The Flavours of OpenFOAM®

Dr Darrin Stephens

Laminar2 Turbulent Pty Ltd

Outline

What is OpenFOAM®?

- OpenFOAM® is a general purpose Computational Mechanics (CM) suite of libraries, solvers, and utilities.
- OpenFOAM® is a registered trademark of the ESI Group.
- OpenFOAM is licensed under the GNU General Public License (GPL).
- OpenFOAM is completely free of charge and can be used in academic and commercial applications.
- Unlike commercial codes, OpenFOAM is not just a monolithic application - it is a general-purpose library that can be used to create a CM solver tuned for a particular problem.

What is OpenFOAM®?

- FOAM is an acronym that stands for **F**ield **O**peration **A**nd **M**anipulation.
- In simple terms, OpenFOAM is a library of tools that provide for the manipulation of fields.
- Examples of fields might include temperature, velocity, pressure, density, magnetic, electric fields and so on.
- Fields can be of scalar, vector or tensor types.
- The library provides a consistent set of tools to perform algebraic and differential operations on fields.
- Library is implemented in the C++ language using object orientation, inheritance, templates and programming patterns.
- The code is fully parallel and does not impose any restrictions on the number of parallel processes.

What is OpenFOAM®?

- In addition to field operations, the library implements additional tools required for the solution of problems:
  - Mesh conversion and generation tools
  - Parallel communication
  - Mapping utilities for multi-physics simulations
  - Comprehensive linear algebra suite for the solution of large linear systems
- All tools are designed to work together and exchange information in order to create solvers for complex problems.
- The publicly available versions of OpenFOAM is not a polished commercial product, and some work is required if one steps out of the pre-defined examples.

Why use OpenFOAM?

There is one main reason.

- Freedom of open source.

The freedom that open source provides drives two motivations for OpenFOAM usage:

- Free as in "free food". You can download and install OpenFOAM without paying any money.
- Free as in "free speech". With OpenFOAM, you have the following freedoms
  - use it for any purpose,
  - modify it to suit your needs,
  - share it with anyone,
  - share your changes.

History of OpenFOAM

- FOAM was originally developed in the 1990s at Imperial College by Henry Weller, Hrvoje Jasak and others.
- FOAM sold as commercial software 2000-2004 by Nabla Ltd. Nabla Ltd. wound up in 2006.
- FOAM was released into the public domain in 2004 as OpenFOAM.
- Henry Weller and others founded OpenCFD Ltd (2004) and later CFD Direct Ltd (2015).
- Hrvoje Jasak founded Wikki Ltd (2004).
- OpenCFD Ltd sold to SGI in 2011, and OpenFOAM Foundation was formed.
- OpenCFD Ltd sold to ESI Group in 2012.

History of OpenFOAM

Imperial College London → **FOAM** ← Nabla Ltd

↓

OpenFOAM

↓

Main Flavours

OpenFOAM vX	OpenFOAM v202X	foam-extend
www.openfoam.org	www.openfoam.com	foam-extend.org
OpenFOAM Foundation (Copyright owners)	OpenCFD Ltd/ESI (Trademark owners)	Wikki Ltd
CFD Direct Ltd	Community	Community

Implemented capability - all flavours

- Finite Volume Method,
- Collocated polyhedral unstructured meshes,
- Second order in space and time,
- Steady and transient solvers with pressure velocity coupling via segregated methods,
- Lagrangian particle tracking,
- Automatic mesh motion with support for topological changes,
- Massive parallelism in domain decomposition mode,
- All components implemented in library form for easy re-use

Implemented capability - all flavours

Physical and multi-physics modelling:

- Turbulence modelling, RANS, DES, LES,
- Multiphase flows and mass transfer,
- Combustion and chemical reactions,
- Rheology models,
- Thermophysical models for liquids and gases.
- Conjugate heat transfer
- Stress analysis and fluid-structure interaction
- 6 DoF
- Aero-acoustics

Flavour differences

Some of the main feature differences:

- Foundation (OpenFOAM vX)
  - Modular framework
- ESI (OpenFOAM v202X)
  - Overset
  - Community add-ons, cfMesh, OpenQBMM, Petsc4Foam
- foam-extend
  - Coupled framework and solvers
  - Immersed boundary method
  - Finite area method (now ported to ESI version)
  - Overset
  - Proper Orthogonal Decomposition

