

Finite Volume Method: A Crash introduction

- Before continuing, we want to remind you that this is a brief introduction to the FVM.
- Let us use the general transport equation as the starting point to explain the FVM,

$$\underbrace{\int_{V_P} \frac{\partial \rho \phi}{\partial t} dV}_{\text{temporal derivative}} + \underbrace{\int_{V_P} \nabla \cdot (\rho \mathbf{u} \phi) dV}_{\text{convective term}} - \underbrace{\int_{V_P} \nabla \cdot (\rho \Gamma_\phi \nabla \phi) dV}_{\text{diffusion term}} = \underbrace{\int_{V_P} S_\phi(\phi) dV}_{\text{source term}}$$

- We want to solve the general transport equation for the transported quantity ϕ in a given domain, with given boundary conditions BC and initial conditions IC.
- This is a second order equation. For good accuracy, it is necessary that the order of the discretization is equal or higher than the order of the equation that is being discretized.
- By the way, starting from this equation we can write down the Navier-Stokes equations (NSE). So everything we are going to address also applies to the NSE.

Finite Volume Method: A Crash introduction

- Let us use the general transport equation as the starting point to explain the FVM,

$$\underbrace{\int_{V_P} \frac{\partial \rho \phi}{\partial t} dV}_{\text{temporal derivative}} + \underbrace{\int_{V_P} \nabla \cdot (\rho \mathbf{u} \phi) dV}_{\text{convective term}} - \underbrace{\int_{V_P} \nabla \cdot (\rho \Gamma_\phi \nabla \phi) dV}_{\text{diffusion term}} = \underbrace{\int_{V_P} S_\phi(\phi) dV}_{\text{source term}}$$

- Hereafter we are going to assume that the discretization practice is at least second order accurate in space and time.
- As consequence of the previous requirement, all dependent variables are assumed to vary linearly around a point P in space and instant t in time,

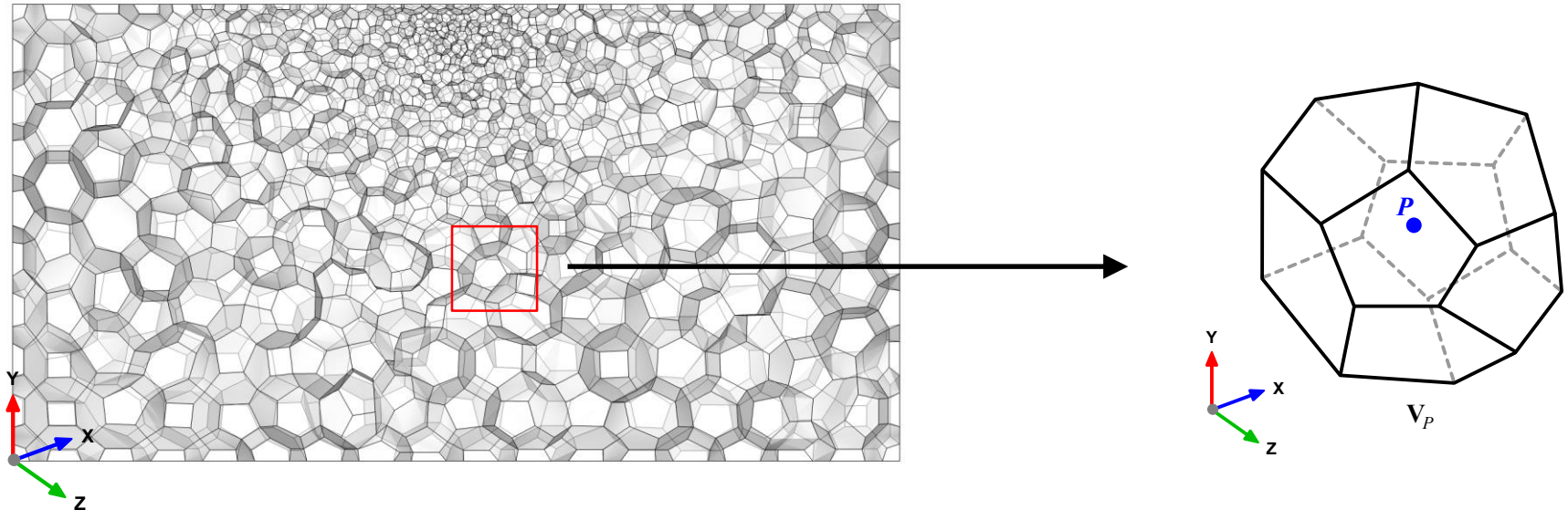
$$\phi(\mathbf{x}) = \phi_P + (\mathbf{x} - \mathbf{x}_P) \cdot (\nabla \phi)_P \quad \text{where} \quad \phi_P = \phi(\mathbf{x}_P)$$

$$\phi(t + \delta t) = \phi^t + \delta t \left(\frac{\partial \phi}{\partial t} \right)^t \quad \text{where} \quad \phi^t = \phi(t)$$

Profile assumptions using Taylor expansions around point P (in space) and point t (in time)

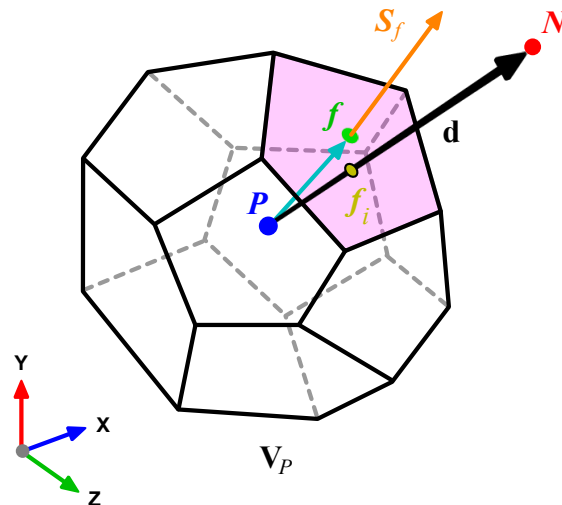
Finite Volume Method: A Crash introduction

- Let us divide the solution domain into a finite number of arbitrary control volumes or cells, such as the one illustrated below.
- Inside each control volume the solution is sought.
- The control volumes can be of any shape (e.g., tetrahedrons, hexes, prisms, pyramids, dodecahedrons, and so on).
- The only requirement is that the elements need to be convex and the faces that made up the control volume need to be planar.
- We also know which control volumes are internal and which control volumes lie on the boundaries.



Finite Volume Method: A Crash introduction

- In the FVM, a lot of overhead goes into the data book-keeping of the domain information.
- We know the following information of every control volume V_P in the domain:
 - The control volume V_P has a volume V and is constructed around point P , which is the centroid of the control volume. Therefore the notation V_P .
 - The vector from the centroid P of V_P to the centroid N of V_N is named \mathbf{d} .
 - We also know all neighbors V_N of the control volume V_P .
 - The control volume faces are labeled f , which also denotes the face center.
 - The location where the vector \mathbf{d} intersects a face is f_i .
 - The face area vector \mathbf{S}_f point outwards from the control volume, is located at the face centroid, is normal to the face and has a magnitude equal to the area of the face.
 - The vector from the centroid P to the face center f is named \mathbf{Pf} .



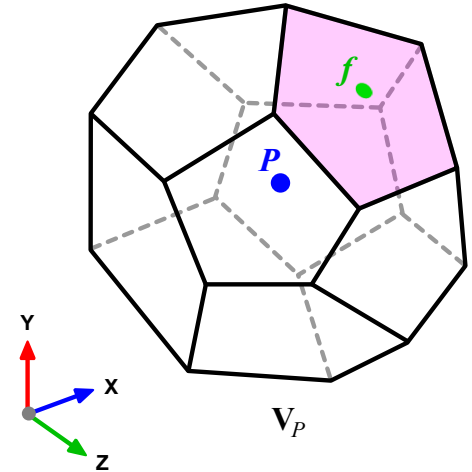
Finite Volume Method: A Crash introduction

- In the control volume illustrated, the centroid P is given by,

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

- In the same way, the centroid of face f is given by

$$\int_{S_f} (\mathbf{x} - \mathbf{x}_P) dS = 0$$



- Finally, we assume that the values of all variables are computed and stored in the centroid of the control volume V_P and that they are represented by a piecewise constant profile (the mean value),

$$\phi_P = \bar{\phi} = \frac{1}{V_P} \int_{V_P} \phi(\mathbf{x}) dV$$

- This is known as the collocated arrangement.
- All the previous approximations are at least second order accurate.

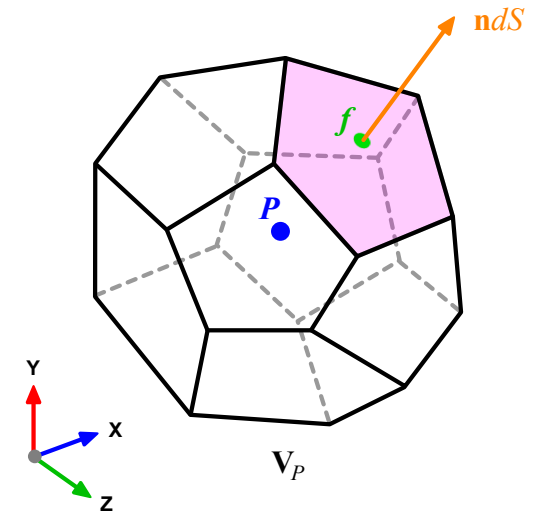
Finite Volume Method: A Crash introduction

- Let us recall the Gauss or Divergence theorem,

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

where ∂V_P is a closed surface bounding the control volume V_P and $d\mathbf{S}$ represents an infinitesimal surface element with associated normal \mathbf{n} pointing outwards of the surface ∂V_P , and $\mathbf{n}dS = d\mathbf{S}$

- The Gauss or Divergence theorem simply states that the outward flux of a vector field through a closed surface is equal to the volume integral of the divergence over the region inside the surface.
- This theorem is fundamental in the FVM, it is used to convert the volume integrals appearing in the governing equations into surface integrals.



Finite Volume Method: A Crash introduction

- Let us use the Gauss theorem to convert the volume integrals into surface integrals,

$$\underbrace{\int_{V_P} \frac{\partial \rho \phi}{\partial t} dV}_{\text{temporal derivative}} + \underbrace{\int_{V_P} \nabla \cdot (\rho \mathbf{u} \phi) dV}_{\text{convective term}} - \underbrace{\int_{V_P} \nabla \cdot (\rho \Gamma_\phi \nabla \phi) dV}_{\text{diffusion term}} = \int_{V_P} \underbrace{S_\phi(\phi) dV}_{\text{source term}}$$



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

$$\frac{\partial}{\partial t} \int_{V_P} (\rho \phi) dV + \oint_{\partial V_P} \underbrace{d\mathbf{S} \cdot (\rho \mathbf{u} \phi)}_{\text{convective flux}} - \oint_{\partial V_P} \underbrace{d\mathbf{S} \cdot (\rho \Gamma_\phi \nabla \phi)}_{\text{diffusive flux}} = \int_{V_P} S_\phi(\phi) dV$$

- At this point the problem reduces to interpolating somehow the cell centered values (known quantities) to the face centers.

Finite Volume Method: A Crash introduction

- Integrating in space each term of the general transport equation and by using Gauss theorem, yields to the following discrete equations for each term

Convective term:

$$\int_{V_P} \underbrace{\nabla \cdot (\rho \mathbf{u} \phi)}_{\text{convective term}} dV = \oint_{\partial V_P} \underbrace{d\mathbf{S} \cdot (\rho \mathbf{u} \phi)}_{\text{convective flux}} = \sum_f \int_f d\mathbf{S} \cdot (\rho \mathbf{u} \phi)_f \approx \sum_f \underbrace{\mathbf{S}_f \cdot (\overline{\rho \mathbf{u} \phi})_f}_{\text{approximation}} = \sum_f \mathbf{S}_f \cdot (\rho \mathbf{u} \phi)_f$$

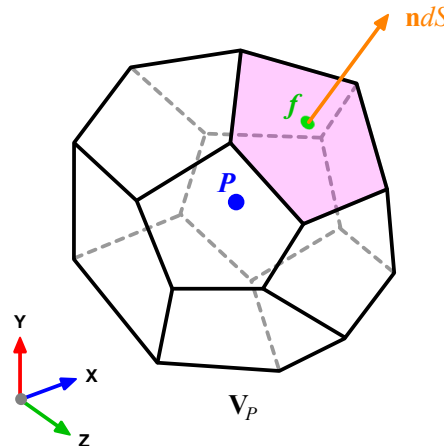
↓

By using Gauss theorem we convert volume integrals into surface integrals

where we have approximated the integrant by means of the mid point rule, which is second order accurate

Gauss theorem:

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



Finite Volume Method: A Crash introduction

- Integrating in space each term of the general transport equation and by using Gauss theorem, yields to the following discrete equations for each term

Diffusive term:

$$\underbrace{\int_{V_P} \nabla \cdot (\rho \Gamma_\phi \nabla \phi) dV}_{\text{diffusion term}} = \underbrace{\oint_{\partial V_P} d\mathbf{S} \cdot (\rho \Gamma_\phi \nabla \phi)}_{\text{diffusive flux}} = \sum_f \int_f d\mathbf{S} \cdot (\rho \Gamma_\phi \nabla \phi)_f \approx \underbrace{\sum_f \mathbf{S}_f \cdot (\overline{\rho \Gamma_\phi \nabla \phi})_f}_{\text{approximation}} = \sum_f \mathbf{S}_f \cdot (\rho \Gamma_\phi \nabla \phi)_f$$

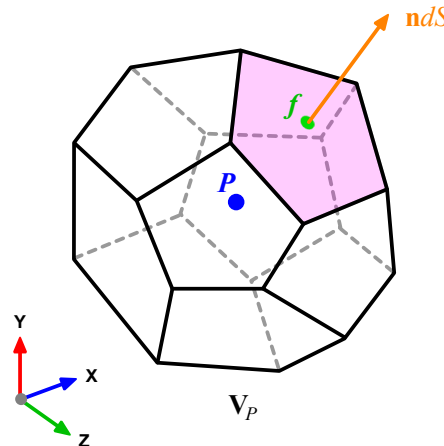
↓

By using Gauss theorem we convert volume integrals into surface integrals

where we have approximated the integrant by means of the mid point rule, which is second order accurate

Gauss theorem:

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



Finite Volume Method: A Crash introduction

- Integrating in space each term of the general transport equation and by using Gauss theorem, yields to the following discrete equations for each term

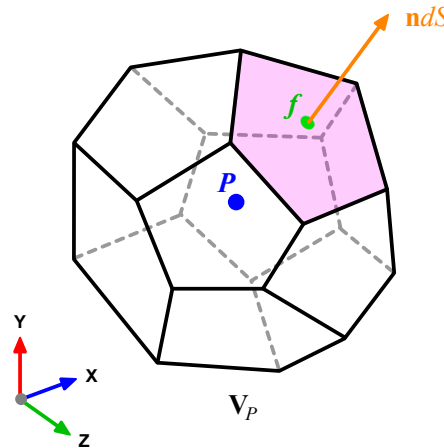
Gradient term:

$$(\nabla \phi)_P = \frac{1}{V_P} \sum_f (\mathbf{S}_f \phi_f)$$

where we have approximated the centroid gradients by using the Gauss theorem.
This method is second order accurate

Gauss theorem:

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



Finite Volume Method: A Crash introduction

- Integrating in space each term of the general transport equation and by using Gauss theorem, yields to the following discrete equations for each term

Source term:

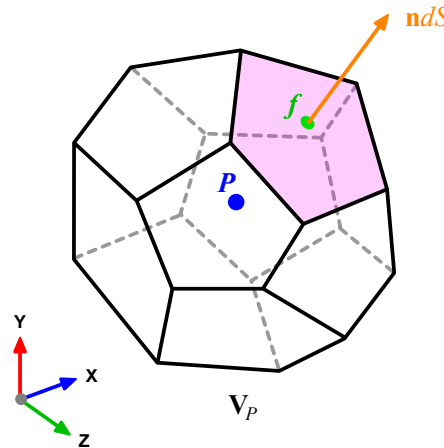
$$\int_{V_P} S_\phi(\phi) dV = S_c V_P + S_p V_P \phi_P$$

This approximation is exact if S_ϕ is either constant or varies linearly within the control volume; otherwise is second order accurate.

S_c is the constant part of the source term and S_p is the non-linear part

Gauss theorem:

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



Finite Volume Method: A Crash introduction

- Using the previous equations to evaluate the general transport equation over all the control volumes, we obtain the following semi-discrete equation

$$\underbrace{\int_{V_P} \frac{\partial \rho \phi}{\partial t} dV}_{\text{temporal derivative}} + \sum_f \underbrace{\mathbf{S}_f \cdot (\rho \mathbf{u} \phi)_f}_{\text{convective flux}} - \sum_f \underbrace{\mathbf{S}_f \cdot (\rho \Gamma_\phi \nabla \phi)_f}_{\text{diffusive flux}} = \underbrace{(S_c V_P + S_p V_P \phi_P)}_{\text{source term}}$$

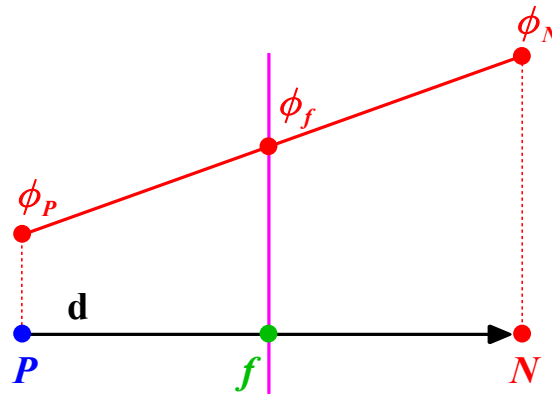
where $\mathbf{S} \cdot (\rho \mathbf{u} \phi) = F^C$ is the convective flux and $\mathbf{S} \cdot (\rho \Gamma_\phi \nabla \phi) = F^D$ is the diffusive flux.

- And recall that all variables are computed and stored at the centroid of the control volumes.
- The face values appearing in the convective and diffusive fluxes have to be computed by some form of interpolation from the centroid values of the control volumes at both sides of face f .

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes

- By looking the figure below, the face values appearing in the convective flux can be computed as follows,



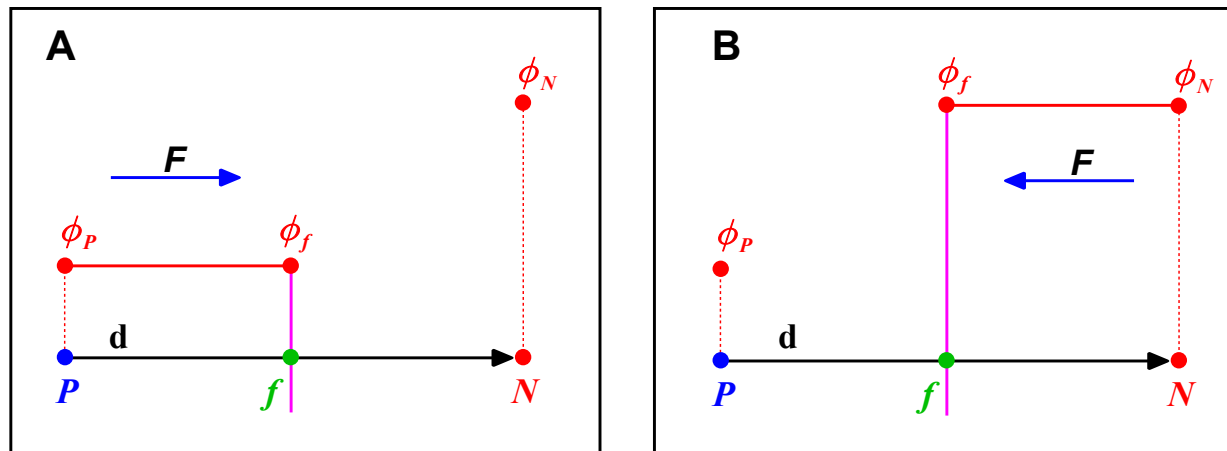
$$\phi_f = f_x \phi_P + (1 - f_x) \phi_N \qquad f_x = \frac{fN}{PN} = \frac{|\mathbf{x}_f - \mathbf{x}_N|}{|\mathbf{d}|}$$

- This type of interpolation scheme is known as linear interpolation or central differencing and it is second order accurate.
- However, it may generate oscillatory solutions (unbounded solutions).

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes

- By looking the figure below, the face values appearing in the convective flux can be computed as follows,



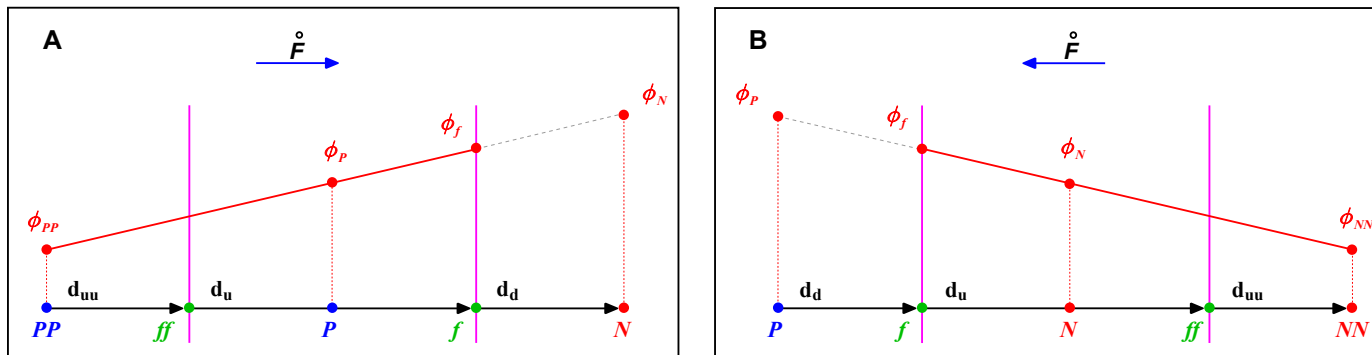
$$\phi_f = \begin{cases} \phi_f = \phi_P & \text{for } \dot{F} \geq 0, \\ \phi_f = \phi_N & \text{for } \dot{F} < 0. \end{cases}$$

- This type of interpolation scheme is known as upwind differencing and it is first order accurate.
- This scheme is bounded (non-oscillatory) and diffusive.

