.14: Layer addition in snappyHexMesh meshing process

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 below. The user has the option of 4 different layer thickness parameters — `expansionRatio`, `finalLayerThickness`, `firstLayerThickness`, `thickness` — *from which they must specify 2 only*; more than 2, and the problem is over-specified.

- `layers`: dictionary specifying layers to be inserted.
- `relativeSizes`: switch that sets whether the specified layer thicknesses are relative to undistorted cell size outside layer or absolute.
- `expansionRatio`: expansion factor for layer mesh, increase in size from one layer to the next.
- `finalLayerThickness`: thickness of layer furthest from the wall, usually in combination with relative sizes according to the `relativeSizes` entry.
- `firstLayerThickness`: thickness of layer nearest the wall, usually in combination with absolute sizes according to the `relativeSizes` entry.
- `thickness`: total thickness of all layers of cells, usually in combination with absolute sizes according to the
- `relativeSizes` entry.
- `minThickness`: minimum thickness of cell layer, either relative or absolute (as above).
- `nGrow`: number of layers of connected faces that are not grown if points do not get extruded; helps convergence of layer addition close to features.
- `featureAngle`: angle above which surface is not extruded.
- `nRelaxIter`: maximum number of snapping relaxation iterations (typcially 5).
- `nSmoothSurfaceNormals`: number of smoothing iterations of surface normals (typically 1).
- `nSmoothNormals`: number of smoothing iterations of interior mesh movement direction (typically 3).
- `nSmoothThickness`: smooth layer thickness over surface patches (typically 10).
- `maxFaceThicknessRatio`: stop layer growth on highly warped cells (typically 0.5).
- `maxThicknessToMedialRatio`: reduce layer growth where ratio thickness to medial distance is large (typically 0.3)
- `minMedianAxisAngle`: angle used to pick up medial axis points (typically 90).

- `nBufferCellsNoExtrude`: create buffer region for new layer terminations (typically 0).
- `nLayerIter`: overall max number of layer addition iterations (typically 50).
- `nRelaxedIter`: max number of iterations after which the controls in the *relaxed* sub dictionary of `meshQuality` are used (typically 20).

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
{
    sphere1_firstSolid
    {
        nSurfaceLayers 1;
    }
    maxY
    {
        nSurfaceLayers 1;
    }
}
```

5.4.9 Mesh quality controls

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

- `maxNonOrtho`: maximum non-orthogonality allowed (degrees, typically 65).
- `maxBoundarySkewness`: max boundary face skewness allowed (typically 20).
- `maxInternalSkewness`: max internal face skewness allowed (typically 4).
- `maxConcave`: max concaveness allowed (typically 80).
- `minFlatness`: ratio of minimum projected area to actual area (typically 0.5)
- `minTetQuality`: minimum quality of tetrahedral cells from cell decomposition; generally deactivated by setting large negative number since v5.0 when new barycentric tracking was introduced, which could handle negative tets.
- `minVol`: minimum cell pyramid volume (typically 1e-13, large negative number disables).
- `minArea`: minimum face area (typically -1).
- `minTwist`: minimum face twist (typically 0.05).
- `minDeterminant`: minimum normalised cell determinant; 1 = hex; ≤ 0 = illegal cell (typically 0.001).
- `minFaceWeight`: 0→0.5 (typically 0.05).
- `minVolRatio`: 0→1.0 (typically 0.01).
- `minTriangleTwist`: > 0 for Fluent compatibility (typically -1).

- `nSmoothScale`: number of error distribution iterations (typically 4).
- `errorReduction`: amount to scale back displacement at error points (typically 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.

5.5 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 section 3.6.3. 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;

starToFoam reads STAR-CD/PROSTAR mesh files.

gambitToFoam reads a GAMBIT `.neu` neutral file;

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

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

5.5.1 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.

5.5.2 starToFoam

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 5.1. 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.

5.5.2.1 General advice on conversion

We strongly recommend that the user run the STAR-CD mesh checking tools before attempting a starToFoam conversion and, after conversion, the checkMesh utility should be run on the newly converted mesh. Alternatively, starToFoam 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.

5.5.2.2 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.

5.5.2.3 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.

5.5.2.4 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.

5.5.2.5 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 *.23* file (versions prior to 3.0) or a *.bnd* file (versions 3.0 and higher). For cells, the command

```
CWRITE
```

outputs the cells to a *.14* or *.cel* file and for vertices, the command

```
VWRITE
```

outputs to file a *.15* or *.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 and the user must change the file extensions: from *.23*, *.14* and *.15* (below STAR-CD version 3.0), or *.pcs*, *.cls* and *.vtx* (STAR-CD version 3.0 and above); to *.bnd*, *.cel* and *.vrt* respectively.

5.5.2.6 Problems with the .vrt file

The *.vrt* file is written in columns of data of specified width, rather than free format. A typical line of data might be as follows, giving a vertex number followed by the coordinates:

```
19422       -0.105988957     -0.413711881E-02 0.000000000E+00
```

If the ordinates are written in scientific notation and are negative, there may be no space between values, *e.g.*:

```
19423       -0.953953117E-01-0.338810333E-02 0.000000000E+00
```

The starToFoam converter reads the data using spaces to delimit the ordinate values and will therefore object when reading the previous example. Therefore, OpenFOAM includes a simple script, foamCorrectVrt to insert a space between values where necessary, *i.e.* it would convert the previous example to:

```
19423       -0.953953117E-01 -0.338810333E-02 0.000000000E+00
```

The foamCorrectVrt script should therefore be executed if necessary before running the starToFoam converter, by typing:

```
foamCorrectVrt <file>.vrt
```

5.5.2.7 Converting the mesh to OpenFOAM format

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

```
starToFoam <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.

5.5.3 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.

5.5.4 ideasToFoam

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:

```
ideasToFoam <meshFile>
```

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

5.5.5 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.

5.6 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.

5.6.1 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
```

5.6.2 Mapping inconsistent fields

When the fields are not consistent, as shown in Figure 5.15, 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 5.15. 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:

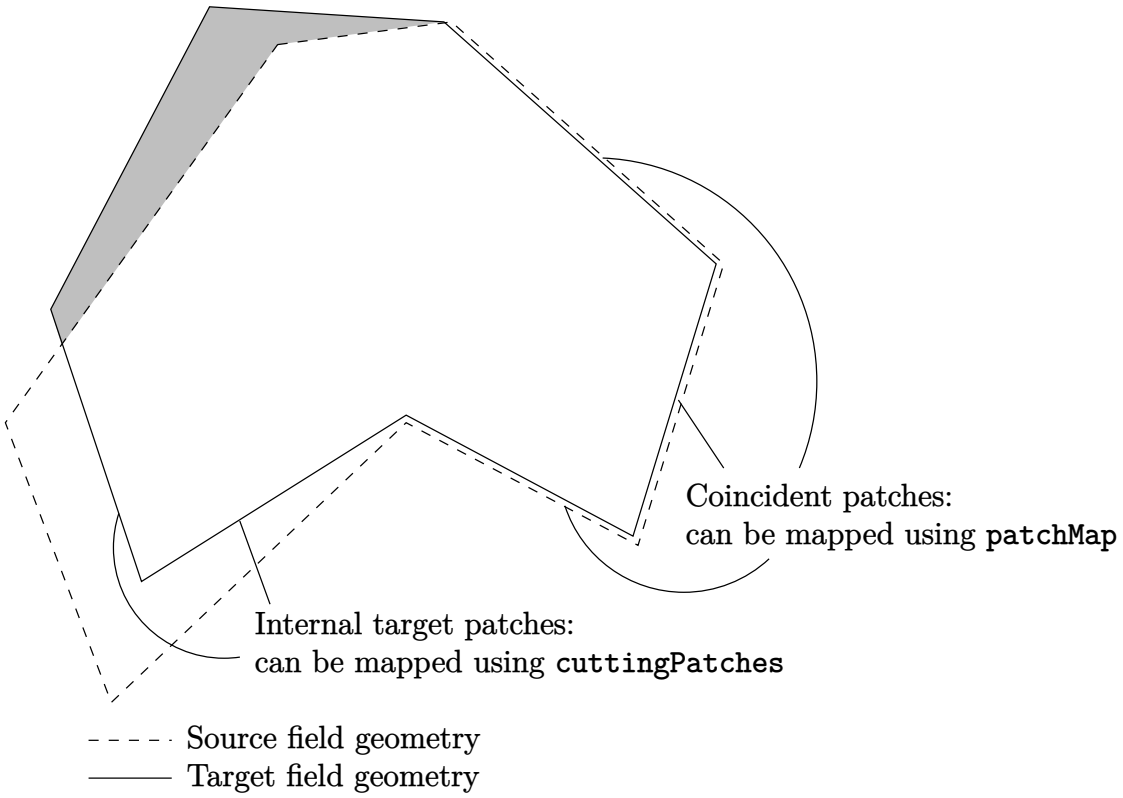


Figure 5.15: Mapping inconsistent fields