The environment variable $WM_PROJECT_DIR holds the OpenFOAM installation path.

```
$WM_PROJECT_DIR
 ├── Allwmake
 ├── applications ...solvers/utilities sources.
 ├── bin
 ├── build
 ├── CONTRIBUTORS.md
 ├── COPYING
 ├── doc
 ├── etc
 ├── META-INFO
 ├── modules
 ├── platforms ...compiled code.
 ├── README.md
 ├── site
 ├── src ...libraries sources.
 ├── tutorials ...Tutorials cases.
 └── wmake
```

Solvers/Utilities

- OpenFOAM is not a single executable.
- Depending on what you want to do, you will need to use a specific application.
- Solvers can be found in the $WM_PROJECT_DIR/applications/solvers directory ($FOAM_SOLVERS)
- Utilities can be found in the $WM_PROJECT_DIR/applications/utilities directory ($FOAM_UTILITIES)
- If you want to get help on how to run an application, type in terminal

```
$> application_name -help
```

Solvers

- Solvers are subdivided according to the physics they address.
- The solvers available in the OpenFOAM installation (version 2212):
  - acoustics
  - basic
  - combustion
  - compressible
  - discreteMethods
  - DNS
  - electromagnetics
  - financial
  - finiteArea
  - heatTransfer
  - incompressible
  - lagrangian
  - multiphase
  - stressAnalysis
- Each sub-directory may contain several sub-directories, one for each solver.
- Each solver directory contains a *.C file with the same name as the directory. This file contains a short description of the solver.

Solvers

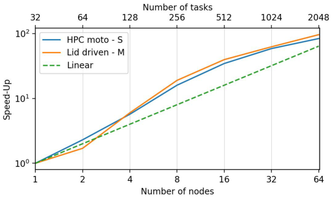
- For example, the *incompressible* sub-directory contains the following solvers:
  - adjointOptimisationFoam
  - adjointShapeOptimizationFoam
  - boundaryFoam
  - icoFoam
  - nonNewtonianIcoFoam
  - pimpleFoam
  - pisoFoam
  - shallowWaterFoam
  - simpleFoam

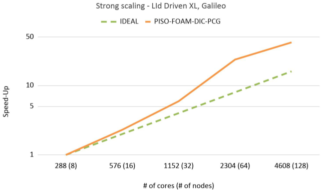
Commonly used OpenFOAM solvers and their supported physics.

Physics	bPF	cMRF	iF	pF	rPF	rPF	sF
transient	✓	✓	✓	✓	✓		
compressible	✓	✓			✓	✓	
turbulence	✓	✓	✓	✓	✓	✓	✓
heat-transfer	✓	✓			✓	✓	
buoyancy	✓	✓			✓		
combustion					✓		
multiphase			✓				
particles					✓		
dynamic mesh			✓	✓		✓	
multi-region		✓					

- bPF - buoyantPimpleFoam
- cMRF - chtMultiRegionFoam
- iF - interFoam
- pF - pimpleFoam
- rPF - reactingParcelFoam
- rPF - rhoPimpleFoam
- sF - simpleFoam

OpenFOAM has been shown to scale to in excess of 4000 cores cores[1]

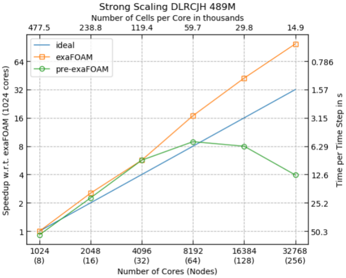




[1] https://wiki.openfoam.com/images/c/c1/OpenFOAM_2020_CINECA_Spisso.pdf

exaFOAM

- Consortium (12 Partners) led by ESI-OpenCFD consisting of experts to work on the co-design of OpenFOAM targeting (pre)-exascale HPC architectures.
- Grant Funded by EuroHPC-03-2019 totalling €5,425,618
- Outcomes implemented in OpenFOAM V202X, example shown below[2]



[2] https://exafoam.eu/wp-content/uploads/2023/07/exaFOAM_Workshop_GrandChallenges.pdf

Object Orientation in OpenFOAM

- OpenFOAM is assembled from components:
  - Core libraries containing discretization, mesh handling, etc.
  - Physical modelling libraries such as thermo-physical models, turbulence models, chemical reactions, particle collision models, etc.
  - Utilities for mesh import and manipulation, parallel processing, etc.
  - Solvers implemented as a top-level code utilising library components.
- Top-level code consists of a few hundred lines of C++ code.
- In essence, top-level code defines a computational programming language.