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes

- By looking the figure below, the face values appearing in the convective flux can be computed as follows,



$$\phi_f = \begin{cases} \phi_P + \frac{1}{2}(\phi_P - \phi_{PP}) = \frac{2}{3}\phi_P - \frac{1}{2}\phi_{PP} & \text{for } \dot{F} \geq 0, \\ \phi_N + \frac{1}{2}(\phi_N - \phi_{NN}) = \frac{2}{3}\phi_N - \frac{1}{2}\phi_{NN} & \text{for } \dot{F} < 0. \end{cases}$$

- This type of interpolation scheme is known as second order upwind differencing (SOU), linear upwind differencing (LUD) or Beam-Warming (BW), and it is second order accurate.
- For highly convective flows or in the presence of strong gradients, this scheme is oscillatory (unbounded).

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes

- To prevent oscillations in the SOU, we add a gradient or slope limiter function $\psi(r)$.

$$\phi_f = \begin{cases} \phi_P + \frac{1}{2}\psi_P^-(\phi_P - \phi_{PP}) & \text{for } \overset{\circ}{F} \geq 0, \\ \phi_N - \frac{1}{2}\psi_P^+(\phi_{NN} - \phi_N) & \text{for } \overset{\circ}{F} < 0. \end{cases}$$

- When the limiter detects strong gradients or changes in slope, it switches locally to low resolution (upwind).
- The concept of the limiter function $\psi(r)$ is based on monitoring the ratio of successive gradients, e.g.,

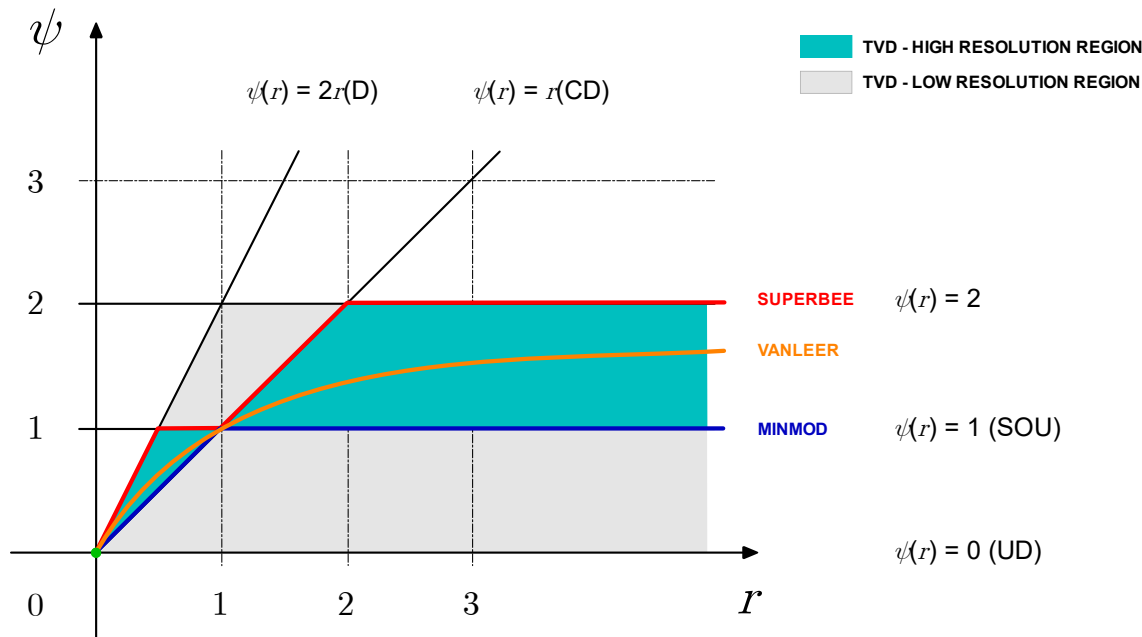
$$r_P^- = \frac{\phi_N - \phi_P}{\phi_P - \phi_{PP}} \quad \text{and} \quad r_P^+ = \frac{\phi_P - \phi_N}{\phi_N - \phi_{NN}}$$

- By adding a well designed limiter function $\psi(r)$, we get a high resolution (second order accurate), and bounded scheme. This is a TVD scheme.

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – TVD schemes

- A TVD scheme, is a second order accurate scheme that does not create new local undershoots and/or overshoots in the solution or amplify existing extremes (high resolution).
- The choice of the limiter function $\psi(r)$ dictates the order of the scheme and its boundedness. High resolution schemes falls in the blue area and low resolution schemes falls in the grey area.
- The drawback of the limiters is that they reduce the accuracy of the scheme **locally** to first order, when $r \leq 0$ (sharp gradient, opposite slopes). However, this is justify when it serves to suppress oscillations.
- The various limiters have different switching characteristics and are selected according to the particular problem and solution scheme. No particular limiter has been found to work well for all problems, and a particular choice is usually made on a trial and error basis.
- The Sweby diagram (Sweby, 1984), gives the necessary and sufficient conditions for a scheme to be TVD.



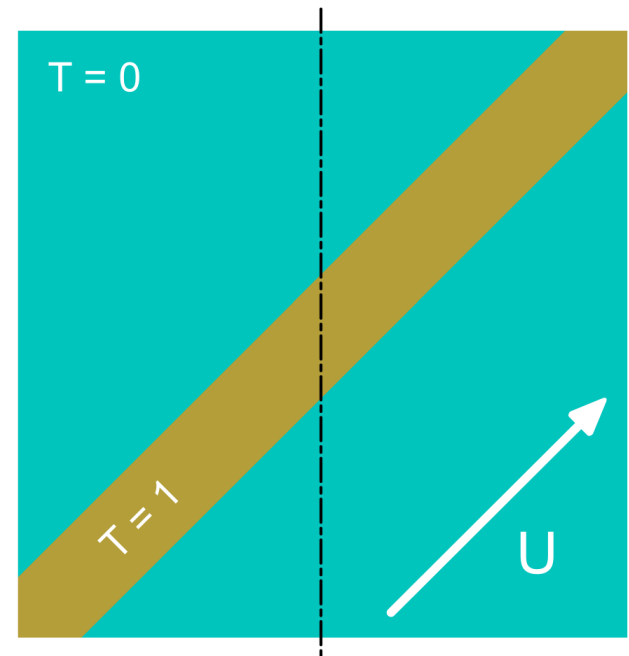
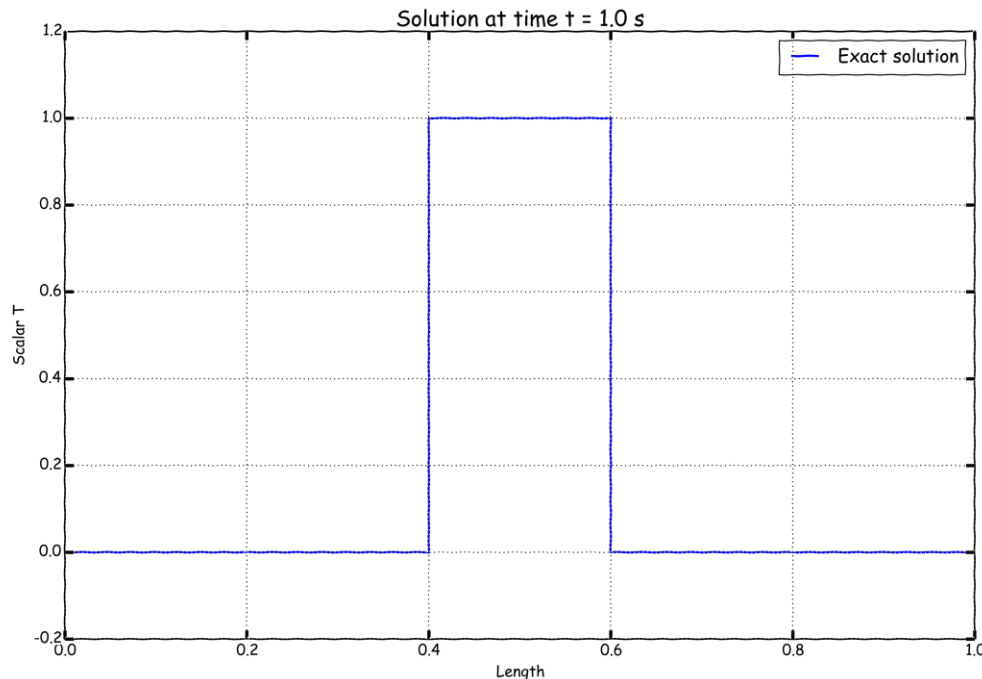
$$\phi_f = \begin{cases} \phi_P + \frac{1}{2}\psi_P^-(\phi_P - \phi_{PP}) & \text{for } \overset{\circ}{F} \geq 0, \\ \phi_N - \frac{1}{2}\psi_P^+(\phi_{NN} - \phi_N) & \text{for } \overset{\circ}{F} < 0. \end{cases}$$

UD = upwind
 SOU = second order upwind
 CD = central differencing
 D = downwind

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – TVD schemes

- Let us see how the superbee, minmod and vanleer TVD schemes behave in a numerical schemes killer test case:
 - The oblique double step profile in a uniform vector field (pure convection).
- By the way, this problem has an exact solution.

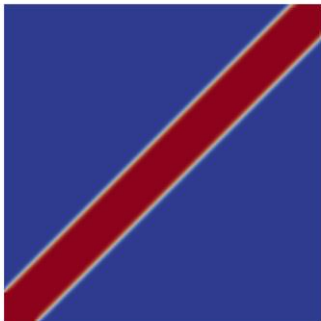


Finite Volume Method: A Crash introduction

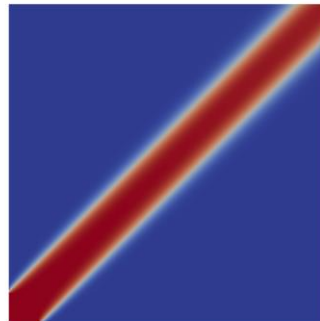
Interpolation of the convective fluxes – Linear and non-linear limiter functions

- Let us see how the superbee, minmod and vanleer TVD schemes behave in a numerical schemes killer test case.
 - The oblique double step profile in a uniform vector field (pure convection).

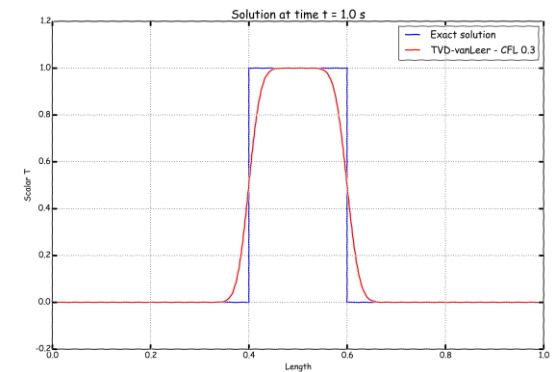
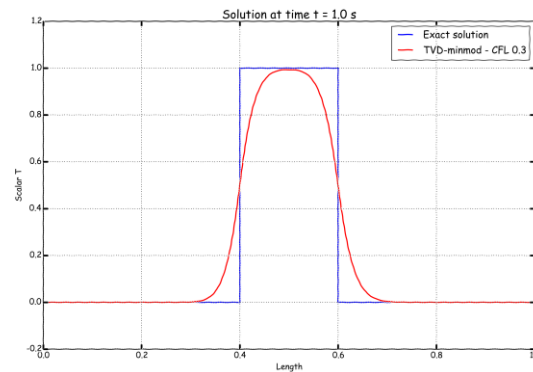
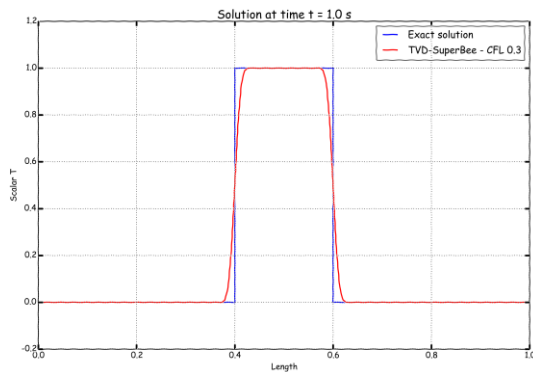
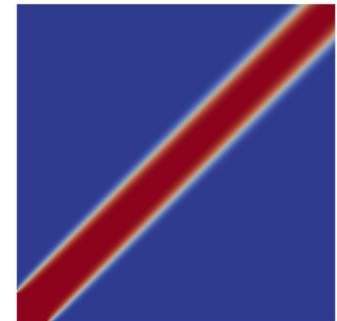
SuperBee - Compressive



Minmod - Diffusive



vanLeer - Smooth

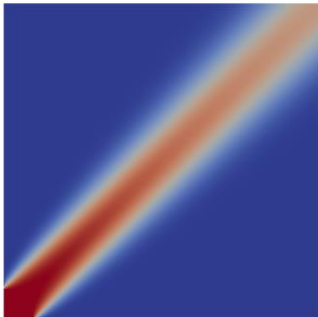


Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – Linear and non-linear limiter functions

- Comparison of linear upwind method (2nd order) and upwind method (1st order).
- The upwind method is extremely stable and non-oscillatory. However is highly diffusive.
- On the other side, the linear upwind method is accurate but oscillatory in the presence of strong gradients.

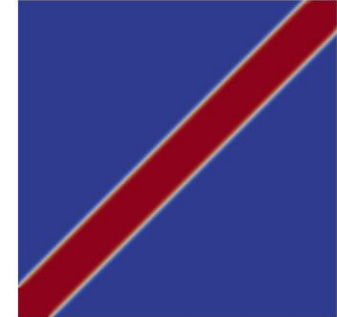
Upwind – 1st order



Linear Upwind – 2nd order



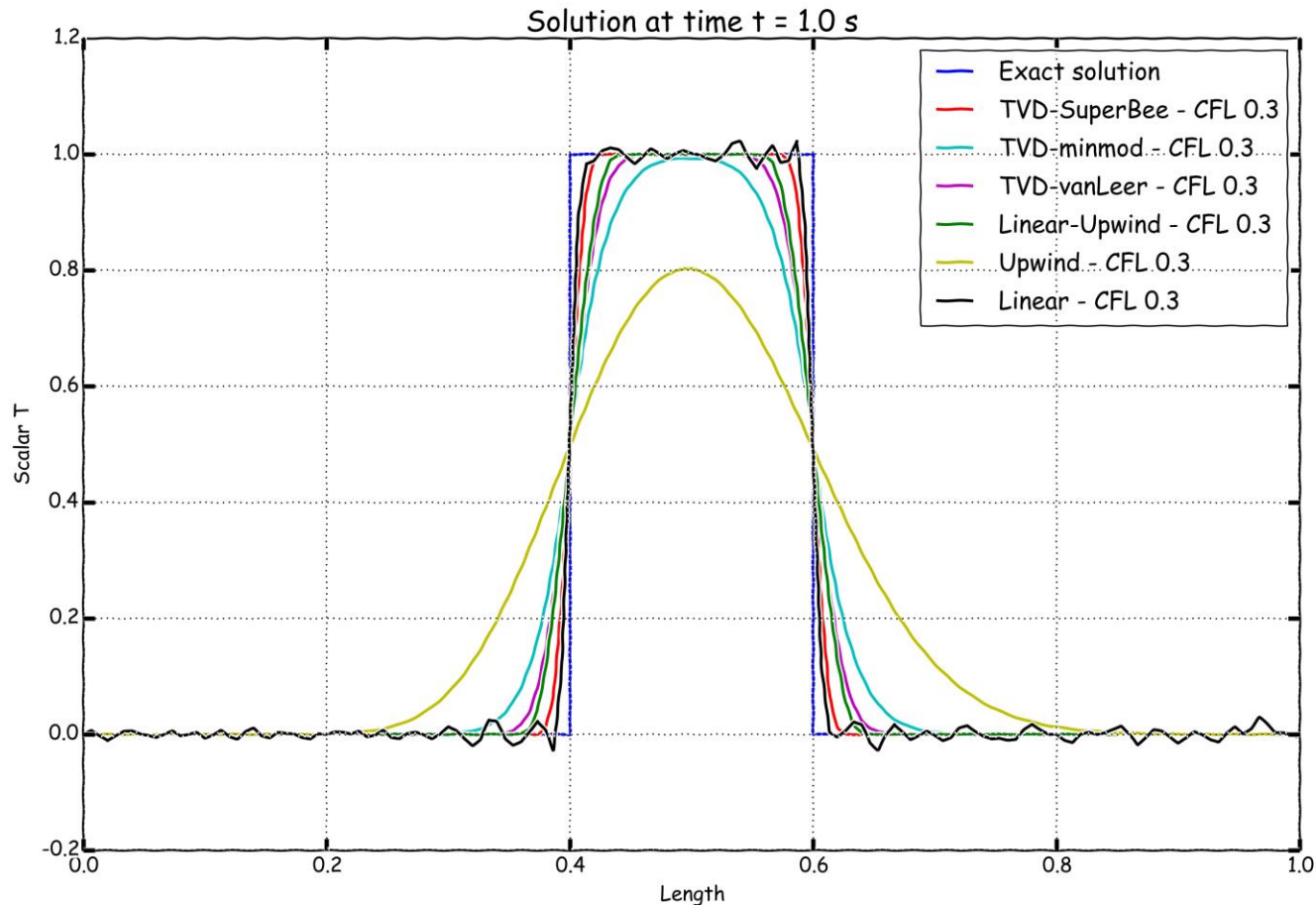
SuperBee – TVD



Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – Linear and non-linear limiter functions

- Let us see how the linear and non-linear limiter functions compare.

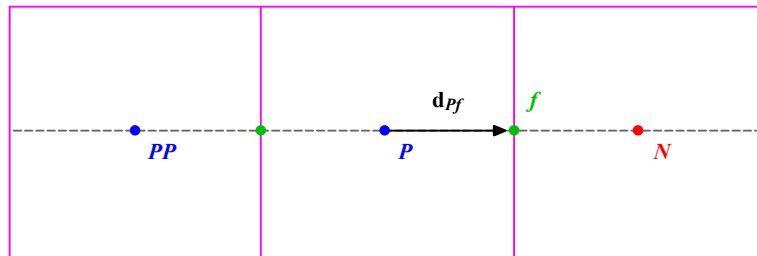


Finite Volume Method: A Crash introduction

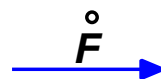
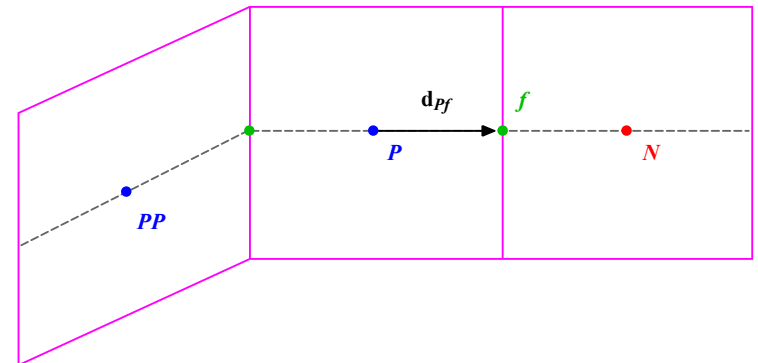
Interpolation of the convective fluxes – Unstructured meshes

- All higher-order schemes we have seen so far assume line structure (figure A).
- That is, the cell centers PP , P , and N are all aligned.
- In unstructured meshes, it is not straightforward to use the previous schemes, as the cell center PP is not aligned with the vector connecting cells P and N (figure B).
- Higher-order schemes for unstructured meshes are an area of active research and new ideas continue to emerge.

A



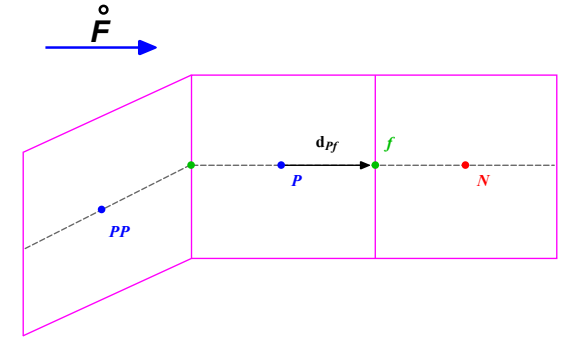
B



Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – Unstructured meshes

- A simple way around this problem is to redefine higher-order schemes in terms of gradients at the control volume P.
- For example, using the gradient of the cells, we can compute the face values as follows,



Upwind $\rightarrow \phi_f = \phi_P$

Central difference $\rightarrow \phi_f = \phi_P + \nabla \phi_f \cdot \mathbf{d}_{Pf}$

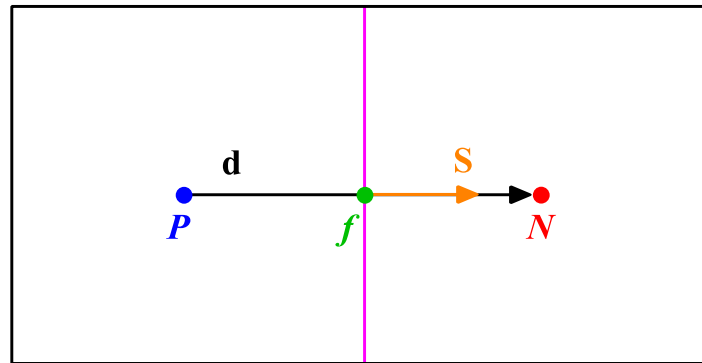
Second order upwind differencing $\rightarrow \phi_f = \phi_P + (2\nabla \phi_P - \nabla \phi_f) \cdot \mathbf{d}_{Pf}$

- Notice that in this new formulation the cell PP does not appear any more.
- The problem now turns in the accurate evaluation of the gradients at the cell and face centers. This can be done using Gauss method as previously explained.

