

Module 6

Finite volume method overview

Roadmap

- 1. Finite Volume Method: A Crash Introduction**
2. On the CFL number
3. Linear solvers in OpenFOAM®
4. Pressure-Velocity coupling in OpenFOAM®
5. Unsteady and steady simulations
6. Understanding residuals
7. Boundary and initial conditions
8. Numerical playground

Finite Volume Method: A Crash introduction

- This a brief introduction to the FVM to illustrate some basic concepts.
- There is much more under the hood.
- We will use the general transport equation as the starting point to explain the FVM,

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

- 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 or any set of equations that can be derived from the general transport equation.

Finite Volume Method: A Crash introduction

- This a brief introduction to the FVM to illustrate some basic concepts.
- There is much more under the hood.
- We will use the general transport equation as the starting point to explain the FVM,

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

Problem statement

- Find the approximate solution to the general transport equation for the transported quantity ϕ in a given domain, with given boundary conditions (BC) and initial conditions (IC).
- It is an initial boundary value problem (IBVP).
- This is a second order equation. Therefore, 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 (in space and time).

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,

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

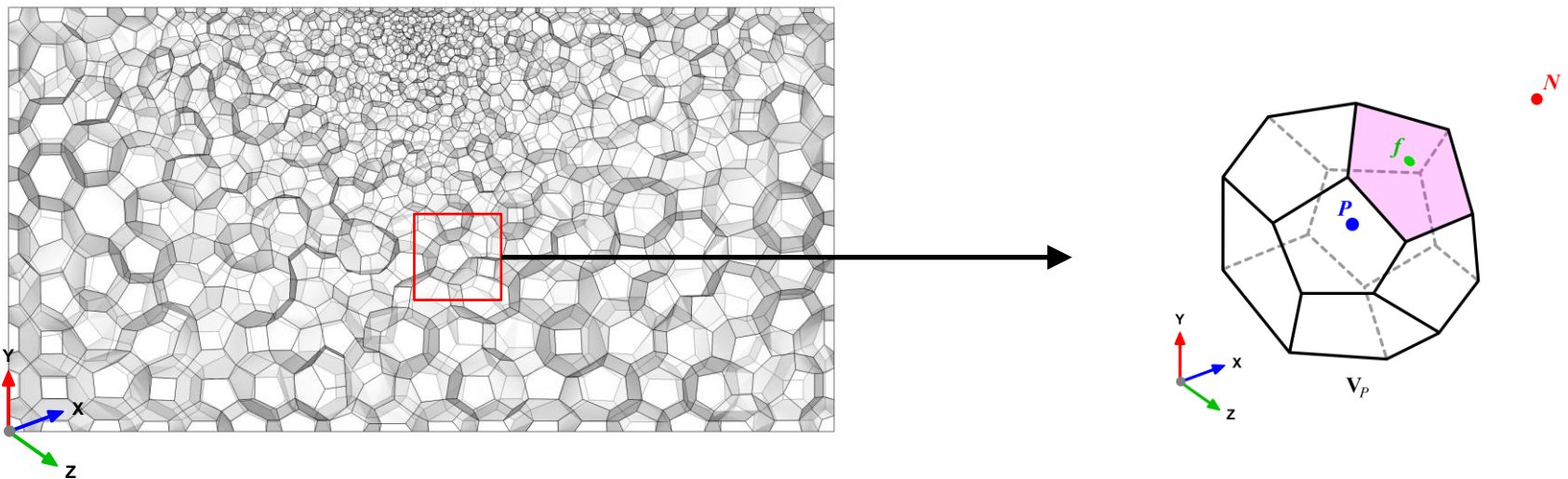
$$\phi(t + \delta t) = \phi^t + \delta t \left(\frac{\partial \phi}{\partial t} \right)^t \quad \text{where} \quad \phi^t = \phi(t)$$

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

Finite Volume Method: A Crash introduction

Domain discretization – Mesh information and variable arrangement

- Domain discretization (or mesh generation), consist in dividing 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 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.



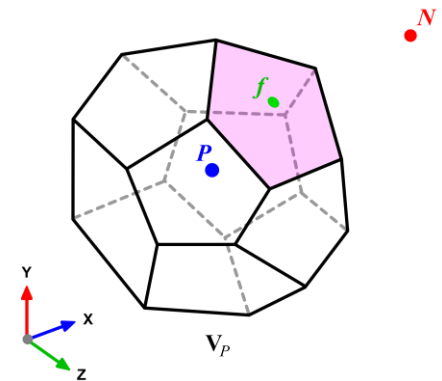
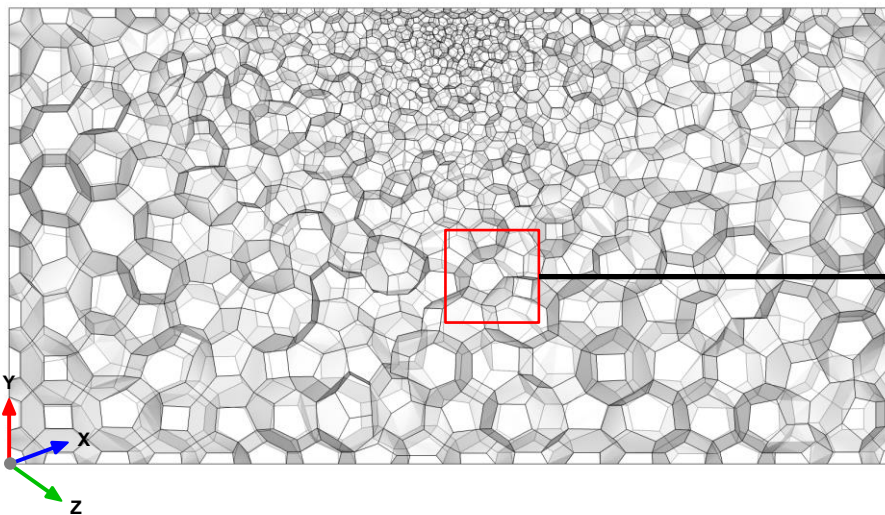
Finite Volume Method: A Crash introduction

Domain discretization – Mesh information and variable arrangement

- In the control volume illustrated, the centroid P and face center f are known.
- We also 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 cell centered collocated arrangement.
- All approximations used so far are at least second order accurate.



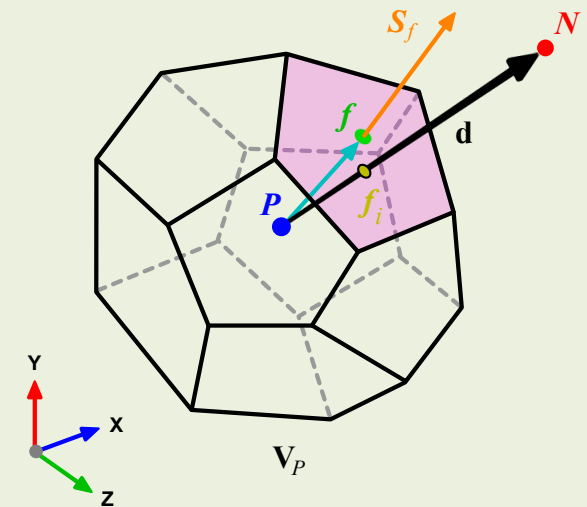
Finite Volume Method: A Crash introduction

Domain discretization – Mesh information and variable arrangement

- Putting all together, it is a lot geometrical information that we need to track.
- A lot of overhead goes into the data book-keeping.
- At the end of the day, the FVM simply consist in conservation of the transported quantities and interpolating information from cell centers to face centers.

Summary:

- 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 the neighboring control volume 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 Pf .



Finite Volume Method: A Crash introduction

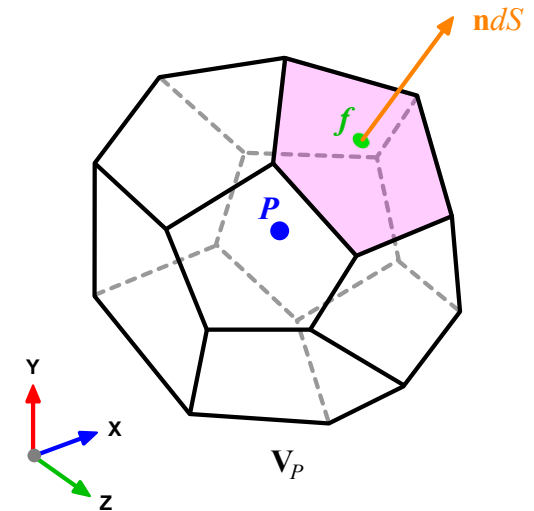
Gauss theorem and face fluxes computation

- 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

Gauss theorem and face fluxes computation

- 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.
- That is, we need to compute the gradient terms, source terms, and convective and diffusive fluxes across the faces.

Finite Volume Method: A Crash introduction

Gauss theorem and face fluxes computation

- 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:

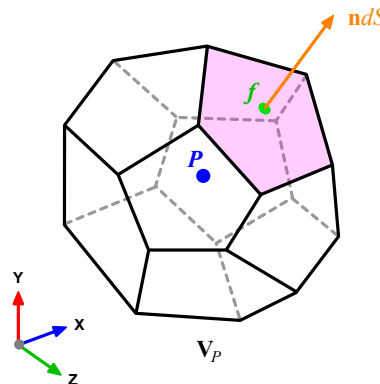
$$\underbrace{\int_{V_P} \nabla \cdot (\rho \mathbf{u} \phi) dV}_{\text{convective term}} = \underbrace{\oint_{\partial V_P} d\mathbf{S} \cdot (\rho \mathbf{u} \phi)}_{\text{convective flux}} = \sum_f \int_f d\mathbf{S} \cdot (\rho \mathbf{u} \phi)_f \approx \underbrace{\sum_f \mathbf{S}_f \cdot (\overline{\rho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

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

Gauss theorem and face fluxes computation

- 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:

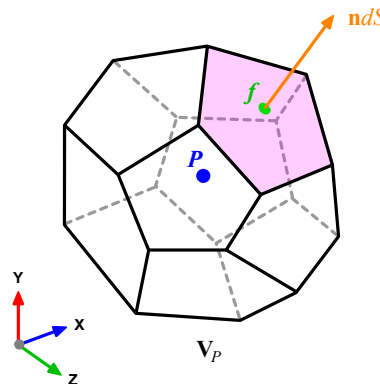
$$\underbrace{\int_{V_P} \nabla \cdot (\rho \Gamma_\phi \nabla \phi) dV}_{\text{diffusion term}} = \underbrace{\oint_{\partial V_P} d\mathbf{S} \cdot (\rho \Gamma_\phi \nabla \phi)}_{\text{diffusive flux}} = \sum_f \int_f d\mathbf{S} \cdot (\rho \Gamma_\phi \nabla \phi)_f \approx \sum_f \underbrace{\mathbf{S}_f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

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

Gauss theorem and face fluxes computation

- 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:

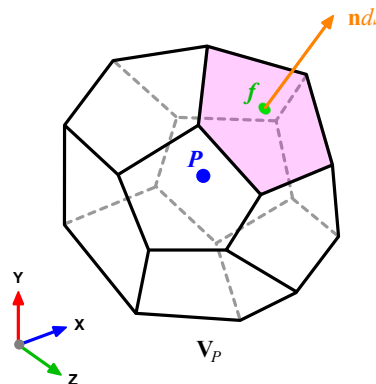
$$(\nabla \phi)_P = \frac{1}{V_P} \sum_f (\mathbf{S}_f \phi_f)$$

where we have approximated the centroid gradients by using the Gauss theorem.

This method is second order accurate and is known as Gauss cell-based.

Gauss theorem:

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



Note:

- There are more methods for gradients computation, e.g., least squares, node-based reconstruction, and so on.
- As there is some algebra involved, we do not provide the demonstration.

Finite Volume Method: A Crash introduction

Gauss theorem and face fluxes computation

- 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:

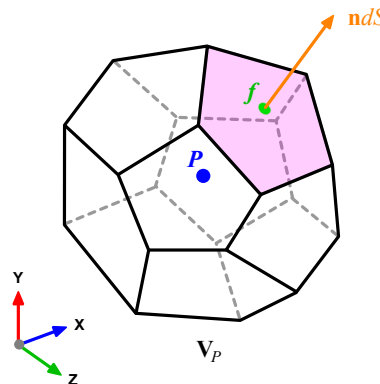
$$\int_{V_P} S_\phi(\phi) dV = S_c V_P + S_p V_P \phi_P$$

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

Gauss theorem and face fluxes computation

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

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

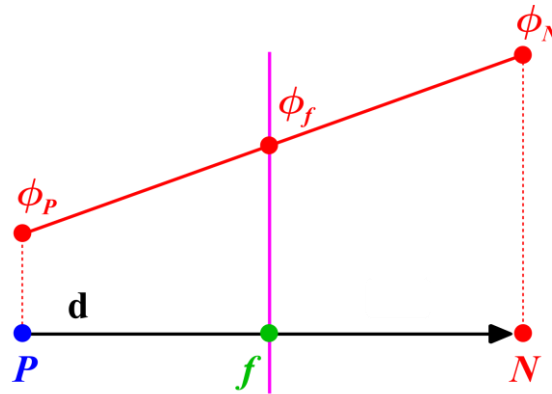
where $\mathbf{S} \cdot (\rho \mathbf{u} \phi) = F^C$ is the convective flux and $\mathbf{S} \cdot (\rho \Gamma_\phi \nabla \phi) = F^D$ is the diffusive flux.

- 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,



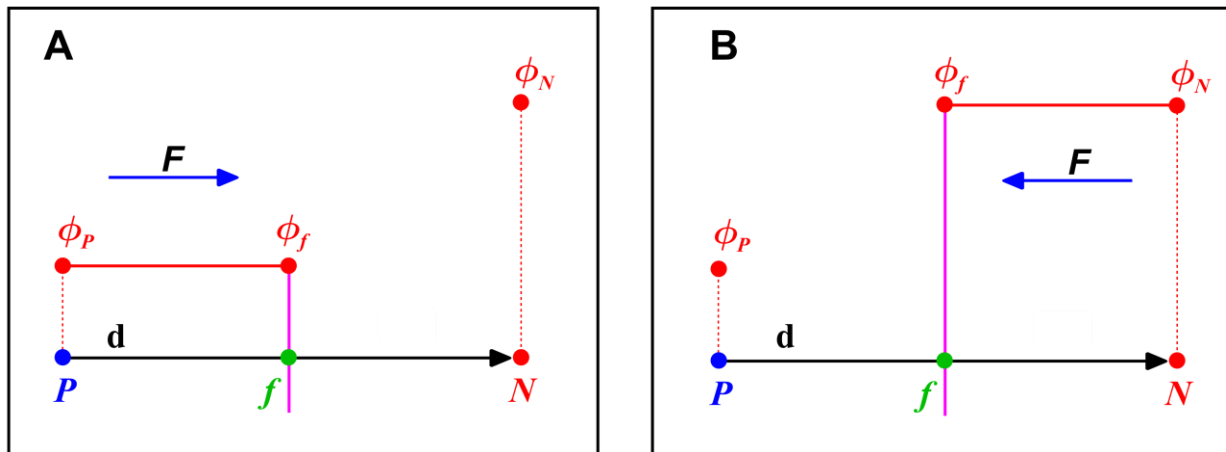
$$\phi_f = f_x \phi_P + (1 - f_x) \phi_N$$
$$f_x = \frac{fN}{PN} = \frac{|\mathbf{x}_f - \mathbf{x}_N|}{|\mathbf{d}|}$$

- 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,



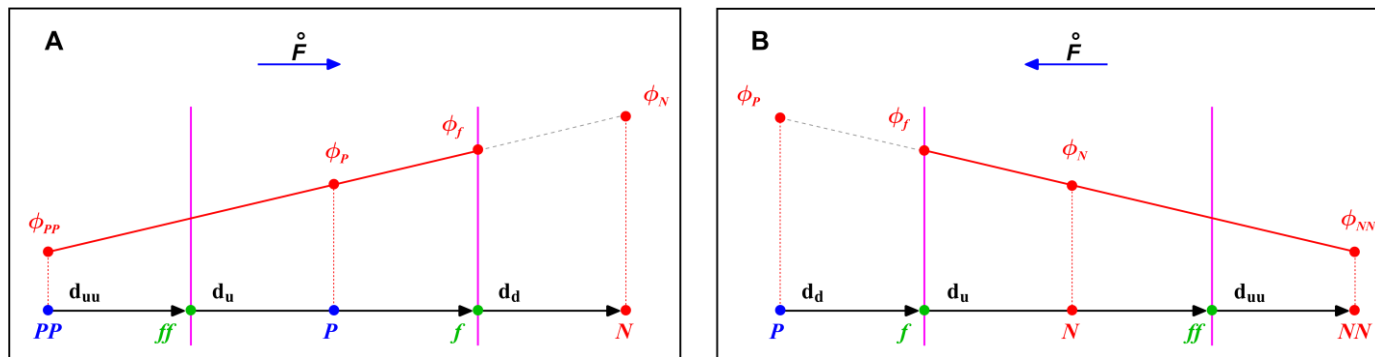
$$\phi_f = \begin{cases} \phi_f = \phi_P & \text{for } \dot{F} \geq 0, \\ \phi_f = \phi_N & \text{for } \dot{F} < 0. \end{cases}$$

- 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.,

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

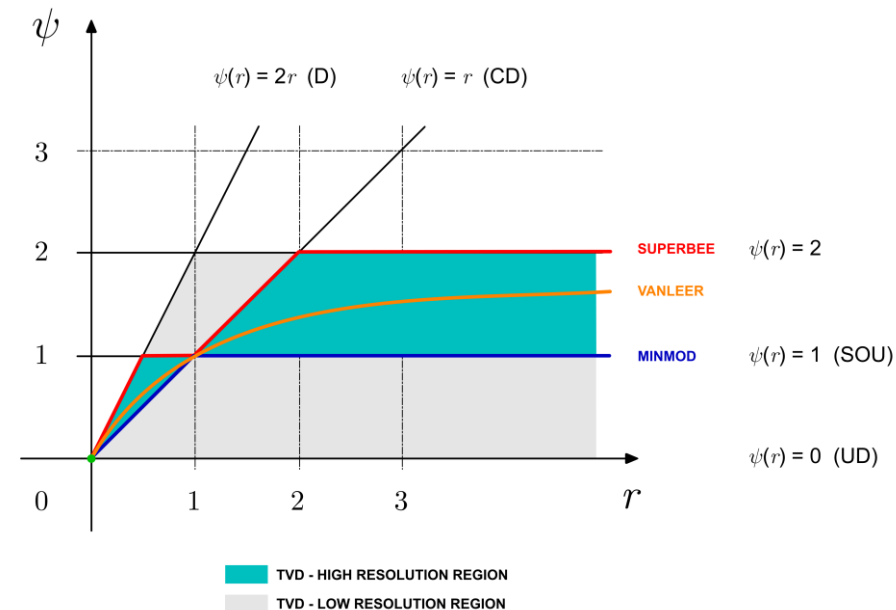
- By adding a well-designed limiter function $\psi(r)$, we get a high resolution (second order accurate) and bounded scheme (HR). This is a TVD scheme.

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – TVD schemes

- A TVD scheme, is a scheme that does not create new local undershoots and/or overshoots in the solution or amplify existing extremes.
- In CFD we want stable, non-oscillatory, bounded, high order schemes.
- The Sweby diagram (Sweby, 1984), gives the necessary and sufficient conditions for a scheme to be TVD.
- In the figure, the shaded area represents the admissible TVD region. However, not all limiter functions are second order.
- 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 (low resolution scheme), when $r < 0$ (sharp gradient, opposite slopes or zero gradient). However, this is justified when it serves to suppress oscillations.
- 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.

