Description and utilization of `interFoam` multiphase solver

1 General description of the OpenFOAM suite

"OpenFOAM (Open Field Operation and Manipulation) is primarily a C++ toolbox for the customisation and extension of numerical solvers for continuum mechanics problems, including computational fluid dynamics (CFD). It comes with a growing collection of pre-written solvers applicable to a wide range of problems. It is produced by UK company OpenCFD Ltd. and is released open source under the GPL.

Original development started in the late 1980s at Imperial College, London, motivated by a desire to find a more powerful and flexible general simulation platform than the defacto standard at the time, Fortran. Since then it has evolved by exploiting the latest advanced features of the C++ language, having been effectively re-written several times over. The predecessor, FOAM, was sold by UK company Nabla Ltd. before being released open source in 2004.

OpenFOAM has been pioneering in a number of ways:

- Amongst the first major scientific packages written in C++ (other leading CFD companies have released or are working on next-generation C++ codes),
- Use of C++ operator overloading to permit relatively simple, top-level human-readable descriptions of partial differential equations makes OpenFOAM akin to a programming language for physical simulation,
- First major general-purpose CFD package to use polyhedral cells. This functionality is a natural consequence of the hierarchical description of simulation objects,
- First and most capable general purpose CFD package to be released under an open-source license,

OpenFOAM compares favourably with the capabilities of most leading general-purpose commercial closed-source CFD packages. It relies on the user's choice of third party pre- and post-processing utilities, and ships with:

- a plugin (paraFoam) for visualisation of solution data and meshes in ParaView.
- a wide range of mesh converters allowing import from a number of leading commercial packages
- an automatic hexahedral mesher to mesh engineering configurations

OpenFOAM was conceived as a continuum mechanics platform but is ideal for building multi-physics simulations. For example, it comes with a library and solvers for efficiently tracking particles in a multiphase flow using the lagrangian approach.

Standard Solvers include:

- Basic CFD
- Incompressible flows
- Compressible flows
- Multiphase flows
- DNS and LES
- Combustion
- Heat transfer
- Electromagnetics
- Solid dynamics
- Finance

Apart from the standard solvers, one of the distinguishing features of OpenFOAM is its relative ease in creating custom solver applications. OpenFoam allows the user to use syntax that closely resemble the partial differential equations being solved"[1].

2 Introduction to Volume of Fluid Theory. Surface reconstruction strategies

2.1 Finite Volume Method

Respect of Finite Volume Method, among all the bibliography, books from Versteeg and Malalasekera [4] and Ferziger and Peric [3] are the most referenced and authoritative. Furthermore Jasak's PhD thesis [5] provides an excellent description of the method using compact and powerful vector and tensorial notation suitable for comprehension and implementation especially in the OpenFOAM's frame.

Following is an annotated version of chapter 3 of Jasak's PhD thesis. Original text is between quotes and annotations are added as footnotes.

2.1.1 Introduction

"The purpose of any discretisation practice is to transform one or more partial differential equations into a corresponding system of algebraic equations. The solution of this system produces a set of values which correspond to the solution of the original equations at some pre-determined locations in space and time, provided certain conditions, to be defined later, are satisfied. The discretisation process can be divided into two steps: the discretisation of the solution domain and equation discretisation ([[6], [5] ref. 97]).

The discretisation of the solution domain produces a numerical description of the computational domain, including the positions of points in which the solution is sought and the description of the boundary. The space is divided into a finite number of discrete regions, called control volumes or cells. For transient simulations, the time interval is also split into a finite number of time-steps. Equation discretisation gives an appropriate transformation

of terms of governing equations into algebraic expressions.

This [section] presents the Finite Volume method (FVM) of discretisation, with the following properties:

- The method is based on discretising the integral form of governing equations over each control volume. The basic quantities, such as mass and momentum, will therefore be conserved at the discrete level.

- Equations are solved in a fixed Cartesian coordinate system on the mesh that does not change in time. The method is applicable to both steady-state and transient calculations.

- The control volumes can be of a general polyhedral shape, with a variable number of neighbours, thus creating an arbitrarily unstructured mesh. All dependent variables share the same control volumes, which is usually called the colocated or non-staggered variable arrangement ([See [5] refs. 117 and 109]).

- Systems of partial differential equations are treated in the segregated way ([See [5] refs. 107 and 137]), meaning that they are solved one at a time, with the inter-equation coupling treated in the explicit manner. Non-linear differential equations are linearised before the discretisation and the non-linear terms are lagged.

[. . .]

2.1.2 Discretisation of the Solution Domain

Discretisation of the solution domain produces a computational mesh on which the governing equations are subsequently solved. It also determines the positions of points in space and time where the solution is sought. The procedure can be split into two parts: discretisation of time and space.

Since time is a parabolic coordinate ([7]), the solution is obtained by marching in time from the prescribed initial condition. For the discretisation of time, it is therefore sufficient to prescribe the size of the time-step that will be used during the calculation.

The discretisation of space for the Finite Volume method used in this study requires a subdivision of the domain into control volumes (CV). Control volumes donot overlap and completely fill the computational domain. In the present study all variables share the same CV-s.

A typical control volume is shown in Fig. 1. The computational point P is located at the

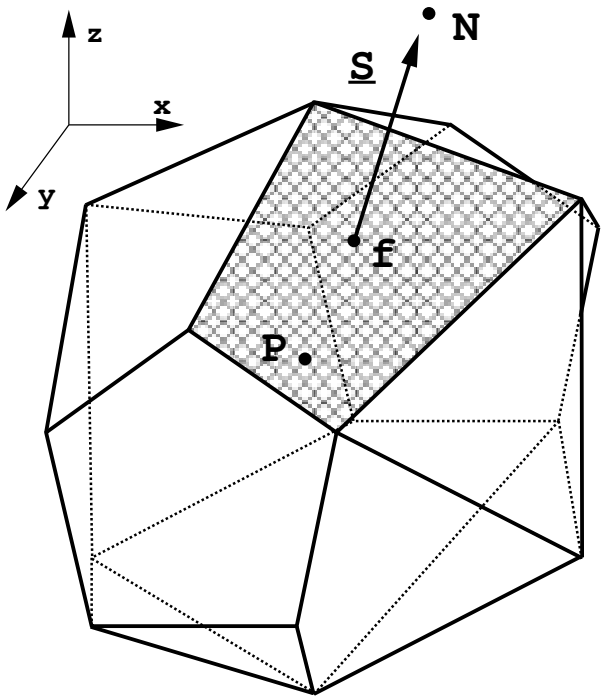


Figure 1: Control volume

centroid of the control volumes[1], such that:

$$\int_{V_P} (\mathbf{x} - \mathbf{x}_P)\, dV = 0 \tag{1}$$

The control volume is bounded by a set of flat faces and each face is shared with only one neighbouring CV. The topology of the control volume is not important – it is a general polyhedron.

The cell faces in the mesh can be divided into two groups – internal faces (between two control volumes) and boundary faces, which coincide with the boundaries of the domain. The face area vector $\mathbf{S}_f$ is constructed for each face in such a way that it points outwards from the cell with the lower label, is normal to the face and has the magnitude equal to the area of the face. The cell with the lower label is called the 'owner' of the face – its label is stored in the 'owner' array. The label of the other cell ('neighbour') is stored in the 'neighbour' array. Boundary face area vectors point outwards from the computational domain – boundary faces

[1] By definition centroid position implies:

$$V_P\, \mathbf{x}_P = \int_{V_P} \mathbf{x}\, dV$$

so grouping at right hand side

$$0 = \int_{V_P} \mathbf{x}\, dV - \mathbf{x}_P \int_{V_P} dV$$

and due $\mathbf{x}_P$ is constant

$$0 = \int_{V_P} (\mathbf{x} - \mathbf{x}_P)\, dV$$

are 'owned' by the adjacent cells. For the shaded face in Fig. 1, the owner and neighbour cell centres are marked with P and N , as the face area vector $\mathbf{S}_f$ points outwards from the P control volume. For simplicity, all faces of the control volume will be marked with f , which also represents the point in the middle of the face (see Fig. 1).

The capability of the discretisation practice to deal with arbitrary control volumes gives considerable freedom in mesh generation. It is particularly useful in meshing complex geometrical configurations in three spatial dimensions. Arbitrarily unstructured meshes also interact well with the concept of local grid refinement, where the computational points are added in the parts of the domain where high resolution is necessary, without disturbing the rest of the mesh.

2.1.3 Discretisation of the Transport Equation

The standard form of the transport equation for a scalar property ϕ is:

$$\underbrace{\frac{\partial \rho\phi}{\partial t}}_{\text{temporal derivative}} + \underbrace{\nabla\cdot\left(\rho\mathbf{U}\phi\right)}_{\text{convection term}} - \underbrace{\nabla\cdot\left(\rho\Gamma_\phi\nabla\phi\right)}_{\text{diffusion term}} = \underbrace{S_\phi\left(\phi\right)}_{\text{source term}} \tag{2}$$

This is a second-order equation, as the diffusion term includes the second derivative of ϕ in space. For good accuracy, it is necessary for the order of the discretisation to be equal to or higher than the order of the equation that is being discretised. The discretisation practice adopted in this study is second-order accurate in space and time and will be presented in the rest of this [section]. The individual terms of the transport equation will be treated separately.

In certain parts of the discretisation it is necessary to relax the accuracy requirement, either to accommodate for the irregularities in the mesh structure or to preserve the boundedness of the solution . Any deviation from the prescribed order of accuracy creates a discretisation error, which is of the order of other terms in the original equation and dissappers only in the limit of excessively fine mesh. Particular attention will therefore be paid to the sources of discretisation error representing such behaviour.

The accuracy of the discretisation method depends on the assumed variation of the function $\phi = \phi\left(\mathbf{x}, t\right)$ in space and time around the point P . In order to obtain a second-order accurate method, this variation must be linear in both space and time, *i.e.* it is assumed that:

$$\phi\left(\mathbf{x}\right) = \phi_P + \left(\mathbf{x} - \mathbf{x}_P\right)\cdot\left(\nabla\phi\right)_P \tag{3}$$

$$\phi\left(t + \Delta t\right) = \phi^t + \Delta t\left(\frac{\partial\phi}{\partial t}\right)^t \tag{4}$$

where

$$\phi_P = \phi\left(\mathbf{x}_P\right) \tag{5}$$

$$\phi^t = \phi\left(t\right) \tag{6}$$

Let us consider the Taylor series expansion in space of a function around the point $\mathbf{x}$:

$$\phi(\mathbf{x}) = \phi_P + (\mathbf{x}-\mathbf{x}_P)\bullet\left(\nabla\phi\right)_P + \frac{1}{2}(\mathbf{x}-\mathbf{x}_P)^2 : \left(\nabla\nabla\phi\right)_P + \frac{1}{3!}(\mathbf{x}-\mathbf{x}_P)^3 :: \left(\nabla\nabla\nabla\phi\right)_P + \ldots + \frac{1}{n!}(\mathbf{x}-\mathbf{x}_P)^n \underbrace{:::}_{\text{n}} \left(\underbrace{\nabla\nabla\ldots\nabla}_{\text{n}}\phi\right)_P + \ldots \tag{7}$$

The expression $(\mathbf{x}-\mathbf{x}_P)^n$ in Eqn. (7) and consequent equations in this study represents the `nth` tensorial product of the vector $(\mathbf{x}-\mathbf{x}_P)$ with itself, producing an `nth` rank tensor. The operator ':::' is the inner product of two `nth` rank tensors, creating a scalar.

Comparison between the assumed variation, Eqn. (3), and the Taylor series expansion, Eqn. (7), shows that the first term of the truncation error scales with $\left|(\mathbf{x}-\mathbf{x}_P)^2\right|$, which is for a 1-D situation equal to the square of the size of the control volume. The assumed spatial variation is therefore second-order accurate in space. An equivalent analysis shows that the truncation error in Eqn. (4) scales with Δt^2, resulting in the second-order temporal accuracy.

The Finite Volume method requires that Eqn. (2) is satisfied over the control volume V_P around the point P in the integral form:

$$\int_t^{t+\Delta t}\left[\frac{\partial}{\partial t}\int_{V_P}\rho\phi\,dV + \int_{V_P}\nabla\bullet\left(\rho\mathbf{U}\phi\right)dV - \int_{V_P}\nabla\bullet\left(\rho\Gamma_\phi\nabla\phi\right)dV\right]dt = \int_t^{t+\Delta t}\left(\int_{V_P}S_\phi\left(\phi\right)dV\right)dt \tag{8}$$

The discretisation of Eqn. (8) will now be examined term by term.

2.1.4 Discretisation of Spatial Terms

Let us first examine the discretisation of spatial terms. The generalised form of Gauss' theorem will be used throughout the discretisation procedure, involving these identities:

$$\int_V \nabla\bullet\mathbf{a}\,dV = \oint_{\partial V}\mathbf{a}\bullet d\mathbf{S} \tag{9}$$

$$\int_V \nabla\phi\,dV = \oint_{\partial V}\phi\,d\mathbf{S} \tag{10}$$

$$\int_V \nabla\mathbf{a}\,dV = \oint_{\partial V}\mathbf{a}\otimes d\mathbf{S} \tag{11}$$

where ∂V is the closed surface bounding the volume V and dS represents an infinitesimal surface element with associated outward pointing normal on ∂V.

A series of volume and surface integrals needs to be evaluated. Taking into account the prescribed variation of ϕ over the control volume P , Eqn. (3), it follows:

$$\begin{aligned}\int_{V_P} \phi\,(\mathbf{x})\; dV &= \int_{V_P} \left[\phi_P + (\mathbf{x}-\mathbf{x}_P)\bullet\left(\nabla\phi\right)_P\right] dV \\ &= \phi_P \int_{V_P} dV + \left[\int_{V_P} (\mathbf{x}-\mathbf{x}_P)\; dV\right]\bullet\left(\nabla\phi\right)_P \\ &= \phi_P\, V_P \end{aligned} \tag{12}$$

where V_P is the volume of the cell. The second integral in Eqn. (3) is equal to zero because the point P is the centroid of the control volume.

Let us now consider the terms under the divergence operator. Having in mind that the CV is bounded by a series of flat faces, Eqn. (9) can be transformed into a sum of integrals over all faces:

$$\begin{aligned}\int_{V_P} \nabla\bullet\mathbf{a} dV &= \oint_{\partial V_P} \mathbf{a}\bullet d\mathbf{S} \\ &= \sum_f \left(\int_f \mathbf{a}\bullet d\mathbf{S}\right)\end{aligned} \tag{13}$$

The assumption of linear variation of ϕ leads to the following expression for the face integral in Eqn. (13)[2]:

$$\begin{aligned}\int_f \mathbf{a} d\mathbf{S} &= \left(\int_f d\mathbf{S}\right)\bullet\mathbf{a}_f + \left[\int_f \left(\mathbf{x}-\mathbf{x}_f\right) d\mathbf{S}\right] : (\nabla\mathbf{a})_f \\ &= \mathbf{S}\bullet\mathbf{a}_f\end{aligned} \tag{14}$$

Combining Eqs. (12, 13 and 14), a second-order accurate discretised form of the Gauss' theorem is obtained[3]:

$$(\nabla\bullet\mathbf{a})\, V_P = \sum_f \mathbf{S}\bullet\mathbf{a}_f \tag{15}$$

Here, the subscript f implies the value of the variable (in this case, $\mathbf{a}$) in the middle of the face and $\mathbf{S}$ is the outward-pointing face area vector. In the current mesh structure, the face

[2] In the first term of right hand side, expresion $\mathbf{a}_f$ is taken off the integral symbol due it is constant, it is the value at the centroid of the face.

[3] Starting with Gauss' theorem for a continuum we have:

$$\int_V \nabla\bullet\mathbf{a}\, dV = \oint_{\partial V} \mathbf{a}\bullet d\mathbf{S}$$

applying equation 12 to left hand side term and 14 to right hand side term:

$$\nabla\bullet\mathbf{a}\, V_P = \sum_f \mathbf{S}\bullet\mathbf{a}_f$$

another useful relation is

$$\nabla\bullet\mathbf{a} = \frac{\sum_f \mathbf{S}\bullet\mathbf{a}_f}{V_P}$$

area vector $\mathbf{S}_f$ point outwards from P only if f is 'owned' by P . For the 'neighbouring' faces $\mathbf{S}_f$ points inwards, which needs to be taken into account in the sum in Eqn. (15). The sum over the faces is therefore split into sums over 'owned' and 'neighbouring' faces:

$$\sum_f \mathbf{S}\cdot\mathbf{a}_f = \sum_{\text{owner}} \mathbf{S}_f\cdot\mathbf{a}_f - \sum_{\text{neighbour}} \mathbf{S}_f\cdot\mathbf{a}_f \tag{16}$$

This is true for every summation over the faces. In the rest of the text, this split is automatically assumed.

Convection term The discretisation of the convection term is obtained using Eqn. (15):

$$\begin{aligned}\int_{V_P} \nabla\cdot\left(\rho\mathbf{U}\phi\right) dV &= \sum_f \mathbf{S}\cdot\left(\rho\mathbf{U}\phi\right)_f \\ &= \sum_f \mathbf{S}\cdot\left(\rho\mathbf{U}\right)_f \phi_f \\ &= \sum_f F\,\phi_f \end{aligned} \tag{17}$$

where F in Eqn. (17) represents the mass flux through the face:

$$F = \mathbf{S}\cdot\left(\rho\mathbf{U}\right)_f \tag{18}$$

The calculation of these face fluxes will later be discussed separately in Section 2.1.8. For now it can be assumed that the flux is calculated from the interpolated values of ρ and U[4].

Eqs. (17 and 18) also require the face value of the variable ϕ calculated from the values in the cell centres, which is obtained using the convection differencing scheme.

Before we continue with the formulation of the convection differencing scheme, it is necessary to examine the physical properties of the convection term. Irrespective of the distribution of the velocity in the domain, the convection term does not violate the bounds of ϕ given by its initial distribution. If, for example, ϕ initially varies between 0 and 1, the convection term will never produce the values of ϕ that are lower than zero or higher than unity. Considering the importance of boundedness in the transport of scalar properties of interest [. . .][5], it is essential to preserve this property in the discretised form of the term.

Convection Differencing Scheme The role of the convection differencing scheme is to determine the value of ϕ on the face from the values in the cell centres. In the framework of arbitrarily unstructured meshes, it would be impractical to use any values other than ϕ_P and ϕ_N, because of the storage overhead associated with the additional addressing information. We shall therefore limit ourselves to differencing schemes using only the nearest neighbours of the control volume.

[4]These are interpolated values at the faces.
[5]See Section 1.2.1 of Jasak's thesis

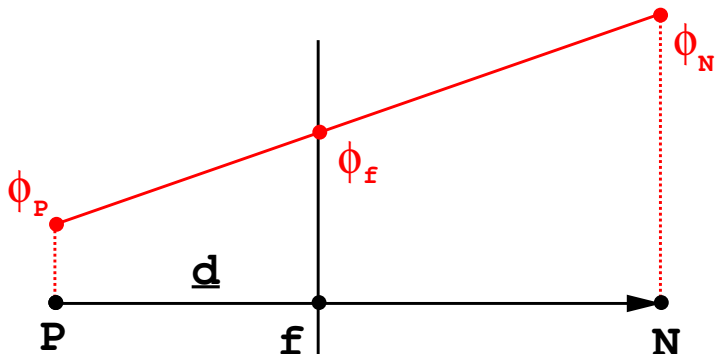


Figure 2: Face interpolation

Assuming the linear variation of ϕ between P and N , Fig. 2, the face value is calculated according to:

$$\phi_f = f_x \phi_P + (1 - f_x)\, \phi_N \tag{19}$$

Here, the interpolation factor f_x is defined as the ratio of distances $\overline{fN}$ and $\overline{PN}$:

$$f_x = \frac{\overline{fN}}{\overline{PN}} \tag{20}$$

The differencing scheme using Eqn. (19) to determine the face value of ϕ is called **Central Differencing** (CD). Although this has been the subject of some debate, Ferziger and Peric [3] show that it is second order accurate even on non-uniform meshes. This is consistent with the overall accuracy of the method. It has been noted, however, that CD causes unphysical oscillations in the solution for convection-dominated problems ([7], [6]), thus violating the boundedness of the solution.