Finite Volume Method: A Crash introduction

Interpolation of diffusive fluxes in a orthogonal mesh

- By looking the figures below, the face values appearing in the diffusive flux in an orthogonal mesh can be computed as follows,



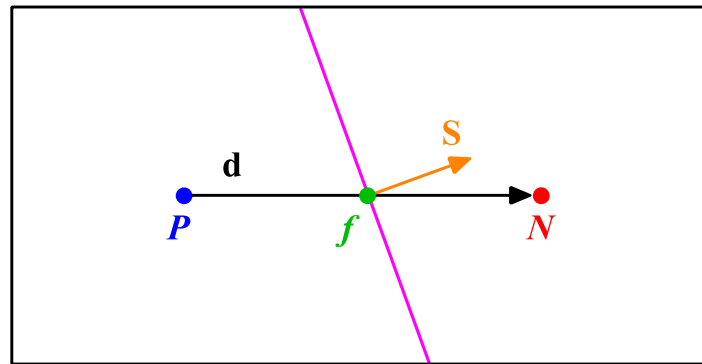
$$\mathbf{S} \cdot (\nabla \phi)_f = |\mathbf{S}| \frac{\phi_N - \phi_P}{|\mathbf{d}|}.$$

- This is a central difference approximation of the first order derivative. This type of approximation is second order accurate.

Finite Volume Method: A Crash introduction

Interpolation of diffusive fluxes in a non-orthogonal mesh

- By looking the figures below, the face values appearing in the diffusive flux in a non-orthogonal mesh (20°) can be computed as follows,



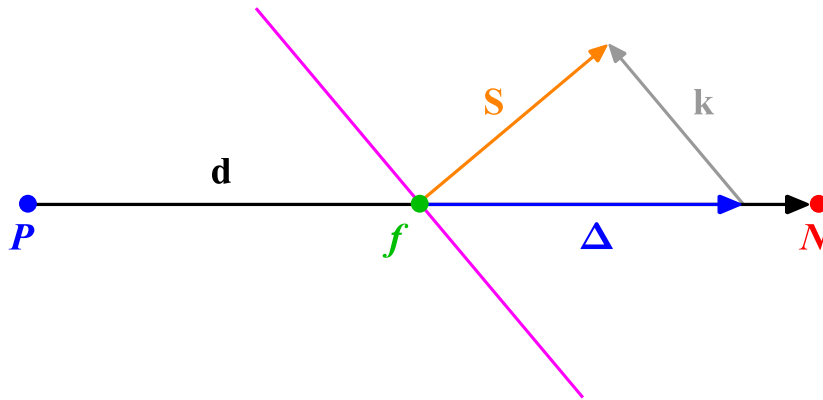
$$\mathbf{S} \cdot (\nabla \phi)_f = \underbrace{|\Delta_\perp| \frac{\phi_N - \phi_P}{|\mathbf{d}|}}_{\text{orthogonal contribution}} + \underbrace{\mathbf{k} \cdot (\nabla \phi)_f}_{\text{non-orthogonal contribution}} .$$

- This type of approximation is second order accurate but involves a larger truncation error. It also uses a larger numerical stencil, which make it less stable.

Finite Volume Method: A Crash introduction

Correction of diffusive fluxes in a non-orthogonal mesh

- By looking the figures below, the face values appearing in the diffusive flux in a non-orthogonal mesh (40°) can be computed as follows.
- Using the over-relaxed approach, the diffusive fluxes can be corrected as follow,



Over-relaxed approach

$$\Delta_{\perp} = \frac{d}{d \cdot S} |S|^2.$$

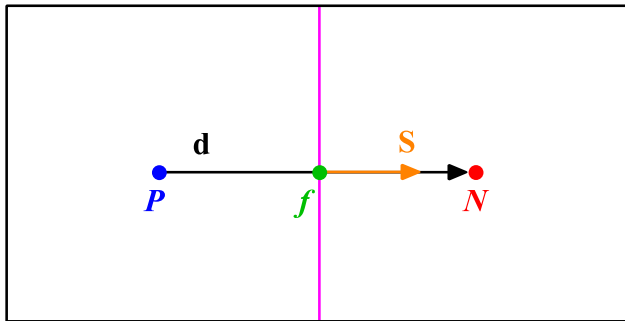
$$S = \Delta_{\perp} + k.$$

$$S \cdot (\nabla \phi)_f = \underbrace{|\Delta_{\perp}| \frac{\phi_N - \phi_P}{|d|}}_{\text{orthogonal contribution}} + \underbrace{k \cdot (\nabla \phi)_f}_{\text{non-orthogonal contribution}}.$$

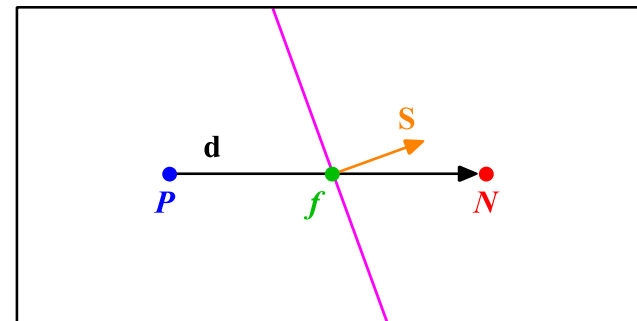
Finite Volume Method: A Crash introduction

Mesh induced errors

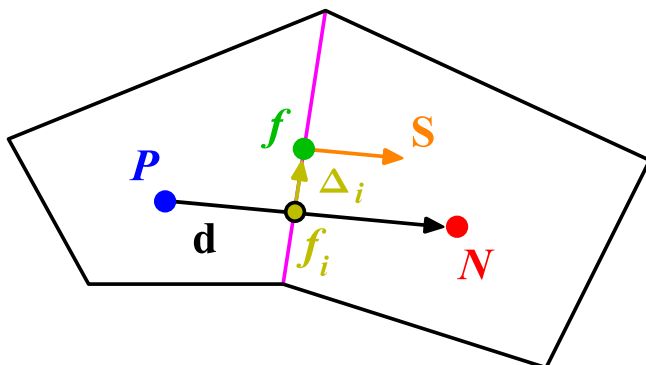
- In order to maintain second order accuracy, and to avoid unboundedness, we need to correct non-orthogonality and skewness errors.
- The ideal case is to have an orthogonal and non skew mesh, but this is the exception rather than the rule.



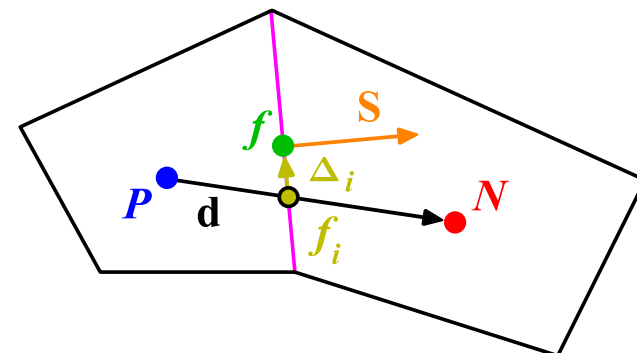
Orthogonal and non skew mesh



Non-orthogonal and non skew mesh



Orthogonal and skew mesh



Non-orthogonal and skew mesh

Finite Volume Method: A Crash introduction

Temporal discretization

- Using the previous equations to evaluate the general transport equation over all the control volumes, we obtain the following semi-discrete equation,

$$\underbrace{\int_{V_P} \frac{\partial \rho \phi}{\partial t} dV}_{\text{temporal derivative}} + \underbrace{\sum_f \mathbf{S}_f \cdot (\rho \mathbf{u} \phi)_f}_{\text{convective flux}} - \underbrace{\sum_f \mathbf{S}_f \cdot (\rho \Gamma_\phi \nabla \phi)_f}_{\text{diffusive flux}} = \underbrace{(S_c V_P + S_p V_P \phi_P)}_{\text{source term}}$$

where $\mathbf{S} \cdot (\rho \mathbf{u} \phi) = F^C$ is the convective flux and $\mathbf{S} \cdot (\rho \Gamma_\phi \nabla \phi) = F^D$ is the diffusive flux.

- After spatial discretization, we can proceed with the temporal discretization. By proceeding in this way we are using the Method of Lines (MOL).
- The main advantage of the MOL method, is that it allows us to select numerical approximations of different accuracy for the spatial and temporal terms. Each term can be treated differently to yield to different accuracies.

Finite Volume Method: A Crash introduction

Temporal discretization

- Now, we evaluate in time the semi-discrete general transport equation

$$\int_t^{t+\Delta t} \left[\left(\frac{\partial \rho \phi}{\partial t} \right)_P V_P + \sum_f \mathbf{S}_f \cdot (\rho \mathbf{u} \phi)_f - \sum_f \mathbf{S}_f \cdot (\rho \Gamma_\phi \nabla \phi)_f \right] dt$$
$$= \int_t^{t+\Delta t} (S_c V_P + S_p V_P \phi_P) dt.$$

- At this stage, we can use any time discretization scheme, e.g., Crank-Nicolson, euler implicit, forward euler, backward differencing, adams-bashforth, adams-moulton.
- It should be noted that the order of the temporal discretization of the transient term does not need to be the same as the order of the discretization of the spatial terms. Each term can be treated differently to yield different accuracies. As long as the individual terms are at least second order accurate, the overall accuracy will also be second order.

Finite Volume Method: A Crash introduction

Linear system solution

- After spatial and temporal discretization and by using equation

$$\int_t^{t+\Delta t} \left[\left(\frac{\partial \rho \phi}{\partial t} \right)_P V_P + \sum_f \mathbf{S}_f \cdot (\rho \mathbf{u} \phi)_f - \sum_f \mathbf{S}_f \cdot (\rho \Gamma_\phi \nabla \phi)_f \right] dt = \int_t^{t+\Delta t} (S_c V_P + S_p V_P \phi_P) dt$$

in every control volume V_P of the domain, a system of linear algebraic equations for the transported quantity ϕ is assembled,

$$\begin{pmatrix} a_{11} & a_{12} & & \ddots & & & \\ a_{21} & a_{22} & a_{23} & & \ddots & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ \ddots & & \ddots & \ddots & \ddots & \ddots & \\ & a_S & & a_W & a_P & a_E & a_N \\ & & \ddots & \ddots & \ddots & \ddots & \\ & & & \ddots & \ddots & \ddots & \ddots \\ & & & & \ddots & \ddots & a_{PP} \end{pmatrix} \times \begin{pmatrix} \phi_S \\ \phi_W \\ \phi_P \\ \phi_E \\ \phi_N \end{pmatrix} = \begin{pmatrix} b_S \\ b_W \\ b_P \\ b_E \\ b_N \end{pmatrix}$$

- This system can be solved by using any iterative or direct method.

Finite Volume Method: A Crash introduction

So, what does OpenFOAM® do?

- It simply discretize in space and time the governing equations in arbitrary polyhedral control volumes over the whole domain. Assembling in this way a large set of linear discrete algebraic equations (DAE), and then it solves this system of DAE to find the solution of the transported quantities.
- Therefore, we need to give to OpenFOAM® the following information:
 - Discretization of the solution domain or the mesh. This information is contained in the directory **constant/polyMesh**
 - Boundary conditions and initials conditions. This information is contained in the directory **0**
 - Physical properties such as density, gravity, diffusion coefficient, viscosity, etc. This information is contained in the directory **constant**
 - Physics involve, such as turbulence modeling, mass transfer, source terms, etc. This information is contained in the directories **constant** and/or **system**
 - How to discretize in space each term of the governing equations (diffusive, convective, gradient and source terms). This information is set in the *system/fvSchemes* dictionary.
 - How to discretize in time the obtained semi-discrete governing equations. This information is set in the *system/fvSchemes* dictionary.
 - How to solve the linear system of discrete algebraic equations (crunch numbers). This information is set in the *system/fvSolution* dictionary.
 - Set runtime parameters and general instructions on how to run the case (such as time step and maximum CFL number). This information is set in the *system/controlDict* dictionary.
 - Additionally, we may set sampling and **functionObjects** for post-processing. This information is contained in the specific dictionaries contained in the directory **system/**

Finite Volume Method: A Crash introduction

Where do we set all the discretization schemes in OpenFOAM®?

```
ddtSchemes ←  $\frac{\partial \phi}{\partial t}$ 
{
  default backward;
}

gradSchemes ←  $\nabla \phi_P$ 
{
  default Gauss linear;
  grad(p) Gauss linear;
}

divSchemes ←  $\nabla \cdot (\mathbf{U} \phi)$ 
{
  default none;
  div(phi,U) Gauss linear;
}

laplacianSchemes ←  $\nabla \cdot \Gamma \nabla \phi$ 
{
  default Gauss linear orthogonal;
}

interpolationSchemes ←  $\begin{cases} \phi_f = f_x \phi_P + (1 - f_x) \phi_N \\ f_x = \frac{f_N}{P_N} = \frac{|\mathbf{x}_f - \mathbf{x}_N|}{|\mathbf{d}|} \end{cases}$ 
{
  default linear;
}

snGradSchemes
{
  default orthogonal; ←  $\mathbf{n}_f \cdot \nabla \phi_f$ 
}
```

- The *fvSchemes* dictionary contains the information related to the discretization schemes for the different terms appearing in the governing equations.
- The discretization schemes can be chosen in a term-by-term basis.
- The keyword **ddtSchemes** refers to the time discretization.
- The keyword **gradSchemes** refers to the gradient term discretization.
- The keyword **divSchemes** refers to the convective terms discretization.
- The keyword **laplacianSchemes** refers to the Laplacian terms discretization.
- The keyword **interpolationSchemes** refers to the method used to interpolate values from cell centers to face centers. It is unlikely that you will need to use something different from linear.
- The keyword **snGradSchemes** refers to the discretization of the surface normal gradients evaluated at the faces.
- Remember, if you want to know the options available for each keyword you can use the banana method.

Finite Volume Method: A Crash introduction

Time discretization schemes

- There are many time discretization schemes available in OpenFOAM®.
- You will find the source code in the following directory:
 - `$WM_PROJECT_DIR/src/finiteVolume/finiteVolume/ddtSchemes`
- These are the time discretization schemes that you will use most of the times:
 - **steadyState**: for steady state simulations (implicit/explicit).
 - **Euler**: time dependent first order (implicit/explicit), bounded.
 - **backward**: time dependent second order (implicit), bounded/unbounded.
 - **CrankNicolson**: time dependent second order (implicit), bounded/unbounded.
- First order methods are bounded and stable, but diffusive.
- Second order methods are accurate, but they might become oscillatory.
- At the end of the day, we always want a second order accurate solution.
- If you keep the CFL less than one when using the Euler method, numerical diffusion is not that much (however, we advise you to do your own benchmarking).

Finite Volume Method: A Crash introduction

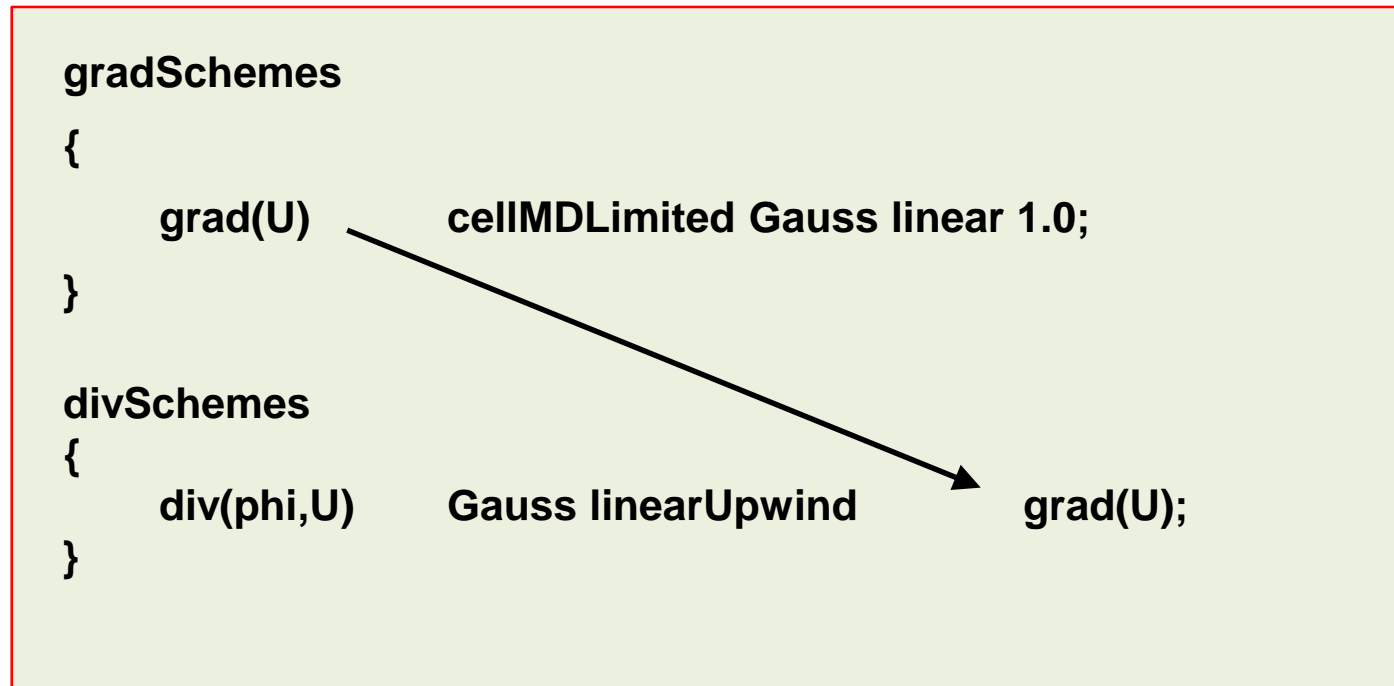
Convective terms discretization schemes

- There are many convective terms discretization schemes available in OpenFOAM® (more than 50 last time we checked).
- You will find the source code in the following directory:
 - `$WM_PROJECT_DIR/src/finiteVolume/interpolation/surfaceInterpolation`
- These are the convective discretization schemes that you will use most of the times:
 - **upwind**: first order accurate.
 - **linearUpwind**: second order accurate, bounded.
 - **linear**: second order accurate, unbounded.
 - **vanLeer**: TVD, second order accurate, bounded.
 - **limitedLinear**: second order accurate, unbounded, but more stable than pure linear. Recommended for LES simulations (kind of similar to the Fromm method).
- First order methods are bounded and stable but diffusive.
- Second order methods are accurate, but they might become oscillatory.
- At the end of the day, we always want a second order accurate solution.

Finite Volume Method: A Crash introduction

Convective terms discretization schemes

- When you use **linearUpwind** for **div(phi,U)**, you need to tell OpenFOAM® how to compute the velocity gradient or **grad(U)**:



- Same applies for scalars (e.g. **k**, **epsilon**, **omega**, **T**)

Finite Volume Method: A Crash introduction

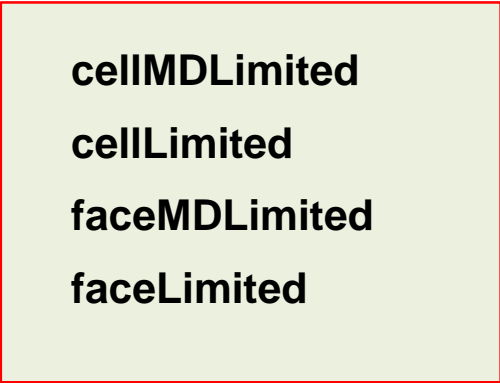
Gradient terms discretization schemes

- There are many gradient discretization schemes available in OpenFOAM®.
- You will find the source code in the following directory:
 - `$WM_PROJECT_DIR/src/finiteVolume/finiteVolume/gradSchemes`
- These are the gradient discretization schemes that you will use most of the times:
 - **Gauss**
 - **leastSquares**
- To avoid overshoots or undershoots when computing the gradients, you can use gradient limiters.
- Gradient limiters increase the stability of the method but add diffusion due to clipping.
- You will find the source code in the following directory:
 - `$WM_PROJECT_DIR/src/finiteVolume/finiteVolume/gradSchemes/limitedGradSchemes`
- These are the gradient limiter schemes available in OpenFOAM®:
 - **cellLimited, cellMDLimited, faceLimited, faceMDLimited**
- All of the gradient discretization schemes are at least second order accurate.