```
16
17  patchMap        (lid movingWall);
18
19  cuttingPatches  ();
20
21
22  // ************************************************************************* //
```

`mapFields <source dir>`

5.6.3 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.

Chapter 6

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 6.1.

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

6.1 ParaView/paraFoam graphical user interface (GUI)

The main post-processing tool provided with OpenFOAM is a reader module to run with ParaView, an open-source, visualization application. The module is compiled into 2 libraries, PVFoamReader and vtkPVFoam using version 5.4.0 of ParaView supplied with the OpenFOAM release. It is recommended that this version of ParaView is used, although it is possible that the latest binary release of the software will run adequately. Further details about ParaView can be found at `http://www.paraview.org`.

ParaView uses the Visualisation Toolkit (VTK) as its data processing and rendering engine and can therefore read any data in VTK format. OpenFOAM includes the foamToVTK utility to convert data from its native format to VTK format, which means that any VTK-based graphics tools can be used to post-process OpenFOAM cases. This provides an alternative means for using ParaView with OpenFOAM.

In summary, we recommend the reader module for ParaView as the primary post-processing tool for OpenFOAM. Alternatively OpenFOAM data can be converted into VTK format to be read by ParaView or any other VTK-based graphics tools.

6.1.1 Overview of ParaView/paraFoam

paraFoam is a script that launches ParaView using the reader module supplied with OpenFOAM. It is executed like any of the OpenFOAM utilities either by the single command from within the case directory or with the `-case` option with the case path as an argument, *e.g.*:

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

ParaView is launched and opens the window shown in Figure 6.1. The case is controlled from the left panel, which contains the following:

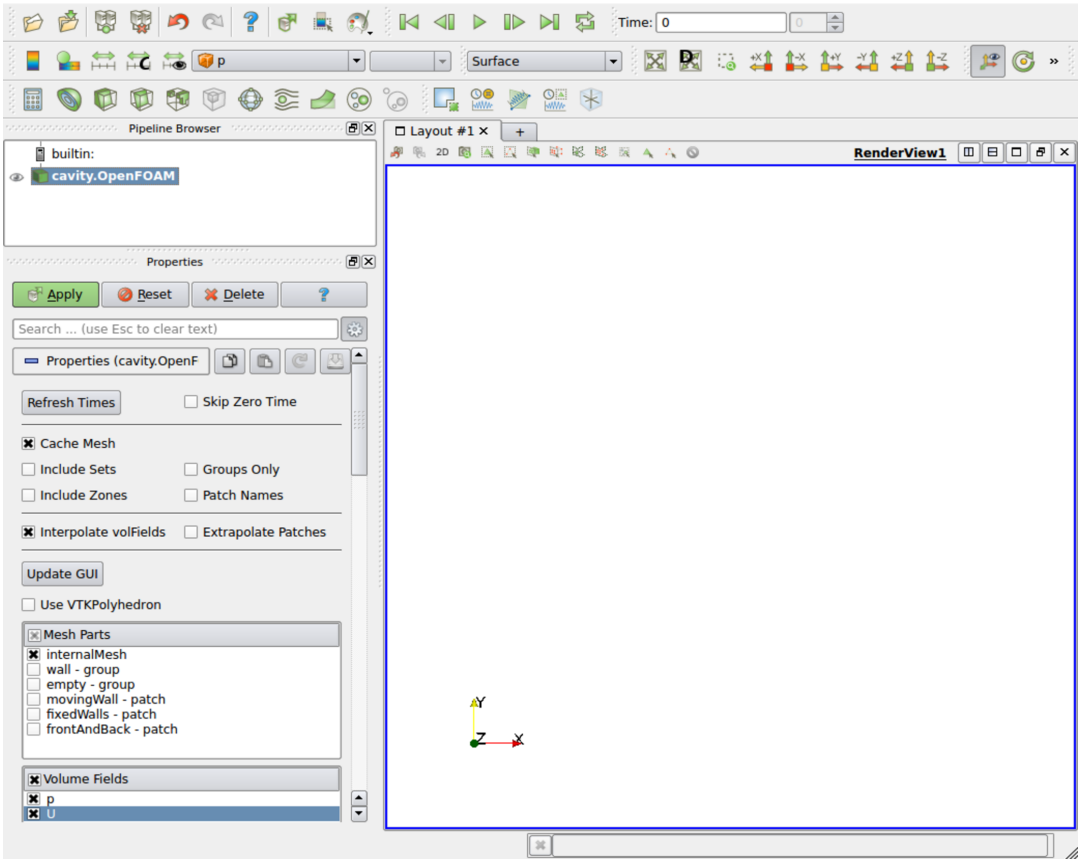


Figure 6.1: The paraFoam window

- The 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;

- The Properties panel contains the input selections for the case, such as times, regions and fields; it includes the Display panel that controls the visual representation of the selected module, *e.g.* colours;

- Other panels can be selected from the `View` menu, including the Information panel which gives case statistics such as mesh geometry and size.

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 can be used to reset the GUI if necessary; and, the Delete button that will delete the active module.

6.1.2 The Parameters panel

The Properties window for the case module includes the Parameters panel that contains the settings for mesh, fields and global controls. The controls are described in Figure 6.2. The

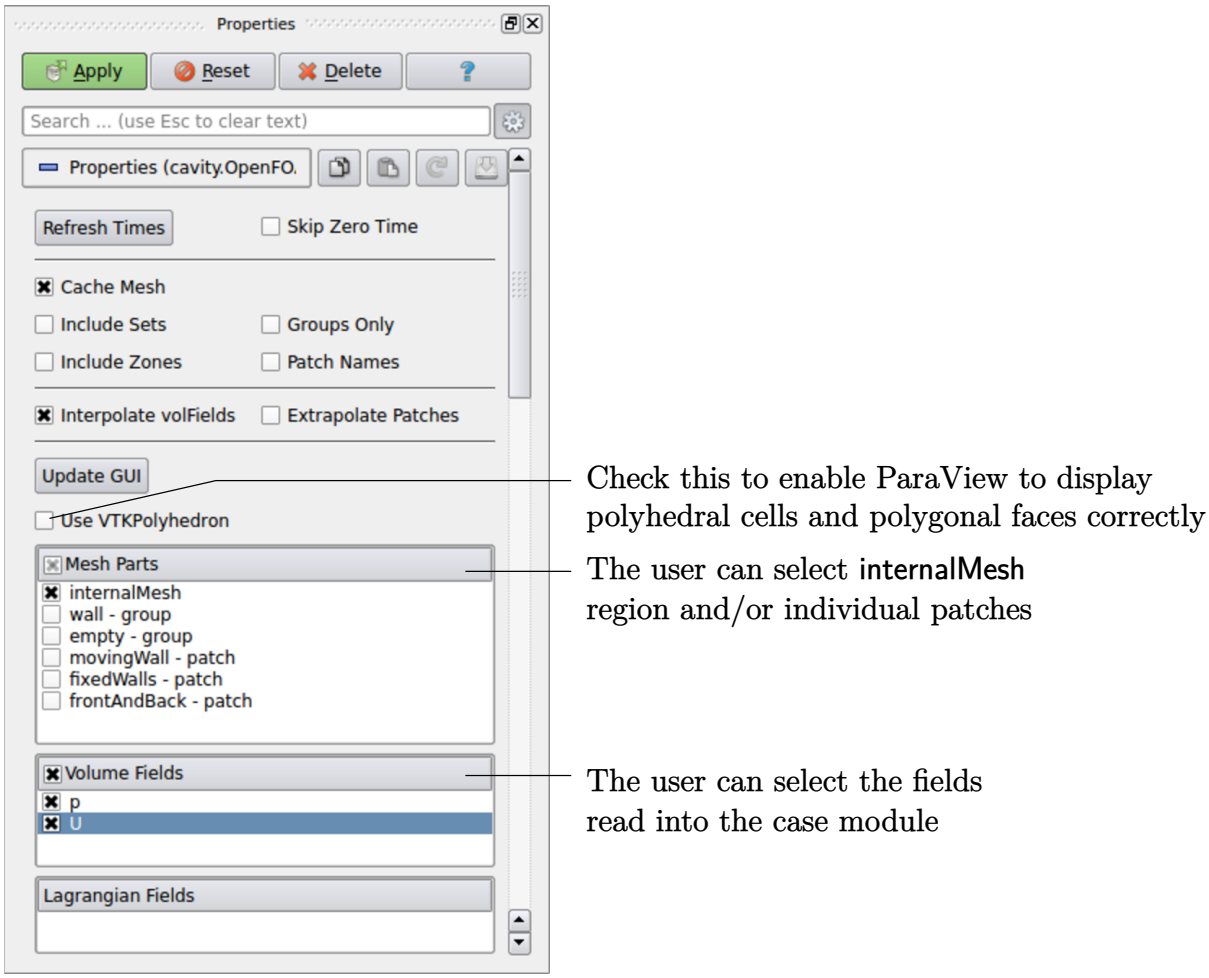


Figure 6.2: The Properties panel for the case module

user can select mesh and field data which is loaded for all time directories into ParaView. The buttons in the `Current Time Controls` and `VCR Controls` toolbars then select the time data to be displayed, as shown is section 6.1.4.

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.

If new data is written to time directories while the user is running ParaView, the user must load the additional time directories by checking the Refresh Times button. Where there are occasions when the case data changes on file and ParaView needs to load the changes, the user can also check the Update GUI button in the Parameters panel and apply the changes.

6.1.3 The Display panel

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

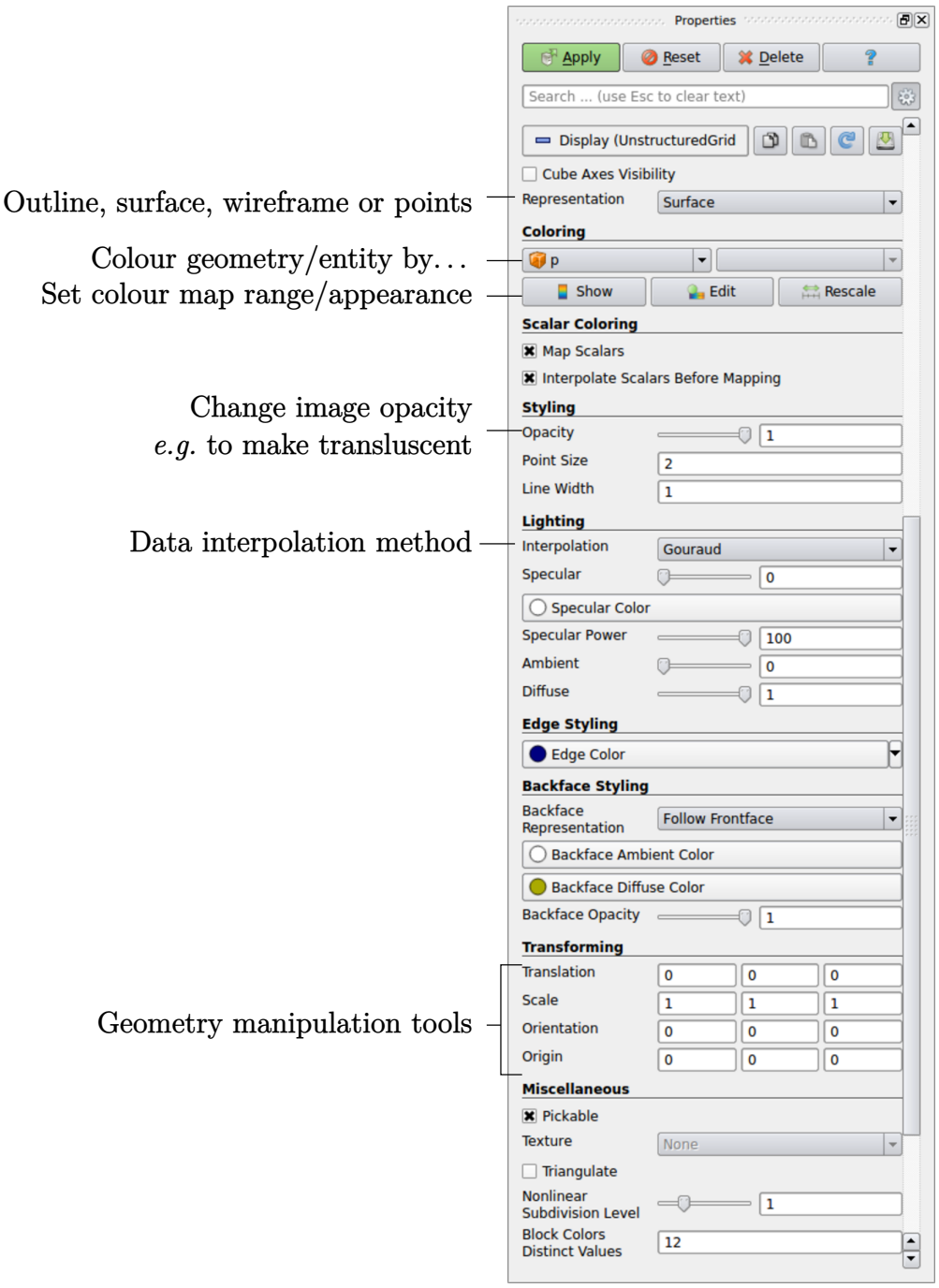


Figure 6.3: The Display panel

- 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 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;
- 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.

6.1.4 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 6.4 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.

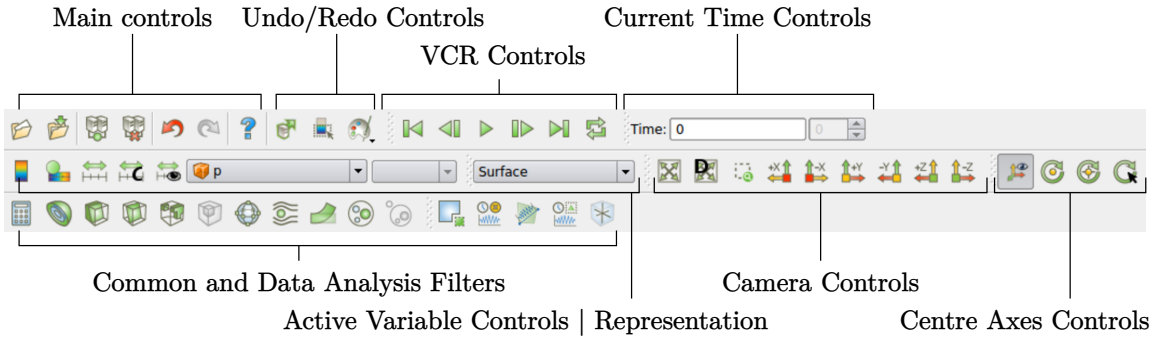


Figure 6.4: Toolbars in ParaView

6.1.5 Manipulating the view

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

6.1.5.1 View settings

The `View Settings` are available in the Render View panel below the Display panel in the Properties window. Settings that are generally important only appear when the user checks the gearwheel button at the top of the Properties window, next to the search bar. These *advanced properties* include setting the background colour, where white is often a preferred choice for creating images for printed and website material.

The Lights button opens 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 Camera Parallel Projection is the usual choice for CFD, especially for 2D cases, and so should generally be checked. Other settings include Cube Axes which displays axes on the selected object to show its orientation and geometric dimensions.

6.1.5.2 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.

6.1.6 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.

6.1.6.1 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.

6.1.7 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 for `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.

6.1.7.1 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.

6.1.8 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.

6.1.9 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.

6.1.10 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. One option is to use the built in foamCreateVideo script from the command line whose usage is shown with the `-help` option.

6.2 Post-processing command line interface (CLI)

Post-processing is provided directly within OpenFOAM through the command line including data processing, sampling (*e.g.* probes, graph plotting) visualisation, case control and run-time I/O. Functionality can be executed by:

- conventional *post-processing*, a data processing activity that occurs *after* a simulation has run;
- *run-time processing*, data processing that is performed *during* the running of a simulation.

Both approaches have advantages. Conventional post-processing allows the user to choose how to analyse data after the results are obtained. Run-time processing offers greater flexibility because it has access to *all* the data in the database of the run at all times, rather than just the data written during the simulation. It also allows the user to monitor processed data during a simulation and provides a greater level of convenience because the processed results can be available immediately to the user when the simulation ends.

There are 3 methods of post-processing that cover the options described above

- Every solver, *e.g.* simpleFoam can be configured to include run-time processing.
- The postProcess utility provides conventional post-processing of data written by a simulation.
- Every solver can be run with the `-postProcess` option, which *only* executes post-processing, but with additional access to data available on the database for the particular solver.

6.2.1 Post-processing functionality

All modes of post-processing access the same functionality implemented in OpenFOAM in the *function object* framework. Function objects can be listed by running a solver with the `-listFunctionObjects` option, *e.g.*