An alternative discretisation scheme that guarantees boundedness is **Upwind Differencing** (UD). The face value of ϕ is determined according to the direction of the flow:

$$\phi_f = \begin{cases} \phi_f = \phi_P & \text{for } F \geq 0 \\ \\ \phi_f = \phi_N & \text{for } F < 0 \end{cases} \tag{21}$$

Boundedness of the solution is guaranteed through the sufficient boundedness criterion for systems of algebraic equations (see *e.g.* [7]). [. . .], boundedness of UD is effectively insured at the expense of the accuracy, by implicitly introducing the numerical diffusion term[6]. This term violates the order of accuracy of the discretisation and can severely distort the solution.

Blended Differencing (BD) ([See [5] ref. 109]) represents an attempt to preserve both boundedness and accuracy of the solution. It is a linear combination of UD, Eqn. (19) and CD, Eqn. (19):

$$\phi_f = (1 - \gamma)\left(\phi_f\right)_{UD} + \gamma\left(\phi_f\right)_{CD} \tag{22}$$

[6]See Jasak's thesis section 3.6

or

$$\begin{aligned}\phi_f = &\left[(1-\gamma)\max\left(\mathrm{sgn}\left(F\right),0\right)+\gamma f_x\right]\phi_P \\ &+\left[(1-\gamma)\min\left(\mathrm{sgn}\left(F\right),0\right)+\gamma\left(1-f_x\right)\right]\phi_N\end{aligned} \tag{23}$$

The blending factor γ, $0 \leq \gamma \leq 1$, determines how much numerical diffusion will be introduced. Peric ([See [5] ref. 109]) proposes a constant γ for all faces of the mesh. For $\gamma = 0$ the scheme reduces to UD.

Many other attempts to find an acceptable compromise between accuracy and boundedness have been made [...][7]. The most promising approach at this stage combines a higher-order scheme with Upwind Differencing on a face-by-face basis, based on different boundedness criteria[8].

Diffusion term The diffusion term will be discretised in a similar way. Using the assumption of linear variation of ϕ and Eqn. 15), it follows:

$$\begin{aligned}\int_{V_P} \nabla\cdot\left(\rho\Gamma_\phi\nabla\phi\right)dV &= \sum_f \mathbf{S}\cdot\left(\rho\Gamma_\phi\nabla\phi\right)_f \\ &= \sum_f \left(\rho\Gamma_\phi\right)_f \mathbf{S}\left(\nabla\phi\right)_f\end{aligned} \tag{24}$$

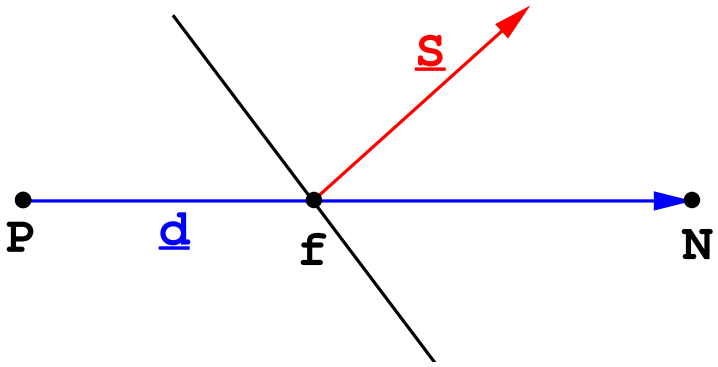


Figure 3: Vector **d** and **S** on a non-orthogonal mesh

If the mesh is orthogonal, i.e. vectors **d** and **S** in Fig. 3 are parallel, it is possible to use the following expression:

$$\mathbf{S}\cdot\left(\nabla\phi\right)_f = |\mathbf{S}|\frac{\phi_N-\phi_P}{|\mathbf{d}|} \tag{25}$$

Using Eqn. (25), the face gradient of ϕ can be calculated from the two values around the face. An alternative would be to calculate the cell-centred gradient for the two cells sharing

[7] See Jasak's thesis Section 1.2.1
[8] See Jasak's thesis Section 3.4

the face as[9]:

$$\left(\nabla\phi\right)_P = \frac{1}{V_P}\sum_f \mathbf{S}\phi_f \tag{26}$$

interpolate it to the face:

$$\left(\nabla\phi\right)_f = f_x\left(\nabla\phi\right)_P + (1-f_x)\left(\nabla\phi\right)_N \tag{27}$$

and dot it with $\mathbf{S}$. Although both of the above-described methods are second-order accurate, Eqn. (27) uses a larger computational molecule[10]. The first term of the truncation error is now four times larger than in the first method, which in turn cannot be used on non-orthogonal meshes.

Unfortunately, mesh orthogonality is more an exception than a rule. In order to make use of the higher accuracy of Eqn. (25), the product $\mathbf{S}\bullet\left(\nabla\phi\right)_f$ is split into two parts:

$$\mathbf{S}\bullet\left(\nabla\phi\right)_f = \underbrace{\mathbf{\Delta}\bullet\left(\nabla\phi\right)_f}_{\text{orthogonal contribution}} + \underbrace{\mathbf{k}\bullet\left(\nabla\phi\right)_f}_{\text{non-orthogonal correction}} \tag{28}$$

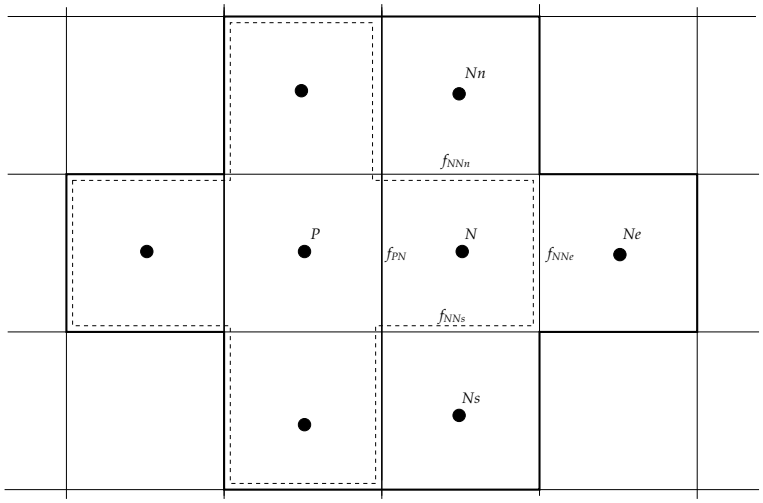
[9]By Gauss' theorem

$$\int_{VP}\nabla\phi\,dV = \oint_{\partial V}\phi\,d\mathbf{S}$$

$$\nabla\phi\,V_P = \sum_f \mathbf{S}\phi$$

$$\nabla\phi = \frac{1}{V_P}\sum_f \mathbf{S}\phi$$

[10]Taking in account that to calculate gradient by 27 equation it is necessary to know all face values in cell P and N, it implies interpolate this values using second neighbours.
According Eqn. 27 for gradient calculation is necessary to do sum in Eqn. 26 for cell N, this sum uses $\phi_{f_{PN}}$, $\phi_{f_{NNn}}$, $\phi_{f_{NNe}}$, $\phi_{f_{NNs}}$ values. Last three of them requires second neighbours for its calculation, for example to calculate $\phi_{f_{NNe}}$, it is neccesary to use ϕ_N and ϕ_{Ne} values.



The two vectors introduced in Eqn. (28), $\mathbf{\Delta}$ and $\mathbf{k}$, have got to satisfy the following condition:

$$\mathbf{S} = \mathbf{\Delta} + \mathbf{k} \tag{29}$$

Vector $\mathbf{\Delta}$ is chosen to be parallel with $\mathbf{d}$. This allows us to use Eqn. (25) on the orthogonal contribution, limiting the less accurate method only to the non-orthogonal part which cannot be treated in any other way.

Many possible decompositions exist and we will examine three:

- **Minimum correction approach**. The decomposition of $\mathbf{S}$, Fig. 4, is done in such a way to keep the non-orthogonal correction in Eqn. (28) as small as possible, by making $\mathbf{\Delta}$ and $\mathbf{k}$ orthogonal:

$$\mathbf{\Delta} = \frac{\mathbf{d} \cdot \mathbf{S}}{\mathbf{d} \cdot \mathbf{d}} \mathbf{d} \tag{30}$$

  with $\mathbf{k}$ calculated from Eqn. (29). As the non-orthogonality increases, the contribution from ϕ_P and ϕ_N decreases.

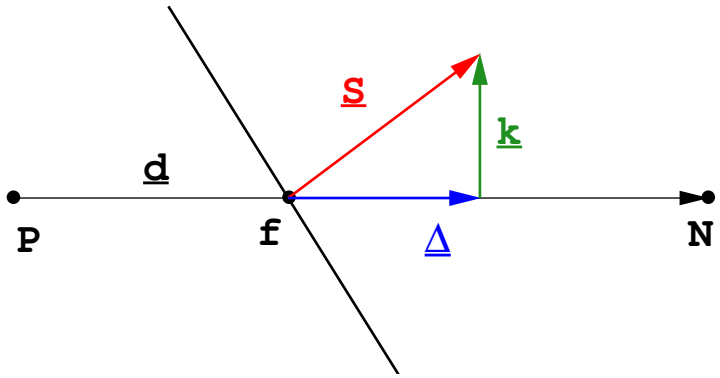


Figure 4: Non-orthogonality treatment in the 'minimum correction' approach

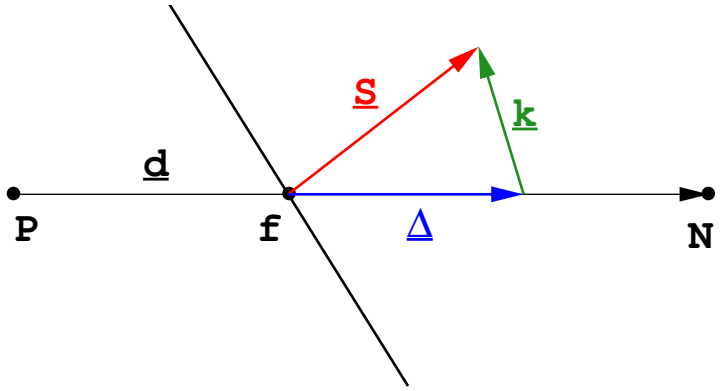


Figure 5: Non-orthogonality treatment in the 'orthogonal correction' approach

- **Orthogonal correction approach**. This approach keeps the contribution from ϕ_P and ϕ_N the same as on the orthogonal mesh irrespective of the non-orthogonality, Fig. 5. To achieve this we define:

$$\mathbf{\Delta} = \frac{\mathbf{d}}{|\mathbf{d}|}|\mathbf{S}| \tag{31}$$

- **Over-relaxed approach**. In this approach, the importance of the term in ϕ_P and ϕ_N is caused to increase with the increase in non-orthogonality:

$$\mathbf{\Delta} = \frac{\mathbf{d}}{\mathbf{d}\cdot\mathbf{S}}|\mathbf{S}|^2 \tag{32}$$

The decomposition of the face area vector is shown in Fig. 6.

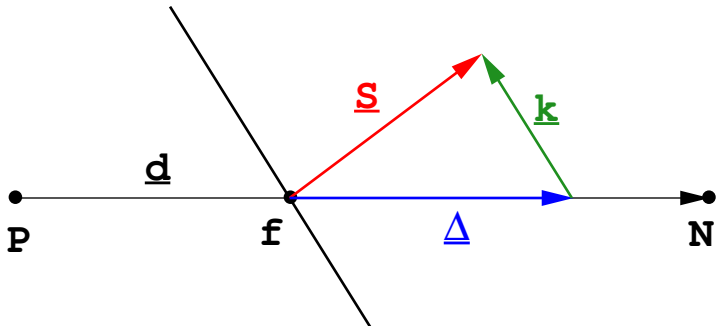


Figure 6: Non-orthogonality treatment in the 'over-relaxed' approach

The diffusion term, Eqn. (24), in its differential form exhibits the bounded behaviour. Its discretised form will preserve this property only on orthogonal meshes. The non-orthogonal correction potentially creates unboundedness, particularly if mesh non-orthogonality is high. If the preservation of boundedness is more important than accuracy, the non-orthogonal correction has got to be limited or completely discarded, thus violating the order of accuracy of the discretisation [...][11].

All of the approaches described above are valid – Eqn. (29) is satisfied for all of them. The difference occurs in their accuracy and stability on non-orthogonal meshes [...][12].

The final form of the discretised diffusion term is the same for all three approaches. The orthogonal part of Eqn. (28) is discretised in the following way: since $\mathbf{d}$ and $\mathbf{\Delta}$ are parallel, it follows that:

$$\mathbf{\Delta}\cdot\left(\nabla\phi\right)_f = |\mathbf{\Delta}|\frac{\phi_N - \phi_P}{|\mathbf{d}|} \tag{33}$$

and Eqn. (28) can be written as:

$$\mathbf{S}\cdot\left(\nabla\phi\right)_f = |\mathbf{\Delta}|\frac{\phi_N - \phi_P}{|\mathbf{d}|} + \mathbf{k}\cdot\left(\nabla\phi\right)_f \tag{34}$$

[11] See Jasak's thesis section 3.6

[12] See Jasak's thesis Sections 3.6 and 3.7.4.

The face interpolate of $\nabla\phi$ is calculated using Eqn. (27)[13]

Source Terms All terms of the original equation that cannot be written as convection, diffusion or temporal terms are treated as sources. The source term, $S_\phi\left(\phi\right)$, can be a general function of ϕ. When deciding on the form of the discretisation for the source, its interaction with other terms in the equation and its influence on boundedness and accuracy should be examined. Some general comments on the treatment of source terms are given in Patankar [7]. A simple procedure will be explained here.

Before the actual discretisation, the source term needs to be linearised:

$$S_\phi\left(\phi\right) = Su + Sp\,\phi \tag{36}$$

where Su and Sp can also depend on ϕ. Following Eqn. (3.12), the volume integral is calculated as:

$$\int_{V_P} S_\phi\left(\phi\right)\,dV = Su\,V_P + Sp\,V_P\,\phi_P \tag{37}$$

The importance of the linearisation becomes clear in implicit calculations. It is advisable to treat the source term as 'implicitly' as possible. This will be further explained in Section 2.1.7.

2.1.5 Temporal Discretisation

In the previous Section, the discretisation of spatial terms has been presented. This can be split into two parts – the transformation of surface and volume integrals into discrete sums and expressions that give the face values of the variable as a function of cell values. Let us again consider the integral form of the transport equation, Eqn. (8):

$$\int_t^{t+\Delta t}\left[\frac{\partial}{\partial t}\int_{V_P}\rho\phi\,dV + \int_{V_P}\nabla\bullet\left(\rho\mathbf{U}\phi\right)\,dV - \int_{V_P}\nabla\bullet\left(\rho\Gamma_\phi\nabla\phi\right)\,dV\right]dt = \int_t^{t+\Delta t}\left(\int_{V_P}S_\phi\left(\phi\right)\,dV\right)dt$$

Using Eqs. (17, 34 and 37), and assuming that the control volumes do notchange in time, Eqn. (8) cn be written as:

$$\int_t^{t+\Delta t}\left[\left(\frac{\partial\rho\phi}{\partial t}\right)_P V_P + \sum_f F\phi_f - \sum_f\left(\rho\Gamma_\phi\right)_f\mathbf{S}\bullet\left(\nabla\phi\right)_f\right]dt = \int_t^{t+\Delta t}\left(Su\,V_P + Sp\,V_P\,\phi_P\right)dt \tag{38}$$

[13] As is explained later (see Section 2.1.7) the face interpolate $\left(\nabla\phi\right)_f$ is calculated explicitely by ϕ values at previous time. Using Rusche's Thesis notation [see [12] Eq. (2.28)] Eq. (34) reads:

$$\mathbf{S}\bullet\left(\nabla\phi\right)_f = |\mathbf{\Delta}|\,\frac{\phi_N - \phi_P}{|\mathbf{d}|} + \mathbf{k}\bullet\left(\nabla\phi^0\right)_f \tag{35}$$

The above expression is usually called the 'semi-discretised' form of the transport equation ([6]).

Having in mind the prescribed variation of the function in time, Eqn. (4), the temporal integrals and the time derivative can be calculated directly as:

$$\left(\frac{\partial \rho\phi}{\partial t}\right)_P = \frac{\rho_P^n \phi_P^n - \rho_P^0 \phi_P^0}{\Delta t} \tag{39}$$

$$\int_t^{t+\Delta t} \phi\left(t\right) = \frac{1}{2}\left(\phi^0 + \phi^n\right)\Delta t \tag{40}$$

where

$$\begin{aligned} \phi^n &= \phi\left(t + \Delta t\right) \\ \phi^0 &= \phi\left(t\right) \end{aligned} \tag{41}$$

Assuming that the density and diffusivity do not change in time, Eqs. (38, 39 and 40) give:

$$\begin{aligned} \frac{\rho_P \phi_P^n - \rho_P \phi_P^0}{\Delta t} V_P + \frac{1}{2}\sum_f F\phi_f^n - \frac{1}{2}\left(\rho\Gamma_\phi\right)_f \mathbf{S}\bullet\left(\nabla\phi\right)_f^n \\ + \frac{1}{2}\sum_f F\phi_f^0 - \frac{1}{2}\left(\rho\Gamma_\phi\right)_f \mathbf{S}\bullet\left(\nabla\phi\right)_f^0 \\ = Su\,Vp + \frac{1}{2}Sp\,Vp\phi_P^n + \frac{1}{2}Sp\,Vp\phi_P^0 \end{aligned} \tag{42}$$

This form of temporal discretisation is called the **Crank-Nicholson** method. It is second-order accurate in time. It requires the face values of ϕ and $\nabla\phi$ as well as the cell values for both old and new time-level. The face values are calculated from the cell values on each side of the face, using the appropriate differencing scheme for the convection term, and Eqn. (34) for diffusion. The evaluation of the non-orthogonal correction term will be discussed later (see Section 2.1.7). Our task is to determine the new value of ϕ_P. Since ϕ_f and $\left(\nabla\phi\right)_f$ also depend on values of ϕ in the surrounding cells, Eqn. (42) produces an algebraic equation:

$$a_P\phi_P^n + \sum_N a_N\phi_N^n = R_P \tag{43}$$

For every control volume, one equation of this form is assembled. The value of ϕ_P^n depends on the values in the neighbouring cells, thus creating a system of algebraic equations:

$$\left[A\right]\left[\phi\right] = \left[R\right] \tag{44}$$

where $[A]$ is a sparse matrix, with coefficients a_P on the diagonal and a_N off the diagonal, $\left[\phi\right]$ is the vector of ϕ-s for all control volumes and $[R]$ is the source term vector. The sparseness pattern of the matrix depends on the order in which the control volumes are labelled, with every off-diagonal coefficient above and below the diagonal corresponding to one of the faces in the mesh. In the rest of this study, Eqn. (44) will be represented by the typical

equation for the control volume, Eqn. (43).

When this system is solved, it gives a new set of ϕ values – the solution for the new time-step. As will be shown later, the coefficient a_P in Eqn. (43) includes the contribution from all terms corresponding to ϕ_n – the temporal derivative, convection and diffusion terms as well as the linear part of the source term. The coefficients a_N include the corresponding terms for each of the neighbouring points. The summation is done over all CV-s that share a face with the current control volume. The source term includes all terms that can be evaluated without knowing the new ϕ's, namely, the constant part of the source term, and the parts of the temporal derivative, convection and diffusion terms corresponding to the old time-level.

The Crank-Nicholson method of temporal discretisation is unconditionally stable ([6]), but does not guarantee boundedness of the solution. Examples of unrealistic solutions given by the Crank-Nicholson scheme can be found in Patankar and Baliga [See [5] ref. 106]. As in the case of the convection term, boundedness can be obtained if the equation is discretised to first order temporal accuracy.

It has been customary to neglect the variation of the face values of ϕ and $\nabla\phi$ in time ([7]). This leads to several methods of temporal discretisation. The new form of the discretised transport equation combines the old and new time-level convection, diffusion and source terms, leaving the temporal derivative unchanged:

$$\frac{\rho_P\phi_P^n - \rho_P\phi_P^0}{\Delta t}V_P + \sum_f F\phi_f - \sum_f \left(\rho\Gamma_\phi\right)_f \mathbf{S}\cdot\left(\nabla\phi\right)_f = Su\,V_P + Sp\,V_P\,\phi_P \tag{45}$$

The resulting equation is only first-order accurate in time and a choice has to be made about the way the face values of ϕ and $\nabla\phi$ are evaluated.

