Numerical Simulation of Effluent Discharges

Applications with OpenFOAM

Abdolmajid Mohammadian,
Hossein Kheirkhah Gildeh, and Xiaohui Yan

First edition published 2023

ISBN: 978-1-032-02048-8 (hbk)
ISBN: 978-1-032-02094-5 (pbk)
ISBN: 978-1-003-18181-1 (ebk)

Chapter 3

An introduction to OpenFOAM

DOI: 10.1201/9781003181811-3

CRC Press
Taylor & Francis Group
Boca Raton London New York

CRC Press is an imprint of the
Taylor & Francis Group, an **informa** business

Chapter 3

An introduction to OpenFOAM

3.1 OpenFOAM solvers for effluent discharge modeling

The Open source Field Operation And Manipulation (OpenFOAM) code is an object-oriented numerical simulation toolbox for continuum mechanics, written in C++ language, released by OpenCFD Ltd (http://www.openfoam.com). It is capable of supporting all typical features of C++ programming: it enables the construction of new types of data specific to the problem to be solved (i.e., a virtual class for turbulence model with virtual functions such as ε, k, μt, etc.), the bundling of data and operations into hierarchical classes preventing accidental corruption (i.e., a base class for storing mesh data and a derived class for accessing them), a natural syntax for user defined classes (i.e., operator overloading), and it easily permits the re-use of code for equivalent operations on different types of data (i.e., templating) (Jasak, 1996; Jasak et al., 2004; Juretic, 2004).

Let us examine in more in detail the characteristics of OpenFOAM that help computational fluid dynamics (CFD) programmers. First and most importantly is that the toolbox implements operator-based implicit and explicit second and fourth-order finite volume (FV) discretization in three-dimensional space and on curved surfaces. Differential operators can be treated like finite volume calculus (FVC) or finite volume method (FVM) operators.

The first approach performs explicit derivatives that return a field while the second method is an implicit derivation that converts the expression into matrix coefficients. The underlying idea is to think about partial differential equations (PDEs) in terms of a sum of single differential operators that can be discretized separately with different discretization schemes. Differential operators within OpenFOAM are defined as follows:

$$fvm :: ddt = \frac{\partial}{\partial t}$$

$$fvm :: d2dt2 = \frac{\partial^2}{\partial t^2}$$

$$fvm :: div = \sum_i \frac{\partial}{\partial x_i}$$

DOI: 10.1201/9781003181811-3

$$fvm :: laplacian = \sum_i \frac{\partial^2}{\partial x_i^2}$$

Building different types of PDEs is now only a matter of combining the same set of basic differential operators in a different way. To give an example of the capability of such a top-level code, consider a standard equation like momentum conservation:

$$\frac{\partial \rho \vec{U}}{\partial t} + \nabla \cdot (\rho \vec{U} \vec{U}) - \nabla \cdot (\mu \nabla \vec{U}) = -\nabla p \tag{3.1}$$

This can be implemented in an astonishingly almost natural language that is ready to be compiled in the C++ source code:

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

This capability lets researchers and engineers concentrate their efforts more on the physics than on programming.

The above example clearly shows that OpenFOAM programmers do not think in terms of cells or faces but in terms of objects (U, rho, phi, etc.) defined as a field of values, no matter the dimension, rank or size, over mesh elements such as points, edges, faces, etc. For example, the velocity field is defined at every cell centroid and boundary-face center, with its given dimensions and the calculated values for each direction, and represented by just a single object U of the class GeometricField.

An important feature allowed by object programming is the dimensional check. The physical quantities of objects are in fact constructed with a reference to their dimensions, and thus, only valid dimensional operations can be performed. Avoiding errors and permitting an easier understanding come directly as a consequence of an easier debug.

OpenFOAM is not really thought of as a ready to use code, but aims to be as flexible as possible in defining new models and solvers in the simplest way. Its strength, in fact, lies in being open, not only in terms of source code but also in its inner structure and hierarchical design, giving the user the opportunity to fully extend its capabilities. Moreover, the possibility of using top-level libraries containing a set of models for the same purpose, which refer to the same interface, guarantees programmers smooth and efficient integration with the built-in functionality.

Most of the selections necessary to set up calculations are done at runtime, meaning that options can change while the code is running.

OpenFOAM consists of a library of efficient CFD-related C++ modules. These features can be combined together to create “Solvers” and “Utilities” which are listed below.

- **OpenFOAM Solvers:**
  - Basic CFD
  - Incompressible Flows
  - Compressible Flows
  - Multiphase Flows
  - DNS and LES
  - Combustion
  - Heat Transfer
  - Electromagnetics
  - Solid Dynamics
  - Finance
- **OpenFOAM Utilities:**
  - Preprocessing
  - The FoamX Case Manager
  - Other Preprocessing Utilities
  - Post-processing
  - The paraFoam Post-processor
  - Third-party Post-processing
  - Other Post-processing Utilities
  - Mesh Processing
  - Mesh Generation
  - Mesh Converts
  - Mesh Manipulation
- **OpenFOAM Libraries:**
  - Model Libraries
  - Turbulence
  - Large-eddy Simulation (LES)
  - Transport Models
  - Thermophysical Models
  - Chemical Kinetics
- **Other Features:**
  - Linear System Solvers
  - ODE System Solvers
  - Parallel Computing
  - Mesh Motion
  - Numerical Method

3.1.1 Model preparation

In the OpenFOAM tool-chain, it is possible to use an entirely open source software (OSS) tool-chain, or a combination of OSS and commercial tools. Figures 3.1 and 3.2 show the simulation process priority and the whole tool-chain of OpenFOAM, respectively.

The aim of this section is to explain how OpenFOAM's existing applications are used in order to simulate the desired positively/negatively buoyant jets.

The preparation of the model is divided into two parts: the first step is to implement the required equations in the basic solver, and the second step involves setting up the case and the input files.

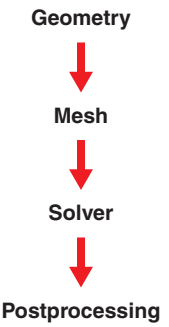

Figure 3.1 Simulation process priority in OpenFOAM.

The present chapter introduces two OpenFOAM solvers for effluent discharge modeling. The first one is based on the standard pisoFoam solver within OpenFOAM and is extended to solve the transport equations. The temperature and concentration evolutions are modeled using the advection-diffusion equation as:

$$\frac{\partial T}{\partial t}+u\frac{\partial T}{\partial x}+v\frac{\partial T}{\partial y}+w\frac{\partial T}{\partial z}=k_{eff}\left(\frac{\partial^2 T}{\partial x^2}+\frac{\partial^2 T}{\partial y^2}+\frac{\partial^2 T}{\partial z^2}\right) \tag{3.2}$$

$$\frac{\partial C}{\partial t}+u\frac{\partial C}{\partial x}+v\frac{\partial C}{\partial y}+w\frac{\partial C}{\partial z}=D\left(\frac{\partial^2 C}{\partial x^2}+\frac{\partial^2 C}{\partial y^2}+\frac{\partial^2 C}{\partial z^2}\right) \tag{3.3}$$

with

$$k_{eff}=\frac{\nu_t}{Pt_r}+\frac{\nu}{P_r} \tag{3.4}$$

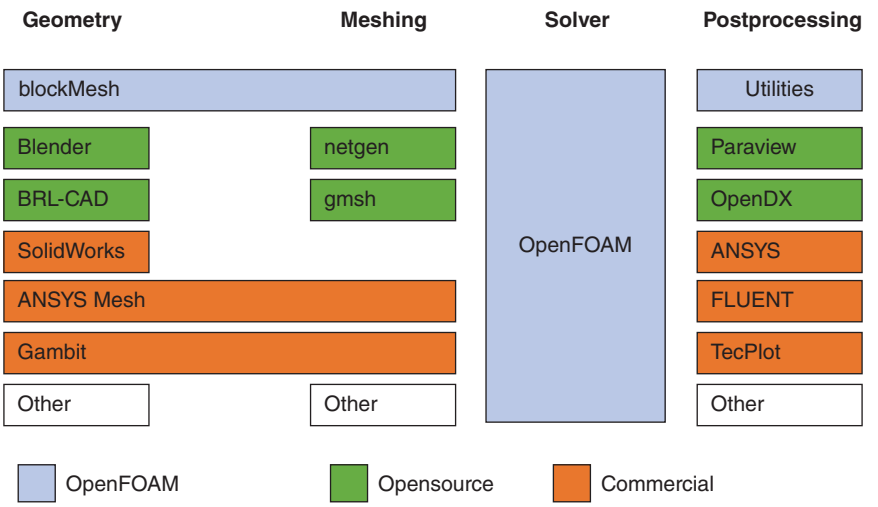

Figure 3.2 OpenFOAM tool-chain.

where T is the fluid temperature, k_{eff} is the heat transfer coefficient, Pt_r is the turbulent Prandtl number, and P_r is the Prandtl number.