Finite Volume Method: A Crash introduction

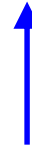
Gradient terms discretization schemes

- These are the gradient limiter schemes available in OpenFOAM®:



cellMDLimited
cellLimited
faceMDLimited
faceLimited

Less diffusive



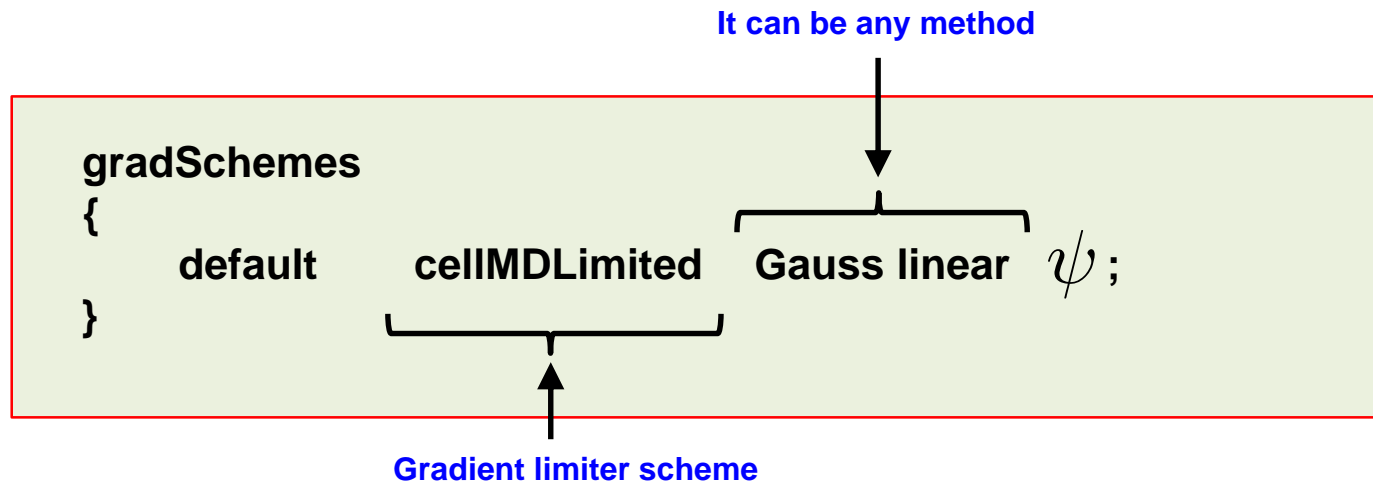
More diffusive

- Cell limiters will limit cell-to-cell values.
- Face limiters will limit face-to-cell values.
- The multi-directional (dimensional) limiters (**cellMDLimited** and **faceMDLimited**), will apply the limiter in each face direction separately.
- The standard limiters (**cellLimited** and **faceLimited**), will apply the limiter to all components of the gradient.
- The default method is the Minmod.

Finite Volume Method: A Crash introduction

Gradient terms discretization schemes

- The gradient limiter implementation in OpenFOAM®, uses a blending factor ψ .



- Setting ψ to 0 is equivalent to turning off the gradient limiter. You gain accuracy but the solution might become unbounded.
- By setting the blending factor equal to 1 the limiter is always on. You gain stability but you give up accuracy (due to gradient clipping).
- If you set the blending factor to 0.5, you get the best of both worlds.
- You can use limiters with all gradient discretization schemes.

Finite Volume Method: A Crash introduction

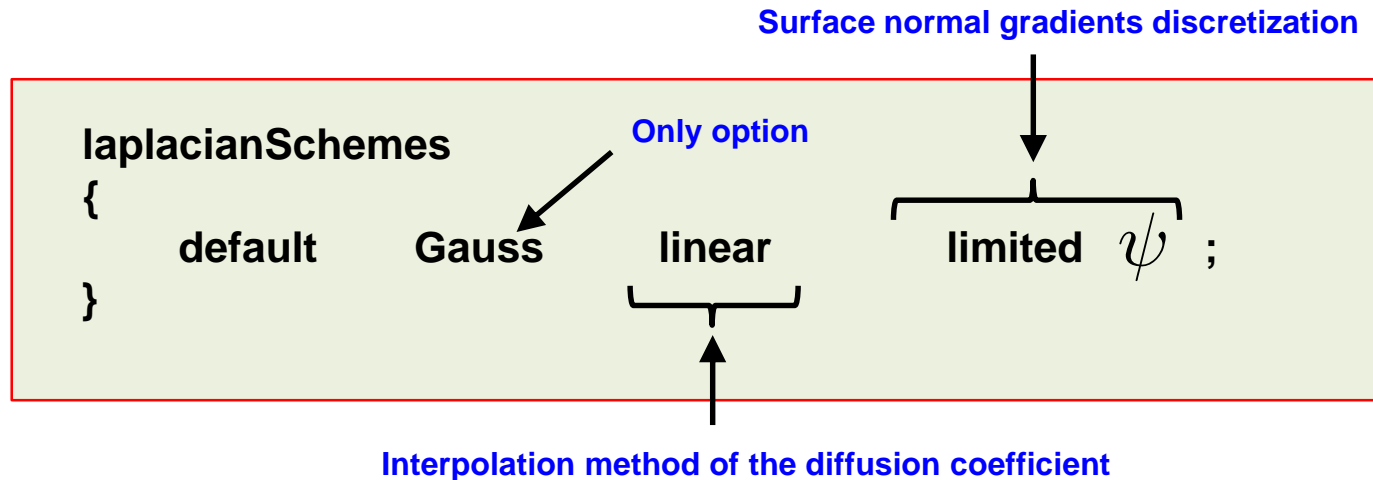
Laplacian terms discretization schemes

- There are many Laplacian terms discretization schemes available in OpenFOAM®.
- You will find the source code in the following directory:
 - `$WM_PROJECT_DIR/src/finiteVolume/finiteVolume/snGradSchemes`
- These are the Laplacian terms discretization schemes that you will use most of the times:
 - **orthogonal**: mainly limited for hexahedral meshes with no grading (a perfect mesh). Second order accurate, bounded on perfect meshes, without non-orthogonal corrections.
 - **corrected**: for meshes with grading and non-orthogonality. Second order accurate, bounded depending on the quality of the mesh, with non-orthogonal corrections.
 - **limited**: for meshes with grading and non-orthogonality. Second order accurate, bounded depending on the quality of the mesh, with non-orthogonal corrections.
 - **uncorrected**: usually used on bad quality meshes with grading and non-orthogonality. Second order accurate, without non-orthogonal corrections. Stable but more diffusive than the limited and corrected methods.

Finite Volume Method: A Crash introduction

Laplacian terms discretization schemes

- The limited method uses a blending factor ψ .

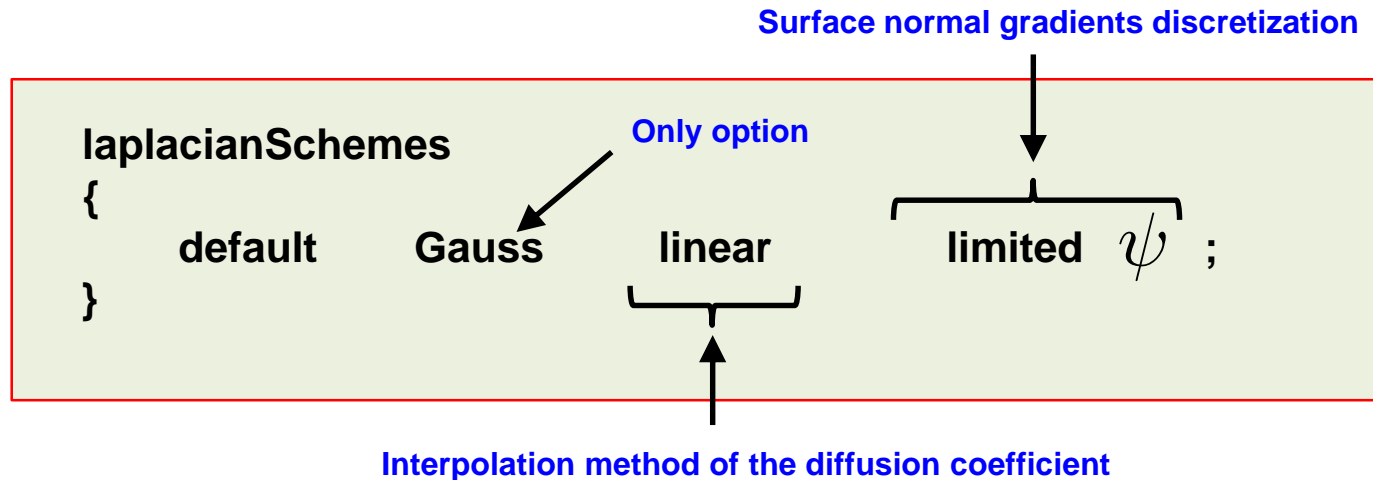


- Setting ψ to 1 is equivalent to using the **corrected** method. You gain accuracy, but the solution might become unbounded.
- By setting the blending factor equal to 0 is equivalent to using the **uncorrected** method. You give up accuracy but gain stability.
- If you set the blending factor to 0.5, you get the best of both worlds. In this case, the non-orthogonal contribution does not exceed the orthogonal part. You give up accuracy but gain stability.

Finite Volume Method: A Crash introduction

Laplacian terms discretization schemes

- The limited method uses a blending factor ψ .



- For meshes with non-orthogonality less than 75, you can set the blending factor to 1.
- For meshes with non-orthogonality between 75 and 85, you can set the blending factor to 0.5
- For meshes with non-orthogonality more than 85, it is better to get a better mesh. But if you definitely want to use that mesh, you can set the blending factor to 0.333, and increase the number of non-orthogonal corrections.
- If you are doing LES or DES simulations, use a blending factor of 1 (this means that you need good meshes).

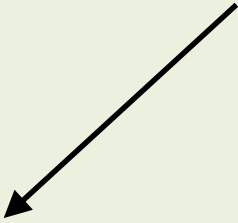
Finite Volume Method: A Crash introduction

Laplacian terms discretization schemes

- The surface normal gradients terms usually use the same method as the one chosen for the Laplacian terms.
- For instance, if you are using the **limited 1** method for the Laplacian terms, you can use the same method for **snGradSchemes**:

```
laplacianSchemes
{
    default          Gauss linear          limited 1;
}


snGradSchemes
{
    default          limited 1;
}
```



Finite Volume Method: A Crash introduction

What method should I use?

```
ddtSchemes
{
    default      CrankNicolson 0;
}
gradSchemes
{
    default      cellLimited Gauss linear 1;
    grad(U)      cellLimited Gauss linear 1;
}
divSchemes
{
    default      none;
    div(phi,U)   Gauss linearUpwindV grad(U);
    div(phi,omega) Gauss linearUpwind default;
    div(phi,k)   Gauss linearUpwind default;
    div((nuEff*dev(T(grad(U)))) Gauss linear;
}
laplacianSchemes
{
    default      Gauss linear limited 1;
}
interpolationSchemes
{
    default      linear;
}
snGradSchemes
{
    default      limited 1;
}
```

- **This setup is recommended for most of the cases.** 
- It is equivalent to the default method you will find in commercial solvers.
- In overall, this setup is second order accurate and fully bounded.
- According to the quality of your mesh, you will need to change the blending factor of the **laplacianSchemes** and **snGradSchemes** keywords.
- To keep temporal diffusion to a minimum, use a CFL number less than 2.
- If during the simulation the turbulence quantities become unbounded, you can safely change the discretization scheme to upwind. After all, turbulence is diffusion.

Finite Volume Method: A Crash introduction

A very accurate but oscillatory numerics

```
ddtSchemes
{
    default      backward;
}
gradSchemes
{
    default      Gauss leastSquares;
}
divSchemes
{
    default      none;
    div(phi,U)   Gauss linear;
    div(phi,omega) Gauss linear;
    div(phi,k)   Gauss linear;
    div((nuEff*dev(T(grad(U)))) Gauss linear;
}
laplacianSchemes
{
    default      Gauss linear limited 1;
}
interpolationSchemes
{
    default      linear;
}
snGradSchemes
{
    default      limited 1;
}
```

- If you are looking for more accuracy, you can use this method.
- In overall, this setup is second order accurate but oscillatory.
- Use this setup with LES simulations or laminar flows with no complex physics, and meshes with overall good quality.
- Use this method with good quality meshes.

Finite Volume Method: A Crash introduction

A very stable but too diffusive numerics

```
ddtSchemes
{
    default      Euler;
}
gradSchemes
{
    default      cellLimited Gauss linear 1;
    grad(U)      cellLimited Gauss linear 1;
}
divSchemes
{
    default      none;
    div(phi,U)   Gauss upwind;
    div(phi,omega) Gauss upwind;
    div(phi,k)   Gauss upwind;
    div((nuEff*dev(T(grad(U)))) Gauss linear;
}
laplacianSchemes
{
    default      Gauss linear limited 0.5;
}
interpolationSchemes
{
    default      linear;
}
snGradSchemes
{
    default      limited 0.5;
}
```

- If you are looking for extra stability, you can use this method.
- This setup is very stable but too diffusive.
- This setup is first order in space and time.
- You can use this setup to start the solution in the presence of bad quality meshes or strong discontinuities.
- Remember, you can start using a first order method and then switch to a second order method.
- **Start robustly, end with accuracy.**

On the CFL number

How to control the CFL number

```
application      pimpleFoam;
startFrom        latestTime;
startTime        0;
stopAt           endTime;
endTime          10;
deltaT           0.0001;
writeControl      runtime;
writeInterval     0.1;
purgeWrite       0;
writeFormat       ascii;
writePrecision    8;
writeCompression off;
timeFormat        general;
timePrecision     6;
runTimeModifiable yes;
adjustTimeStep    yes;
maxCo             2.0;
maxDeltaT         0.001;
```

- You can control the CFL number by changing the mesh cell size or changing the time-step size.
- The easiest way is by changing the time-step size.
- If you refine the mesh, and you would like to have the same CFL number as the base mesh, you will need to decrease the time-step size.
- On the other side, if you coarse the mesh and you would like to have the same CFL number as the base mesh, you will need to increase the time-step size.
- The keyword **deltaT** controls the time-step size of the simulation (0.0001 seconds in this generic case).
- If you use a solver that supports adjustable time-step (**adjustTimeStep**), you can set the maximum CFL number and maximum allowable time-step using the keywords **maxCo** and **maxDeltaT**, respectively.

On the CFL number

How to control the CFL number

```
application      pimpleFoam;
startFrom        latestTime;
startTime        0;
stopAt           endTime;
endTime          10;
deltaT           0.0001;
writeControl      runtime;
writeInterval     0.1;
purgeWrite        0;
writeFormat       ascii;
writePrecision    8;
writeCompression off;
timeFormat        general;
timePrecision     6;
runTimeModifiable yes;
adjustTimeStep    yes;
maxCo             2.0;
maxDeltaT         0.001;
```

- The option **adjustTimeStep** will automatically adjust the time step to achieve the maximum desired courant number (**maxCo**) or time-step size (**maxDeltaT**).
- When any of these conditions is reached, the solver will stop scaling the time-step size.
- To use these features, you need to turn-on the option **adjustTimeStep**.
- Remember, the first time step of the simulation is done using the value defined with the keyword **deltaT** and then it is automatically scaled (up or down), to achieve the desired maximum values (**maxCo** and **maxDeltaT**).
- It is recommended to start the simulation with a low time-step in order to let the solver scale-up the time-step size.
- If you want to change the values on-the-fly, you need to turn-on the option **runTimeModifiable**.
- The feature **adjustTimeStep** is only present in the **PIMPLE** family solvers, but it can be added to any solver by modifying the source code.

On the CFL number

The output screen

- This is the output screen of a solver supporting the option **adjustTimeStep**.
- In this case **maxCo** is equal 2 and **maxDeltaT** is equal to 0.001.
- Notice that the solver reached the maximum allowable **maxDeltaT**.

```
Courant Number mean: 0.10863988 max: 0.73950028 ← Courant number (mean and maximum values)
deltaT = 0.001 ← Current time-step
Time = 30.000289542261612 ← Simulation time

PIMPLE: iteration 1 ← One PIMPLE iteration (outer loop), this is equivalent to PISO
DILUPBiCG: Solving for Ux, Initial residual = 0.003190933, Final residual = 1.0207483e-09, No Iterations 5
DILUPBiCG: Solving for Uy, Initial residual = 0.0049140114, Final residual = 8.5790109e-10, No Iterations 5
DILUPBiCG: Solving for Uz, Initial residual = 0.010705877, Final residual = 3.5464756e-09, No Iterations 4
GAMG: Solving for p, Initial residual = 0.024334674, Final residual = 0.0005180308, No Iterations 3
GAMG: Solving for p, Initial residual = 0.00051825089, Final residual = 1.6415538e-05, No Iterations 5
time step continuity errors : sum local = 8.768064e-10, global = 9.8389717e-11, cumulative = -2.6474162e-07
GAMG: Solving for p, Initial residual = 0.00087813032, Final residual = 1.6222017e-05, No Iterations 3
GAMG: Solving for p, Initial residual = 1.6217958e-05, Final residual = 6.4475277e-06, No Iterations 1
time step continuity errors : sum local = 3.4456296e-10, global = 2.6009599e-12, cumulative = -2.6473902e-07
ExecutionTime = 33091.06 s  ClockTime = 33214 s ← CPU time and wall clock

fieldMinMax domainminandmax output:
min(p) = -0.59404715 at location (-0.019 0.02082288 0.072) on processor 1
max(p) = 0.18373302 at location (-0.02083962 -0.003 -0.136) on processor 1
min(U) = (0.29583255 -0.4833922 -0.0048229716) at location (-0.02259661 -0.02082288 -0.072) on processor 0
max(U) = (0.59710937 0.32913292 0.020043679) at location (0.11338793 -0.03267608 0.12) on processor 3
min(nut) = 1.6594481e-10 at location (0.009 -0.02 0.024) on processor 0
max(nut) = 0.00014588174 at location (-0.02083962 0.019 0.072) on processor 1

yPlus yplus output:
patch square y+ : min = 0.44603573, max = 6.3894913, average = 2.6323389
writing field yPlus
```


Linear solvers in OpenFOAM®

- After spatial and temporal discretization and by using equation

$$\int_t^{t+\Delta t} \left[\left(\frac{\partial \rho \phi}{\partial t} \right)_P V_P + \sum_f \mathbf{S}_f \cdot (\rho \mathbf{u} \phi)_f - \sum_f \mathbf{S}_f \cdot (\rho \Gamma_\phi \nabla \phi)_f \right] dt = \int_t^{t+\Delta t} (S_c V_P + S_p V_P \phi_P) dt$$

in every control volume V_P of the domain, a system of linear algebraic equations for the transported quantity ϕ is assembled

$$\begin{pmatrix} a_{11} & a_{12} & & \ddots & & & \\ a_{21} & a_{22} & a_{23} & & \ddots & & \\ & \ddots & \ddots & \ddots & \ddots & \ddots & \\ \ddots & & \ddots & \ddots & \ddots & \ddots & \ddots \\ & a_S & & a_W & a_P & a_E & a_N \\ & & \ddots & \ddots & \ddots & \ddots & \ddots \\ & & & \ddots & \ddots & \ddots & \ddots \\ & & & & \ddots & \ddots & a_{PP} \end{pmatrix} \times \begin{pmatrix} \phi_S \\ \phi_W \\ \phi_P \\ \phi_E \\ \phi_N \end{pmatrix} = \begin{pmatrix} b_S \\ b_W \\ b_P \\ b_E \\ b_N \end{pmatrix}$$

- This system can be solved by using any iterative or direct method.

Linear solvers in OpenFOAM®

Linear solvers

```
solvers ←
{
  p
  {
    solver      PCG;
    preconditioner DIC;
    tolerance   1e-06;
    relTol      0;
  }
  pFinal
  {
    $p;
    relTol 0;
  }
  U
  {
    solver      PBiCGStab;
    preconditioner DILU;
    tolerance   1e-08;
    relTol      0;
  }
}

PISO ←
{
  nCorrectors 2;
  nNonOrthogonalCorrectors 1;
}
```

- The equation solvers, tolerances, and algorithms are controlled from the sub-dictionary **solvers** located in the *fvSolution* dictionary file.
- In the dictionary file *fvSolution*, and depending on the solver you are using you will find the additional sub-dictionaries **PISO**, **PIMPLE**, and **SIMPLE**, which will be described later.
- In this dictionary is where we are telling OpenFOAM® how to crunch numbers.
- The **solvers** sub-dictionary specifies each linear-solver that is used for each equation being solved.
- If you forget to define a linear-solver, OpenFOAM® will let you know what are you missing.
- The syntax for each entry within the **solvers** sub-dictionary uses a keyword that is the word relating to the variable being solved in the particular equation and the options related to the linear-solver.

Linear solvers in OpenFOAM®

Linear solvers

```
solvers
{
    p
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0;
    }
    pFinal
    {
        $p;
        relTol 0;
    }
    U
    {
        solver          PBiCGStab;
        preconditioner  DILU;
        tolerance       1e-08;
        relTol          0;
    }
}

PISO
{
    nCorrectors 2;
    nNonOrthogonalCorrectors 1;
}
```

The diagram illustrates the configuration of linear solvers for three variables: pressure (p), final pressure correction (pFinal), and velocity (U). The 'solvers' block contains three sub-entries. The 'p' entry is configured with the PCG solver, DIC preconditioner, an absolute tolerance of 1e-06, and a relative tolerance of 0. The 'pFinal' entry inherits the solver settings from 'p' (indicated by '\$p;') and sets a relative tolerance of 0. The 'U' entry is configured with the PBiCGStab solver, DILU preconditioner, an absolute tolerance of 1e-08, and a relative tolerance of 0. Brackets on the right side of the code block group the settings for each variable, and arrows point from the variable names to these groups.

- In this generic case, to solve the pressure (**p**) we are using the **PCG** method with the **DIC** preconditioner, an absolute **tolerance** equal to 1e-06 and a relative tolerance **relTol** equal to 0.
- The entry **pFinal** refers to the final pressure correction (notice that we are using macro syntax), and we are using a relative tolerance **relTol** equal to 0.
- To solve the velocity field (**U**) we are using the **PBiCGStab** method with the **DILU** preconditioner, an absolute **tolerance** equal to 1e-08 and a relative tolerance **relTol** equal to 0.
- The linear solvers will iterate until reaching any of the tolerance values set by the user or reaching a maximum value of iterations (optional entry).
- FYI, solving for the velocity is relative inexpensive, whereas solving for the pressure is expensive.
- The pressure equation is particularly important as it governs mass conservation.
- If you do not solve the equations accurately enough (tolerance), the physics might be wrong.
- Selection of the tolerance is of paramount importance.