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

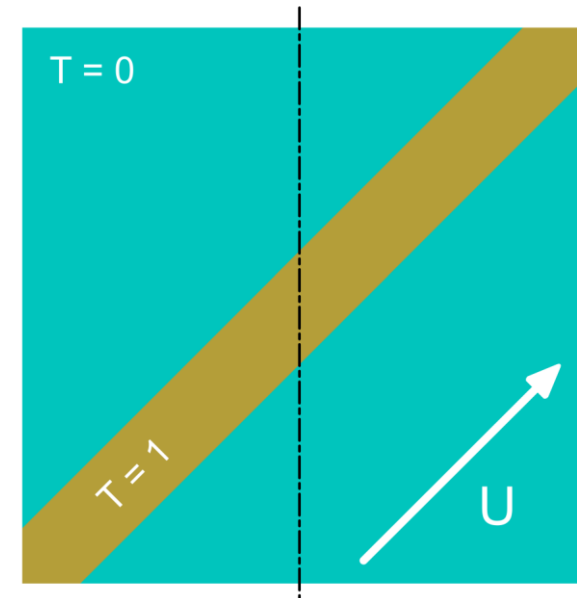
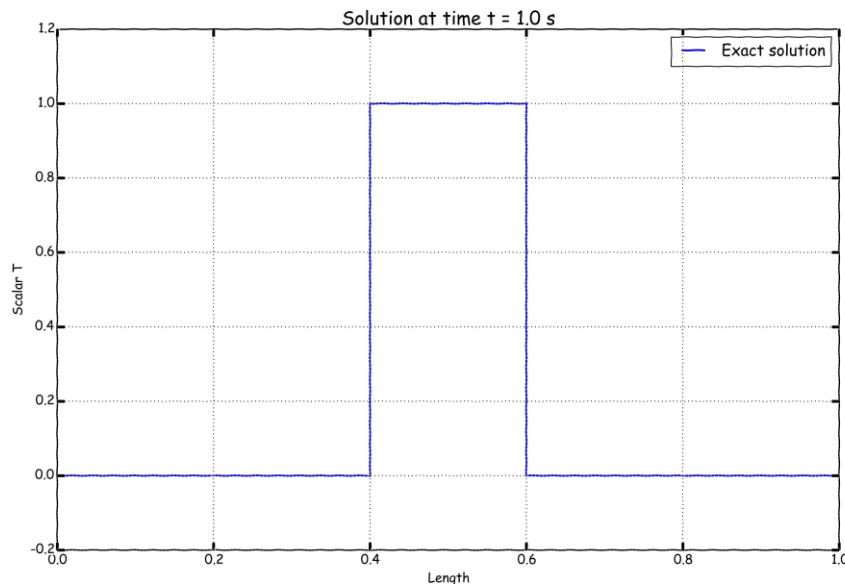


UD = upwind
SOU = second order upwind
CD = central differencing
D = downwind

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – TVD schemes

- Let us see how the upwind, linear upwind, linear, and Minmod TVD schemes behave in a numerical schemes killer test case:
 - The oblique double step profile in a uniform vector field (pure convection).
- Even if this problem seems to be easy, from the numerical point of view is difficult to resolve due to the strong discontinuities.
- This problem has an exact solution.

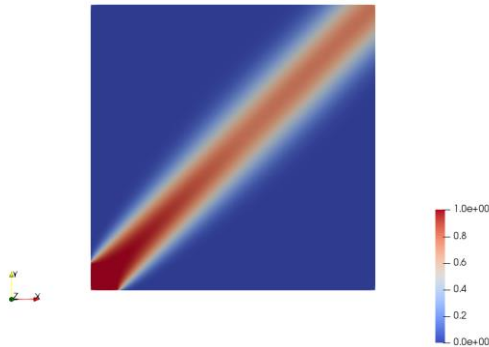


Finite Volume Method: A Crash introduction

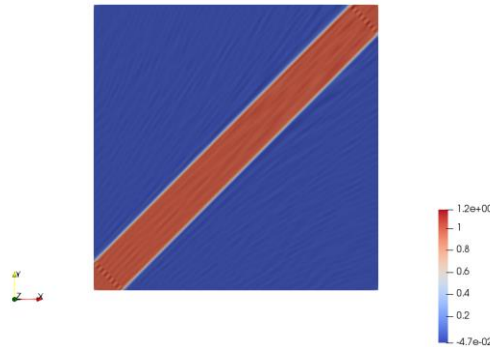
Interpolation of the convective fluxes – TVD schemes

- Qualitative comparison of the upwind, linear upwind, linear, and Minmod TVD schemes

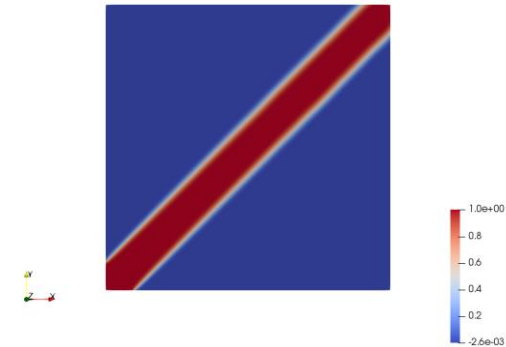
Upwind – 1st order
Very bounded but too Diffusive



Linear – 2nd order
Very accurate but too oscillatory



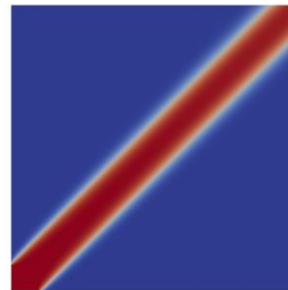
Linear Upwind – 2nd order
Bounded and accurate



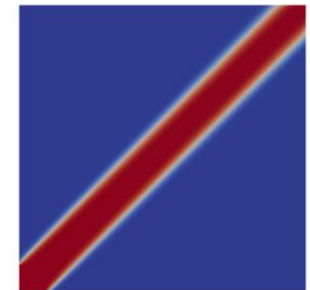
SuperBee – TVD high resolution
Compressive



Minmod – TVD high resolution
Diffusive



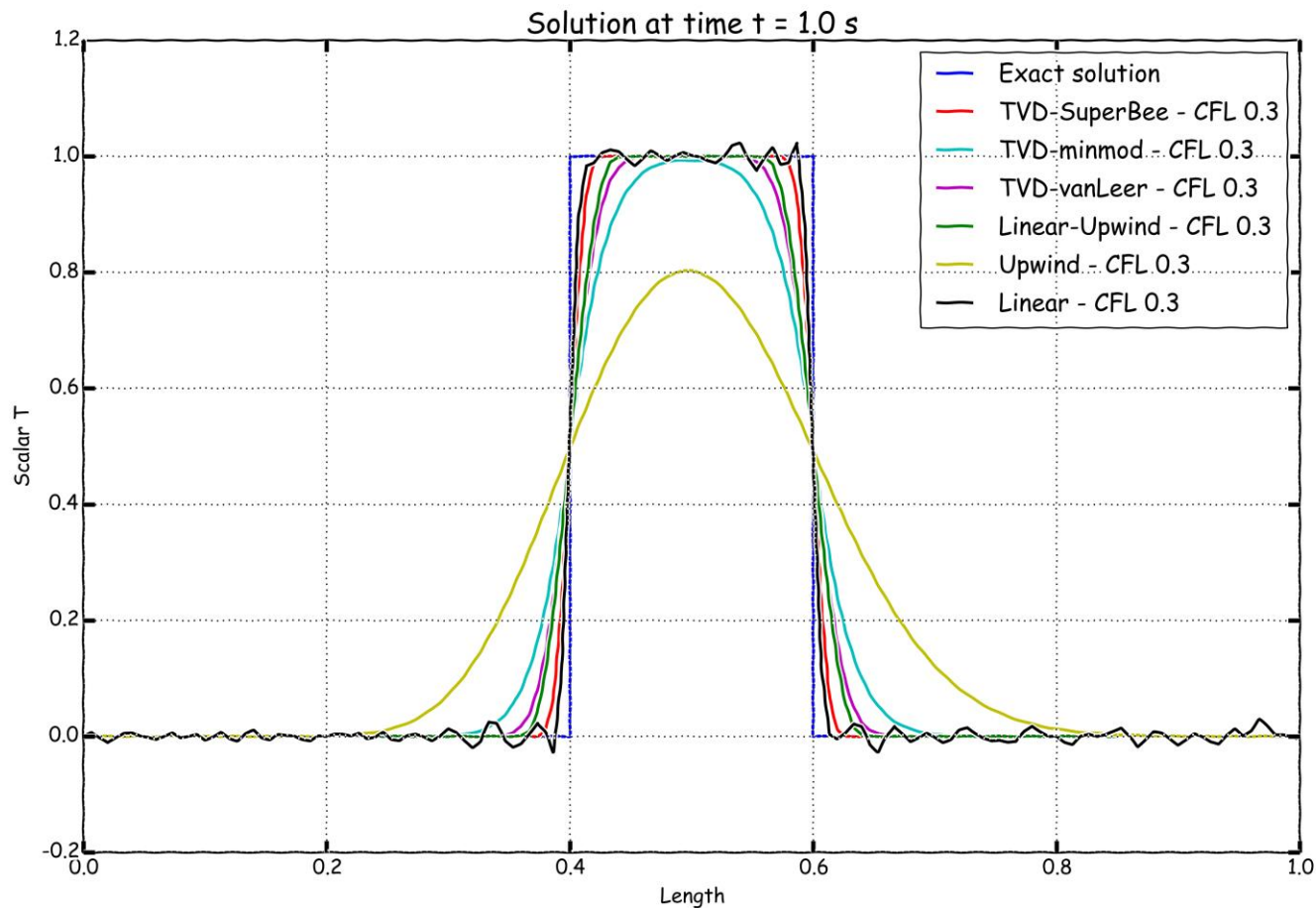
vanLeer – TVD high resolution
Smooth



Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – TVD schemes

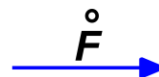
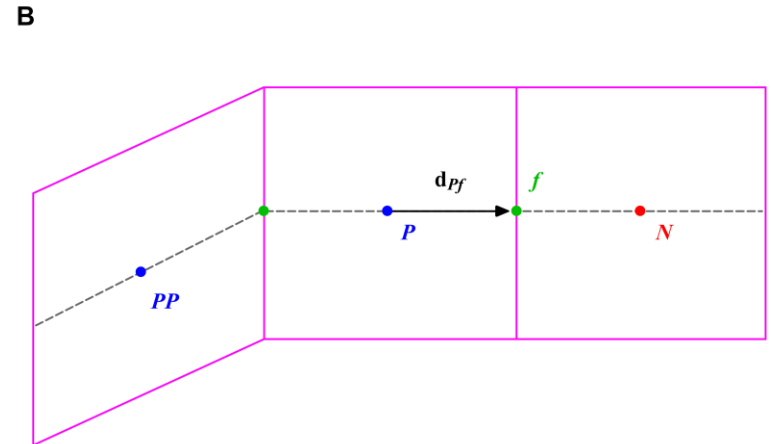
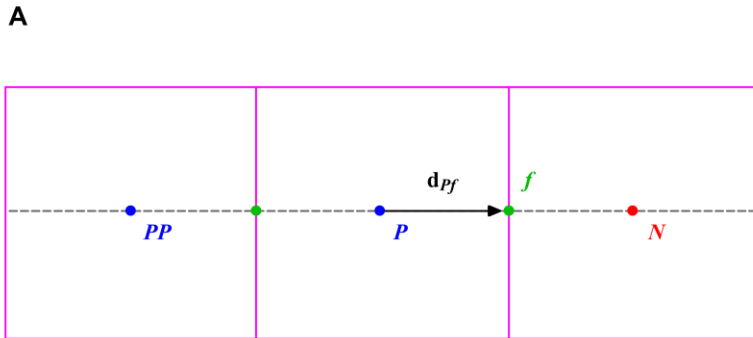
- Quantitative comparison of the upwind, linear upwind, linear, and Minmod TVD schemes



Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – Unstructured meshes

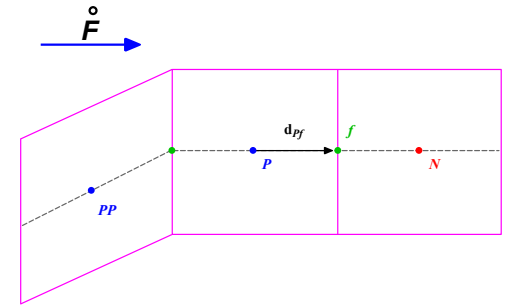
- In the previous explanation, we assumed a line structure (figure A). That is, the cell centers PP , P , and N are all aligned.
- In unstructured meshes (which are often used in industrial cases), most of the times the cell center PP is not aligned with the vector connecting cells P and N (figure B). Therefore, extending the previous formulations to these meshes is not very straightforward.
- Higher-order schemes for unstructured meshes are an area of active research, and new ideas continue to emerge.



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

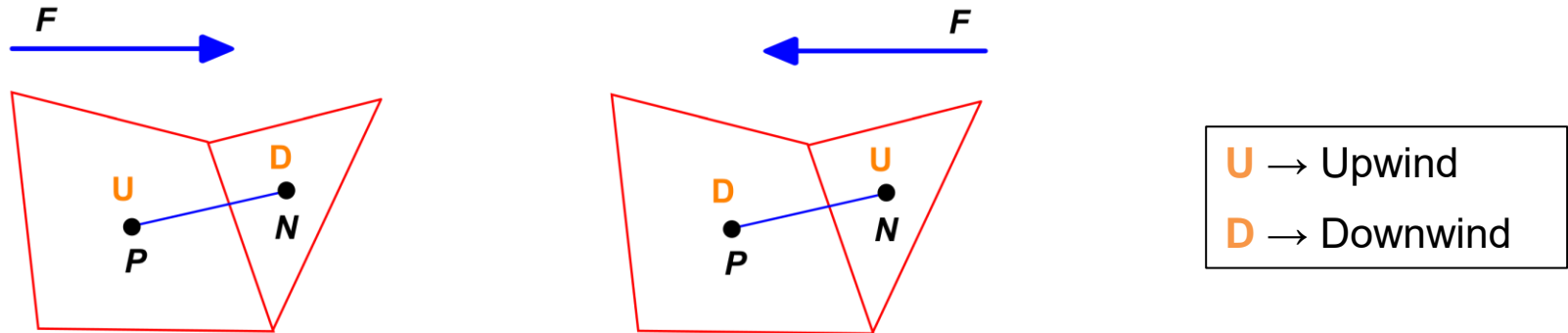
- Notice that in this new formulation the cell PP does not appear anymore.
- The problem now turns in the accurate evaluation of the gradients at the cell and face centers.
- For example, the gradients at the cell centers can be computed using the Gauss method, and then interpolated to the face centers.
- At this point, we are only missing the reconstruction of the cell center gradients at the face centers, this is explained latter.

Finite Volume Method: A Crash introduction

Interpolation of the convective fluxes – Unstructured meshes

- In unstructured meshes, as often the value of the node PP (of NN) is not available or straightforward to compute, the ratio of successive gradients r can be computed as follows [1],

$$r = \left[\frac{(2\nabla\phi_P \cdot \mathbf{d}_{PN})}{\phi_D - \phi_U} - 1 \right]$$



- As you can see, the value of r depends on the flow direction.
- There are many ways to compute r . This is an area of active research

Reference:

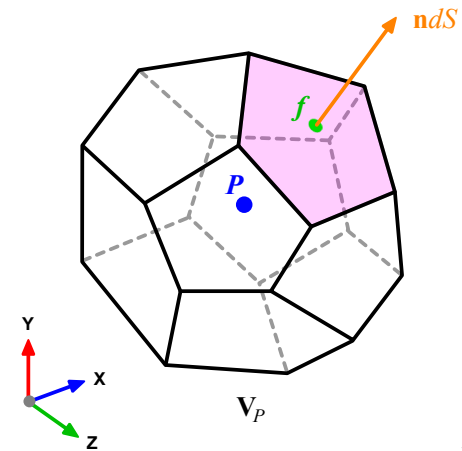
[1] Darwish, M. S., Moukalled, F., "TVD schemes for unstructured grids"

Finite Volume Method: A Crash introduction

Gradients computation at cell centers

- There are many methods for the computation of the cell centered gradients, e.g., least squares, Gauss cell-based, Gauss node-based, and so on.
- Using the Gauss cell-based method, the cell centered gradients can be computed as follows,

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



$$\mathbf{n}dS = d\mathbf{S}$$

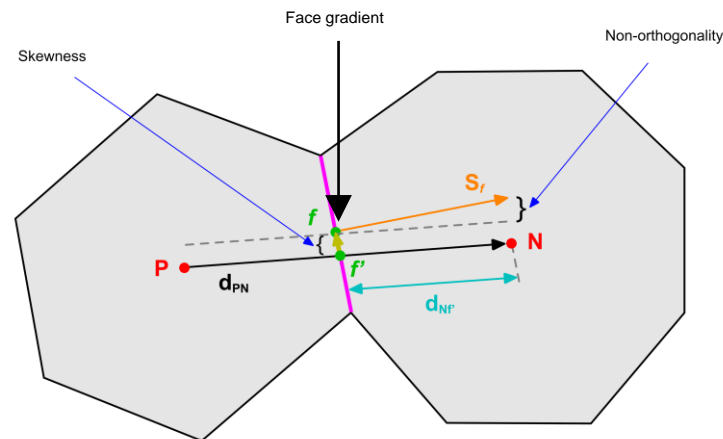
- This approximation is second order accurate given that the mesh quality is acceptable, and the volume of the cell is finite.
- In general, the least squares method tends to be more accurate.

Finite Volume Method: A Crash introduction

Gradients reconstruction at face centers

- Face gradients $\nabla\phi_f$ arise from the discretization process of the convective and diffusive terms.
- One way to reconstruct the face gradient $\nabla\phi_f$, is by using weighted interpolation of the cell centered quantities $\nabla\phi_P$ and $\nabla\phi_N$.
- Mesh non-orthogonality and skewness introduce errors when approximating the face gradients, so corrections need to be added.
- This is an iterative process, where we compute successively better approximations to the gradients starting from an initial approximation.

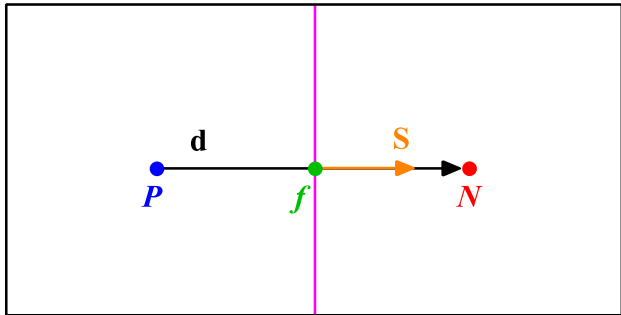
$$\nabla\phi_f = f_x \nabla\phi_P + (1 - f_x) \nabla\phi_N \quad \text{where} \quad f_x = fN/PN$$



Finite Volume Method: A Crash introduction

Interpolation of diffusive fluxes in an orthogonal mesh

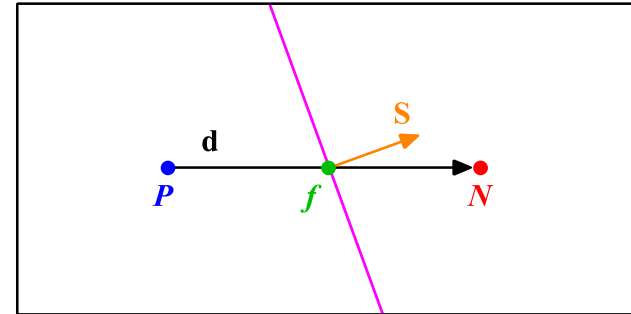
- By looking the figure below, the face values appearing in the diffusive flux in an orthogonal mesh can be computed as follows,



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

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

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



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

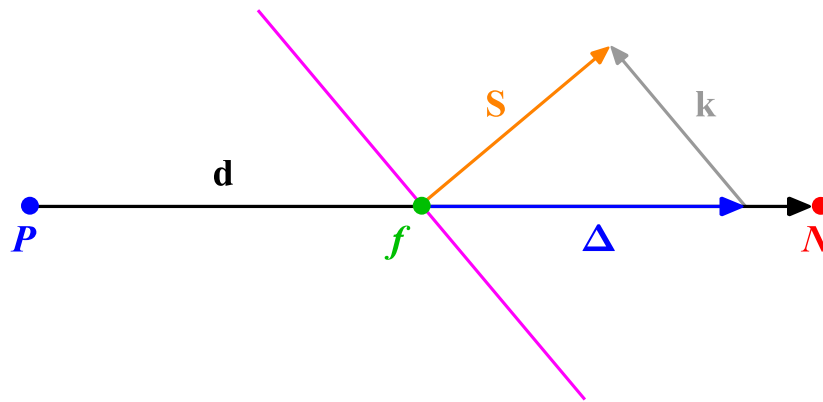
- This type of approximation is second order accurate but involves a larger truncation error. It also uses a larger numerical stencil, which make it less stable.