The densimetric Froude number Fr is one of the most important parameters in characterizing the flow. The Fr number is the ratio of inertia to buoyancy force, which can be expressed as:

$$Fr = \frac{W_j}{\sqrt{\mathrm{g}' D}} \tag{3.5}$$

with

$$\mathrm{g}' = \mathrm{g}\frac{\rho_a - \rho_j}{\rho_a} \tag{3.6}$$

where W_j is the initial velocity, D is the diameter of the discharge port, g is the gravitational acceleration, ρ_a is the ambient density, and ρ_j is the jet's initial density.

The density difference, which is incorporated in the densimetric Froude number, produces the buoyancy. Liquid wastes discharged from industrial outfalls are categorized into two major classes based on their density. One type is the effluent that has a higher density than the ambient water body. In this case, the discharged effluent has a tendency to sink as a negatively buoyant jet. The second type is the effluent that has a lower density than the ambient water body and is hence defined as a (positively) buoyant jet, which causes the effluent to rise. The density can be calculated based on the equation of state of seawater of Millero and Poisson (1981).

The governing equations are the well-known Navier-Stokes equations for three-dimensional, incompressible fluids as follows:

Continuity equation:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} + \frac{\partial w}{\partial z} = 0 \tag{3.7}$$

Momentum equations:

$$\begin{aligned}\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} + w\frac{\partial u}{\partial z} = &-\frac{1}{\rho}\frac{\partial P}{\partial x} + \frac{\partial}{\partial x}\left(\upsilon_{eff}\left(\frac{\partial u}{\partial x}\right)\right) \\ &+ \frac{\partial}{\partial y}\left(\upsilon_{eff}\left(\frac{\partial u}{\partial y}\right)\right) + \frac{\partial}{\partial z}\left(\upsilon_{eff}\left(\frac{\partial u}{\partial z}\right)\right)\end{aligned} \tag{3.8}$$

$$\begin{aligned}\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} + w\frac{\partial v}{\partial z} = &-\frac{1}{\rho}\frac{\partial P}{\partial y} + \frac{\partial}{\partial x}\left(\upsilon_{eff}\left(\frac{\partial v}{\partial x}\right)\right) + \frac{\partial}{\partial y}\left(\upsilon_{eff}\left(\frac{\partial v}{\partial y}\right)\right) \\ &+ \frac{\partial}{\partial z}\left(\upsilon_{eff}\left(\frac{\partial v}{\partial z}\right)\right) - \mathrm{g}\frac{\rho - \rho_0}{\rho}\end{aligned} \tag{3.9}$$

$$\begin{aligned}\frac{\partial w}{\partial t} + u\frac{\partial w}{\partial x} + v\frac{\partial w}{\partial y} + w\frac{\partial w}{\partial z} = &-\frac{1}{\rho}\frac{\partial P}{\partial z} + \frac{\partial}{\partial x}\left(\upsilon_{eff}\left(\frac{\partial w}{\partial x}\right)\right) \\ &+ \frac{\partial}{\partial y}\left(\upsilon_{eff}\left(\frac{\partial w}{\partial y}\right)\right) + \frac{\partial}{\partial z}\left(\upsilon_{eff}\left(\frac{\partial w}{\partial z}\right)\right)\end{aligned} \tag{3.10}$$

where u, v, and w are the mean velocity components in the x, y, and z direction, respectively, t is the time, P is the fluid pressure, υ_{eff} represents the effective kinematic viscosity ($\upsilon_{eff} = \upsilon t + \upsilon$), υt is the turbulent kinematic viscosity, g is the gravitational acceleration, ρ is the fluid density, and ρ_0 is the reference fluid density.

One should note that the equations are divided by density (ρ), and the buoyancy term is added to the momentum equation in the vertical direction (y-coordinate) to account for variable density effects.

As can be observed from the governing equations, the velocity cannot be calculated until the pressure is solved, and the pressure cannot be computed until the velocity is known. The pisoFoam solver starts by estimating the pressure, and then uses this estimated pressure to calculate an intermediate velocity field and the mass fluxes at the cells' faces. With this information, it then solves the pressure equation. The fluxes are then corrected to satisfy continuity, velocities are corrected on the basis of the new pressure field, and the transport equation is solved using the finite volume method.

The temporal term may be discretized by first-order implicit Euler scheme. The advection and diffusion terms are discretized by the standard finite-volume method using Gaussian integration with a linear interpolation scheme for calculating values at face centers from cell centers. For the pressure field, the preconditioned conjugate gradient (PCG) method is used to solve the linear system. The preconditioned bioconjugate gradient (PBiCG) method may be used for other fields: U, T, C, k, ε, and ω. In order to enhance the rate of convergence for iterative solvers, the diagonal incomplete Cholesky (DIC) preconditioner is used to calculate the pressure field. This is a simplified diagonal-based preconditioner for symmetric matrices. The diagonal incomplete lower upper (DILU) preconditioner is used for the other fields: U, T, C, k, ε, and ω, which mostly include asymmetric matrices to be solved.

Another solver that is popular for discharge modeling is the standard twoLiquidMixingFoam solver in OpenFOAM, which is an incompressible multiphase solver for miscible fluids. The governing equations for this solver can be expressed as:

$$\nabla \cdot U = 0 \tag{3.11}$$

$$\frac{\partial \rho U}{\partial t} + \nabla \cdot (\rho UU) = -\nabla \cdot (p_{rgh}) - gh\nabla\rho + \nabla \cdot (\rho T) \tag{3.12}$$

with

$$\rho = \alpha_1\rho_1 + \alpha_2\rho_2 = \alpha_1\rho_1 + (1-\alpha_1)\rho_2 \tag{3.13}$$

$$T = -\frac{2}{3}\bar{\mu}_{eff}\nabla \cdot \mathrm{UI} + \bar{\mu}_{eff}\nabla U + \bar{\mu}_{eff}(\nabla U)^{\mathrm{T}} \tag{3.14}$$

$$\bar{\mu}_{eff} = \alpha_1(\mu_{eff})_1 + \alpha_2(\mu_{eff})_2 \tag{3.15}$$

$$(\mu_{eff})_{\mathrm{i}} = (\mu - \mu_{\mathrm{t}})_{\mathrm{i}} \tag{3.16}$$

where t is time, U is velocity, ρ is density, p_{rgh} is dynamic pressure, p represents pressure, and $\nabla \cdot (p_{rgh})$ and $gh\nabla\rho$ are obtained by using $P = p_{rgh} + \rho gh$, g is gravitational

acceleration, α is volume fraction, μ is dynamic viscosity, and μt is turbulent viscosity, and subscript i denotes either fluid 1 or 2. In such simulations, fluids 1 and 2 represent the jet and ambient water, respectively.

The alpha diffusion equation is given by:

$$\frac{\partial \alpha_1}{\partial t} + \nabla \cdot (U\alpha_1) = \nabla \cdot \left(\left(D_{ab} + \frac{\nu_t}{S_C} \right) \nabla \alpha_1 \right) \tag{3.17}$$

where D_{ab} is the molecular diffusivity, ν_t is the turbulent eddy viscosity, and S_C is the turbulent Schmidt number.

The continuity, momentum, and alpha diffusion equations can be solved using the finite volume method by using twoLiquidMixingFoam within the OpenFOAM framework. This solver has been widely employed and validated in previous studies (e.g., Gildeh et al., 2021).

3.2 Mesh generation in OpenFOAM

3.2.1 Basic steps of mesh generation in OpenFOAM

Mesh generation may not seem an easy task in OpenFOAM at first glance. It is in fact a common challenge in all CFD models. Creating a suitable mesh that characterizes the geometry of the problem and helps with numerical stability is the first and most important step in building a numerical model. Creating a mesh in OpenFOAM starts with the following steps:

1 Label the coordinate system: draw a three-dimensional Cartesian coordinate system, with the vertical direction as the y-axis.
2 Draw geometric figures: first draw the upper part and then the lower part.
3 Add basic auxiliary lines: it is necessary to ensure that each line runs through; if not, auxiliary lines need to be added.
4 Bold vertices: all intersections are vertices.
5 Determine the blocks: improve the auxiliary line; the numbering starts from 0, such as B0, B1, B2, etc.; check whether the block element is perfect.
6 Coding vertices: all vertices are numbered and must be coded in a reasonable order. The coding involving the order in OpenFOAM is advanced in the order of x, y, and z directions.

An example of a mesh generated by following the above steps is shown in Figure 3.3.

The specific process is as follows:

1 First, taking the above sketch as an example, enter the vertex information in the order of x, y, and z (assume a small value for z such as 0.1).
2 Enter the blocks information and provide the information for the eight vertices. The number of x, y, and z meshes are set to 15, 15, and 1, respectively, and the ratio of adjusting the mesh size is (1 1 1).