Linear solvers in OpenFOAM®

Linear solvers

```
solvers
{
    p
    {
        solver          PCG;
        preconditioner   DIC;
        tolerance        1e-06;
        relTol           0;
    }
    pFinal
    {
        $p;
        relTol 0;
    }
    U
    {
        solver          PCG; ←
        preconditioner   DILU;
        tolerance        1e-08;
        relTol           0;
    }
}

PISO
{
    nCorrectors 2;
    nNonOrthogonalCorrectors 1;
}
```

- The linear solvers distinguish between symmetric matrices and asymmetric matrices.
- The symmetry of the matrix depends on the structure of the equation being solved.
- Pressure is a symmetric matrix and velocity is an asymmetric matrix.
- If you use the wrong linear solver, OpenFOAM® will complain and will let you know what options are valid.
- In the following error screen, we are using a symmetric solver for an asymmetric matrix,

→ **FOAM FATAL IO ERROR :**

Unknown asymmetric matrix solver PCG

Valid asymmetric matrix solvers are :

```
4
(
  BICCG
  GAMG
  P
  smoothSolver
)
```

Linear solvers in OpenFOAM®

Linear solvers

```
solvers
{
    p
    {
        solver          PCG;
        preconditioner   DIC;
        tolerance       1e-06;
        relTol          0;
    }
    pFinal
    {
        $p;
        relTol 0;
    }
    U
    {
        solver          PBiCGStab;
        preconditioner   DILU;
        tolerance       1e-08;
        relTol          0;
    }
}

PISO
{
    nCorrectors 2;
    nNonOrthogonalCorrectors 1;
}
```

- The linear solvers are iterative, *i.e.*, they are based on reducing the equation residual over a succession of solutions.
- The residual is a measure of the error in the solution so that the smaller it is, the more accurate the solution.
- More precisely, the residual is evaluated by substituting the current solution into the equation and taking the magnitude of the difference between the left and right hand sides (L2-norm).

$$|\mathbf{A}\phi^k - \mathbf{b}| = |\mathbf{r}^k|$$

- It is also normalized to make it independent of the scale of the problem being analyzed.

$$\text{Residual} = \frac{|\mathbf{r}|}{\text{Normalization factor}} < \text{Tolerance}$$

Linear solvers in OpenFOAM®

Linear solvers

```
solvers
{
    p
    {
        solver          PCG;
        preconditioner   DIC;
        tolerance        1e-06;
        relTol           0;
    }
    pFinal
    {
        $p;
        relTol 0;
    }
    U
    {
        solver          PBiCG;
        preconditioner   DILU;
        tolerance        1e-08;
        relTol           0;
        minIter          3;
        maxIter          100;
    }
}

PISO
{
    nCorrectors 2;
    nNonOrthogonalCorrectors 1;
}
```

- Before solving an equation for a particular field, the initial residual is evaluated based on the current values of the field.
- After each solver iteration the residual is re-evaluated. The solver stops if either of the following conditions are reached:
 - The residual falls below the solver tolerance, **tolerance**.
 - The ratio of current to initial residuals falls below the solver relative tolerance, **relTol**.
 - The number of iterations exceeds a maximum number of iterations, **maxIter**.
- The solver tolerance should represent the level at which the residual is small enough that the solution can be deemed sufficiently accurate.
- The keyword **maxIter** is optional and the default value is 1000.
- The user can also define the minimum number of iterations using the keyword **minIter**. This keyword is optional and the default value is 0.

Linear solvers in OpenFOAM®

Linear solvers

- These are the linear solvers available in OpenFOAM®:
 - **GAMG** → Multigrid solver
 - **PBiCG** → Newton-Krylov solver
 - **PBiCGStab** → Newton-Krylov solver
 - **PCG** → Newton-Krylov solver
 - **smoothSolver** → Smooth solver
 - **diagonalSolver**
- You will find the source code of the linear solvers in the following directory:
 - `$WM_PROJECT_DIR/src/OpenFOAM/matrices/lduMatrix/solvers`

Linear solvers in OpenFOAM®

Linear solvers

- When using Newton-Krylov solvers, you need to define preconditioners.
- These are the preconditioners available in OpenFOAM®:
 - **DIC**
 - **FDIC**
 - **diagonal**
 - **DILU**
 - **GAMG**
 - **noPreconditioner**
- You will find the source code in the following directory:
 - `$WM_PROJECT_DIR/src/OpenFOAM/matrices/lduMatrix/preconditioners`
- The **smoothSolver** solver requires the specification of a smoother.
- These are the smoothers available in OpenFOAM®:
 - **DIC**
 - **DILUGaussSeidel**
 - **nonBlockingGaussSeidel**
 - **DICGaussSeidel**
 - **FDIC**
 - **symGaussSeidel**
 - **DILU**
 - **GaussSeidel**
- You will find the source code in the following directory:
 - `$WM_PROJECT_DIR/src/OpenFOAM/matrices/lduMatrix/smoothers`

Linear solvers in OpenFOAM®

Linear solvers

- As you can see, when it comes to linear solvers there are many options and combinations available in OpenFOAM®.
- When it comes to choosing the linear solver, there is no written theory.
- It is problem and hardware dependent (type of the mesh, physics involved, processor cache memory, network connectivity, partitioning method, and so on).
- Most of the times using the **GAMG** method (geometric-algebraic multi-grid), is the best choice for symmetric matrices (e.g., pressure).
- The **GAMG** method should converge fast (less than 20 iterations). If it's taking more iterations, try to change the smoother.
- And if it is taking too long or it is unstable, use the **PCG** solver.
- When running with many cores (more than 1000), using the **PCG** might be a better choice.

Linear solvers in OpenFOAM®

Linear solvers

- For asymmetric matrices, the **PBiCGStab** method with **DILU** preconditioner is a good choice.
- The **smoothSolver** solver with smoother **GaussSeidel**, also performs very well.
- If the **PBiCGStab** method with **DILU** preconditioner mysteriously crashed with an error related to the preconditioner, use the **smoothSolver** or change the preconditioner.
- But in general the **PBiCGStab** solver should be faster than the **smoothSolver** solver.
- Remember, asymmetric matrices are assembled from the velocity (**U**), and the transported quantities (**k**, **omega**, **epsilon**, **T**, and so on).
- Usually, computing the velocity and the transported quantities is inexpensive and fast, so it is a good idea to use a tight tolerance (1e-8) for these fields.
- The diagonal solver is used for back-substitution, for instance, when computing density using the equation of state (we know **p** and **T**).

Linear solvers in OpenFOAM®

Linear solvers

- A few comments on the linear solvers residuals (we will talk about monitoring the residuals later on).
 - Residuals are not a direct indication that you are converging to the right solution.
 - The first time-steps the solution might not converge, this is acceptable.
 - Also, you might need to use a smaller time-step during the first iterations to maintain solver stability.
 - If the solution is not converging, try to reduce the time-step size.

```
Time = 50
```

```
Courant Number mean: 0.044365026 max: 0.16800273
```

```
smoothSolver: Solving for Ux, Initial residual = 1.0907508e-09, Final residual = 1.0907508e-09, No Iterations 0
```

```
smoothSolver: Solving for Uy, Initial residual = 1.4677462e-09, Final residual = 1.4677462e-09, No Iterations 0
```

```
DICPCG: Solving for p, Initial residual = 1.0020944e-06, Final residual = 1.0746895e-07, No Iterations 1
```

```
time step continuity errors : sum local = 4.0107145e-11, global = -5.0601748e-20, cumulative = 2.637831e-18
```

```
ExecutionTime = 4.47 s  ClockTime = 5 s
```

```
fieldMinMax minmaxdomain output:
```

```
min(p) = -0.37208345 at location (0.025 0.975 0.5)
```

```
max(p) = 0.77640927 at location (0.975 0.975 0.5)
```

```
min(U) = (0.00028445255 -0.00028138799 0) at location (0.025 0.025 0.5)
```

```
max(U) = (0.00028445255 -0.00028138799 0) at location (0.025 0.025 0.5)
```

Residuals

Linear solvers in OpenFOAM®

Linear solvers

- So how do we set the tolerances?
- The pressure equation is particularly important, so we should resolve it accurately. Solving the pressure equation is the expensive part of the whole iterative process.
- For the pressure equation you can start the simulation with a **tolerance** equal to **1e-6** and **relTol** equal to **0.01**. After a while you change these values to **1e-6** and **0.0**, respectively.
- If the solver is too slow, you can change the convergence criterion to **1e-4** and **relTol** equal to **0.05**. You usually will do this during the first iterations.

Loose tolerance

```
p
{
    solver          PCG;
    preconditioner   DIC;
    tolerance        1e-6;
    relTol           0.01;
}
```

Tight tolerance

```
p
{
    solver          PCG;
    preconditioner   DIC;
    tolerance        1e-6;
    relTol           0.0;
}
```

Linear solvers in OpenFOAM®

Linear solvers

- For the velocity field (**U**) and the transported quantities (asymmetric matrices), you can use the following criterion.
- Solving for these variables is relative inexpensive, so you can start right away with a tight tolerance

Loose tolerance

```
U
{
    solver          PBiCGStab;
    preconditioner  DILU;
    tolerance       1e-8;
    relTol          0.01;
}
```

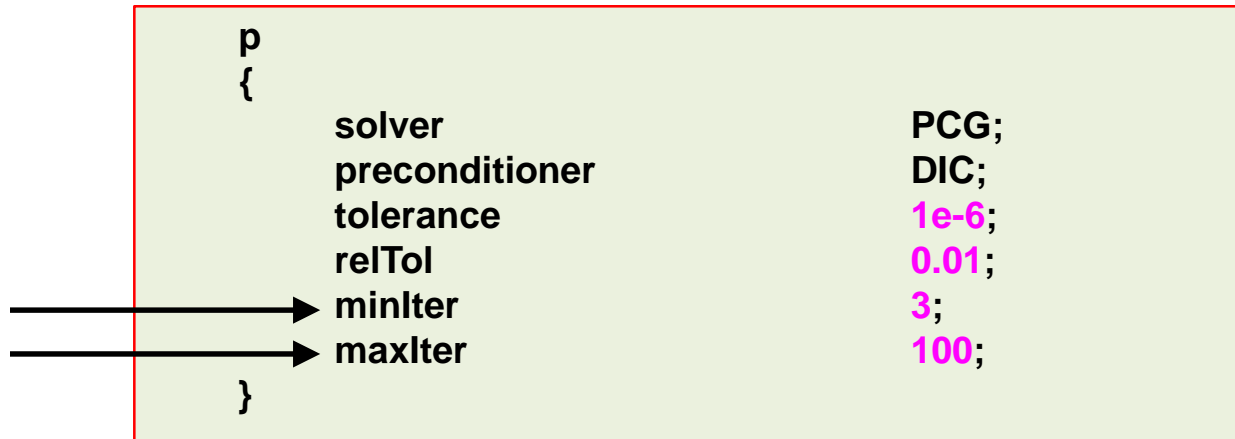
Tight tolerance

```
U
{
    solver          PBiCGStab;
    preconditioner  DILU;
    tolerance       1e-8;
    relTol          0.0;
}
```

Linear solvers in OpenFOAM®

Linear solvers

- It is also a good idea to set the minimum number of iterations (**minIter**) to 3.
- If your solver is doing too many iterations, you can set the maximum number of iterations (**maxIter**).
- But be careful, if the solver reach the maximum number of iterations it will stop, we are talking about unconverged iterations.
- Setting the maximum number of iterations is specially useful during the first time-steps where the linear solver takes longer to converge.
- You can set **minIter** and **maxIter** in all symmetric and asymmetric linear solvers.



The diagram shows a code block for a variable `p` within a curly brace. The code specifies the solver as `PCG;`, the preconditioner as `DIC;`, the tolerance as `1e-6;`, the relative tolerance as `0.01;`, the minimum number of iterations as `3;`, and the maximum number of iterations as `100;`. Two arrows point from the left to the `minIter` and `maxIter` settings, highlighting them.

```
p
{
    solver
    preconditioner
    tolerance
    relTol
    minIter
    maxIter
}
```

PCG;
DIC;
1e-6;
0.01;
3;
100;

Linear solvers in OpenFOAM®

Linear solvers

- When you use the **PISO** or **PIMPLE** method, you also have the option to set the tolerance for the final pressure corrector step (**pFinal**).
- By proceeding in this way, you can put all the computational effort only in the last corrector step (**pFinal**).
- For all the intermediate corrector steps, you can use a more relaxed convergence criterion.
- For example, you can use the following solver and tolerance criterion for all the intermediate corrector steps (**p**), then in the final corrector step (**pFinal**) you tighten the solver tolerance.

Loose tolerance for p

```
p
{
    solver          PCG;
    preconditioner   DIC;
    tolerance        1e-4;
    relTol           0.05;
}
```

Tight tolerance for pFinal

```
pFinal
{
    solver          PCG;
    preconditioner   DIC;
    tolerance        1e-6;
    relTol           0.0;
}
```

Linear solvers in OpenFOAM®

Linear solvers

- When you use the **PISO** or **PIMPLE** method, you also have the option to set the tolerance for the final pressure corrector step (**pFinal**).
- By proceeding in this way, you can put all the computational effort only in the last corrector step (**pFinal** in this case).
- For all the intermediate corrector steps (**p**), you can use a more relaxed convergence criterion.
- If you proceed in this way, it is recommended to do at least 2 corrector steps (**nCorrectors**).

```
Courant Number mean: 0.10556573 max: 0.65793603
deltaT = 0.00097959184
Time = 10
```

```
PIMPLE: iteration 1
```

```
DILUPBiCG: Solving for Ux, Initial residual = 0.0024649332, Final residual = 2.3403547e-09, No Iterations 4
```

```
DILUPBiCG: Solving for Uy, Initial residual = 0.0044355904, Final residual = 1.8966277e-09, No Iterations 4
```

```
DILUPBiCG: Solving for Uz, Initial residual = 0.010100894, Final residual = 1.4724403e-09, No Iterations 4
```

```
GAMG: Solving for p, Initial residual = 0.018497918, Final residual = 0.00058090899, No Iterations 3
```

```
GAMG: Solving for p, Initial residual = 0.00058090857, Final residual = 2.5748489e-05, No Iterations 5
```

```
time step continuity errors : sum local = 1.2367812e-09, global = 2.8865505e-11, cumulative = 1.057806e-08
```

```
GAMG: Solving for p, Initial residual = 0.00076032002, Final residual = 2.3965621e-05, No Iterations 3
```

```
GAMG: Solving for p, Initial residual = 2.3961044e-05, Final residual = 6.3151172e-06, No Iterations 2
```

```
time step continuity errors : sum local = 3.0345314e-10, global = -3.0075104e-12, cumulative = 1.0575052e-08
```

```
DILUPBiCG: Solving for omega, Initial residual = 0.00073937735, Final residual = 1.2839908e-10, No Iterations 4
```

```
DILUPBiCG: Solving for k, Initial residual = 0.0018291502, Final residual = 8.5494234e-09, No Iterations 3
```

```
ExecutionTime = 29544.18 s ClockTime = 29600 s
```

p

p

pFinal

1

2



nCorrectors

Linear solvers in OpenFOAM®

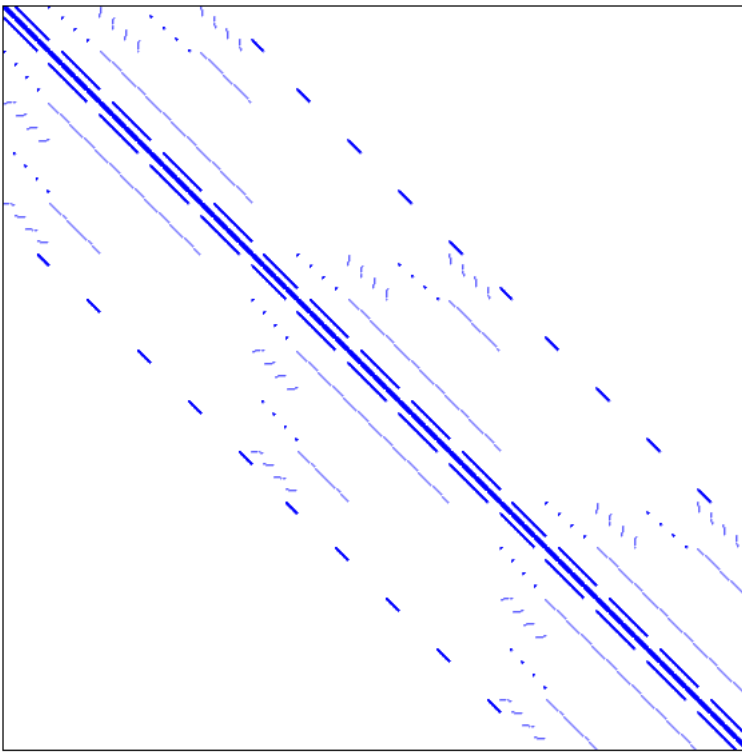
Linear solvers

- As we are solving a sparse matrix, the more diagonal the matrix is, the best the convergence rate will be.
- So it is highly advisable to use the utility `renumberMesh` before running the simulation.
 - `$> renumberMesh -overwrite`
- The utility `renumberMesh` can dramatically increase the speed of the linear solvers, specially during the firsts iterations.
- You will find the source code and the master dictionary in the following directory:
 - `$WM_PROJECT_DIR/applications/utilities/mesh/manipulation/renumberMesh`

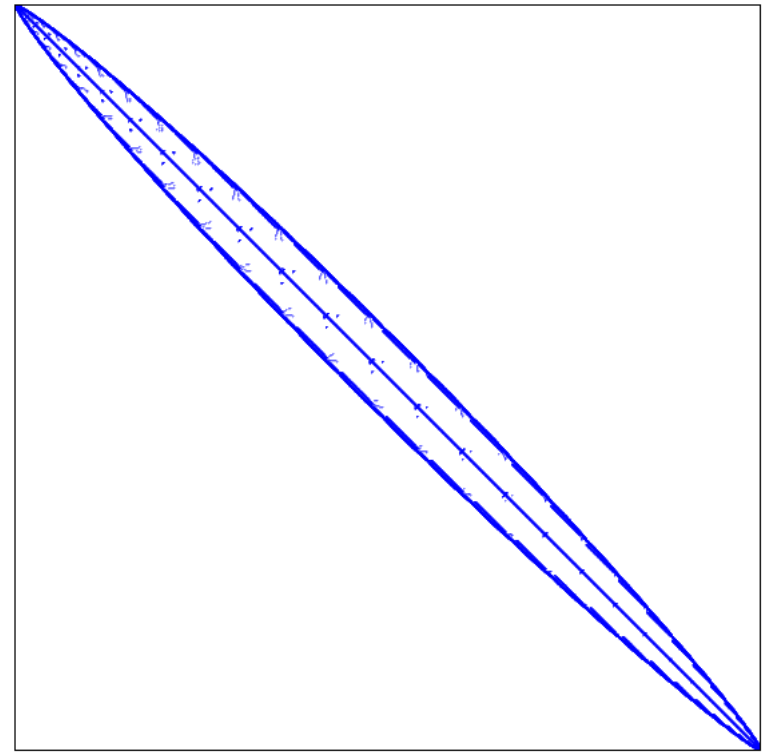
Linear solvers in OpenFOAM®

Linear solvers

- The idea behind reordering is to make the matrix more diagonally dominant, therefore, speeding up the iterative solver.



Matrix structure plot before reordering



Matrix structure plot after reordering

Note:
This is the actual pressure matrix from an OpenFOAM® model case

Linear solvers in OpenFOAM®

On the multigrid solvers

- The development of multigrid solvers (**GAMG** in OpenFOAM®), together with the development of high resolution TVD schemes and parallel computing, are among the most remarkable achievements of the history of CFD.
- Most of the time using the **GAMG** linear solver is fine. However, if you see that the linear solver is taking too long to converge or is converging in more than 100 iterations, it is better to use the **PCG** linear solver.
- Particularly, we have found that the **GAMG** linear solver in OpenFOAM® does not perform very well when you scale your computations to more than 500 processors.
- Also, we have found that for some multiphase cases the **PCG** method outperforms the **GAMG**.
- But again, this is problem and hardware dependent.
- As you can see, you need to always monitor your simulations (stick to the screen for a while). Otherwise, you might end-up using a solver that is performing poorly. And this translate in increased computational time and costs.

Linear solvers in OpenFOAM®

On the multigrid solvers tolerances

- If you go for the **GAMG** linear solver for symmetric matrices (e.g., pressure), the following tolerances are acceptable for most of the cases.

Loose tolerance for p

```
p
{
    solver          GAMG;
    tolerance        1e-6;
    relTol           0.01;
    smoother         GaussSeidel;
    nPreSweeps        0;
    nPostSweeps       2;
    cacheAgglomeration on;
    agglomerator       faceAreaPair;
    nCellsInCoarsestLevel 100;
    mergeLevels       1;
    minIter           3;
}
```

Tight tolerance for pFinal

```
pFinal
{
    solver          GAMG;
    tolerance        1e-6;
    relTol           0;
    smoother         GaussSeidel;
    nPreSweeps        0;
    nPostSweeps       2;
    cacheAgglomeration on;
    agglomerator       faceAreaPair;
    nCellsInCoarsestLevel 100;
    mergeLevels       1;
    minIter           3;
}
```

NOTE:

The GAMG parameters are not optimized, that is up to you.
Most of the times is safe to use the default parameters.