In **explicit discretisation**, the face values of ϕ and $\nabla\phi$ are determined from the old time-field:

$$\phi_f = f_x\phi_P^0 + (1 - f_x)\,\phi_N^0 \tag{46}$$

$$\mathbf{S}\cdot\left(\nabla\phi\right)_f = |\mathbf{\Delta}|\,\frac{\phi_N^0 - \phi_P^0}{|d|} + \mathbf{k}\cdot\left(\nabla\phi\right)_f^0 \tag{47}$$

The linear part of the source term is also evaluated using the old-time value. Eqn. (45) can be written in the form:

$$\phi_P^n = \phi_P^0 + \frac{\Delta t}{\rho_P\,V_P}\left[\sum_f F\phi_f - \sum_f \left(\rho\Gamma_\phi\right)_f \mathbf{S}\cdot\left(\nabla\phi\right)_f + Su\,V_P + Sp\,V_P\phi_P^0\right] \tag{48}$$

The consequence of this choice is that all terms on the r.h.s. of Eqn. (48) depend only on the old-time field. The new value of ϕ_P can be calculated directly – it is no longer necessary to solve the system of linear equations. The drawback of this method is the Courant number limit (Courant *et al.* [See [5] ref. 32]). The Courant number is defined as:

$$Co = \frac{U_f\cdot\mathbf{d}}{\Delta t} \tag{49}$$

If the Courant number is larger than unity, the explicit system becomes unstable. This is a severe limitation, especially if we are trying to solve a steady-state problem.

The Euler Implicit method expresses the face-values in terms of the new time-level cell values:

$$\phi_f = f_x \phi_P^n + (1 - f_x)\, \phi_N^n \tag{50}$$

$$\mathbf{S}\bullet\left(\nabla\phi\right)_f = |\mathbf{\Delta}|\, \frac{\phi_N^n - \phi_P^n}{|\mathbf{d}|} + \mathbf{k}\bullet\left(\nabla\phi\right)_f \tag{51}$$

This is still only first order accurate but, unlike the explicit method, it creates an system of equations like Eqn. (43). The coupling in the system is much stronger than in the explicit approach and the system is stable even if the Courant number limit is violated ([6]). Unlike the explicit method, this form of temporal discretisation guarantees boundedness.

Backward Differencing in time is a temporal scheme which is second-order accurate in time and still neglects the temporal variation of the face values. In orderto achieve this, each individual term of Eqn. (38) needs to be discretised to second order accuracy.

The discretised form of the temporal derivative in Eqn. (45) can be obtained in the following way: consider the Taylor series expansion of ϕ in time around $\phi\,(t + \Delta t) = \phi^n$:

$$\phi\,(t) = \phi^0 = \phi^n - \frac{\partial\phi}{\partial t}\Delta t + \frac{1}{2}\frac{\partial^2\phi}{\partial t^2} + o\left(\Delta t^3\right) \tag{52}$$

The temporal derivative can therefore be expressed as:

$$\frac{\partial\phi}{\partial t} = \frac{\phi^n - \phi^0}{\Delta t} + \frac{1}{2}\frac{\partial^2\phi}{\partial t^2}\Delta t + o\left(\Delta t^2\right) \tag{53}$$

In spite of the prescribed linear variation of ϕ in time, Eqn. (39) approximates the temporal derivative only to first order accuracy, since the first term of the truncation error in Eqn. (53) scales with Δt. However, if the temporal derivative is discretised up to second order, the whole discretisation of the transport equation will be second-order accurate even if the temporal variation of ϕ_f and $\left(\nabla\phi\right)_f$ is neglected.

In order to achieve this, the Backward Differencing in time uses three time levels. The additional Taylor series expansion for the 'second old' time level is:

$$\phi\,(t - \Delta t) = \phi^{00} = \phi^n - 2\frac{\delta\phi}{\delta t}\Delta t + 2\frac{\delta^2\phi}{\delta t}\Delta t^2 + o\left(\Delta r^3\right) \tag{54}$$

It is now possible to eliminate the term in the truncation error which scales with Δt. Combining Eqs. (52 and 54) the second-order approximation of the temporalderivative is:

$$\frac{\partial\phi}{\partial t} = \frac{\frac{3}{2} - 2\phi^0 + \frac{1}{2}\phi^{00}}{\Delta t} \tag{55}$$

Again, the boundedness of the solution cannot be guaranteed [...][14]. The final form of the discretised equation with Backward Differencing in time is:

[14]For a comparison between the Backward Differencing and the Crank-Nicholson method see Jasak's thesis Section 3.6

$$\frac{\frac{3}{2}\rho_P\phi^n - 2\rho_P\phi^0 + \frac{1}{2}\rho_P\phi^{00}}{\Delta t}V_P + \sum_f F\phi_f^n - \sum_f \left(\rho\Gamma_\phi\right)\mathbf{S}\cdot\left(\nabla\phi\right)_f^n = Su\,V_P + Sp\,V_P\,\phi_P^n \tag{56}$$

This produces a system of algebraic equations that must be solved for ϕ_n^P.

Steady-state problems are quite common in fluid flows. Their characteristic is that the solution is not a function of time, `i.e.` the transport equation reduces to:

$$\nabla\cdot\left(\rho\nabla\phi\right) - \nabla\cdot\left(\rho\Gamma_\phi\nabla\phi\right) = Su + Sp\,\phi \tag{57}$$

If we are solving a single equation of this type, the solution can be obtained in a single step. This is generally not the case: fluid flow problems require a solution of non-linear systems of coupled equations. If the non-linearity of the system is lagged, which is the case in the segregated approach used in this study, it is still necessary to solve the system in an iterative manner. In order to speed up the convergence, an implicit formulation is preferred. The convergence of the iterative procedure can be improved through under-relaxation, which will be described in Section 2.1.7

2.1.6 Implementation of Boundary Conditions

Let us now consider the implementation of boundary conditions. The computational mesh includes a series of faces which coincide with the boundaries of the physical domain under consideration. The conditions there are prescribed through the boundary conditions.

In order to simplify the discussion, the boundary conditions are divided into **numerical** and **physical boundary conditions**.

There are two basic types of numerical boundary conditions. Dirichlet (or fixed value) boundary condition prescribes the value of the variable on the boundary. Von Neumann boundary condition, on the other hand, prescribes the gradient of the variable normal to the boundary. These two types of boundary conditions can be built into the system of algebraic equations, Eqn. (43), before the solution.

Physical boundary conditions are symmetry planes, walls, inlet and outlet conditions for fluid flow problems, adiabatic or fixed temperature boundaries for heat transfer problems `etc`. Each of these conditions is associated with a set of numerical boundary conditions on each of the variables that is being calculated. Some more complicated boundary conditions (radiation boundaries, for example) may specify the interaction between the boundary value and the gradient on the boundary.

Numerical Boundary Conditions Before we consider the implementation of numerical boundary conditions, we have to address the treatment of non-orthogonality on the boundary. Consider a control volume with a boundary face b, shown in Fig. 7. In this situation, the vector $\mathbf{d}$ extends only to the centre of the boundary face.

It is assumed that a boundary condition specified for the boundary face is valid along the whole of the face. The decomposition of the face area vector into the orthogonal and

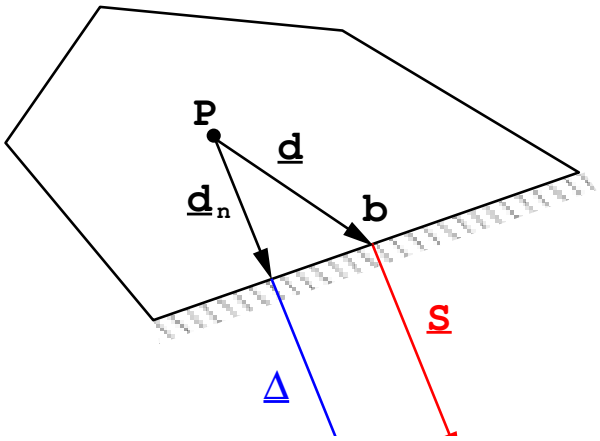


Figure 7: Control volume with a boundary face

non-orthogonal part is now simple: the vector **k** in 7 is parallel to the face. The orthogonal part of the face area vector (**Δ** in Fig. 7) is therefore equal to **S**, but is no longer located in the middle of the face.

The vector between the cell centre and the boundary face is now normal to the boundary:

$$\mathbf{d}_n = \frac{\mathbf{S}}{|\mathbf{S}|}\frac{\mathbf{d}\cdot\mathbf{S}}{|\mathbf{S}|} \tag{58}$$

and the correction vector **k** is not used.

- **Fixed Value Boundary Condition**

  The fixed value boundary condition prescribes the value of ϕ at the face b to be ϕ_b . This has to be taken into account in the discretisation of the convection and diffusion terms on the boundary face.

  Convection term. According to Eqn. (17), the convection term is discretised as:

$$\int_{V_P} \nabla\cdot\left(\rho\mathbf{U}\phi\right)\, dV = \sum_f F\,\phi_f \tag{59}$$

  It is known that the value of ϕ on the boundary face is ϕ_b. Therefore, the term for the boundary face is:

$$F_b\,\phi_b \tag{60}$$

  wher F_b is the face flux.

  Diffusion term. The diffusion term is discretised according to Eqn.(24).

$$\int_{V_P} \nabla\cdot\left(\rho\Gamma_\phi\nabla\phi\right)\, dV = \sum_f \left(\rho\Gamma_\phi\right)_f \mathbf{S}\cdot\left(\nabla\phi\right)_f \tag{61}$$

The face gradient at b is calculated from the known face value and the cell centre value:

$$\mathbf{S} \cdot \left(\nabla\phi\right)_b = |\mathbf{S}| \frac{\phi_b - \phi_P}{|\mathbf{d}_n|} \tag{62}$$

because $\mathbf{S}$ and $\mathbf{d}_n$ are parallel.

- **Fixed Gradient Boundary Condition**

  In the case of the fixed gradient boundary condition, the dot-product of the gradient and the outward pointing unit normal is prescribed on the boundary:

$$\left(\frac{\mathbf{S}}{|\mathbf{S}|} \cdot \nabla\phi\right)_b = g_b \tag{63}$$

  Convection term. The face value of ϕ is calculated from the value in the cell centre and the prescribed gradient:

$$\begin{aligned} \phi_b &= \phi_P + \mathbf{d}_n \cdot \left(\nabla\phi\right) \\ &= \phi_P + |\mathbf{d}_n| \; g_b \end{aligned} \tag{64}$$

  Diffusion term. The dot product between the face area vector and $\left(\nabla\phi\right)_b$ is known to be

$$|\mathbf{S}| \; g_b \tag{65}$$

  and the resulting term

$$\left(\rho\Gamma_\phi\right)_b |\mathbf{S}| \; g_b \tag{66}$$

  As the vector $\mathbf{d}_n$ does not point to the middle of the boundary face, the face integrals in the fixed gradient boundary condition are calculated only to first order accuracy. This can be rectified by including the boundary face correction based on the vector $\mathbf{k}$ (Fig. 7) and the component of the gradient parallel to the face in the first cell next to the boundary. [. . .][15].

Physical Boundary Conditions Let us now consider some physical boundary conditions for fluid flow calculations. For simplicity, we shall start with the **incompressible flow**.

- **Inlet boundary**. The velocity field at the inlet is prescribed and, for consistency, the boundary condition on pressure is zero gradient ([6]).
- **Outlet boundary**. The outlet boundary condition should be specified in such a way that the overall mass balance for the computational domain is satisfied.

  This can be done in two ways:

[15] However, an of the same type is neglected for the internal faces of the mesh and this correction is omitted for the sake of consistency, see Jasak's thesis Section 3.6

The velocity distribution for the boundary is projected from the inside of the domain (first row of cells next to the boundary). These velocities are scaled to satisfy overall continuity. This approach, however, leads to instability if inflow through a boundary specified as outlet occurs. The boundary condition on pressure is again zero gradient.

The other approach does not require the velocity distribution across the outlet – the pressure distribution is specified instead. The fixed value boundary condition is used for the pressure, with the zero gradient boundary condition on velocity. Overall mass conservation is guaranteed by the pressure equation [. . .][16].

- **Symmetry plane boundary**. The symmetry plane boundary condition implies that the component of the gradient normal to the boundary should be fixed to zero. The components parallel to it are projected to the boundary face from the inside of the domain.

- **Impermeable no-slip walls**. The velocity of the fluid on the wall is equal to that of the wall itself, so the fixed value boundary conditions prevail. As the flux through the solid wall is known to be zero, the pressure gradient condition is zero gradient.

Compressible flows at low Mach numbers are subject to the same approach as above. The situation is somewhat more complex in case of transonic and supersonic flows – the number of boundary conditions fixed at the inlet and outlet depends on the number of characteristics pointing into the domain. For these cases the reader is referred to [6] or [5] ref. 135.

For turbulent flows, the inlet and outlet boundary conditions on turbulence variables (k and , ϵ for example) are typically assigned to fixed values and zero gradients, respectively. The boundary conditions for the turbulence properties on the wall depend on the form of the selected turbulence model and the near-wall treatment.

2.1.7 Solution Techniques for Systems of Linear Algebraic Equations

Let us again consider the system of algebraic equations created by the discretisation, Eqn. (43):

$$a_P \phi_P^n + \sum_N a_N \phi_N^n = R_P$$

This system can be solved in several different ways. Existing solution algorithms fall into two main categories: **direct** and **iterative** methods. Direct methods give the solution of the system of algebraic equations in a finite number of arithmetic operations. Iterative methods start with an initial guess and then continue to improve the current approximation of the solution until some 'solution tolerance' is met. While direct methods are appropriate for small systems, the number of operations necessary to reach the solution raises with the number of equations squared, making them prohibitively expensive for large systems (See [[5] ref. 97]). Iterative methods are more economical, but they usually pose some requirements on the matrix.

The matrix resulting from Eqn. (43) is sparse – most of the matrix coefficients are equal to zero. If it were possible to choose a solver which preserves the sparsity pattern of the matrix, the computer memory requirements would be significantly decreased. Unlike direct solvers, some iterative methods preserve the sparseness of the original matrix. These properties

[16] See Section 2.1.8

make the use of iterative solvers very attractive.

Iterative solvers require diagonal dominance in order to guarantee convergence. A matrix is said to be diagonally equal if the magnitude of the diagonal (central) coefficient is equal to the sum of magnitudes of off-diagonal coefficients. The additional condition for diagonal dominance is that $|a_P| > |a_N|$ for at least one row of the matrix.

In order to improve the solver convergence, it is desirable to increase the diagonal dominance of the system. Discretisation of the linear part of the source term, Eqn. (37), is closely related to this issue. If $Sp < 0$, its contribution increases diagonal dominance and Sp is included into the diagonal. In the case of $Sp > 0$, diagonal dominance would be decreased. It is more effective to include this term into the source and update it when the new solution is available. This measure is, however, not sufficient to guarantee the diagonal dominance of the matrix.

The analysis of the structure of the matrix brings us back to the issue of boundedness. The sufficient boundedness criterion for systems of algebraic equations mentioned in Section 2.1.4 states that the boundedness of the solution will be preserved for diagonally equal systems of equations with positive coefficients. This allows us to examine the discretised form of all the terms in the transport equation from the point of view of boundedness and diagonal dominance and identify the troublesome parts of discretisation.

The convection term creates a diagonally equal matrix only for Upwind Differencing. Any other differencing scheme will create negative coefficients, violate the diagonal equality and potentially create unbounded solution. In the case of Central Differencing on a uniform mesh, the problem is further complicated by the fact that the central coefficient is equal to zero. In order to improve the quality of the matrix for higher-order differencing schemes, Khosla and Rubin (See [[5] ref. 73]) propose a **deferred correction implementation** for the convection term. Here, any differencing scheme is treated as an upgrade of UD. The part of the convection term corresponding to UD is treated implicitly (`i.e.` built into the matrix) and the other part is added into the source term. This, however, does not affect the boundedness in spite of the fact that the matrix is now diagonally equal, as the 'troublesome' part of the discretisation still exists in the source term.

The diffusion term creates a diagonally equal matrix only if the mesh is orthogonal. On non-orthogonal meshes, the second term in Eqn. (34) introduces the 'second neighbours' of the control volume into the computational molecule with negative coefficients, thus violating diagonal equality. As a consequence of mesh non-orthogonality, the boundedness of the solution cannot be guaranteed. The increase in the computational molecule would result in a higher number of non-zero matrix coefficients, implying a considerable increase in the computational effort re-quired to solve the system. On the other hand, the non-orthogonal correction is usually small compared to the implicit part of the diffusion term. *It is therefore reasonable to treat it through the source term. In this study, the diffusion term will therefore be split into the implicit orthogonal contribution, which includes only the first neighbours of the cell and creates a diagonally equal matrix and the non-orthogonal correction, which will be added to the source*. If it is important to resolve the non-orthogonal parts of the diffusion operators (like in the case of the pressure equation, see 2.1.8), non-orthogonal correctors are included. The system of algebraic equations, Eqn. (43), will be solved several times. Each new solution will be used to update the non-orthogonal correction terms, until the desired tolerance is met.

It should again be noted that this practice only improves the quality of the matrix but does not guarantee boundedness. If boundedness is essential, the non-orthogonal contribution should be discarded, thus creating a discretisation error [...][17].

At this point, the difference between the non-orthogonality treatments proposed in Section [Diffusion Term] becomes apparent. The decomposition of the face area vector into the orthogonal and non-orthogonal part determines the split between the implicit and explicit part of the term, with the consequences on the accuracy and convergence of non-orthogonal correctors. The comparison of different treatments is based on several criteria:

- On which angle of non-orthogonality is it necessary to introduce non-orthogonal correctors – how good an approximation of the converged solution can be obtained after only one solution of the system?
- How many non-orthogonal correctors are needed to meet a certain tolerance?
- How does the number of solver iterations change with the number of correctors?
- If the non-orthogonal correction needs to be discarded for the sake of boundedness, which approach causes the smallest discretisation error?[18]

[...]
The discretisation of the temporal derivative creates only the diagonal coefficient and a source term contribution, thus increasing the diagonal dominance. Unfortunately, the sufficient boundedness criterion cannot be used to establish the boundedness of the discretisation, as it does not take into account the influence of the source term.

The above discussion concentrates on the analysis of the discretisation on a term-by-term basis. In reality, all of the above coefficients contribute to the matrix, thus influencing the properties of the system. It has been shown that the only terms that actually enhance the diagonal dominance are the linear part of the source and the temporal derivative.

In steady-state calculations, the beneficial influence of the temporal derivative on the diagonal dominance does not exist. In order to enable the use iterative solvers, the diagonal dominance needs to be enhanced in some other way, namely through **under-relaxation**. Consider the original system of equations, Eqn. (43):

$$a_P \phi_P^n + \sum_N a_N \phi_N^n = R_P$$

Diagonal dominance is created through an artificial term added to both left and right-hand side of Eqn. (43):

$$a_P \phi_P^n + \frac{1-\alpha}{\alpha} a_P \phi_P^n + \sum_N a_N \phi_N^n = R_P + \frac{1-\alpha}{\alpha} a_P \phi_P^0 \tag{67}$$

or

$$\frac{a_P}{\alpha} \phi_P^n + \sum_N a_N \alpha_N^n = R_P + \frac{1-\alpha}{\alpha} a_P \phi_P^0 \tag{68}$$

[17] See Jasak's thesis Section 3.6.

[18] The discretisation error for the diffusion term is be derived in Jasak's thesis Section 3.6. For numerical results for the convergence of three non-orthogonality see Section 3.7.of same work

Here, ϕ_P^0 here represents the value of ϕ from the previous iteration and α is the under-relaxation factor ($0 < \alpha \leq 1$). Additional terms cancel out when steady-state is reached $\left(\phi_P^n = \phi_P^0\right)$.

