OpenFOAM

The Open Source CFD Toolbox

Programmer's Guide

Version v2106
28th June 2021

Typeset in LaTeX.

License

1. Definitions

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

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

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

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

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

f. **"Work"** means the literary and/or artistic work offered under the terms of this License including without limitation any production in the literary, scientific and artistic domain, whatever may be the mode or form of its expression including digital form, such as a book, pamphlet and other writing; a lecture, address, sermon or other work of the same nature; a dramatic or dramatico-musical work; a choreographic work or entertainment in dumb show; a musical composition with or without words; a cinematographic work to which are assimilated works expressed by a process analogous to cinematography; a work of drawing, painting, architecture, sculpture, engraving or lithography; a photographic work to which are assimilated works expressed by a process analogous to photography; a work of applied art; an illustration, map, plan, sketch or three-dimensional work relative to geography, topography, architecture or science; a performance; a broadcast; a phonogram; a compilation of data to the extent it is protected as a copyrightable work; or a work performed by a variety or circus performer to the extent it is not otherwise considered a literary or artistic work.

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

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

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

2. Fair Dealing Rights

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

3. License Grant

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

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

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

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

4. Restrictions

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

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

Trademarks

Contents

Chapter 1

Introduction

1.1 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.

1.1.1 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.

1.1.2 Object-orientation and C++

Programming 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.

1.1.3 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.

1.1.4 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; 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 *$WM_PROJECT_DIR/doc/Doxygen/html/index.html*. This local documentation needs to be compiled by running the Allrun script in the Doxygen directory. An online version for the most recent release can be found at http://www.openfoam.com/documentation/cpp-guide/html/.

1.2 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, however, is supplied with the wmake compilation script that is based on make but is considerably more versatile and easier to use; wmake can, in fact, be used on any code, not simply 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 1.1. A class is defined through a set of instructions such as object construction, data storage and class member functions. The file containing 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 *nc.so* at runtime. This is known as *dynamic linking*.

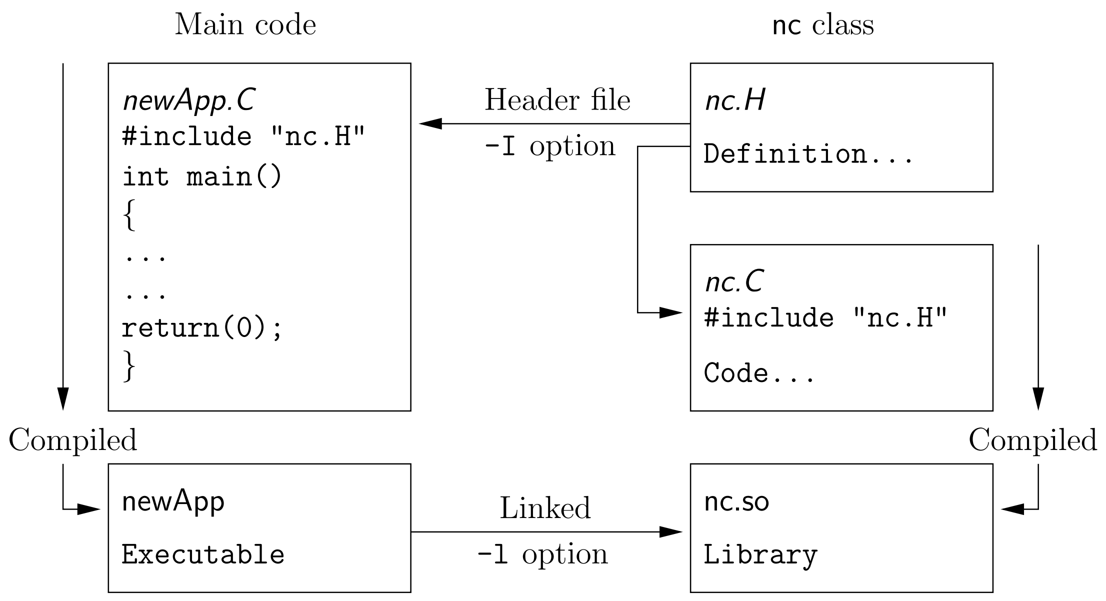


Figure 1.1: Header files, source files, compilation and linking

1.2.1 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, including the class declaration code itself. Any piece of *.C* code can resource any number of classes and must begin with all the *.H* files required to declare these classes. The classes in turn can resource other classes and begin with the relevant *.H* files. By searching recursively down the class hierarchy we can produce a complete list of header 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 `# include` statements, *e.g.*

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

causes the compiler to suspend reading from the current file to read the file specified. Any self-contained piece of code can 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`.
- Has been tested on Apollo, SUN, SGI, HP (HPUX), Compaq (DEC), IBM (AIX), Cray, Ardent, Stardent, PC Linux, PPC Linux, NEC, SX4, Fujitsu VP1000.

1.2.2 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 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 1.2. The directory must also contain a *Make* subdirectory containing 2 files, *options* and *files*, that are described in the following sections.

1.2.2.1 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*;

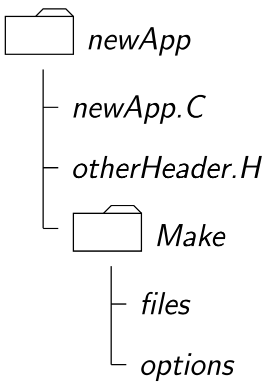


Figure 1.2: Directory structure for an application

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.

1.2.2.2 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<libraryPath1> \
    -L<libraryPath2> \
    ...              \
    -L<libraryPathN> \
    -l<library1>     \
    -l<library2>     \
    ...              \
    -l<libraryN>
```

Let us reiterate that the directory paths are preceeded by the `-L` flag, the library names are preceeded by the `-l` flag.

1.2.2.3 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. As might be expected, 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
```

1.2.2.4 Running wmake

The wmake script is executed by typing:

```
wmake <optionalArguments> <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.

If a user wishes to build an application executable or dynamic library, then no `<optionalArguments>` are required. However `<optionalArguments>` may be specified for building libraries *etc.* as described in Table 1.1.

Argument	Type of compilation
`all`	wmake all subdirectories, running Allwmake files if present
`exe`	Compile statically linked executable
`lib`	Compile statically linked archive lib (.a)
`libo`	Compile statically linked lib (.o)
`libso`	Compile dynamically linked lib (.so)
`dep`	Compile lnInclude and dependencies only

Table 1.1: Optional compilation arguments to wmake.

1.2.2.5 wmake environment variables

For information, the environment variable settings used by wmake are listed in Table 1.2.

1.2.3 Removing dependency lists: wclean and wrmdep

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

```
wclean <optionalArguments> <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.

If a user wishes to remove the dependency files and files from the *Make* directory, then no `<optionalArguments>` are required. However if `lib` is specified in `<optionalArguments>` a local *lnInclude* directory will be deleted also.

An additional script, wrmdep removes all dependency *.dep* files recursively down the directory tree from the point at which it is executed. This can be useful when updating OpenFOAM libraries. With the `-a`/`-all`/`all` options the *.dep* files are removed for all platforms rather than just the current platform. More usefull scripts for compilation housekeeping are located in the directory *$WM_PROJECT_DIR/wmake*.

1.2.4 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     |
    \\  /    A nd           | www.openfoam.com
     \\/     M anipulation  |
-------------------------------------------------------------------------------
    Copyright (C) 2011-2017 OpenFOAM Foundation
-------------------------------------------------------------------------------
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
```

Main paths	
$WM_PROJECT_INST_DIR	Full path to installation directory, *e.g.*$HOME/OpenFOAM
$WM_PROJECT	Name of the project being compiled: OpenFOAM
$WM_PROJECT_VERSION	Version of the project being compiled: v2106
$WM_PROJECT_DIR	Full path to locate binary executables of OpenFOAM release, *e.g.*$HOME/OpenFOAM/OpenFOAM-v2106
$WM_PROJECT_USER_DIR	Full path to locate binary executables of the user *e.g.*$HOME/OpenFOAM/${USER}-v2106
Other paths/settings	
$WM_ARCH	Machine architecture: Linux, SunOS
$WM_ARCH_OPTION	32 or 64 bit architecture
$WM_CC	Compiler command, *e.g.*gcc, clang
$WM_COMPILER	Compiler tag being used, *e.g.*Gcc - gcc 4.5.x, Clang, ICC - Intel
$WM_COMPILER_LIB_ARCH	Compiler 32 or 64 bit architecture
$WM_COMPILE_OPTION	Compilation option: Debug - debugging, Opt optimisation.
$WM_DIR	Full path of the *wmake* directory
$WM_MPLIB	Parallel communications library: OPENMPI, MPICH
$WM_OPTIONS	= WM_ARCHWM_COMPILER... ...$WM_COMPILE_OPTION$WM_MPLIB *e.g.*linux64ClangDPInt32Opt
$WM_PRECISION_OPTION	Precision of the compiled binaries, SP, single precision or DP, double precision

Table 1.2: Environment variable settings for wmake.

```
    (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

Group
    grpIncompressibleSolvers

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

    \heading Solver details
    The solver uses the PISO algorithm to solve the continuity equation:

        \f[
            \div \vec{U} = 0
        \f]

    and momentum equation:

        \f[
            \ddt{\vec{U}} + \div \left( \vec{U} \vec{U} \right) - \div \gvec{R}
          = - \grad p
        \f]

    Where:
    \vartable
        \vec{U} | Velocity
        p       | Pressure
        \vec{R} | Stress tensor
    \endvartable

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

    \heading Required fields
    \plaintable
        U       | Velocity [m/s]
        p       | Kinematic pressure, p/rho [m2/s2]
        \<turbulence fields\> | As required by user selection
    \endplaintable

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

#include "fvCFD.H"
#include "singlePhaseTransportModel.H"
#include "turbulentTransportModel.H"
#include "pisoControl.H"
#include "fvOptions.H"

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

int main(int argc, char *argv[])
{
    argList::addNote
    (
        "Transient solver for incompressible, turbulent flow,"
        " using the PISO algorithm."
    );

    #include "postProcess.H"

    #include "addCheckCaseOptions.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.timeName() << nl << endl;

        #include "CourantNo.H"

        // Pressure-velocity PISO corrector
        {
            #include "UEqn.H"

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

        laminarTransport.correct();
        turbulence->correct();

        runTime.write();

        runTime.printExecutionTime(Info);
    }

    Info<< "End\n" << endl;

    return 0;
}

```

```

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

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 incompressibleRASModels, incompressibleLESModels and incompressibleTransportModels 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:

```
EXE_INC = \
    -I$(LIB_SRC)/finiteVolume/lnInclude \
    -I$(LIB_SRC)/meshTools/lnInclude \
    -I$(LIB_SRC)/sampling/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/turbulenceModels/lnInclude \
    -I$(LIB_SRC)/TurbulenceModels/incompressible/lnInclude \
    -I$(LIB_SRC)/transportModels \
    -I$(LIB_SRC)/transportModels/incompressible/singlePhaseTransportModel

EXE_LIBS = \
    -lfiniteVolume \
    -lfvOptions \
    -lmeshTools \
    -lsampling \
    -lturbulenceModels \
    -lincompressibleTurbulenceModels \
    -lincompressibleTransportModels
```

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:

```
pisoFoam.C

EXE = $(FOAM_APPBIN)/pisoFoam
```

The user can compile pisoFoam by going to the *$FOAM_SOLVERS/incompressible/pisoFoam* directory and typing:

```
wmake
```

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

```
.../OpenFOAM/OpenFOAM-v2106/applications/solvers/incompressible/pisoFoam
Making dependency list for source file pisoFoam.C
clang++ -m64 -D<options> -W<options> -I<options> -c pisoFoam.C
-o .../platforms/linux64/applications/solvers/incompressible/pisoFoam/pisoFoam.o
clang++ -m64 -D<options> -W<options> -I<options>
.../platforms/linux64/applications/solvers/incompressible/pisoFoam/pisoFoam.o
-L<library-paths> -l<libraries>
-o .../platforms/linux64/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:

```
.../OpenFOAM/OpenFOAM-v2106/applications/solvers/incompressible/pisoFoam
make: '.../platforms/linux64/bin/pisoFoam' is up to date.
```

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

```
wclean
```

and running wmake.

1.2.5 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/v2106/controlDict* file. The list of possible switches is extensive and can be viewed by running the foamDebugSwitches application. Most of the switches correspond 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`. There are some switches that control messaging at a higher level than most, listed in Table 1.3.

In addition, there are some switches that control certain operational and optimisation issues. These switches are also listed in Table 1.3. 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.

High level debugging switches - sub-dictionary *DebugSwitches*	
`level`	Overall level of debugging messaging for OpenFOAM- - 3 levels `0`, `1`, `2`
`lduMatrix`	Messaging for solver convergence during a run - 3 levels `0`, `1`, `2`
Optimisation switches - sub-dictionary *OptimisationSwitches*	
`fileModific-ationSkew`	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.
`fileModific-ationChecking`	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 exist, `timeStampMaster`, `inotifyMaster`.
`commsType`	Parallel communications type: `nonBlocking`, `scheduled`, `blocking`.
`floatTransfer`	If `1`, will compact numbers to `float` precision before transfer; default is `0`
`nProcsSimpleSum`	Optimises global sum for parallel processing; sets number of processors above which hierarchical sum is performed rather than a linear sum (default `16`)

Table 1.3: Runtime message switches.

1.2.6 Linking new user-defined libraries to existing 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"
);
```

Chapter 2

Tensor mathematics

This Chapter describes how tensors and their algebraic operations are programmed in OpenFOAM, beginning with a short introduction to co-ordinate systems.

2.1 Coordinate system

OpenFOAM is primarily designed to solve problems in continuum mechanics, *i.e.* the branch of mechanics concerned with the stresses in solids, liquids and gases and the deformation or flow of these materials. OpenFOAM is therefore based in 3 dimensional space and time and deals with physical entities described by tensors. The coordinate system used by OpenFOAM is the right-handed rectangular Cartesian axes as shown in Figure 2.1. This system of axes is constructed by defining an origin O from which three lines are drawn at right angles to each other, termed the Ox, Oy, Oz axes. 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.

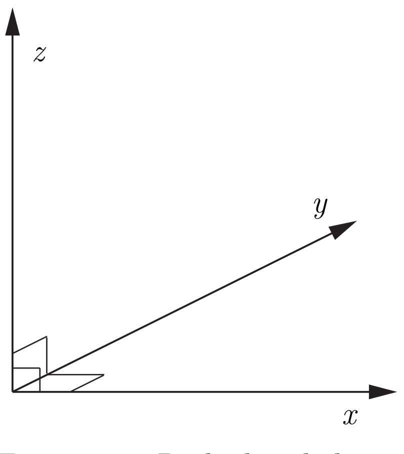


Figure 2.1: Right handed axes

2.2 OpenFOAM tensor classes

The term tensor describes an entity that belongs to a particular space and obeys certain mathematical rules. Briefly, tensors are represented by a set of *component values* relating to a set of unit base vectors; in OpenFOAM the unit base vectors $\mathbf{i}_x$, $\mathbf{i}_y$ and $\mathbf{i}_z$ are aligned with the right-handed rectangular Cartesian axes x, y and z respectively. The base vectors are therefore orthogonal, *i.e.* at right-angles to one another. See A for a summary of tensor notation and operation used in the following sections. Every tensor has the following attributes:

Dimension d of the particular space to which they belong, *i.e.* $d = 3$ in OpenFOAM;

Rank An integer $r \geq 0$, such that the number of component values $= d^r$.

While OpenFOAM is set to 3 dimensions, it offers tensors of ranks 0 to 3 as standard while being written in such a way to allow this basic set of ranks to be extended indefinitely. Tensors of rank 0 and 1, better known as scalars and vectors, should be familiar to readers; tensors of rank 2 and 3 may not be so familiar. For completeness all ranks of tensor offered as standard in OpenFOAM are reviewed below.

Rank 0 'scalar' Any property which can be represented by a single real number, denoted by characters in italics, *e.g.* mass m, volume V, pressure p and viscosity μ.

Rank 1 'vector' An entity which can be represented physically by both magnitude and direction. In component form, the vector $\mathbf{a} = (a_1, a_2, a_3)$ relates to a set of Cartesian axes x, y, z respectively. The *index notation* presents the same vector as a_i, $i = 1, 2, 3$, although the list of indices $i = 1, 2, 3$ will be omitted in this book, as it is intuitive since we are always dealing with 3 dimensions.

Rank 2 'tensor' or second rank tensor, $\mathbf{T}$ has 9 components which can be expressed in array notation as:

$$\mathbf{T} = T_{ij} = \begin{pmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ T_{31} & T_{32} & T_{33} \end{pmatrix} \tag{2.1}$$

The components T_{ij} are now represented using 2 indices since $r = 2$ and the list of indices $i, j = 1, 2, 3$ is omitted as before. The components for which $i = j$ are referred to as the diagonal components, and those for which $i \neq j$ are referred to as the off-diagonal components. The *transpose* of $\mathbf{T}$ is produced by exchanging components across the diagonal such that

$$\mathbf{T}^{\mathrm{T}} = T_{ji} = \begin{pmatrix} T_{11} & T_{21} & T_{31} \\ T_{12} & T_{22} & T_{32} \\ T_{13} & T_{23} & T_{33} \end{pmatrix} \tag{2.2}$$

Note: a rank 2 tensor is often colloquially termed 'tensor' since the occurrence of higher order tensors is fairly rare.

Symmetric rank 2 The term 'symmetric' refers to components being symmetric about the diagonal, *i.e.* $T_{ij} = T_{ji}$. In this case, there are only 6 independent components since $T_{12} = T_{21}$, $T_{13} = T_{31}$ and $T_{23} = T_{32}$. OpenFOAM distinguishes between symmetric and non-symmetric tensors to save memory by storing 6 components rather than 9 if the tensor is symmetric. Most tensors encountered in continuum mechanics are symmetric.

Rank 3 has 27 components and is represented in index notation as P_{ijk} which is too long to represent in array notation as in Equation 2.1.

Symmetric rank 3 Symmetry of a rank 3 tensor is defined in OpenFOAM to mean that $P_{ijk} = P_{ikj} = P_{jik} = P_{jki} = P_{kij} = P_{kji}$ and therefore has 10 independent components. More specifically, it is formed by the outer product of 3 identical vectors, where the outer product operation is described in Section A.2.4.

OpenFOAM contains a C++ class library primitive that contains the classes for the tensor mathematics described so far. The basic tensor classes that are available as standard in OpenFOAM are listed in Table 2.1. The Table also lists the functions that allow the user to access individual components of a tensor, known as access functions.

Rank	Common name	Basic class	Access functions
0	Scalar	scalar	
1	Vector	vector	`x()`, `y()`, `z()`
2	Tensor	tensor	`xx()`, `xy()`, `xz()`...

Table 2.1: Basic tensor classes in OpenFOAM

We can declare the tensor

$$\mathbf{T} = \begin{pmatrix} 1 & 2 & 3 \\ 4 & 5 & 6 \\ 7 & 8 & 9 \end{pmatrix} \tag{2.3}$$

in OpenFOAM by the line:

```
tensor T(1, 2, 3, 4, 5, 6, 7, 8, 9);
```

We can then access the component T_{13}, or T_{xz} using the `xz()` access function. For instance the code

```
Info << ``Txz = '' << T.xz() << endl;
```

outputs to the screen:

```
Txz = 3
```

2.2.1 Algebraic tensor operations in OpenFOAM

The algebraic operations described in Section A.2 are all available to the OpenFOAM tensor classes using syntax which closely mimics the notation used in written mathematics. Some functions are represented solely by descriptive functions, *e.g.*`symm()`, but others can also be executed using symbolic operators, *e.g.*`*`. All functions are listed in Table 2.2.

Operation	Comment	Mathematical Description	Description in OpenFOAM
Addition		$\mathbf{a} + \mathbf{b}$	`a + b`
Subtraction		$\mathbf{a} - \mathbf{b}$	`a - b`
Scalar multiplication		$s\mathbf{a}$	`s * a`
Scalar division		$\mathbf{a}/s$	`a / s`
Outer product	rank $\mathbf{a}, \mathbf{b} >= 1$	$\mathbf{ab}$	`a * b`
Inner product	rank $\mathbf{a}, \mathbf{b} >= 1$	$\mathbf{a} \bullet \mathbf{b}$	`a & b`

Continued on next page

Continued from previous page

Operation	Comment	Mathematical Description	Description in OpenFOAM
Double inner product	rank $\mathbf{a}, \mathbf{b} >= 2$	$\mathbf{a} \mathbin{\mathbf{:}} \mathbf{b}$	`a && b`
Cross product	rank $\mathbf{a}, \mathbf{b} = 1$	$\mathbf{a} \times \mathbf{b}$	`a ^ b`
Square		$\mathbf{a}^2$	`sqr(a)`
Magnitude squared		$\lvert\mathbf{a}\rvert^2$	`magSqr(a)`
Magnitude		$\lvert\mathbf{a}\rvert$	`mag(a)`
Power	$n = 0, 1, ..., 4$	$\mathbf{a}^n$	`pow(a,n)`
Component average	$i = 1, ..., N$	$\overline{a_i}$	`cmptAv(a)`
Component maximum	$i = 1, ..., N$	$\max(a_i)$	`cmptMax(a)`
Component minimum	$i = 1, ..., N$	$\min(a_i)$	`cmptMin(a)`
Scale		$\mathrm{scale}(\mathbf{a},\mathbf{b})$	`cmptMultiply(a,b)`
Geometric transformation	transforms $\mathbf{a}$ using tensor $\mathbf{T}$		`transform(T,a)`
Operations exclusive to tensors of rank 2			
Transpose		$\mathbf{T}^{\mathrm{T}}$	`T.T()`
Diagonal		$\mathrm{diag}\,\mathbf{T}$	`diag(T)`
Trace		$\mathrm{tr}\,\mathbf{T}$	`tr(T)`
Deviatoric component		$\mathrm{dev}\,\mathbf{T}$	`dev(T)`
Symmetric component		$\mathrm{symm}\,\mathbf{T}$	`symm(T)`
Skew-symmetric component		$\mathrm{skew}\,\mathbf{T}$	`skew(T)`
Determinant		$\det\mathbf{T}$	`det(T)`
Cofactors		$\mathrm{cof}\,\mathbf{T}$	`cof(T)`
Inverse		$\mathrm{inv}\,\mathbf{T}$	`inv(T)`
Hodge dual		$*\,\mathbf{T}$	`*T`
Operations exclusive to scalars			
Sign (boolean)		$\mathrm{sgn}(s)$	`sign(s)`
Positive (boolean)		$s >= 0$	`pos(s)`
Negative (boolean)		$s < 0$	`neg(s)`
Limit	n scalar	$\mathrm{limit}(s, n)$	`limit(s,n)`
Square root		$\sqrt{s}$	`sqrt(s)`
Exponential		$\exp s$	`exp(s)`
Natural logarithm		$\ln s$	`log(s)`
Base 10 logarithm		$\log_{10} s$	`log10(s)`
Sine		$\sin s$	`sin(s)`
Cosine		$\cos s$	`cos(s)`
Tangent		$\tan s$	`tan(s)`
Arc sine		$\mathrm{asin}\, s$	`asin(s)`
Arc cosine		$\mathrm{acos}\, s$	`acos(s)`
Arc tangent		$\mathrm{atan}\, s$	`atan(s)`
Hyperbolic sine		$\sinh s$	`sinh(s)`
Hyperbolic cosine		$\cosh s$	`cosh(s)`
Hyperbolic tangent		$\tanh s$	`tanh(s)`
Hyperbolic arc sine		$\mathrm{asinh}\, s$	`asinh(s)`
Hyperbolic arc cosine		$\mathrm{acosh}\, s$	`acosh(s)`
Hyperbolic arc tangent		$\mathrm{atanh}\, s$	`atanh(s)`
Error function		$\mathrm{erf}\, s$	`erf(s)`
Complement error function		$\mathrm{erfc}\, s$	`erfc(s)`

Continued on next page

Continued from previous page

Operation	Comment	Mathematical Description	Description in OpenFOAM
Logarithm gamma function		$\ln \Gamma s$	`lgamma(s)`
Type 1 Bessel function of order 0		$\mathrm{J}_0\, s$	`j0(s)`
Type 1 Bessel function of order 1		$\mathrm{J}_1\, s$	`j1(s)`
Type 2 Bessel function of order 0		$\mathrm{Y}_0\, s$	`y0(s)`
Type 2 Bessel function of order 1		$\mathrm{Y}_1\, s$	`y1(s)`

$\mathbf{a}, \mathbf{b}$ are tensors of arbitrary rank unless otherwise stated
s is a scalar, N is the number of tensor components

Table 2.2: Algebraic tensor operations in OpenFOAM

2.3 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 ($\mathrm{kg\,m\,s^{-2}}$). 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 encourages the user to attach dimensional units to any tensor and will then perform dimension checking of any tensor operation.