Object-Oriented analysis

- Main objects are operators, e.g. time derivative, divergence, curl, Laplacian, etc.
- Operators act on fields, and depending on the desired result of the operation in OpenFOAM, we distinguish two implementation concepts:
  - Field (explicit evaluation of the operator acting on a field)
  - Matrix (implicit evaluation of operator acting on a field)
- We also need a few more components to represent equations in OpenFOAM:
  - Computational domain (space-time representation)
  - Basic containers (scalars, vectors, and tensors)
  - Fields and their associated algebra
  - Matrices for sparse linear systems
  - Discretization methods for operators

Computational Domain / Field Algebra

- The computational domain is represented through space-time domain implementation:
  - Mesh primitives: points, faces, cells → `point`, `face`, `cell` classes
  - Space: computational mesh → `polyMesh` class
  - Time: time steps (database) → `time` class
- Field algebra and containers:
  - Tensor: list of numbers + algebra → `scalar`, `vector`, `tensor` classes
  - Boundary condition: list of values + condition → `patchField` class
  - Physical dimensions: dimension set → `dimensionSet` class
  - Geometric field: field + mesh + boundary conditions → `geometricField` class
  - Field algebra: $+$, $-$, $*$, $/$, $\sin()$, $\exp()$ → field operators (C++ operator overloading)

- Linear equation systems and linear solvers:
  - Linear equation matrix: table of matrix coefficients → `lduMatrix` class
  - Solvers: iterative solvers → `lduMatrix::solver`
- Discretization methods:
  - Interpolation: differencing schemes → `interpolation` class.
  - Differentiation: ddt, div, grad, etc. → `fvc`, `fec` classes.
  - Discretization: ddt, div, Laplacian, etc. → `fvm`, `fam` classes.
- Top-level code:
  - Model library: library → `turbulenceModel` class for example.
  - Application: `main()` → `pisoFoam` solver for example

Equation mimicking

- Uses equation mimicking
- Consider PDE for the transport of turbulent kinetic energy:

$$\frac{\partial k}{\partial t} + \nabla\cdot(\mathbf{u}\,k) - \nabla\cdot[(\nu+\nu_t)\nabla k] = \nu_t\left[\frac{1}{2}(\nabla\mathbf{u} + \nabla\mathbf{u}^T)\right]^2 - \frac{\varepsilon_0}{k_0}\,k$$

- Implementation:

```
solve
(
      fvm::ddt(k)
    + fvm::div(phi,k)
    - fvm::laplacian(nu()+nut,k)
   == nut*magSqr(symm(fvc::grad(U)))
    - fvm::Sp(epsilon/k,k)
);
```

Model implementation

- Model-to-model interaction is handled through common interfaces.
- New components do not disturb existing code, resulting in fewer bugs.
- Run-time selection tables for dynamic binding of new functionality.
- This approach is used for every implementation:
  - Differencing schemes
  - Gradient calculations
  - Boundary conditions
  - Linear equation solvers
  - Physical models
  - Mesh motion algorithms

The key aspect of model implementation is class inheritance and factory mechanism.

Model implementation

- Consider the turbulence model:

```
class turbulenceModel
{
    virtual volTensorField R() const = 0;
    virtual fvVectorMatrix divR
    (
        volVectorField& U
    ) const = 0;
    virtual void correct() = 0;
};
```

- New turbulence model implementation funkyTurbulence:

```
class funkyTurbulence :
    public turbulenceModel {};
```

Model implementation

- Main code sees only virtual interface:

```
autoPtr<turbulenceModel> turbulence
(
    turbulenceModel::New(U,phi,laminarTransport)
);

fvVectorMatrix UEqn
(
    fvm::ddt(rho,U)
  + fvm::div(phi, U)
  + turbulence->divR(U)
 ==   - fvc::grad(p)
);
```

- Turbulence model selected by user at run-time.