- Remember, the non-orthogonal angle is the angle between the vector **S** and the vector **d**

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

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

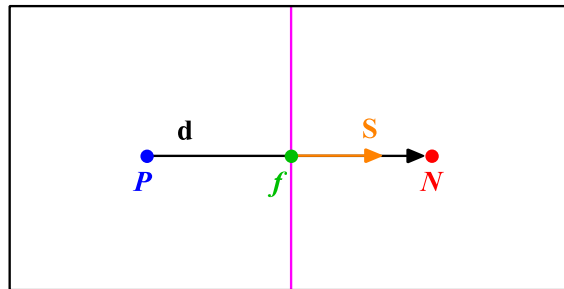
$$\mathbf{S} = \Delta_{\perp} + \mathbf{k}.$$

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

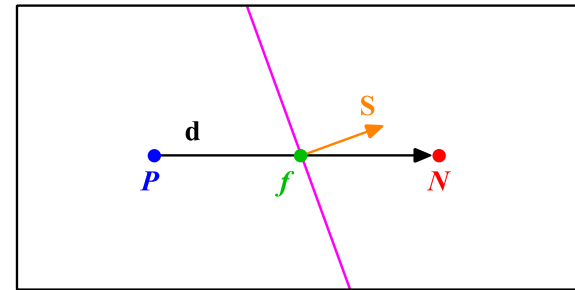
Finite Volume Method: A Crash introduction

Mesh induced errors

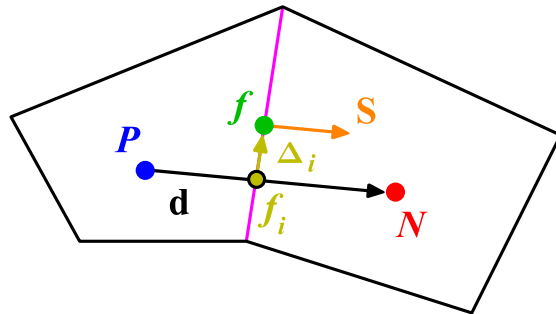
- In order to maintain second order accuracy, and to avoid unboundedness, we need to correct non-orthogonality and skewness errors.
- The ideal case is to have an orthogonal and non skew mesh, but this is the exception rather than the rule.



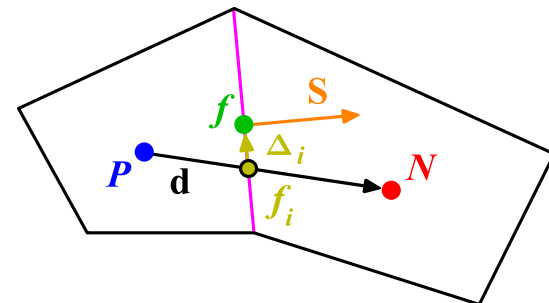
Orthogonal and non skew mesh



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,

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

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.

- 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,

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

- 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, dynamic meshes, multiphase models, combustion models, etc.
 - This information is contained in the directories **constant** and/or **system**

Finite Volume Method: A Crash introduction

So, what does OpenFOAM® do?

- Therefore, we need to give to OpenFOAM® the following information:
 - 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, maximum CFL number, solution saving frequency, and so on).
 - This information is set in the `system/controlDict` dictionary.
 - Additionally, we may set sampling and monitors for post-processing (**functionObjects**).
 - This information is set in the `system/controlDict` dictionary or in the specific sampling dictionaries located 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 term discretization.
- The keyword **laplacianSchemes** refers to the Laplacian term 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

Time discretization schemes

- The **Crank-Nicolson** method as it is implemented in OpenFOAM®, uses a blending factor.

```
ddtSchemes
{
    default    CrankNicolson  $\psi$ ;
}
```

- Setting ψ to 0 is equivalent to running a pure **Euler** scheme (robust but first order accurate).
- By setting the blending factor equal to 1 you use a pure **Crank-Nicolson** (accurate but oscillatory, formally second order accurate).
- If you set the blending factor to 0.5, you get something in between first order accuracy and second order accuracy, or in other words, you get the best of both worlds.
- A blending factor of 0.7-0.9 is safe to use for most applications (stable and accurate).

Finite Volume Method: A Crash introduction

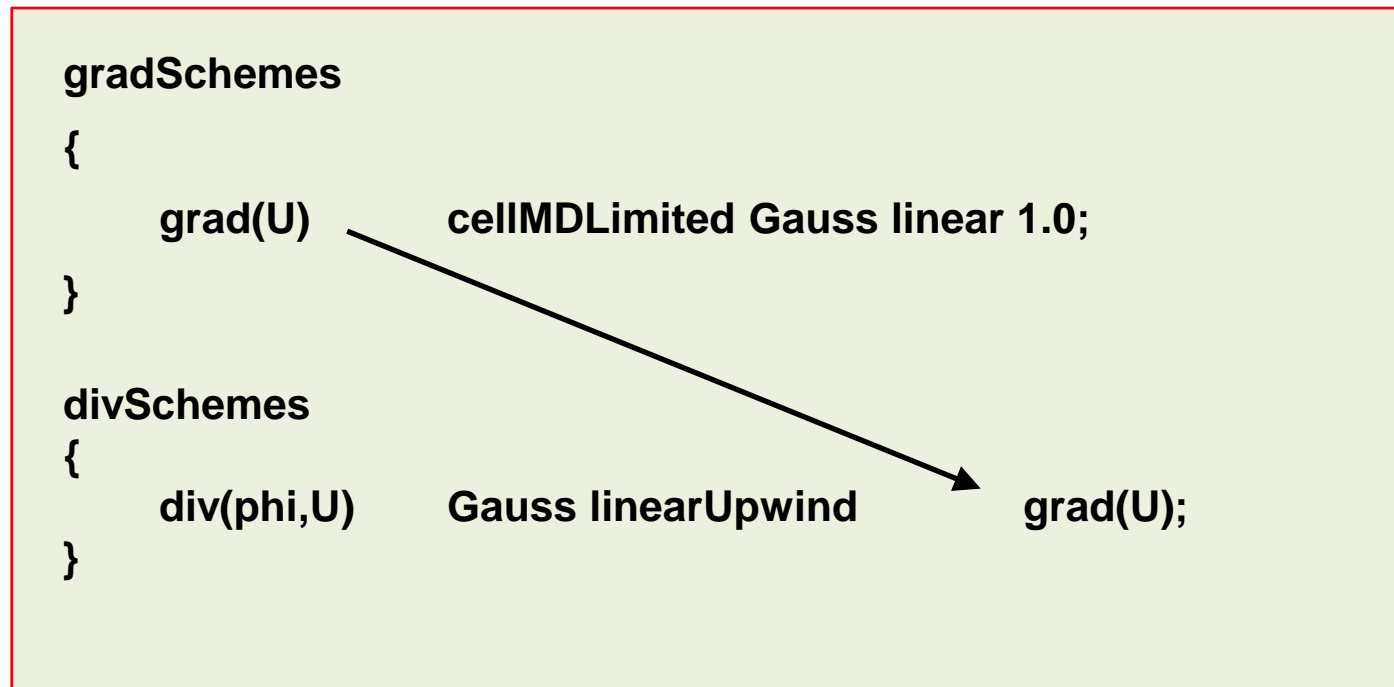
Convective terms discretization schemes

- 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.
 - A good TVD scheme (**vanLeer** or **Minmod**): 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).
 - **LUST**: blended 75% **linear** and 25% **linearUpwind** scheme
- 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 every transported quantity (e.g. **k**, **epsilon**, **omega**, **T**)

Finite Volume Method: A Crash introduction

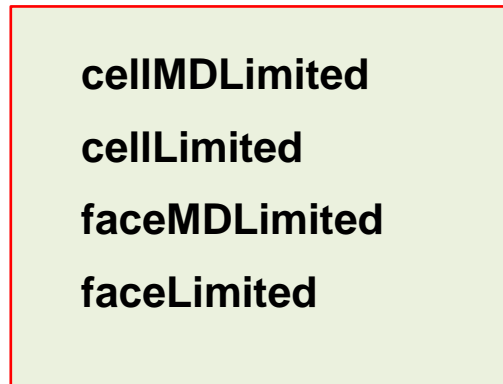
Gradient terms discretization schemes

- There are many gradient discretization schemes available in OpenFOAM®.
- You will find the source code in the following directory:
 - `$WM_PROJECT_DIR/src/finiteVolume/finiteVolume/gradSchemes`
- These are the gradient discretization schemes that you will use most of the times:
 - **Gauss linear** (cell-based method)
 - **Gauss pointLinear** (node-based method; more accurate than the cell-based method)
 - **leastSquares**
- To avoid overshoots or undershoots when computing the gradients, you can use gradient limiters.
- Gradient limiters increase the stability of the method but add diffusion due to clipping.
- You will find the source code in the following directory:
 - `$WM_PROJECT_DIR/src/finiteVolume/finiteVolume/gradSchemes/limitedGradSchemes`
- These are the most important 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®:



Less diffusive



More diffusive

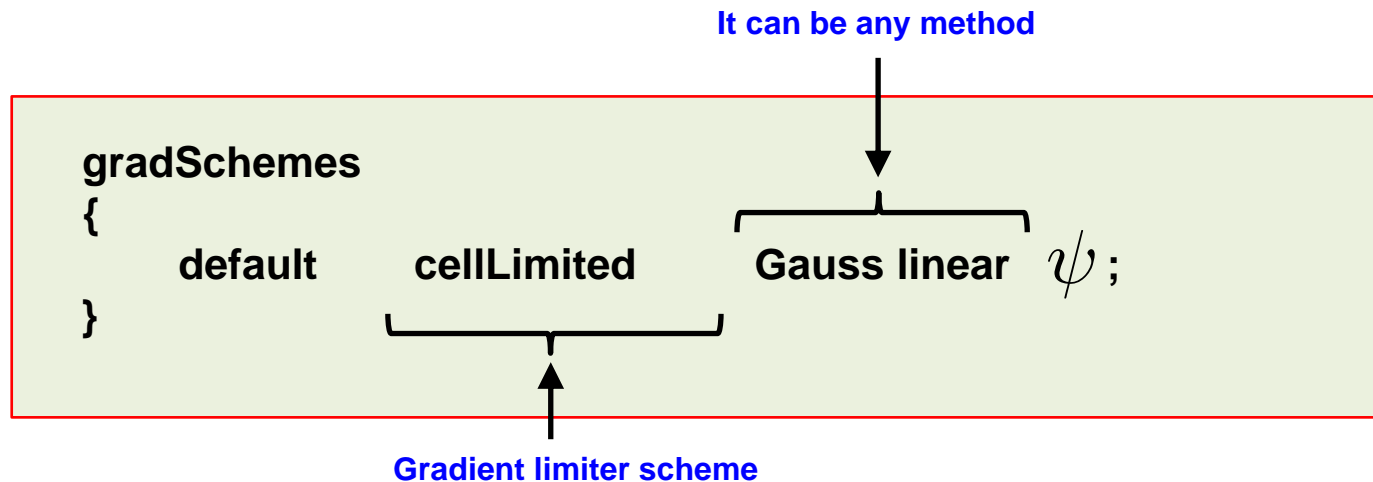
Note: for smooth field variation, cell limiting may provide less numerical dissipation on meshes with skewed cells

- 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 set to be very aggressive (kind of saying that it 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 limited to hexahedral meshes with very low non-orthogonality. Second order accurate, without non-orthogonal corrections. Stable but more diffusive than the limited and corrected methods.

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

Can be computed using the over-relaxed approach

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

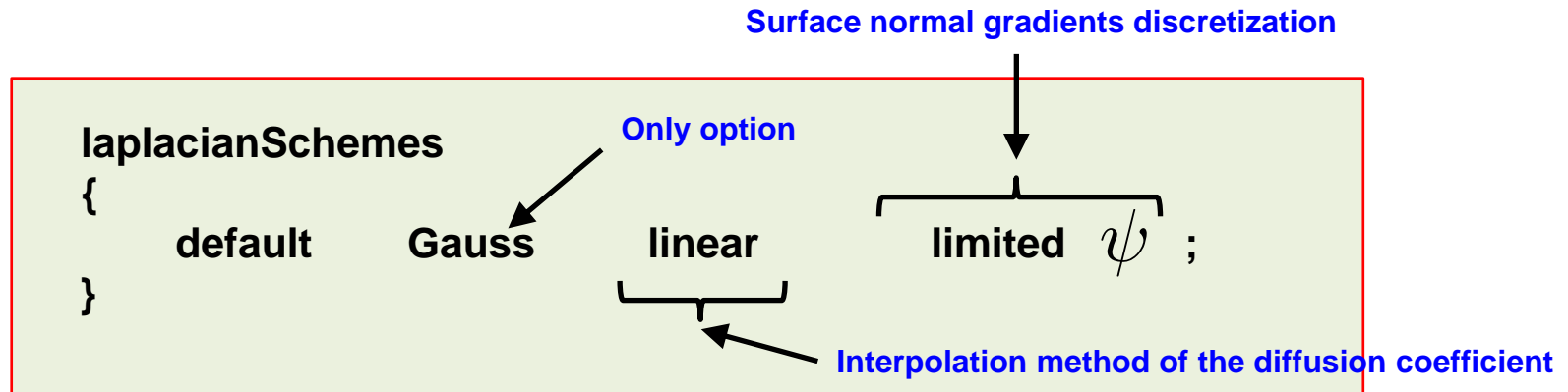
$$\longrightarrow S \cdot (\nabla \phi)_f = \underbrace{|\Delta_\perp| \frac{\phi_N - \phi_P}{|d|}}_{\text{orthogonal contribution}} + \underbrace{\mathbf{k} \cdot (\nabla \phi)_f}_{\text{non-orthogonal contribution}}.$$

Can be computed using the over-relaxed approach

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.
- For meshes with non-orthogonality less than 70, you can set the blending factor to 1.
- For meshes with non-orthogonality between 70 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 want to use that mesh, you can set the blending factor to 0.333-0.5, 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

- Just to make it clear, the blending factor ψ is used to avoid the non-orthogonal contribution exceeding the orthogonal part.
- That is, non-orthogonal contribution \leq orthogonal contribution.

The blending factor works as a limiter acting on this term (non-orthogonal contribution)

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

Implicit part Explicit part

- In meshes with large non-orthogonality, the explicit term can lead to unboundedness and eventually divergence.
- This limiting is local, similar to the treatment done for the connective terms when using slope limiters and TVD schemes.
- The explicit contribution is added to the RHS of the linear system (source term), so if this term becomes too large it will lead to convergence problems.
- It becomes harder to guarantee diagonal dominance of the matrix of coefficient.

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;    //or 0.3;
}
gradSchemes
{
    default          cellLimited Gauss linear 0.5;
    grad(U)          cellLimited Gauss linear 1;
}
divSchemes
{
    default          none;
    div(phi,U)       Gauss linearUpwind 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 0.5;
}
interpolationSchemes
{
    default          linear;
}
snGradSchemes
{
    default          limited 0.5;
}
```

- **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.
- To keep temporal diffusion to a minimum, use a CFL number less than 2, and preferably below 1.
- If during the simulation the turbulence quantities become unbounded, you can safely change the discretization scheme to upwind. After all, turbulence is diffusion.
- For gradient discretization the **leastSquares** method is more accurate. But we have found that it is a little bit oscillatory in tetrahedral meshes.

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 limitedLinear 1;
    div(phi,k) Gauss limitedLinear 1;
    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 a CFL number less than 2, and preferably below 1.
- Instead of the **linear** method for **div(phi,U)**, you can use **limitedLinear** or **LUST**,

div(phi,U) Gauss limitedLinear 1;

div(phi,U) Gauss LUST default;

Finite Volume Method: A Crash introduction

A very stable but too diffusive numerics

```
ddtSchemes
{
    default          Euler;
}
gradSchemes
{
    default          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.**
- You can use this method for troubleshooting. If the solution diverges, you better check boundary conditions, physical properties, and so on.

Roadmap

- ~~1. Finite Volume Method: A Crash Introduction~~
- 2. On the CFL number**
3. Linear solvers in OpenFOAM®
4. Pressure-Velocity coupling in OpenFOAM®
5. Unsteady and steady simulations
6. Understanding residuals
7. Boundary and initial conditions
8. Numerical playground

On the CFL number

- First of all, what is the CFL or Courant number?
- In one dimension, the CFL number is defined as,

$$CFL = \frac{u \Delta t}{\Delta x}$$

- The CFL number is a measure of how much information (u) traverses a computational grid cell (Δx) in a given time-step (Δt).
- The CFL number is not a magical number.
- The CFL number is a necessary condition to guarantee the stability of the numerical scheme.
- But not all numerical schemes have the same stability requirements.
- By doing a linear stability study, we can find the stability requirements of each numerical scheme (but this is out of the scope of this lecture).

On the CFL number

- Let us now talk about the **CFL number condition**. The **CFL number condition** is the maximum allowable CFL number a solver can use.
- For the **N** dimensional case, the CFL number condition becomes,

$$CFL = \Delta t \sum_{i=1}^n \frac{u_i}{\Delta x_i} \leq CFL_{max}$$

- CFD solvers can be explicit and implicit.
- Explicit and implicit solvers have different stability requirements.
- Implicit numerical methods are **unconditionally stable**.
- In other words, they are not constrained to the **CFL number condition**.
- However, the fact that you are using a numerical method that is unconditionally stable, **does not mean that you can choose a time step of any size**.
- The time-step must be chosen in such a way that it resolves the time-dependent features, and it maintains the solver stability.
- When we use implicit solvers, we need to assemble a large system of equations.
- The memory requirements of implicit methods are much higher than those of explicit methods.
- In OpenFOAM®, most of the solvers are implicit.
- In our personal experience, we have been able to go up to a $CFL = 5.0$ while maintaining the accuracy and without increasing too much the computational cost.
- But as we are often interested in the unsteadiness of the solution, we usually use a CFL number in the order of 1.0

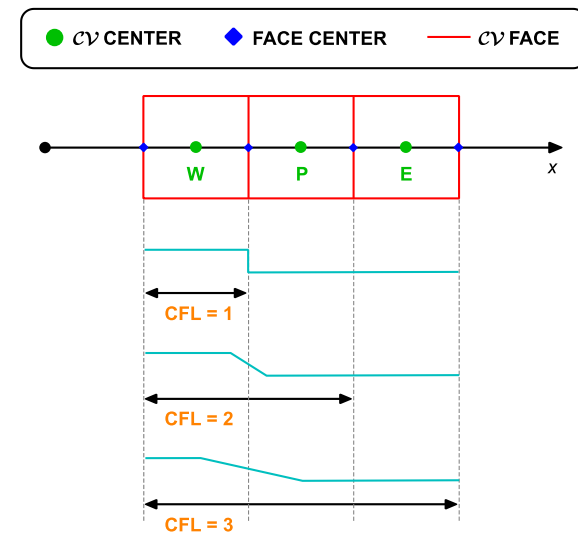
On the CFL number

The CFL number for dummies

- I like to see the CFL number as follows,

$$CFL = \frac{u \Delta t}{\Delta x} = \frac{u}{\Delta x / \Delta t} = \frac{\text{speed of the PDE}}{\text{speed of the mesh}}$$

- It is an indication of the amount of information that propagates through one cell (or many cells), in one time-step.
- By the way, and this is extremely important, the CFL condition is a necessary condition for stability (and hence convergence).
- But it is not always sufficient to guarantee stability.
- Other properties of the discretization schemes that you should observe are: conservationness, boundedness, transportiveness, and 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 time step size is set in the *controlDict* dictionary.
- 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
```

Roadmap

- ~~1. Finite Volume Method: A Crash Introduction~~
- ~~2. On the CFL number~~
- 3. Linear solvers in OpenFOAM®**
- ~~4. Pressure-Velocity coupling in OpenFOAM®~~
- ~~5. Unsteady and steady simulations~~
- ~~6. Understanding residuals~~
- ~~7. Boundary and initial conditions~~
- ~~8. Numerical playground~~

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

■ = Diagonal contribution
□ = Off-diagonal contribution

$$\begin{bmatrix} a_1 & \square & & \square & & \\ \square & \blacksquare & \square & & \square & \\ & \ddots & \ddots & \ddots & & \ddots \\ \ddots & & \ddots & \ddots & \ddots & \ddots \\ & \square & & \square & a_P & \square & \square \\ & & \ddots & & \ddots & \ddots & \ddots \\ & & & \square & \square & \blacksquare & \square \\ & & & & \square & \square & a_N \end{bmatrix} \times \begin{bmatrix} \phi_1 \\ \vdots \\ \phi_P \\ \vdots \\ \phi_N \end{bmatrix} = \begin{bmatrix} b_1 \\ \vdots \\ b_P \\ \vdots \\ b_N \end{bmatrix}$$

$$\mathbf{A} \phi = \mathbf{b}$$

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

Linear solvers in OpenFOAM®

Linear solvers – *fvSolution* dictionary