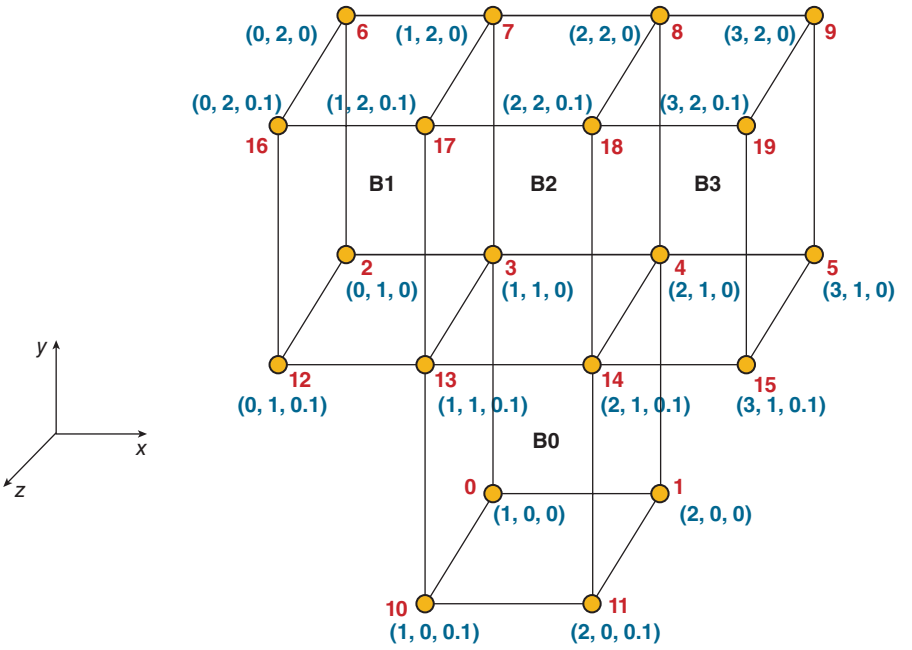

Figure 3.3 Example of input sketch for use in OpenFOAM.

3 Enter the boundary information and name it. The open boundary between the entrance and the exit is represented by patch; the wall is named fixedWalls, which is directly represented by wall; the front and rear surfaces are named frontAndBack, which is represented by empty.

```
Vertices                // enter the vertex information in the order
of x, y, and z.
(
        (1 0 0) //0
        (2 0 0) //1
        (0 1 0) //2
        (1 1 0) //3
        (2 1 0) //4
        (3 1 0) //5
        (0 2 0) //6
        (1 2 0) //7
        (2 2 0) //8
        (3 2 0) //9
        (1 0 0.1) //10
        (2 0 0.1) //11
        (0 1 0.1) //12
        (1 1 0.1) //13
        (2 1 0.1) //14
        (3 1 0.1) //15
        (0 2 0.1) //16
        (1 2 0.1) //17
```

```
       (2 2 0.1) //18
       (3 2 0.1) //19
       );
blocks                          // Enter the blocks information
(
         hex (0 1 4 3 10 11 14 13) (15 15 1) simpleGrading
(1 1 1)
         hex (2 3 7 6 12 13 17 16) (15 15 1) simpleGrading
(1 1 1)
         hex (3 4 8 7 13 14 18 17) (15 15 1) simpleGrading
(1 1 1)
         hex (4 5 9 8 14 15 19 18) (15 15 1) simpleGrading
(1 1 1)
);
edges
(
);
Boundary              // Enter the boundary information and name it

(
        Inlet
        {
             type patch;             // The open boundary between the
entrance and the exit is represented by patch faces
             (
                      (0 1 11 10)
             );
        }
        outlet              //
        {
               type patch;
               faces
               (
                      (6 7 17 16)
                      (7 8 18 17)
                      (8 9 19 18)
                );
        }
        fixedWalls                       // the wall is named
fixedWalls, which is directly represented by wall
                                            faces
        {
               type wall;
               (
                      (0 3 13 10)
                      (2 3 13 12)
                      (2 6 16 12)
                      (1 4 14 11)
                      (4 5 15 14)
                      (5 9 19 15)
               );
        }
```

```
        frontAndBack            // the front and rear surfaces are
named frontAndBack, which is represented by empty.
        {
                type empty;
                faces
                (
                        (0 1 4 3)
                        (2 3 7 6)
                        (3 4 8 7)
                        (4 5 9 8)

                        (10 11 14 13)
                        (12 13 17 16)
                        (13 14 18 17)
                        (14 15 19 18)
                );
        }
);

mergePatchPairs
(
);
```

3.2.2 Common mesh generation methods

Mesh generation is one of the initial tasks that a CFD modeler should complete before solving the problem at hand. The mesh system should capture the geometry and topology of the problem as well as provide the computational cells to solve the mathematical equations. There are different methods in mesh generation in OpenFOAM. The following describes four of them.

1 Cartesian grid generation method
One can approximate curvilinear geometry of complex geometry in Cartesian coordinates by step-by-step approximation. But, this method takes a lot of time and is very tedious to use. Although it is possible to refine the mesh, the stepwise approximation is not smooth and thus leads to large errors, which waste computational resources. Therefore, there are limitations to using CFD methods based on simple coordinate systems (Cartesian or cylindrical), as these systems fail when modeling complex geometries such as airfoil shapes.
2 Structured grid generation method
In this method, elements are aligned in a specific way, or they follow a structured pattern, thus it is called a structured grid. The intersection of coordinate lines identifies the grid points, which then can be arranged in an array.

   The advantage of the structured grid is that the geometric information of each node and the control volume must be stored, but the neighbor relationship of the node can be automatically obtained according to the grid numbering rule. Therefore, the data structure is simple, and it is not necessary to store this kind of information, which is a major advantage of structured grids. In addition, it also has the following advantages: a) the grid generation speed is fast; b) the grid

generation quality is good; c) most of the fitting of the surface or space is obtained by parametric or spline interpolation (i.e., the resulting area is smooth) and it is easier to approach the actual model. It can easily realize the boundary fitting of the region, and is suitable for the calculation of fluid and surface stress concentration.

However, the structured grid also has certain disadvantages: the scope of application is relatively narrow and it is only suitable for graphics with regular shapes.

Structured methods are further divided into single-block and multiblock methods:

1 Structured monolithic grid
A single block is used to cover the entire geometry. It is used for simple shapes, where a single block is sufficient to cover the domain.
2 Structured multiblock grid
In structured meshing, another method used to mesh complex geometry is called the multiblock method. Suppose that in the core of a cylinder, users want to generate a finer mesh or a different pattern than the outer regions; in order to separate these regions, sometimes multiple bounding boxes are used. Similarly, for complex geometries, it is not possible to cover shapes with a single bounding box, so different shapes in the same CFD space generate different bounding boxes, and these blocks can then be combined to form a single structured net grid. Since the mesh generation process involves multiple blocks, it is called a multiblock structured mesh.
3 Unstructured mesh generation method
In this method, all the discrete cells generated in the CFD space are randomly arranged and do not follow any single pattern, as the name "unstructured grid" suggests. Unstructured mesh means that the interior points in the mesh area do not have the same adjacent cells, which can be of various shapes such as tetrahedron, prism, or hexahedron. The number of meshes connected to different interior points within the meshed region varies.

   The advantage of the unstructured mesh generation method is that it has no regular topology and no concept of layers. The distribution of grid nodes is arbitrary and therefore flexible. However, larger memory capacity is required for computation.
4 Hybrid mesh generation method
Hybrid meshes are the most complex meshing processes that are used to study complex geometries. Because the geometry is complex and the outline of the shape cannot be captured by using a single element type, multiple types are available such as prisms, wedges, and hexahedrons. Similarly, for volume meshes of spheres along the surface of a sphere, tetrahedra are used, while hexahedra are used internally. Therefore, the combination of different shapes provides a hybrid mesh.

3.2.3 Parameter definition

There are various parameters in OpenFOAM that are utilized in the blockMeshDict file as described below.

1 Vertices parameter definition
The vertices parameter lists all the point coordinates contained in the block. The point number starts from 0.

2 Blocks parameter definition
The blocks parameter settings include shape flags and divisions. The shape flags are basically hex because the block is generally the sequence number of the 8 points required by the block specified in the first () after the hexahedron hex, and arranged as shown in the block coding sequence diagram. The second () is used to define the number of mesh divisions in the xyz direction. blockMesh supports two division methods simpleGrading and edgeGrading.

simpleGrading () specifies the use of uniform scaling in three directions, and specifies the size of these three proportions respectively. For example, the simpleGrading (2 3 4) setting means that the scale factors along the x, y, and z directions are 2, 3, and 4, respectively.

edgeGrading () gives the complete cell scale for each boundary, from the first voxel to the last voxel direction.

For example, simpleGrading (2 2 2 2 3 3 3 3 4 4 4 4); this setting means that the scale along sides 0~3 is 2, the scale along sides 4~7 is 3, and the scale along sides 8~11 is 4, and the effect is the same as using simpleGrading above.