```
simpleFoam -listFunctionObjects
```

The list represents the underlying post-processing functionality. Almost all the functionality is packaged into a set of configured tools that are conveniently integrated within the post-processing CLI. Those tools are located in *$FOAM_ETC/caseDicts/postProcessing* and are listed by running postProcess with the `-list` option.

```
postProcess -list
```

This produces a list of tools that are described in the following sections.

6.2.1.1 Field calculation

age Calculates and writes out the time taken for a particle to travel from an inlet to the location.

components Writes the component scalar fields (*e.g.* Ux, Uy, Uz) of a field (*e.g.* U).

CourantNo Calculates the Courant Number field from the flux field.

ddt Calculates the Eulerian time derivative of a field.

div Calculates the divergence of a field.

enstrophy Calculates the enstrophy of the velocity field.

fieldAverage Calculates and writes the time averages of a given list of fields.

flowType Calculates and writes the `flowType` of velocity field where: -1 = rotational flow; 0 = simple shear flow; +1 = planar extensional flow.

grad Calculates the gradient of a field.

Lambda2 Calculates and writes the second largest eigenvalue of the sum of the square of the symmetrical and anti-symmetrical parts of the velocity gradient tensor.

log Calculates the natural logarithm of the specified scalar field.

MachNo Calculates the Mach Number field from the velocity field.

mag Calculates the magnitude of a field.

magSqr Calculates the magnitude-squared of a field.

PecletNo Calculates the Peclet Number field from the flux field.

Q Calculates the second invariant of the velocity gradient tensor.

randomise Adds a random component to a field, with a specified perturbation magnitude.

reconstruct Calculates the reconstruction of a field; *e.g.* to construct a cell-centred velocity `U` from the face-centred flux `phi`.

scale Multiplies a field by a scale factor

shearStress Calculates the shear stress, outputting the data as a `volSymmTensorField`.

streamFunction Writes the stream-function `pointScalarField`, calculated from the specified flux `surfaceScalarField`.

surfaceInterpolation Calculates the surface interpolation of a field.

totalEnthalpy Calculates and writes the total enthalpy $h_a + K$ as the `volScalarField` Ha.

turbulenceFields Calculates specified turbulence fields and stores it on the database.

turbulenceIntensity Calculates and writes the turbulence intensity field I.

vorticity Calculates the vorticity field, i.e. the curl of the velocity field.

wallHeatFlux Calculates the heat flux at wall patches, outputting the data as a `volVectorField`.

wallHeatTransferCoeff Calculates the estimated incompressible flow heat transfer coefficient at wall patches, outputting the data as a `volScalarField`.

wallShearStress Calculates the shear stress at wall patches, outputting the data as a `volVectorField`.

writeCellCentres Writes the cell-centres volVectorField and the three component fields as `volScalarFields`; useful for post-processing thresholding.

writeCellVolumes Writes the cell-volumes `volScalarField`

writeVTK Writes out specified objects in VTK format, *e.g.* fields, stored on the case database.

yPlus Calculates the turbulence y+, outputting the data as a `yPlus` field.

6.2.1.2 Field operations

add Add a list of fields.

divide From the first field, divide the remaining fields in the list.

multiply Multiply a list of fields.

subtract From the first field, subtracts the remaining fields in the list.

uniform Create a uniform field.

6.2.1.3 Forces and force coefficients

forceCoeffsCompressible Calculates lift, drag and moment coefficients by summing forces on specified patches for a case where the solver is compressible (pressure is in units M/(LT^2), *e.g.* Pa).

forceCoeffsIncompressible Calculates lift, drag and moment coefficients by summing forces on specified patches for a case where the solver is incompressible (pressure is kinematic, *e.g.* m^2/s^2).

forcesCompressible Calculates pressure and viscous forces over specified patches for a case where the solver is compressible (pressure is in units M/(LT^2), *e.g.* Pa).

forcesIncompressible Calculates pressure and viscous forces over specified patches for a case where the solver is incompressible (pressure is kinematic, *e.g.* m^2/s^2).

6.2.1.4 Sampling for graph plotting

graphCell Writes graph data for specified fields along a line, specified by start and end points. One graph point is generated in each cell that the line intersects.

graphUniform Writes graph data for specified fields along a line, specified by start and end points. A specified number of graph points are used, distributed uniformly along the line.

graphCellFace Writes graph data for specified fields along a line, specified by start and end points. One graph point is generated on each face and in each cell that the line intersects.

graphFace Writes graph data for specified fields along a line, specified by start and end points. One graph point is generated on each face that the line intersects.

graphLayerAverage Generates plots of fields averaged over the layers in the mesh

6.2.1.5 Lagrangian data

dsmcFields Calculate intensive fields `UMean`, `translationalT`, `internalT`, `overallT` from averaged extensive fields from a DSMC calculation.

6.2.1.6 Monitoring minima and maxima

cellMax Writes out the maximum cell value for one or more fields.

cellMaxMag Writes out the maximum cell value magnitude for one or more fields.

cellMin Writes out the minimum cell value for one or more fields.

cellMinMag Writes out the maximum cell value magnitude for one or more fields.

6.2.1.7 Numerical data

residuals For specified fields, writes out the initial residuals for the first solution of each time step; for non-scalar fields (*e.g.* vectors), writes the largest of the residuals for each component (*e.g.* x, y, z).

6.2.1.8 Control

stopAtClockTime Stops the run when the specified clock time in second has been reached and optionally write results before stopping.

stopAtFile Stops the run when the file *stop* is created in the case directory.

time Writes run time, CPU time and clock time and optionally the CPU and clock times per time step.

timeStep Writes the time step to a file for monitoring.

writeObjects Writes out specified objects, *e.g.* fields, stored on the case database.

6.2.1.9 Pressure tools

staticPressureIncompressible Calculates the pressure field in normal units, *i.e.* Pa in SI, from kinematic pressure by scaling by a specified density.

totalPressureCompressible Calculates the total pressure field in normal units, *i.e.* Pa in SI, for a case where the solver is compressible.

totalPressureIncompressible Calculates the total pressure field for a case where the solver is incompressible, in kinematic units, *i.e.* m^2/s^2 in SI.

6.2.1.10 Combustion

Qdot Calculates and outputs the heat release rate for the current combustion model.

XiReactionRate Writes the turbulent flame-speed and reaction-rate `volScalarFields` for the Xi-based combustion models.

6.2.1.11 Multiphase

populationBalanceMoments Calculates and writes out integral (integer moments) or mean properties (mean, variance, standard deviation) of a size distribution computed with multiphaseEulerFoam. Requires solver post-processing.

phaseForces Calculates the blended interfacial forces acting on a given phase, *i.e.* drag, virtual mass, lift, wall-lubrication and turbulent dispersion. Note that it works only in solver post-processing mode and in combination with multiphaseEulerFoam. For a simulation involving more than two phases, the accumulated force is calculated by looping over all `phasePairs` the phase is a part of.

phaseMap Writes the phase-fraction map field `alpha.map` with incremental value ranges for each phase e.g., with values 0 for water, 1 for air, 2 for oil, *etc.*

populationBalanceSizeDistribution Writes out the size distribution computed with multiphaseEulerFoam for the entire domain or a volume region. Requires solver post-processing.

6.2.1.12 Probes

boundaryProbes Writes out values of fields at a cloud of points, interpolated to specified boundary patches.

interfaceHeight Reports the height of the interface above a set of locations. For each location, it writes the vertical distance of the interface above both the location and the lowest boundary. It also writes the point on the interface from which these heights are computed.

internalProbes Writes out values of fields interpolated to a specified cloud of points.

probes Writes out values of fields from cells nearest to specified locations.

6.2.1.13 Surface region

faceZoneAverage Calculates the average value of one or more fields on a `faceZone`.

faceZoneFlowRate Calculates the flow rate through a specified face zone by summing the flux on patch faces. For solvers where the flux is volumetric, the flow rate is volumetric; where flux is mass flux, the flow rate is mass flow rate.

patchAverage Calculates the average value of one or more fields on a patch.

patchDifference Calculates the difference between the average values of fields on two specified patches. Calculates the average value of one or more fields on a patch.

patchFlowRate Calculates the flow rate through a specified patch by summing the flux on patch faces. For solvers where the flux is volumetric, the flow rate is volumetric; where flux is mass flux, the flow rate is mass flow rate.

patchIntegrate Calculates the surface integral of one or more fields on a patch.

triSurfaceDifference Calculates the difference between the average values of fields on two specified triangulated surfaces.

triSurfaceVolumetricFlowRate Calculates volumetric flow rate through a specified triangulated surface by interpolating velocity onto the triangles and integrating over the surface area. Triangles need to be small (<= cell size) for an accurate result.

6.2.1.14 'Pluggable' solvers

particles Tracks a cloud of parcels driven by the flow of the continuous phase.

phaseScalarTransport Solves a transport equation for a scalar field within one phase of a multiphase simulation.

scalarTransport Solves a transport equation for a scalar field.

6.2.1.15 Visualisation tools

cutPlaneSurface Writes out cut-plane surface files with interpolated field data in VTK format.

isoSurface Writes out iso-surface files with interpolated field data in VTK format.

patchSurface Writes out patch surface files with interpolated field data in VTK format.

streamlinesLine Writes out files of stream lines with interpolated field data in VTK format, with initial points uniformly distributed along a line.

streamlinesPatch Writes out files of stream lines with interpolated field data in VTK format, with initial points randomly selected within a patch.

streamlinesPoints Writes out files of stream lines with interpolated field data in VTK format, with specified initial points.

streamlinesSphere Writes out files of stream lines with interpolated field data in VTK format, with initial points randomly selected within a sphere.

6.2.2 Run-time data processing

When a user wishes to process data during a simulation, they need to configure the case accordingly. The configuration process is as follows, using an example of monitoring flow rate at an outlet patch named `outlet`.

Firstly, the user should include the flowRatePatch function in `functions` sub-dictionary in the case *controlDict* file, using the `#includeFunc` directive.