Units are defined using the dimensionSet class, *e.g.*

```
dimensionSet pressureDims(1, -1, -2, 0, 0, 0, 0);
```

No.	Property	Unit	Symbol
1	Mass	kilogram	k
2	Length	metre	m
3	Time	second	s
4	Temperature	Kelvin	K
5	Quantity	moles	mol
6	Current	ampere	A
7	Luminous intensity	candela	cd

Table 2.3: S.I. base units of measurement

where each of the values corresponds to the power of each of the S.I. base units of measurement listed in Table 2.3. The line of code declares `pressureDims` to be the dimensionSet for pressure $\mathrm{kg\,m\,s^{-2}}$ since the first entry in the `pressureDims` array, 1, corresponds to k^1, the second entry, -1, corresponds to m^{-1} *etc.*. A tensor with units is defined using the dimensioned<Type> template class, the <Type> being scalar, vector, tensor, *etc.*. The dimensioned<Type> stores a variable name of class word,the value <Type> and a dimensionSet

```
dimensionedTensor sigma
    (
        "sigma",
        dimensionSet(1, -1, -2, 0, 0, 0, 0),
        tensor(1e6,0,0,0,1e6,0,0,0,1e6),
    );
```

creates a tensor with correct dimensions of pressure, or stress

$$\boldsymbol{\sigma} = \begin{pmatrix} 10^6 & 0 & 0 \\ 0 & 10^6 & 0 \\ 0 & 0 & 10^6 \end{pmatrix} \tag{2.4}$$

Chapter 3

Discretisation procedures

So far we have dealt with algebra of tensors at a point. The PDEs we wish to solve involve derivatives of tensors with respect to time and space. We therefore need to extend our description to a *tensor field*, *i.e.* a tensor that varies across time and spatial domains. In this Chapter we will first present a mathematical description of all the differential operators we may encounter. We will then show how a tensor field is constructed in OpenFOAM and how the derivatives of these fields are discretised into a set of algebraic equations.

3.1 Differential operators

Before defining the spatial derivatives we first introduce the nabla *vector operator* ∇, represented in index notation as ∂_i:

$$\nabla \equiv \partial_i \equiv \frac{\partial}{\partial x_i} \equiv \left(\frac{\partial}{\partial x_1}, \frac{\partial}{\partial x_2}, \frac{\partial}{\partial x_3} \right) \tag{3.1}$$

The nabla operator is a useful notation that obeys the following rules:

- it operates on the tensors to its right and the conventional rules of a derivative of a product, *e.g.* $\partial_i ab = (\partial_i a)\, b + a\, (\partial_i b)$;
- otherwise the nabla operator behaves like any other vector in an algebraic operation.

3.1.1 Gradient

If a scalar field s is defined and continuously differentiable then the gradient of s, ∇s is a vector field

$$\nabla s = \partial_i s = \left(\frac{\partial s}{\partial x_1}, \frac{\partial s}{\partial x_2}, \frac{\partial s}{\partial x_3} \right) \tag{3.2}$$

The gradient can operate on any tensor field to produce a tensor field that is one rank higher. For example, the gradient of a vector field $\mathbf{a}$ is a second rank tensor field

$$\nabla \mathbf{a} = \partial_i a_j = \begin{pmatrix} \partial a_1/\partial x_1 & \partial a_2/\partial x_1 & \partial a_3/\partial x_1 \\ \partial a_1/\partial x_2 & \partial a_2/\partial x_2 & \partial a_3/\partial x_2 \\ \partial a_1/\partial x_3 & \partial a_2/\partial x_3 & \partial a_3/\partial x_3 \end{pmatrix} \tag{3.3}$$

3.1.2 Divergence

If a vector field **a** is defined and continuously differentiable then the divergence of **a** is a scalar field

$$\nabla \bullet \mathbf{a} = \partial_i a_i = \frac{\partial a_1}{\partial x_1} + \frac{\partial a_2}{\partial x_2} + \frac{\partial a_3}{\partial x_3} \tag{3.4}$$

The divergence can operate on any tensor field of rank 1 and above to produce a tensor that is one rank lower. For example the divergence of a second rank tensor field **T** is a vector field (expanding the vector as a column array for convenience)

$$\nabla \bullet \mathbf{T} = \partial_i T_{ij} = \begin{pmatrix} \partial T_{11}/\partial x_1 + \partial T_{21}/\partial x_1 + \partial T_{31}/\partial x_1 \\ \partial T_{12}/\partial x_2 + \partial T_{22}/\partial x_2 + \partial T_{32}/\partial x_2 \\ \partial T_{13}/\partial x_3 + \partial T_{23}/\partial x_3 + \partial T_{33}/\partial x_3 \end{pmatrix} \tag{3.5}$$

3.1.3 Curl

If a vector field **a** is defined and continuously differentiable then the curl of **a**, $\nabla \times \mathbf{a}$ is a vector field

$$\nabla \times \mathbf{a} = e_{ijk}\partial_j a_k = \left(\frac{\partial a_3}{\partial x_2} - \frac{\partial a_2}{\partial x_3}, \frac{\partial a_1}{\partial x_3} - \frac{\partial a_3}{\partial x_1}, \frac{\partial a_2}{\partial x_1} - \frac{\partial a_1}{\partial x_2} \right) \tag{3.6}$$

The curl is related to the gradient by

$$\nabla \times \mathbf{a} = 2\,(* \operatorname{skew} \nabla \mathbf{a}) \tag{3.7}$$

3.1.4 Laplacian

The Laplacian is an operation that can be defined mathematically by a combination of the divergence and gradient operators by $\nabla^2 \equiv \nabla \bullet \nabla$. However, the Laplacian should be considered as a single operation that transforms a tensor field into another tensor field of the same rank, rather than a combination of two operations, one which raises the rank by 1 and one which reduces the rank by 1.

In fact, the Laplacian is best defined as a *scalar operator*, just as we defined nabla as a vector operator, by

$$\nabla^2 \equiv \partial^2 \equiv \frac{\partial^2}{\partial x_1^2} + \frac{\partial^2}{\partial x_2^2} + \frac{\partial^2}{\partial x_3^2} \tag{3.8}$$

For example, the Laplacian of a scalar field s is the scalar field

$$\nabla^2 s = \partial^2 s = \frac{\partial^2 s}{\partial x_1^2} + \frac{\partial^2 s}{\partial x_2^2} + \frac{\partial^2 s}{\partial x_3^2} \tag{3.9}$$

3.1.5 Temporal derivative

There is more than one definition of temporal, or time, derivative of a tensor. To describe the temporal derivatives we must first recall that the tensor relates to a property of a volume of material that may be moving. If we track an infinitesimally small volume of material, or particle, as it moves and observe the change in the tensorial property ϕ in time, we have the *total*, or *material* time derivative denoted by

$$\frac{D\phi}{Dt} = \lim_{\Delta t \to 0} \frac{\Delta \phi}{\Delta t} \tag{3.10}$$

However in continuum mechanics, particularly fluid mechanics, we often observe the change of a ϕ in time at a fixed point in space as different particles move across that point. This change at a point in space is termed the *spatial* time derivative which is denoted by $\partial/\partial t$ and is related to the material derivative by:

$$\frac{D\phi}{Dt} = \frac{\partial\phi}{\partial t} + \mathbf{U} \bullet \nabla\phi \tag{3.11}$$

where $\mathbf{U}$ is the velocity field of property ϕ. The second term on the right is known as the convective rate of change of ϕ.

3.2 Overview of discretisation

The term discretisation means *approximation of a problem into discrete quantities*. The FV method and others, such as the finite element and finite difference methods, all discretise the problem as follows:

Spatial discretisation Defining the solution domain by a set of points that fill and bound a region of space when connected;

Temporal discretisation (For transient problems) dividing the time domain into into a finite number of time intervals, or steps;

Equation discretisation Generating a system of algebraic equations in terms of discrete quantities defined at specific locations in the domain, from the PDEs that characterise the problem.

3.2.1 OpenFOAM lists and fields

OpenFOAM frequently needs to store sets of data and perform functions, such as mathematical operations, on the data. OpenFOAM therefore provides an array template class List<Type>, making it possible to create a list of any object of class Type that inherits the functions of the Type. For example a List of vector is List<vector>.

Lists of the tensor classes are defined as standard in OpenFOAM by the template class Field<Type>. For better code legibility, all instances of Field<Type>, *e.g.*Field<vector>, are renamed using `typedef` declarations as scalarField, vectorField, tensorField, symmTensorField, tensorThirdField and symmTensorThirdField. Algebraic operations can be performed between Fields subject to obvious restrictions such as the fields having the same number of elements. OpenFOAM also supports operations between a field and single tensor, *e.g.* all values of a Field `U` can be multiplied by the scalar 2 with the operation `U = 2.0 * U`.

3.3 Discretisation of the solution domain

Discretisation of the solution domain is shown in Figure 3.1. The space domain is discretised into computational mesh on which the PDEs are subsequently discretised. Discretisation of time, if required, is simple: it is broken into a set of time steps Δt that may change during a numerical simulation, perhaps depending on some condition calculated during the simulation.

On a more detailed level, discretisation of space requires the subdivision of the domain into a number of cells, or control volumes. The cells are contiguous, *i.e.* they do not overlap one another and completely fill the domain. A typical cell is shown in Figure 3.2.