Linear solvers in OpenFOAM®

Linear solvers tolerances – Steady simulations

- The previous tolerances are fine for unsteady solver.
- For extremely coupled problems you might need to have tighter tolerances.
- You can use the same tolerances for steady solvers. However, it is acceptable to use a looser criterion.
- You can also set the convergence controls based on residuals of fields. The controls are specified in the **residualControls** sub-dictionary of the dictionary file *fvSolution*.

```
SIMPLE
```

```
{
```

```
    nNonOrthogonalCorrectors 2;
```

```
    residualControl
```

```
{
```

```
        p 1e-4;
```

```
        U 1e-4;
```

```
}
```

```
}
```

Residual control for every
field variable you are solving

Linear solvers in OpenFOAM®

Exercises

- Choose any tutorial or a case of your own and do a benchmarking of the linear solvers.
- Using your benchmarking case, find the optimal parameters for the **GAMG** solver.
- Use different linear solvers for **p** and **pFinal** (symmetric matrices). Do you see any advantage?
- Do a benchmarking of the different reordering methods available
(Hint: look for the dictionary **reorderMeshDict**)
- Compare the performance of the asymmetric solvers **PBiCG**, **PBiCGStab**, and **smoothSolver**. Do you see any significant difference between both solvers?
- Is it possible to switch between segregated and coupled linear solvers on-the-fly?
- In what files are located the controls of the **SIMPLE**, **PISO**, and **PIMPLE** methods?
(Hint: for example, using **grep** look for the keyword **nCorrectors** in the directory **src/finiteVolume**)

Pressure-Velocity coupling in OpenFOAM®

- To solve the Navier-Stokes equations we need to use a solution approach able to deal with the nonlinearities of the governing equations and with the coupled set of equations.

$$\begin{aligned}\frac{\partial \rho}{\partial t} + \nabla \cdot (\rho \mathbf{u}) &= 0, \\ \frac{\partial (\rho \mathbf{u})}{\partial t} + \nabla \cdot (\rho \mathbf{u} \mathbf{u}) &= -\nabla p + \nabla \cdot \tau, \\ \frac{\partial (\rho e_t)}{\partial t} + \nabla \cdot (\rho e_t \mathbf{u}) &= \nabla \cdot q - \nabla \cdot (p \mathbf{u}) + \tau : \nabla \mathbf{u}, \\ &+ \end{aligned}$$

Additional equations deriving from models, such as, volume fraction, chemical reactions, turbulence modeling, combustion, multi-species, etc.

Pressure-Velocity coupling in OpenFOAM®

- Many numerical methods exist to solve the Navier-Stokes equations, just to name a few:
 - Pressure-correction methods (Predictor-Corrector type).
 - SIMPLE, SIMPLEC, SIMPLER, PISO.
 - Projection methods.
 - Fractional step (operator splitting), MAC, SOLA.
 - Density-based methods and preconditioned solvers.
 - Riemann solvers, ROE, HLLC, AUSM+, ENO, WENO.
 - Artificial compressibility methods.
 - Artificial viscosity methods.
- The most widely used approaches for solving the NSE are:
 - Pressure-based approach (predictor-corrector).
 - Density-based approach.
- Historically speaking, the pressure-based approach was developed for low-speed incompressible flows, while the density-based approach was mainly developed for high-speed compressible flows.
- However, both methods have been extended and reformulated to solve and operate for a wide range of flow conditions beyond their original intent.

Pressure-Velocity coupling in OpenFOAM®

- Pressure-based methods are the default option in most of CFD solvers (OpenFOAM® included).
- Pressure-based methods are intrinsically implicit.
- Two pressure-based solution methods are generally available, namely:
 - Segregated method.
 - Coupled method.
- In the pressure-based approach the velocity field is obtained from the momentum equations (there is some mathematical manipulation involved).
- In the segregated algorithm, the individual governing equations for the primitive variables are solved one after another.
- The coupled approach solves the continuity, momentum, and energy equation simultaneously, that is, coupled together.
- The segregated algorithm is memory-efficient, since the discretized equations need only be stored in the memory one at a time.
- However, the solution convergence is relatively slow (in comparison to coupled solvers) as the equations are solved one at a time.
- In OpenFOAM®, you will find segregated pressure-based solvers.
- But coupled pressure-based solvers are under active development.

Pressure-Velocity coupling in OpenFOAM®

- In OpenFOAM®, you will find segregated pressure-based solvers.
- The following methods are available:
 - **SIMPLE** (Semi-Implicit Method for Pressure-Linked Equations)
 - **SIMPLEC** (SIMPLE Corrected/Consistent)
 - **PISO** (Pressure Implicit with Splitting Operators)
- Additionally, you will find something called **PIMPLE**, which is an hybrid between **SIMPLE** and **PISO**. This method is formulated for very large time-steps and pseudo-transient simulations.
- You will find the solvers in the following directory:
 - `$WM_PROJECT_DIR/applications/solvers`

Pressure-Velocity coupling in OpenFOAM®

- In OpenFOAM®, the **PISO** and **PIMPLE** methods are formulated for unsteady simulations.
- Whereas, the **SIMPLE** and **SIMPLEC** methods are formulated for steady simulations.
- If conserving time is not a priority, you can use the **PIMPLE** method for pseudo transient simulations.
- The pseudo transient **PIMPLE** method is more stable than the **SIMPLE** method, but it has a higher computational cost.
- Depending on the method and solver you are using, you will need to define a specific sub-dictionary in the dictionary file *fvSolution*.
- For instance, if you are using the **PISO** method, you will need to specify the **PISO** sub-dictionary.
- And depending on the method, each sub-dictionary will have different entries.

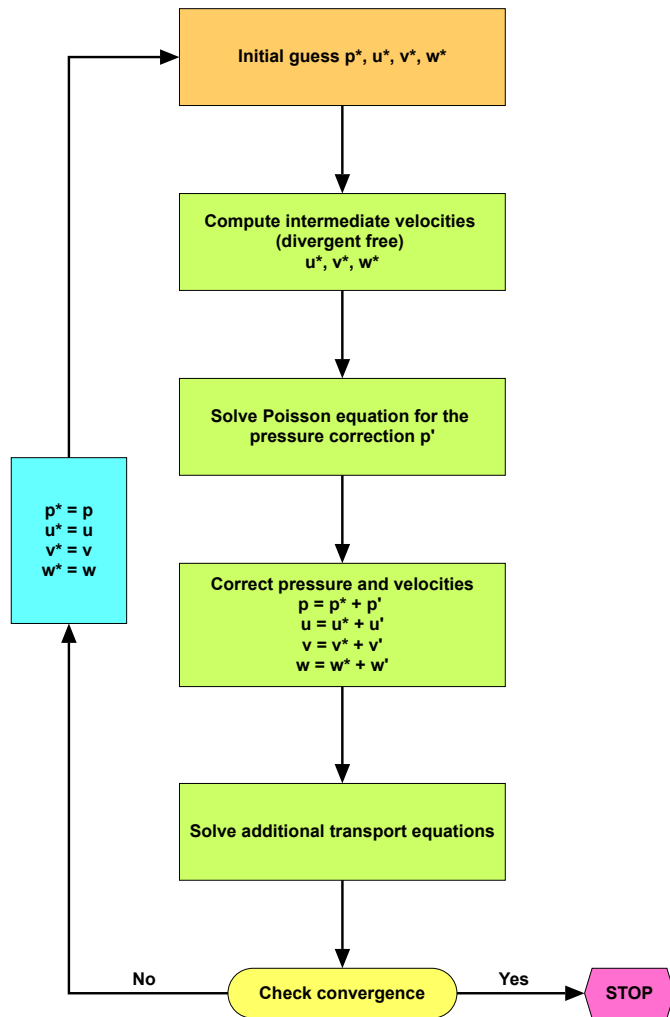
Pressure-Velocity coupling in OpenFOAM®

On the origins of the methods

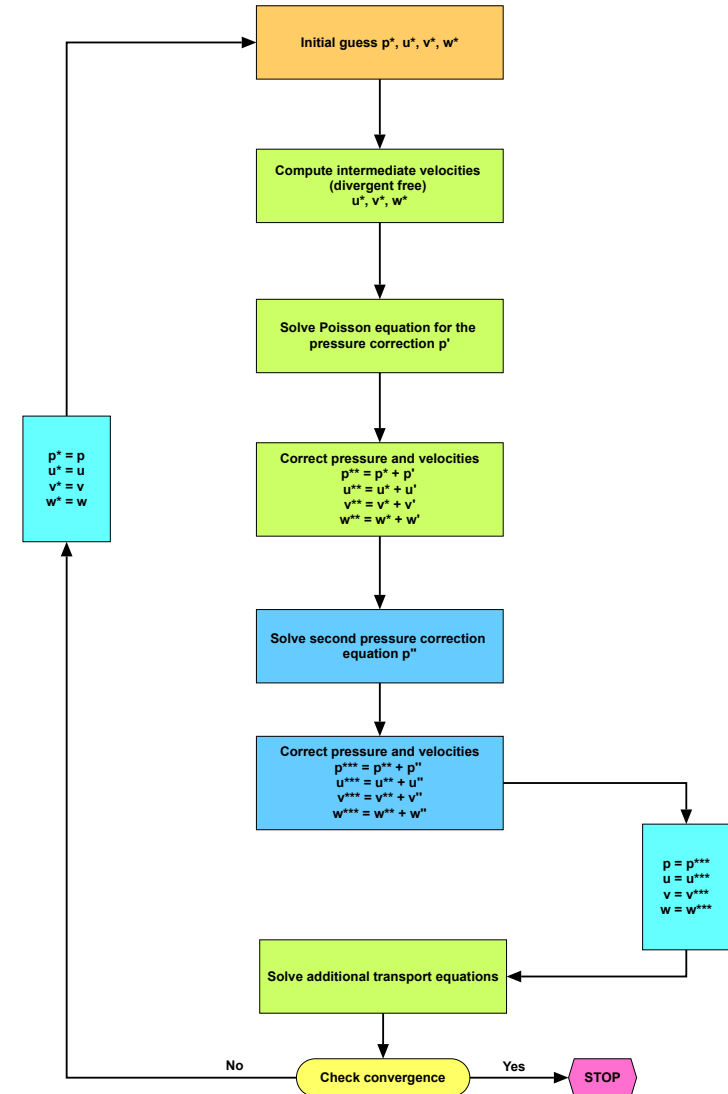
- **SIMPLE**
 - S. V. Patankar and D. B. Spalding, “A calculation procedure for heat, mass and momentum transfer in three-dimensional parabolic flows”, Int. J. Heat Mass Transfer, 15, 1787-1806 (1972).
- **SIMPLE-C**
 - J. P. Van Doormaal and G. D. Raithby, “Enhancements of the SIMPLE method for predicting incompressible fluid flows”, Numer. Heat Transfer, 7, 147-163 (1984).
- **PISO**
 - R. I. Issa, “Solution of the implicitly discretized fluid flow equations by operator-splitting”, J. Comput. Phys., 62, 40-65 (1985).
- **PIMPLE**
 - Unknown origins.
 - Useful reference:
 - I. E. Barton, “Comparison of SIMPLE and PISO-type algorithms for transient flows, Int. J. Numerical methods in fluids, 26,459-483 (1998).

Pressure-Velocity coupling in OpenFOAM®

Pressure-velocity coupling using the SIMPLE method



Pressure-velocity coupling using the PISO method



Pressure-Velocity coupling in OpenFOAM®

The SIMPLE sub-dictionary

- This sub-dictionary is located in the dictionary file *fvSolution*.
- It controls the options related to the **SIMPLE** pressure-velocity coupling method.
- The **SIMPLE** method only makes 1 correction.
- An additional correction to account for mesh non-orthogonality is available when using the **SIMPLE** method. The number of non-orthogonal correctors is specified by the **nNonOrthogonalCorrectors** keyword.
- The number of non-orthogonal correctors is chosen according to the mesh quality. For orthogonal meshes you can use 0, whereas, for non-orthogonal meshes it is recommended to do at least 1 correction.
- However, it is strongly recommended to do at least 1 non-orthogonal correction.

SIMPLE

{

nNonOrthogonalCorrectors 1;

}

Pressure-Velocity coupling in OpenFOAM®

The SIMPLE sub-dictionary

- You can use the optional keyword **consistent** to enable or disable the **SIMPLEC** method.
- This option is disabled by default.
- In the **SIMPLEC** method, the cost per iteration is marginally higher but the convergence rate is better so the number of iterations can be reduced.
- The **SIMPLEC** method relaxes the pressure in a consistent manner and additional relaxation of the pressure is not generally necessary.
- In addition, convergence of the **p-U** system is better and still is reliable with less aggressive relaxation of the momentum equation.

SIMPLE

{

consistent **yes;**
nNonOrthogonalCorrectors **1;**

}

Pressure-Velocity coupling in OpenFOAM®

The SIMPLE sub-dictionary

- Typical under-relaxation factors for the **SIMPLE** and **SIMPLEC** methods.
- Remember the under-relaxation factors are problem dependent.

SIMPLE

```
relaxationFactors
{
    fields
    {
        p          0.3;
    }
    equations
    {
        U          0.7;
        k          0.7;
        omega      0.7;
    }
}
```

SIMPLEC

```
relaxationFactors
{
    equations
    {
        U          0.9;
        k          0.9;
        omega      0.9;
    }
}
```


Pressure-Velocity coupling in OpenFOAM®

The SIMPLE sub-dictionary

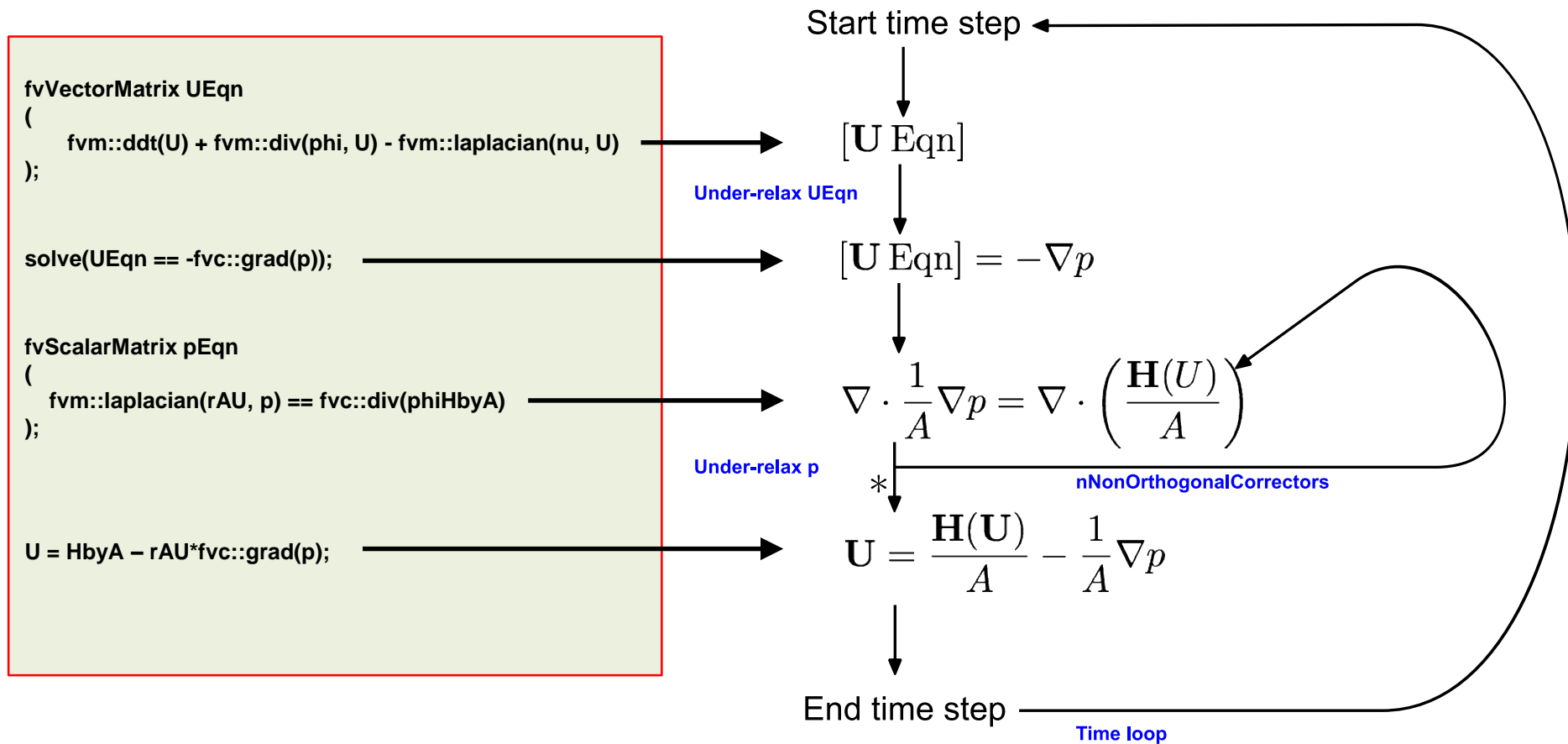
- If you are planning to use the **SIMPLEC** method, we recommend you to use under-relaxation factors are little bit more loose that the commonly recommended values.
- If during the simulation you still have some stability problems, try to reduce all the values to 0.5.
- Remember the under-relaxation factors are problem dependent.
- It is also recommended to start the simulation with low values (about 0.3), and then increase the values slowly up to 0.7.

SIMPLEC

```
relaxationFactors
{
    fields
    {
        p      0.7;
    }
    equations
    {
        p      0.7;
        U      0.7;
        k      0.7;
        omega  0.7;
    }
}
```

Pressure-Velocity coupling in OpenFOAM®

The SIMPLE loop in OpenFOAM®



$$\ast \phi = \mathbf{S}_f \cdot [(\mathbf{H}/A)_f - (1/A)_f(\nabla p)_f]$$

Pressure-Velocity coupling in OpenFOAM®

The PISO sub-dictionary

- This sub-dictionary is located in the dictionary file *fvSolution*.
- It controls the options related to the **PISO** pressure-velocity coupling method.
- The **PISO** method requires at least one correction (**nCorrectors**). To improve accuracy and stability you can increase the number of corrections.
- For good accuracy and stability, it is recommended to use 2 **nCorrectors**.
- An additional correction to account for mesh non-orthogonality is available when using the **PISO** method. The number of non-orthogonal correctors is specified by the **nNonOrthogonalCorrectors** keyword.
- The number of non-orthogonal correctors is chosen according to the mesh quality. For orthogonal meshes you can use 0, whereas, for non-orthogonal meshes it is recommended to do at least 1 correction.

PISO

{

nCorrectors 2;

nNonOrthogonalCorrectors 1;

}

Pressure-Velocity coupling in OpenFOAM®

The PISO sub-dictionary

- You can use the optional keyword **momentumPredictor** to enable or disable the momentum predictor step.
- The momentum predictor helps in stabilizing the solution as we are computing better approximations for the velocity.
- It is clear that this will add an extra computational cost, which is negligible.
- In most of the solvers, this option is enabled by default.
- It is recommended to use this option for highly convective flows (high Reynolds number). If you are working with low Reynolds flow or creeping flows it is recommended to turn it off.
- Remember, when you enable the option **momentumPredictor**, you will need to define the linear solvers for the variables **.*Final** (we are using regex notation).

PISO

{

momentumPredictor yes;

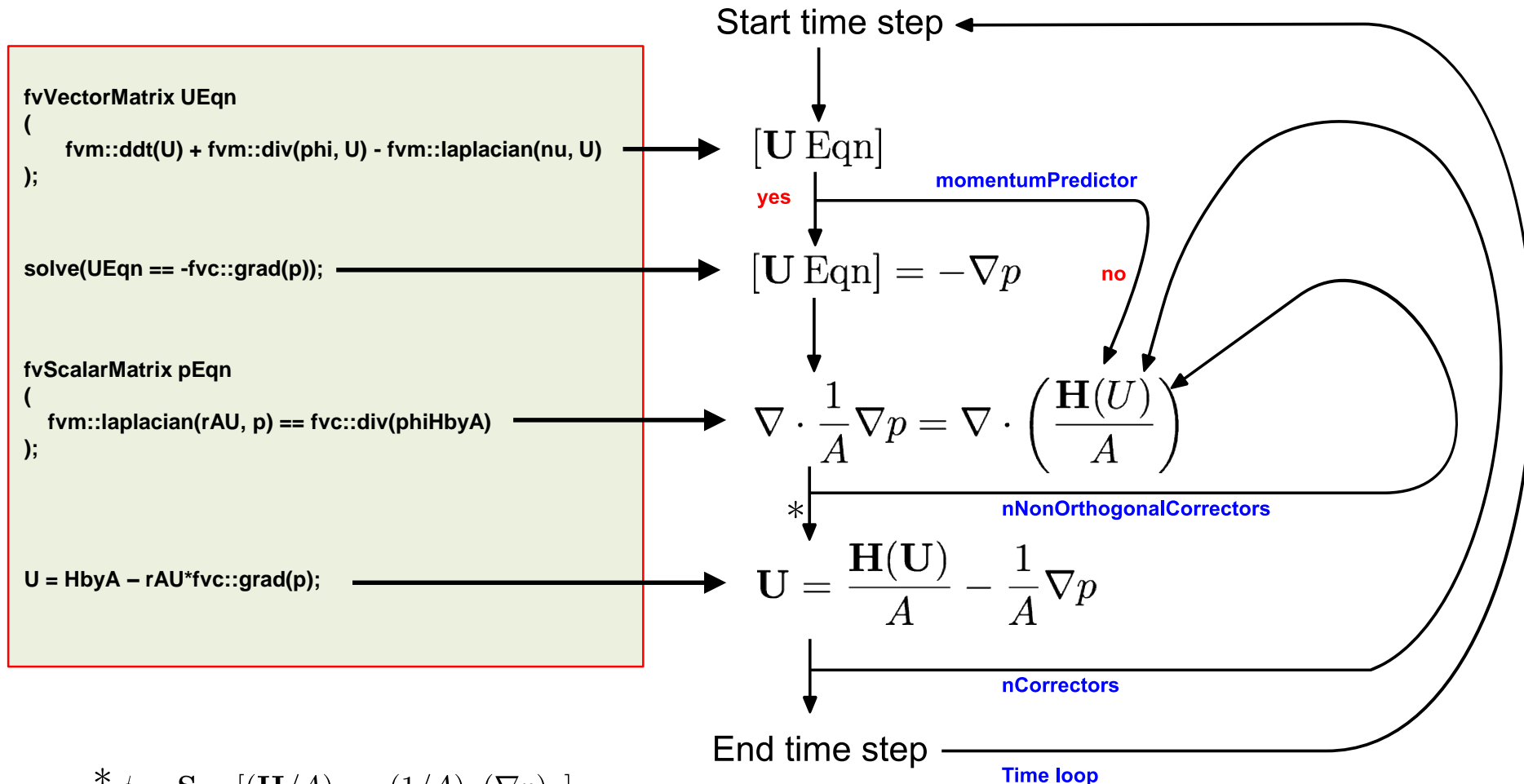
nCorrectors 2;

nNonOrthogonalCorrectors 1;

}

Pressure-Velocity coupling in OpenFOAM®

The PISO loop in OpenFOAM®



Pressure-Velocity coupling in OpenFOAM®

The PIMPLE sub-dictionary

- This sub-dictionary is located in the dictionary file *fvSolution*.
- It controls the options related to the **PIMPLE** pressure-velocity coupling method.
- The **PIMPLE** method works very similar to the **PISO** method. In fact, setting the keyword **nOuterCorrectors** to 1 is equivalent to running using the **PISO** method.
- The keyword **nOuterCorrectors** controls a loop outside the **PISO** loop.
- To gain more stability, especially when using large time-steps, you can use more outer correctors (**nOuterCorrectors**). Have in mind that this will highly increase the computational cost.
- Also, if you use under-relaxation factors, your solution is not anymore time accurate. You are working in pseudo-transient simulations mode.