```
functions
{
    #includeFunc  flowRatePatch
    ... other function objects here ...
}
```

That will include the functionality in the *flowRatePatch* configuration file, located in the directory hierarchy beginning with *$FOAM_ETC/caseDicts/postProcessing*.

The configuration of *flowRatePatch* requires the `name` of the patch to be supplied. **Option 1** for doing this is that the user copies the *flowRatePatch* file into their case *system* directory. The foamGet script copies the file conveniently, *e.g.*

```
foamGet flowRatePatch
```

The patch `name` can be edited in the copied file to be `outlet`. When the solver is run, it will pick up an included function in the local case *system* directory, in precedence over *$FOAM_ETC/caseDicts/postProcessing*. The flow rate through the patch will be calculated and written out into a file within a directory named *postProcessing*.

Option 2 for specifying the patch name is to provide the name as an argument to the flowRatePatch in the `#includeFunc` directive, using the syntax `keyword=entry`.

```
functions
{
    #includeFunc  flowRatePatch(patch=outlet)
    ... other function objects here ...
}
```

In the case where the keyword is `field` or `fields`, only the entry is needed when specifying an argument to a function. For example, if the user wanted to calculate and write out the magnitude of velocity into time directories during a simulation they could simply add the following to the `functions` sub-dictionary in *controlDict*.

```
functions
{
    #includeFunc  mag(U)
    ... other function objects here ...
}
```

This works because the function's argument `U` is represented by the keyword `field`, see *$FOAM_ETC/caseDicts/postProcessing/fields/mag*.

Some functions require the setting of many parameters, *e.g.* to calculate forces and generate elements for visualisation, *etc.* For those functions, it is more reliable and convenient to copy and configure the function using option 1 (above) rather than through arguments.

6.2.3 The postProcess utility

The user can execute post-processing functions after the simulation is complete using the postProcess utility. Let us illustrate the use of postProcess using the pitzDaily case from the tutorials directory. The case can be copied, *e.g.* into the user's *run* directory; the user can then go into the case directory, generate the mesh with blockMesh and then run simpleFoam

```
run
cp -r $FOAM_TUTORIALS/incompressible/simpleFoam/pitzDaily .
```

```
cd pitzDaily
blockMesh
simpleFoam
```

Now the user can run execute post-processing functions with postProcess. The `-help` option provides a summary of its use.

```
postProcess -help
```

Simple functions like mag can be executed using the `-func` option; text on the command line generally needs to be quoted (`"..."`) if it contains punctuation characters.

```
postProcess -func "mag(U)"
```

This operation calculates and writes the field of magnitude of velocity into a file named *mag(U)* in each time directory. Similarly, the flowRatePatch example can be executed using postProcess.

```
postProcess -func "flowRatePatch(name=outlet)"
```

Let us say the user now wants to calculate total pressure $= p + |U|^2/2$ for incompressible flow with kinematic pressure, p. The function is available, named totalPressureIncompressible, which the user could attempt first to run as follows.

```
postProcess -func totalPressureIncompressible
```

This returns the following error message.

```
--> FOAM Warning : functionObject pressure: Cannot find required field p
```

The error message is telling the user that the pressure field *p* is not loaded; the same is true of the velocity field *U*. For the function to work, both fields can be loaded as **comma separated** arguments.

```
postProcess -func "totalPressureIncompressible(p,U)"
```

Alternatively the user can load a **space separated** list of fields using the `-fields` option, which the function can access.

```
postProcess -fields "(p U)" -func totalPressureIncompressible
```

Both options work effectively because the pressure and velocity data is available directly from the files, *p* and *U*.

6.2.4 Solver post-processing

A more complex example is calculating wall shear stress using the wallShearStress function.

```
postProcess -fields "(p U)" -func wallShearStress
```

Even loading relevant fields, the post-processing fails with the following message.

```
--> FOAM FATAL ERROR:
Unable to find turbulence model in the database
```

The message is telling us that the postProcess utility has not constructed the necessary models that the solver, simpleFoam, used when running the simulation, *i.e.* a turbulence model. This is a situation where we need to post-process (as opposed to run-time process) using the solver with the `-postProcess` option so that the modelling will be available that the post-processing function needs. Help for this operation can be printed with the following command.

```
simpleFoam -postProcess -help
```

It can be seen that the options for a solver with `-postProcess` are the same as when running postProcess utility. This means that the `-func` option can be used to execute the wallShearStress function effectively.

```
simpleFoam -postProcess -func wallShearStress
```

Note that no fields need to be supplied, either by function arguments `"(p,U)"` or using `"-fields (p U)"`, because simpleFoam itself constructs and stores the required fields. Functions can also be selected by the `#includeFunc` directive in `functions` in the *controlDict* file, instead of the `-func` option.

6.3 Sampling and monitoring data

There are a set of general post-processing functions for sampling data across the domain for graphs and visualisation. Several functions also provide data in a single file, in the form of time versus values, that can be plotted onto graphs. This time-value data can be monitored during a simulation with the foamMonitor script.

6.3.1 Probing data

The functions for probing data are boundaryProbes, internalProbes and probes as listed in section 6.2.1.12. All functions work on the basis that the user provides some point locations and a list of fields, and the function writes out values of the fields are those locations. The differences between the functions are as follows.

- probes identifies the nearest cells to the probe locations and writes out the cell values; data is written into a single file in time-value format, suitable for plotting a graph.

- boundaryProbes and internalProbes interpolate field data to the probe locations, with the locations being snapped onto boundaries for boundaryProbes; data sets are written to separate files at scheduled write times (like fields). data.

Generally probes is more suitable for monitoring values at smaller numbers of locations, whereas the other functions are typically for sampling at large numbers of locations.

As an example, the user could use the pitzDaily case set up in section 6.2.3. The probes function is best configured by copying the file to the local system directory using foamGet.

```
foamGet probes
```

The user can modify the `probeLocations` in the *probes* file as follows.

```
12
13  #includeEtc "caseDicts/postProcessing/probes/probes.cfg"
14
15  fields (p U);
16  probeLocations
17  (
18      (0.01 0 0)
19  );
20
21  // ************************************************************************* //
```

The configuration is completed by adding the `#includeFunc` directive to `functions` in the *controlDict* file.

```
functions
{
    #includeFunc  probes
    ... other function objects here ...
}
```

When simpleFoam is run, time-value data is written into *p* and *U* files in *postProcessing/probes/0*.

6.3.2 Sampling for graphs

The graphUniform function samples data for graph plotting. To use it, the *graphUniform* file can be copied into the *system* directory to be configured. We will configure it here using the pitzDaily case as before. The file is simply copied using foamGet.

```
foamGet graphUniform
```

The start and end points of the line, along which data is sampled, should be edited; the entries below provide a vertical line across the full height of the geometry 0.01 m beyond the back step.