3 Edges parameter definition
By default, the edge connecting two vertices is straight, but when the edge is a curve, it can be specified by an entry in a list named edges. This list can be ignored if the geometry space does not include curves. In Section 3.2.1, the edge connecting any two points is a straight line, so the list is ignored in the definition. Common boundary types in OpenFOAM are shown in Table 3.1.

Consider the following example:

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

This means that the keyword arc is followed by the label numbers 1 and 5 of the two vertices connected by the edge. When using the arc keyword to specify an arc, a point through the circle where the arc is located must be specified; then the arc passes through the internal point (1.1, 0.0, 0.5).

4 Boundary parameter definition
This section is used to define the name and type of the mesh boundary, as well as a list of vertex numbers for each face (numbering order satisfies the right-hand criterion). The name of the user-defined boundary, and the type of boundary

Table 3.1 Common boundary types in OpenFOAM

Keyword	*Description*	*Parameters to be supplemented*
Arc	Connect using arcs	A point through which the arc passes
simpleSpline	Connect using splines	A series of interior points
polyLine	Connect using a series of straight lines	A series of interior points
polySpline	Connect using a series of splines	A series of interior points
Line[a]	Straight connection	-

Note:
[a] The line keyword was not originally required, but it was added to the keyword list for completeness.

can be specified by the parameter type. The boundary types include: wall (wall), symmetry plane (symmetryPlane), periodic boundary (cyclic), inconsistent periodic boundary (cyclicAMI), two-dimensional boundary axisymmetric boundary (wedge), and 2D boundary (empty).

When using some specific boundary types, attention must be paid to their usage scenarios and associated files. Currently, OpenFOAM only supports 3D grid calculation. If users want to calculate 2D problems, they can set both side walls in the 3D calculation domain as empty boundaries, which are used as 2D calculation boundaries. The faces parameter specifies the order of the points that make up the boundary face, which conforms to the right-hand rule.

5 mergePatchPairs parameter definition

blockMesh allows users to create meshes using multiple blocks. In the case of including multiple blocks, it is necessary to deal with the connection problem between each block. There are two methods for merging as described in the following.

i Face matching

It is required that if block A and block B are to be merged, the vertices of the interface patch A1 and patch B1 are exactly the same. When connecting blocks by face matching, the user does not need to define parameters within mergePatchPairs. blockMesh will automatically match these two patches into inner faces, as shown in the following example:

```
mergePatchPairs
(
);
```

ii Face fusion

There is a mapping relationship between the two patches in two blocks, and the vertices are not required to be exactly the same. The fusion rules are as follows.

- The masterPatch on the main surface remains unchanged, and the coordinates of all points on it remain unchanged.
- If there is a gap between the masterPatch on the main surface and the slavePatch on the secondary surface, project the slavePatch on the secondary surface onto the masterPatch on the main surface to meet the surface fusion requirements.
- Adjust the position of the nodes on the secondary surface through the minimum tolerance value, improve the node matching degree between the primary surface and the secondary surface, and remove the subtle edges smaller than the minimum tolerance.
- When the main and auxiliary surfaces partially overlap, the overlapping part will become an internal surface, and the nonoverlapping part will still be an external surface, and boundary conditions need to be defined.

If the secondary surface slavePatch is fully integrated into the main surface, the secondary surface will be removed. When connecting blocks by face fusion, the two patches to be merged in the mergePatchPairs parameter must be defined, as shown in the following example:

```
mergePatchPairs
(
(<masterPatch> <slavePatch>)
);
```

As seen above, and in general, the grid generation is one of the most challenging tasks in creating a CFD model and OpenFOAM is not an exception. Section 3.5 reviews different utilities in OpenFOAM for mesh generation to solve the discharge mixing problems.

Now that we have discussed available solvers in OpenFOAM and mesh generation and its importance in solving the effluent discharge problems, and before moving to Section 3.4 to discuss the post processing of the results, we will review an example of an effluent discharge model in pisoFoam solver.

3.3 Effluent discharge model preparation in OpenFOAM using pisoFoam solver

3.3.1 PISO algorithm

The governing equations are solved numerically using the finite volume method. The solver, which is used within OpenFOAM, is the modified pisoFoam. This solver is a transient solver for incompressible flow. The code first predicts the velocity field by solving the velocity equation (momentum equations). Although the continuity equation is presented, pisoFoam does not actually solve it; instead, it solves a pressure Poisson equation in PISO (Pressure-Implicit with Splitting of Operators). Rather than solve all of the coupled equations in a coupled or iterative sequential fashion, PISO splits the operators into an implicit predictor and multiple explicit corrector steps. At each time step, velocity, concentration, and temperature are predicted, and then pressure and velocity are corrected. The velocity is predicted implicitly because of the greater stability of implicit methods, which means that a set of coupled linear equations, expressed in matrix-vector form as Ax = b, are solved. It then solves the concentration and temperature equations (Equations 3.2 and 3.3). As in the velocity predictor, the concentration and temperature are predicted implicitly from Equations 3.2 and 3.3.

In order to solve the equations, a new solver, called mypisoFoam, was developed and added to OpenFOAM. Here, the implementation of the momentum equation is shown in mypisoFoam solver as an example.

As previously explained, each term in a PDE is represented in OpenFOAM code using the classes of static functions finiteVolumeMethod and finiteVolumeCalculus, shortened as fvm and fvc, respectively. fvm and fvc contain static functions, representing deferential operators that discretize the terms in the PDE.

Equations, and terms of equations are declared as tmp<Type> where <Type> is either <fvVectorMatrix> if the equation is a vector equation, like the momentum equation, or <fvScalarMatrix> if the equation is a scalar equation, like the ε-equation. The names indicate that the resulting discretized equations are stored as matrices.

The first part of the code for momentum equations (Equations 3.8–3.10) is to introduce a new definition of density that is a function of both temperature and salinity. The density calculation is based on Millero and Poisson (1981) as written in Equations 3.18–3.21 below.

$$\rho = f(S,T) \tag{3.18}$$

Here, the density is calculated for both the jet and the ambient water according to the equation of state of seawater proposed by Millero and Poisson (1981):

$$\rho = \rho_t + AS + BS^{3/2} + CS \tag{3.19}$$

where

$$\begin{aligned} A &= 8.24493\times10^{-1} - 4.0899\times10^{-3}T + 7.6438\times10^{-5}T^2 - 8.2467\times10^{-7}T^3 \\ &\quad + 5.3875\times10^{-9}T^4 \\ B &= -5.72466\times10^{-3} + 1.0227\times10^{-4}T - 1.6546\times10^{-6}T^2 \\ C &= 4.8314\times10^{-4} \end{aligned} \tag{3.20}$$

and ρ_t is the density of water that varies with the temperature as follows:

$$\begin{aligned} \rho_t &= 999.842594 + 6.793952\times10^{-2}T - 9.095290\times10^{-3}T^2 \\ &\quad + 1.001685\times10^{-4}T^3 - 1.120083\times10^{-6}T^4 + 6.536336\times10^{-9}T^5 \end{aligned} \tag{3.21}$$

This has been implemented in the main solver as follows:

```
rho==1000;
rho==(rho/1000*(999.842594+6.793952e-2*T-(9.095290e-3*pow(T,2))+
(1.001685e-4*pow(T,3))-(1.120083e-6*pow(T,4))+(6.536336e-9*pow(T,5))+
S*(8.24493e-1-4.0899e-3*T+(7.6438e-5*pow(T,2))-(8.2467e-7*pow(T,3))+
(5.3875e-9*pow(T,4)))+(pow(S,1.5))*(-5.72466e-3+1.0227e-4*T-1.6546e-6*pow(T,2))+S*(4.8314e-4)));
```

The second part of the UEqn.H implements the LHS of the momentum equations (Equations 3.8–3.10).

```
///
    fvVectorMatrix UEqn
    (
        fvm::ddt(U)
      + fvm::div(phi, U)
      + turbulence->divDevReff(U)
     - g*(rho-1000)/1000

    );
///

            UEqn.relax();
```

And finally, the last part of the algorithm adds the RHS (which is equal to the pressure gradient) and solves the momentum equations.

```
if (momentumPredictor)
{
    solve(UEqn == -fvc::grad(p));
}
```

The PISO algorithm, which is used in the mypisoFoam solver can be understood, for simplicity, by considering a one-dimensional, inviscid flow along the x-direction. So, the momentum equation is simplified to:

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} = -\frac{1}{\rho}\frac{\partial P}{\partial x} \tag{3.22}$$