In this study, the iterative solution procedure used to solve the system of algebraic equations is the Conjugate Gradient (CG) method, originally proposed by Hestens and Steifel ([See [5] ref. 63]). It guarantees that the exact solution will be obtained in the number of iterations smaller or equal to the number of equations in the system. The convergence rate of the solver depends on the dispersion of the eigenvalues of the matrix $[A]$ in Eqn. (44) and and can be improved through pre-conditioning. For symmetric matrices, the Incomplete Cholesky preconditioned Conjugate Gradient (ICCG) solver will be used. The method is described in detail by Jacobs, ([See [5] ref. 67]). The adopted solver for asymmetric matrices is the Bi-CGSTAB by van der Vorst ([See [5] ref. 136]).

2.1.8 Discretisation Procedure for the Navier-Stokes System

In this Section, a discretisation procedure for the Navier-Stokes equations will be presented. We shall start with the incompressible form of the system [given by the continuity equation and the Navier-Stokes equations]:

$$\nabla \bullet \mathbf{U} = 0 \tag{69}$$

$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \bullet (\mathbf{U}\mathbf{U}) - \nabla \bullet (\nu \nabla \mathbf{U}) = -\nabla p \tag{70}$$

Two issues require special attention: non-linearity of the momentum equation and the pressure-velocity coupling.

The non-linear term in Eqn. (70) is $\nabla \bullet (\mathbf{U}\mathbf{U})$, i.e. velocity is 'being transported by itself'. The discretised form of this expression would be quadratic in velocity and the resulting system of algebraic equations would therefore be non-linear. There are two possible solutions for this problem – either use a solver for non-linear systems, or linearise the convection term. Section 2.1.4 describes the discretisation of this term:

$$\begin{aligned} \nabla \bullet (\mathbf{U}\mathbf{U}) &= \sum_f \mathbf{S} \bullet (\mathbf{U})_f (\mathbf{U})_f \\ &= \sum_f F \, (\mathbf{U})_f \\ &= a_P \mathbf{U}_P + \sum_f a_N \mathbf{U}_N \end{aligned}$$

where F, a_P and a_N are a function of $\mathbf{U}$. The important issue is that the fluxes F should satisfy the continuity equation, [first equation in 70]. Eqs. (70) should therefore be solved together, resulting in an even larger (non-linear) system. Having in mind the complexity of non-linear equation solvers and the computational effort required, linearisation of the term is preferred. Linearisation of the convection term implies that an existing velocity (flux) field that satisfies [continuity] will be used to calculate a_P and a_N.

The linearisation does not have any effect in steady-state calculations. When the steady-state is reached, the fact that a part of the non-linear term has been lagged is not significant.

In transient flows two different approaches can be adopted: either to iterate over non-linear terms or to neglect the lagged non-linearity effects. Iteration can significantly increase the computational cost, but only if the time-step is large. The advantage is that the non-linear system is fully resolved for every time-step, whose size limitation comes only from the temporal accuracy requirements. If it is necessary to resolve the temporal behaviour well, a small time-step is needed. On the other hand, if the time-step is small, the change between consecutive solutions will also be small and it is therefore possible to lag the non-linearity without any significant effect. In this study, the PISO procedure proposed by Issa (See [[5] ref. 66]) is used for pressure-velocity coupling in transient calculations. For steady-state calculations, a SIMPLE pressure-velocity coupling procedure by Patankar [7] is used.

In Section 2.1.9 the problem of pressure-velocity coupling is presented. The pressure equation is derived for the incompressible Navier-Stokes system. Generalisation to compressible and transonic flows can be found in *e.g.* Demirdzic *et al.* [[5] ref. 39]. Section 2.1.10 describes the pressure-velocity coupling algorithms. Finally, in Section 2.1.11, a solution procedure for incompressible Navier-Stokes equations with a turbulence model is presented.

2.1.9 Derivation of the Pressure Equation

In order to derive the pressure equation, a semi-discretised form of the momentum equation will be used:

$$a_P \mathbf{U}_P = \mathbf{H}(\mathbf{U}) - \nabla p \tag{71}$$

In the spirit of the Rhie and Chow procedure [[5] ref. 117, [8]], the pressure gradient term is not discretised at this stage. Eqn. (71) is obtained from the integral form of the momentum equation, using the discretisation procedure described previously. It has been consequently divided through by the volume in order to enable face interpolation of the coefficients.

The $\mathbf{H}(\mathbf{U})$ term consists of two parts: the 'transport part', including the matrix coefficients for all neighbours multiplied by corresponding velocities and the 'source part' including the source part of the transient term and all other source terms apart from the pressure gradient (in our case, there are no additional source terms):

$$\mathbf{H}(\mathbf{U}) = -\sum_f a_N \mathbf{U} + \frac{\mathbf{U}^0}{\Delta t} \tag{72}$$

The discretised form of the continuity equation is:

$$\nabla \cdot \mathbf{U} = \sum_f \mathbf{S} \cdot \mathbf{U}_f = 0 \tag{73}$$

Eqn. (71) is used to express $\mathbf{U}$:

$$\mathbf{U}_P = \frac{\mathbf{H}(\mathbf{U})}{a_P} - \frac{1}{a_P} \nabla p \tag{74}$$

Velocities on the cell face are expressed as the face interpolate of Eqn. (74):

$$\mathbf{U}_f = \left(\frac{\mathbf{H}(\mathbf{U})}{a_P} \right)_f - \left(\frac{1}{a_P} \right) (\nabla p)_f \tag{75}$$

This will be used later to calculate the face fluxes.

When Eqn. (75) is substituted into Eqn. (73), the following form of the pressure equation is obtained:

$$\begin{aligned} \nabla\cdot\left(\frac{1}{a_P}\nabla p\right) &= \nabla\cdot\left(\frac{\mathbf{H}(\mathbf{U})}{a_P}\right) \\ &= \sum_f \mathbf{S}\left(\frac{\mathbf{H}(\mathbf{U})}{a_P}\right)_f \end{aligned} \tag{76}$$

The Laplacian on the l.h.s. of Eqn. (76) is discretised in the standard way (see Section 2.1.4).

The final form of the discretised incompressible Navier-Stokes system is:

$$a_P\mathbf{U}_P = \mathbf{H}(\mathbf{U}) - \sum_f \mathbf{S}(p)_f \tag{77}$$

$$\sum_f \mathbf{S}\cdot\left[\left(\frac{1}{a_P}\right)_f (\nabla p)_f\right] = \sum_f \mathbf{S}\left(\frac{\mathbf{H}(\mathbf{U})}{a_P}\right)_f \tag{78}$$

The face flux F is calculated using Eqn. (75):

$$F = \mathbf{S}\cdot\mathbf{U}_f = \mathbf{S}\cdot\left[\left(\frac{\mathbf{H}(\mathbf{U})}{a_P}\right)_f - \left(\frac{1}{a_P}\right)_f (\nabla p)_f\right] \tag{79}$$

When Eqn. (76) is satisfied, the face fluxes are guaranteed to be conservative.

2.1.10 Pressure-Velocity Coupling

Consider the discretised form of the Navier-Stokes system, Eqs. (77 and 78). The form of the equations shows linear dependence of velocity on pressure and vice-versa. This inter-equation coupling requires a special treatment.

[[5] ref. 117]
Simultaneous algorithms (Caretto *et al.* [[5] ref. 23], Vanka [[5] ref. 143]) operate by solving the complete system of equations simultaneously over the whole domain. Such a procedure might be considered when the number of computational points is small and the number of simultaneous equations is not too large. The resulting matrix includes the inter-equation coupling and is several times larger than the number of computational points. The cost of a simultaneous solution is great, both in the number of operations and memory requirements.

In the **segregated approach** (*e.g.* Patankar [7], Issa [[5] ref. 66]) the equations are solved in sequence. A special treatment is required in order to establish the necessary inter-equation coupling. PISO [[5] ref. 66], SIMPLE [7] and their derivatives are the most popular methods of dealing with inter-equation coupling in the pressure-velocity system.

The PISO Algorithm for Transient Flows This pressure-velocity treatment for transient flow calculations has been originally proposed by Issa [[5] ref. 66]. Let us again consider the discretised Navier-Stokes system for incompressible flow, Eqs. (77 and 78). The PISO algorithm can be described as follows:

- The momentum equation is solved first. The exact pressure gradient source term is not known at this stage – the pressure field from the previous time-step is used instead. This stage is called the **momentum predictor**. The solution of the momentum equation, Eqn. (77), gives an approximation of the new velocity field.
- Using the predicted velocities, the $\mathbf{H}(\mathbf{U})$ operator can be assembled and the pressure equation can be formulated. The solution of the pressure equation gives the first estimate of the new pressure field. This step is called the **pressure solution**.
- Eqn. (79) gives a set of conservative fluxes consistent with the new pressure field. The velocity field should also be corrected as a consequence of the new pressure distribution. Velocity correction is done in an explicit manner, using Eqn. (74). This is the **explicit velocity correction** stage.

A closer look to Eqn. (74) reveals that the velocity correction actually consists of two parts – a correction due to the change in the pressure gradient ($\frac{1}{a_P}\nabla p$ term) and the transported influence of corrections of neighbouring velocities ($\frac{\mathbf{H}(\mathbf{U})}{a_P}$ term). The fact that the velocity correction is explicit means that the latter part is neglected – it is effectively assumed that the whole velocity error comes from the error in the pressure term. This, of course, is not true. It is therefore necessary to correct the $\mathbf{H}(\mathbf{U})$ term, formulate the new pressure equation and repeat the procedure. In other words, the PISO loop consists of an implicit momentum predictor followed by a series of pressure solutions and explicit velocity corrections. The loop is repeated until a pre-determined tolerance is reached.

Another issue is the dependence of $\mathbf{H}(\mathbf{U})$ coefficients on the flux field. After each pressure solution, a new set of conservative fluxes is available. It would be therefore possible to recalculate the coefficients in $\mathbf{H}(\mathbf{U})$. This, however, is not done: it is assumed that the non-linear coupling is less important than the pressure-velocity coupling, consistent with the linearisation of the momentum equation. The coefficients in $\mathbf{H}(\mathbf{U})$ are therefore kept constant through the whole correction sequence and will be changed only in the next momentum predictor.

The SIMPLE Algorithm If a steady-state problem is being solved iteratively, it is not necessary to fully resolve the linear pressure-velocity coupling, as the changes between consecutive solutions are no longer small. Non-linearity of the system becomes more important, since the effective time-step is much larger.

The SIMPLE algorithm by Patankar [7] is formulated to take advantage of these facts:

- An approximation of the velocity field is obtained by solving the momentum equation. The pressure gradient term is calculated using the pressure distribution from the previous iteration or an initial guess. The equation is under-relaxed in an implicit manner (see Eqn. 68), with the **velocity under-relaxation** factor α_U.
- The pressure equation is formulated and solved in order to obtain the new pressure distribution.

- A new set of conservative fluxes is calculated using Eqn. (79). As it has been noticed before, the new pressure field includes both the pressure error and convection-diffusion error. In order to obtain a better approximation of the 'correct' pressure field, it would be necessary to solve the pressure equation again. On the other hand, the non-linear effects are more important than in the case of transient calculations. It is enough to obtain an approximation of the pressure field and recalculate the **H** (**U**) coefficients with the new set of conservative fluxes. The pressure solution is therefore under-relaxed in order to take into account the velocity part of the error:

$$p^{new} = p^{old} + \alpha_p \left(p^p - p^{old} \right) \tag{80}$$

where

p^{new} is the approximation of the pressure field that will be used in the next momentum predictor,

p^{old} is the pressure field used in the momentum predictor,

p^p is the solution of the pressure equation,

α_p is the **pressure under-relaxation factor**, $\left(0 < \alpha_p \leq 1\right)$.

If the velocities are needed before the next momentum solution, the explicit velocity correction, Eqn. (74), is performed.

Peric, [[5] ref. 109] gives an analysis of the under-relaxation procedure based on the expected behaviour of the second corrector in the PISO sequence. The recommended values of under-relaxation factors are (Peric, [[5] ref. 109]):

- $\alpha_p = 0.2$ for the pressure and
- $\alpha_U = 0.8$ for momentum.

2.1.11 Solution Procedure for the Navier-Stokes System

It is now possible to describe the solution sequence for the Navier-Stokes system with additional coupled transport equations (*e.g.* a turbulence model, combustion equations, energy equation or some other equations that influence the system).

In transient calculations, all inter-equation couplings apart from the pressure-velocity system are lagged. If it is necessary to ensure a closer coupling between some of the equations (*e.g.* energy and pressure in combustion), they are included in the PISO loop. A **transient solution procedure** for incompressible turbulent flows can be summarised as follows:

1. Set up the initial conditions for all field values.
2. Start the calculation of the new time-step values.
3. Assemble and solve the momentum predictor equation with the available face fluxes.
4. Go through the PISO loop until the tolerance for pressure-velocity system is reached. At this stage, pressure and velocity fields for the current time-step are obtained, as well as the new set of conservative fluxes.

5. Using the conservative fluxes, solve all other equations in the system. If the flow is turbulent, calculate the effective viscosity from the turbulence variables.

6. If the final time is not reached, return to step 2.

The solution procedure for **steady-state incompressible turbulent flow** is similar:

1. Set all field values to some initial guess.

2. Assemble and solve the under-relaxed momentum predictor equation.

3. Solve the pressure equation and calculate the conservative fluxes. Update the pressure field with an appropriate under-relaxation. Perform the explicit velocity correction using Eqn. (74)

4. Solve the other equations in the system using the available fluxes, pressure and velocity fields. In order to improve convergence, under-relax the other equations in an implicit manner, as shown in Eqn. (68).

5. Check the convergence criterion for all equations. If the system is not converged, start a new iteration on step 2."

2.1.12 An alternative derivation of the Pressure Equation

In order to provide more clarity in the derivation of pressure equation following is a new approach on this topic introduced by Prof. Jasak in his Lecture Notes for the University of Zagreb[9]. This is based on the Schur Complement concept.

"Consider a general block matrix system M, consisting of 4 block matrices, A, B, C, and D, which are respectively $p \times p$, $p \times q$, $q \times p$ and $q \times q$ matrices and A is invertible:

$$\begin{bmatrix} A & B \\ C & D \end{bmatrix} \tag{81}$$

This structure will arise naturally when trying to solve a block system of equations

$$\begin{aligned} A\,x + B\,y &= a \\ C\,x + D\,y &= b \end{aligned} \tag{82}$$

The Schur complement arises when trying to eliminate x from the system using partial Gaussian elimination by multiplying the first row with $A^{?1}$:

$$A^{-1}Ax + A^{-1}By = A^{-1}a \tag{83}$$

and

$$x = A^{-1}a - A^{-1}By \tag{84}$$

Substituting the above into the second row:

$$\left(D - CA^{-1}B\right) y = b - CA^{-1}a \tag{85}$$

Let us repeat the same set of operations on the block form of the pressure-velocity system, attempting to assemble a pressure equation. Note that the operators in the block system could be considered both as differential operators and in a discretised form

$$\begin{bmatrix} [A_{\mathbf{u}}] & [\nabla(.)] \\ [\nabla\bullet(.)] & [0] \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} 0 \\ 0 \end{bmatrix} \tag{86}$$

Formally, this leads to the following form of the pressure equation:

$$[\nabla\bullet(.)] \left[A_{\mathbf{u}}^{-1}\right] [\nabla(.)] [p] = 0 \tag{87}$$

Here, $A_{\mathbf{u}}^{-1}$ represent the inverse of the momentum matrix in the discretised form, *which acts as diffusivity in the Laplace equation for the pressure*.

From the above, it is clear that the governing equation for the pressure is a Laplacian, with the momentum matrix acting as a diffusion coefficient. However, the form of the operator is very inconvenient:

- While $[A_{\mathbf{u}}]$ is a sparse matrix, its inverse is likely to be dense
- Discretised form of the divergence and gradient operator are sparse and well-behaved. However, a triple product with $\left[A_{\mathbf{u}}^{-1}\right]$ would result in a dense matrix, making it expensive to solve

The above can be remedied be decomposing the momentum matrix before the triple product into the diagonal part and off-diagonal matrix:

$$[A_{\mathbf{u}}] = [D_{\mathbf{u}}] + [LU_{\mathbf{u}}] \tag{88}$$

where $[D_{\mathbf{u}}]$ only contains diagonal entries. $[D_{\mathbf{u}}]$ *is easy to invert and will preserve the sparseness pattern in the triple product*. Revisiting Eqn. (86) before the formation of the Schur complement and moving the off-diagonal component of $[A_{\mathbf{u}}]$ onto r.h.s. yields:

$$\begin{bmatrix} [D_{\mathbf{u}}] & [\nabla(.)] \\ [\nabla\bullet(.)] & [0] \end{bmatrix} \begin{bmatrix} \mathbf{u} \\ p \end{bmatrix} = \begin{bmatrix} -[LU_{\mathbf{u}}][\mathbf{u}] \\ 0 \end{bmatrix} \tag{89}$$

A revised formulation of the pressure equation via a Schur's complement yields:

$$[\nabla\bullet(.)] \left[D_{\mathbf{u}}^{-1}\right] [\nabla(.)] [p] = [\nabla\bullet(.)] \left[D_{\mathbf{u}}^{-1}\right] [LU_{\mathbf{u}}] [\mathbf{u}] \tag{90}$$

In both cases, matrix $\left[D_{\mathbf{u}}^{-1}\right]$ is simple to assemble.

It follows that the pressure equation is a Poisson equation with the *diagonal part of the discretised momentum acting as diffusivity and the divergence of the velocity on the r.h.s.*

Derivation of the pressure equation We shall now rewrite the above derivation formally without resorting to the assembly of Schur's complement in order to show the identical result

We shall start by discretising the momentum equation using the techniques described before. For the purposes of derivation, the pressure gradient term will remain in the differential form. For each CV, the discretised momentum equation yields:

$$a_P^{\mathbf{u}} \mathbf{u}_P + \sum_N a_N^{\mathbf{u}} \mathbf{u}_N = \mathbf{r} - \nabla p \tag{91}$$

For simplicity, we shall introduce the H(u) operator, containing the off-diagonal part of the momentum matrix and any associated r.h.s. contributions:

$$\mathbf{H}(\mathbf{u}) = \mathbf{r} - \sum_N a_N^{\mathbf{u}} \mathbf{u}_N \tag{92}$$

Using the above, it follows:

$$a_P^{\mathbf{u}} \mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \nabla p \tag{93}$$

and

$$\mathbf{u}_P = \left(a_P^{\mathbf{u}}\right)^{-1} (\mathbf{H}(\mathbf{u}) - \nabla p) \tag{94}$$

Substituting the expression for $\mathbf{u}_P$ into the incompressible continuity equation $\nabla \bullet \mathbf{u} = 0$ yields

$$\nabla \bullet \left[\left(a_P^{\mathbf{u}}\right)^{-1} \nabla p\right] = \nabla \bullet \left(\left(a_P^{\mathbf{u}}\right)^{-1} \mathbf{H}(\mathbf{u})\right) \tag{95}$$

We have again arrived to the identical form of the pressure equation

Note the implied decomposition of the momentum matrix into the diagonal and off-diagonal contribution, where $\left(a_P^{\mathbf{u}}\right)$ is an coefficient in $[D_{\mathbf{u}}]$ matrix and $\mathbf{H}(\mathbf{u})$ is the product $[LU_{\mathbf{u}}][u]$, both appearing in the previous derivation

Assembling Conservative Fluxes Pressure equation has been derived from the continuity condition and the role of pressure is to guarantee a divergence-free velocity field

Looking at the discretised form of the continuity equation

$$\nabla \bullet \mathbf{u} = \sum_f \mathbf{s}_f \bullet \mathbf{u} = \sum_f F \tag{96}$$

where F is the face flux

$$F = \mathbf{s}_f \bullet \mathbf{u} \tag{97}$$

Therefore, conservative face flux should be created from the solution of the pressure equation. If we substitute expression for $\mathbf{u}$ into the flux equation, it follows:

$$F = -\left(a_P^{\mathbf{u}}\right)^{-1} \mathbf{s}_f \bullet \nabla p + \left(a_P^{\mathbf{u}}\right)^{-1} \mathbf{s}_f \bullet \mathbf{H}(\mathbf{u}) \tag{98}$$

A part of the above, $\left(a_P^{\mathbf{u}}\right)^{-1} \mathbf{s}_f \bullet \nabla p$ appears during the discretisation of the Laplacian, for each face. This is discretised as follows:

$$\left(a_P^{\mathbf{u}}\right)^{-1} \mathbf{s}_f \bullet \nabla p = \left(a_P^{\mathbf{u}}\right)^{-1} \frac{\left|\mathbf{s}_f\right|}{|d|} (p_N - p_P) = a_N^p (p_N - p_P) \tag{99}$$

Here, $a_N^p = \left(a_P^{\mathbf{u}}\right)^{-1} \frac{|\mathbf{s}_f|}{|d|}$ is equal to the off-diagonal matrix coefficient in the pressure Laplacian.