```
13
14  start   (0.01  0.025 0);
15  end     (0.01 -0.025 0);
16  nPoints 100;
17
18  fields  (U p);
19
20  axis    distance; // The independent variable of the graph. Can be "x",
21                    // "y", "z", "xyz" (all coordinates written out), or
22                    // "distance" (from the start point).
23
24  #includeEtc "caseDicts/postProcessing/graphs/graphUniform.cfg"
25
26  // ************************************************************************* //
```

The configuration is completed by adding the `#includeFunc` directive to `functions` in the *controlDict* file.

```
functions
{
    #includeFunc  graphUniform
    ... other function objects here ...
}
```

simpleFoam can be then run; try running simply with the `-postProcess` option. Distance-value data is written into files in time directories within *postProcessing/graphUniform*. The user can quickly display the data for x-component of velocity, U_x in the last time *296*, by running gnuplot and plotting values.

```
gnuplot
gnuplot> set style data linespoints
gnuplot> plot "postProcessing/graphUniform/296/line_U.xy" u 2:1
```

This produces the graph shown in Figure 6.5. The formatting of the graph is specified

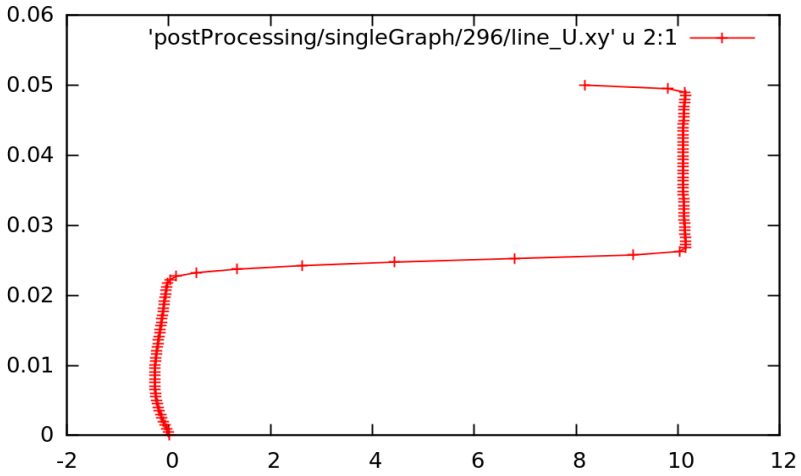


Figure 6.5: Graph of U_x at $x = 0.01$, uniform sampling

in configuration files in *$FOAM_ETC/caseDicts/postProcessing/graphs*. The *graphUniform.cfg* file in that directory includes the configuration as follows.

```
 8
 9  #includeEtc "caseDicts/postProcessing/graphs/graph.cfg"
10
11  sets
12  (
13      line
14      {
15          type            lineUniform;
16          axis            $axis;
17          start           $start;
18          end             $end;
19          nPoints         $nPoints;
20      }
21  );
22
23  // ************************************************************************* //
```

It shows that the sampling type is `lineUniform`, meaning the sampling uses a uniform distribution of points along a line. The other parameters are included by macro expansion

from the main file and specify the line start and end, the number of points and the distance parameter specified on the horizontal axis of the graph.

An alternative graph function object, *graphCell*, samples the data at locations nearest to the cell centres. The user can copy that function object file and configure it as shown below.

```
13
14  start   (0.01 -0.025 0);
15  end     (0.01  0.025 0);
16  fields  (U p);
17
18  axis    distance; // The independent variable of the graph. Can be "x",
19                    // "y", "z", "xyz" (all coordinates written out), or
20                    // "distance" (from the start point).
21
22  #includeEtc "caseDicts/postProcessing/graphs/graphCell.cfg"
23
24  // ************************************************************************* //
```

Running simpleFoam produces the graph in Figure 6.6.

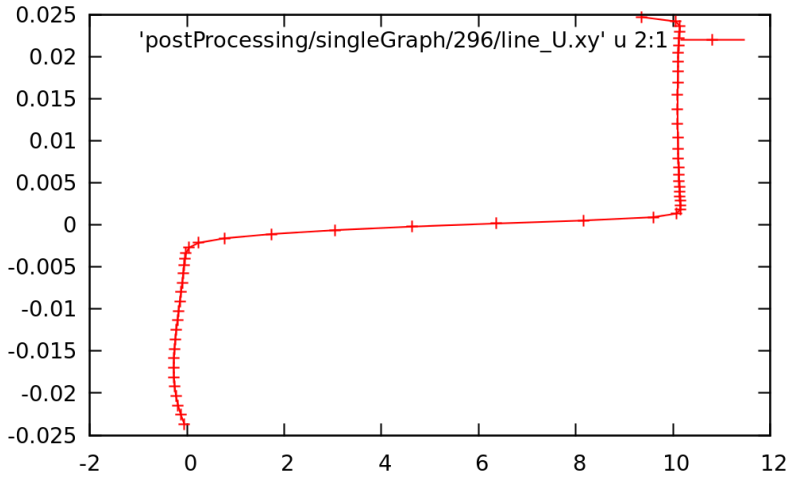


Figure 6.6: Graph of U_x at $x = 0.01$, mid-point sampling

6.3.3 Sampling for visualisation

There are several surfaces and streamlines functions, listed in Section 6.2.1.15, that can be used to generate files for visualisation. The use of streamlinesLine is already configured in the pitzDaily case.

To generate a cutting plane, the cutPlaneSurface function can be configured by copying the *cutPlaneSurface* file to the *system* directory using foamGet.

```
foamGet cutPlaneSurface
```

The file is configured by setting the origin and normal of the plane and the field data to be sampled. We can edit the file to produce a cutting plane along the pitzDaily geometry, normal to the z-direction.

```
16
17  fields      (p U);
18
19  interpolate true; // If false, write cell data to the surface triangles.
20                    // If true, write interpolated data at the surface points.
21
22  #includeEtc "caseDicts/postProcessing/surface/cutPlaneSurface.cfg"
23
24  // ************************************************************************* //
```

The function can be included as normal by adding the `#includeFunc` directive to `functions` in the *controlDict* file. Alternatively, the user could test running the function using the solver post-processing by the following command.

```
simpleFoam -postProcess -func cutPlaneSurface
```

This produces VTK format files of the cutting plane with pressure and velocity data in time directories in the *postProcessing/cutPlaneSurface* directory. The user can display the cutting plane by opening ParaView (type `paraview`), then doing `File->Open` and selecting one of the files, *e.g. postProcessing/cutPlaneSurface/296/U_zNormal.vtk* as shown in Figure 6.7.

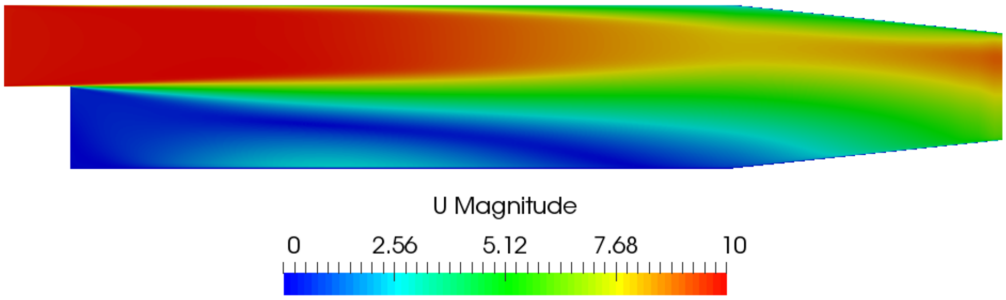


Figure 6.7: Cutting plane with velocity

6.3.4 Live monitoring of data

Functions like probes produce a single file of time-value data, suitable for graph plotting. When the function is executed during a simulation, the user may wish to monitor the data live on screen. The foamMonitor script enables this; to discover its functionality, the user run it with the `-help` option. The help option includes an example of monitoring residuals that we can demonstrate in this section.

Firstly, include the residuals function in the *controlDict* file.

```
functions
{
#includeFunc  residuals
... other function objects here ...
}
```

The default fields whose residuals are captured are *p* and *U*. Should the user wish to configure other fields, they should make copy the *residuals* file in their *system* and edit the `fields` entry accordingly. All functions files are within the *$FOAM_ETC/caseDicts* directory. The *residuals* file can be located using foamInfo:

```
foamInfo residuals
```

It can then be copied into the *system* directory conveniently using foamGet:

```
foamGet residuals
```

The user can then run simpleFoam in the background.

```
simpleFoam > log &
```

The user should then run foamMonitor using the `-l` option for a log scale y-axis on the *residuals* file as follows. If the command is executed before the simulation is complete, they can see the graph being updated live.