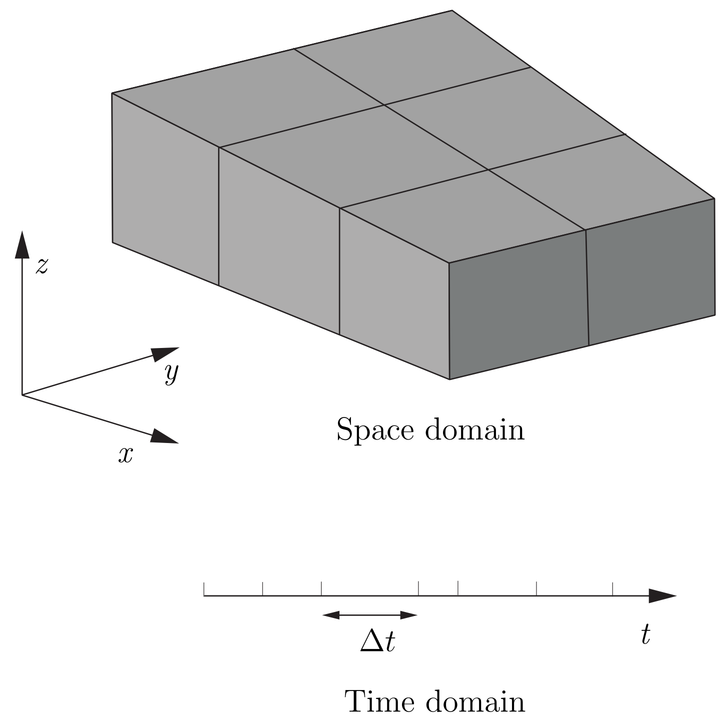


Figure 3.1: Discretisation of the solution domain

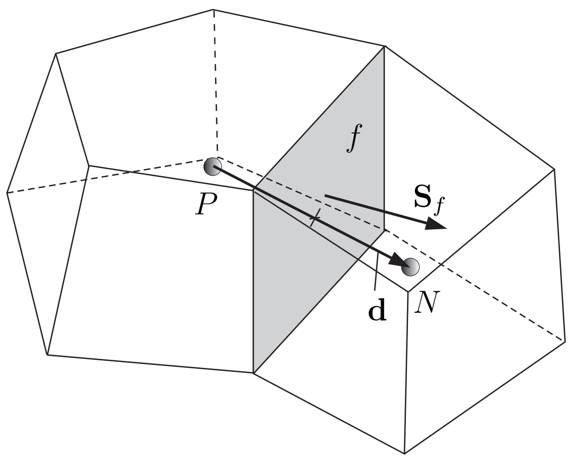


Figure 3.2: Parameters in finite volume discretisation

Dependent variables and other properties are principally stored at the cell centroid P although they may be stored on faces or vertices. The cell is bounded by a set of flat faces, given the generic label f. In OpenFOAM there is no limitation on the number of faces bounding each cell, nor any restriction on the alignment of each face. This kind of mesh is often referred to as "arbitrarily unstructured" to differentiate it from meshes in which the cell faces have a prescribed alignment, typically with the coordinate axes. Codes with arbitrarily unstructured meshes offer greater freedom in mesh generation and manipulation in particular when the geometry of the domain is complex or changes over time.

Whilst most properties are defined at the cell centroids, some are defined at cell faces. There are two types of cell face.

Internal faces Those faces that connect two cells (and it can never be more than two). For each internal face, OpenFOAM designates one adjoining cell to be the face *owner* and the other to be the *neighbour*;

Boundary faces Those belonging to one cell since they coincide with the boundary of the domain. These faces simply have an owner cell.

3.3.1 Defining a mesh in OpenFOAM

There are different levels of mesh description in OpenFOAM, beginning with the most basic mesh class, named polyMesh since it is based on polyhedra. A polyMesh is constructed using the minimum information required to define the mesh geometry described below and presented in Figure 3.3:

Points A list of cell vertex point coordinate vectors, *i.e.* a vectorField, that is renamed pointField using a `typedef` declaration;

Faces A list of cell faces List<face>, or faceList, where the face class is defined by a list of vertex numbers, corresponding to the pointField;

Cells a list of cells List<cell>, or cellList, where the cell class is defined by a list of face numbers, corresponding to the faceList described previously.

Boundary a polyBoundaryMesh decomposed into a list of patches, polyPatchList representing different regions of the boundary. The boundary is subdivided in this manner to allow different boundary conditions to be specified on different patches during a solution. All the faces of any polyPatch are stored as a single block of the faceList, so that its faces can be easily accessed using the slice class which stores references to the first and last face of the block. Each polyPatch is then constructed from

- a slice;
- a word to assign it a name.

FV discretisation uses specific data that is derived from the mesh geometry stored in polyMesh. OpenFOAM therefore extends the polyMesh class to fvMesh which stores the additional data needed for FV discretisation. fvMesh is constructed from polyMesh and stores the data in Table 3.1 which can be updated during runtime in cases where the mesh moves, is refined *etc.*.

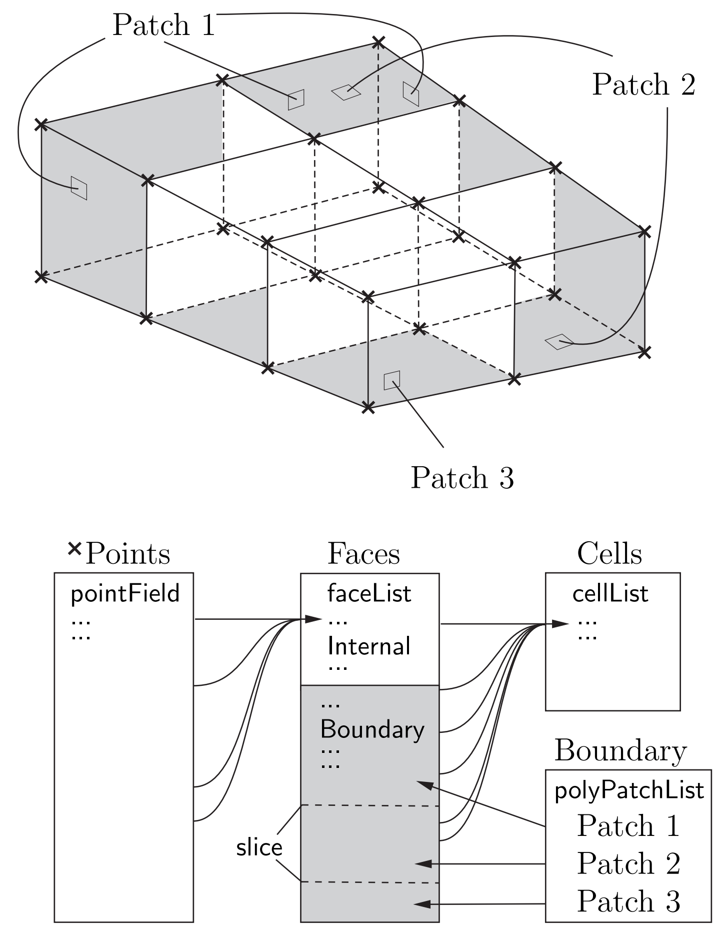


Figure 3.3: Schematic of the basic mesh description used in OpenFOAM

3.3.2 Defining a geometricField in OpenFOAM

So far we can define a field, *i.e.* a list of tensors, and a mesh. These can be combined to define a tensor field relating to discrete points in our domain, specified in OpenFOAM by the template class geometricField<Type>. The Field values are separated into those defined within the internal region of the domain, *e.g.* at the cell centres, and those defined on the domain boundary, *e.g.* on the boundary faces. The geometricField<Type> stores the following information:

Internal field This is simply a Field<Type>, described in Section 3.2.1;

Boundary field This is a Boundary, in which a Field is defined for the faces of each patch and a Field is defined for the patches of the boundary. This is then a field of fields, stored within an object of the FieldField<Type> class. A reference to the fvBoundaryMesh is also stored [**].

Mesh A reference to an fvMesh, with some additional detail as to the whether the field is defined at cell centres, faces, *etc.*.

Dimensions A dimensionSet, described in Section **??**.

Old values Discretisation of time derivatives requires field data from previous time steps. The geometricField<Type> will store references to stored fields from the previous, or old, time step and its previous, or old-old, time step where necessary.

Class	Description	Symbol	Access function
volScalarField	Cell volumes	V	`V()`
surfaceVectorField	Face area vectors	$\mathbf{S}_f$	`Sf()`
surfaceScalarField	Face area magnitudes	$\lvert\mathbf{S}_f\rvert$	`magSf()`
volVectorField	Cell centres	$\mathbf{C}$	`C()`
surfaceVectorField	Face centres	$\mathbf{C}_f$	`Cf()`
surfaceScalarField	Face motion fluxes **	ϕ_g	`phi()`

Table 3.1: fvMesh stored data.

Previous iteration values The iterative solution procedures can use under-relaxation which requires access to data from the previous iteration. Again, if required, geometricField<Type> stores a reference to the data from the previous iteration.

As discussed in Section 3.3, we principally define a property at the cell centres but quite often it is stored at the cell faces and on occasion it is defined on cell vertices. The GeometricField<Type> is renamed using `typedef` declarations to indicate where the field variable is defined as follows:

volField<Type> A field defined at cell centres;

surfaceField<Type> A field defined on cell faces;

pointField<Type> A field defined on cell vertices.

These `typedef` field classes of geometricField<Type>are illustrated in Figure 3.4. A geometricField<Type> inherits all the tensor algebra of Field<Type> and has all operations subjected to dimension checking using the dimensionSet. It can also be subjected to the FV discretisation procedures described in the following Section. The class structure used to build geometricField<Type> is shown in Figure 3.5[1].

3.4 Equation discretisation

Equation discretisation converts the PDEs into a set of algebraic equations that are commonly expressed in matrix form as:

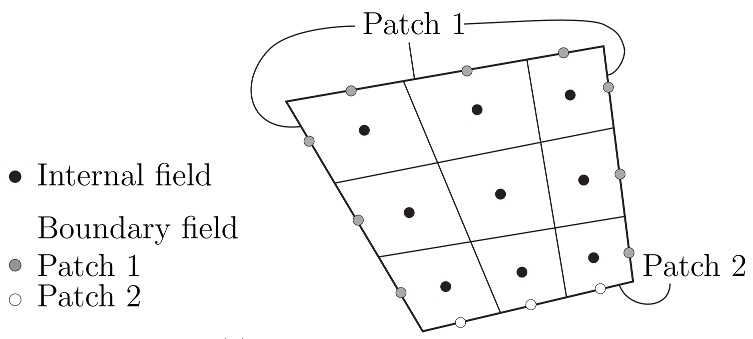
$$[A]\,[x] = [b] \tag{3.12}$$

where $[A]$ is a square matrix, $[x]$ is the column vector of dependent variable and $[b]$ is the source vector. The description of $[x]$ and $[b]$ as 'vectors' comes from matrix terminology rather than being a precise description of what they truly are: a list of values defined at locations in the geometry, *i.e.* a geometricField<Type>, or more specifically a volField<Type> when using FV discretisation.

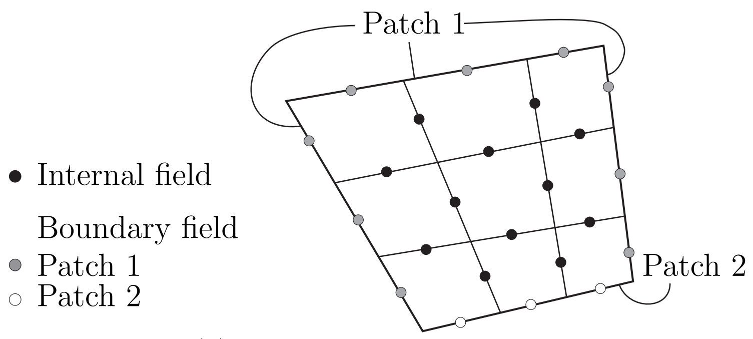
$[A]$ is a list of coefficients of a set of algebraic equations, and cannot be described as a geometricField<Type>. It is therefore given a class of its own: fvMatrix. fvMatrix<Type> is created through discretisation of a geometric<Type>Field and therefore inherits the <Type>. It supports many of the standard algebraic matrix operations of addition `+`, subtraction `-` and multiplication `*`.