Note that in order for the face flux to be conservative, assembly of the flux must be completely consistent with the assembly of the pressure equation (*e.g.* non-orthogonal correction) ".

2.2 Volume of Fluid Method

2.2.1 Introduction

Volume of Fluid (VOF) Method was presented by Hirt & Nichols [10] and started a new trend in multiphase flow simulation. It relies on the definition of an *indicator* function. This function allows us to know wheter the cell is occupied by one fluid or another, or a mix of both, quoting the original paper:

> "Suppose [...] that we define a function F whose value is unity at any point occupied by fluid an zero otherwise. The average value of F in a cell would then represent the fractional volume of the cell occupied by fluid. In particular, a unit value of F would correspond to cell full of fluid, while a zero value would indicate that the cell contained no fluid [19]. Cell with F values between zero and one must then contain a free surface [...]
> The normal direction to the boundary lies in the direction in which de value of F changes most rapidly. Because F is a step function, however, its derivatives must be computed in a special way [...]. When properly computed, the derivatives can be used to determine the boundary normal. Finally, when both the normal direction and the value of F in the boundary cell are known, a line cutting the cell can be constructed that approximates the interface there. This boundary location can then be used in the setting of boundary conditions".

There are several ways to implement this concept, but in the framework of OpenFOAM is worthy to analize its special way to do so. Methodology is described *in extenso* by Ubbink [11] and Rusche [12], but a concise and up to date[20] explanation is given by Berberovic *et. al.* [13]. Following are cited some sections of this paper and an explanation of PISO loop in VOF solvers is added.

2.2.2 Mathematical model

"In the conventional volume-of-fluid (VOF) method [10], the transport equation for an indicator function, representing the volume fraction of one phase, is solved simultaneously with the continuity and momentum equations:

$$\nabla \cdot \mathbf{U} = 0 \tag{100}$$

$$\frac{\partial \gamma}{\partial t} + \nabla \cdot (\mathbf{U}\,\gamma) = 0 \tag{101}$$

$$\frac{\partial (\rho \mathbf{U})}{\partial t} + \nabla \cdot (\rho \mathbf{U}\mathbf{U}) = -\nabla p + \nabla \cdot \mathbf{T} + \rho \mathbf{f}_b \tag{102}$$

where $\mathbf{U}$ represents the velocity field shared by the two fluids troughout the flow domain, γ is the phase fraction, $\mathbf{T}$ is the deviatoric viscous stress tensor $\mathbf{T} = 2\mu\mathbf{S} - 2\mu\,(\nabla \cdot \mathbf{U})\,\mathbf{I}/3$, with the mean rate of strain tensor $\mathbf{S} = 0.5\left[\nabla \mathbf{U} + (\nabla \mathbf{U})^T\right]$ and $\mathbf{I} = \delta_{ij}$, ρ is density, p is pressure, $\mathbf{f}_b$ are body forces pero unit mass. In VOF simulations the latter forces include gravity and

[19] This statament can be interpreted how $F = 1$ means cell completely occupied by one fluid and $F = 0$ completely occupied by another fluid.

[20] It has be taken in account that OpenFOAM is evolving constantly and some implementation details can change, its important to use actualized bibliography

surface tension effects at the interface. The phase fraction γ can take values whithin the range $0 \leqslant \gamma \leqslant 1$, with the values of zero and one corresponding to regions accomodating only one phase. e.g., $\gamma = 0$ for gas and $\gamma = 1$ for liquid. Accordingly, gradients of the phase fraction are encountered only in the region of the interface.

Two inmiscible fluids are considered as one effective fluid throughout the domain, the physical properties of which are calculated as weighted averages based on the distrubution of the liquid volume fraction, thus being equal to the properties of each fluid in their corresponding occcupied regions and varying only across the interface,

$$\rho = \rho_l \gamma + \rho_g (1 - \gamma) \tag{103}$$

$$\mu = \mu_l \gamma + \mu_g (1 - \gamma) \tag{104}$$

where ρ_l and ρ_g are densities of liquid and gas, respectively.

One of the critical issues in numerical simulations of free surface flows using the VOF model is the conservation of the phase fraction. This is specially the case in flows with high density ratios, where small errors in volume fraction may lead to significant errors in calculations of physical properties. Accurate calculation of the phase fraction distribution is crucial for a proper evaluation of surface curvature, which is required for the determination of surface tension force and the corresponding pressure gradient acrosss the free surface. The interface region between two phases is tipically smeared over a few grid cells and is therefore highly sensitive to grid resolution.

Is not a simple task to assure boundedness and conservativeness of the phase fraction. Various attemps have been made in order to overcome these difficulties (see [13], refs. 26-29). Furthermore, the definition of velocity by which the free surface is advancesd, as a single velocity being shared by both phases, is misleading, e.g. no conclusion can be made as to what extent the velocity of each particular phase contributes to the velocity of the efective fluid.

In the present study a modified approach similar to one proposed in [12] is used, with an advanced model formulated by OpenCFD Ltd. [2], relying on a two-fluid formulation of the conventional volume-of-fluid model in the framework of finite volume method. Its systematic derivation is outlined below. In this model an additional convective term originating from modeling the velocity in terms of weighted average of the corresponding liquid and gas velocities is introduced into the transport equation for phase fraction, providing a sharper interface resolution. The model makes use of the two-fluid Eulerian model for two-phase flow, where phase fraction equations are solved separately for each individual phase (see [13], ref. 32); hence the equations for each of the phase fractions can be expressed as

$$\frac{\partial \gamma}{\partial t} + \nabla \cdot (\mathbf{U}_l \gamma) = 0 \tag{105}$$

$$\frac{\partial (1 - \gamma)}{\partial t} + \nabla \cdot \left[\mathbf{U}_g (1 - \gamma) \right] = 0 \tag{106}$$

where the subscripts l and g denote the liquid and gaseous phase, respectively. Assuming that the contributions of the liquid and gas velocities to the evolution of the free surface are

proportional to the corresponding phase fraction, and defining the velocity of the effective fluid in a VOF model as a weighted average [14]

$$\mathbf{U} = \gamma \mathbf{U}_l + (1-\gamma)\,\mathbf{U}_g \tag{107}$$

Eq. (105) can be rearranged[21] and used as an evolution equation for the phase fraction γ,

$$\frac{\partial \gamma}{\partial t} + \nabla\cdot(\mathbf{U}\gamma) + \nabla\cdot[\mathbf{U}_r\gamma\,(1-\gamma)] = 0 \tag{108}$$

where $\mathbf{U}_r = \mathbf{U}_l - \mathbf{U}_g$ is the vector of relative velocity, designated as the 'compression velocity'.

Accordingly, the equation governing the volume fraction [Eq. (108)] contains an additional convective term, referred to as the 'compression term' keeping in mind its role to 'compress' the free surface towardss a sharper one (it should be noted that the wording compression represents just a denotation and does not relate to compressible flow). In comparison to Eq. (101), *this term appears as an artificial contribution to convection of the phase fraction*, but since the derivation of Eq. (108) relies on the velocity defined by Eq. (107), a strong coupling between the classical VOF and two-fluid model is achieved. The additional convective term contributes significantly to a higher interface resolution, thus avoiding the

[21] Starting with the transport equation for γ we have:

$$\frac{\partial \gamma}{\partial t} + \nabla\cdot(\mathbf{U}\gamma)$$

now, replacing the velocity U by its definition as a weighted average,

$$\frac{\partial \gamma}{\partial t} + \nabla\left\{\left[\gamma\mathbf{U}_l + (1-\gamma)\;\mathbf{U}_g\right]\gamma\right\} = 0$$

from the definition of the relative velocity we can isolate $\mathbf{U}_g$

$$\mathbf{U}_g = \mathbf{U}_l - \mathbf{U}_r$$

then replacing in the above transport equations

$$\frac{\partial \gamma}{\partial t} + \nabla\cdot\{[\mathbf{U}_l - (1-\gamma)\;\mathbf{U}_r]\;\gamma\} = 0$$

rearranging terms

$$\underbrace{\frac{\partial \gamma}{\partial t} + \nabla\cdot(\mathbf{U}_l\,\gamma)}_{=0} - \nabla\cdot[(1-\gamma)\;\gamma\,\mathbf{U}_r] = 0$$

the first two terms of left hand side are zero by the definition of tranport equation for γ at the 'liquid' phase, then we have,

$$\nabla\cdot[(1-\gamma)\;\gamma\,\mathbf{U}_r] = 0$$

as it's expressed above this term vanishes in the continuum formulation because in this case interface front is step function, then is term is ever completely zero.

Now adding this new term to the γ transport equation we obtain

$$\frac{\partial \gamma}{\partial t} + \nabla\cdot(\mathbf{U}\gamma) + \nabla\cdot[\mathbf{U}_r\gamma\,(1-\gamma)] = 0$$

this term has no meaning in the continuum formulation but is suitable to compresss the interface in the discrete formulation, specially when the interface is not sharp enough.

need to devise a special scheme for convection, such as CICSAM (see [13], ref. 34). This term is active only within the interface region and vanishes at both limits of the phase fraction. Therefore it does not affect the solution outside this region. *Moreover if free surface is defined in a theoretical sense as having an infinitesimally small thickness, the (relative) velocity* $\mathbf{U}_r$ *vanishes and the expression* (108) *reduces to the conventional form* (101).

In addition to properly reflecting the physics of the flow, the main advantage of such formulation is in the possibility of capturing the interface region much more sharply in comparison to the classical VOF approach. Numerical diffusion, unavoidably introduced through the discretization of convective terms, can be controlled and minimized through the discretization of the compression term, thus allowing sharp interface resolution. The details of its numerical treatment are given [later]. [. . .].

The momentum equation, Eq. 102, is modified in order to account for the effects of surface tension. The surface tension at the liquid-gas interface generates an additional pressure gradient resulting in a force, which is evaluated per unit volume using the continuum surface force (CSF) model (see [13], ref. 35).

$$\mathbf{f}_\sigma = \sigma\kappa\nabla\gamma \tag{109}$$

where κ is the mean curvature of the free surface, determined from the expressions

$$\kappa = -\nabla\cdot\left(\frac{\nabla\gamma}{\left|\nabla\gamma\right|}\right) \tag{110}$$

Equation (109) is only valid for the cases with constant surface tension, as considered here. In the case of variable surface tension, e.g., due to nonuniformly distributed temperature, surface tension gradients are encountered, generating an additional shear stress at the interface, which should be taken into account.

Both fluids are considered to be Newtonian and incompressible $\nabla\cdot\mathbf{U} = 0$, and the rate of strain tensor is linearly related to the stress tensor, which is decomposed into a more convenient form for discretization,

$$\nabla\cdot\mathbf{T} = \mu\left[\nabla\mathbf{U} + (\nabla\mathbf{U})^T\right] = \nabla\cdot(\mu\nabla\mathbf{U}) + (\nabla\mathbf{U})\cdot\nabla\mu \tag{111}$$

In a single pressure system as considered for the present VOF method, the normal component of the pressure gradient at a stationary nonvertical solid wall, with no-slip condition on velocity, must be different for each phase due to the hydrostatic component ρg when the phases are separated at the wall, i.e., if a contact line exists. In order to simplify the definition of boundary conditions, it is common to define a modified pressure as

$$p_d = p - \rho\mathbf{g}\cdot\mathbf{x} \tag{112}$$

where $\mathbf{x}$ is the position vector. It can be easely show that the gradient of modified pressure p_d of the static pressure gradient, the body force due to gravity and an additional contribution originating from the density gradient. In order to satisfy the momentum equation, the pressure gradient is expressed using Eq. (112) whereas the momentum equation is rearranged to read [12]

$$\frac{\rho\mathbf{U}}{\partial t} + \nabla\cdot(\rho\mathbf{U}\mathbf{U}) - \nabla\cdot(\mu\nabla\mathbf{U}) - (\nabla\mathbf{U})\cdot\nabla\gamma = -\nabla p_d - \mathbf{g}\cdot\mathbf{x}\nabla\rho + \sigma\kappa\nabla\gamma \tag{113}$$

Body forces due to pressure gradient and gravity are implicitly accounted for by the first two terms on the right-hand side of the Eq. (113). Summing up, the present mathematical model is given by the continuity equation, Eq. (100), phase fraction equation, Eq. (108), and momentum equation, Eq. (113).

The model is closed by supplying an appropriate expression for the compression velocity $\mathbf{U}_r$. In order to ensure that this velocity does not bias the solution in any way, *it must act only in the perpendicular direction to the interface*. Furthermore, by inspection of Eq. (113) it is evident that only the values of $\mathbf{U}_r$ on the grid cell faces will be used, being in accordance with the discretization of the convective term. The model for $\mathbf{U}_r$ is described in detail [later].

2.2.3 Computational method

Discrezation of the "compression term" For the discretization of the compression term in Eq. (108) the relative velocity at cell faces, formulated based on the maximum velocity magnitude at the interface region and its direction, is determined from the gradient of phase fraction as follows:

$$U_{r,f} = n_f \min\left[C_\gamma \frac{|\phi|}{|\mathbf{S}_f|}, \max\left(\frac{|\phi|}{|\mathbf{S}_f|}\right)\right] \tag{114}$$

where ϕ is face volume flux, and n_f is face unit normal flux, calculated at cell faces in the interface region using the phase fraction gradient at cell faces,

$$n_f = \frac{(\nabla\gamma)_f}{\left|(\nabla\gamma)_f + \delta_n\right|} \cdot \mathbf{S}_f \tag{115}$$

In the normalization of the phase fraction gradient in Eq. (115) and Eq. (110), a stabilization factor δ_n is used, which accounts for nonuniformity of the grid,

$$\delta_n = \frac{\varepsilon}{\left(\frac{\sum_N V_i}{N}\right)^{1/3}} \tag{116}$$

where N is the number of computational cells and ε is a small parameter, set to $10^{?8}$ here.

The model is relatively simple and robust, relying basically on the definition of the velocity in Eq. (107). If there is a small bulk motion of the gaseous phase in the vicinity of the free surface, the relative velocity will be close to the velocity of the liquid phase. If the velocities of both phases are of the same order of magnitude, the intensity of the free surface compression is controlled by the constant C_γ, which yields no contribution if set to zero, a conservative compression if the value is one [...], and enhanced compression for values greater than one [16]. *It should be noted that the face volume flux in Eq. (114) is not evaluated using the face interpolation of the velocity, but is determined as a conservative volume flux resulting from the pressure-velocity coupling algorithm.*[22]

Adaptive time step control In order to ensure stability of the solution procedure, the calculations are performed using a self-adapting time step which is adjusted at the beginning

[22] This topic has been discused in Section 2.1.12, paragraph Assembling Conservative Fluxes.

of the time iteration loop based on the Courant number defined as

$$\mathrm{Co} = \frac{\mathbf{U}_f \cdot \mathbf{S}_f}{\mathbf{d} \cdot \mathbf{S}_f} \Delta t \tag{117}$$

where $\mathbf{d}$ is a vector between calculation points of control volumes sharing the face. i.e. $\mathbf{d} = \overline{PN}$ and Δt is time step. Using values for $\mathbf{U}_f$ and Δ_t from previous time step, a maximum local Courant number Co^0 is calculated and the new time step is evaluated from the expression

$$\Delta_t^n = \min\left\{ \frac{\mathrm{Co}_{\max}}{\mathrm{Co}^0} \Delta t^0, \left(1 + \lambda_1 \frac{\mathrm{Co}_{\max}}{\mathrm{Co}^0}\right) \Delta t^0, \lambda_2 \Delta_t^0, \Delta t_{\max} \right\} \tag{118}$$

where $\Delta t_{\max}$ and $\mathrm{Co}_{\max}$ are prescribed limit values for the time step and Courant number, respectively.

According to this prescription the new time step will decrease if Co^0 overshoots $\mathrm{Co}_{\max}$ and increase otherwise. To avoid time step oscillations that may lead to instability, the increase of the time step is damped using factors λ_1 and λ_2, according to the conditions in Eq. (118).

[. . .]

[. . .] At the startup of the simulation, usually some very small initial time step Δt_{init} is used, which could lead to a very small maximum local value of the Courant number and a new time step that would be too large for the start, and vice versa. Therefore, at the beginning of the calculation an intermediate value for the initial time step is calculated as

$$\Delta t_{\mathrm{init}}^* = \min\left(\frac{\mathrm{Co}_{\max} \Delta t_{\mathrm{init}}}{\mathrm{Co}^0}; \Delta t_{\max} \right) \tag{119}$$

This intermediate value is than used as Δt^0 in Eq. (118) providing the value of Co^0 for the first time step to be close to the prescribed limit value $\mathrm{Co}_{\max}$. [. . .].

Temporal subcycling It is common in VOF-based methods that the convergence and stability of the solution procedure are very sensitive with respect to the equation for phase fraction. Bounded discretization schemes for divergence terms and time step control are both used to overcome these difficulties and, although it is generally recommended to keep the maximum local Courant number much below unity, it is beneficial to solve the phase fraction equation in several subcycles within a single time step. The time step to be used in a single time subcycle is set by dividing the global time step by the preset number of subcycles,

$$\Delta t_{\mathrm{sc}} = \frac{\Delta t}{n_{\mathrm{sc}}} \tag{120}$$

After the phase fraction γ in each subcycle is updated, a corresponding mass flux $F_{\mathrm{sc},i}$ through cell faces is calculated.

The total mass flux F corresponding to the global time step, which is needed in the momentum equation, is then obtained from

$$F = \rho \mathbf{U}_f \cdot \mathbf{S}_f = \sum_{i=1}^{n_{\mathrm{sc}}} \frac{\Delta t_{\mathrm{sc}}}{\Delta t} F_{\mathrm{sc},i} \tag{121}$$

In addition to providing a more accurate solution of the phase fraction equation, this algorithm also enables the global time step size to be greater for the solution of other transport equations, thereby considerably speeding up the solution procedure".

PISO loop In order to achieve a properly coupling between velocity and pressure in `interFoam` is necessary to adapt the PISO loop to the momentum equation for interphase solver and derive a new pressure equation. Starting with momentum equation (113) we have:

$$\frac{\rho\mathbf{U}}{\partial t} + \nabla\cdot(\rho\mathbf{U}\mathbf{U}) - \nabla\cdot(\mu\nabla\mathbf{U}) - (\nabla\mathbf{U})\cdot\nabla\gamma = -\nabla p_d - \mathbf{g}\cdot\mathbf{x}\nabla\rho + \sigma\kappa\nabla\gamma$$

Now following the guidelines given in Section 2.1.12 we can obtain a discretized form of momentum equation (compare it with Eq. 93)

$$a_P^{\mathbf{u}}\mathbf{u}_P = \mathbf{H}(\mathbf{u}) - \nabla p_d - \mathbf{g}\cdot\mathbf{x}\nabla\rho + \sigma\kappa\nabla\gamma \tag{122}$$

isolating the velocity at cell centres:

$$\mathbf{u}_P = \left[a_P^{\mathbf{u}}\right]^{-1}\{[\mathbf{H}(\mathbf{u}) - \mathbf{g}\cdot\mathbf{x}\nabla\rho + \sigma\kappa\nabla\gamma] - \nabla p_d\} \tag{123}$$

replacing this velocity in continuity equation is possible to assemble a Poisson equation for pressure p_d

$$\nabla\cdot\left\{\left[a_P^{\mathbf{u}}\right]^{-1}\nabla p_d\right\} = \nabla\cdot\left\{\left[a_P^{\mathbf{u}}\right]^{-1}[\mathbf{H}(\mathbf{u}) - \mathbf{g}\cdot\mathbf{x}\nabla\rho + \sigma\kappa\nabla\gamma]\right\} \tag{124}$$

Finally it is necessary to obtain fluxes that obeys continuity, it is achieved by

$$F = -\left(a_P^{\mathbf{u}}\right)^{-1}\mathbf{s}_f\cdot\nabla p_d + \left(a_P^{\mathbf{u}}\right)^{-1}\mathbf{s}_f\cdot[\mathbf{H}(\mathbf{u}) - \mathbf{g}\cdot\mathbf{x}\nabla\rho + \sigma\kappa\nabla\gamma] \tag{125}$$

3 OpenFOAM library overview. Solver examples

3.1 Fields and variables. Discrete differential operators

Computational implementation of above explained discretisation is carried out by means of OpenFOAM C++ libraries. These libraries can be used for general purposes like field handling, postproccesing and calculus. Libraries are accompanied by a huge solver set modelling different physical phenomena.