By using the Euler implicit time stepping with linear interpolation of values to the cell faces and linearization of the convective term by taking the convective velocity from the old time step n, the discretized implicit velocity predictor forms the following equation:

$$\left[\frac{1}{\Delta t}+\left(\frac{u^n_{i+\frac{1}{2}}-u^n_{i-\frac{1}{2}}}{2\Delta x}\right)\right]\Delta V u_i^* + \left(\frac{u^n_{i+\frac{1}{2}}}{2\Delta x}\right)\Delta V u_{i+1}^* - \left(\frac{u^n_{i-\frac{1}{2}}}{2\Delta x}\right)\Delta V u_{i-1}^* = \frac{u_i^n}{\Delta t}\Delta V - \left(\frac{1}{\rho}\frac{\partial P}{\partial x}\right)_i^n \Delta V \tag{3.23}$$

where the predicted values are denoted by *. Notice that pressure is taken from the old time step n since it is yet unknown. Now, the cell volume ΔV can be divided out as follows to get the correct coefficient matrices and vectors that are used in the corrector step:

$$\left[\frac{1}{\Delta t}+\left(\frac{u^n_{i+\frac{1}{2}}-u^n_{i-\frac{1}{2}}}{2\Delta x}\right)\right]u_i^* + \left(\frac{u^n_{i+\frac{1}{2}}}{2\Delta x}\right)u_{i+1}^* - \left(\frac{u^n_{i-\frac{1}{2}}}{2\Delta x}\right)u_{i-1}^* = \frac{u_i^n}{\Delta t} - \left(\frac{1}{\rho}\frac{\partial P}{\partial x}\right)_i^n \tag{3.24}$$

In vector form, this becomes:

$$Cu^* = r - \nabla P^n \tag{3.25}$$

where C is the coefficient array multiplying the solution u^* vector and r is the right-hand side explicit terms. If the viscous and turbulent stress terms are included, they would modify the coefficient matrix C and would not change the general form of the matrix-vector equation. This equation can be changed to:

$$Au^* + H'u^* = r - \nabla P^n \tag{3.26}$$

where A is the diagonal matrix of C and H' is the off-diagonal matrix of A (i.e., $A + H' = C$). Using a matrix solver, the above equation is solved for the predicted velocity u^*.

Moreover, the discretized explicit velocity corrector is written as:

$$\left[\frac{1}{\Delta t}+\left(\frac{u^n_{i+\frac{1}{2}}-u^n_{i-\frac{1}{2}}}{2\Delta x}\right)\right]u_i^{**} + \left(\frac{u^n_{i+\frac{1}{2}}}{2\Delta x}\right)u_{i+1}^* - \left(\frac{u^n_{i-\frac{1}{2}}}{2\Delta x}\right)u_{i-1}^* = \frac{u_i^n}{\Delta t} - \left(\frac{1}{\rho}\frac{\partial P}{\partial x}\right)_i^* \tag{3.27}$$

The first corrected velocity u^{**} is being solved from the predicted velocity u^*, old velocity u^n, and the first corrected pressure P^*. The problem is that the corrected pressure is still unknown. This equation can be expressed in matrix-vector form, as in Equation (3.26):

$$Au^{**} + H'u^* = r - \nabla P^* \tag{3.28}$$

Introducing $H = r - H'u^*$ and inverting A (which is easy since it is diagonal), Equation (3.28) becomes:

$$u^{**} = A^{-1}H - A^{-1}\nabla P^* \tag{3.29}$$

The point of the corrector step is to make the corrected velocity field divergence free so that it adheres to the continuity equation. By applying the divergence to the above equation and recognizing that $\nabla u^{**} = 0$ due to the continuity equation, this yields a Poisson equation for the first corrected pressure:

$$\nabla^2\left(A^{-1}p^*\right) = \nabla\cdot(A^{-1}H) \tag{3.30}$$

When the first corrected pressure p* has been calculated, Equation (3.30) can be solved for the first corrected velocity *u***.

The higher correction steps might be applied using the same A matrix and H vector. The second correction step is also shown below:

$$\nabla^2\left(A^{-1}p^{**}\right) = \nabla\cdot(A^{-1}H) \tag{3.31}$$

$$u^{***} = A^{-1}H - A^{-1}\nabla P^{**} \tag{3.32}$$

where *p*** and *u**** are the second corrected pressure and velocity, respectively. This method works for the other implicit time stepping schemes, for instance Crank-Nicholson or second-order backward. Issa (1985) states that if a second order accurate time stepping scheme is used, then three corrector steps should be used to reduce the discretization error due to PISO algorithm to second-order.

The mypisoFoam solver only solves continuity and momentum equations; hence, we added the advection-diffusion equation for concentration and temperature to this solver and compiled it to use for our case, as explained as follows.

3.3.2 A new solver is born

The PISO loop within the main solver is shown in the following.

```
// --- PISO loop

for (int corr=0; corr<nCorr; corr++)
{
    volScalarField rAU(1.0/UEqn.A());

    U = rAU*UEqn.H();
    phi = (fvc::interpolate(U) & mesh.Sf())
        + fvc::ddtPhiCorr(rAU, U, phi);

    adjustPhi(phi, U, p);

    // Non-orthogonal pressure corrector loop
    for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
    {
        // Pressure corrector

        fvScalarMatrix pEqn
        (
            fvm::laplacian(rAU, p) == fvc::div(phi)
        );

        pEqn.setReference(pRefCell, pRefValue);

        if
        (
            corr == nCorr-1
         && nonOrth == nNonOrthCorr
        )
        {
            pEqn.solve(mesh.solver("pFinal"));
        }
        else
        {
            pEqn.solve();
        }
```

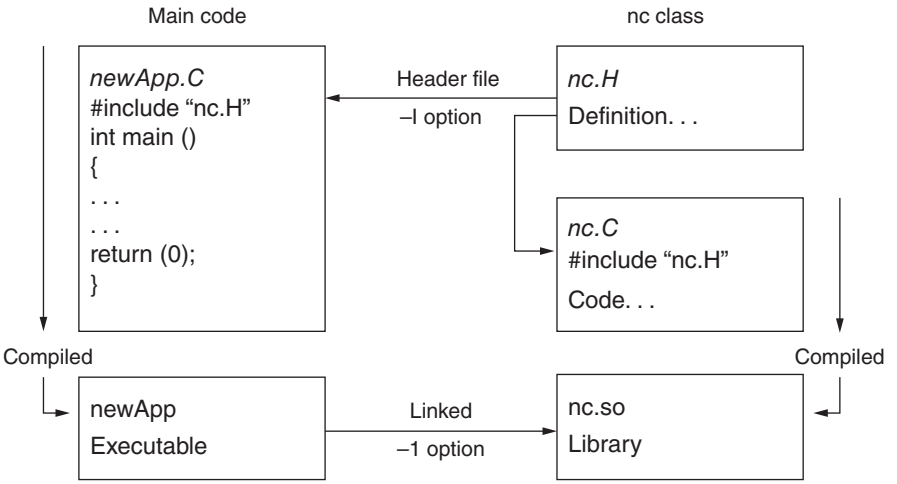

Figure 3.4 Header files, source files, compilation, and linking.

Compilation is an integral part of application development that requires careful management since every piece of code requires its own set of instructions to access dependent components of the OpenFOAM library. In UNIX/Linux systems, these instructions are often organized 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 be used on any code, not only the OpenFOAM library. To understand the compilation process, we first need to explain certain aspects of C++ and its file structure, shown schematically in Figure 3.4. 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 as 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 (mypisoFoam.C in our case), 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.

As a means of checking for 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 compiled.

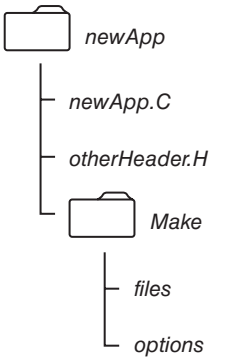

Figure 3.5 Directory structure for an application.

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.

OpenFOAM applications are organized 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 3.5.

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, we may create a new class or add 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.

As explained above, pisoFoam solver includes three dictionaries in its directory:

i Make directory, which includes two dictionaries (files, and options) for calling several header files and to address the compilation folder
ii (ii) createFields.H, which contains the required fields to be solved such as pressure and velocity fields. It was required to introduce concentration and temperature fields here as seen below.

```
Info<< "Reading field S\n" << endl;
volScalarField S
(
    IOobject
    (
        "S",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
);

Info<< "Reading field T\n" << endl;
volScalarField T
(
    IOobject
    (
        "T",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
    ),
    mesh
```

iii pisoFoam.C, which is the main C++ code for the solver that includes the momentum equation and PISO loop. We need to add concentration and temperature equations here as well. These implementations are shown below.

```
    fvScalarMatrix TEqn
    (
        fvm::ddt(rho,T)
      + fvm::div(rhophi, T)
      - fvm::laplacian(kappaEff, T)
    );

    TEqn.relax();
    TEqn.solve();


    fvScalarMatrix SEqn
    (
        fvm::ddt(rho,S)
      + fvm::div(rhophi, S)
      - fvm::laplacian(kappaEff, S)
    );

    SEqn.relax();
    SEqn.solve();
}
```

3.3.3 Preparation of the case file

The basic directory structure for an OpenFOAM case, which contains the minimum set of files required to run an application, is shown in Figure 3.6 and described as follows.