Each term in a PDE is represented individually in OpenFOAM code using the classes of static functions finiteVolumeMethod and finiteVolumeCalculus, abbreviated by a `typedef`

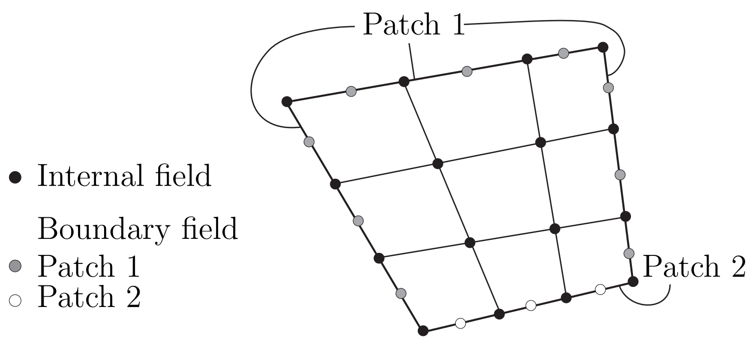
[1]The diagram is not an exact description of the class hierarchy, rather a representation of the general structure leading from some primitive classes to geometric<Type>Field.



(a) A volField<Type>



(b) A surfaceField<Type>



(c) A pointField<Type>

Figure 3.4: Types of geometricField<Type> defined on a mesh with 2 boundary patches (in 2 dimensions for simplicity)

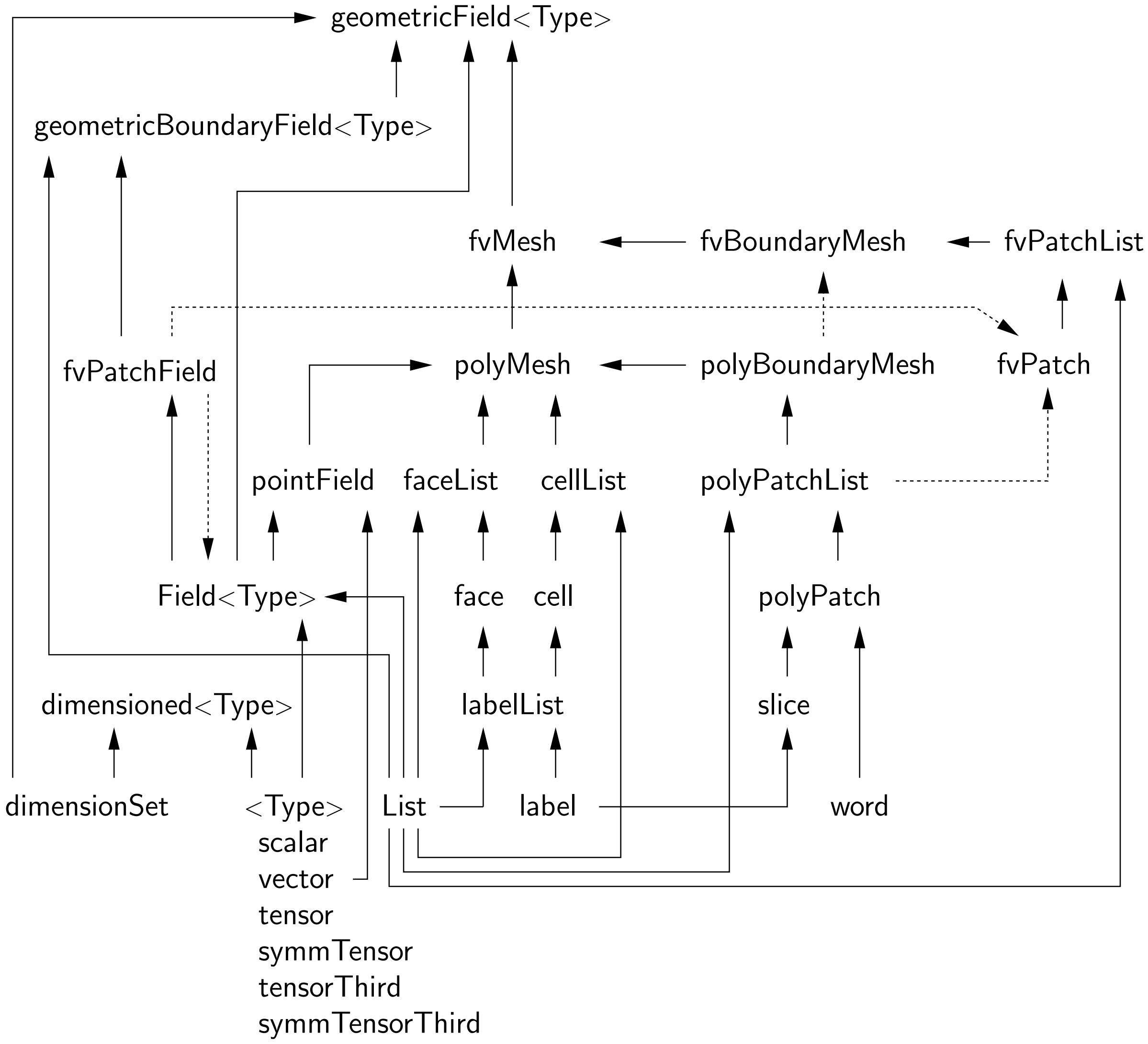


Figure 3.5: Basic class structure leading to geometricField<Type>

to fvm and fvc respectively. fvm and fvc contain static functions, representing differential operators, *e.g.* ∇^2, $\nabla\bullet$ and $\partial/\partial t$, that discretise geometricField<Type>s. The purpose of defining these functions within two classes, fvm and fvc, rather than one, is to distinguish:

- functions of fvm that calculate implicit derivatives of and return an fvMatrix<Type>
- some functions of fvc that calculate explicit derivatives and other explicit calculations, returning a geometricField<Type>.

Figure 3.6 shows a geometricField<Type> defined on a mesh with 2 boundary patches and illustrates the explicit operations merely transform one field to another and drawn in 2D for simplicity.

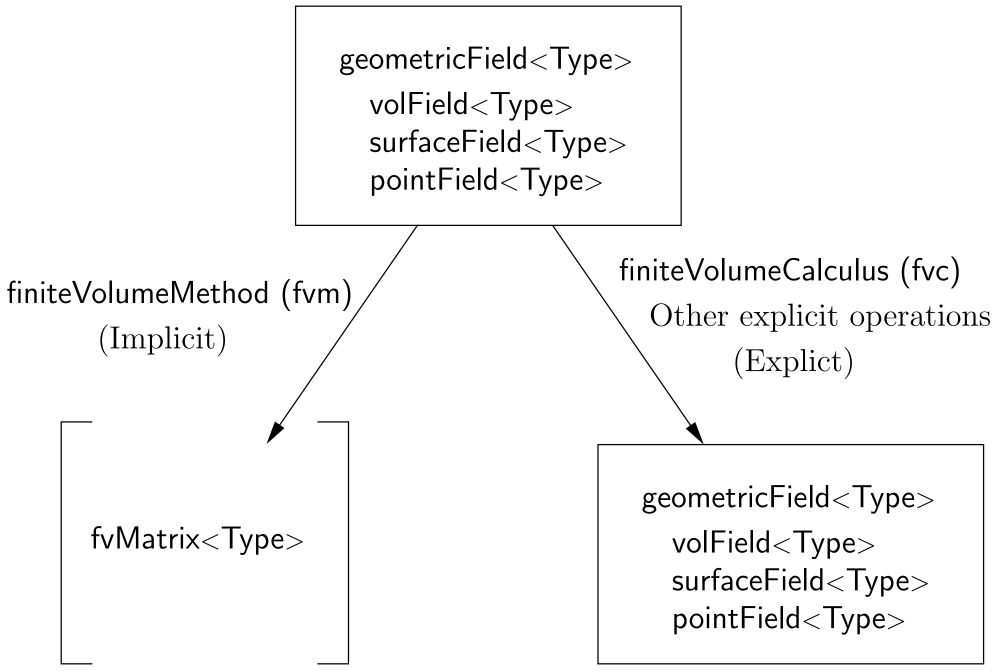


Figure 3.6: A geometricField<Type> and its operators

Table 3.2 lists the main functions that are available in fvm and fvc to discretise terms that may be found in a PDE. FV discretisation of each term is formulated by first integrating the term over a cell volume V. Most spatial derivative terms are then converted to integrals over the cell surface S bounding the volume using Gauss's theorem

$$\int_V \nabla \star \phi \; dV = \int_S d\mathbf{S} \star \phi \tag{3.13}$$

where $\mathbf{S}$ is the surface area vector, ϕ can represent any tensor field and the star notation $\star$ is used to represent any tensor product, *i.e.* inner, outer and cross and the respective derivatives: divergence $\nabla\bullet\phi$, gradient $\nabla\phi$ and $\nabla\times\phi$. Volume and surface integrals are then linearised using appropriate schemes which are described for each term in the following Sections. Some terms are always discretised using one scheme, a selection of schemes is offered in OpenFOAM for the discretisation of other terms. The choice of scheme is either made by a direct specification within the code or it can be read from an input file at job run-time and stored within an fvSchemes class object.

Term description	Implicit / Explicit	Text expression	fvm::/fvc:: functions
Laplacian	Imp/Exp	$\nabla^2\phi$	`laplacian(phi)`
		$\nabla \bullet \Gamma\nabla\phi$	`laplacian(Gamma, phi)`
Time derivative	Imp/Exp	$\frac{\partial\phi}{\partial t}$	`ddt(phi)`
		$\frac{\partial\rho\phi}{\partial t}$	`ddt(rho,phi)`
Second time derivative	Imp/Exp	$\frac{\partial}{\partial t}\left(\rho\frac{\partial\phi}{\partial t}\right)$	`d2dt2(rho, phi)`
Convection	Imp/Exp	$\nabla \bullet (\psi)$	`div(psi,scheme)`*
		$\nabla \bullet (\psi\phi)$	`div(psi, phi, word)`*
			`div(psi, phi)`
Divergence	Exp	$\nabla \bullet \chi$	`div(chi)`
Gradient	Exp	$\nabla\chi$	`grad(chi)`
		$\nabla\phi$	`gGrad(phi)`
			`lsGrad(phi)`
			`snGrad(phi)`
			`snGradCorrection(phi)`
Grad-grad squared	Exp	$\lvert\nabla\nabla\phi\rvert^2$	`sqrGradGrad(phi)`
Curl	Exp	$\nabla \times \phi$	`curl(phi)`
Source	Imp	$\rho\phi$	`Sp(rho,phi)`
	Imp/Exp†		`SuSp(rho,phi)`

†`fvm::SuSp` source is discretised implicit or explicit depending on the sign of `rho`.
†An explicit source can be introduced simply as a vol<Type>Field, *e.g.*`rho*phi`.
Function arguments can be of the following classes:
`phi`: vol<Type>Field
`Gamma`: scalar volScalarField, surfaceScalarField, volTensorField, surfaceTensorField.
`rho`: scalar, volScalarField
`psi`: surfaceScalarField.
`chi`: surface<Type>Field, vol<Type>Field.

Table 3.2: Discretisation of PDE terms in OpenFOAM

3.4.1 The Laplacian term

The Laplacian term is integrated over a control volume and linearised as follows:

$$\int_V \nabla \bullet (\Gamma \nabla \phi)\, dV = \int_S d\mathbf{S} \bullet (\Gamma \nabla \phi) = \sum_f \Gamma_f \mathbf{S}_f \bullet (\nabla \phi)_f \tag{3.14}$$

The face gradient discretisation is implicit when the length vector **d** between the centre of the cell of interest P and the centre of a neighbouring cell N is orthogonal to the face plane, *i.e.* parallel to $\mathbf{S}_f$:

$$\mathbf{S}_f \bullet (\nabla \phi)_f = |S_f| \frac{\phi_N - \phi_P}{|\mathbf{d}|} \tag{3.15}$$

In the case of non-orthogonal meshes, an additional explicit term is introduced which is evaluated by interpolating cell centre gradients, themselves calculated by central differencing cell centre values.