PIMPLE

{

momentumPredictor yes;

nOuterCorrectors 1;

nCorrectors 2;

nNonOrthogonalCorrectors 1;

}

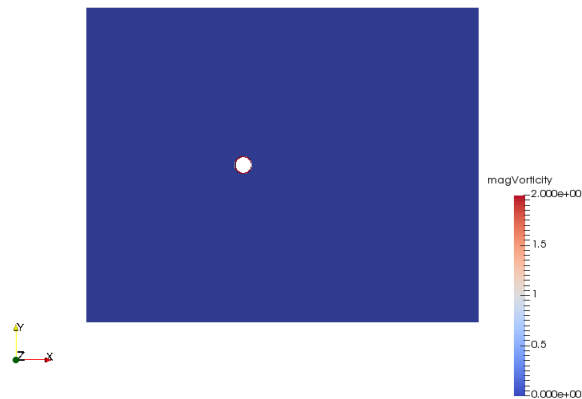
Unsteady and steady simulations

- A few examples of unsteady applications:

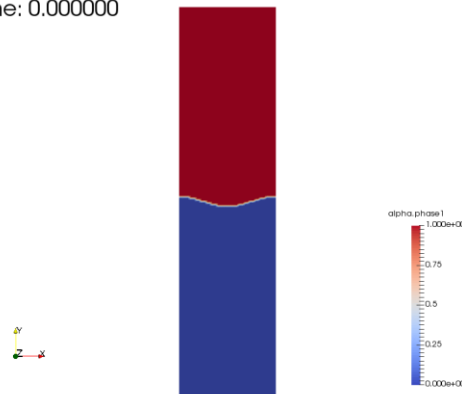
Vortex shedding

www.wolfdynamics.com/wiki/FVM_uns/ani1.gif

Time: 0.000000



Time: 0.000000



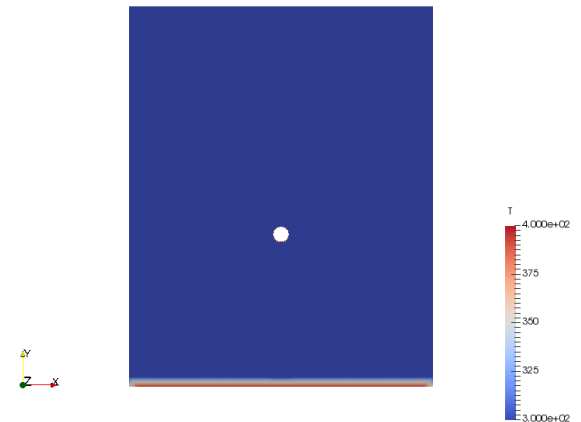
Multiphase flow

www.wolfdynamics.com/wiki/FVM_uns/ani3.gif

Buoyant flow

www.wolfdynamics.com/wiki/FVM_uns/ani2.gif

Time: 0.000000

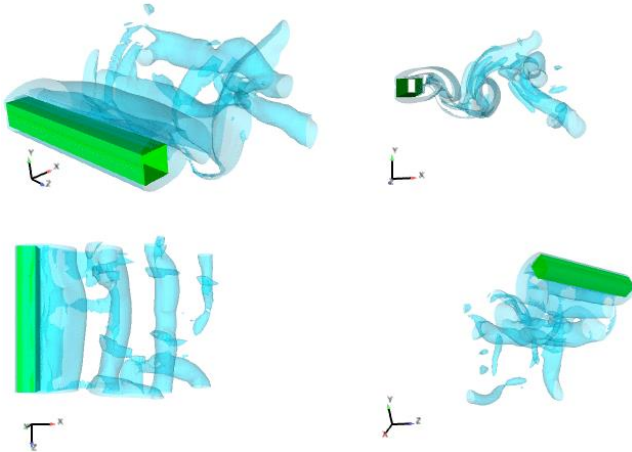


Unsteady and steady simulations

- A few examples of unsteady applications:

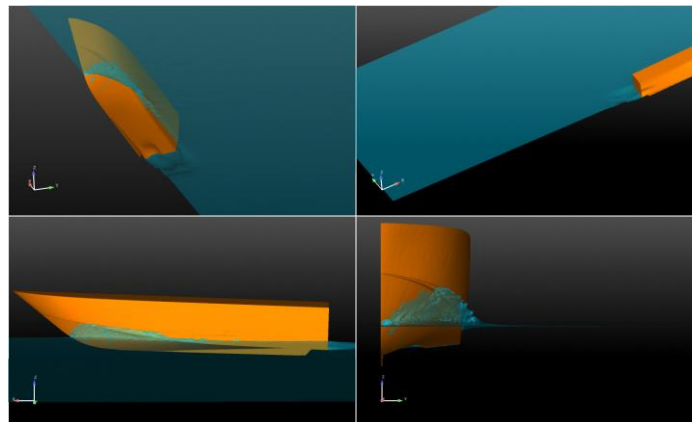
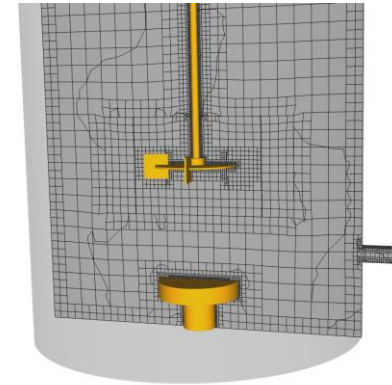
Turbulent flows - SRS

www.wolfdynamics.com/wiki/FVM_uns/ani4.gif



Sliding grids – Continuous stirred tank reactor

www.wolfdynamics.com/wiki/FVM_uns/ani5.gif



Marine applications - Sea keeping

www.wolfdynamics.com/wiki/FVM_uns/ani6.gif

Unsteady and steady simulations

- Nearly all flows in nature and industrial applications are unsteady (also known as transient or time-dependent).
- Unsteadiness is due to:
 - Instabilities.
 - Non-equilibrium initial conditions.
 - Time-dependent boundary conditions.
 - Source terms.
 - Chemical reactions.
 - Moving or deforming bodies.
 - Turbulence.
 - Buoyancy.
 - Convection.
 - Multiple phases

Unsteady and steady simulations

How to run unsteady simulations in OpenFOAM®?

- Select the time step. The time-step must be chosen in such a way that it resolves the time-dependent features and maintains solver stability.
- Select the temporal discretization scheme.
- Set the tolerance (absolute and/or relative) of the linear solvers.
- Monitor the CFL number.
- Monitor the stability and boundedness of the solution.
- Monitor a quantity of interest.
- And of course, you need to save the solution with a given frequency.
- Have in mind that unsteady simulations generate a lot of data.
- End time of the simulation?, it is up to you.
- In the *controlDict* dictionary you need to set runtime parameters and general instructions on how to run the case (such as time step and maximum CFL number). You also set the saving frequency.
- In the *fvSchemes* dictionary you need to set the temporal discretization scheme.
- In the *fvSolution* dictionary you need to set the linear solvers.
- Also, you will need to set the number of corrections of the velocity-pressure coupling method used (e.g. **PISO** or **PIMPLE**), this is done in the *fvSolution* dictionary.
- Additionally, you may set **functionObjects** in the *controlDict* dictionary. The **functionObjects** are used to do sampling, probing and co-processing while the simulation is running.

Unsteady and steady simulations

How to run unsteady simulations in OpenFOAM®?

```
startFrom    latestTime;
startTime    0; ←
stopAt       endTime;
endTime      10; ←
deltaT       0.0001; ←
writeControl runTime;
writeInterval 0.1; ←
purgeWrite   0;
writeFormat  ascii;
writePrecision 8;
writeCompression off;
timeFormat   general;
timePrecision 6;
runTimeModifiable yes; ←
adjustTimeStep yes;
maxCo        2.0;
maxDeltaT    0.001;
```

- The *controlDict* dictionary contains runtime simulation controls, such as, start time, end time, time step, saving frequency and so on. Most of the entries are self-explanatory.
- This generic case starts from time 0 (**startTime**), and it will run up to 10 seconds (**endTime**).
- It will write the solution every 0.1 seconds (**writeInterval**) of simulation time (**runTime**).
- The time step of the simulation is 0.0001 seconds (**deltaT**).
- It will keep all the solution directories (**purgeWrite**).
- It will save the solution in ascii format (**writeFormat**) with a precision of 8 digits (**writePrecision**).
- And as the option **runTimeModifiable** is on (**yes**), we can modify all these entries while we are running the simulation.

Unsteady and steady simulations

How to run unsteady simulations in OpenFOAM®?

```
startFrom    latestTime;

startTime    0;

stopAt       endTime;

endTime      10;

deltaT       0.0001;

writeControl  runtime;

writeInterval 0.1;

purgeWrite   0;

writeFormat   ascii;

writePrecision 8;

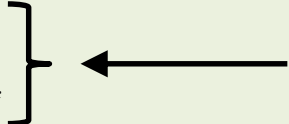
writeCompression off;

timeFormat    general;

timePrecision 6;

runTimeModifiable yes;

adjustTimeStep yes;
maxCo          2.0;
maxDeltaT      0.001;
```



- In this generic case, the solver supports adjustable time-step (**adjustTimeStep**).
- The option **adjustTimeStep** will automatically adjust the time step to achieve the maximum desired courant number (**maxCo**) or time-step size (**maxDeltaT**).
- When any of these conditions is reached, the solver will stop scaling the time-step size.
- Remember, the first time step of the simulation is done using the value defined with the keyword **deltaT** and then it is automatically scaled (up or down), to achieve the desired maximum values (**maxCo** and **maxDeltaT**).
- It is recommended to start the simulation with a low time-step in order to let the solver scale-up the time-step size.
- The feature **adjustTimeStep** is only present in the **PIMPLE** family solvers, but it can be added to any solver by modifying the source code.
- If you are planning to use large time steps (CFL much higher than 1), it is recommended to do at least 3 correctors steps (**nCorrectors**) in **PISO/PIMPLE** loop.

Unsteady and steady simulations

How to run unsteady simulations in OpenFOAM®?

```
startFrom    latestTime;

startTime    0;

stopAt       endTime;

endTime      10;

deltaT       0.0001;

writeControl adjustableRunTime; ←

writeInterval 0.1;

purgeWrite   0;

writeFormat  ascii;

writePrecision 8;

writeCompression off;

timeFormat   general;

timePrecision 6;

runTimeModifiable yes;

adjustTimeStep yes;
maxCo          2.0;
maxDeltaT      0.001; } ←
```

- A word of caution about adjustable time-step (**adjustTimeStep**).
- This option will automatically adjust the time step to achieve the maximum desired courant number (**maxCo**) or time-step size (**maxDeltaT**).
- If the **maxDeltaT** condition is not reached, the solver will adapt the time-step to achieve the target **maxCo**, and as the time-step is not fixed this might introduce spurious oscillations in the solution.
- It is recommended to use this option at the beginning of the simulation and as soon as the solution stabilizes try fixed the time-step.
- Also, try to avoid using adjustable time step together with the option **adjustableRunTime**.
- The option **adjustableRunTime** will adjust the time-step to save the solution at the precise write intervals, and this might introduce numerical oscillations due to the fact that the time-step is changing.
- Also, the fact that you are using an adaptive time-step can have a negative effect when doing signal analysis.

Unsteady and steady simulations

How to run unsteady simulations in OpenFOAM®?

```
ddtSchemes ←  $\frac{\partial \phi}{\partial t}$ 
{
    default backward;
}

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

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

laplacianSchemes
{
    default Gauss linear orthogonal;
}

interpolationSchemes
{
    default linear;
}

snGradSchemes
{
    default orthogonal;
}
```

- The *fvSchemes* dictionary contains the information related to time discretization and spatial discretization schemes.
- In this generic case we are using the **backward** method for time discretization (**ddtSchemes**).
- This scheme is second order accurate but oscillatory.
- The parameters can be changed on-the-fly.

Unsteady and steady simulations

How to run unsteady simulations in OpenFOAM®?

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

    pFinal
    {
        $p;
        relTol 0;
    }

    "U.*"
    {
        solver          smoothSolver;
        smoother         symGaussSeidel;
        tolerance        1e-08;
        relTol           0;
    }
}

PIMPLE
{
    nOuterCorrectors 1;
    nCorrectors      2;
    nNonOrthogonalCorrectors 1;
}
```

- The *fvSolution* dictionary contains the instructions of how to solve each discretized linear equation system.
- As for the *controlDict* and *fvSchemes* dictionaries, the parameters can be changed on-the-fly.
- In this generic case, to solve the pressure (**p**) we are using the **PCG** method with the **DIC** preconditioner, an absolute **tolerance** equal to 1e-06 and a relative tolerance **relTol** equal to 0.
- The entry **pFinal** refers to the final pressure correction (notice that we are using macro syntax), and we are using a relative tolerance **relTol** equal to 0.
- To solve **U** and **UFinal** (**U.*** using regex), we are using the **smoothSolver** method with an absolute **tolerance** equal to 1e-08 and a relative tolerance **relTol** equal to 0.
- The solvers will iterate until reaching any of the tolerance values set by the user or reaching a maximum value of iterations (optional entry).
- FYI, solving for the velocity is relative inexpensive, whereas solving for the pressure is expensive.

Unsteady and steady simulations

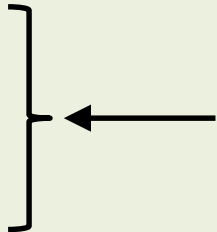
How to run unsteady simulations in OpenFOAM®?

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

    pFinal
    {
        $p;
        relTol 0;
    }

    "U.*"
    {
        solver          smoothSolver;
        smoother         symGaussSeidel;
        tolerance        1e-08;
        relTol           0;
    }
}

PIMPLE
{
    nOuterCorrectors 1;
    nCorrectors      2;
    nNonOrthogonalCorrectors 1;
}
```



- The *fvSolution* dictionary also contains the **PIMPLE** and **PISO** sub-dictionaries.
- The **PIMPLE** sub-dictionary contains entries related to the pressure-velocity coupling method (the **PIMPLE** method).
- Setting the keyword **nOuterCorrectors** to 1 is equivalent to running using the **PISO** method.
- Remember, you need to do at least one **PISO** loop (**nCorrectors**).
- To gain more stability, especially when using large time-steps, you can use more outer correctors (**nOuterCorrectors**).
- Adding corrections increase the computational cost (**nOuterCorrectors** and **nCorrectors**).
- In this generic case, we are using 1 outer correctors (**nOuterCorrectors**), 2 inner correctors or **PISO** correctors (**nCorrectors**), and 1 correction due to non-orthogonality (**nNonOrthogonalCorrectors**).
- If you are using large time steps (CFL much higher than 1), it is recommended to do at least 3 correctors steps (**nCorrectors**) in **PISO/PIMPLE** loop.

Unsteady and steady simulations

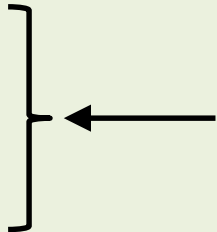
How to run unsteady simulations in OpenFOAM®?

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

    pFinal
    {
        $p;
        relTol 0;
    }

    U
    {
        solver          smoothSolver;
        smoother         symGaussSeidel;
        tolerance       1e-08;
        relTol          0;
    }
}