Laplacian Scalar Equation

- PDE for Laplacian scalar equation:

$$\frac{\partial T}{\partial t} - \nabla^2(\mathcal{D}_T\, T) = 0$$

- Code:

```
for(int nonOrth=0;
    nonOrth<=nNonOrthCorr;
    nonOrth++)
{
    solve
    (
        fvm::ddt(T) - fvm::laplacian(DT, T)
    );
}
```

Scalar Transport Equation

- PDE for scalar transport:

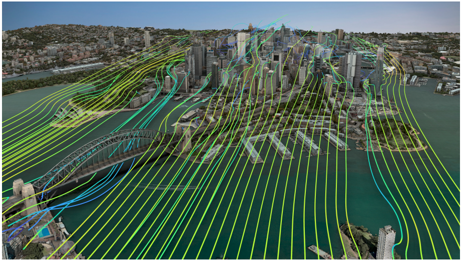
$$\frac{\partial T}{\partial t} + \nabla \cdot (\mathbf{U}\, T) - \nabla^2(\mathcal{D}_T\, T) = 0$$

- Code:

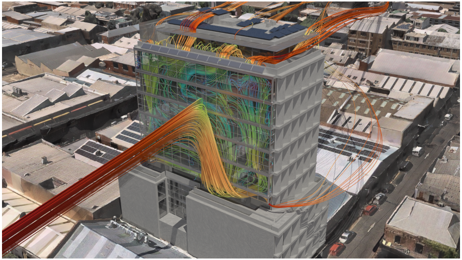
```
solve
(
      fvm::ddt(T)
    + fvm::div(phi, T)
    - fvm::laplacian(DT, T)
);
```

Examples

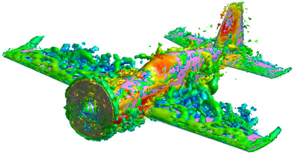
- The following slides are only a part of what is possible with OpenFOAM
- Chosen for (personal) interest and illustration of the range of capabilities rather than an exhaustive illustration of the range of capabilities
- All results shown are from simulations conducted either by Laminar2 Turbulent or Applied CCM.