3.4.2 The convection term

The convection term is integrated over a control volume and linearised as follows:

$$\int_V \nabla \bullet (\rho \mathbf{U} \phi)\, dV = \int_S d\mathbf{S} \bullet (\rho \mathbf{U} \phi) = \sum_f \mathbf{S}_f \bullet (\rho \mathbf{U})_f \phi_f = \sum_f F \phi_f \tag{3.16}$$

The face field ϕ_f can be evaluated using a variety of schemes:

Central differencing (CD) is second-order accurate but unbounded

$$\phi_f = f_x \phi_P + (1 - f_x)\phi_N \tag{3.17}$$

where $f_x \equiv \overline{fN}/\overline{PN}$ where $\overline{fN}$ is the distance between f and cell centre N and $\overline{PN}$ is the distance between cell centres P and N.

Upwind differencing (UD) determines ϕ_f from the direction of flow and is bounded at the expense of accuracy

$$\phi_f = \begin{cases} \phi_P & \text{for } F \geq 0 \\ \phi_N & \text{for } F < 0 \end{cases} \tag{3.18}$$

Blended differencing (BD) schemes combine UD and CD in an attempt to preserve boundedness with reasonable accuracy,

$$\phi_f = (1 - \gamma)(\phi_f)_{UD} + \gamma (\phi_f)_{CD} \tag{3.19}$$

OpenFOAM has several implementations of the Gamma differencing scheme to select the blending coefficient γ but it offers other well-known schemes such as van Leer, SUPERBEE, MINMOD *etc.*.

3.4.3 First time derivative

The first time derivative $\partial/\partial t$ is integrated over a control volume as follows:

$$\frac{\partial}{\partial t}\int_V \rho\phi \, dV \tag{3.20}$$

The term is discretised by simple differencing in time using:

new values $\phi^n \equiv \phi(t + \Delta t)$ at the time step we are solving for;

old values $\phi^o \equiv \phi(t)$ that were stored from the previous time step;

old-old values $\phi^{oo} \equiv \phi(t - \Delta t)$ stored from a time step previous to the last.

One of three discretisation schemes can be declared using the `ddtSchemes` keyword in the appropriate input file, described in detail in section ?? of the User Guide.

Euler implicit scheme, `Euler`, that is first order accurate in time:

$$\frac{\partial}{\partial t}\int_V \rho\phi \, dV = \frac{(\rho_P\phi_P V)^n - (\rho_P\phi_P V)^o}{\Delta t} \tag{3.21}$$

Backward differencing scheme, `backward`, that is second order accurate in time by storing the old-old values and therefore with a larger overhead in data storage than `Euler`:

$$\frac{\partial}{\partial t}\int_V \rho\phi \, dV = \frac{3\,(\rho_P\phi_P V)^n - 4\,(\rho_P\phi_P V)^o + (\rho_P\phi_P V)^{oo}}{2\Delta t} \tag{3.22}$$

Crank Nicolson scheme,

3.4.4 Second time derivative

The second time derivative is integrated over a control volume and linearised as follows:

$$\frac{\partial}{\partial t}\int_V \rho\frac{\partial\phi}{\partial t} \, dV = \frac{(\rho_P\phi_P V)^n - 2\,(\rho_P\phi_P V)^o + (\rho_P\phi_P V)^{oo}}{\Delta t^2} \tag{3.23}$$

It is first order accurate in time.

3.4.5 Divergence

The divergence term described in this Section is strictly an explicit term that is distinguished from the convection term of Section 3.4.2, *i.e.* in that it is not the divergence of the product of a velocity and dependent variable. The term is integrated over a control volume and linearised as follows:

$$\int_V \nabla \bullet \phi \, dV = \int_S d\mathbf{S} \bullet \phi = \sum_f \mathbf{S}_f \bullet \phi_f \tag{3.24}$$

The `fvc::div` function can take as its argument either a surface<Type>Field, in which case ϕ_f is specified directly, or a vol<Type>Field which is interpolated to the face by central differencing as described in Section 3.4.10:

3.4.6 Gradient

The gradient term is an explicit term that can be evaluated in a variety of ways. The scheme can be evaluated either by selecting the particular grad function relevant to the discretisation scheme, *e.g.*`fv::gaussGrad`, `fv::leastSquaresGrad` *etc.*, or by using the `fvc::grad` function combined with the appropriate `keyword` in an input file

Gauss integration is invoked using the `fvc::grad` function with `Gauss` or directly using the `fvc::gGrad` function. The discretisation is performed using the standard method of applying Gauss' theorem to the volume integral:

$$\int_V \nabla\phi \, dV = \int_S d\mathbf{S}\, \phi = \sum_f \mathbf{S}_f \phi_f \tag{3.25}$$

As with the `fvc::div` function, the Gaussian integration `fvc::grad` function can take either a surfaceField<Type> or a volField<Type> as an argument.

Least squares method is based on the following idea:

1. a value at point P can be extrapolated to neighbouring point N using the gradient at P;
2. the extrapolated value at N can be compared to the actual value at N, the difference being the error;
3. if we now minimise the sum of the square of weighted errors at all neighbours of P with the respect to the gradient, then the gradient should be a good approximation.

Least squares is invoked using the `fvc::grad` function with `leastSquares` or directly using the `fv::leastSquaresGrad` function. The discretisation is performed as by first calculating the tensor $\mathbf{G}$ at every point P by summing over neighbours N:

$$\mathbf{G} = \sum_N w_N^2 \mathbf{dd} \tag{3.26}$$

where $\mathbf{d}$ is the vector from P to N and the weighting function $w_N = 1/|\mathbf{d}|$. The gradient is then evaluated as:

$$(\nabla\phi)_P = \sum_N w_N^2 \mathbf{G}^{-1} \bullet \mathbf{d}\, (\phi_N - \phi_P) \tag{3.27}$$

Surface normal gradient The gradient normal to a surface $\mathbf{n}_f \bullet (\nabla\phi)_f$ can be evaluated at cell faces using the scheme

$$(\nabla\phi)_f = \frac{\phi_N - \phi_P}{|\mathbf{d}|} \tag{3.28}$$

This gradient is called by the function `fvc::snGrad` and returns a surfaceField<Type>. The scheme is directly analogous to that evaluated for the Laplacian discretisation scheme in Section 3.4.1, and in the same manner, a correction can be introduced to improve the accuracy of this face gradient in the case of non-orthogonal meshes.

3.4.7 Grad-grad squared

The grad-grad squared term is evaluated by: taking the gradient of the field; taking the gradient of the resulting gradient field; and then calculating the magnitude squared of the result. The mathematical expression for grad-grad squared of ϕ is $|\nabla(\nabla\phi)|^2$.

3.4.8 Curl

The curl is evaluated from the gradient term described in Section 3.4.6. First, the gradient is discretised and then the curl is evaluated using the relationship from Equation 3.7, repeated here for convenience

$$\nabla \times \phi = 2 \, {*}(\text{skew} \, \nabla\phi)$$

3.4.9 Source terms

Source terms can be specified in 3 ways

Explicit Every explicit term is a volField<Type>. Hence, an explicit source term can be incorporated into an equation simply as a field of values. For example if we wished to solve Poisson's equation $\nabla^2\phi = f$, we would define `phi` and `f` as volScalarField and then do

```
solve(fvm::laplacian(phi) == f)
```

Implicit An implicit source term is integrated over a control volume and linearised by

$$\int_V \rho\phi \, dV = \rho_P V_P \phi_P \tag{3.29}$$

Implicit/Explicit The implicit source term changes the coefficient of the diagonal of the matrix. Depending on the sign of the coefficient and matrix terms, this will either increase or decrease diagonal dominance of the matrix. Decreasing the diagonal dominance could cause instability during iterative solution of the matrix equation. Therefore OpenFOAM provides a mixed source discretisation procedure that is implicit when the coefficients that are greater than zero, and explicit for the coefficients less than zero. In mathematical terms the matrix coefficient for node P is $V_P \max(\rho_P, 0)$ and the source term is $V_P \phi_P \min(\rho_P, 0)$.

3.4.10 Other explicit discretisation schemes

There are some other discretisation procedures that convert volField<Type>s into surface<Type>Fields and visa versa.

Surface integral `fvc::surfaceIntegrate` performs a summation of surface<Type>Field face values bounding each cell and dividing by the cell volume, *i.e.* $(\sum_f \phi_f)/V_P$. It returns a volField<Type>.

Surface sum `fvc::surfaceSum` performs a summation of surface<Type>Field face values bounding each cell, *i.e.* $\sum_f \phi_f$ returning a volField<Type>.

Average `fvc::average` produces an area weighted average of surface<Type>Field face values, *i.e.* $(\sum_f S_f\phi_f)/\sum_f S_f$, and returns a volField<Type>.

Reconstruct

Face interpolate The geometric<Type>Field function `faceInterpolate()` interpolates volField<Type> cell centre values to cell faces using central differencing, returning a surface<Type>Field.

3.5 Temporal discretisation

Although we have described the discretisation of temporal derivatives in Sections 3.4.3 and 3.4.4, we need to consider how to treat the spatial derivatives in a transient problem. If we denote all the spatial terms as $\mathcal{A}\phi$ where $\mathcal{A}$ is any spatial operator, *e.g.* Laplacian, then we can express a transient PDE in integral form as

$$\int_t^{t+\Delta t} \left[\frac{\partial}{\partial t} \int_V \rho\phi \; dV + \int_V \mathcal{A}\phi \; dV \right] \; dt = 0 \tag{3.30}$$

Using the Euler implicit method of Equation 3.21, the first term can be expressed as

$$\begin{aligned} \int_t^{t+\Delta t} \left[\frac{\partial}{\partial t} \int_V \rho\phi \; dV \right] \; dt &= \int_t^{t+\Delta t} \frac{(\rho_P\phi_P V)^n - (\rho_P\phi_P V)^o}{\Delta t} \; dt \\ &= \frac{(\rho_P\phi_P V)^n - (\rho_P\phi_P V)^o}{\Delta t} \Delta t \end{aligned} \tag{3.31}$$

The second term can be expressed as

$$\int_t^{t+\Delta t} \left[\int_V \mathcal{A}\phi \; dV \right] \; dt = \int_t^{t+\Delta t} \mathcal{A}^*\phi \; dt \tag{3.32}$$

where $\mathcal{A}^*$ represents the spatial discretisation of $\mathcal{A}$. The time integral can be discretised in three ways:

Euler implicit uses implicit discretisation of the spatial terms, thereby taking current values ϕ^n.

$$\int_t^{t+\Delta t} \mathcal{A}^*\phi \; dt = \mathcal{A}^*\phi^n \Delta t \tag{3.33}$$

It is first order accurate in time, guarantees boundedness and is unconditionally stable.

Explicit uses explicit discretisation of the spatial terms, thereby taking old values ϕ^o.

$$\int_t^{t+\Delta t} \mathcal{A}^*\phi \; dt = \mathcal{A}^*\phi^o \Delta t \tag{3.34}$$

It is first order accurate in time and is unstable if the Courant number Co is greater than 1. The Courant number is defined as

$$Co = \frac{\mathbf{U}_f \bullet \mathbf{d}}{|\mathbf{d}|^2 \Delta t} \tag{3.35}$$

where $\mathbf{U}_f$ is a characteristic velocity, *e.g.* velocity of a wave front, velocity of flow.

Crank Nicholson uses the trapezoid rule to discretise the spatial terms, thereby taking a mean of current values ϕ^n and old values ϕ^o.

$$\int_t^{t+\Delta t} \mathcal{A}^*\phi \; dt = \mathcal{A}^* \left(\frac{\phi^n + \phi^o}{2} \right) \Delta t \tag{3.36}$$

It is second order accurate in time, is unconditionally stable but does not guarantee boundedness.