```
foamMonitor -l postProcessing/residuals/0/residuals.dat
```

It produces the graph of residuals for pressure and velocity in Figure 6.8.

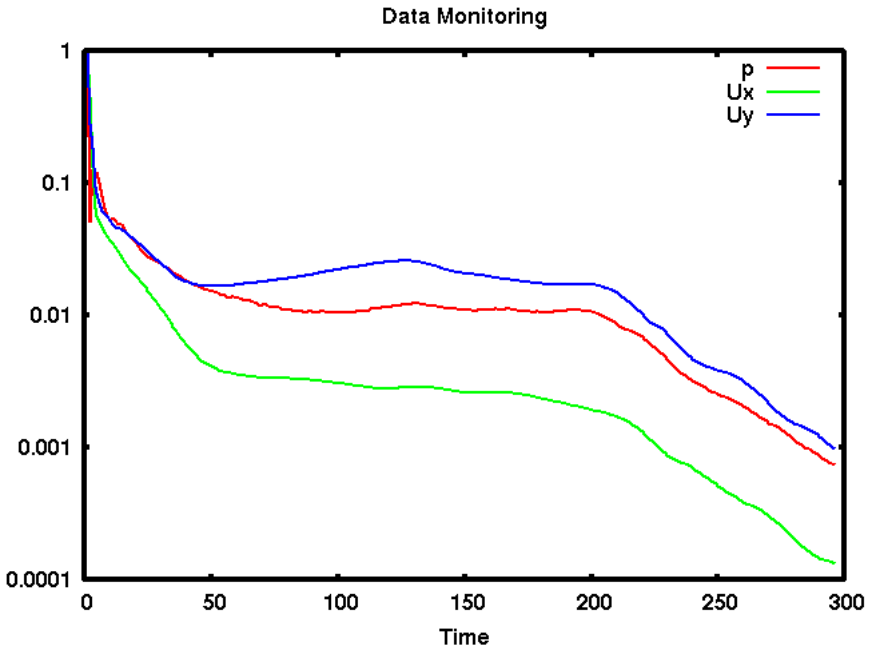


Figure 6.8: Live plot of residuals with foamMonitor

6.4 Third-Party post-processing

OpenFOAM includes the following applications for converting data to formats for post-processing with several third-party tools. For EnSight, it additionally includes a reader module, described in the next section.

foamDataToFluent Translates OpenFOAM data to Fluent format.

foamToEnsight Translates OpenFOAM data to EnSight format.

foamToEnsightParts Translates OpenFOAM data to Ensight format. An Ensight part is created for each `cellZone` and patch.

foamToGMV Translates foam output to GMV readable files.

foamToTetDualMesh Converts `polyMesh` results to `tetDualMesh`.

foamToVTK Legacy VTK file format writer.

smapToFoam Translates a STAR-CD SMAP data file into OpenFOAM field format.

6.4.1 Post-processing with Ensight

OpenFOAM offers the capability for post-processing OpenFOAM cases with EnSight, with a choice of 2 options:

- converting the OpenFOAM data to EnSight format with the foamToEnsight utility;
- reading the OpenFOAM data directly into EnSight using the ensight74FoamExec module.

6.4.1.1 Converting data to Ensight format

The foamToEnsight utility converts data from OpenFOAM to EnSight file format. For a given case, foamToEnsight is executed like any normal application. foamToEnsight creates a directory named *Ensight* in the case directory, *deleting any existing Ensight directory in the process.* The converter reads the data in all time directories and writes into a case file and a set of data files. The case file is named *EnSight_Case* and contains details of the data file names. Each data file has a name of the form *EnSight_nn.ext*, where *nn* is an incremental counter starting from 1 for the first time directory, 2 for the second and so on and *ext* is a file extension of the name of the field that the data refers to, as described in the case file, *e.g.*`T` for temperature, `mesh` for the mesh. Once converted, the data can be read into EnSight by the normal means:

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.

6.4.1.2 The ensightFoamReader reader module

EnSight provides the capability of using 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. It is this library that EnSight needs to use which means that it must be able to locate it on the filing system as described in the following section.

In order to run the EnSight reader, it is necessary to set some environment variables correctly. The settings are made in the *bashrc* (or *cshrc*) file in the *$WM_PROJECT_DIR/etc/-apps/ensightFoam* directory. The environment variables associated with EnSight are prefixed by $CEI_ or $ENSIGHT7_ and listed in Table 6.1. With a standard user setup, only $CEI_HOME may need to be set manually, to the path of the EnSight installation.

The principal difficulty in using the EnSight reader lies in the fact that EnSight expects that a case to be defined by the contents of a particular file, rather than a directory as it is in OpenFOAM. Therefore in following the 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 of a 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 6.1: 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.

Chapter 7

Models and physical properties

OpenFOAM includes a large range of solvers, each designed for a specific class of flow, as described in section 3.5. Each solver uses a particular set of models which calculate physical properties and simulate phenomena like transport, turbulence, thermal radiation, *etc.*

From OpenFOAM v10 onwards, a distinction is made between material *properties* and models for phenomena such as those mentioned above. **Properties are specified in *physicalProperties* file in the *constant* directory.** In the case of fluids, properties in *physicalProperties* relate to a fluid at rest. They are the properties you might look up from a table in a book, so can be dependent on temperature T, based on some function.

Properties described in *physicalProperties* do not include any dependency on the flow itself. For example, turbulence, visco-elasticity and the variation of viscosity ν with strain-rate, are all specified in a *momentumTransport* file in the *constant* directory. This chapter includes a description of models for viscosity which are dependent on strain-rate in section 7.3 and turbulence models in section 7.2. Thermophysical models, which are specified in the *physicalProperties* file (since they represent temperature dependency of properties) are described in section 7.1.

7.1 Thermophysical models

Thermophysical models are concerned with: thermodynamics, *e.g.* relating internal energy e to temperature T; transport, *e.g.* the dependence of properties such as ν on temperature; and state, *e.g.* dependence of density ρ on T and pressure p. Thermophysical models are specified in the *physicalProperties* dictionary.

A thermophysical model required an entry named `thermoType` which specifies the package of thermophysical modelling that is used in the simulation. OpenFOAM includes a large set of pre-compiled combinations of modelling, built within the code using C++ templates. It can also compile on-demand a combination which is not pre-compiled during a simulation.

Thermophysical modelling packages begin with the equation of state and then adding more layers of thermophysical modelling that derive properties from the previous layer(s). The keyword entries in `thermoType` reflects the multiple layers of modelling and the underlying framework in which they combined. Below is an example entry for `thermoType`:

```
thermoType
{
    type            hePsiThermo;
```

```
    mixture         pureMixture;
    transport       const;
    thermo          hConst;
    equationOfState perfectGas;
    specie          specie;
    energy          sensibleEnthalpy;
}
```

The keyword entries specify the choice of thermophysical models, *e.g.* `transport constant` (constant viscosity, thermal diffusion), `equationOfState perfectGas` , *etc.* In addition there is a keyword entry named `energy` that allows the user to specify the form of energy to be used in the solution and thermodynamics. The following sections explains the entries and options in the `thermoType` package.

7.1.1 Thermophysical and mixture models

Each solver that uses thermophysical modelling constructs an object of a specific thermophysical model class. The model classes are listed below.

fluidThermo Thermophysical model for a general fluid with fixed composition. The solvers using fluidThermo are rhoSimpleFoam, rhoPorousSimpleFoam rhoPimpleFoam, buoyantFoam, rhoParticleFoam and thermoFoam.

psiThermo Thermophysical model for gases only, with fixed composition, used by rhoCentralFoam.

fluidReactionThermo Thermophysical model for fluid of varying composition, including reactingFoam, chtMultiRegionFoam and chemFoam.

psiuReactionThermo Thermophysical model for combustion solvers that model combustion based on laminar flame speed and regress variable, *e.g.*XiFoam, PDRFoam.

multiphaseMixtureThermo Thermophysical models for multiple phases used by compressibleMultiphaseInterFoam.

solidThermo Thermophysical models for solids used by chtMultiRegionFoam.

The `type` keyword (in the *thermoType* sub-dictionary) specifies the underlying thermophysical model used by the solver. The user can select from the following.

- `hePsiThermo`: available for solvers that construct fluidThermo, fluidReactionThermo and psiThermo.
- `heRhoThermo`: available for solvers that construct fluidThermo, fluidReactionThermo and multiphaseMixtureThermo.
- `heheuPsiThermo`: for solvers that construct psiuReactionThermo.
- `heSolidThermo`: for solvers that construct solidThermo.

The `mixture` specifies the mixture composition. The option typically used for thermophysical models without reactions is `pureMixture`, which represents a mixture with fixed composition. When `pureMixture` is specified, the thermophysical models coefficients are specified within a sub-dictionary called `mixture`.

For mixtures with variable composition, required by thermophysical models with reactions, the `multicomponentMixture` option is used. Species and reactions are listed in a chemistry file, specified by the `foamChemistryFile` keyword. The `multicomponentMixture` model then requires the thermophysical models coefficients to be specified for each specie within sub-dictionaries named after each specie, *e.g.* `O2`, `N2`.

For combustion based on laminar flame speed and regress variables, constituents are a set of mixtures, such as `fuel`, `oxidant` and `burntProducts`. The available mixture models for this combustion modelling are `homogeneousMixture`, `inhomogeneousMixture` and `veryInhomogeneousMixture`.

Other models for variable composition are `egrMixture`, `singleComponentMixture`.

7.1.2 Transport model

The transport modelling concerns evaluating dynamic viscosity μ, thermal conductivity κ and thermal diffusivity α (for internal energy and enthalpy equations). The current `transport` models are as follows:

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

`sutherland` 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:

$$\mu = \frac{A_s\sqrt{T}}{1 + T_s/T}. \tag{7.1}$$

`polynomial` calculates μ and κ as a function of temperature T from a polynomial of any order N, *e.g.*:

$$\mu = \sum_{i=0}^{N-1} a_i T^i. \tag{7.2}$$

`logPolynomial` calculates $\ln(\mu)$ and $\ln(\kappa)$ as a function of $\ln(T)$ from a polynomial of any order N; from which μ, κ are calculated by taking the exponential, *e.g.*:

$$\ln(\mu) = \sum_{i=0}^{N-1} a_i [\ln(T)]^i. \tag{7.3}$$

`Andrade` calculates $\ln(\mu)$ and $\ln(\kappa)$ as a polynomial function of T, *e.g.* for μ:

$$\ln(\mu) = a_0 + a_1 T + a_2 T^2 + \frac{a_3}{a_4 + T}. \tag{7.4}$$

`tabulated` uses uniform tabulated data for viscosity and thermal conductivity as a function of pressure and temperature.

`icoTabulated` uses non-uniform tabulated data for viscosity and thermal conductivity as a function of temperature.

`WLF` (Williams-Landel-Ferry) calculates μ as a function of temperature from coefficients C_1 and C_2 and reference temperature T_r specified by keywords `C1`, `C2` and `Tr`; μ is calculated according to:

$$\mu = \mu_0 \exp\left(\frac{-C_1(T - T_r)}{C_2 + T - T_r}\right) \tag{7.5}$$

7.1.3 Thermodynamic models

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

eConst 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`.

eIcoTabulated calculates c_v by interpolating non-uniform tabulated data of (T, c_p) value pairs, *e.g.*:
```
( (200 1005) (400 1020) );
```

ePolynomial calculates c_v as a function of temperature by a polynomial of any order N:

$$c_v = \sum_{i=0}^{N-1} a_i T^i. \tag{7.6}$$

ePower calculates c_v as a power of temperature according to:

$$c_v = c_0 \left(\frac{T}{T_{\text{ref}}} \right)^{n_0}. \tag{7.7}$$

eTabulated calculates c_v by interpolating uniform tabulated data of (T, c_p) value pairs, *e.g.*:
```
( (200 1005) (400 1020) );
```

hConst 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`.

hIcoTabulated calculates c_p by interpolating non-uniform tabulated data of (T, c_p) value pairs, *e.g.*:
```
( (200 1005) (400 1020) );
```

hPolynomial calculates c_p as a function of temperature by a polynomial of any order N:

$$c_p = \sum_{i=0}^{N-1} a_i T^i. \tag{7.8}$$

hPower calculates c_p as a power of temperature according to:

$$c_p = c_0 \left(\frac{T}{T_{\text{ref}}} \right)^{n_0}. \tag{7.9}$$

hTabulated calculates c_p by interpolating uniform tabulated data of (T, c_p) value pairs, *e.g.*:
```
( (200 1005) (400 1020) );
```

janaf 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 7.1. 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:

$$c_p = R((((a_4 T + a_3)T + a_2)T + a_1)T + a_0). \tag{7.10}$$

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.

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	$a_0 \ldots a_4$	`highCpCoeffs (a0 a1 a2 a3 a4...`
High temperature enthalpy offset	a_5	`a5...`
High temperature entropy offset	a_6	`a6)`
Low temperature coefficients	$a_0 \ldots a_4$	`lowCpCoeffs (a0 a1 a2 a3 a4...`
Low temperature enthalpy offset	a_5	`a5...`
Low temperature entropy offset	a_6	`a6)`

Table 7.1: JANAF thermodynamics coefficients.

7.1.4 Composition of each constituent

There is currently only one option for the `specie` model which specifies the composition of each constituent. That model is itself named `specie`, which is specified by the following entries.

- `nMoles`: number of moles of component. This entry is only used for combustion modelling based on regress variable with a homogeneous mixture of reactants; otherwise it is set to 1.
- `molWeight` in grams per mole of specie.

7.1.5 Equation of state

The following equations of state are available in the thermophysical modelling library.

adiabaticPerfectFluid Adiabatic perfect fluid:

$$\rho = \rho_0 \left(\frac{p + B}{p_0 + B} \right)^{1/\gamma}, \tag{7.11}$$

where ρ_0, p_0 are reference density and pressure respectively, and B is a model constant.

Boussinesq Boussinesq approximation

$$\rho = \rho_0 \left[1 - \beta \left(T - T_0 \right) \right] \tag{7.12}$$

where β is the coeffient of volumetric expansion and ρ_0 is the reference density at reference temperature T_0.

icoPolynomial Incompressible, polynomial equation of state:

$$\rho = \sum_{i=0}^{N-1} a_i T^i, \tag{7.13}$$

where a_i are polynomial coefficients of any order N.

icoTabulated Tabulated data for an incompressible fluid using (T, ρ) value pairs, *e.g.*