A brief description of this libraries was published in [15], further information can be found in OpenFOAM User Guide chapter 3 [16] and Programmer's Guide chapters 1-2 [17] whose reading is highly encouraged.

Following are several excerpts from Weller *et. al.* paper with the aim to introduce in OpenFOAM libraries. Can be some differences between actual notation and sources's notation because of the natural evolving of the software.

3.1.1 Implementation of tensor fields

"The majority of fluid dynamics can be described using the tensor calculus of up to rank 2, i.e., scalars, vectors, and second-rank tensors. Therefore three basic classes have been created: `scalarField`, `vectorField`, and `tensorField`. [...]

These tensor field classes are somewhat different from a mathematical tensor field in that they contain no positional information; they are essentially ordered lists of tensors, and so only pointwise operations (i.e., tensor algebra) can be performed at this level. The operators implemented [in OpenFOAM] include addition and subtraction, multiplication by scalars, formation of various inner products, and the vector and outer products of vectors (resulting in vectors and tensors, respectively). In addition, operations such as taking the trace and determinant of a tensor are included as well as functions to obtain the eigenvalues and eigenvectors; these are not necessary for solution of fluid systems but are of importance for postprocessing the data [...].

Since C++ implements operator overloading, it is possible to make the tensor algebra resemble mathematical notation by overloading +, -, *, etc. The one problem inherent here is that the precedence of the various operators is preset, which makes it quite difficult to find an operator for the dot product with the correct precedence and that looks correct.

The next level of tensors are referred to as '*geometric tensor fields*' and contain the positional information lacking in the previous classes. Again, there are classes for the three ranks of tensors currently implemented, `volScalarField`, `volVectorField`, and `volTensorField`. At first, the relationship between, for example, `scalarField` and volScalarField should be [...], derivation. However, this would allow the compiler to accept `scalarField1+volScalarField` as an operation, which would not be appropriate, and so encapsulation is used instead. In addition to the additional metrical information necessary to perform differentiation, which is contributed by a reference to a 'mesh class' `fvMesh` (see below), these classes contain boundary information, previous time steps necessary for the temporal discretization, and dimension set information. All seven base SI dimensions are stored, and all algebraic expressions implemented above this level are dimensionally checked at execution. It is therefore impossible to execute a dimensionally incorrect expression in [OpenFOAM]. This has no significant runtime penalty whatsoever: typical fields have 10^4-10^5 tensors in them, and dimension checking is done once per field operation.

Currently two types of tensor-derivative classes are implemented in [OpenFOAM]: `finiteVolumeCalculus` *or* `fvc`, *which performs an explicit evaluation from predetermined data and returns a geometric tensor field, and* `finiteVolumeMethod` *or* `fvm`, *which returns a matrix representation of the operation, which can be solved to advance the dependent variable(s) by a time step*. `fvm` will be described later in more detail [...]. The `fvc` class has no private data and merely implements static member functions that map from one tensor field to another. Use of a static class in this manner mimics the concept of a namespace [...], and by implementing the operations in this manner, a clear distinction is drawn between the data and the operations on the data. The member functions of this class implement the finite-volume equivalent of various differential operators, for example, the expression

```
vorticity = 0.5*fvc::curl(U);
```

calculates the vorticity of a vector field $\mathbf{U}$ as $\frac{1}{2}\nabla \times \mathbf{U}$. (For reasons of space, not all the variables in the program fragments used to illustrate points will be defined. The names are,

however, usually quite descriptive). This also illustrates the ease with which [OpenFOAM] can be used to manipulate tensorial data as a postprocessing exercise. Any [OpenFOAM] code can be thought of as an exercise in mapping from one tensor field to another, and it matters little whether the mapping procedure involves the solution of a differential equation or not. Hence, writing a short code to calculate the vorticity of a vector field is a matter of reading in the data (for which other functions, not described here, are provided), performing this manipulation, and writing out the results. Very complicated expressions can be built up in this way with considerable ease.

All possible tensorial derivatives are implemented in [OpenFOAM]: $\partial/\partial t$, $\nabla\bullet$, ∇ and $\nabla\times$. In addition, the Laplacian operator is implemented independently rather than relying on the use of ∇ followed by $\nabla\bullet$. This enables improved discretization practices to be used for this operator. The one numerical issue that has to be dealt with at the top level of the code is the choice of differencing scheme to be used to calculate the derivative. Again, because of the data hiding in OOP, the numerics can be effectively divorced from the high-level issues of modeling: improved differencing schemes can be implemented and tested separately from the codes that they will eventually be used in. The choice can be made at the modeling level by using a switch in the operator. Hence, the temporal derivative $\partial/\partial t$ can be invoked as

```
volVectorField dUdt = fvc::ddt(U, EI)
```

where the second entry specifies which differencing scheme to use (in this case Euler implicit). Several temporal differencing schemes are available, with a default corresponding to the scheme that gets the most use, in this case, backward differencing. Other selection methods are possible, but this one is the simplest.

3.1.2 Implementation of partial-differential-equation classes

The `fvc` *methods correspond directly to tensor differential operators, since they map tensor fields to tensor fields*. [CFD] requires the solution of partial differential equations, which is accomplished by converting them into systems of difference equations by linearizing them and applying discretization procedures. The resulting matrices are inverted using a suitable matrix solver.

The differential operators $\nabla\bullet$, ∇, and $\nabla\times$ lead to sparse matrices, which for unstructured meshes have a complex structure requiring indirect addressing and appropriate solvers. [OpenFOAM] currently uses the conjugate-gradient method, with incomplete Cholensky preconditioning (ICCG), to solve symmetric matrices. For asymmetric matrices the Bi-CGSTAB method is used[23]. *The matrix inversion is implemented using face addressing throughout, a method in which elements of the matrix are indexed according to which cell face they are associated with. Both transient and steady-state solutions of the equation systems are obtained by time-marching, with the time step being selected to guarantee diagonal dominance of the matrices, as required by the solvers*[24].

In order that standard mathematical notation can be used to create matrix representations of a differential equation, classes of equation object called [`fvScalarMatrix`], [`fvVectorMatrix`],

[23] See original paper cites for details of these methods

[24] Face addressing is a key concept in OpenFOAM because relates mesh generation and description, matrix element indexing and solving

etc., are defined to handle addressing issues, storage allocation, solver choice, and the solution. These classes store the matrices that represent the equations. The standard mathematical operators + and - are overloaded to add and subtract matrix objects. In addition, all the tensorial derivatives $\partial/\partial t$, $\nabla\bullet$, $\nabla\times$, etc., are implemented as member functions of a class `finiteVolumeMethod` (abbreviated to fvm), *which construct appropriate matrices using the finite-volume discretization*. Numerical considerations are relevant in deciding the exact form of many of the member functions. For instance, in the FVM, divergence terms are represented by surface integrals over the control volumes δV_i. Thus the divergence function call is `div(phi,Q)`, *where* `phi` *is the flux, a field whose values are recorded on the cell faces, and* `Q` *is the quantity being transported by the flux, and is a field whose values are on the cell centers*. For this reason, this operation cannot be represented as a function call of the form `div(phi*Q)`. *Again, the Laplacian operator is implemented as a single separate call rather than as calls to* `div` *and* `grad`, *since its numerical representation is different*. Various forms of source term are also implemented. A source term can be explicit, in which case it is a special kind of equation object with entries only in the source vector [...], or it can be made implicit, with entries in the matrix [...]. Construction of an explicit source term is provided for by further overloading + (and -) to provide operations such as `fvm+volScalarField`. Construction of an implicit source is arranged by providing a function `Sp(a,Q)`, thus specifying the dependent variable `Q`to be solved for.

Thus it is possible to build up the matrix system appropriate to any equation by summing the individual terms in the equation. As an example, consider the mass conservation equation $\partial/partialt + \nabla\bullet\left(\phi\right) = 0$, where $\phi = \rho\mathbf{U}$. The matrix system can be assembled by writing

```
fvMatrixScalar rhoEq
(
fvm::ddt(rho) + fvc::div(phi)
);
```

where the velocity flux `phi` *has been evaluated previously*,and solved by the call

```
rhoEq.solve( );
```

to advance the value of ρ by one timestep. Where necessary, the solution tolerance can be explicitly specified. For completeness, the operation == is defined to represent mathematical equality between two sides of an equation. *This operator is here entirely for stylistic reasons, since the code automatically rearranges the equation (all implicit terms go into the matrix, and all explicit terms contribute to the source vector)*. In order for this to be possible, the operator chosen must have the lowest priority, which is why == was used; this also emphasizes that this represents equality of the equation, not assignment.

3.1.3 Mesh topology and boundary conditions

Geometric information is contributed to the geometric fields by the class `fvMesh`, *which consists of a list of vertices, a list of internal cells, and a list of boundary patches (which in turn are lists of cell faces)*. The vertices specify the mesh geometry, whereas the topology of any cell–be it one dimension (1D) (a line), two dimensions (2D) (a face), or three dimensions (3D) (a cell)–is specified as an ordered list of the indices together with a shape primitive describing the relationship between the ordering in the list and the vertices in the shape. These primitive shapes are defined at run time, and so the range of primitive shapes can be extended

with ease, although the 3D set `tetrahedron` (four vertices), `pyramid` (five vertices), `prism` (six vertices), and `hexahedron` (eight vertices) cover most eventualities. In addition, each *n*-dimensional primitive shape knows about its decomposition into (*n*-1)-dimensional shapes, which are used in the creation of addressing lists as, for example, cell-to-cell connectivity.

Boundary conditions are regarded as an integral part of the field rather than as an added extra. `fvMesh` *incorporates a set of patches that define the exterior boundary ∂D of the domain*. Every patch carries a boundary condition, which is dealt with by every `fvm` operator in an appropriate manner. Different classes of patch treat calculated, fixed value, fixed gradient, zero gradient, symmetry, cyclic, and other boundary conditions, all of which are derived from a base class `patchField`. All boundary conditions have to provide the same types of information, that is, that they have the same interface but different implementations. This is therefore a good example of polymorphism within the code. From these basic elements, boundaries suitable for inlets, outlets, walls, etc., can be devised for each specific situation. An additional patchField, processor is also available".

3.2 Solver examples

3.2.1 Scalar transport equation

One of the simplest solvers and a good starting point in OpenFOAM solver comprehension is `scalarTransportFoam`, this solver allows solving an unsteady scalar advection-diffusion equation such as:

$$\frac{\partial C}{\partial t} + \nabla \cdot (\mathbf{U}\, C) - \Gamma \nabla^2 C = 0 \tag{126}$$

where C is the scalar concentration, $\mathbf{U}$ the advective velocity field and Γ the diffusivity. In this case the solver's code is:

```
1  #include "fvCFD.H"
2
3  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
4
5  int main(int argc, char *argv[])
6  {
7
8  #   include "setRootCase.H"
9
10 #   include "createTime.H"
11 #   include "createMesh.H"
12 #   include "createFields.H"
13
14
15 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
16
17     Info<< "\nCalculating scalar transport\n" << endl;
18
19 #   include "CourantNo.H"
20
21     for (runTime++; !runTime.end(); runTime++)
22     {
23         Info<< "Time = " << runTime.timeName() << nl << endl;
24
25 #       include "readSIMPLEControls.H"
26
```

```
27          for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
28          {
29              solve
30              (
31                  fvm::ddt(T)
32                + fvm::div(phi, T)
33                - fvm::laplacian(DT, T)
34              );
35          }
36
37          runTime.write();
38      }
39
40      Info<< "End\n" << endl;
41
42      return(0);
43  }
44
45
46
47  // ************************************************************************* //
48
49  // createFields.H
50
51      Info<< "Reading field T\n" << endl;
52
53      volScalarField T
54      (
55          IOobject
56          (
57              "T",
58              runTime.timeName(),
59              mesh,
60              IOobject::MUST_READ,
61              IOobject::AUTO_WRITE
62          ),
63          mesh
64      );
65
66
67      Info<< "Reading field U\n" << endl;
68
69      volVectorField U
70      (
71          IOobject
72          (
73              "U",
74              runTime.timeName(),
75              mesh,
76              IOobject::MUST_READ,
77              IOobject::AUTO_WRITE
78          ),
79          mesh
80      );
81
82
83      Info<< "Reading transportProperties\n" << endl;
84
85      IOdictionary transportProperties
86      (
```

```
87          IOobject
88          (
89              "transportProperties",
90              runTime.constant(),
91              mesh,
92              IOobject::MUST_READ,
93              IOobject::NO_WRITE
94          )
95      );
96
97
98      Info<< "Reading diffusivity D\n" << endl;
99
100     dimensionedScalar DT
101     (
102         transportProperties.lookup("DT")
103     );
104
105 #   include "createPhi.H"
106
107
108 // ************************************************************************* //
109
110 // createPhi.H
111
112 #ifndef createPhi_H
113 #define createPhi_H
114
115 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
116
117 Info<< "Reading/calculating face flux field phi\n" << endl;
118
119 surfaceScalarField phi
120 (
121     IOobject
122     (
123         "phi",
124         runTime.timeName(),
125         mesh,
126         IOobject::READ_IF_PRESENT,
127         IOobject::AUTO_WRITE
128     ),
129     linearInterpolate(U) & mesh.Sf()
130 );
131
132 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
133
134 #endif
```

Let's comment the code:

- Line 1: `fvCFD.H` is included with the aim of have available all the FVM machinery.
- Lines 2-11: starts the main function. Command line parameters, time variables and mesh set up.
- Line 12: `createFields.H` reads the initial conditions for $\mathbf{U}$, T and diffusivity DT (Γ) (see code in lines 47-107). This file include at the end `createPhi.H` (lines 108-134), read (if it's present) or calculate the face flux field ϕ.

- Line 17: Shows a message by standard output indicating that calculation begins.
- Line 19: `CourantNo.H` is included it calculates and outputs the mean and maximum Courant Numbers.
- Lines 21-38: temporal main cycle controlled by `runTime` object.
- Line 23: now within the temporal cycle, this commands prints the actual time.
- Line 25: `readSIMPLEControls.H` is included, allowing in this case to read the number of non-orthogonal corrections.
- Lines 27-35: As was expressed in Section 2.1.4, paragraph Diffusion Term, laplacian term is discretized by means of face gradient. This discretizacion requires non-orthogonal corrections in case of non orthogonal mesh is used. Then this loop applies this correction as many times as is indicated by the `nNonOrthCorr` variable. Note that the function `solve` solves every time the same problem. It requires some explanation, `solve(fvm::ddt(T)+fvm::div(phi, T)-fvm::laplacian(DT, T));` implies that three systems of equations are assembled, one for each term. Every system has it own matrix, guess values and r.h.s. values. Because `fvm` methods only contributes to matrix and T is the guess field, these terms when are added form a system of equations with zero r.h.s. Right hand side remains zero because == operator hasn't been used indicating no source term. In each temporal step r.h.s. starts being zero, but we have n steps of orthogonal corrections, then as was expressed in Section 2.1.4, paragraph Diffusion Term and in section 2.1.7, orthogonal correction is applied by means of a contribution to source term from the face gradient (calculated by previous step values). So in each non-orthogonal correction step the r.h.s. changes forcing the system to a correct diffusion term calculation[25].
- Line 37: this line indicates writing the fields to hard disk.
- Line 39-43: finally program shows a message by standard output indicating end of calculation and returns the control to the system.

3.2.2 Isothermal, incompressible and laminar Navier-Stokes solver

As our next step in complexity let's study the isothermal, incompressible and laminar Navier-Stokes solver, namely `icoFoam`, this solves the system of equations:

$$\nabla \cdot \mathbf{U} = 0$$
$$\frac{\partial \mathbf{U}}{\partial t} + \nabla \cdot (\mathbf{U}\mathbf{U}) - \nabla \cdot (\nu \nabla \mathbf{U}) = -\nabla p$$

as was explained in Sections 2.1.8-2.1.12 using the PISO algorithm for unsteady calculations.

The code is that follows:

```
1 #include "fvCFD.H"
2
3 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
```

[25] See Section 3.1 of [17] for an excellent example of non-orthogonal correction

```
4
5  int main(int argc, char *argv[])
6  {
7
8  #   include "setRootCase.H"
9  #   include "createTime.H"
10 #   include "createMesh.H"
11 #   include "createFields.H"
12 #   include "initContinuityErrs.H"
13
14 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
15
16     Info<< "\nStarting time loop\n" << endl;
17
18     for (runTime++; !runTime.end(); runTime++)
19     {
20         Info<< "Time = " << runTime.timeName() << nl << endl;
21
22 #       include "readPISOControls.H"
23 #       include "CourantNo.H"
24
25         fvVectorMatrix UEqn
26         (
27             fvm::ddt(U)
28           + fvm::div(phi, U)
29           - fvm::laplacian(nu, U)
30         );
31
32         solve(UEqn == -fvc::grad(p));
33
34         // --- PISO loop
35
36         for (int corr=0; corr<nCorr; corr++)
37         {
38             volScalarField rUA = 1.0/UEqn.A();
39
40             U = rUA*UEqn.H();
41             phi = (fvc::interpolate(U) & mesh.Sf())
42                 + fvc::ddtPhiCorr(rUA, U, phi);
43
44             adjustPhi(phi, U, p);
45
46             for (int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
47             {
48                 fvScalarMatrix pEqn
49                 (
50                     fvm::laplacian(rUA, p) == fvc::div(phi)
51                 );
52
53                 pEqn.setReference(pRefCell, pRefValue);
54                 pEqn.solve();
55
56                 if (nonOrth == nNonOrthCorr)
57                 {
58                     phi -= pEqn.flux();
59                 }
60             }
61
62 #           include "continuityErrs.H"
63
```

```
64              U -= rUA*fvc::grad(p);
65              U.correctBoundaryConditions();
66          }
67
68          runTime.write();
69
70          Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
71              << " ClockTime = " << runTime.elapsedClockTime() << " s"
72              << nl << endl;
73      }
74
75      Info<< "End\n" << endl;
76
77      return(0);
78  }
79
80  // ************************************************************************* //
81
82  // createFields.H
83
84    Info<< "Reading transportProperties\n" << endl;
85
86      IOdictionary transportProperties
87      (
88          IOobject
89          (
90              "transportProperties",
91              runTime.constant(),
92              mesh,
93              IOobject::MUST_READ,
94              IOobject::NO_WRITE
95          )
96      );
97
98      dimensionedScalar nu
99      (
100         transportProperties.lookup("nu")
101     );
102
103     Info<< "Reading field p\n" << endl;
104     volScalarField p
105     (
106         IOobject
107         (
108             "p",
109             runTime.timeName(),
110             mesh,
111             IOobject::MUST_READ,
112             IOobject::AUTO_WRITE
113         ),
114         mesh
115     );
116
117
118     Info<< "Reading field U\n" << endl;
119     volVectorField U
120     (
121         IOobject
122         (
123             "U",
```

```
124             runTime.timeName(),
125             mesh,
126             IOobject::MUST_READ,
127             IOobject::AUTO_WRITE
128         ),
129         mesh
130     );
131
132 #   include "createPhi.H"
133
134     label pRefCell = 0;
135     scalar pRefValue = 0.0;
136     setRefCell(p, mesh.solutionDict().subDict("PISO"), pRefCell, pRefValue);
```

Respecto to this code we have:

- Lines 1-15: the initialization is similar to `scalarTransportFoam` but in this case `initContinuityErrs.H` file is included to initilialize the calculations of error and in `createFields.H` reference pressure is set (see lines 134-136).
- Line 16: a message indicating the starting of time loop is sent to standard output.
- Lines 18-73: temporal loop controlled by `runTime` object.
- Line 20: a message indicating the actual time is sent to standard output.
- Line 22: the inclusion of `readPISOControls.H` archive allows reading the PISO solver parameters, in this case specially the number of non-orthogonal corrections (`nNonOrthCorr`) in pressure equation solving and the number of PISO loops (`nCorr`).
- Line 23: Courant number is calculated for actual step.
- Lines 25-32: Navier-Stokes equations are solved for $\mathbf{U}$ using ϕ and p from a previous (or initial) step. Firstly left hand side of equation is assembled and then `solve(UEqn == -fvc::grad(p));` sentence is used meaning the right hand side is the face gradient of p, this is explicitely calculated because a previous p field value is used. So, `fvm` operators contribute to system matrix and `fvc` to the source term. Solving this system is called the *momentum predictor*. "This gives us a velocity field that is not divergence free, but aproximately satisfies momentum" [18].
- Lines 34-66: PISO loop is performed (see Section 2.1.12) (check Section 2.1.12 paragraph Derivation of the pressure equation). This loop is performed as many times as indicated by the `nCorr` variable. As is indicated in this section we have to calculate the reciprocal of diagonal coefficients `UEqn` realated matrix, to use it in later calculus, so in line 38 we have: `volScalarField rUA = 1.0/UEqn.A();` where have made use of `UEqn.A()` method to extract the diagonal coefficients.
  The next step is to solve the Pressure Equation created specially for this incompressible problem or Eq. (95), this is achieved by several intermediate steps. First of all the part between parenthesis of r.h.s in Eq. (95) is calculated in line 40, using the method `UEqn.H()` to extract the off-diagonal part of the `UEqn` associated matrix. Here is important to note that by Eq. (17), the divergence is calculated as a sum of face fluxes, then it is important to give to this operator an appropriate face flux. This is calculated in lines 41-42, 44.; `fvc::interpolate(U) & mesh.Sf()` recalls the standard calculation of ϕ (or F by Jasak's Thesis nomenclature), like in `createPhi.H`, but in this case we can

apply different interpolation schemes indicated in `system/fvSchemes` file. Another term is added to ϕ calculation, `fvc::ddtPhiCorr(rUA, U, phi)` which "accounts for the divergence of the face flux of the face velocity field by taking out the difference between the interpolated velocity and the flux" [18][26]
In line 44 there is another correction, in this case with the aim of "[adjusting] the inlet and outlet fluxes to obey continuitym, which is neccesary for creating a well-posed problem where a solution for pressure exists" [18].
Now is possible to solve Eq. (95) (line 54), but being involved a gradient calculation is newly necessary to do the non-orthogonal corrections. Another thing to do is set the reference pressure (line 53) before solving.
Once non-orthogonal correction loop is finalized ϕ flux is corrected by p as is indicated in equation (98), then continuity errors are calculated and reported in line 62.
Now it is possible to refresh **U** by Eq. (94) (line 64) and correct boundary field of this this `volVectorField` (line 65).