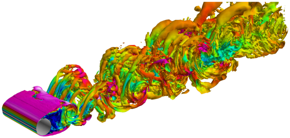


Pedestrian comfort and safety

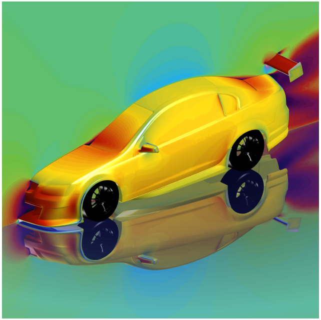


Solar-heated vented facade simulation

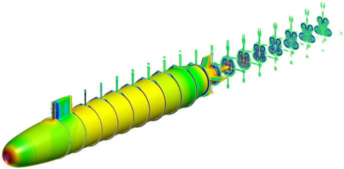




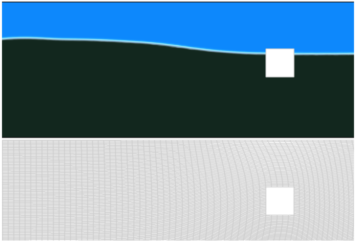
LES simulation



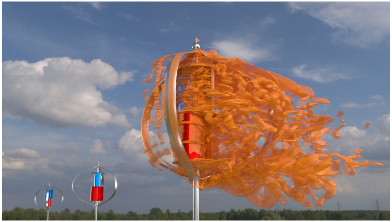
Steady-state RANS



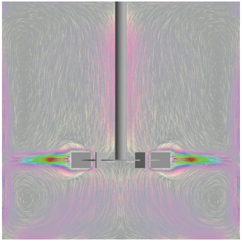
Steady ship resistance and propulsion



Unsteady 6-DoF motions with wave excitation

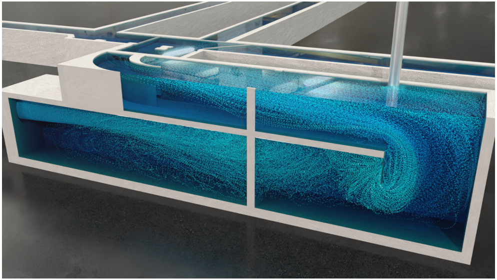


Vertical axis wind turbine development

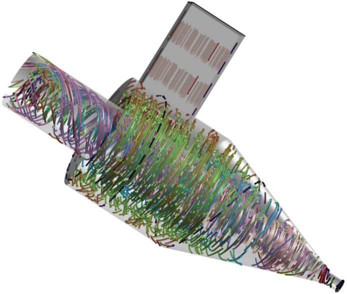


Ruston turbine in a mixed tank

Examples - Multiphase



Ozone contactor tank, multiphase flow mixing



Cyclone seperator

OpenFOAM learning resources

Official sources:

- User guide:
  https://www.openfoam.com/documentation/user-guide
- Tutorial guide:
  https://www.openfoam.com/documentation/tutorial-guide
- Extended Code guide (new in 2023):
  https://doc.openfoam.com/2306/
- Tutorial Wiki: https://wiki.openfoam.com/Tutorials

OpenFOAM learning resources

Unofficial sources:

- OpenFoam Wiki: https://openfoamwiki.net/index.php/Main_Page
- Chalmers University CFD with OpenSource Software course: https://www.tfd.chalmers.se/~hani/kurser/OS_CFD/
- CFD Online forums: https://www.cfd-online.com/Forums/openfoam/
- The OpenFOAM Technology Primer book by Mooney, Maric and Höpken: https://www.researchgate.net/publication/267569764_The_OpenFOAM_Technology_Primer
- The Finite Volume Method in Computational Fluid Dynamics book by Moukalled, Magani and Darwish: https://link.springer.com/book/10.1007/978-3-319-16874-6

Case Structure

- A simulation (case) in OpenFOAM is a hierarchy of directories and text files:

```
<case>
├─ system
│   ├─ controlDict
│   ├─ fvSchemes
│   └─ fvSolution
├─ constant
│   ├─ ...Properties
│   └─ polyMesh
│       ├─ points
│       ├─ cells
│       ├─ faces
│       └─ boundary
└─ time directories
```

Case Structure

- Any solver run will expect to have the following three directories in the case directory:
  - `system`
  - `constant`
  - `time` directories (e. g. `0, 0.1,` etc)
- Each directory contains a series of files and sub-directories that help define the computational problem.

Case Structure

- Computational data is classified either as heavy- or light-weight data.
- Different formats are used for storing different types of data.
- The `system` directory contains light-weight data required to control execution of the code.
- The `constant` directory contains a mixture of light- and heavy-weight data.
- `Time` directories almost exclusively store heavy-weight data.

Dictionaries

- A dictionary is an entity that contains data entries that can be retrieved by the I/O system by means of keywords.
- The general format for dictionary entry:

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

- A sub-dictionary is identified by the name followed by data entries within curly braces:

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

File dictionary

- Each data file must have a description entry:

```
FoamFile
{
    version     2.0;
    format      ascii;
    class       dictionary;
    object      transportProperties;
}
```

Dictionary examples

- An example of a file dictionary, `transportProperties`:

```
transportModel   Newtonian;
nu               1.5e-05;
```

- An example of a sub-dictionary:

```
kEpsilonCoeffs
{
    Cmu          0.09;
    C1           1.44;
    C2           1.92;
    alphaEps     0.76923;
}
```

Dimensions

- OpenFOAM is fully dimensional with dimensions required for each field and physical property
- Dimensions are checked for consistency in all field operations
- Some entries assume dimensions and are not required, such as viscosity.