```
rho ( (200 1010) (400 980) );
```

incompressiblePerfectGas Perfect gas for an incompressible fluid:

$$\rho = \frac{1}{RT} p_{\rm ref}, \tag{7.14}$$

where $p_{\rm ref}$ is a reference pressure.

linear Linear equation of state:

$$\rho = \psi p + \rho_0, \tag{7.15}$$

where ψ is compressibility (not necessarily $(RT)^{-1}$).

PengRobinsonGas Peng Robinson equation of state:

$$\rho = \frac{1}{zRT} p, \tag{7.16}$$

where the complex function $z = z(p, T)$ can be referenced in the source code in *PengRobinsonGasI.H*, in the *$FOAM_SRC/thermophysicalModels/specie/equationOfState/* directory.

perfectFluid Perfect fluid:

$$\rho = \frac{1}{RT} p + \rho_0, \tag{7.17}$$

where ρ_0 is the density at $T = 0$.

perfectGas Perfect gas:

$$\rho = \frac{1}{RT} p. \tag{7.18}$$

rhoConst Constant density:

$$\rho = \text{constant}. \tag{7.19}$$

rhoTabulated Uniform tabulated data for a compressible fluid, calculating ρ as a function of T and p.

rPolynomial Reciprocal polynomial equation of state for liquids and solids:

$$\frac{1}{\rho} = C_0 + C_1 T + C_2 T^2 - C_3 p - C_4 p T \tag{7.20}$$

where C_i are coefficients.

7.1.6 Selection of energy variable

The user must specify the form of energy to be used in the solution, either internal energy e and enthalpy h, and in forms that include the heat of formation Δh_f or not. This choice is specified through the `energy` keyword.

We refer to *absolute* energy where heat of formation is included, and *sensible* energy where it is not. For example absolute enthalpy h is related to sensible enthalpy h_s by

$$h = h_s + \sum_i c_i \Delta h_f^i \tag{7.21}$$

where c_i and h_f^i are the molar fraction and heat of formation, respectively, of specie i. In most cases, we use the sensible form of energy, for which it is easier to account for energy change due to reactions. Keyword entries for `energy` therefore include *e.g.* `sensibleEnthalpy`, `sensibleInternalEnergy` and `absoluteEnthalpy`.

7.1.7 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;

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

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

The following is an example entry for a specie named `fuel` modelled using sutherland transport and janaf thermodynamics:

```
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 const transport and hConst thermodynamics:

```
air
{
    specie
    {
        nMoles          1;
        molWeight       28.96;
    }
    thermodynamics
```

```
    {
        Cp              1004.5;
        Hf              2.544e+06;
    }
    transport
    {
        mu              1.8e-05;
        Pr              0.7;
    }
}
```

7.2 Turbulence models

The *momentumTransport* 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 simulation (RAS) modelling;

`LES` uses large-eddy simulation (LES) modelling.

7.2.1 Reynolds-averaged simulation (RAS) modelling

If `RAS` is selected, the choice of RAS modelling is specified in a `RAS` sub-dictionary which requires the following entries.

- `model`: name of RAS turbulence model.
- `turbulence`: switch to turn the solving of turbulence modelling on/off.
- `printCoeffs`: switch to print model coeffs to terminal at simulation start up.
- <`model`>`Coeffs`: dictionary of coefficients for the respective `model`, to override the default coefficients.

Turbulence models can be listed by running a solver with the `-listMomentumTransportModels` option, *e.g.*

```
simpleFoam -listMomentumTransportModels
```

With simpleFoam, the incompressible models are listed. The compressible models are listed for a compressible solver, *e.g.* rhoSimpleFoam.

The RAS models used in the tutorials can be listed using foamSearch with the following command. The lists of available models are given in the following sections.

```
foamSearch $FOAM_TUTORIALS momentumTransport RAS/model
```

Users can locate tutorials using a particular model, *e.g.* buoyantKEpsilon, using foamInfo.

```
foamInfo buoyantKEpsilon
```

7.2.1.1 Incompressible RAS turbulence models

For incompressible flows, the RAS `model` can be chosen from the list below.

LRR Launder, Reece and Rodi Reynolds-stress turbulence model for incompressible flows.

LamBremhorstKE Lam and Bremhorst low-Reynolds number k-epsilon turbulence model for incompressible flows.

LaunderSharmaKE Launder and Sharma low-Reynolds k-epsilon turbulence model for incompressible flows.

LienCubicKE Lien cubic non-linear low-Reynolds k-epsilon turbulence models for incompressible flows.

LienLeschziner Lien and Leschziner low-Reynolds number k-epsilon turbulence model for incompressible flows.

RNGkEpsilon Renormalization group k-epsilon turbulence model for incompressible flows.

SSG Speziale, Sarkar and Gatski Reynolds-stress turbulence model for incompressible flows.

ShihQuadraticKE Shih's quadratic algebraic Reynolds stress k-epsilon turbulence model for incompressible flows

SpalartAllmaras Spalart-Allmaras one-eqn mixing-length model for incompressible external flows.

kEpsilon Standard k-epsilon turbulence model for incompressible flows.

kOmega Standard high Reynolds-number k-omega turbulence model for incompressible flows.

kOmega2006 Standard (2006) high Reynolds-number k-omega turbulence model for incompressible flows.

kOmegaSST Implementation of the k-omega-SST turbulence model for incompressible flows.

kOmegaSSTLM Langtry-Menter 4-equation transitional SST model based on the k-omega-SST RAS model.

kOmegaSSTSAS Scale-adaptive URAS model based on the k-omega-SST RAS model.

kkLOmega Low Reynolds-number k-kl-omega turbulence model for incompressible flows.

qZeta Gibson and Dafa'Alla's q-zeta two-equation low-Re turbulence model for incompressible flows

realizableKE Realizable k-epsilon turbulence model for incompressible flows.

v2f Lien and Kalitzin's v2-f turbulence model for incompressible flows, with a limit imposed on the turbulent viscosity given by Davidson et al.

7.2.1.2 Compressible RAS turbulence models

For compressible flows, the RAS `model` can be chosen from the list below.

LRR Launder, Reece and Rodi Reynolds-stress turbulence model for compressible flows.

LaunderSharmaKE Launder and Sharma low-Reynolds k-epsilon turbulence model for compressible and combusting flows including rapid distortion theory (RDT) based compression term.

RNGkEpsilon Renormalization group k-epsilon turbulence model for compressible flows.

SSG Speziale, Sarkar and Gatski Reynolds-stress turbulence model for compressible flows.

SpalartAllmaras Spalart-Allmaras one-eqn mixing-length model for compressible external flows.

buoyantKEpsilon Additional buoyancy generation/dissipation term applied to the k and epsilon equations of the standard k-epsilon model.

kEpsilon Standard k-epsilon turbulence model for compressible flows including rapid distortion theory (RDT) based compression term.

kOmega Standard high Reynolds-number k-omega turbulence model for compressible flows.

kOmega2006 Standard (2006) high Reynolds-number k-omega turbulence model for compressible flows.

kOmegaSST Implementation of the k-omega-SST turbulence model for compressible flows.

kOmegaSSTLM Langtry-Menter 4-equation transitional SST model based on the k-omega-SST RAS model.

kOmegaSSTSAS Scale-adaptive URAS model based on the k-omega-SST RAS model.

realizableKE Realizable k-epsilon turbulence model for compressible flows.

v2f Lien and Kalitzin's v2-f turbulence model for compressible flows, with a limit imposed on the turbulent viscosity given by Davidson et al.

7.2.2 Large eddy simulation (LES) modelling

If `LES` is selected, the choice of LES modelling is specified in a `LES` sub-dictionary which requires the following entries.

- `model`: name of LES turbulence model.
- `delta`: name of delta δ model.
- <`model`>`Coeffs`: dictionary of coefficients for the respective `model`, to override the default coefficients.
- <`delta`>`Coeffs`: dictionary of coefficients for the `delta` model.

The LES models used in the tutorials can be listed using foamSearch with the following command. The lists of available models are given in the following sections.

```
foamSearch $FOAM_TUTORIALS momentumTransport LES/model
```

7.2.2.1 Incompressible LES turbulence models

For incompressible flows, the LES `model` can be chosen from the list below.

DeardorffDiffStress Differential SGS Stress Equation Model for incompressible flows

Smagorinsky The Smagorinsky SGS model.

SpalartAllmarasDDES SpalartAllmaras DDES turbulence model for incompressible flows

SpalartAllmarasDES SpalartAllmarasDES DES turbulence model for incompressible flows

SpalartAllmarasIDDES SpalartAllmaras IDDES turbulence model for incompressible flows

WALE The Wall-adapting local eddy-viscosity (WALE) SGS model.

dynamicKEqn Dynamic one equation eddy-viscosity model

dynamicLagrangian Dynamic SGS model with Lagrangian averaging

kEqn One equation eddy-viscosity model

kOmegaSSTDES Implementation of the k-omega-SST-DES turbulence model for incompressible flows.

7.2.2.2 Compressible LES turbulence models

For compressible flows, the LES `model` can be chosen from the list below.

DeardorffDiffStress Differential SGS Stress Equation Model for compressible flows

Smagorinsky The Smagorinsky SGS model.