```
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 tell OpenFOAM® how to crunch numbers.
- The **solvers** sub-dictionary specifies each linear solver that is used for each equation being solved.
- The linear solvers distinguish between symmetric matrices and asymmetric matrices.
- If you forget to define a linear-solver or use the wrong one, OpenFOAM® will let you know.
- 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 – *fvSolution* dictionary

```
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 structure of the `fvSolution` dictionary. It shows three main sections: `solvers`, `PISO`, and `pFinal`. The `solvers` section contains three sub-dictionaries: `p`, `pFinal`, and `U`. Each sub-dictionary has entries for `solver`, `preconditioner`, `tolerance`, and `relTol`. Arrows point from the text in the list to the corresponding entries in the dictionary: `PCG` and `DIC` for the `p` solver, `PBiCGStab` and `DILU` for the `U` solver, and `1e-06` for the `p` solver's `tolerance`.

- 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 (disabled).
- 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 iterative 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 relatively 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, and it might be problem dependent.

Linear solvers in OpenFOAM®

Linear solvers – *fvSolution* dictionary

```
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 – *fvSolution* dictionary

```
solvers
{
    p
    {
        solver          PCG;
        preconditioner  DIC;
        tolerance       1e-06;
        relTol          0.01;
    }
    pFinal
    {
        $p;
        relTol 0;
    }
    U
    {
        solver          PBiCGStab;
        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 (segregated) available in OpenFOAM®:

- | | | | |
|--------------------|------------------------|-------------------------|------------------------|
| • GAMG | → Multigrid solver | • PCG | → Newton-Krylov solver |
| • PBiCG | → Newton-Krylov solver | • smoothSolver | → Smooth solver |
| • PBiCGStab | → Newton-Krylov solver | • diagonalSolver | |

- You will find the source code of the linear solvers in the following directory:

- `$WM_PROJECT_DIR/src/OpenFOAM/matrices/lduMatrix/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 – General remarks

- 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 100 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 500), using the **PCG** might be a better choice.

Linear solvers in OpenFOAM®

Linear solvers – General remarks

- 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 – General remarks

- 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 after a while, 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 tolerances

- 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 (symmetric matrix), you can start the simulation with a **tolerance** equal to **1e-6** and **relTol** equal to **0.01**.
- And after a while, you change these values to **1e-6** and **0.0**, respectively.
- If the linear solver is taking too much time, 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 tolerances

- For the velocity field (**U**) and the transported quantities (asymmetric matrices), you can use the following criterion.
- Solving for these variables is relatively inexpensive, so you can start right away with a tight tolerance.
- As a side note, the relative tolerance (**relTol**) is the difference between the initial residuals and the current final residuals.

Loose tolerance

```
U
{
    solver          PBiCGStab;
    preconditioner  DILU;
    tolerance       1e-8;
    relTol          0.001;
}
```

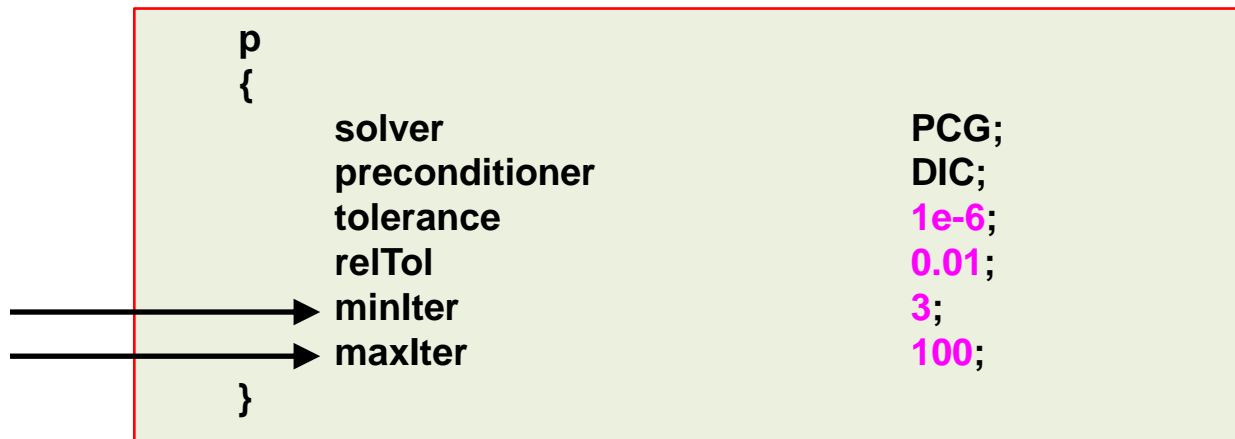
Tight tolerance

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

Linear solvers in OpenFOAM®

Linear solvers tolerances

- It is also a good idea to set the minimum number of iterations (**minIter**), we recommend using a value of 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 time-steps or outer-iterations.
- Setting the maximum number of iterations is especially 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.



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

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

Linear solvers in OpenFOAM®

Linear solvers tolerances

- When you use the **PISO** or **PIMPLE** method with the **momentumPredictor** option (which is enabled by default), 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 tight the solver tolerance.

Loose tolerance for p

```
p
{
    solver          PCG;
    preconditioner   DIC;
    tolerance        1e-4;
    relTol           0.01;
}
```

Tight tolerance for pFinal

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


Linear solvers in OpenFOAM®

Linear solvers tolerances

- When you use the **PISO** or **PIMPLE** method with the **momentumPredictor** option (which is enabled by default), 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

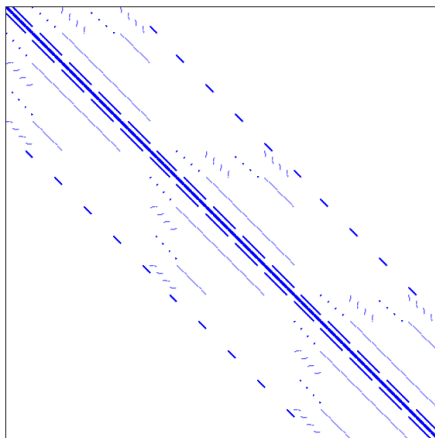
1
2
↑

nCorrectors

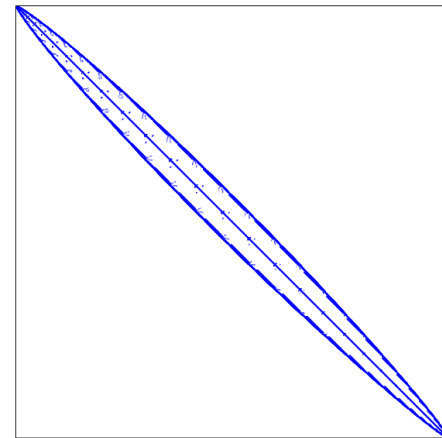
Linear solvers in OpenFOAM®

Linear solvers – Matrix reordering

- 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 first iterations.
- 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

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 **GAMG** 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 proposed 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.
- For steady simulations using the **SIMPLE** method, you can 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®

Linear solvers benchmarking of a model case

Case	Linear solver for P	Preconditioner or smoother	MR	Time	QOI
IC1	PCG	FDIC	NO	278	2.8265539
IC2	smoothSolver	symGaussSeidel	NO	2070	2.8271198
IC3	ICCG	GAMG	NO	255	2.8265538
IC4	GAMG	GaussSeidel	NO	1471	2.8265538
IC5	PCG	GAMG-GaussSeidel	NO	302	2.8265538
IC6	GAMG	GaussSeidel	YES	438	2.8265539
IC7	PCG	FDIC	YES	213	2.8265535
IC8	PCG	GAMG-GaussSeidel	YES	283	2.8265538
IC9	ICCG	GAMG	YES	261	2.8265538
IC10	PCG	DIC	NO	244	2.8265539

Solver used = `icoFoam` – Incompressible case

MR = matrix reordering (`renumberMesh`)

QOI = quantity of interest. In this case the maximum velocity at the outlet (m/s)

TIME = clock time (seconds)

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, conduct the following numerical experiments:
 - 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**)

Roadmap

- ~~1. Finite Volume Method: A Crash Introduction~~
- ~~2. On the CFL number~~
- ~~3. Linear solvers in OpenFOAM®~~
- 4. Pressure-Velocity coupling in OpenFOAM®**
- ~~5. Unsteady and steady simulations~~
- ~~6. Understanding residuals~~
- ~~7. Boundary and initial conditions~~
- ~~8. Numerical playground~~

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.

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

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.

Pressure-Velocity coupling in OpenFOAM®

- 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.
- In OpenFOAM®, you will find segregated pressure-based solvers.
- The segregated pressure-based solvers in OpenFOAM®, solve a modified pressure equation (pressure-Poisson equation).
- The following methods are available:
 - **SIMPLE** (Semi-Implicit Method for Pressure-Linked Equations)
 - **SIMPLEC** (SIMPLE Corrected/Consistent)
 - **PISO** (Pressure Implicit with Splitting Operators)
- You will find the solvers in the following directory:
 - `$WM_PROJECT_DIR/applications/solvers`
- Additionally, you will find something called **PIMPLE**, which is a hybrid between **SIMPLE** and **PISO** (known as iterative **PISO** outside OpenFOAM® jargon).
 - This formulation can give you more accuracy and stability when using very large time-steps or in pseudo-transient simulations.

Pressure-Velocity coupling in OpenFOAM®

- In OpenFOAM®, the **PISO** (PISO non-iterative) and **PIMPLE** (PISO iterative) 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 in pseudo transient mode.
- The pseudo transient **PIMPLE** method is more stable than the **SIMPLE** method, but it has a higher computational cost.
- Also, the pseudo transient **PIMPLE** method tends to be faster than the fully transient **PIMPLE** when reaching steady states.
- 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”, Numerical 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 outside OpenFOAM® ecosystem (we are referring to the semantics).
 - It is equivalent to **PISO** with outer iterations (iterative time-advancement of the solution).
 - Useful reference (besides **PISO** reference):
 - I. E. Barton, “Comparison of SIMPLE and PISO-type algorithms for transient flows, Int. J. Numerical methods in fluids, 26,459-483 (1998).
 - P. Oliveira and R. I. Issa, “An improved piso algorithm for the computation of buoyancy-driven flows”, Numerical Heat Transfer, 40, 473-493 (2001).

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 one 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 non-orthogonal corrections. However, it is strongly recommended to do at least 1 non-orthogonal correction (this helps stabilizing the solution).
- For non-orthogonal meshes, it is recommended to do at least 1 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 is reduced.
- The **SIMPLEC** method relaxes the pressure in a consistent manner and additional relaxation of the pressure is not generally necessary (but it is recommended).
- In addition, convergence of the **p-U** system is better and still is reliable with less aggressive relaxation factors of the momentum equation.

SIMPLE

{

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

}

Pressure-Velocity coupling in OpenFOAM®

The SIMPLE sub-dictionary

- These are the typical (or industry standard) 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
{
    fields
    {
        p          1.0;
    }
    equations
    {
        p          1.0;
        U          0.9;
        k          0.9;
        omega      0.9;
    }
}
```

Usually there is no need to under-relax pressure; however, it is advisable.

Pressure-Velocity coupling in OpenFOAM®

The SIMPLE sub-dictionary

SIMPLEC

- If you are planning to use the **SIMPLEC** method, we recommend you use under-relaxation factors that are little bit smaller than the industry standard 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.
- If you are having convergence problems, it is recommended to start the simulation with low values (about 0.3), and then increase the values slowly up to 0.7 or 0.9 (for faster convergence).

```
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®

```
fvVectorMatrix UEqn  
(  
    fvm::ddt(U) + fvm::div(phi, U) - fvm::laplacian(nu, U)  
);
```

Momentum equation without the pressure gradient term

```
solve(UEqn == -fvc::grad(p));
```

Momentum predictor

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

Pressure equation

```
U = HbyA - rAU*fvc::grad(p);
```

Momentum corrector

This is an excerpt of the actual source code of the solver

Under-relax U Eqn

Under-relax p

Update flux $\phi = S_f \cdot \left[\left(\frac{H(U)}{A} \right)_f - \left(\frac{1}{A} \right)_f (\nabla p)_f \right]$

Solve additional transport equations

Start simulation

U Eqn

$\partial U / \partial t + \nabla \cdot (UU) - \nu \nabla^2 U$

U Eqn = $-\nabla p$

$\nabla \cdot \frac{1}{A} \nabla p = \nabla \cdot \left(\frac{H(U)}{A} \right) + f(\nabla p)$

nNonOrthogonalCorrectors

Non-orthogonal corrections loop

$U = \frac{H(U)}{A} - \frac{1}{A} \nabla p$

SIMPLE loop convergence?

No — SIMPLE loop

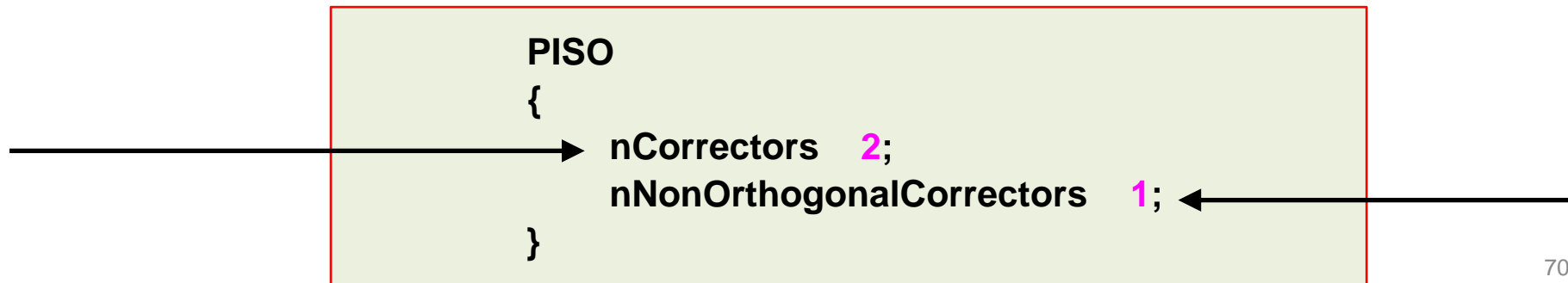
Yes

End simulation

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**).
- For good accuracy and stability (specially in unstructured meshes), it is recommended to use at least 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 non-orthogonal corrections. However, it is strongly recommended to do at least 1 non-orthogonal correction (this helps stabilizing the solution).
- For non-orthogonal meshes, it is recommended to do at least 1 correction.



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 most of the times 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.
- Note that when you enable the option **momentumPredictor**, you will need to define the linear solvers for the variables **.*Final** (we are using regex notation).
- Also, if you want to use URF you will need to apply then to all field variables (including **.*Final**).

PISO

{

momentumPredictor yes;

nCorrectors 2;

nNonOrthogonalCorrectors 1;

}

Pressure-Velocity coupling in OpenFOAM®

The PISO loop in OpenFOAM® (PISO with non-iterative marching – NITA –)

```
fvVectorMatrix UEqn  
(  
    fvm::ddt(U) + fvm::div(phi, U) - fvm::laplacian(nu, U)  
);
```

```
solve(UEqn == -fvc::grad(p));
```

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

```
U = HbyA - rAU*fvc::grad(p);
```

Momentum equation without the pressure gradient term

Under-relax U Eqn

Momentum predictor

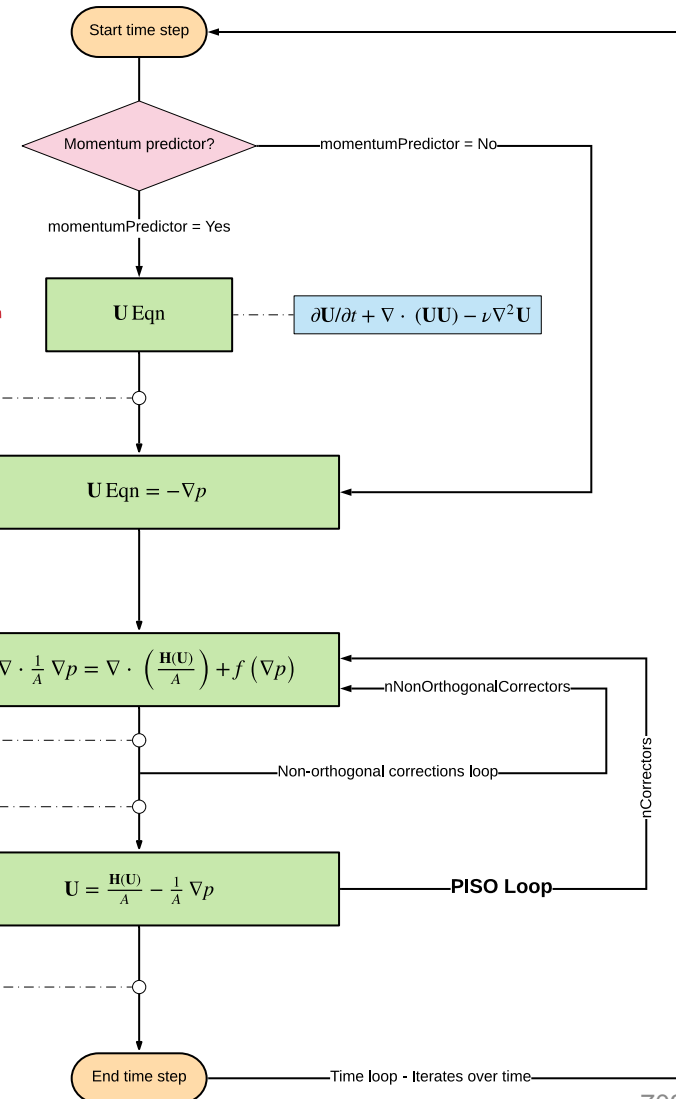
Pressure equation

Under-relax p

Update flux $\phi = S_f \cdot \left[\left(\frac{H(U)}{A} \right)_f - \left(\frac{U}{A} \right)_f (\nabla p)_f \right]$

Momentum corrector

Solve additional transport equations



This is an excerpt of the actual source code of the solver

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 or when dealing with complex physics (combustion, chemical reactions, shock waves, and so on), you can use more outer correctors (**nOuterCorrectors**).
 - Usually between 2 and 5 corrections for computational efficiency.
- Have in mind that increasing the number of **nOuterCorrectors** will highly increase the computational cost.

PIMPLE

{

momentumPredictor yes;

nOuterCorrectors 1;

nCorrectors 2;

nNonOrthogonalCorrectors 1;

}

Pressure-Velocity coupling in OpenFOAM®

The PIMPLE sub-dictionary

- You can use under-relaxation factors (URF) with the **PIMPLE** solvers.
- By using URF, you will gain more stability in time dependent solutions (as they control the amount of change of field variables within the time-step).
- However, if you use too low URF values, your solution might not be time-accurate anymore.
- You can use the same or larger URF values as those for steady simulation.
- Note that when you enable the option **momentumPredictor**, you will need to define the linear solvers for the variables **.*Final** (we are using regex notation).
- You can assign URF to all variables (including **.*Final**), to only the intermediate field variables (**U**, **p**, **k**, and so on), or to only the **.*Final** variables (**UFinal**, **pFinal**, **kFinal**, and so on).
- We recommend to use URF in all variables.

PIMPLE

{

momentumPredictor yes;

nOuterCorrectors 1;

nCorrectors 2;

nNonOrthogonalCorrectors 1;

}

Pressure-Velocity coupling in OpenFOAM®

The PIMPLE loop in OpenFOAM® (PISO with iterative marching – ITA –)

```
fvVectorMatrix UEqn  
(  
    fvm::ddt(U) + fvm::div(phi, U) - fvm::laplacian(nu, U)  
);
```

Momentum equation without the pressure gradient term

Under-relax U Eqn