- Lines 68-78: program finalizes writing data to disk and execution times to standard output.

3.2.3 Volume of Fluid laminar solver

Having analized previous solvers it is time to face the description of `interFoam` solver. Previuos work is worthy because for VOF method is necessary to solve scalar transport equations and manage pressure-velocity coupling in momentum equations solving.

Thus, having in mind this theory and code, we recall section 2.2 to analize the following code:

```
1  #include "fvCFD.H"
2  #include "MULES.H"
3  #include "subCycle.H"
4  #include "interfaceProperties.H"
5  #include "twoPhaseMixture.H"
6
7  // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
8
9  int main(int argc, char *argv[])
10 {
11     #include "setRootCase.H"
12     #include "createTime.H"
13     #include "createMesh.H"
14     #include "readEnvironmentalProperties.H"
15     #include "readPISOControls.H"
16     #include "initContinuityErrs.H"
17     #include "createFields.H"
18     #include "readTimeControls.H"
19     #include "correctPhi.H"
20     #include "CourantNo.H"
21     #include "setInitialDeltaT.H"
22
23 // * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * * //
24
25     Info<< "\nStarting time loop\n" << endl;
```

[26] See also: http://www.cfd-online.com/Forums/openfoam-solving/60096-ddtphicorr.html and http://www.cfd-online.com/Forums/openfoam-solving/59636-why-say-uses-e2-80-98pseudostaggered-e2-80-99-finite-volume-numerics.html for replies on this topic by Henry Weller and Hrvoje Jasak.

```
26
27      while (runTime.run())
28      {
29          #include "readPISOControls.H"
30          #include "readTimeControls.H"
31          #include "CourantNo.H"
32          #include "setDeltaT.H"
33
34          runTime++;
35
36          Info<< "Time = " << runTime.timeName() << nl << endl;
37
38          twoPhaseProperties.correct();
39
40          #include "gammaEqnSubCycle.H"
41
42          #include "UEqn.H"
43
44          // --- PISO loop
45          for (int corr=0; corr<nCorr; corr++)
46          {
47              #include "pEqn.H"
48          }
49
50          #include "continuityErrs.H"
51
52          p = pd + rho*gh;
53
54          runTime.write();
55
56          Info<< "ExecutionTime = " << runTime.elapsedCpuTime() << " s"
57              << " ClockTime = " << runTime.elapsedClockTime() << " s"
58              << nl << endl;
59      }
60
61      Info<< "End\n" << endl;
62
63      return(0);
64  }
65
66
67  // ************************************************************************* //
68
69  // createFields.H
70
71     Info<< "Reading field pd\n" << endl;
72      volScalarField pd
73      (
74          IOobject
75          (
76              "pd",
77              runTime.timeName(),
78              mesh,
79              IOobject::MUST_READ,
80              IOobject::AUTO_WRITE
81          ),
82          mesh
83      );
84
85      Info<< "Reading field gamma\n" << endl;
```

```
86      volScalarField gamma
87      (
88          IOobject
89          (
90              "gamma",
91              runTime.timeName(),
92              mesh,
93              IOobject::MUST_READ,
94              IOobject::AUTO_WRITE
95          ),
96          mesh
97      );
98
99      Info<< "Reading field U\n" << endl;
100     volVectorField U
101     (
102         IOobject
103         (
104             "U",
105             runTime.timeName(),
106             mesh,
107             IOobject::MUST_READ,
108             IOobject::AUTO_WRITE
109         ),
110         mesh
111     );
112
113 #   include "createPhi.H"
114
115
116     Info<< "Reading transportProperties\n" << endl;
117     twoPhaseMixture twoPhaseProperties(U, phi, "gamma");
118
119     const dimensionedScalar& rho1 = twoPhaseProperties.rho1();
120     const dimensionedScalar& rho2 = twoPhaseProperties.rho2();
121
122
123     // Need to store rho for ddt(rho, U)
124     volScalarField rho
125     (
126         IOobject
127         (
128             "rho",
129             runTime.timeName(),
130             mesh,
131             IOobject::READ_IF_PRESENT
132         ),
133         gamma*rho1 + (scalar(1) - gamma)*rho2,
134         gamma.boundaryField().types()
135     );
136     rho.oldTime();
137
138
139     // Mass flux
140     // Initialisation does not matter because rhoPhi is reset after the
141     // gamma solution before it is used in the U equation.
142     surfaceScalarField rhoPhi
143     (
144         IOobject
145         (
```

```
            "rho*phi",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::NO_WRITE
        ),
        rho1*phi
    );


    Info<< "Calculating field g.h\n" << endl;
    volScalarField gh("gh", g & mesh.C());
    surfaceScalarField ghf("gh", g & mesh.Cf());


    volScalarField p
    (
        IOobject
        (
            "p",
            runTime.timeName(),
            mesh,
            IOobject::NO_READ,
            IOobject::AUTO_WRITE
        ),
        pd + rho*gh
    );


    label pdRefCell = 0;
    scalar pdRefValue = 0.0;
    setRefCell(pd, mesh.solutionDict().subDict("PISO"), pdRefCell, pdRefValue);


    // Construct interface from gamma distribution
    interfaceProperties interface(gamma, U, twoPhaseProperties);

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

// correctPhi.H


{
#   include "continuityErrs.H"

    wordList pcorrTypes(pd.boundaryField().types());

    for (label i=0; i<pd.boundaryField().size(); i++)
    {
        if (pd.boundaryField()[i].fixesValue())
        {
            pcorrTypes[i] = fixedValueFvPatchScalarField::typeName;
        }
    }

    volScalarField pcorr
    (
        IOobject
        (
            "pcorr",
```

```
206             runTime.timeName(),
207             mesh,
208             IOobject::NO_READ,
209             IOobject::NO_WRITE
210         ),
211         mesh,
212         dimensionedScalar("pcorr", pd.dimensions(), 0.0),
213         pcorrTypes
214     );
215
216     dimensionedScalar rUAf("(1|A(U))", dimTime/rho.dimensions(), 1.0);
217
218     adjustPhi(phi, U, pcorr);
219
220     for(int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
221     {
222         fvScalarMatrix pcorrEqn
223         (
224             fvm::laplacian(rUAf, pcorr) == fvc::div(phi)
225         );
226
227         pcorrEqn.setReference(pdRefCell, pdRefValue);
228         pcorrEqn.solve();
229
230         if (nonOrth == nNonOrthCorr)
231         {
232             phi -= pcorrEqn.flux();
233         }
234     }
235
236 #   include "continuityErrs.H"
237 }
238
239 // ************************************************************************* //
240
241 // gammaEqnSubCycle.H
242
243
244 label nGammaCorr
245 (
246     readLabel(piso.lookup("nGammaCorr"))
247 );
248
249 label nGammaSubCycles
250 (
251     readLabel(piso.lookup("nGammaSubCycles"))
252 );
253
254 if (nGammaSubCycles > 1)
255 {
256     dimensionedScalar totalDeltaT = runTime.deltaT();
257     surfaceScalarField rhoPhiSum = 0.0*rhoPhi;
258
259     for
260     (
261         subCycle<volScalarField> gammaSubCycle(gamma, nGammaSubCycles);
262         !(++gammaSubCycle).end();
263     )
264     {
265 #       include "gammaEqn.H"
```

```
266         rhoPhiSum += (runTime.deltaT()/totalDeltaT)*rhoPhi;
267     }
268
269     rhoPhi = rhoPhiSum;
270 }
271 else
272 {
273 #       include "gammaEqn.H"
274 }
275
276 interface.correct();
277
278 rho == gamma*rho1 + (scalar(1) - gamma)*rho2;
279
280
281 // ************************************************************************* //
282
283 // UEqn.H
284
285   surfaceScalarField muf = twoPhaseProperties.muf();
286
287     fvVectorMatrix UEqn
288     (
289         fvm::ddt(rho, U)
290       + fvm::div(rhoPhi, U)
291       - fvm::laplacian(muf, U)
292       - (fvc::grad(U) & fvc::grad(muf))
293     //- fvc::div(muf*(fvc::interpolate(dev(fvc::grad(U))) & mesh.Sf()))
294     );
295
296     if (momentumPredictor)
297     {
298         solve
299         (
300             UEqn
301          ==
302             fvc::reconstruct
303             (
304                 (
305                     fvc::interpolate(interface.sigmaK())*fvc::snGrad(gamma)
306                   - ghf*fvc::snGrad(rho)
307                   - fvc::snGrad(pd)
308                 ) * mesh.magSf()
309             )
310         );
311     }
312
313
314 // ************************************************************************* //
315
316 // pEqn.H
317
318 {
319     volScalarField rUA = 1.0/UEqn.A();
320     surfaceScalarField rUAf = fvc::interpolate(rUA);
321
322     U = rUA*UEqn.H();
323
324     surfaceScalarField phiU
325     (
```

```
326         "phiU",
327         (fvc::interpolate(U) & mesh.Sf()) + fvc::ddtPhiCorr(rUA, rho, U, phi)
328     );
329
330     phi = phiU +
331         (
332             fvc::interpolate(interface.sigmaK())*fvc::snGrad(gamma)
333           - ghf*fvc::snGrad(rho)
334         )*rUAf*mesh.magSf();
335
336     adjustPhi(phi, U, pd);
337
338     for(int nonOrth=0; nonOrth<=nNonOrthCorr; nonOrth++)
339     {
340         fvScalarMatrix pdEqn
341         (
342             fvm::laplacian(rUAf, pd) == fvc::div(phi)
343         );
344
345         pdEqn.setReference(pdRefCell, pdRefValue);
346
347         if (corr == nCorr-1 && nonOrth == nNonOrthCorr)
348         {
349             pdEqn.solve(mesh.solver(pd.name() + "Final"));
350         }
351         else
352         {
353             pdEqn.solve(mesh.solver(pd.name()));
354         }
355
356         if (nonOrth == nNonOrthCorr)
357         {
358             phi -= pdEqn.flux();
359         }
360     }
361
362     U += rUA*fvc::reconstruct((phi - phiU)/rUAf);
363     U.correctBoundaryConditions();
364 }
365
366
367 // ************************************************************************* //
368
369 // gammaEqn.H
370
371 {
372     word gammaScheme("div(phi,gamma)");
373     word gammarScheme("div(phirb,gamma)");
374
375     surfaceScalarField phic = mag(phi/mesh.magSf());
376     phic = min(interface.cGamma()*phic, max(phic));
377     surfaceScalarField phir = phic*interface.nHatf();
378
379     for (int gCorr=0; gCorr<nGammaCorr; gCorr++)
380     {
381         surfaceScalarField phiGamma =
382             fvc::flux
383             (
384                 phi,
385                 gamma,
```

```
386                 gammaScheme
387             )
388           + fvc::flux
389             (
390                 -fvc::flux(-phir, scalar(1) - gamma, gammarScheme),
391                 gamma,
392                 gammarScheme
393             );
394
395         MULES::explicitSolve(gamma, phi, phiGamma, 1, 0);
396
397         rhoPhi = phiGamma*(rho1 - rho2) + phi*rho2;
398     }
399
400     Info<< "Liquid phase volume fraction = "
401         << gamma.weightedAverage(mesh.V()).value()
402         << " Min(gamma) = " << min(gamma).value()
403         << " Max(gamma) = " << max(gamma).value()
404         << endl;
405 }
406
407
```

- Lines 1-6: especific `.H` files are included.

- Lines 9-21: main function begins, case initialization. Particularly `readEnvironmental Properties.H` reads the gravitational acceleration. `createFields.H` includes some additional lines respect other solvers, in lines 116-120 `twoPhaseProperties` object is created with properties read from disk, from this densities are loaded in `rho1` and `rho2`. Another important fields are `gh ghf` accounting the product of distance and gravitational acceleration for further calculations (see lines 156-158). p field is created from relative pressure and fluid column contribution, and reference pressure is set (see lines 161-177). At the end `interface` object is created to hold interface properties such as surface tension, curvature, etc.
  In line 19 `correcPhi.H` file is included (see lines 185-236), here field `phi` previously created is corrected by imposed pressures at boundaries.

- Lines 25-32: after starting messages main loop begins reading controls, calculating Courant number and setting the timestep. Last action is performed by `setDeltaT.H` code, to mantain a constant maximum Courant number, following the steps outlined in section 2.2.3 paragraph Adaptive Time Step Control.

- Lines 34-36: time is avanced and reported by standar output.

- Line 38: `correct()` method of `twoPhaseProperties` object is invoked to calculate the new kinematic viscosity field as a division of Eq. (104) by Eq. (103).

- Line 40: Gamma equation subcycle is performed by `gammaEqnSubCycle.H`. In order to analize this subscycle we continue our analysis in lines 244-278. First of all `nGammaCorr` and `nGammaSubCycles` values are read from PISO section of `./system/fvSolution` dictionary. These parameters indicates the number of corrections in gamma equation (108) necessary to converge to a solution (this is equation is implicit in γ) and the number of subcycles (n_{sc})as was explained in Section 2.2.3. In line 254 if `nGammaSubCycles` is greater tha one the subcycle is started, firstly storing the Δt of Eq. (120) (the main loop

time interval), then F in Eq. (121) is initialized in zero. In lines 254-267 gamma subcycle is formally performed using a `subCycle` object. In this loop Eq. (121) summation is calculated next solving gamma in equation by `gammaEqn.H` code. Finally `rhoPhi` is overwritten by the value calculate in the subcycle. In case if `nGammaSubCycles` is equal to one, `gammaEqn.H` is solved once. As the last thing in the `gammaEqnSubCycle.H`, curvature κ and density field are updated [Eq. 103].
Now looking inside `gammaEqn.H`, it is possible to understand the method used for gamma field advection. Then, let's in lines 372-405. In lines 372-373, words are defined to indicate the divergence schemes to flux calculating methods. Next in line 375, term $\frac{|\phi|}{|\mathbf{s}_f|}$ in Eq. 114 is calculated, them in the next line the expresion of minimum is obtained. C_γ coefficient is obtained by a method of `interface` object. Finally $U_{r,f}$, i.e. the compressive velocity at the faces is obtained in line 377, in this case interface normal (n_f) is obtained by `nHatf()` method of `interface` object. Is important to take in account that `phir` are compressive velocities at faces not *fluxes*.
In order to integrate Eq. (108) an special technique, called MULES[27], is used. This technique requires giving the advective field as a flux. Due this equation is implicit in gamma (velocity fields changes if gamma changes) an iterative solution is performed in lines 379-398. This loop is controlled by `nGammaCorr` that indicates the maximum number of correction loops. To a best understanding of loop's code let's explain the equations implementation.

```
surfaceScalarField phiGamma = fvc::flux(phi, gamma, gammaScheme)
+ fvc::flux(-fvc::flux(-phir, scalar(1) - gamma, gammarScheme), gamma,
gammarScheme);
```

`phiGamma` represents the second and third term in l.h.s. of Eq. (108) without applying the divergence operator, second term is dicretized as `fvc::flux(phi, gamma, gammaScheme)` giving a flux of the product between phi and gamma fields and using the indicated scheme for upwinding. As the next step, the third term is calculated, here `fvc::flux` is called twice due the neccesity of expressing some magnitudes as fluxes before operate with them. gamma wasn't defined as a flux, then in order to operate with it we use `fvc::flux` first to calculate the product $\mathbf{U}_r\ (1-\gamma)$ [`fvc::flux(-phir, scalar(1) - gamma, gammarScheme)`], and then `fvc::flux` is used again the calculate the final product by gamma. Now all quantities are fluxes. Finally MULES is called in order to solve for gamma, giving it the unknow (`gamma`), the overall flux (`phi`), the non-temporal terms (`phiGamma`), and the bounds for the unknow ($0 < \gamma < 1$).
Finally in the loop, the `rhoPhi` field is recovered from `phiGamma` and the densities (line 397). Lines 400-404 give a report by standard output about gamma field.

- Line 42: once gamma equation subcycle is performed is posible to solve the momentum and continuity equations and its coupling via a PISO loop. In this line `UEqn.H` file (lines 285-311) is included. In this file the l.h.s. of Eq. 113 is assembled and if it's indicated a momentum predictor is calculated assembling the r.h.s. and solving for $\mathbf{U}$. `fvc::reconstruct` is used to generate a cell based volumetric field from a face flux field, needed to assemble the system.

[27] "`[interFoam]` uses the multidimensional universal limiter for explicit solution (MULES) method, created by OpenCFD, to maintain boundedness of the phase fraction independent of underlying numerical scheme, mesh structure, etc. The choice of schemes for convection are therfore not restricted to those that are strongly stable or bounded, e.g. upwind differencing"[16].

- Lines 44-48: PISO loop over pressure equation, number of correction loops is indicated by `nCorr` which was read from dictionaries. To perfor this loop lines in `pEqn.H` are executed (lines 316-364). In line 319 the reciprocal of diagonal coefficients (`rUA`)is calculated (see first term of r.h.s in Eq. 123), this magnitude is used often later. Next, `rUA` is calculated at faces via interpolation with central differencing scheme (line 320), this is necessary due some operators needs face values as arguments.
  Now, is necessary to assemble the argument of divergence at r.h.s. of Eq. (124). This argument in two fluxes, one involving only the non-diagonal coefficients of the matrix and another including gravity and surface tension terms. The first one `phiU` is calculated in lines 322-328, including the correction described in `icoFoam` solver, then the extra terms are added in lines 330-334, giving the flux `phi`. This flux is adjust like in `icoFoam` in line 336. Now we have the argument of divergence in r.h.s. of pressure equation. Due it an equation involving a laplacian, which discretized by a gradient, non orthogonal corrections are needed. This correction loop is performed in lines 338-360, as many times as is indicated by `nNonOrthCorr`. Equation for pressure is defined in lines 340-343 as in indicated by Eq. (124). In line 345 reference pressure is set to start the equation solution (lines 347-354). Once pressure equation is solved as many time as was indicated for the non-orthogonal correction loop, new conservative fluxes are assembled as in Eq. (125) in lines 356-359.
  Velocity field is finally recovered in line 362 by means of Eq. (123) that is implemented as:

  ```
  U += rUA*fvc::reconstruct((phi - phiU)/rUAf);
  ```

  note that as for fluxes term $\left[a_P^{\mathbf{u}}\right]^{-1}$ used was calculated at faces (`rUAf`), then after using `reconstruct` method to give the fields at cell centres is necessary to scale the field by `rUA`. In line 363 velocity field obtained is corrected to satisfy boundary conditions.