No	1	2	3	4	5	6	7
Property	Mass	Length	Time	Temperature	Quantity	Current	Luminous Intensity
Unit	Kilogram	meters	second	Kelvin	moles	ampere	candela

- Kinematic viscosity m^2/s would be represented as

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

system directory

- The `system` directory is the repository for files responsible for the control of execution of the solver.
- Files in `system` directory are run-time changeable, the solver reads their contents if they were changed during the run.
- `system` files:
  - `controlDict` is reponsible for the solver execution.
  - `fvSchemes` is reponsible for the specification of the discretization schemes.
  - `fvSolution` is responsible for linear and non-linear solver controls.

controlDict

- `controlDict` file contains entries that are used to control run time behaviour of the solver.
- In general, there are several sections in `controlDict` file responsible for the control of different aspects of the solver execution:
  - time control
  - data writing
  - data reading
  - run-time loadable functionality

Time controls:

- `startFrom` - controls the start time of the simulation
  - `firstTime` - earliest time step from the set of time directories
  - `startTime` - time specified by the `startTime` keyword entry
  - `latestTime` - most recent time step from the set of time directories
- `startTime` - start time for the simulation with `startFrom startTime`

controlDict

Time controls:

- `stopAt`
  - `endTime` - time specified by the `endTime` keyword entry
  - `writeNow` - stops simulation on completion of current time step and writes data
  - `noWriteNow` - stops simulation on completion of current time step and does not write out data
  - `nextWrite` - stops simulation on completion of next scheduled write time, specified by `writeControl`
- `endTime` - end time for the simulation when `stopAt endTime;` is specified
- `deltaT` - time step of the simulation

Data writing

- `writeControl` - controls the timing of write output to file:
  - `timeStep` - writes data every `writeInterval` time steps
  - `runtime` - writes data every `writeInterval` seconds of simulated time
  - `adjustableRunTime` - Writes data every `writeInterval` seconds of simulated time, adjusting the time steps to coincide with the `writeInterval` if necessary. Most useful in cases with automatic time step adjustment.
  - `cpuTime` - writes data every `writeInterval` seconds of CPU time
  - `clockTime` - writes data out every `writeInterval` seconds of real time

controlDict

- `writeInterval` - integer used in conjunction with `writeControl` described above
- `purgeWrite` - integer representing a limit on the number of time directories that are stored by overwriting time directories on a cyclic basis
- `writeFormat` - specifies the format of the data files.
  - `ascii` - ASCII format, written to `writePrecision` significant figures
  - `binary` - Binary format.
- `writePrecision` - integer used in conjunction with `writeFormat` described above (default = 6)
- `writeCompression` - specifies the compression of the data files
  - `uncompressed` - no compression
  - `compressed` - gzip compression

- `timeFormat` - Choice of format of the naming of the time directories:
  - `fixed` - +/-m.dddddd where number d is set by `timePrecision`
  - `scientific` - +/-m.dddddd+/-xx where number d is set by `timePrecision`
  - `general` - specifies scientific format if the exponent is less than -4 or greater than or equal to that specified by `timePrecision`
- `timePrecision` - integer used in conjunction with `timeFormat` described above (default = 6)

controlDict

Data reading:

- `runTimeModifiable` - yes/no switch for whether dictionaries, e. g. `controlDict`, are re-read by solver at the beginning of each time step
- `libs` - list of additional libraries to be loaded at run-time
- `functions` - list of functions, e.g. probes to be loaded at run-time

- `fvSolution`: controls linear and non-linear solver execution
- Consists of at least two sections: `solvers` and `PISO`, `SIMPLE`, or `PIMPLE`.

- Linear solver controls:

```
solvers
{
    p
    {
        solver            PCG;
        preconditioner    DIC;
        tolerance         1e-06;
        relTol            0;
    };
    U
    {
        solver            PBiCG;
        preconditioner    DILU;
        tolerance         1e-05;
        relTol            0;
    };
}
```

fvSolution

- Non-linear controls:

```
PISO
{
    nCorrectors         2;
    nNonOrthogonalCorrectors 0;
    pRefCell            0;
    pRefValue           0;
}
```

fvSchemes

- `fvSchemes`: controls discretization methods for various terms fields and PDE operators
- All entries are given in the form of dictionaries that are parsed at the run time.
- The following dictionaries are required for a typical flow problem:
  - `ddtSchemes`
  - `gradSchemes`
  - `divSchemes`
  - `laplacianSchemes`
  - `interpolationSchemes`
  - `snGradSchemes`

- Time discretization scheme is defined as:

```
ddtSchemes
{
    default          Euler;
}
```