PISO
{
    nCorrectors 2;
    nNonOrthogonalCorrectors 1;
}
```

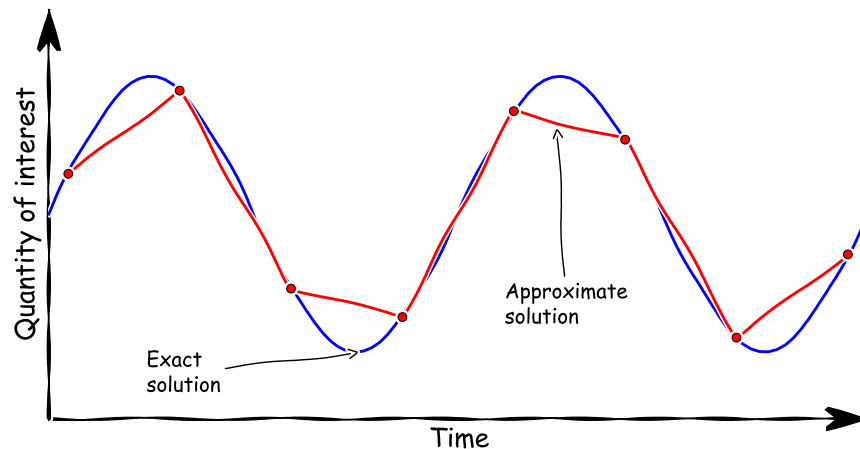
A diagram consisting of a large right-facing curly bracket and an arrow pointing left towards it, positioned to the right of the PISO sub-dictionary in the code block.

- If you use the **PISO** method for pressure-velocity coupling, you will need to define the **PISO** sub-dictionary.
- In this generic case we are doing two **PISO** corrections and one orthogonal correction.
- You need to do at least one **PISO** loop (**nCorrectors**).
- If you are using large time steps (CFL much higher than 1), it is recommended to do at least 3 correctors steps (**nCorrectors**) in **PISO/PIMPLE** loop.

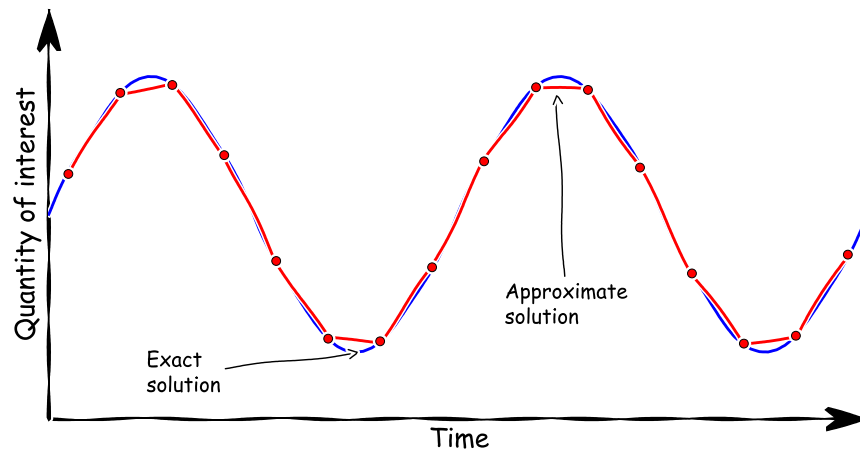
Unsteady and steady simulations

How to choose the time-step in unsteady simulations and monitor the solution

- Remember, when running unsteady simulations the time-step must be chosen in such a way that it resolves the time-dependent features and maintains solver stability.



When you use large time steps you do not resolve well the physics

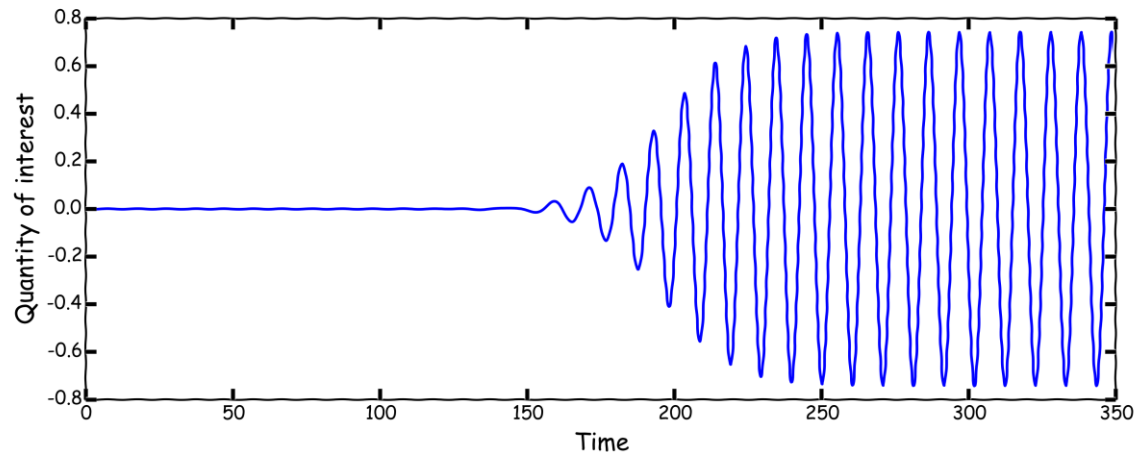
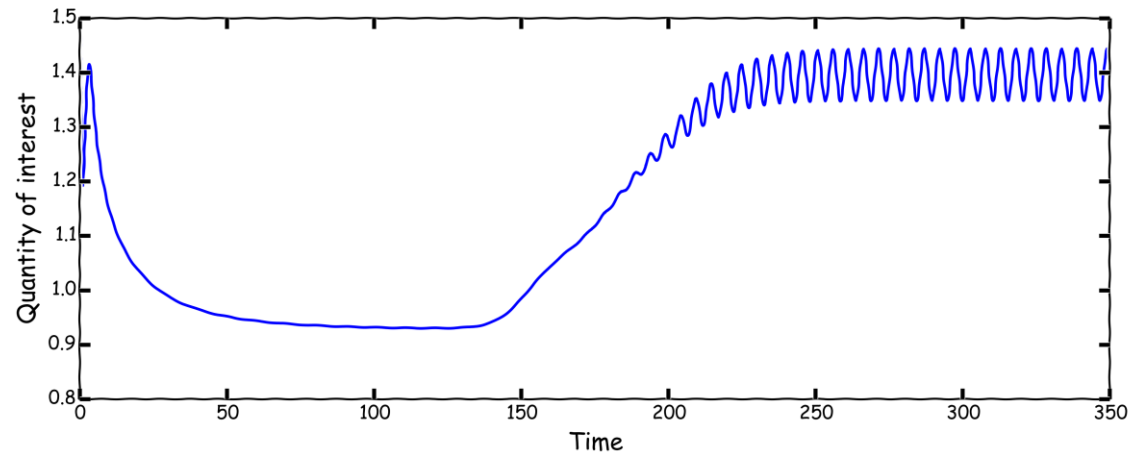


By using a smaller time step you resolve better the physics and you gain stability

Unsteady and steady simulations

Monitoring unsteady simulations

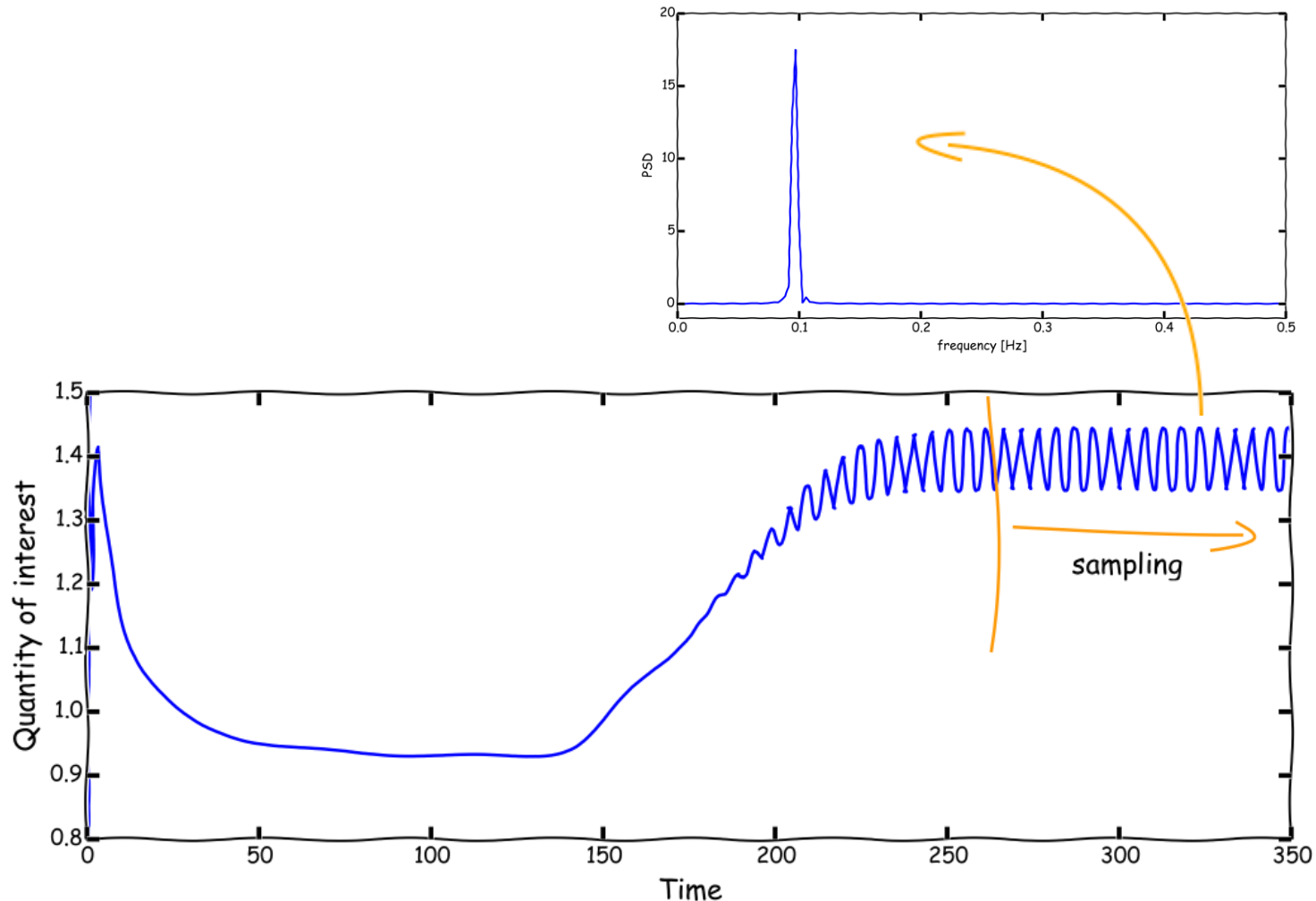
- When running unsteady simulations, it is highly advisable to monitor a quantity of interest.
- The quantity of interest can fluctuate in time, this is an indication of unsteadiness.



Unsteady and steady simulations

Sampling unsteady simulations

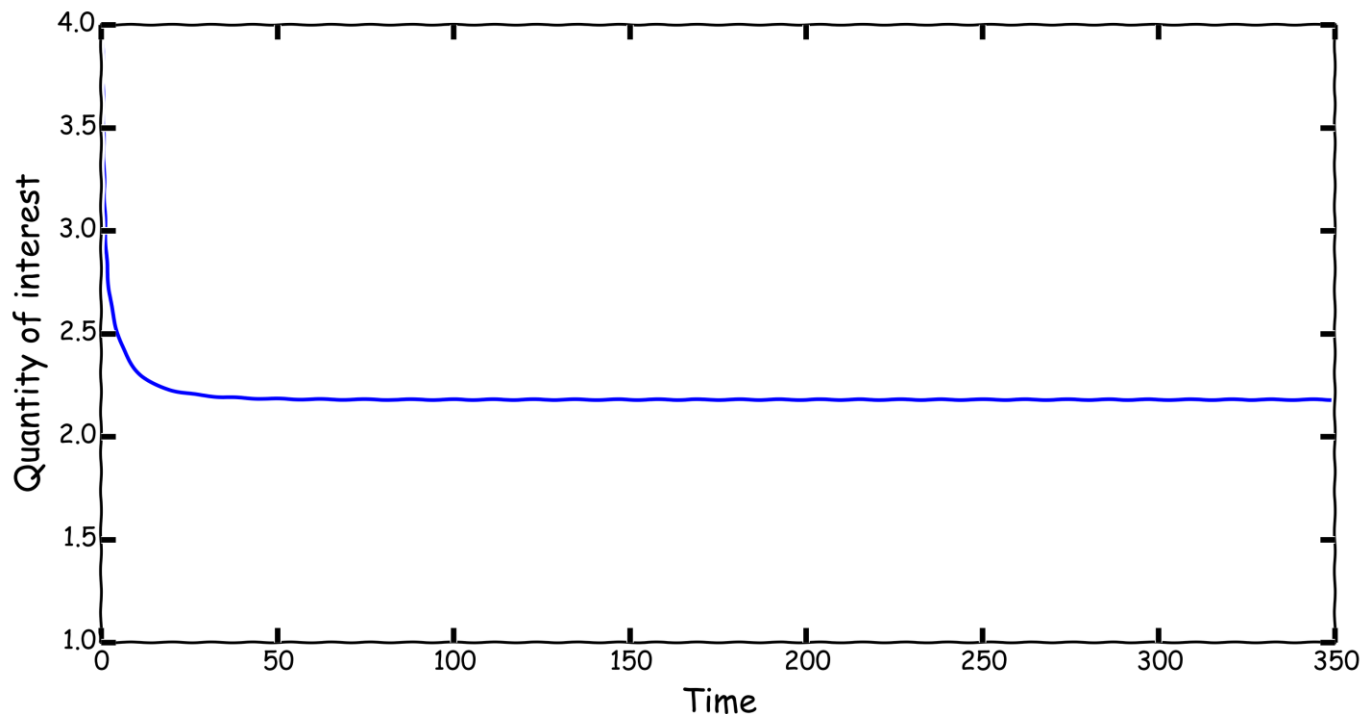
- Remember to choose wisely where to do the sampling.



Unsteady and steady simulations

I am running an unsteady simulations and the QOI does not change

- When you run unsteady simulations, flow variables can stop changing with time. When this happens, we say we have arrived to a steady state.
- Remember this is the exception rather than the rule.
- If you use a steady solver, you will arrive to the same solution (maybe not), in much less iterations.



Unsteady and steady simulations

What about steady simulations?

- First of all, steady simulations are a big simplification of reality.
- Steady simulations is a trick used by CFDers to get fast outcomes with results that might be even more questionable.
- As mentioned before, most of the flows you will encounter are unsteady.
- In steady simulations we made two assumptions:
 - We ignore unsteady fluctuations. That is, we neglect the temporal derivative in the governing equations.
 - We perform time averaging when dealing with stationary turbulence (RANS modeling)
- The advantage of steady simulations is that they require low computational resources, give fast outputs, and are easier to post-process and analyze.
- In OpenFOAM® is possible to run steady simulation.
- To do so, you need to use the appropriate solver and use the right discretization scheme.
- As you are not solving the temporal derivative, you do not need to set the time step. However, you need to tell OpenFOAM® how many iterations you would like to run.
- You can also set the residual controls (**residualControl**), in the *fvSolution* dictionary file. You set the **residualControl** in the **SIMPLE** sub-dictionary.
- If you do not set the residual controls, OpenFOAM® will run until reaching the maximum number of iterations (**endTime**).

Unsteady and steady simulations

What about steady simulations?

- You also need to set the under-relaxation factors.
- The under-relaxation factors control the change of the variable ϕ .

$$\phi_P^n = \phi_P^{n-1} + \alpha(\phi_P^{n*} - \phi_P^{n-1})$$

- If $\alpha < 1$ we are using under-relaxation.
- Under-relaxation is a feature typical of steady solvers using the **SIMPLE** method.
- If you do not set the under-relaxation factors, OpenFOAM® will use the default hard-wired values (1.0 for all field variables or no under-relaxation).
- These are the under-relaxation factors commonly used with **SIMPLE** (industry standard),

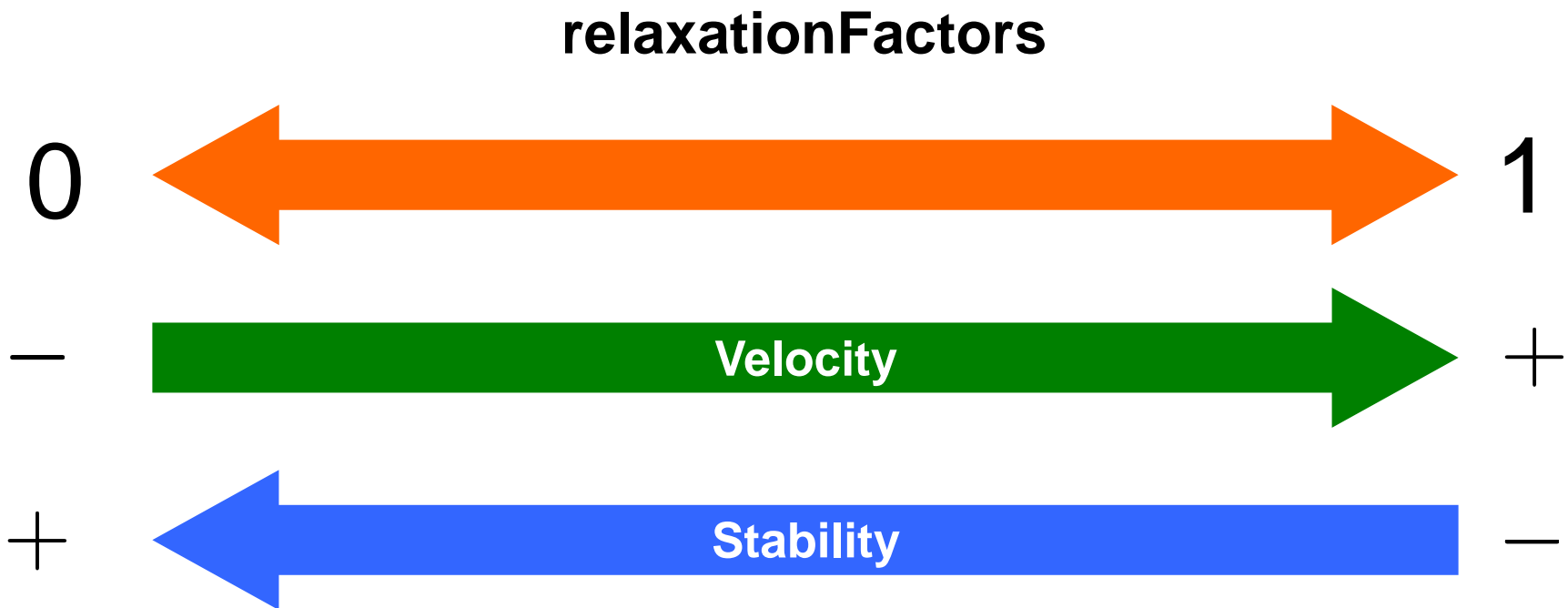
p	0.3;
U	0.7;
k	0.7;
omega	0.7;
epsilon	0.7;

- According to the physics involved you will need to add more under-relaxation factors.
- Finding the right under-relaxation factors involved experience and a lot of trial and error.

Unsteady and steady simulations

What about steady simulations?

- The under-relaxation factors are bounded between 0 and 1.



- Selecting the under-relaxation factors it is kind of equivalent to selecting the right time step.

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

- In the *controlDict* dictionary you need to set runtime parameters and general instructions on how to run the case (such as the number of iterations to run). You also set the saving frequency.
- In the *fvSchemes* dictionary you need to set the temporal discretization scheme, for steady simulations it must be **steadyState**.
- In the *fvSolution* dictionary you need to set the linear solvers, under-relaxation factors and residual controls.
- Also, you will need to set the number of corrections of the velocity-pressure coupling method used (e.g. **SIMPLE**), this is done in the *fvSolution* dictionary.
- Additionally, you may set **functionObjects** in the *controlDict* dictionary. The **functionObjects** are used to do sampling, probing and co-processing while the simulation is running.

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

```
startFrom    latestTime;
startTime    0; ←
stopAt       endTime;
endTime      10000; ←
deltaT       1; ←
writeControl runTime;
writeInterval 100; ←
purgeWrite   10; ←
writeFormat  ascii;
writePrecision 8;
writeCompression off;
timeFormat   general;
timePrecision 6;
runTimeModifiable yes; ←
```

- The *controlDict* dictionary contains runtime simulation controls, such as, start time, end time, time step, saving frequency and so on. Most of the entries are self-explanatory.
- As we are doing a steady simulation, let us talk about iterations instead of time (seconds).
- This generic case starts from iteration 0 (**startTime**), and it will run up to 10000 iterations (**endTime**).
- It will write the solution every 100 iterations (**writeInterval**) of simulation time (**runTime**).
- It will advance the solution one iteration at a time (**deltaT**).
- It will keep the last 10 saved solutions (**purgeWrite**).
- It will save the solution in ascii format (**writeFormat**) with a precision of 8 digits (**writePrecision**).
- And as the option **runTimeModifiable** is on (**true**), we can modify all these entries while we are running the simulation.

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

```
ddtSchemes
{
    default    steadyState; ←  $\frac{\partial \phi}{\partial t}$ 
}

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

divSchemes
{
    default    none;
    div(phi,u) bounded Gauss linear;
}

laplacianSchemes
{
    default    Gauss linear orthogonal;
}

interpolationSchemes
{
    default    linear;
}

snGradSchemes
{
    default    orthogonal;
}
```

- The *fvSchemes* dictionary contains the information related to time discretization and spatial discretization schemes.
- In this generic case and as we are interested in using a steady solver, we are using the **steadyState** method for time discretization (**ddtSchemes**).
- It is not a good idea to switch between steady and unsteady schemes on-the-fly.
- For steady state cases, the bounded form can be applied to the divSchemes, in this case, div(phi,U) **bounded** Gauss linear.
- This adds a linearized, implicit source contribution to the transport equation of the form,

$$\frac{\partial \mathbf{u}}{\partial t} + \nabla \cdot (\mathbf{u}\mathbf{u}) - (\nabla \cdot \mathbf{u})\mathbf{u} = \nabla \cdot (\Gamma \nabla \mathbf{u}) + S$$

- This term removes a component proportional to the continuity error. This acts as a convergence aid to tend towards a bounded solution as the calculation proceeds.
- At convergence, this term becomes zero and does not contribute to the final solution.

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

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

    U
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-08;
        relTol          0;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 2;

    residualControl
    {
        p 1e-4;
        U 1e-4;
    }
}
```

- The *fvSolution* dictionary contains the instructions of how to solve each discretized linear equation system.
- As for the *controlDict* and *fvSchemes* dictionaries, the parameters can be changed on-the-fly.
- In this generic case, to solve the pressure (**p**) we are using the **PCG** method with the **DIC** preconditioner, an absolute **tolerance** equal to 1e-06 and a relative tolerance **relTol** equal to 0.
- To solve **U** we are using the **smoothSolver** method with an absolute **tolerance** equal to 1e-08 and a relative tolerance **relTol** equal to 0.
- The solvers will iterate until reaching any of the tolerance values set by the user or reaching a maximum value of iterations (optional entry).

Unsteady and steady simulations

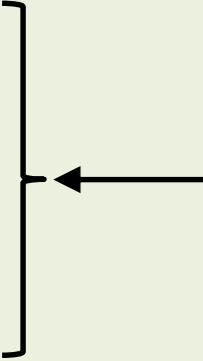
How to run steady simulations in OpenFOAM®?

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

    U
    {
        solver          smoothSolver;
        smoother        symGaussSeidel;
        tolerance       1e-08;
        relTol          0;
    }
}

SIMPLE
{
    nNonOrthogonalCorrectors 2;

    residualControl
    {
        p 1e-4;
        U 1e-4;
    }
}
```



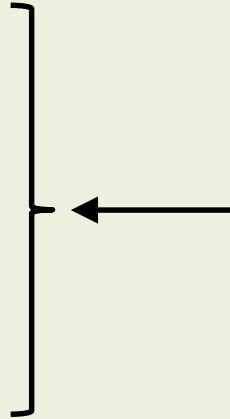
- The *fvSolution* dictionary also contains the **SIMPLE** sub-dictionary .
- The **SIMPLE** sub-dictionary contains entries related to the pressure-velocity coupling method (the **SIMPLE** method).
- Increasing the number of **nNonOrthogonalCorrectors** corrections will add more stability but at a higher computational cost.
- Remember, **nNonOrthogonalCorrectors** is used to improve the gradient computation due to mesh quality.
- In this generic case, we are doing 2 corrections due to non-orthogonality (**nNonOrthogonalCorrectors**).
- The **SIMPLE** sub-dictionary also contains convergence controls based on residuals of fields. The controls are specified in the **residualControls** sub-dictionary.
- The user needs to specify a tolerance for one or more solved fields and when the residual for every field falls below the corresponding residual, the simulation terminates.
- If you do not set the **residualControls**, the solver will iterate until reaching the maximum number of iterations set in the *controlDict* dictionary.

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

relaxationFactors

```
{  
  fields  
  {  
    p 0.3;  
  }  
  equations  
  {  
    U 0.7;  
  }  
}
```



- The *fvSolution* dictionary also contains the **relaxationFactors** sub-dictionary.
- The **relaxationFactors** sub-dictionary which controls under-relaxation, is a technique used for improving stability when using steady solvers.
- Under-relaxation works by limiting the amount which a variable changes from one iteration to the next, either by modifying the solution matrix and source (**equations** keyword) prior to solving for a field or by modifying the field directly (**fields** keyword).
- An optimum choice of under-relaxation factors is one that is small enough to ensure stable computation but large enough to move the iterative process forward quickly.
- These are the under-relaxation factors commonly used (**SIMPLE**),

```
fields  
{  
    p      0.3;  
}  
equations  
{  
    U      0.7;  
    k      0.7;  
    omega  0.7;  
}
```

Unsteady and steady simulations

Steady simulations vs. Unsteady simulations

- Steady simulations require less computational power than unsteady simulations.
- They are also much faster than unsteady simulations.
- But sometimes they do not converge to the right solution.
- They are easier to post-process and analyze (you just need to take a look at the last saved solution).
- You can use the solution of an unconverged steady simulation as initial conditions for an unsteady simulation.
- Remember, steady simulations are not time accurate, therefore we can not use them to compute temporal statistics or compute the shedding frequency