```
solve(UEqn == -fvc::grad(p));
```

Momentum predictor

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

Pressure equation

Under-relax p

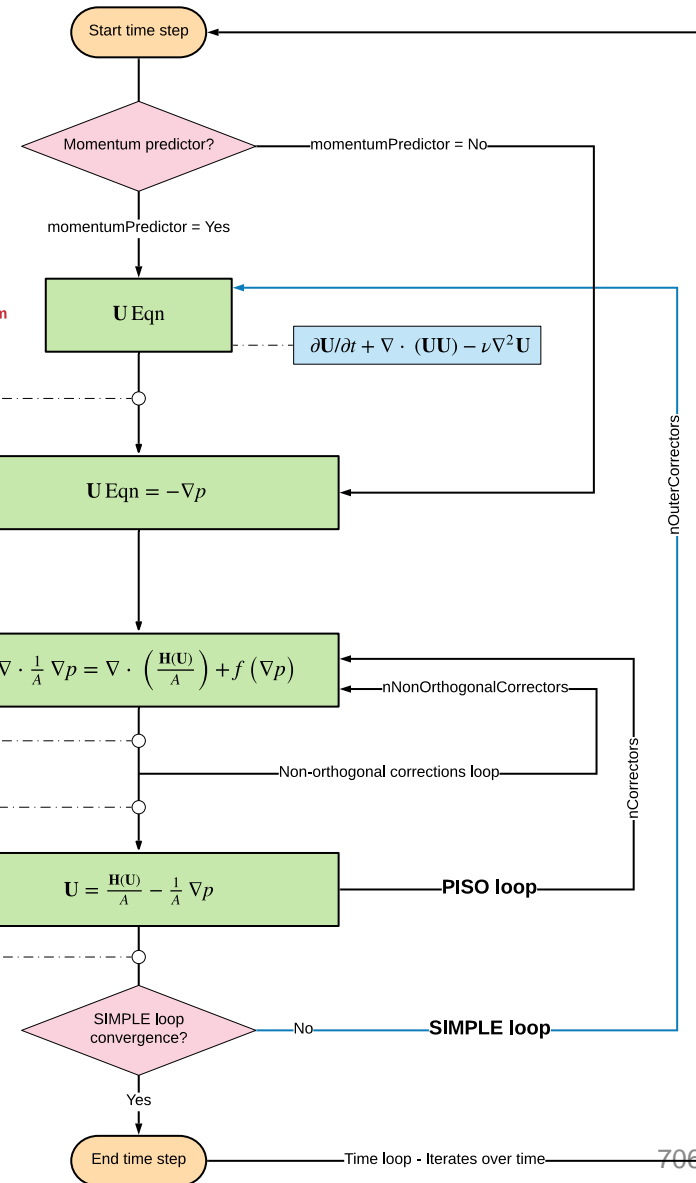
Update flux $\phi = S_f \cdot \left[(H/A)_f - (1/A)_f (\nabla p)_f \right]$

```
U = HbyA - rAU*fvc::grad(p);
```

Momentum corrector

Solve additional transport equations

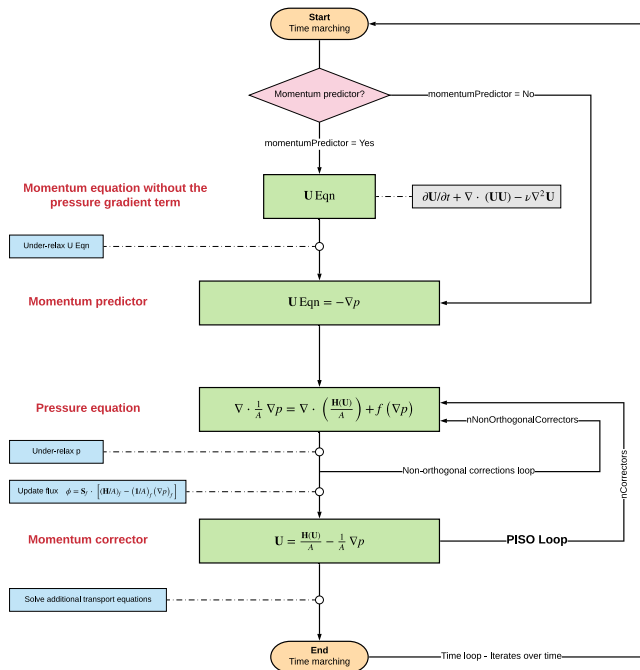
This is an excerpt of the actual source code of the solver



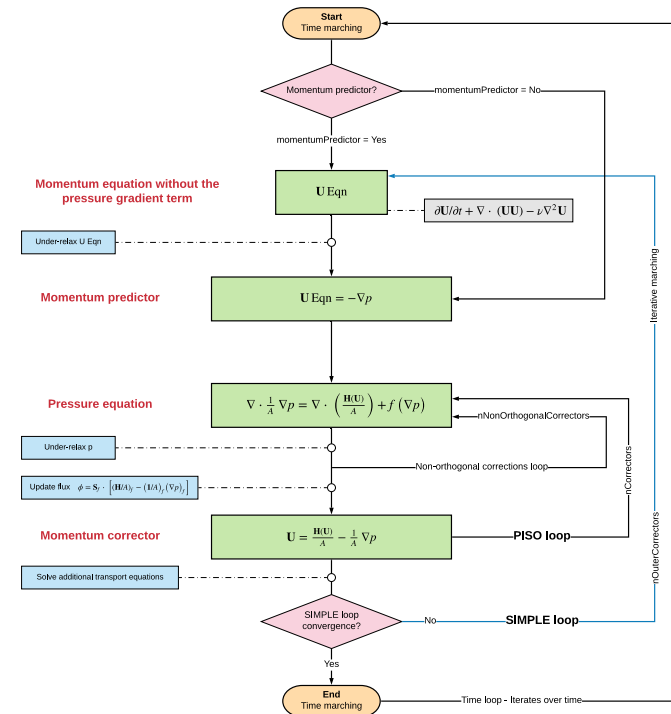
Pressure-Velocity coupling in OpenFOAM®

Comparison of PISO with non-iterative time-advancement (PISO-NITA) against PISO with Iterative time-advancement (PISO-ITA)

- The main difference between both methods is the outer loop present in the **PISO-ITA**.
- This outer loop gives more stability and allow the use of very large time-steps (CFL numbers).
- The recommended CFL number of the **PISO-NITA** is below 2 (for good accuracy and stability).



PISO-NITA



PISO-ITA (PIMPLE in OpenFOAM®)

Roadmap

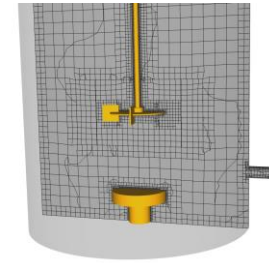
- ~~1. Finite Volume Method: A Crash Introduction~~
- ~~2. On the CFL number~~
- ~~3. Linear solvers in OpenFOAM®~~
- ~~4. Pressure-Velocity coupling in OpenFOAM®~~
- 5. Unsteady and steady simulations**
- ~~6. Understanding residuals~~
- ~~7. Boundary and initial conditions~~
- ~~8. Numerical playground~~

Unsteady and steady simulations

- Nearly all flows in nature and industrial applications are unsteady (also known as transient or time-dependent).
- Unsteadiness can be due to:
 - Instabilities.
 - Non-equilibrium initial conditions.
 - Time-dependent boundary conditions.
 - Source terms.
 - Chemical reactions and finite rate chemistry.
 - Phase change.
 - Moving or deforming bodies.
 - Turbulence.
 - Buoyancy and heat transfer.
 - Discontinuities.
 - Multiple phases.
 - Fluid structure interaction.
 - Combustion.
 - And much more.

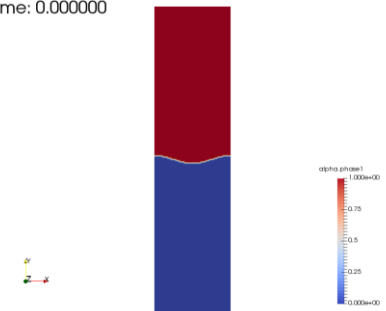


Sliding grids – Continuous stirred tank reactor
www.wolfdynamics.com/wiki/FVM_uns/ani5.gif

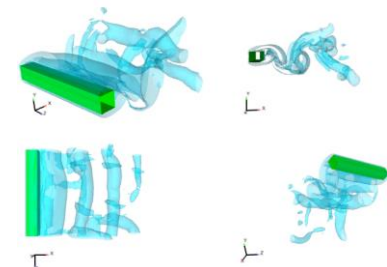


Multiphase flow
www.wolfdynamics.com/wiki/FVM_uns/ani3.gif

Time: 0.000000



Turbulent flows - SRS
www.wolfdynamics.com/wiki/FVM_uns/ani4.gif



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®?

```
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®?

```
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.
- To reduce parsing time and file size, it is recommended to use binary format to write the solution.

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;
```

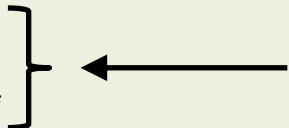
```
timePrecision 6;
```

```
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, and at least 2 outer correctors in the **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;
    }
}

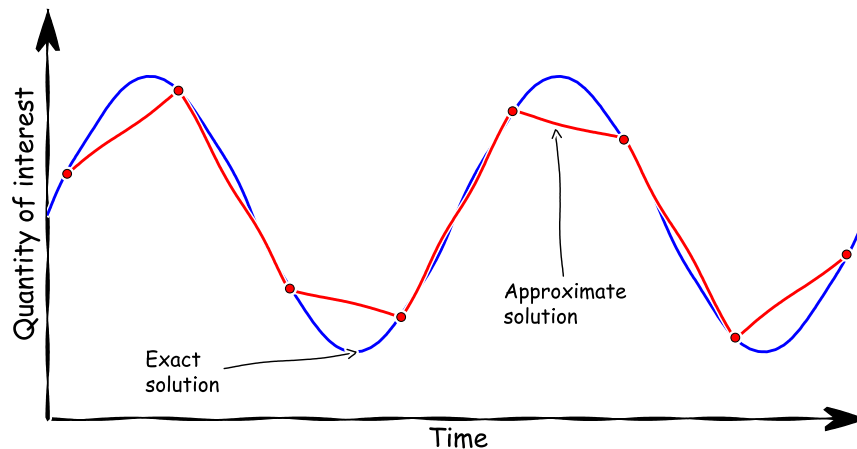
PIMPLE
{
    nOuterCorrectors 1;
    nCorrectors      2;
    nNonOrthogonalCorrectors 1;
}
```

- 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.
- To set these parameters, follow the guidelines given in the previous section.
- Depending on the solver you are using, you will need to define the sub-dictionary **PISO** or **PIMPLE**.
- Setting the keyword **nOuterCorrectors** to 1 in **PIMPLE** solvers is equivalent to running using the **PISO** method.
- To gain more stability, especially when using large time-steps, you can use more outer correctors (**nOuterCorrectors**).
- 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.
- Remember, in both **PISO** and **PIMPLE** method you need to do at least one correction (**nCorrectors**).
- Adding corrections increase the computational cost (**nOuterCorrectors** and **nCorrectors**).

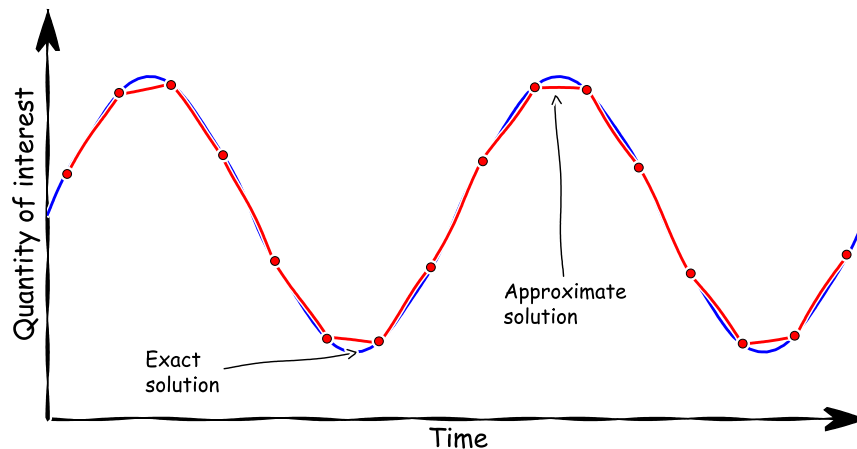
Unsteady and steady simulations

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

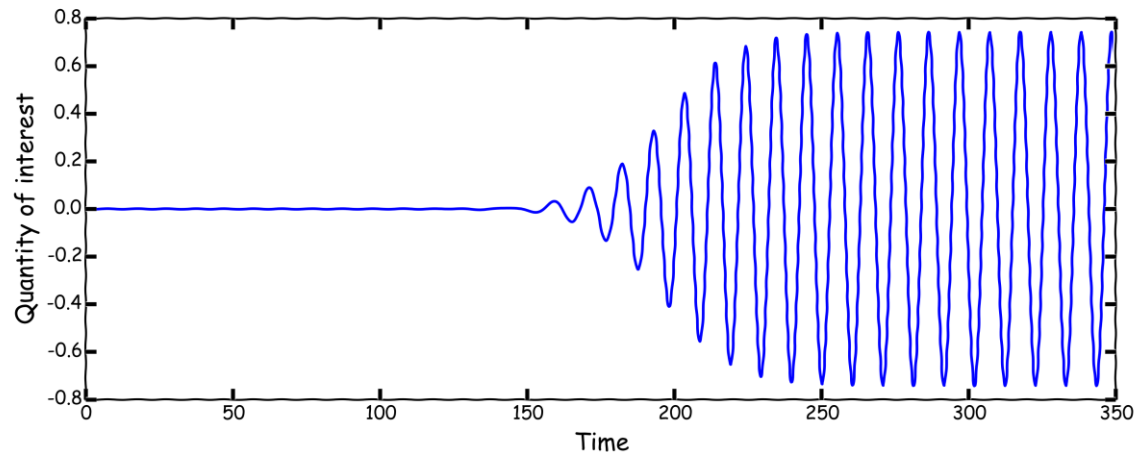
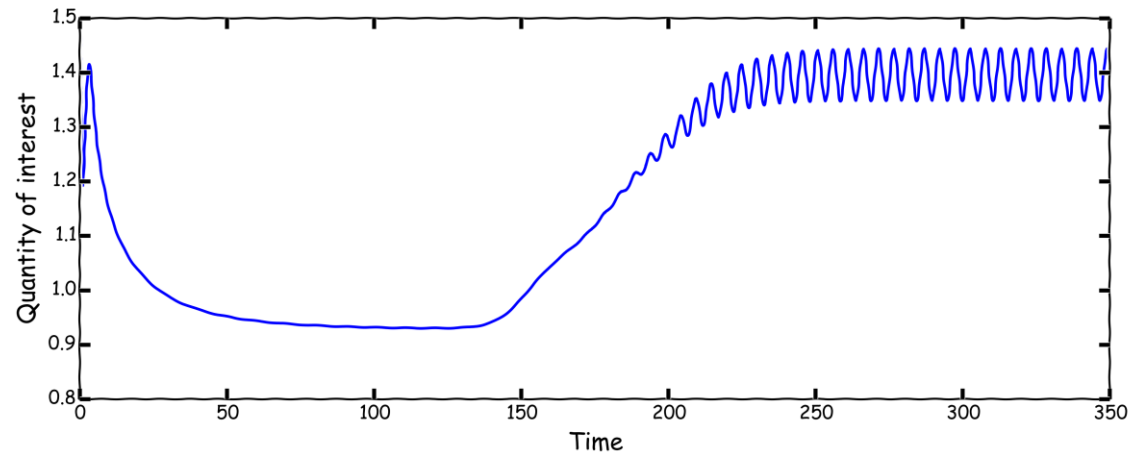


By using a smaller time step you resolve better the physics and you gain stability

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

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.
- Remember, most of the flows you will encounter are unsteady so be careful of this hypothesis.
- In steady simulations, we made two assumptions:
 - We ignore unsteady fluctuations. That is, we neglect the time 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.
- To do so, you need to use the appropriate solver and use the right discretization scheme.
- As you are not solving the time 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

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).
- Remember to set also the saving frequency.
- In the *fvSchemes* dictionary you need to set the time 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** or **SIMPLEC**), 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®?

- The under-relaxation factors (URF) control the change of the variable ϕ .

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

- Under-relaxation is a feature typical of steady solvers using the **SIMPLE** family of methods.
- These are the URF commonly used with **SIMPLE** and **SIMPLEC** (industry standard),

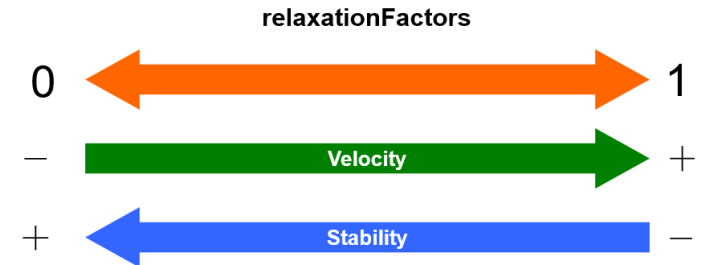
SIMPLE		SIMPLEC		Pressure Usually does not require under-relaxing
p	0.3;	p	1;	
U	0.7;	U	0.9;	
k	0.7;	k	0.9;	
omega	0.7;	omega	0.9;	

- According to the physics involved you will need to add more under-relaxation factors.
- Finding the right URF involved experience and some trial and error.
- Selecting the URF it is kind of equivalent to selecting the right time step.
- Many times, steady simulations diverge because of wrongly chosen URF.

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

- The URF are bounded between 0 and 1.
- If you set the URF close to one you increase the convergence rate but loose solution stability.
- On the other hand, if you set the URF close to zero you gain stability but reduce convergence rate.



- 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.
- Under-relaxation can be implicit (equation in OpenFOAM) or explicit (field in OpenFOAM).

$$\frac{a_P \phi}{\alpha} = \sum_N a_N \phi_N + b + \frac{1 - \alpha}{\alpha} a_P \phi_{n-1}$$

Implicit URF

$$\phi = \phi_{n-1} + \alpha \Delta \phi$$

Explicit URF

- You can relate URF to the CFL number as follows,

$$CFL = \frac{\alpha}{1 - \alpha}$$

$$\alpha = \frac{CFL}{1 + CFL}$$

- A small CFL number is equivalent to small URF.

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®?

```
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

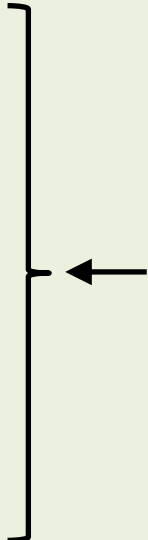
How to run steady simulations in OpenFOAM®?