3.5.1 Treatment of temporal discretisation in OpenFOAM

At present the treatment of the temporal discretisation is controlled by the implementation of the spatial derivatives in the PDE we wish to solve. For example, let us say we wish to solve a transient diffusion equation

$$\frac{\partial \phi}{\partial t} = \kappa \nabla^2 \phi \tag{3.37}$$

An Euler implicit implementation of this would read

```
solve(fvm::ddt(phi) == kappa*fvm::laplacian(phi))
```

where we use the `fvm` class to discretise the `Laplacian` term implicitly. An explicit implementation would read

```
solve(fvm::ddt(phi) == kappa*fvc::laplacian(phi))
```

where we now use the `fvc` class to discretise the `Laplacian` term explicitly. The Crank Nicholson scheme can be implemented by the mean of implicit and explicit terms:

```
solve
    (
    fvm::ddt(phi)
    ==
    kappa*0.5*(fvm::laplacian(phi) + fvc::laplacian(phi))
    )
```

3.6 Boundary Conditions

Boundary conditions are required to complete the problem we wish to solve. We therefore need to specify boundary conditions on all our boundary faces. Boundary conditions can be divided into 2 types:

Dirichlet prescribes the value of the dependent variable on the boundary and is therefore termed 'fixed value' in this guide;

Neumann prescribes the gradient of the variable normal to the boundary and is therefore termed 'fixed gradient' in this guide.

When we perform discretisation of terms that include the sum over faces $\sum_f$, we need to consider what happens when one of the faces is a boundary face.

Fixed value We specify a fixed value at the boundary ϕ_b

- We can simply substitute ϕ_b in cases where the discretisation requires the value on a boundary face ϕ_f, *e.g.* in the convection term in Equation 3.16.
- In terms where the face gradient $(\nabla\phi)_f$ is required, *e.g.* Laplacian, it is calculated using the boundary face value and cell centre value,

$$\mathbf{S}_f \bullet (\nabla\phi)_f = |S_f| \frac{\phi_b - \phi_P}{|\mathbf{d}|} \tag{3.38}$$

Fixed gradient The fixed gradient boundary condition g_b is a specification on inner product of the gradient and unit normal to the boundary, or

$$g_b = \left(\frac{\mathbf{S}}{|\mathbf{S}|} \bullet \nabla \phi \right)_f \tag{3.39}$$

- When discretisation requires the value on a boundary face ϕ_f we must interpolate the cell centre value to the boundary by

$$\begin{aligned} \phi_f &= \phi_P + \mathbf{d} \bullet (\nabla \phi)_f \\ &= \phi_P + |\mathbf{d}|\, g_b \end{aligned} \tag{3.40}$$

- ϕ_b can be directly substituted in cases where the discretisation requires the face gradient to be evaluated,

$$\mathbf{S}_f \bullet (\nabla \phi)_f = |S_f|\, g_b \tag{3.41}$$

3.6.1 Physical boundary conditions

The specification of boundary conditions is usually an engineer's interpretation of the true behaviour. Real boundary conditions are generally defined by some physical attributes rather than the numerical description as described of the previous Section. In incompressible fluid flow there are the following physical boundaries

Inlet The velocity field at the inlet is supplied and, for consistency, the boundary condition on pressure is zero gradient.

Outlet The pressure field at the outlet is supplied and a zero gradient boundary condition on velocity is specified.

No-slip impermeable wall The velocity of the fluid is equal to that of the wall itself, *i.e.* a fixed value condition can be specified. The pressure is specified zero gradient since the flux through the wall is zero.

In a problem whose solution domain and boundary conditions are symmetric about a plane, we only need to model half the domain to one side of the symmetry plane. The boundary condition on the plane must be specified according to

Symmetry plane The symmetry plane condition specifies the component of the gradient normal to the plane should be zero. [Check**]

Appendix A

General tensor mathematics

This Chapter describes tensors and their algebraic operations and how they are represented in mathematical text.

A.1 Tensors

A.1.1 Tensor notation

OpenFOAM deals with problems involving complex PDEs in 3 spatial dimensions and in time. It is vital from the beginning to adopt a notation for the equations which is compact yet unambiguous. To make the equations easy to follow, we must use a notation that encapsulates the idea of a tensor as an entity in the own right, rather than a list of scalar components. Additionally, any tensor operation should be perceived as an operation on the entire tensor entity rather than a series of operations on its components.

Consequently, *tensor notation* is preferred in which any tensor of rank 1 and above, *i.e.* all tensors other than scalars, are represented by letters in bold face, *e.g.* $\mathbf{a}$. This actively promotes the concept of a tensor as a entity in its own right since it is denoted by a single symbol, and it is also extremely compact. The potential drawback is that the rank of a bold face symbol is not immediately apparent, although it is clearly not zero. However, in practice this presents no real problem since we are aware of the property each symbol represents and therefore intuitively know its rank, *e.g.* we know velocity $\mathbf{U}$ is a tensor of rank 1.

A further, more fundamental idea regarding the choice of notation is that the mathematical representation of a tensor should not change depending on our coordinate system, *i.e.* the vector $\mathbf{a}$is the same vector irrespective of where we view it from. The tensor notation supports this concept as it implies nothing about the coordinate system. However, other notations, *e.g.* a_i, expose the individual components of the tensor which naturally implies the choice of coordinate system. The unsatisfactory consequence of this is that the tensor is then represented by a set of values which are not unique — they depend on the coordinate system.

That said, the index notation, introduced in Section A.1, is adopted from time to time in this book mainly to expand tensor operations into the constituent components. When using the index notation, we adopt the *summation convention* which states that whenever the same letter subscript occurs twice in a term, the that subscript is to be given all values, *i.e.* $1, 2, 3$, and the results added together, *e.g.*

$$a_i b_i = \sum_{i=1}^{3} a_i b_i = a_1 b_1 + a_2 b_2 + a_3 b_3 \tag{A.1}$$

In the remainder of the text the symbol $\sum$ is omitted since the repeated subscript indicates the summation.

A.2 Algebraic tensor operations

This section describes all the algebraic operations for tensors that are available in OpenFOAM. Let us first review the most simple tensor operations: addition, subtraction, and scalar multiplication and division. Addition is commutative and associative and only valid between tensors of the same rank. Subtraction is also only valid between tensors of the same rank, but is neither commutative or associative. The operations are performed by addition/subtraction of the respective components of the tensors, *e.g.* the subtraction of two vectors **a** and **b** is

$$\mathbf{a} - \mathbf{b} = a_i - b_i = (a_1 - b_1, a_2 - b_2, a_3 - b_3) \tag{A.2}$$

Multiplication of any tensor **a** by a scalar s is also commutative and associative and is performed by multiplying all the tensor components by the scalar. For example,

$$s\mathbf{a} = sa_i = (sa_1, sa_2, sa_3) \tag{A.3}$$

Division between a tensor **a** and a scalar is only relevant when the scalar is the second argument of the operation, *i.e.*

$$\mathbf{a}/s = a_i/s = (a_1/s, a_2/s, a_3/s) \tag{A.4}$$

Following these operations are a set of more complex products between tensors of rank 1 and above, described in the following Sections.

A.2.1 The inner product

The inner product operates on any two tensors of rank r_1 and r_2 such that the rank of the result $r = r_1 + r_2 - 2$. Inner product operations with tensors up to rank 3 are described below:

- The inner product of two vectors **a** and **b** is commutative and produces a scalar $s = \mathbf{a} \bullet \mathbf{b}$ where

$$s = a_i b_i = a_1 b_1 + a_2 b_2 + a_3 b_3 \tag{A.5}$$

- The inner product of a tensor **T** and vector **a** produces a vector $\mathbf{b} = \mathbf{T} \bullet \mathbf{a}$, represented below as a column array for convenience

$$b_i = T_{ij} a_j = \begin{pmatrix} T_{11}a_1 + T_{12}a_2 + T_{13}a_3 \\ T_{21}a_1 + T_{22}a_2 + T_{23}a_3 \\ T_{31}a_1 + T_{32}a_2 + T_{33}a_3 \end{pmatrix} \tag{A.6}$$

  It is non-commutative if **T** is non-symmetric such that $\mathbf{b} = \mathbf{a} \bullet \mathbf{T} = \mathbf{T}^{\mathrm{T}} \bullet \mathbf{a}$ is

$$b_i = a_j T_{ji} = \begin{pmatrix} a_1 T_{11} + a_2 T_{21} + a_3 T_{31} \\ a_1 T_{12} + a_2 T_{22} + a_3 T_{32} \\ a_1 T_{13} + a_2 T_{23} + a_3 T_{33} \end{pmatrix} \tag{A.7}$$

- The inner product of two tensors $\mathbf{T}$ and $\mathbf{S}$ produces a tensor $\mathbf{P} = \mathbf{T} \bullet \mathbf{S}$ whose components are evaluated as:

  $$P_{ij} = T_{ik} S_{kj} \tag{A.8}$$

  It is non-commutative such that $\mathbf{T} \bullet \mathbf{S} = \left(\mathbf{S}^{\mathrm{T}} \bullet \mathbf{T}^{\mathrm{T}} \right)^{\mathrm{T}}$

- The inner product of a vector $\mathbf{a}$ and third rank tensor $\mathbf{P}$ produces a second rank tensor $\mathbf{T} = \mathbf{a} \bullet \mathbf{P}$ whose components are

  $$T_{ij} = a_k P_{kij} \tag{A.9}$$

  Again this is non-commutative so that $\mathbf{T} = \mathbf{P} \bullet \mathbf{a}$ is

  $$T_{ij} = P_{ijk} a_k \tag{A.10}$$

- The inner product of a second rank tensor $\mathbf{T}$ and third rank tensor $\mathbf{P}$ produces a third rank tensor $\mathbf{Q} = \mathbf{T} \bullet \mathbf{P}$ whose components are

  $$Q_{ijk} = T_{il} P_{ljk} \tag{A.11}$$

  Again this is non-commutative so that $\mathbf{Q} = \mathbf{P} \bullet \mathbf{T}$ is

  $$Q_{ijk} = P_{ijl} T_{lk} \tag{A.12}$$

A.2.2 The double inner product of two tensors

The double inner product of two second-rank tensors $\mathbf{T}$ and $\mathbf{S}$ produces a scalar $s = \mathbf{T} \colon \mathbf{S}$ which can be evaluated as the sum of the 9 products of the tensor components

$$\begin{aligned} s = T_{ij} S_{ij} = \; & T_{11} S_{11} + T_{12} S_{12} + T_{13} S_{13} + \\ & T_{21} S_{21} + T_{22} S_{22} + T_{23} S_{23} + \\ & T_{31} S_{31} + T_{32} S_{32} + T_{33} S_{33} \end{aligned} \tag{A.13}$$

The double inner product between a second rank tensor $\mathbf{T}$ and third rank tensor $\mathbf{P}$ produces a vector $\mathbf{a} = \mathbf{T} \colon \mathbf{P}$ with components

$$a_i = T_{jk} P_{jki} \tag{A.14}$$

This is non-commutative so that $\mathbf{a} = \mathbf{P} \colon \mathbf{T}$ is

$$a_i = P_{ijk} T_{jk} \tag{A.15}$$

A.2.3 The triple inner product of two third rank tensors

The triple inner product of two third rank tensors $\mathbf{P}$ and $\mathbf{Q}$ produces a scalar $s = \mathbf{P} \,\vdots\, \mathbf{Q}$ which can be evaluated as the sum of the 27 products of the tensor components

$$s = P_{ijk} Q_{ijk} \tag{A.16}$$

A.2.4 The outer product

The outer product operates between vectors and tensors as follows:

- The outer product of two vectors **a** and **b** is non-commutative and produces a tensor $\mathbf{T} = \mathbf{ab} = (\mathbf{ba})^{\mathrm{T}}$ whose components are evaluated as:

$$T_{ij} = a_i b_j = \begin{pmatrix} a_1b_1 & a_1b_2 & a_1b_3 \\ a_2b_1 & a_2b_2 & a_2b_3 \\ a_3b_1 & a_3b_2 & a_3b_3 \end{pmatrix} \tag{A.17}$$

- An outer product of a vector **a** and second rank tensor **T** produces a third rank tensor $\mathbf{P} = \mathbf{aT}$ whose components are

$$P_{ijk} = a_i T_{jk} \tag{A.18}$$

  This is non-commutative so that $\mathbf{P} = \mathbf{T}\,\mathbf{a}$ produces

$$P_{ijk} = T_{ij} a_k \tag{A.19}$$

A.2.5 The cross product of two vectors

The cross product operation is exclusive to vectors only. For two vectors **a** with **b**, it produces a vector $\mathbf{c} = \mathbf{a} \times \mathbf{b}$ whose components are

$$c_i = e_{ijk} a_j b_k = (a_2b_3 - a_3b_2, a_3b_1 - a_1b_3, a_1b_2 - a_2b_1) \tag{A.20}$$

where the *permutation symbol* is defined by

$$e_{ijk} = \begin{cases} 0 & \text{when any two indices are equal} \\ +1 & \text{when } i,j,k \text{ are an even permutation of 1,2,3} \\ -1 & \text{when } i,j,k \text{ are an odd permutation of 1,2,3} \end{cases} \tag{A.21}$$

in which the even permutations are 123, 231 and 312 and the odd permutations are 132, 213 and 321.