3.3.3.1 The constant directory

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

3.3.3.2 The system directory

This sets parameters associated with the solution procedure itself. It contains at least the following three files: controlDict where run control parameters are set including start/end time, time step, and parameters for data output; fvSchemes where discretization

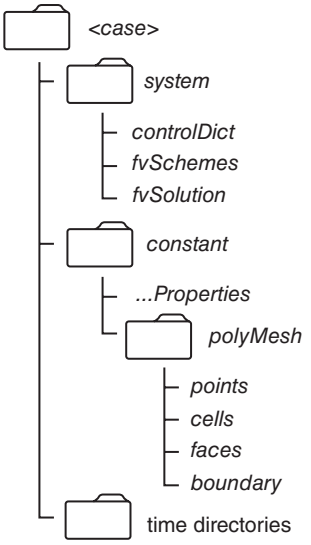

Figure 3.6 Case directory structure.

schemes used in the solution may be selected at run-time; and, fvSolution where the equation solvers, tolerances, and other algorithm controls are set for the run.

3.3.3.3 The "time" directories

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

The implementations and modifications, which have been done, are summarized as follows.

3.3.3.4 Constant directory

In this problem, constant directory includes polyMesh subdirectory where the mesh is built there using the blockMeshDict utility. Boundary conditions are also identified there. The domain is divided into a number of blocks. Each block should have eight vertices. All the vertices for all the blocks are sorted under the vertices dictionary. Each vertex has an x, y, and z coordinate value. The vertices are numbered in order starting from zero for the first vertex.

As mentioned, each block has eight vertices. These vertices are specified by their number according to their appearance in the vertices dictionary. The line for each block starts with the word "hex" meaning that the mesh will be hexahedral followed by the number of the eight vertices connecting the block. After specifying the vertices connecting one block comes the number of grid points in the three directions: x, y, and z.

The patches dictionary specifies the boundary patches in the mesh. Each boundary face should be connected by four vertices. First, the type of the boundary condition

is specified for a patch and then the name of the patch. If a boundary patch is sharing many blocks, an entry for each piece of the patch in each block should be specified under the patch dictionary. The sample files for blockMeshDict and boundary are shown below, respectively.

```
convertToMeters 0.01;

vertices
(
        (0 0 0)//0
        (120 0 0)//1
        (0 0.6405 0)//2
        (120 0.6405 0)//3
        (0 1.5935 0)//4
        (120 1.5935 0)//5
        (0 50 0)//6
        (120 50 0)//7
        (0 0 19.5235)//8
        (120 0 19.5235)//9
        (0 0.6405 19.5235)//10
        (120 0.6405 19.5235)//11
        (0 1.5935 19.5235)//12
        (120 1.5935 19.5235)//13
        (0 50 19.5235)//14
        (120 50 19.5235)//15
        (0 0 20)//16
        (120 0 20)//17
        (0 0.6405 20)//18
        (120 0.6405 20)//19
        (0 1.5935 20)//20
        (120 1.5935 20)//21
        (0 50 20)//22
        (120 50 20)//23
```

```
7
(
    inlet
    {
        type            patch;
        nFaces          8;
        startFace       227375;
    }
    leftWall
    {
        type            wall;
        nFaces          1073;
        startFace       227383;
    }
    outlet
    {
        type            patch;
        nFaces          1081;
        startFace       228456;
    }
    lowerWall
    {
        type            wall;
        nFaces          1656;
        startFace       229537;
    }
    atmosphere
```

The transportProperties dictionary within the constant directory sets the transport model for the fluid to be Newtonian and then provides information about its viscosity and density. This is shown as follows.

```
transportModel  Newtonian;

nu              nu [ 0 2 -1 0 0 0 0 ] 1e-06;
mykappaEff      mykappaEff [ 1 -1 -1 0 0 0 0 ] 1e-03;

CrossPowerLawCoeffs
{
    nu0             nu0 [ 0 2 -1 0 0 0 0 ] 1e-06;
    nuInf           nuInf [ 0 2 -1 0 0 0 0 ] 1e-06;
    m               m [ 0 0 1 0 0 0 0 ] 1;
    n               n [ 0 0 0 0 0 0 0 ] 1;
}

BirdCarreauCoeffs
{
    nu0             nu0 [ 0 2 -1 0 0 0 0 ] 1e-06;
    nuInf           nuInf [ 0 2 -1 0 0 0 0 ] 1e-06;
    k               k [ 0 0 1 0 0 0 0 ] 0;
    n               n [ 0 0 0 0 0 0 0 ] 1;
}


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

Table 3.2 RANS turbulence models for incompressible fluids within OpenFOAM

Model format in OpenFOAM	*Model name*
Laminar	Dummy turbulence model for laminar flow
kEpsilon	Standard high-Re k$-\varepsilon$ model
kOmega	Standard high-Re k$-\omega$ model
kOmegaSST	k$-\omega$-SST model
RNGkEpsilon	RNG k$-\varepsilon$ model
NonlinearKEShih	Nonlinear Shih k$-\varepsilon$ model
LienCubicKE	Lien cubic k$-\varepsilon$ model
qZeta	q$-\zeta$ model
LaunderSharmaKE	Launder-Sharma low-Re k$-\varepsilon$ model
LamBremhorstKE	Lam-Bremhorst low-Re k$-\varepsilon$ model
LienCubicKELowRe	Lien cubic low-Re k$-\varepsilon$ model
LienLeschzinerLowRe	Lien-Leschziner low-Re k$-\varepsilon$ model
LRR	Launder-Reece-Rodi RSTM
LaunderGibsonRSTM	Launder-Gibson RSTM with wall-reflection terms
realizableKE	Realizable k$-\varepsilon$ model
SpalartAllmaras	Spalart-Allmaras one-eqn mixing-length model

Moreover, in the turbulenceProperties the type of turbulence model is mentioned (RANS or LES) and in the other dictionary, the exact turbulence model should be mentioned regarding the turbulence type. OpenFOAM has many different turbulence models, which makes it a popular and strong application for simulations of turbulent flows. It contains various models for both incompressible and compressible fluids. These turbulence models are divided in two well-known categories: RANS (Reynolds Averaged Navier Stokes) and LES (Large Eddy Simulation).

The list of available RANS and LES models in OpenFOAM is presented in Tables 3.2 and 3.3.

3.3.3.5 System directory

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

The terms that must typically be assigned a numerical scheme in fvSchemes range from derivatives, e.g., gradient ∇, and interpolations of values from one set of points to another. The aim in OpenFOAM is to offer an unrestricted choice to the user. For example, while linear interpolation is effective in many cases, OpenFOAM offers complete freedom to choose from a wide selection of interpolation schemes for all interpolation terms.

The derivative terms further exemplify this freedom of choice. The user first has a choice of discretization practice, where standard Gaussian finite volume integration is the common choice. Gaussian integration is based on summing values on cell faces,

Table 3.3 LES turbulence models for incompressible fluids within OpenFOAM

Model format in OpenFOAM	*Model name*
Smagorinsky	Smagorinsky model
Smagorinsky2	Smagorinsky model with 3-D filter
dynSmagorinsky	Dynamic Smagorinsky
homogenousDynSmagorinsky	Homogeneous dynamic Smagorinsky model
dynLagrangian	Lagrangian two equation eddy-viscosity model
scaleSimilarity	Scale similarity model
mixedSmagorinsky	Mixed Smagorinsky/scale similarity model
dynMixedSmagorinsky	Dynamic mixed Smagorinsky/scale similarity model
kOmegaSSTSAS	k−ω-SST scale adaptive simulation (SAS) model
oneEqEddy	k-equation eddy-viscosity model
dynOneEqEddy	Dynamic k-equation eddy-viscosity model
locDynOneEqEddy	Localized dynamic k-equation eddy-viscosity model
spectEddyVisc	Spectral eddy viscosity model
LRDDiffStress	LRR differential stress model
DeardorffDiffStress	Deardorff differential stress model
SpalartAllmaras	Spalart-Allmaras model
SpalartAllmarasDDES	Spalart-Allmaras delayed detached eddy simulation (DDES) model
SpalartAllmarasIDDES	Spalart-Allmaras improved DDES (IDDES) model

which must be interpolated from cell centers. The user again has a completely free choice of interpolation scheme, with certain schemes being specifically designed for particular derivative terms, especially the convection divergence ∇. terms.

The set of terms, for which numerical schemes must be specified, are subdivided within the fvSchemes dictionary into the categories listed in Table 3.4. Each keyword in Table 3.4 is the name of a subdictionary which contains terms of a particular type, e.g., gradSchemes contains all the gradient derivative terms such as grad(p) (which represents ∇p). Further examples can be seen in the extract from an fvSchemes dictionary as listed in Table 3.4.

Table 3.4 Main keywords used in fvSchemes

Keyword	*Category of mathematical terms*
interpolationSchemes	Point-to-point interpolations of values
snGradSchemes	Component of gradient normal to a cell face
gradSchemes	Gradient ∇
divSchemes	Divergence ∇.
laplacianSchemes	Laplacian ∇^2
timeScheme	First and second time derivatives $\frac{\partial}{\partial t}$, $\frac{\partial^2}{\partial t^2}$
fluxRequired	Fields which require the generation of a flux