```
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;
    }
}
```



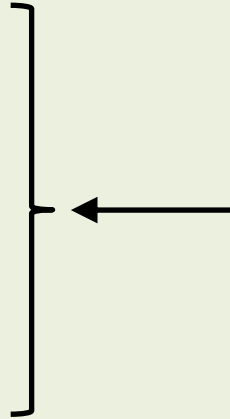
- The *fvSolution* dictionary contains the instructions of how to solve each discretized linear equation system.
- As for the *controlDict* and *fvSchemes* dictionaries, the parameters can be changed on-the-fly.
- To set these parameters, follow the guidelines given in the previous section.
- Increasing 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.
- 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 prior to solving for a field (**equations** keyword) or by modifying the field directly (**fields** keyword).
- Under-relaxing the equations is also known as implicit under-relaxation.
- Whereas, under-relaxing the fields is also known as explicit under-relaxation.
- 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.
- In this case we are using the industry standard URF.
- Remember, URF are problem dependent.
- If you do not define URF, the solver will not under-relax.

Unsteady and steady simulations

How to run steady simulations in OpenFOAM®?

- To enable the consistent formulation of the **SIMPLE** method, you need to add the following keyword to the **SIMPLE** sub-dictionary,

```
SIMPLE
{
    consistent yes;
    nNonOrthogonalCorrectors 3;
}
```

Enabled/disabled consistent formulation of the SIMPLE loop

- The following URF are recommended,

SIMPLE