SpalartAllmarasDDES SpalartAllmaras DDES turbulence model for compressible flows

SpalartAllmarasDES SpalartAllmarasDES DES turbulence model for compressible flows

SpalartAllmarasIDDES SpalartAllmaras IDDES turbulence model for compressible flows

WALE The Wall-adapting local eddy-viscosity (WALE) SGS model.

dynamicKEqn Dynamic one equation eddy-viscosity model

dynamicLagrangian Dynamic SGS model with Lagrangian averaging

kEqn One equation eddy-viscosity model

kOmegaSSTDES Implementation of the k-omega-SST-DES turbulence model for compressible flows.

7.2.3 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` sub-dictionary file, whose keyword name is that of the model with `Coeffs` appended, *e.g.* `kEpsilonCoeffs` for the `kEpsilon` model. If the `printCoeffs` switch is `on` in the `RAS` sub-dictionary, 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` sub-dictionary file and edit the entries as required.

7.2.4 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 the turbulent viscosity field ν_t in the *0/nut* file. For example, a *0/nut* file:

```
16
17  dimensions      [0 2 -1 0 0 0 0];
18
19  internalField   uniform 0;
20
21  boundaryField
22  {
23      movingWall
24      {
25          type            nutkWallFunction;
26          value           uniform 0;
27      }
28      fixedWalls
29      {
30          type            nutkWallFunction;
31          value           uniform 0;
32      }
33      frontAndBack
34      {
35          type            empty;
36      }
37  }
38
39
40  // ************************************************************************* //
```

There are a number of wall function models available in the release, *e.g.* `nutWallFunction`, `nutRoughWallFunction`, `nutUSpaldingWallFunction`, `nutkWallFunction` and `nutkAtmWallFunction`. The user can get the full list of wall function models using foamInfo:

```
foamInfo wallFunction
```

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*.

7.3 Transport/rheology models

In OpenFOAM, simulations that include flow without energy/heat require modelling of the fluid stress. Many simulations assume a **Newtonian fluid** in which a viscosity ν is specified in *physicalProperties*, *e.g.* by

```
viscosityModel constant;

nu              1.5e-05;
```

This viscosity is a single value which is **constant** in time and **uniform** over the solution domain. Non-Newtonian models can be specified in the *momentumTransport* file, including:

- a family of generalisedNewtonian models for a non-uniform viscosity which is a function of strain rate $\dot{\gamma} = \sqrt{2}|\,\mathrm{symm}(\nabla \mathbf{U})|$, described in sections 7.3.1, 7.3.2, 7.3.3, 7.3.4, 7.3.5 and 7.3.6;

- a set of visco-elastic models, including Maxwell, Giesekus and PTT (Phan-Thien & Tanner), described in sections 7.3.7, 7.3.8 and 7.3.9, respectively;
- the lambdaThixotropic model, described in section 7.3.10.

7.3.1 Bird-Carreau model

The Bird-Carreau generalisedNewtonian model is

$$\nu = \nu_\infty + (\nu_0 - \nu_\infty)\left[1 + (k\dot{\gamma})^a\right]^{(n-1)/a} \tag{7.22}$$

where the coefficient a has a default value of 2. An example specification of the model in *momentumTransport* is:

```
viscosityModel BirdCarreau;

nuInf   1e-05;
k       1;
n       0.5;
```

The constant, uniform viscosity at zero strain-rate, ν_0, is specified in the *physicalProperties* file.

7.3.2 Cross Power Law model

The Cross Power Law generalisedNewtonian model is:

$$\nu = \nu_\infty + \frac{\nu_0 - \nu_\infty}{1 + (m\dot{\gamma})^n} \tag{7.23}$$

An example specification of the model in *momentumTransport* is:

```
viscosityModel CrossPowerLaw;

nuInf   1e-05;
m       1;
n       0.5;
```

The constant, uniform viscosity at zero strain-rate, ν_0, is specified in the *physicalProperties* file.

7.3.3 Power Law model

The Power Law generalisedNewtonian model provides a function for viscosity, limited by minimum and maximum values, $\nu_{\min}$ and $\nu_{\max}$ respectively. The function is:

$$\nu = k\dot{\gamma}^{n-1} \quad \nu_{\min} \leq \nu \leq \nu_{\max} \tag{7.24}$$

An example specification of the model in *momentumTransport* is:

```
viscosityModel powerLaw;

nuMax    1e-03;
nuMin    1e-05;
k        1e-05;
n        0.5;
```

7.3.4 Herschel-Bulkley model

The Herschel-Bulkley generalisedNewtonian model combines the effects of Bingham plastic and power-law behavior in a fluid. For low strain rates, the material is modelled as a very viscous fluid with viscosity ν_0. Beyond a threshold in strain-rate corresponding to threshold stress τ_0, the viscosity is described by a power law. The model is:

$$\nu = \min\left(\nu_0, \tau_0/\dot{\gamma} + k\dot{\gamma}^{n-1}\right) \tag{7.25}$$

An example specification of the model in *momentumTransport* is:

```
viscosityModel HerschelBulkley;

tau0      0.01;
k         0.001;
n         0.5;
```

The constant, uniform viscosity at zero strain-rate, ν_0, is specified in the *physicalProperties* file.

7.3.5 Casson model

The Casson generalisedNewtonian model is a basic model used in blood rheology that specifies minimum and maximum viscosities, $\nu_{\min}$ and $\nu_{\max}$ respectively. Beyond a threshold in strain-rate corresponding to threshold stress τ_0, the viscosity is described by a "square-root" relationship. The model is:

$$\nu = \left(\sqrt{\tau_0/\dot{\gamma}} + \sqrt{m}\right)^2 \quad \nu_{\min} \leq \nu \leq \nu_{\max} \tag{7.26}$$

An example specification of model parameters for blood is:

```
viscosityModel Casson;

m         3.934986e-6;
tau0      2.9032e-6;
nuMax     13.3333e-6;
nuMin     3.9047e-6;
```

7.3.6 General strain-rate function

A `strainRateFunction` generalisedNewtonian model exists that allows a user to specify viscosity as a function of strain rate at run-time. It uses the same Function1 functionality to specify the function of strain-rate, used by time varying properties in boundary conditions described in section 5.2.3.4. An example specification of the model in *momentumTransport* is shown below using the `polynomial` function:

```
viscosityModel  strainRateFunction;

function polynomial ((0 0.1) (1 1.3));
```

7.3.7 Maxwell model

The Maxwell laminar visco-elastic model solves an equation for the fluid stress tensor τ:

$$\frac{\partial \tau}{\partial t} + \nabla \cdot (\mathbf{U}\tau) = 2\,\text{symm}\left[\tau \cdot \nabla \mathbf{U}\right] - 2\frac{\nu_\text{M}}{\lambda}\,\text{symm}(\nabla \mathbf{U}) - \frac{1}{\lambda}\tau \tag{7.27}$$

where ν_M (`nuM`) is the "Maxwell" viscosity and λ (`lambda`) is the relaxation time. An example specification of model parameters is shown below:

```
simulationType laminar;

laminar
{
    model               Maxwell;

    MaxwellCoeffs
    {
        nuM             0.002;
        lambda          0.03;
    }
}
```

If an additional constant, uniform viscosity at zero strain-rate, ν_0, is specified in the *physicalProperties* file, the model becomes equivalent to an Oldroyd-B visco-elastic model. The Maxwell model includes a multi-mode option where τ is a sum of stresses, each with an associated relaxation time λ.

7.3.8 Giesekus model

The Giesekus laminar visco-elastic model is similar to the Maxwell model but includes an additional "mobility" term in the equation for τ:

$$\frac{\partial \tau}{\partial t} + \nabla \cdot (\mathbf{U}\tau) = 2\,\text{symm}\left[\tau \cdot \nabla \mathbf{U}\right] - 2\frac{\nu_\text{M}}{\lambda}\,\text{symm}(\nabla \mathbf{U}) - \frac{1}{\lambda}\tau - \frac{\alpha_\text{G}}{\nu_\text{M}}\left[\tau_i \cdot \tau_i\right] \tag{7.28}$$

where α_G (`alphaG`) is the mobility parameter. An example specification of model parameters is shown below:

```
simulationType laminar;

laminar
{
    model               Giesekus;

    GiesekusCoeffs
    {
        nuM             0.002;
        lambda          0.03;
        alphaG          0.1;
    }
}
```

The Giesekus model includes a multi-mode option where τ is a sum of stresses, each with an associated relaxation time λ and mobility coefficient α_G.

7.3.9 Phan-Thien-Tanner (PTT) model

The Phan-Thien-Tanner (PTT) laminar visco-elastic model is also similar to the Maxwell model but includes an additional "extensibility" term in the equation for τ, suitable for polymeric liquids:

$$\frac{\partial \tau}{\partial t} + \nabla \bullet (\mathbf{U}\tau) = 2\,\text{symm}\left[\tau \bullet \nabla \mathbf{U}\right] - 2\frac{\nu_M}{\lambda}\,\text{symm}(\nabla \mathbf{U}) - \frac{1}{\lambda}\exp\left(-\frac{\varepsilon\lambda}{\nu_M}\,\text{tr}(\tau)\right)\tau \quad (7.29)$$

where ε (`epsilon`) is the extensibility parameter. An example specification of model parameters is shown below:

```
simulationType laminar;

laminar
{
    model               PTT;

    PTTCoeffs
    {
        nuM             0.002;
        lambda          0.03;
        epsilon         0.25;
    }
}
```

The PTT model includes a multi-mode option where τ is a sum of stresses, each with an associated relaxation time λ and extensibility coefficient ε.

7.3.10 Lambda thixotropic model

The Lambda Thixotropic laminar model calculates the evolution of a structural parameter λ (`lambda`) according to:

$$\frac{\partial \lambda}{\partial t} + \nabla \bullet (\mathbf{U}\lambda) = a(1-\lambda)^b - c\dot{\gamma}^d\lambda \quad (7.30)$$

with model coefficients a, b, c and d. The viscosity ν is then calculated according to:

$$\nu = \frac{\nu_\infty}{1 - K\lambda^2} \quad (7.31)$$

where the parameter $K = \sqrt{\nu_\infty/\nu_0}$. The viscosities ν_0 and ν_∞ are limiting values corresponding to $\lambda = 1$ and $\lambda = 0$.

An example specification of the model in *momentumTransport* is:

```
simulationType laminar;
```

```
laminar
{
    model               lambdaThixotropic;

    lambdaThixotropicCoeffs
    {
        a       1;
        b       2;
        c       1e-3;
        d       3;
        nu0     0.1;
        nuInf   1e-4;
    }
}
```

Index