```
ddtSchemes
{
    default         CrankNicholson 0.5;
}

gradSchemes
{
    default         Gauss linear;
    grad(p)         Gauss linear;
    grad(U)         Gauss linear;

}

divSchemes
{
    default         none;

    div(phi,U)      Gauss cubic corrected;
    div(phi,S)      Gauss cubic corrected;
    div(phi,T)      Gauss cubic corrected;
    div(rhophi,S)       Gauss limitedLinear 1;
    div(rhophi,T)       Gauss limitedLinear 1;
    div(phi,k)      Gauss limitedLinear 1;
    div(phi,epsilon) Gauss limitedLinear 1;
    div(phi,R)      Gauss limitedLinear 1;
    div(R)          Gauss linear;
    div(phi,nuTilda) Gauss limitedLinear 1;
    div((nuEff*dev(T(grad(U))))) Gauss linear;
    div(nonlinearStress) Gauss linear;

}

laplacianSchemes
```

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

```
solvers
{
    p
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0.1;
    }

    pFinal
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;
    }

    U
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0;
    }
//
    S
    {
        solver          PBiCG;
        preconditioner  DILU;
        tolerance       1e-05;
        relTol          0;
    }
    T
    {
        solver          PBiCG;
```

fvSolution contains a set of subdictionaries that are specific to the solver being run. However, there is a small set of standard subdictionaries that cover most of those used by the standard solvers. These subdictionaries include solvers, relaxationFactors, PISO, and SIMPLE algorithms (PISO has already been explained).

3.3.3.6 Linear solver control

The first subdictionary in our example, and one that appears in all solver applications, is solver. It specifies each linear-solver that is used for each discretized equation; it is

Table 3.5 Linear solvers in OpenFOAM

Solver	*Keyword*
Preconditioned (bi-)conjugate gradient	PCG/PBiCG[a]
Solver using a smoother	smoothSolver
Generalized geometric-algebraic multigrid	GAMG
Diagonal solver for explicit systems	diagonal

Note:
[a] PCG for symmetric matrices, PBiCG for asymmetric.

emphasized that the term linear-solver refers to the method of number-crunching used to solve the set of linear equations, as opposed to application solver, which describes the set of equations and algorithms used to solve a particular problem.

The syntax for each entry within solvers uses a keyword that is the word relating to the variable being solved in the particular equation. For example, pisoFoam solves equations for velocity U and pressure P, hence the entries for U and p. The keyword is followed by a dictionary containing the type of solver and the parameters that the solver uses. The solver is selected through the solver keyword from the choices in OpenFOAM, listed in Table 3.5. The parameters, including tolerance, and preconditioner are described in the following sections.

In the current study, for pressure field, PCG (Preconditioned Conjugate Gradient) is used for each discretized equation. PCG is a linear solver, the same as PBiCG (Preconditioned Bio Conjugate Gradient), which has been used for the other remaining fields, U, T, C, k, ε, and ω.

3.3.3.7 Solution tolerances

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

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

- the residual falls below the solver tolerance, designated as tolerance.
- the ratio of current to initial residuals falls below the solver relative tolerance, relTol.
- the number of iterations exceeds a maximum number of iterations, maxIter.

3.3.3.8 Preconditioned conjugate gradient solvers

There are a range of options for preconditioning of matrices in the conjugate gradient solvers, represented by the preconditioner keyword in the solver dictionary. The preconditioners are listed in Table 3.6.

Table 3.6 Preconditioner options in OpenFOAM

Preconditioner	*Keyword*
Diagonal incomplete-Cholesky (symmetric)	DIC
Faster diagonal incomplete-Cholesky (DIC with caching)	FDIC
Diagonal incomplete-LU (asymmetric)	DILU
Diagonal	Diagonal
Geometric-algebraic multigrid	GAMG
No preconditioning	None

In numerical analysis and linear algebra, a preconditioner M of a matrix A is a matrix such that M-1A has a smaller condition number than A. Preconditioners are useful in iterative methods to solve linear system $Ax = b$ for x since the rate of convergence for most iterative solvers increases as the condition number of a matrix decreases as a result of preconditioning. In this example, DIC (Diagonal Incomplete Cholesky) preconditioner can be used for the pressure field. This is a simplified diagonal-based preconditioner for the symmetric matrices. However, DILU (Diagonal Incomplete LU) preconditioner is used for the other fields which mostly include asymmetric matrices to be solved.

3.3.3.9 Time control

The OpenFOAM solvers begin all runs by setting up a database. The database controls I/O and, since output of data is usually requested at intervals of time during the run, time is an inextricable part of the database. The controlDict dictionary sets input parameters essential for the creation of the database. A sample controlDict dictionary is shown in the following. Only the time control and writeInterval entries are truly compulsory.