```
relaxationFactors
{
    fields
    {
        p 0.3;
    }
    equations
    {
        U 0.7;
        k 0.6;
        omega 0.6;
    }
}
```

SIMPLEC

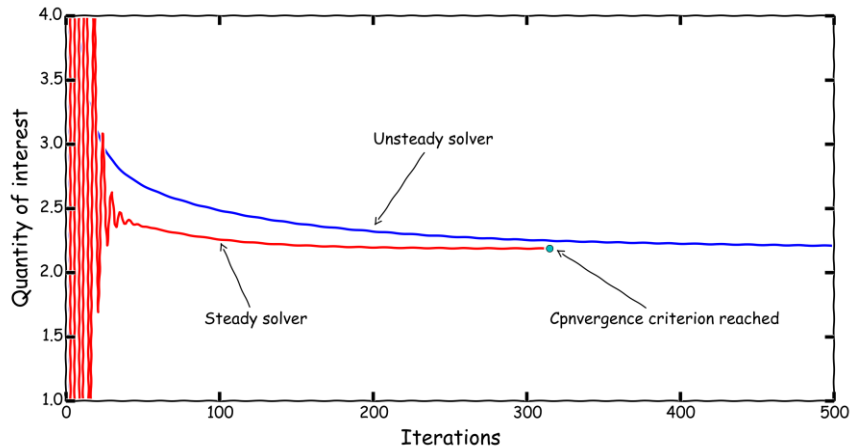
```
relaxationFactors
{
    fields
    {
        p 0.7;
    }
    equations
    {
        U 0.7;
        k 0.7;
        omega 0.7;
    }
}
```

Unsteady and steady simulations

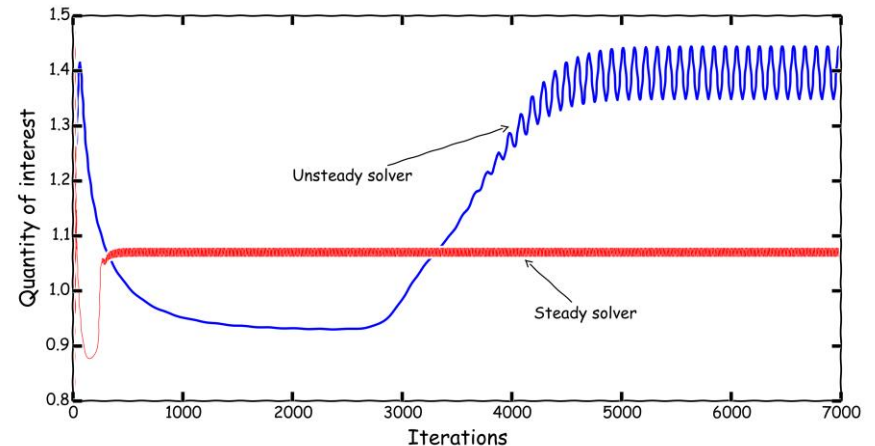
Steady simulations vs. Unsteady simulations

- 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, is not a good idea to compute a dominant frequency using steady simulations, e.g., vortex shedding frequency.

Steady solution QOI



unsteady solution QOI

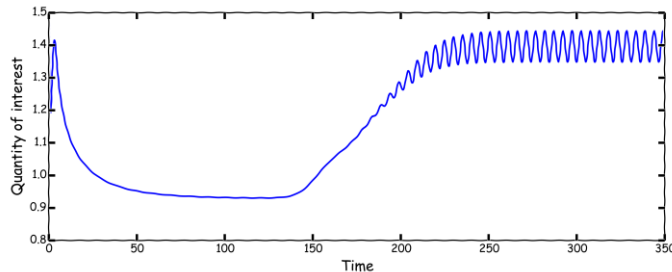


Roadmap

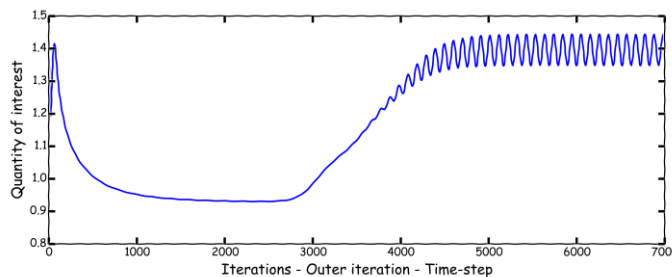
- ~~1. Finite Volume Method: A Crash Introduction~~
- ~~2. On the CFL number~~
- ~~3. Linear solvers in OpenFOAM®~~
- ~~4. Pressure-Velocity coupling in OpenFOAM®~~
- ~~5. Unsteady and steady simulations~~
- 6. Understanding residuals**
- ~~7. Boundary and initial conditions~~
- ~~8. Numerical playground~~

Understanding residuals

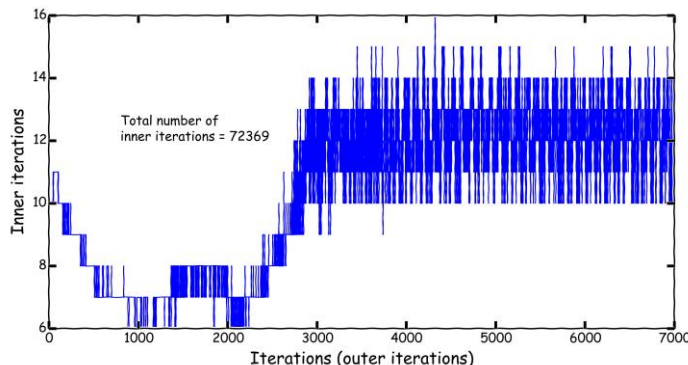
- Before talking about residuals, let us clarify something.
- When we talk about iterations in unsteady simulations, we are talking about the time-step or outer-iterations.



← 1. To arrive to this physical time of the monitored QOI



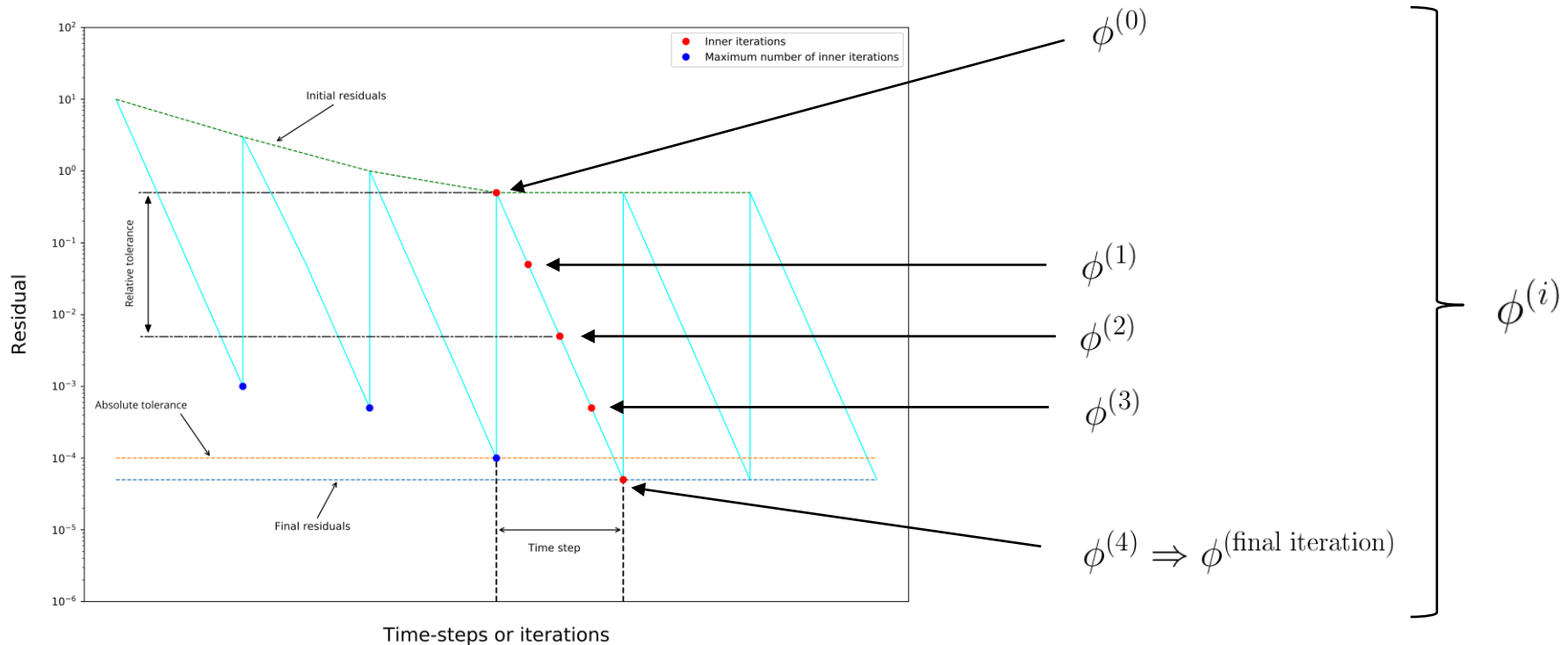
← 2. We iterate this many times



← 3. And we iterate inside each time-step (or outer-iteration), until reaching the linear solver tolerance or maximum number of iterations.

Understanding residuals

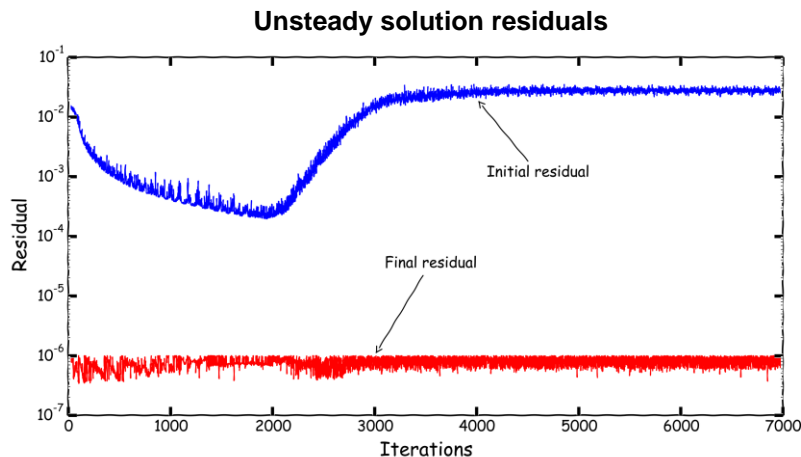
- To get a better idea of how iterative methods work, and what are initial residuals and final residuals, let us take another look at a residual plot.



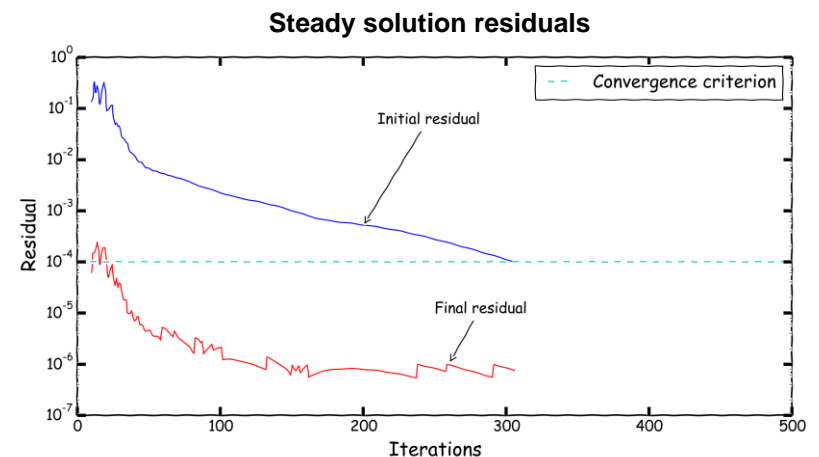
- $\phi^{(0)}$ is the initial guess used to start the iterative solver.
- You can use any value at iteration 0, but usually is a good choice to take the previous solution vector.
- If the following condition is fulfilled $\|A\phi^i - b\| \leq |r|$ (where r is the convergence criterion or tolerance), the linear solver will stop iterating and will advance to the next time-step.
- By working in an iterative way, every single iteration $\phi^{(i)}$ is a better approximation of the previous iteration $\phi^{(i-1)}$.
- Sometimes the linear solver might stop before reaching the predefined convergence criterion because it has reached the maximum number of iterations, you should be careful of this because we are talking about unconverged iterations.

Understanding residuals

- This is a typical residual plot for an unsteady simulation.
- Ideally, the solution should converge at every time-step (final residuals tolerance).
- If the solution is not converging, that is, the residuals are not reaching the predefined final residual tolerance, try to reduce the time-step size.
- 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.
- You can also increase the number of maximum inner iterations.
- If the initial residuals fall below the convergence criterion, you might say that you have arrived at a steady solution (the exception rather than the rule).



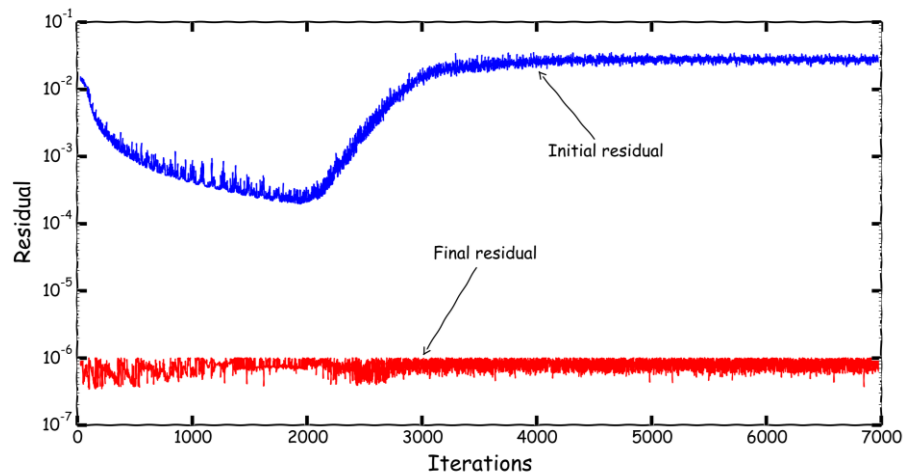
- This is a typical residual plot for a steady simulation.
- In this case, the initial residuals are falling below the convergence criterion (monotonic convergence), hence we have reached a steady-state.
- In the solver does not reach the convergence criteria or the residuals get stalled, it does not mean that the solution is diverging, it is just an indication of unsteadiness, and it might be better to run using an unsteady solver.
- In comparison to unsteady solvers, steady solvers require less iterations to arrive to a converge solution, if they arrive.



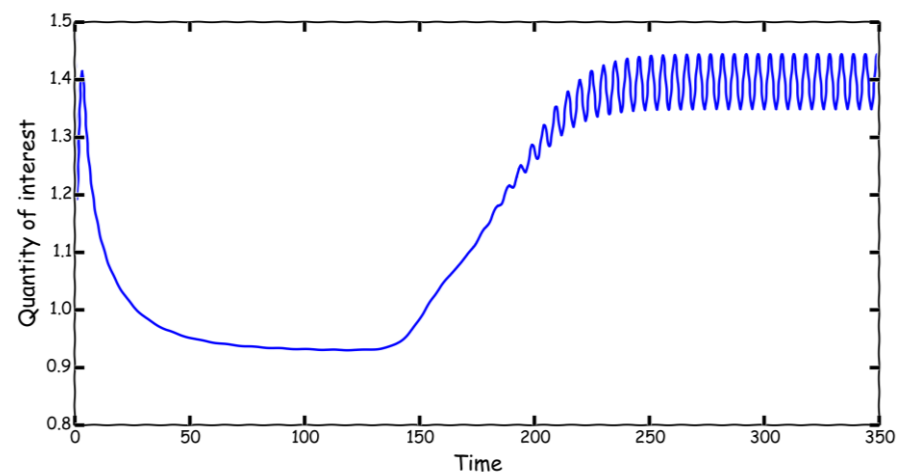
Understanding residuals

- Remember, residuals are not a direct indication that you are converging to the right solution.
- It is better to monitor a quantity of interest (QOI).
- And by the way, you should get physically realistic values.
- In this case, if you monitor the residuals, you might get the impression that the simulation is diverging.
- Instead, if you monitor a QOI you will realize that there is an initial transient (long one by the way), then the onset of an instability, and then a periodic behavior of the phenomenon.
- You should assess the convergence of the solution and compute the unsteady statistics in the time window where the behavior of the QOI is periodic.
- To monitor the stability, you can check the minimum and maximum values of the field variables.
- If you have bounded quantities, check that you do not have over-shoots or under-shoots.

Residuals

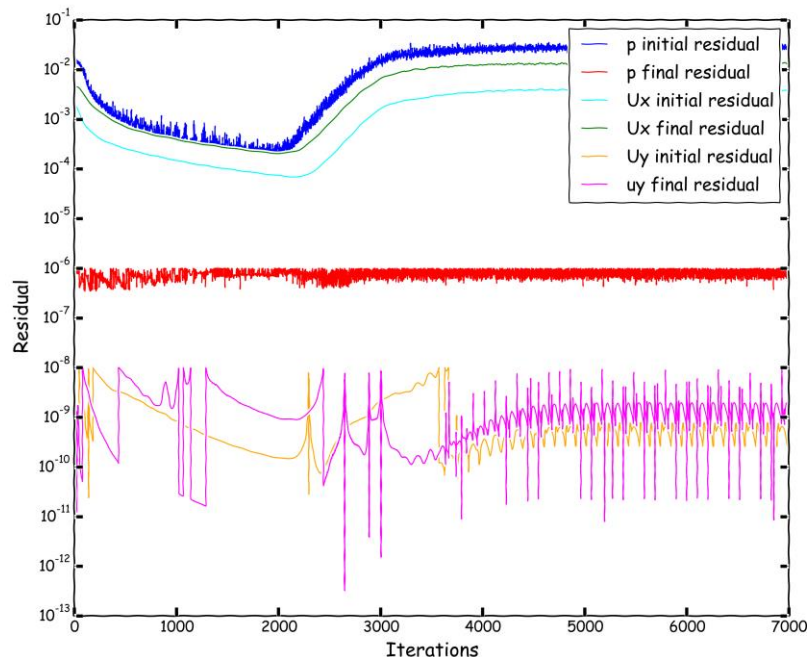


QOI

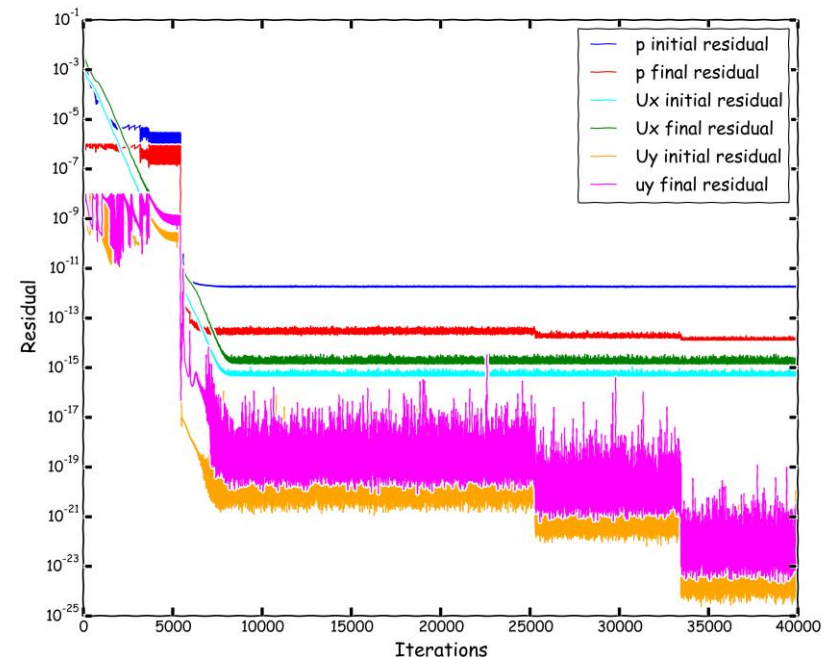


Understanding residuals

- This is the output of the residuals for all field variables of an unsteady case.
- Notice that at the beginning the residuals show a monotonic behavior.
- Then, after a while the convergence rate changes.
- This not necessarily means that the solution is diverging, it might be an indication of unsteadiness.

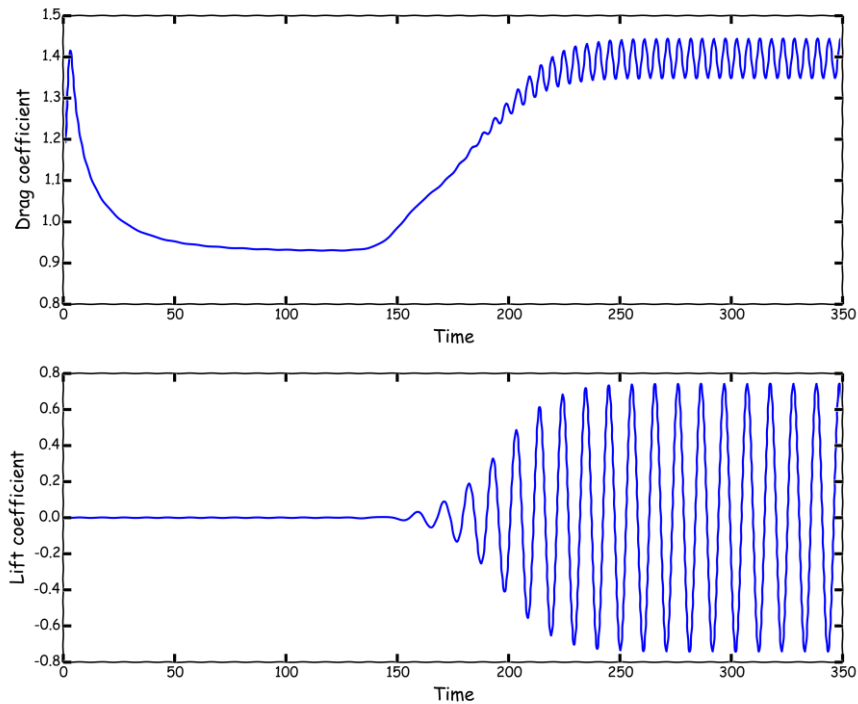


- This is the output of the residuals for all field variables of a steady case.
- The jumps are due to the changes in tolerance introduced while running the simulation.
- As you can see, the residuals are falling in a monotonic way.

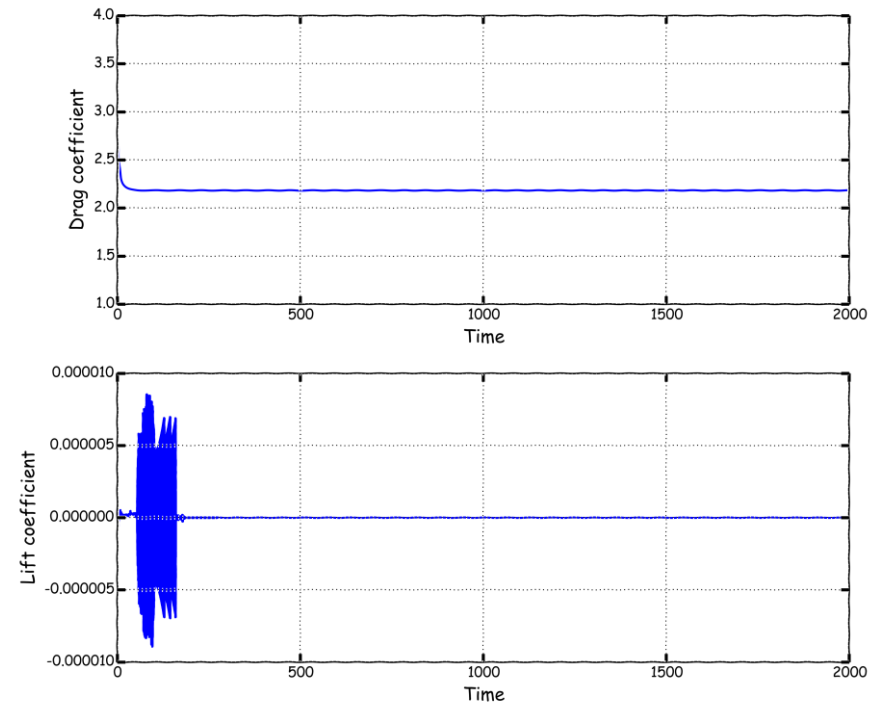


Understanding residuals

- This is the output of the aerodynamic coefficients for an unsteady case.



- This is the output of the aerodynamic coefficients for a steady case.



Roadmap

- ~~1. Finite Volume Method: A Crash Introduction~~
- ~~2. On the CFL number~~
- ~~3. Linear solvers in OpenFOAM®~~
- ~~4. Pressure-Velocity coupling in OpenFOAM®~~
- ~~5. Unsteady and steady simulations~~
- ~~6. Understanding residuals~~
- 7. Boundary and initial conditions**
- ~~8. Numerical playground~~

Boundary conditions and initial conditions

On the initial boundary value problem (IBVP)

- First of all, when we use a CFD solver to find the approximate solution of the governing equations, we are solving an Initial Boundary Value Problem (IBVP).
- In an IBVP, we need to impose appropriate boundary conditions and initial conditions.
- No need to say that the boundary conditions and initial conditions need to be physically realistic.
- Boundary conditions are a required component of the numerical method, they tell the solver what is going on at the boundaries of the domain.
- You can think of boundary conditions as source terms.
- Initial conditions are also a required component of the numerical method, they define the initial state of the problem.

Boundary conditions and initial conditions

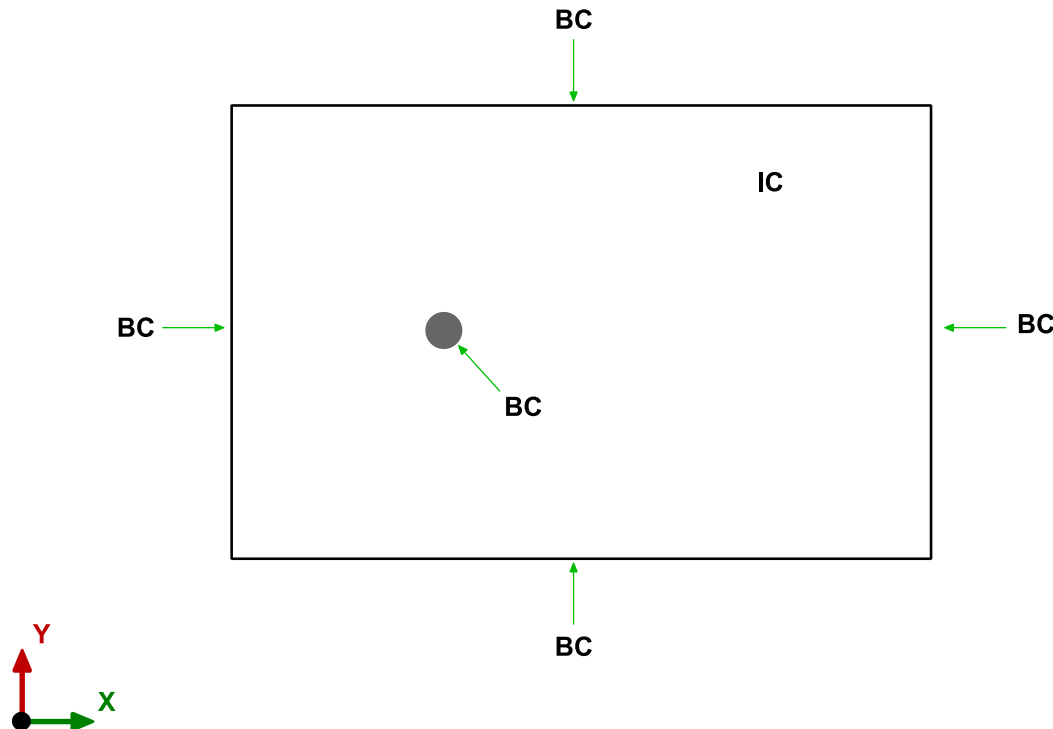
A few words about boundary conditions

- Boundary conditions (BC) can be divided into three fundamental mathematical types:
 - **Dirichlet boundary conditions:** when we use this BC, we prescribe the value of a variable at the boundary.
 - **Neumann boundary conditions:** when we use this BC, we prescribe the gradient normal to the boundary.
 - **Robin Boundary conditions:** this BC is a mixed of Dirichlet boundary conditions and Neumann boundary
- You can use any of these three boundary conditions in OpenFOAM®.
- During this discussion, the semantics is not important, that depends of how you want to call the BCs or how they are named in the solver, *i.e.*, in, inlet, inflow, velocity inlet, incoming flow and so on.
- Defining boundary conditions involves:
 - Finding the location of the boundary condition in the domain.
 - Determining the boundary condition type.
 - Giving the required physical information.
- The choice of the boundary conditions depend on:
 - Geometrical considerations.
 - Physics involved.
 - Information available at the boundary condition location.
 - Numerical considerations.
- And most important, you need to understand the physics involved.

Boundary conditions and initial conditions

A few words about boundary conditions

- To define boundary conditions you need to know the location of the boundaries (where they are in your mesh).
- You also need to supply the information at the boundaries.
- Last but not least important, you must know the physics involved.



Boundary conditions and initial conditions

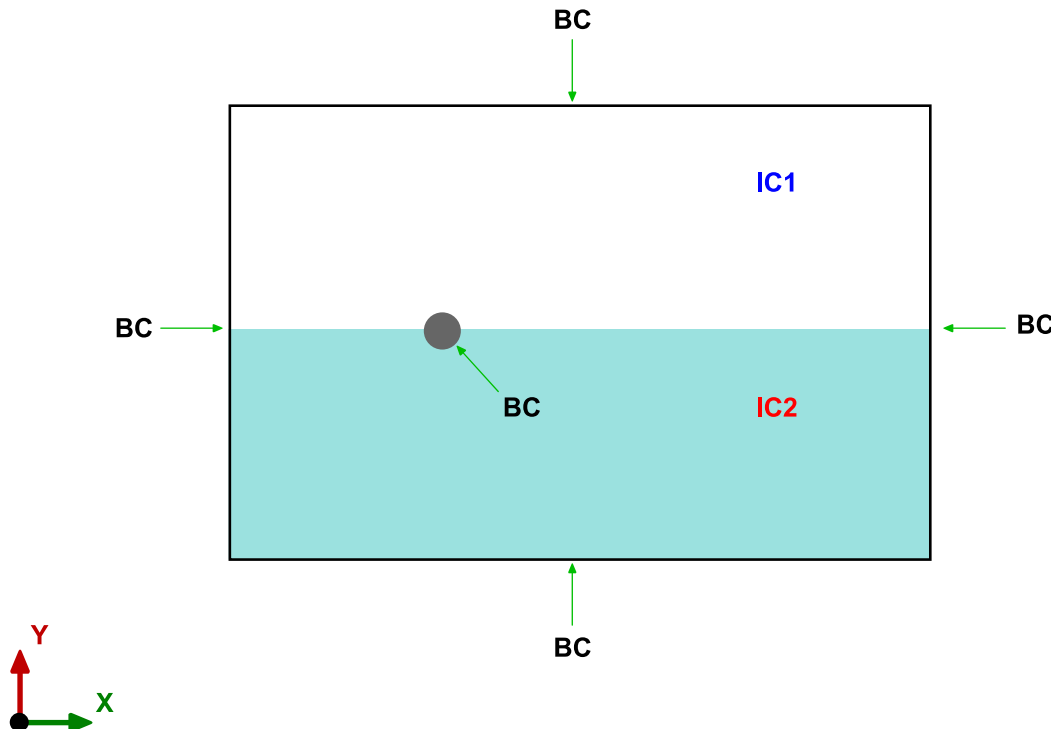
A few words about initial conditions

- Initial conditions (IC) can be divided into two groups:
 - **Uniform initial conditions.**
 - **Non-uniform initial conditions.**
- For non-uniform IC, the value used can be obtained from:
 - Another simulation (including a solution with different grid resolution).
 - A mathematical function
 - A potential solver.
 - Reduced order models.
 - Experimental results.
- Defining initial conditions involves:
 - Finding the location of the initial condition in the domain.
 - Determining the initial condition type.
 - Giving the required physical information.
- The choice of the initial conditions depend on:
 - Geometrical considerations.
 - Physics involved.
 - Information available.
 - Numerical considerations.
- And most important, you need to understand the physics involved.

Boundary conditions and initial conditions

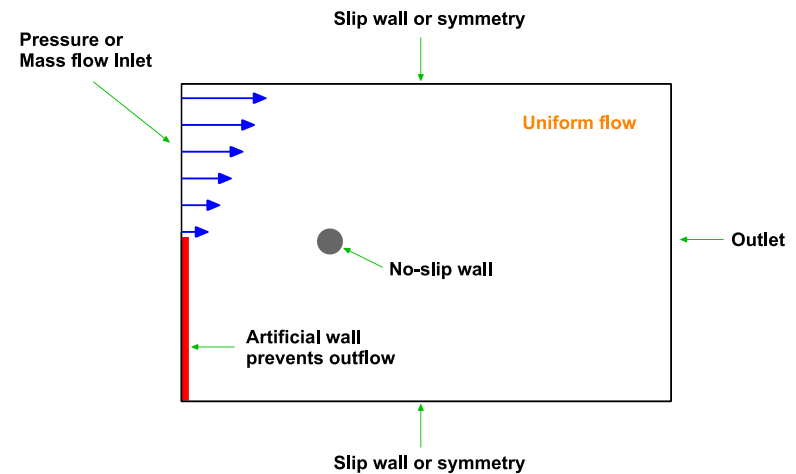
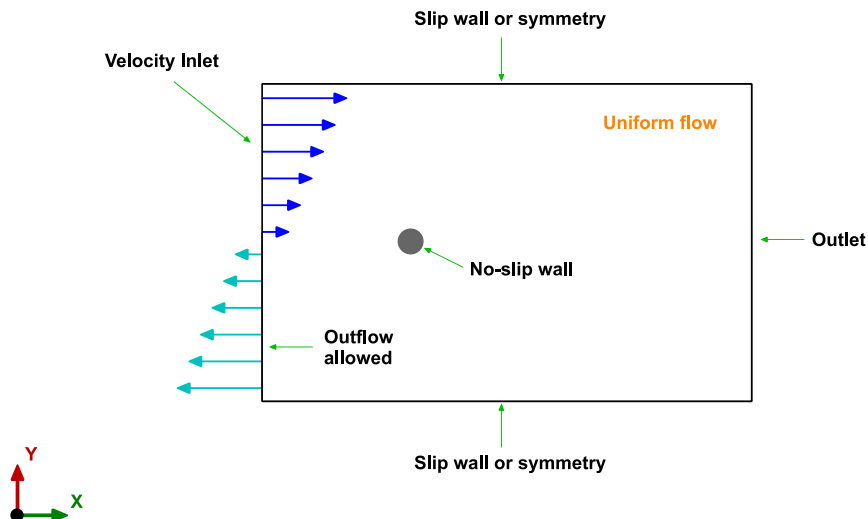
A few words about initial conditions

- For initial conditions, you need to supply the initial information or initial state of your problem.
- This information can be a uniform value or a non-uniform value.
- You can apply the initial conditions to the whole domain or separated zones of the domain.
- Last but not least important, you must know the physics involved.



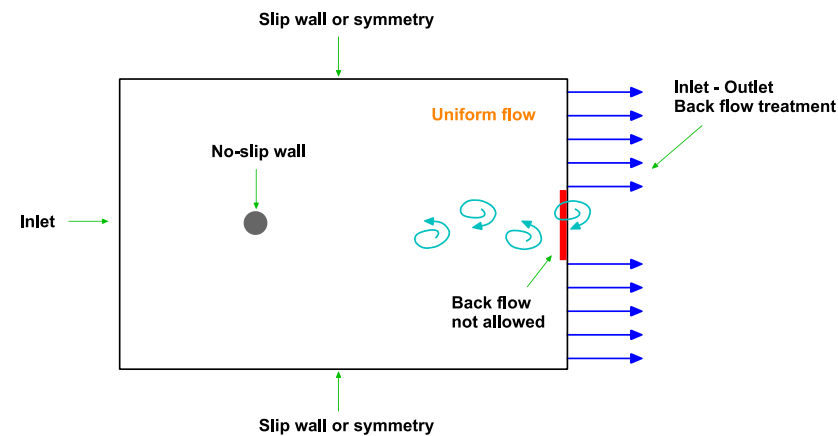
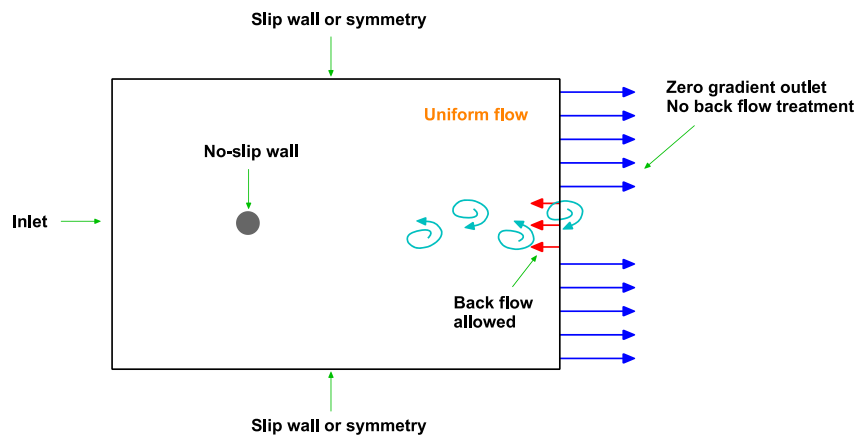
Boundary conditions and initial conditions

- **Inlets and outlets boundary conditions:**
 - Inlets are for regions where inflow is expected; however, inlets might support outflow when a velocity profile is specified.
 - Pressure boundary conditions do not allow outflow at the inlets.
 - Velocity specified inlets are intended for incompressible flows.
 - Pressure and mass flow inlets are suitable for compressible and incompressible flows.
 - Same concepts apply to outlets, which are regions where outflow is expected.



Boundary conditions and initial conditions

- **Zero gradient (Neumann) and backflow boundary conditions:**
 - Zero gradient boundary conditions extrapolates the values from the domain. They require no information.
 - Zero gradient boundary conditions can be used at inlets, outlets, and walls.
 - Backflow boundary conditions provide a generic outflow/inflow condition, with specified inflow/outflow for the case of backflow.
 - In the case of a backflow outlet, when the flux is positive (out of domain) it applies a Neumann boundary condition (zero gradient), and when the flux is negative (into of domain), it applies a Dirichlet boundary condition (fixed value).
 - Same concept applies to backflow inlets.



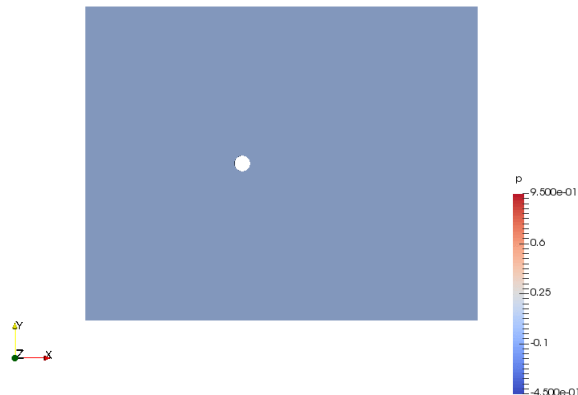
Boundary conditions and initial conditions

- **On the outlet pressure boundary condition**
 - Some combinations of boundary conditions are very stable, and some are less reliable.
 - And some configurations are unreliable.
 - Inlet velocity at the inlet and pressure zero gradient at the outlet. This combination should be avoided because the static pressure level is not fixed.
 - Qualitatively speaking, the results are very different.
 - This simulation will eventually crash.



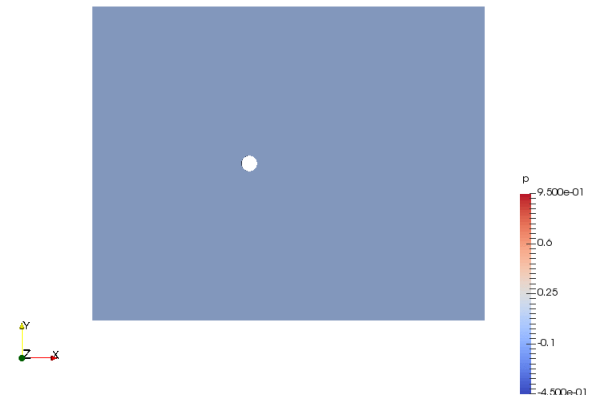
BCs 1. Inlet velocity and fixed outlet pressure
www.wolfdynamics.com/wiki/BC/aniBC1.gif

Time: 0.000000



BCs 2. Inlet velocity and zero gradient outlet pressure
www.wolfdynamics.com/wiki/BC/aniBC2.gif

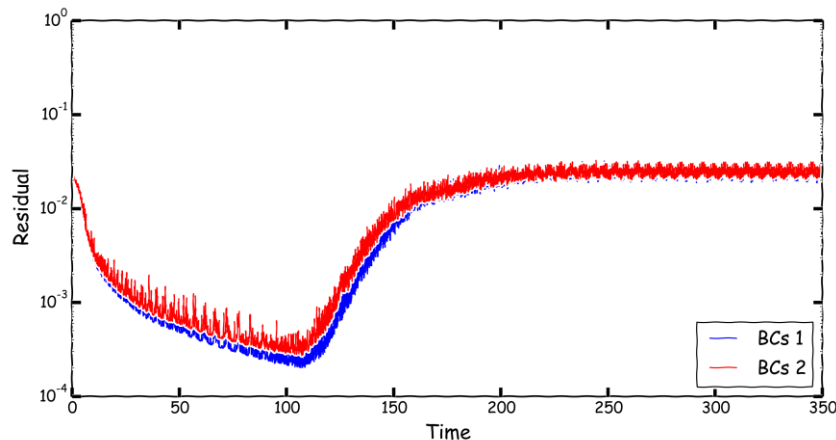
Time: 0.000000



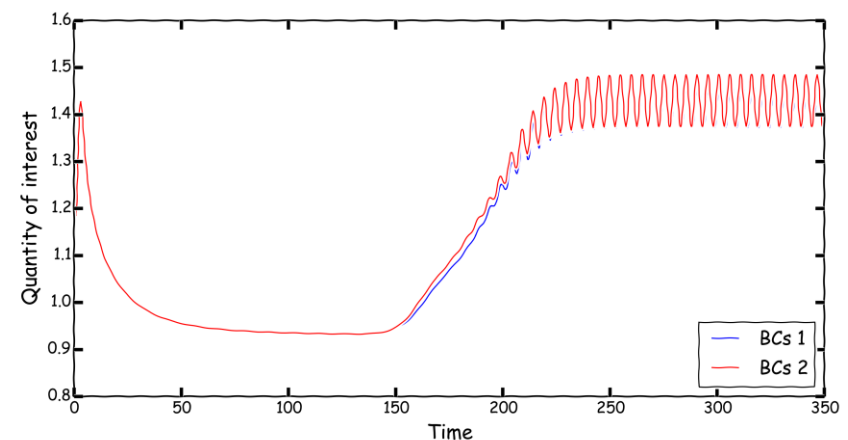
Boundary conditions and initial conditions

- **On the outlet pressure boundary condition**
 - If you only rely on a QOI and the residuals, you will not see any major difference between the two cases with different outlet pressure boundary condition.
 - This is very misleading.