- Lines 50-58: to finalize the temporal loop continuity errors are printed by standard output, total pressure is calculated, fields are written to hard disk and messages about execution times are presented.

- Lines 61-63. Once temporal loop is finalized End messages is printed by standard output and control is returned to the system.

4 Aplications

4.1 The sloshing problem. General description

As a validation of `interFoam` solver we prepose to the Sloshing Problem (see Figure 8. In this example the problem is to solve the little amplitude movement of viscous fluid an rectangular tank. The initial position of free surface is given by

$$a(x) = 1.5 + a_0 \sin\left[\pi\left(1/2 - x\right)\right] \tag{127}$$

where a_0 is the amplitude of the sinusoidal initial perturbation and x is the coordinate along the free surface of fluid in rest state (dashed line in Figure 8). Movement is driven by gravitational forces and damped by viscous shear. Boundary conditions are *slip* in all over the boundaries. On the free surface pressure is zero. In the example, inferior fluid is named

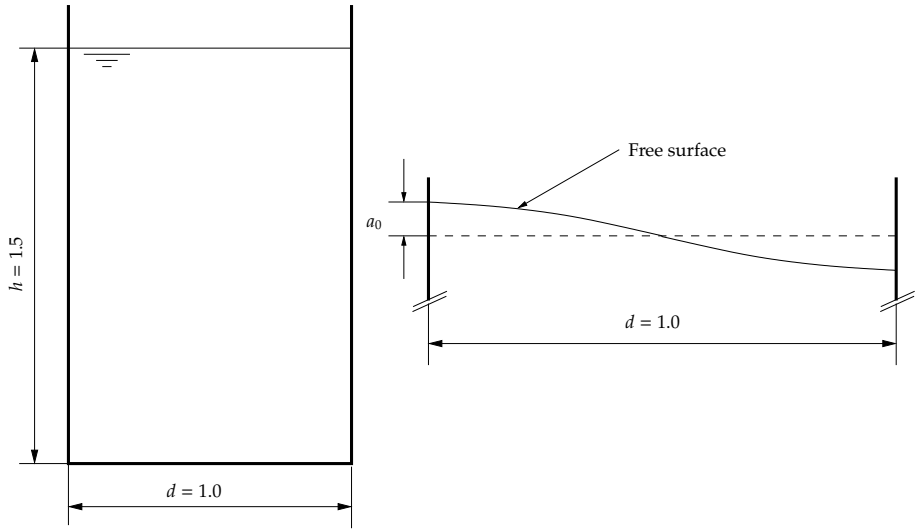


Figure 8: Dimensions and initial position for free surface in sloshing problem

liquid and superior one *gas*, having then a multiphase problem.

The analytical solution for the linearized case was given by Prosperetti [19]

$$a(t) = \frac{4\nu^2 k^2}{8\nu^2 k^4 + \omega_0^2} a_0 \operatorname{erfc}\left(\nu k^2 t\right)^{1/2} + \sum_{i=1}^{4} \frac{z_i}{Z_i} \left(\frac{\omega_0^2 a_0}{z_i^2 - \nu k^2}\right) \exp\left[\left(z_i^2 - \nu k^2\right) t\right] \operatorname{erfc}\left(z_i t^{1/2}\right) \tag{128}$$

where ν is the fluid's kinematic viscosity, k the wave number, $\omega_0^2 = g\,k$ is the inviscid natural frequency, and each z_i is a root of the following algebraic equation:

$$z^4 + k^2 \nu z^2 + 4\left(k^2 \nu^3\right)^{3/2} z + \nu^2 k^4 + \omega_0^2 = 0 \tag{129}$$

where $Z_1 = (z_2 - z_1)(z_3 - z_1)(z_4 - z_1)$, Z_2, Z_3, Z_4 are obtained by circular permutation of the indexes and errfc() is the complex error fuction. This expressios is valid for plane waves of little amplitude in a infinite depth domain.

The example is solved with $a_0 = 0.01$, and $g = 1.0$, all in metric units as is the standard in OpenFOAM. For the fluids, the liquid has $\nu_1 = 0.01$ and $\rho_1 = 1000$ whereas the gas has $\nu_2 = 1.48 \times 10^{-5}$ and $\rho_2 = 1$. Surface tension was negligible.

4.2 Discretization and solving

To solve the problem by OpenFOAM a 2D mesh (really OpenFOAM works ever with 3D meshes, in this case is volumetric mesh with only one layer in perpedicular coordinate) of 12004 points and 11531 cells (triangles and rectangles, really wedges and hexahedron) (see Figures 9.a and 9.b.

Gravity and fluids properties were loaded in `constant/environmetalProperties` and `constant/transportProperties` dictionaries. Repect `system/fvSolution` dictionary, PISO loop subsection was set with 3 corrections (`nCorrectors`) and 4 non-orthogonal corrections (`nNonOrthogonalCorrectors` with the momemtum predictor deactivated. Respect gamma equation parameters in PISO subsection were set in `nGammaCorr=1` and `cGamma=1`.

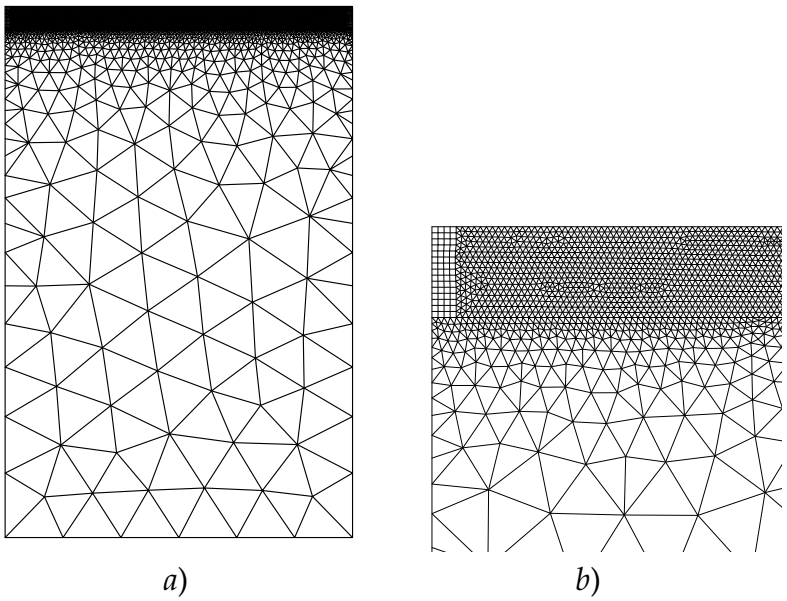


Figure 9: a) Overall view of used mesh, b) Detail of near free surface meshing.

Preconditionated Conjugate Gradient (PCG) solver was used for `pcorr`, `pd` and `pdFinal` and Preconditionated Bi-conjugate Gradient for the velocity. As preconditioners Diagonal Incomplete-Cholesky (DIC) was used for the pressures and Diagonal Incomplete LU (DILU) for velocities. This parameter were take from Dam Break tutorial (including in OpenFOAM suite distribution).

Discrezation schemes were set in `system/fvSchemes` dictionary for gradientes, Gauss Linear Scheme was used, for divergences in term `div(rho*phi,U)` Gauss Limited LinearV was used, in term `div(phi,gamma)` Gauss VanLeer and in `div(phirb,gamma)` Gauss Interface Compression. The scheme for laplacian was set in Gauss Linear Corrected, for interpolation schemes a linear scheme was used and for surface normal gradients (`snGradSchemes`) the corrected scheme was set (See [16] for explanations about the dictionaries and its parameters).

Parameters changed in `system/controlDict` dictionary were times step (`deltaT`), start time (`startTime`, usually 0 if resume is not needed), end time (`endTime`), write interval (`writeInterval`, this parameter was fixed in 0.05), maximum Courant number (`maxCo`) and maximum time interval allowed, (`maxDeltaT`, remember that timestep is adjusted on the fly as was explained in Section 2.2.3, paragraph Adaptive time step control).

Some parameters were changed in order to reproduce the analytical solution. For residuals the default setting were `pcorr`=1e-10, `pd`=1e-10 and relative tolerance 0.05, `pdFinal`=1e-7 and `U`=1e-6.

In Figure 10 first analysis is shown, here the parameter changed was the number subcycles in gamma equation solution. This parameter has little effect in amplitude but not so much in termporal integration.

Second analysis (see Figure 11 allows to analize the effect of changes in residuals. Results shown allows to say that default settings for `interFoam` solver are enough actual purposes.

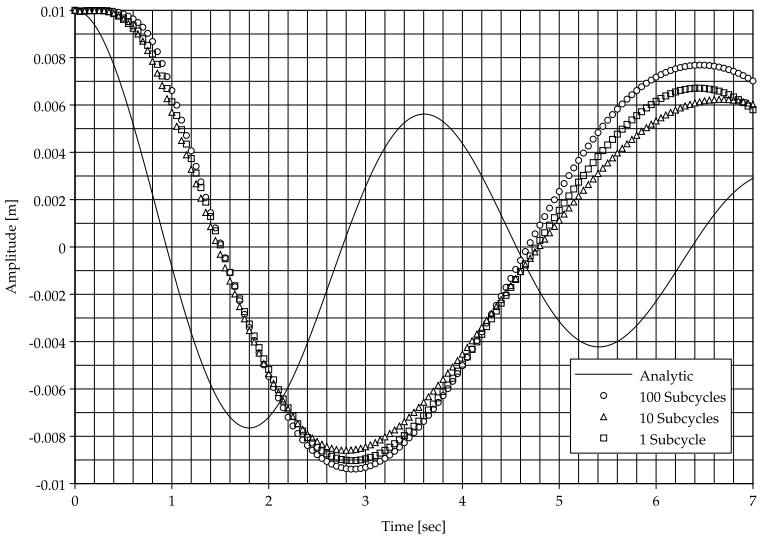


Figure 10: Comparison of analytical solution and different numerical solutions varying gamma subcycles, default setting for residuals were kept, time step: 0.001, maximum Courant number: 0.5 and maximum time step: 1.

Is important to note that residuals were, in all cases, decreased. No test were carried out with greater residuals, but elapsed time for default residuals was acceptable.

In last tests time step and temporal discretization were changed (see Figure 12 and 13). In the first one time step was decreased from the original (0.001) to 10 times smaller, and 100 times smaller, maximum Courant number and maximum time step was decreased by the same ratio, temporal discretization scheme was kept isn in default settings (Backward Euler). As is shown in Figure 12, time step has an important impact in free surface tracking.

Finally to avoid to small timestep (0.001/100 time step gives an acceptable solution but requires relatively longer calculation time), temporal discretization is changed (see Figure 13. For original timestep (0.001) Crank-Nicholson has no effect, so smaller timestep is tested (decreasing maximum Courant number and maximum timestep as was explained) with this scheme in two cases, one with default residuals (d.r.) and another with all residuals set in 1e-10. As is shown in the figure best results of all (from Figure 12 and 13) are obtained with default setting for residuals and Crank-Nicholson scheme (for 0.001/10 time step).

5 Conclusions

In this work a brief description of the Finite Volume Method and Volume of Fluid Method were given. These description is based on bibliography related to the OpenFOAM suite, where results were obtained.

Nowadays OpenFOAM community is growing fast, and this software is starting to be considerated a threaten for commercial companies and an interisting tool for academics. So the importance of a good understanding of this tool is superlative. In this way, this work is a contribution to this comprehension, describing in detail three important solvers as advection-diffusion solver, Navier-Stokes and VOF.

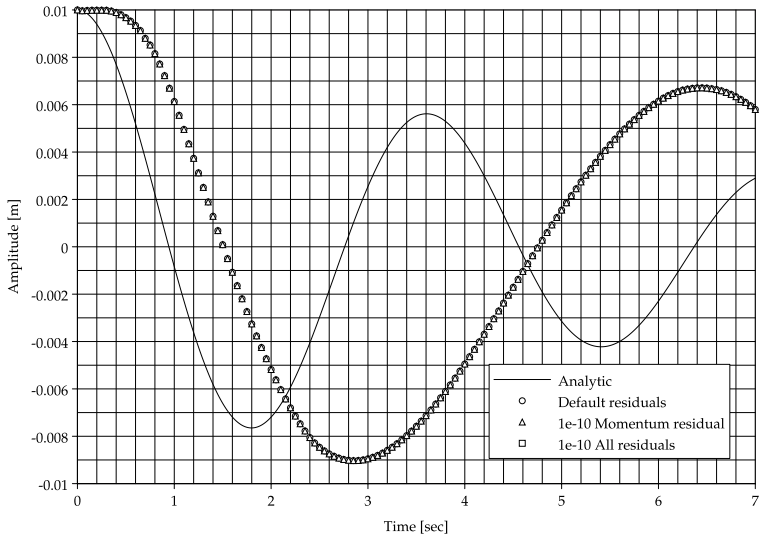


Figure 11: Comparison of analytical solution and different numerical solutions varying residuals from original to change in velocity and all equations residuals. Time step: 0.001, maximum Courant number: 0.5 and maximum time step: 1.

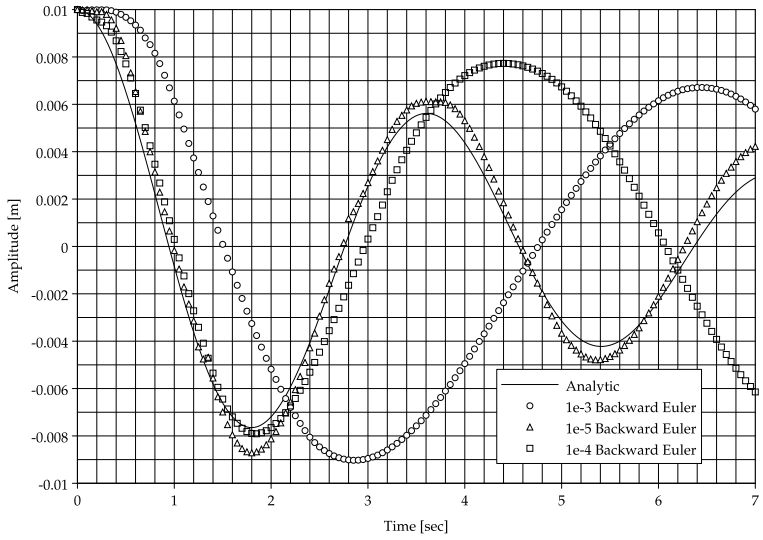


Figure 12: Comparison of analytical solution and different numerical solutions varying times step for Backward Euler discretization scheme (For time step, maximum Courant number and and maximum time step see text).

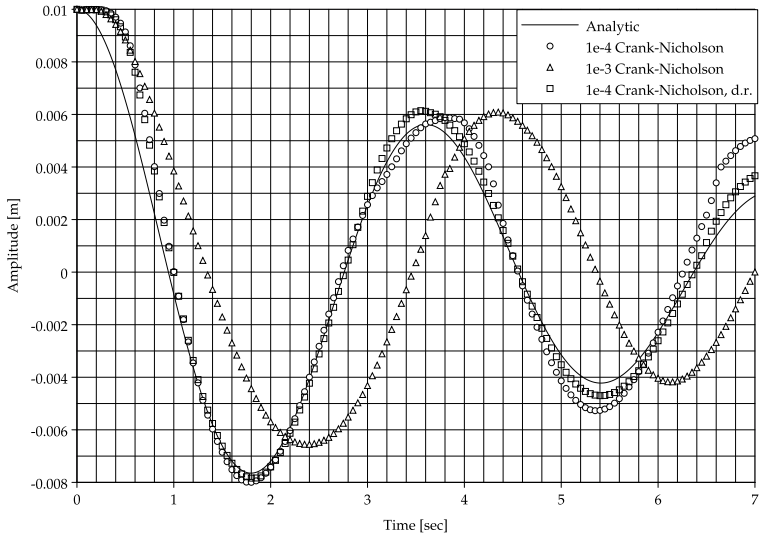


Figure 13: Comparison of analytical solution and different numerical solutions varying timestep and residuals, both for Crank-Nicholson discretization scheme.

Writing this description required a lot research in forums, personal contact with developers and users, reverse engineering and theoretical revising. All this factors are now joined and crosslinked, in order to connect the code with the underlying theory.

As a practical example sloshing problem was solved showing relative concordance with theoretical results. Obviously a lot of another important parameters weren't changed, as the mesh, compressive schemes, corrections in MULES solver, preconditioners, solvers, etc. but it's a good start point, and the intention was only show an example using the examined tool. More research can be done fixing a real problem and working on it.

6 Acknowledgements

I would give my sincere thanks to my advisors Noberto M. Nigro, Mario A. Storti and Damian Ramajo (from CIMEC-INTEC, CONICET/UNL, Argentin) for their support. I want to recognize also free help received from Patricio Bohorquez (Universidad de Malaga, Spain), Ola Widlund (CEA, Grenoble, France), Laurence R. McGlashan (Computational Modelling Group, University of Cambridge), Martin Romagnoli (Universidad Nacional de Rosario, Argentina) and Daniel Wei (Tongji University, Shangai, China) and all contributors in CFD-Online forums, OpenFOAMWiki, OpenFOAM Workshops, etc.

References

[1] http://en.wikipedia.org/wiki/OpenFOAM

[2] http://www.openfoam.org

[3] Ferziger, J.H.; Peric, M. *Computational Methods for Fluid Dynamics*, 1995

[4] Versteeg, H.K.; Malalasekera, W. *An introduction to Computational Fluid Dynamics*, 1st. edition.

[5] 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, London, 1996.

[6] Hirsch, C. *Numerical Computation of internal and external flows*, 1991.

[7] Patankar, S.V. *Numerical Heat Transfer and Fluid Flow*, 1981.

[8] Karrholm, F.P. *Rhie-Chow interpolation in OpenFOAM*, 2006.

[9] Jasak, H. *Numerical Solution Algorithms for Compressible Flows*, Lecture Notes for University of Zagreb.

[10] Hirt, C.W.; Nichols B.D. *Volume of Fluid (VOF) Method for the Dynamics of Free Boundaries*, Journal of Computational Physics, Vol. 39, 1, p. 201-225

[11] Ubbink, O. *Numerical prediction of two fluid systems with sharp interfaces*, Ph.D Thesis, Imperial College of Science, Technology and Medicine, London, 1997.

[12] Ubbink, H. *Computational Fluid Dynamics of Dispersed Two-Phase Flows at High Phase Fractions*, Ph.D Thesis, Imperial College of Science, Technology and Medicine, London, 2002.

[13] Berberovic, E.; Van Hinsberg, N.P.; Jakirlic, S.; Roisman, I.V; Tropea, C. *Drop impact onto a liquid layer of finite thicness: Dynamics of the cavity evolution*, Physical Review E, 79, 2009.

[14] OpenCFD, Technical Report No. TR/HGW/02, 2005 (unpublished).

[15] *A tensorial approach to computational continuum mechanics using object-oriented techniques*, Computers in Physics, Vol. 12, 6, p. 620-631, 1998

[16] OpenCFD Ltd. *OpenFOAM, The Open Source CFD Toolbox, User Guide*, 2009.

[17] OpenCFD Ltd. *OpenFOAM, The Open Source CFD Toolbox, Programmer's Guide*, 2009.

[18] http://openfoamwiki.net/index.php/IcoFoam

[19] Prosperetti, A *Motion of Two Superposed Viscous Fluids*, Physics of Fluids, 24(7):1217-1223, 1981.