```
application     pisoFoamIIII;
startFrom       startTime;
startTime       0;
stopAt          endTime;
endTime         90;
deltaT          0.001;
writeControl    adjustableRunTime;
writeInterval   1;
purgeWrite      0;
writeFormat     ascii;
writePrecision  6;
writeCompression off;
timeFormat      general;
timePrecision   6;
```

3.4 Postprocessing with ParaView

ParaView is an open-source platform based on the VTK (visualization toolkit) function library. VTK uses the Pipeline (pipeline) operating mechanism, which can process almost any type of data, and provides many corresponding classes to manipulate various types of data. The steps for VTK to visualize volume data are shown in Figure 3.7.

OpenFOAM can be installed in the Windows Subsystem for Linux (WSL), but a very troublesome problem is that WSL does not have a graphical interface, only a console window. During the post-processing process, users cannot directly call ParaView via the paraFoam command to start ParaView to post-process the calculation results of OpenFOAM. The essence of paraFoam is actually a script that calls ParaView. Users can rewrite this script and install ParaView under windows to post-process the calculation results of OpenFOAM. Once the results are written to the time directory, they can be viewed with ParaView. If a case is running with ParaView open, the output data on the time directory is not automatically loaded. At this point, clicking Apply in the Properties window will download the data to ParaView.

3.4.1 Isosurfaces and contour plots

When using ParaView post-processing calculation results to draw contours/surfaces, only scalars can be selected. If users want to draw contours/surfaces of vector components, such as the contours of velocity *ux*, what should they do?

Take the results from any simulation to make the contour of the velocity U_X. When contouring the profile, only the scalar p can be selected after contour, so if the velocity component can be extracted, it can be selected here. Delete contour1 first, and with slice1 selected, apply the Extract Component filter, set the properties, select u for the input array, x for the component, and name the output array *u-x*.

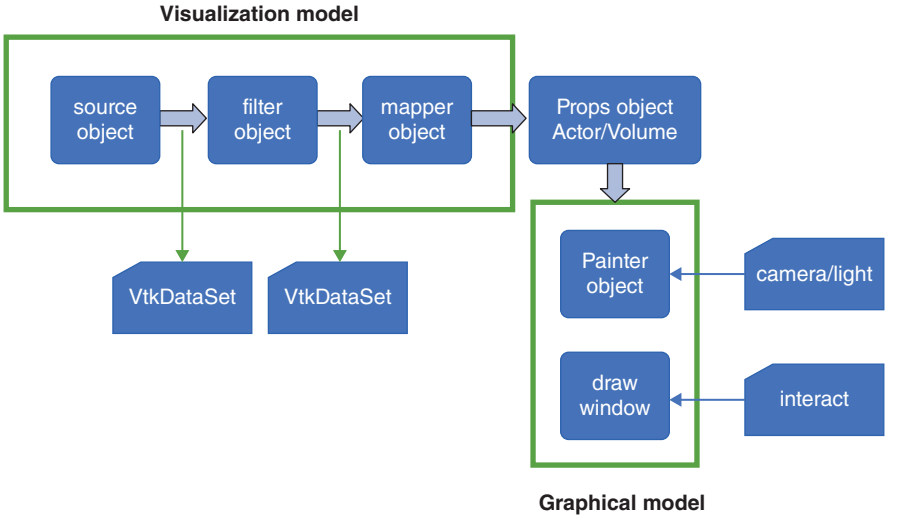

Figure 3.7 VTK visualization method.

In order to draw a simple pressure graph, in the Display panel, select the following: select the Color panel and rescale from color to data range via the menu. In the Style panel, choose Surface from the Representation menu; in the Color panel, choose p and Rescale to Data Range.

To view the solution for a moment, the user can change the current time to the target moment using the VCR Controls or the Current Time Controls located in the toolbar below the top menu of the ParaView window. Click the icon to get a continuous display, then click the icon to get a single pressure value in a cell; in this case, each cell will be represented by a single color with no grading.

If the model itself is 3D, the contours will be a series of 2D surfaces representing constant values, i.e., isosurfaces. The Contour's Properties panel includes a list of isosurfaces that can be edited, most conveniently using the New Range window, where the selected scale field is selected from the drop-down menu.

Color bars can be included by clicking the Toggle Color Legend Visibility button in the Active Variable Controls toolbar or by selecting Show Color Legend from the View menu.

To set a series of properties of the color bar, including text size, font selection, and numbering format of the ticks, the user should click the Edit Color Map button in the Active Variable Controls toolbar or the Color panel in the Display window. The color bar can be dragged with the mouse. Position in the image window.

If users use a newer version of ParaView, they will see that ParaView uses a blue to white to red color scale by default instead of the more common blue to green to red (rainbow)

To change the color stop, the user can do the following: select a selection preset in the color stop editor, then choose a blue to red rainbow. If the user wants to use this color bar all the time, after clicking the "OK" confirmation button, the user can click the "Set Default" button. At this point, the color of the geometric surface is changed by the user, and the result of the change can be seen by rotating the image.

3.4.2 Vector plots

It is best to delete other created modules before the flow velocity vector, there are two ways: to delete completely, highlight the relevant module in the Pipeline Browser, and click Delete in the respective Properties panel; and by toggling the eye button of the relevant module in the Pipeline Browser to disable.

1 At the beginning, ensure that the OpenFOAM file is active, i.e., the small eye icon in front of the file is bright, then select filter->Alphabetical (the last option)->cell centers, and then click Apply.
2 At this time, the small eyes in front of the cell center are active. Then perform the following operations: filter——>Alphabetical——>Glyph.
3 Set: In the orient option in the Object Inspector, because it is a Vector, select OFF in scale Mode, and fill in an appropriate value in the set scale factor, then try to observe the effect.
4 Coloring, if there is no color, you can set it in display
5 Users can try to change the parameters in ARROW to make the picture more visually appealing.

The speed can be represented by arrows in ParaView. To make a simple vector diagram, do the following: open the prepared file, find the generated file, the user can select all grid elements and all variables to apply, the default display is its pressure information, the user can enter u to view its speed information, for quality problems users can choose Glyph, which can make a vector plot, which is to use arrows to represent the speed and size of the water flow, click Glyph, click Apply, and click play to view the animation. ParaView's default arrow size changes with pressure, but users may want more arrow sizes with velocity u. Users can select u in the scale array to view the speed information, and click Apply after conversion.

The user can choose to play the animation from the beginning. Click the scale factor to change the arrow size. When users do not want to change the size of the arrow, but only want to know the direction of the flow velocity, they can select the unscaled array from the scale array, click Apply, and click Play to view the animation, which will clearly show the direction of the flow velocity.

To add an arrow, select Stream Tracer, add Glyph (upper left corner) on it, change the properties of Glyph: Glyph Type to Cone (cone); (arrow shape is cone) Orientation Array to u (speed); Scale Array to For u; (set the arrow size according to the speed) Scale Factor reset (reset button) (adjust the overall arrow size). Colored by temperature, with Glyph still selected in the pipeline window, change the coloring to Temp. This way, the size of the arrow represents the magnitude of the velocity, and the color of the arrow represents the temperature.

Note that although the glyph is represented as flowing through the wall, its size is zero because ParaView chooses to orient the glyph in the x-direction when the glyph shrinks and the velocity size is 0.

3.4.3 Streamline plots

The user should disable modules, such as those for the vector illustration above, before continuing post-processing in ParaView.

Now, in order to draw velocity streamlines, keep the cavity.OpenFOAM module highlighted in the Pipeline Browser to generate high quality streamline images: first, the user should select Stream Tracer from the Filter menu, then click Apply, and second, set according to the specific model Parameters, specify the Seed points (origin) along the Line Source, which runs perpendicular to the geometric center. Click Apply to generate the trajectory, and then select the Tube from the Filter to generate a high-quality streamline diagram. Then click Apply.

1 Obtain a 3D streamline diagram
  Proceed as follows:
    After opening the chosen model, select Filters -> Stream Tracer
    Without changing any parameters, some streamline diagrams in 3D can be obtained.
  Modify the Seeds setting parameters:
  a Using the point Source setting:
    1 Select point Source
    2 Determine the position/coordinates of the point
    3 Determine the number of points n

4 Determine the distribution radius of points, and randomly distribute n points in the spherical space with this value as the radius

b Use the line Source setting:

1 Select high resolution line Source
2 Determine the positions/coordinates of the two ends of the line
3 Determine the resolution n

The number of points thus obtained is $n + 1$ (including both endpoints)

Tips:

The user can also click and hold the point to be moved with the left mouse button. While pressing the left mouse button, moving the mouse can also change the position of the point, and releasing the left mouse button can determine the position of the point.

Usually, the default is 3D display. In this 3D mode, the user can scroll the middle button of the mouse to zoom. Press the middle button and drag the mouse to pan the model. If you hold the left button of the mouse and drag, you can rotate the graphic. There are also a series of tabs; the user can click these buttons to lock the viewing angle in a specific orientation of some coordinate values. For example, to observe the model from the positive direction of the y-axis, click the button below.

However, sometimes the results of some examples are two-dimensional, and the corresponding velocity is also two-dimensional, with only two components in the x and y directions. After loading the results in ParaView, if the user wants to draw the streamline, they will find that the streamline drawing button Stream Tracer is grayed out and the function is unavailable. In this case, how should the user draw the streamline of the 2D velocity vector?

First of all, it needs to be clear that the reason why Stream TracerQ is unavailable is that ParaView can only recognize 3D vectors, and many functions only work on 3D vectors. Therefore, 2D vectors need to increase the component in the z direction. The Calculator filter can be used to create new 3D velocity vectors.

2 Cross-sectional streamline diagram

The Stream Tracer of ParaView cannot directly make a streamline on a section. For example, for the OpenFOAM example, even if it is a two-dimensional example, after intercepting a surface, the streamline cannot be obtained with Stream Tracer. There is a solution. The following describes how to obtain streamlines on the section through a combination of a series of filters. Taking the pitzdaily example, the steps are as follows:

a Make a section (slice)

b Use the Surface Vector filter on the section; the purpose of this is to project the velocity vector onto the plane.

c Use the Mask Points filter on the obtained SurfaceVector. The purpose of this is to generate a series of reference points. When drawing the streamline in the future, the position and density of the streamline will be determined by the position of these reference points.

The On Ratio parameter controls the density of the points; Maximum number of points controls the total number of points; Random Sampling, enables the random point mode, if it is a nonrandom mode, the points will be selected according to the coordinates from small to large. Assuming that On Ratio = 2560, there are a total of 1000 points, but if the Maximum number of points

is set to 100, then only the first 100 points with the smallest coordinates will be taken, and if the random mode is turned on, the distribution of points is basically uniform fill for this flow area. Generate Vertices, select whether to display reference points; if enabled, a dot matrix will be displayed.

d Select Stream Tracer with Custom Source, Input, and Seed Source in Filter The streamline diagram of the section is obtained. The density of the streamline can be controlled by the number of Mask Points.

3.4.4 Two ways for ParaView to create animation

First save the image sequence in ParaView. Click file>save Animation to save image sequences in tif, png, jpg, and other formats. The number of frames to be saved can be specified; the default is one frame per timestep. Note: If there are many pictures, it is best to create a new folder. Secondly, there are two ways to convert the picture sequence to video or GIF.

The first way: use the third-party post-processing tool with OpenFOAM, type in the terminal under the example folder. The specific usage of foamCreateVideo can be viewed with foamCreateVideo -help.

The second way: use the convert/usr/bin/convert -quality 90% frames*.png movie.gif in the Ubuntu system. Obviously, -quality 90% means the video quality, where frames are the image sequence name, and * means any name that follows, such as frames.000.png, movie is the name of the generated gif.

References

Gildeh, H.K., Mohammadian, A., and Nistor, I. Inclined dense effluent discharge modelling in shallow waters. *J. Environ. Fluid Mech.* 2021, 21, 955–998. doi:10.1007/s10652-021-09805-6.

Issa, R.I. Solution of the implicitly discretized fluid flow equations by operator-splitting. *J. Comput. Phys.* 1985, 62, 40–65.

Jasak, H. *Error analysis and estimation for the finite volume method with applications to fluid flows*. Ph.D. Thesis, Imperial College of Science, Technology and Medicine, 1996.

Jasak, H., Weller, H., and Nordin, N. In cylinder CFD simulation using a C++ object-oriented toolkit. *SAE Technical Papers*, 2004.

Juretic, F. *Error analysis in finite volume*. Ph.D. thesis, Imperial College of Science, Technology and Medicine, 2004.

Millero, F.J. and Poisson, A. International one-atmosphere equation of state of sea water. *J. Deep-Sea Res.* 1981, 28A(6), 625–629.