A.2.6 Other general tensor operations

Some less common tensor operations and terminology used by OpenFOAM are described below.

Square of a tensor is defined as the outer product of the tensor with itself, *e.g.* for a vector **a**, the square $\mathbf{a}^2 = \mathbf{aa}$.

***n*th power** of a tensor is evaluated by n outer products of the tensor, *e.g.* for a vector **a**, the 3rd power $\mathbf{a}^3 = \mathbf{aaa}$.

Magnitude squared of a tensor is the rth inner product of the tensor of rank r with itself, to produce a scalar. For example, for a second rank tensor **T**, $|\mathbf{T}|^2 = \mathbf{T} \boldsymbol{:} \mathbf{T}$.

Magnitude is the square root of the magnitude squared, *e.g.* for a tensor **T**, $|\mathbf{T}| = \sqrt{\mathbf{T} \boldsymbol{:} \mathbf{T}}$. Vectors of unit magnitude are referred to as *unit vectors*.

Component maximum is the component of the tensor with greatest value, inclusive of sign, *i.e.* not the largest magnitude.

Component minimum is the component of the tensor with smallest value.

Component average is the mean of all components of a tensor.

Scale As the name suggests, the scale function is a tool for scaling the components of one tensor by the components of another tensor of the same rank. It is evaluated as the product of corresponding components of 2 tensors, *e.g.*, scaling vector **a** by vector **b** would produce vector **c** whose components are

$$c_i = \mathrm{scale}(\mathbf{a}, \mathbf{b}) = (a_1 b_1, a_2 b_2, a_3 b_3) \tag{A.22}$$

A.2.7 Geometric transformation and the identity tensor

A second rank tensor **T** is strictly defined as a linear vector function, i.e. it is a function which associates an argument vector **a** to another vector **b** by the inner product $\mathbf{b} = \mathbf{T} \bullet \mathbf{a}$. The components of **T** can be chosen to perform a specific geometric transformation of a tensor from the x, y, z coordinate system to a new coordinate system x^*, y^*, z^*; **T** is then referred to as the *transformation tensor*. While a scalar remains unchanged under a transformation, the vector **a** is transformed to $\mathbf{a}^*$ by

$$\mathbf{a}^* = \mathbf{T} \bullet \mathbf{a} \tag{A.23}$$

A second rank tensor **S** is transformed to $\mathbf{S}^*$ according to

$$\mathbf{S}^* = \mathbf{T} \bullet \mathbf{S} \bullet \mathbf{T}^{\mathrm{T}} \tag{A.24}$$

The *identity tensor* **I** is defined by the requirement that it transforms another tensor onto itself. For all vectors **a**

$$\mathbf{a} = \mathbf{I} \bullet \mathbf{a} \tag{A.25}$$

and therefore

$$\mathbf{I} = \delta_{ij} = \begin{pmatrix} 1 & 0 & 0 \\ 0 & 1 & 0 \\ 0 & 0 & 1 \end{pmatrix} \tag{A.26}$$

where δ_{ij} is known as the *Kronecker delta* symbol.

A.2.8 Useful tensor identities

Several identities are listed below which can be verified by under the assumption that all the relevant derivatives exist and are continuous. The identities are expressed for scalar s and vector **a**.

$$\begin{aligned}
&\nabla \bullet (\nabla \times \mathbf{a}) \equiv 0 \\
&\nabla \times (\nabla s) \equiv \mathbf{0} \\
&\nabla \bullet (s\mathbf{a}) \equiv s\nabla \bullet \mathbf{a} + \mathbf{a} \bullet \nabla s \\
&\nabla \times (s\mathbf{a}) \equiv s\nabla \times \mathbf{a} + \nabla s \times \mathbf{a} \\
&\nabla(\mathbf{a} \bullet \mathbf{b}) \equiv \mathbf{a} \times (\nabla \times \mathbf{b}) + \mathbf{b} \times (\nabla \times \mathbf{a}) + (\mathbf{a} \bullet \nabla)\mathbf{b} + (\mathbf{b} \bullet \nabla)\mathbf{a} \\
&\nabla \bullet (\mathbf{a} \times \mathbf{b}) \equiv \mathbf{b} \bullet (\nabla \times \mathbf{a}) - \mathbf{a} \bullet (\nabla \times \mathbf{b}) \\
&\nabla \times (\mathbf{a} \times \mathbf{b}) \equiv \mathbf{a}(\nabla \bullet \mathbf{b}) - \mathbf{b}(\nabla \bullet \mathbf{a}) + (\mathbf{b} \bullet \nabla)\mathbf{a} - (\mathbf{a} \bullet \nabla)\mathbf{b} \\
&\nabla \times (\nabla \times \mathbf{a}) \equiv \nabla(\nabla \bullet \mathbf{a}) - \nabla^2 \mathbf{a} \\
&(\nabla \times \mathbf{a}) \times \mathbf{a} \equiv \mathbf{a} \bullet (\nabla \mathbf{a}) - \nabla(\mathbf{a} \bullet \mathbf{a})
\end{aligned} \tag{A.27}$$

It is sometimes useful to know the $e - \delta$ identity to help to manipulate equations in index notation:

$$e_{ijk}e_{irs} = \delta_{jr}\delta_{ks} - \delta_{js}\delta_{kr} \tag{A.28}$$

A.2.9 Operations exclusive to tensors of rank 2

There are several operations that manipulate the components of tensors of rank 2 that are listed below:

Transpose of a tensor $\mathbf{T} = T_{ij}$ is $\mathbf{T}^{\mathrm{T}} = T_{ji}$ as described in Equation 2.2.

Symmetric and skew (antisymmetric) tensors As discussed in section A.1, a tensor is said to be symmetric if its components are symmetric about the diagonal, i.e. $\mathbf{T} = \mathbf{T}^{\mathrm{T}}$. A skew or antisymmetric tensor has $\mathbf{T} = -\mathbf{T}^{\mathrm{T}}$ which intuitively implies that $T_{11} = T_{22} = T_{33} = 0$. Every second order tensor can be decomposed into symmetric and skew parts by

$$\mathbf{T} = \underbrace{\frac{1}{2}(\mathbf{T} + \mathbf{T}^{\mathrm{T}})}_{symmetric} + \underbrace{\frac{1}{2}(\mathbf{T} - \mathbf{T}^{\mathrm{T}})}_{skew} = \mathrm{symm}\,\mathbf{T} + \mathrm{skew}\,\mathbf{T} \tag{A.29}$$

Trace The trace of a tensor $\mathbf{T}$ is a scalar, evaluated by summing the diagonal components

$$\mathrm{tr}\,\mathbf{T} = T_{11} + T_{22} + T_{33} \tag{A.30}$$

Diagonal returns a vector whose components are the diagonal components of the second rank tensor $\mathbf{T}$

$$\mathrm{diag}\,\mathbf{T} = (T_{11}, T_{22}, T_{33}) \tag{A.31}$$

Deviatoric and hydrostatic tensors Every second rank tensor $\mathbf{T}$ can be decomposed into a deviatoric component, for which $\mathrm{tr}\,\mathbf{T} = 0$ and a hydrostatic component of the form $\mathbf{T} = s\mathbf{I}$ where s is a scalar. Every second rank tensor can be decomposed into deviatoric and hydrostatic parts as follows:

$$\mathbf{T} = \underbrace{\mathbf{T} - \frac{1}{3}(\mathrm{tr}\,\mathbf{T})\,\mathbf{I}}_{deviatoric} + \underbrace{\frac{1}{3}(\mathrm{tr}\,\mathbf{T})\,\mathbf{I}}_{hydrostatic} = \mathrm{dev}\,\mathbf{T} + \mathrm{hyd}\,\mathbf{T} \tag{A.32}$$

Determinant The determinant of a second rank tensor is evaluated by

$$\begin{aligned}\det \mathbf{T} = \begin{vmatrix} T_{11} & T_{12} & T_{13} \\ T_{21} & T_{22} & T_{23} \\ T_{31} & T_{32} & T_{33} \end{vmatrix} &= \begin{aligned} &T_{11}(T_{22}T_{33} - T_{23}T_{32}) - \\ &T_{12}(T_{21}T_{33} - T_{23}T_{31}) + \\ &T_{13}(T_{21}T_{32} - T_{22}T_{31}) \end{aligned} \\ &= \frac{1}{6}e_{ijk}e_{pqr}T_{ip}T_{jq}T_{kr}\end{aligned} \tag{A.33}$$

Cofactors The *minors* of a tensor are evaluated for each component by deleting the row and column in which the component is situated and evaluating the resulting entries as a 2×2 *determinant*. For example, the minor of T_{12} is

$$\begin{vmatrix} \cancel{T_{11}} & \cancel{T_{12}} & \cancel{T_{13}} \\ T_{21} & \cancel{T_{22}} & T_{23} \\ T_{31} & \cancel{T_{32}} & T_{33} \end{vmatrix} = \begin{vmatrix} T_{21} & T_{23} \\ T_{31} & T_{33} \end{vmatrix} = T_{21}T_{33} - T_{23}T_{31} \tag{A.34}$$

The cofactors are *signed minors* where each minor is component is given a sign based on the rule

$$\begin{aligned} &+\text{ve if } i+j \text{ is even} \\ &-\text{ve if } i+j \text{ is odd} \end{aligned} \tag{A.35}$$

The cofactors of $\mathbf{T}$ can be evaluated as

$$\operatorname{cof} \mathbf{T} = \frac{1}{2} e_{jkr} e_{ist} T_{sk} T_{tr} \tag{A.36}$$

Inverse The inverse of a tensor can be evaluated as

$$\operatorname{inv} \mathbf{T} = \frac{\operatorname{cof} \mathbf{T}^{\mathrm{T}}}{\det \mathbf{T}} \tag{A.37}$$

Hodge dual of a tensor is a vector whose components are

$$* \mathbf{T} = (T_{23}, -T_{13}, T_{12}) \tag{A.38}$$

A.2.10 Operations exclusive to scalars

OpenFOAM supports most of the well known functions that operate on scalars, *e.g.* square root, exponential, logarithm, sine, cosine *etc.*., a list of which can be found in Table 2.2. There are 3 additional functions defined within OpenFOAM that are described below:

Sign of a scalar s is

$$\operatorname{sgn}(s) = \begin{cases} 1 & \text{if } s \geq 0, \\ -1 & \text{if } s < 0. \end{cases} \tag{A.39}$$

Positive of a scalar s is

$$\operatorname{pos}(s) = \begin{cases} 1 & \text{if } s \geq 0, \\ 0 & \text{if } s < 0. \end{cases} \tag{A.40}$$

Limit of a scalar s by the scalar n

$$\operatorname{limit}(s, n) = \begin{cases} s & \text{if } s < n, \\ 0 & \text{if } s \geq n. \end{cases} \tag{A.41}$$

Index