- Other possibilities are: `CoEuler`, `CrankNicholson`, `Euler`, `SLTS`, `backward`, `bounded`, `localEuler`, `steadyState`

fvSchemes

- Computations of gradients are defined as:

```
gradSchemes
{
    default         Gauss linear;
    grad(p)         Gauss linear;
}
```

- Other possibilities are: `cellLimited`, `cellLimited<Venkatakrishnan>`, `cellLimited<cubic>`, `cellMDLimited`, `edgeCellsLeastSquares`, `faceLimited`, `faceMDLimited`, `leastSquares`, `fourth`

- Divergence discretization is defined as:

```
divSchemes
{
    default         none;
    div(phi,U)      Gauss linear;
}
```

- Find out other possibilities by "asking" the solver:

```
divSchemes
{
    default         none;
    div(phi,U)      Gauss ?;
}
```

- Please observe that only Gauss integration is possible.

fvSchemes

- Surface normal gradient is prescribed as:

```
snGradScheme
{
    default          corrected;
}
```

- Laplacian Schemes prescribed as:

```
laplacianSchemes
{
  default          none;
  laplacian(nuEff,U) Gauss linear corrected;
  laplacian((1|A(U)),p) Gauss linear corrected;
  laplacian(1,p) Gauss linear limited 1;
}
```

Structure of constant directory

- As the directory's name indicates, the `constant` directory is a place for files and directories considered constant during the run.
- While solvers have the ability to change their settings at run time, the `constant` directory is not a subject to that look-up.
- The `constant` directory is a placeholder for the mesh files and material properties data needed for the run.
- In the case of laminar flows, the `transportProperties` file is required to specify the kinematic viscosity.

polyMesh

- Mesh and topological data is stored in the `constant/polyMesh` directory:
  - `boundary`: contains a list of boundary patches with their index offsets.
  - `faces`: contains the list of faces together with points describing the face positively oriented in counter-clock direction.
  - `owner`/`neighbour`: contains the list of faces consecutively numbered from 0 to $N_{faces} - 1$ by their position in the list with index of the neighbour/owner cell.
  - `points`: contains the list of points consecutively numbered by the position in the list together with the coordinate position of the point as a triplet of values.

Time directories

- `0` directory is a special case of a general time directory.
- A general time directory stores the boundary, field and mesh data (case of dynamic mesh) required for post-processing and solver restarts.
- Therefore, field initialization, computed field values and corresponding boundary conditions are located in these directories.
- In order to start the solver run, at least one time directory is required.

Time directories

- In the case of time dependent simulations starting from the time `0`, the time directory is typically called "`/0`" directory containing the corresponding field files required by a given solver.
- Each solver requires different data to be specified on boundaries and within the computational domain.

Types of Boundaries

- OpenFOAM has a concept of primitive and numerical boundary conditions.
  - Primitive boundaries are the actual surface patches where the numerical condition is applied.
  - Numerical boundary conditions assign a field value at the surface patch.
- Several types of primitive boundaries are available in OpenFOAM
  - `Constrained` - Applies a constraint to a field.
  - `Unconstrained` - Assigns field values through a numerical condition.

Types of Boundaries

- Constrained primitive boundaries
  - `symmetry`
  - `cyclic`
  - `empty`
  - `wedge`
  - `processor`
- `symmetry` and `cyclic` boundaries are used to put constraints on fields.
- `empty` used for 2D simulations.
- `wedge` used for axisymmetric simulations.
- `processor` boundary is used for inter-processor communication.

Types of Boundaries

- Unconstrained primitive boundaries
  - `patch`
  - `wall`
- In general, `patch` boundary is commonly used for inlets, outlets, and slip walls.
- `wall` boundary is used for no-slip walls. This boundary condition is not contained in the `patch` boundary condition because specialised modelling options can be used on this boundary condition, such as turbulence.
- Numerical boundary conditions can be one of the following types:
  - fixed value (Dirichlet)
  - specified gradient (Neumann)
  - mixed condition (Robin = Dirichlet + Neumann)

Laminar2 Turbulent (L2T)
Dr Darrin Stephens
Chief Technical Officer
Email: darrin@l2t.au