 - However, when you visualize the solution, you will realize that something is wrong. This is a case where pretty pictures can be used to troubleshoot the solution.
 - Quantitative speaking, the results are very similar.
 - However, this simulation will eventually crash.

Residual plot for pressure

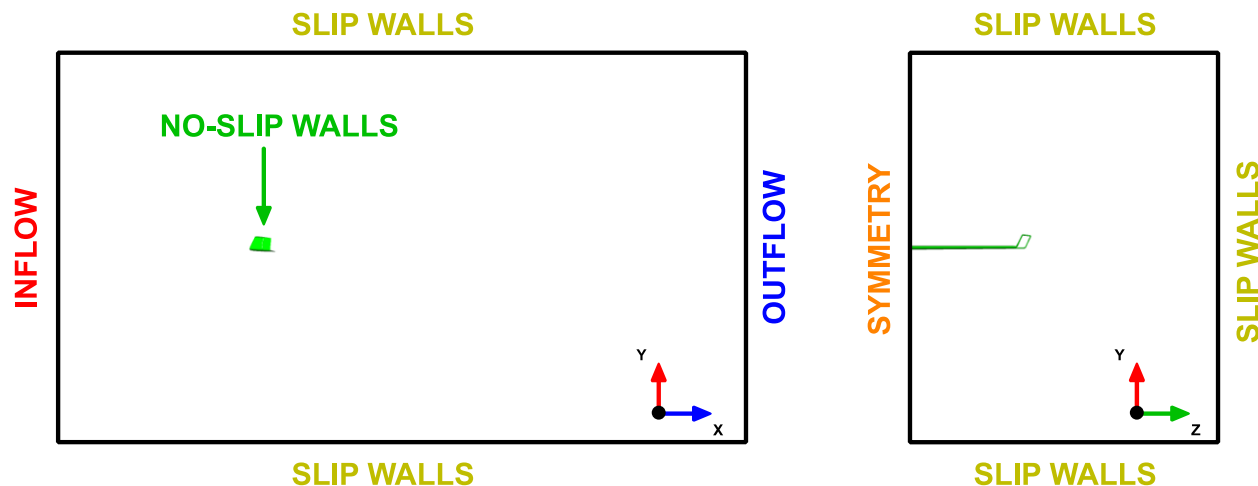


Quantity of interest – Force coefficient on the body



Boundary conditions and initial conditions

- **Symmetry boundary conditions:**
 - Symmetry boundary conditions are a big simplification of the problem. However, they help to reduce mesh cell count.
 - Have in mind that symmetry boundary conditions only apply to **planar faces**.
 - To use symmetry boundary conditions, both the geometry and the flow field must be symmetric.
 - Mathematically speaking, setting a symmetry boundary condition is equivalent to zero normal velocity at the symmetry plane, and zero normal gradients of all variables at the symmetry plane.
 - Physically speaking, they are equivalent to slip walls.



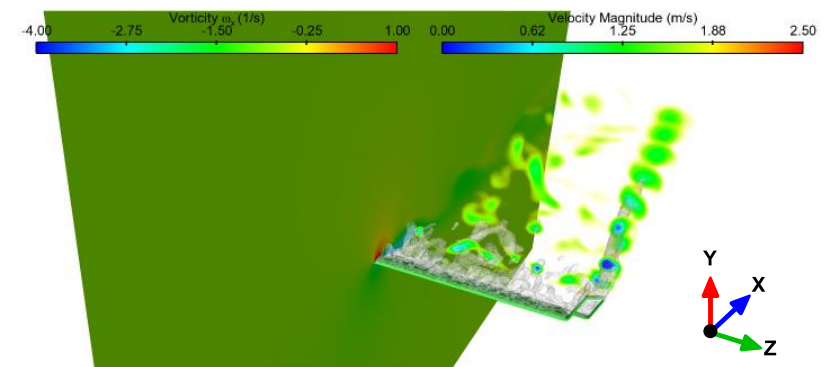
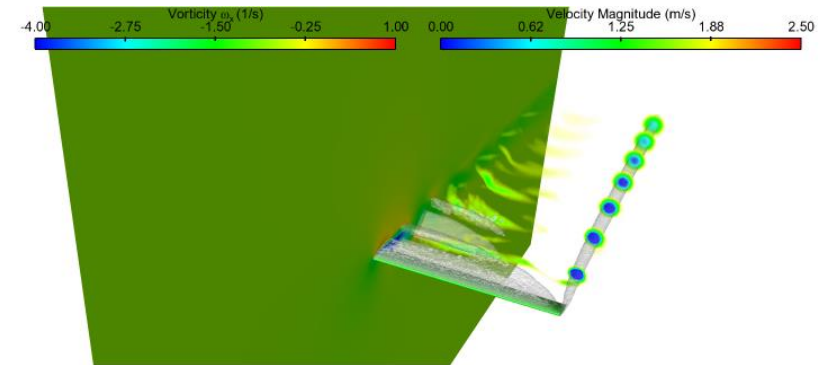
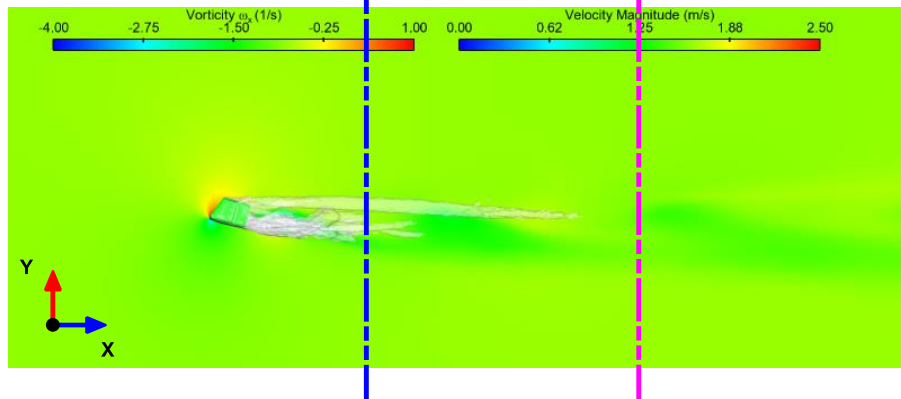
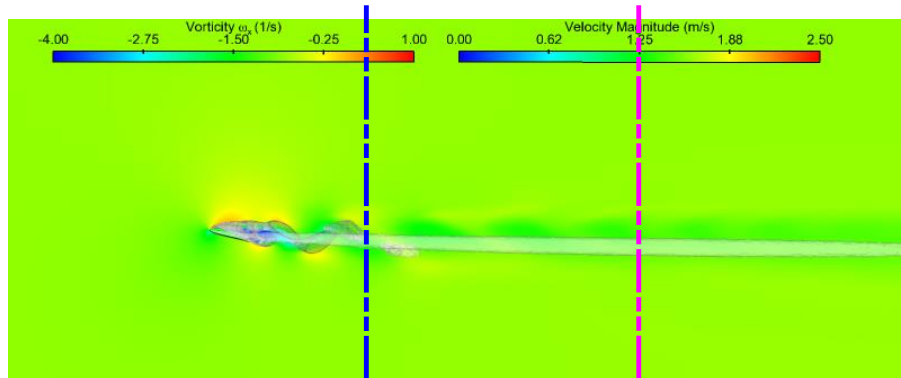
Boundary conditions and initial conditions

- **Location of the outlet boundary condition:**
 - Place outlet boundary conditions as far as possible from recirculation zones or backflow conditions, by doing this you increase the stability.
 - Remember, backflow conditions requires special treatment.

Possible backflow

Might be OK

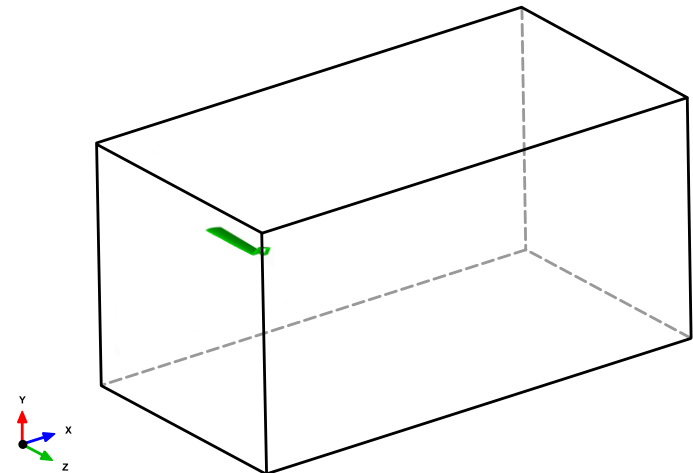
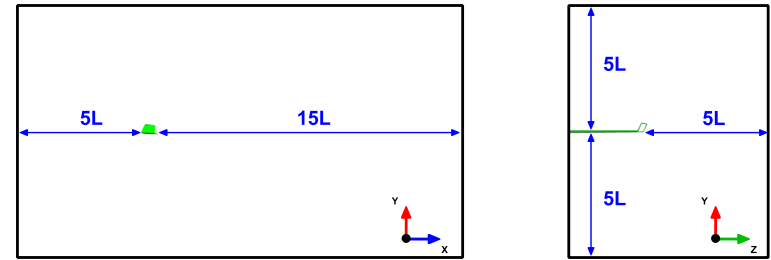
Far enough so the flow can be considered fully developed



Boundary conditions and initial conditions

Domain dimensions (when the dimensions are not known)

- If you do not have any constrain in the domain dimensions, you can use as a general guideline the dimensions illustrated in the figure, where L is a reference length (in this case, L is the wing chord).
- The values illustrated in the figure are on the conservative side, but if you want to play safe, multiply the values by two or more.
- Always verify that there are no significant gradients normal to any of the boundaries patches.
- If there are, you should consider increasing the domain dimensions.
- The larger the domain, the better.



Boundary conditions and initial conditions

A few considerations and guidelines

- Boundary conditions and initial conditions need to be physically realistic.
- Poorly defined boundary conditions can have a significant impact on your solution.
- Initial conditions are as important as the boundary conditions.
- A good initial condition can improve the stability and convergence rate.
- On the other hand, unphysical initial conditions can slow down the convergence rate or can cause divergence.
- You need to define boundary conditions and initials conditions for every single variable you are solving.
- Setting the right boundary conditions is extremely important, but you need to understand the physics.
- You need to understand the physics in order to set the right boundary conditions.
- Do not force the flow at the outlet, use a zero normal gradient for all flow variables except pressure. The solver extrapolates the required information from the interior.
- Be careful with backward flow at the outlets (flow coming back to the domain) and backward flow at inlets (reflection waves), they required special treatment.
- If possible, select inflow and outflow boundary conditions such that the flow either goes in or out normal to the boundaries.
- At outlets, use zero gradient boundary conditions only with incompressible flows and when you are sure that the flow is fully developed.
- Outlets that discharge to the atmosphere can use a static pressure boundary condition. This is interpreted as the static pressure of the environment into which the flow exhausts.

Boundary conditions and initial conditions

A few considerations and guidelines

- Inlets that take flow into the domain from the atmosphere can use a total pressure boundary condition (e.g. open window).
- Mass flow inlets produce a uniform velocity profile at the inlet.
- Pressure specified boundary conditions allow a natural velocity profile to develop.
- The required values of the boundary conditions and initial conditions depend on the equations you are solving, and physical models used, e.g.,
 - For incompressible and laminar flows you will need to set only the velocity and pressure.
 - If you are solving a turbulent compressible flow you will need to set velocity, pressure, temperature and the turbulent variables.
 - For multiphase flows you will need to set the primitives variables for each phase. You will also need to initialize the phases.
 - If you are doing turbulent combustion or chemical reactions, you will need to define the species, reactions and turbulent variables.
- Minimize grid skewness, non-orthogonality, growth rate, and aspect ratio near the boundaries. You do not want to introduce diffusion errors early in the simulation, especially close to the inlets.
- Try to avoid large gradients in the direction normal to the boundaries and near inlets and outlets.
 - That is to say, put your boundaries far away from where things are happening.

Boundary conditions and initial conditions

- OpenFOAM® distinguish between **base type** boundary conditions and **numerical type** boundary conditions.

Base type boundary conditions

- Base type boundary conditions are based on geometry information (surface patches) or on inter-processor communication link (halo boundaries).
- Base type boundary conditions are defined in the file *boundary* located in the directory **constant/polyMesh**
- The file *boundary* is automatically created when you generate or convert the mesh.
- When you convert a mesh to OpenFOAM® format, you might need to manually modify the file *boundary*. This is because the conversion utilities do not recognize the boundary type of the original mesh.
- Remember, if a base type boundary condition is missing, OpenFOAM® will complain and will tell you where and what is the error.
- Also, if you misspelled something OpenFOAM® will complain and will tell you where and what is the error

Numerical type boundary conditions

- Numerical type boundary condition assigns the value to the field variables in the given surface patch.
- Numerical type boundary conditions are defined in the field variables dictionaries located in the directory **0** (e.g. *U*, *p*).
- When we talk about numerical type boundary conditions, we are referring to Dirichlet, Neumann or Robin boundary conditions.
- You need to manually create the field variables dictionaries (e.g. *0/U*, *0/p*, *0/T*, *0/k*, *0/omega*).
- Remember, if you forget to define a numerical boundary condition, OpenFOAM® will complain and will tell you where and what is the error.
- Also, if you misspelled something OpenFOAM® will complain and will tell you where and what is the error.

Boundary conditions and initial conditions

- The following base type and numerical type boundary conditions are constrained or paired.
- That is, the type needs to be same in the *boundary* dictionary and field variables dictionaries (e.g., $0/U$, $0/p$, $0/T$, $0/k$, $0/omega$).

Base type	Numerical type
<i>constant/polyMesh/boundary</i>	$0/U - 0/p - 0/T - 0/k - 0/omega$ (IC/BC)
cyclic cyclicAMI empty processor symmetry symmetryPlane wedge	cyclic cyclicAMI empty processor symmetry symmetryPlane wedge

- These are known as **constraint** patches in OpenFOAM.
- To find a complete list and the source code location of these patches, go to the directory `$WM_PROJECT_DIR` and type in the terminal:
 - `$> find . -type d -iname *constraint*`

Boundary conditions and initial conditions

- The base type **patch** can be any of the boundary conditions available in OpenFOAM®.
- Mathematically speaking; they can be Dirichlet, Neumann or Robin boundary conditions.

Base type	Numerical type
<i>constant/polyMesh/boundary</i>	<i>0/U - 0/p - 0/T - 0/k - 0/omega (IC/BC)</i>
patch	calculated codedFixedValue epsilonWallFunction fixedValue flowRateInletVelocity inletOutlet movingWallVelocity rotatingWallVelocity slip supersonicFreeStream totalPressure zeroGradient ... and so on Refer to the doxygen documentation or the source code for a list of all numerical boundary conditions available.

Boundary conditions and initial conditions

- The **wall** base type boundary condition is defined as follows:

Base type	Numerical type	
<i>constant/polyMesh/boundary</i>	<i>0/U</i>	<i>0/p</i>
wall	type fixedValue; value uniform (0 0 0);	zeroGradient

- This boundary condition is not contained in the patch base type boundary conditions group, because specialize modeling options can be used on this boundary condition.
- An example is turbulence modeling, where turbulence can be generated or dissipated at the walls.

Boundary conditions and initial conditions

- To deal with **backflow** at outlets, you can use the following boundary condition:

Base type	Numerical type	
<i>constant/polyMesh/boundary</i>	$0/U$	$0/p$
patch	type inletOutlet; inletValue uniform (0 0 0); value uniform (0 0 0);	type fixedValue; value uniform 0;

- The **inletValue** keyword is used for the reverse flow.
- In this case, if flow is coming back into the domain, it will use the value set using the keyword **inletValue**. Otherwise, it will use a **zeroGradient** boundary condition.
- For the turbulent variables (**k**, **omega**, **epsilon**, and so on), you can use **inletOutlet** type (pay attention that these quantities are scalars).

Boundary conditions and initial conditions

- Typical boundary conditions are as follows (external aerodynamics),

Boundary type description	Pressure	Velocity	Turbulence fields
Inlet face	zeroGradient	fixedValue	fixedValue
Outlet face	fixedValue	inletOutlet	inletOutlet
Wall face	zeroGradient	fixedValue	Wall functions *
Symmetry face	symmetry	symmetry	symmetry
Periodic face	cyclic	cyclic	cyclic
Empty face (2D)	empty	empty	empty
Slip wall **	slip	slip	slip

* Wall functions can be: **kqWallFunction**, **omegaWallFunction**, **nutkWallFunction**, and so on (next slide).

** The base type can be **wall** or **patch**.

Boundary conditions and initial conditions

- And when dealing with turbulence modeling, these are the most often used wall boundary conditions.
- To use these boundary conditions (wall functions) and to be able to compute y^+ , the primitive patch (the patch type defined in the *boundary* dictionary), must be of type **wall**.
- We will talk more about this when dealing with turbulence modeling.

Field	Wall functions – High RE	Resolved BL – Low RE
nut	nut(-)WallFunction * or nutUSpaldingWallFunction ** (with 0 or a small number)	nutUSpaldingWallFunction **, nutkWallFunction , nutUWallFunction , nutLowReWallFunction or fixedValue *** (with 0 or a small number)
k, q, R	kqRWallFunction ** (with inlet value or a small number) $k_{wall} = k$	kqRWallFunction ** or kLowReWallFunction (with inlet value, 0, or a small number) or fixedValue *** (with 0 or a small number)
epsilon	epsilonWallFunction (with inlet value) $\epsilon_{wall} = \epsilon$	epsilonWallFunction (with inlet value) or zeroGradient *** or fixedValue *** (with 0 or a small number)
omega	omegaWallFunction ** (with a large number) $\omega_{wall} = 10 \frac{6\nu}{\beta y^2} \quad \beta = 0.075$	omegaWallFunction ** or fixedValue *** (both with a large number)
nuTilda	–	fixedValue (one to ten times the molecular viscosity, a small number, or 0)

* nutUWallFunction or nutkWallfunction

** Recommended options for y^+ insensitive treatment (continuous wall functions)

*** Will disable wall functions. The equations will be integrated down to the viscous sublayer with no damping or corrections.

Boundary conditions and initial conditions

- Finally, remember that the name of the base type boundary condition and the name of the numerical type boundary condition needs to be the same, if not, OpenFOAM® will complain.
- Pay attention to this, specially if you are converting the mesh from another format.
- Also, do not use spaces or funny characters when assigning the names to the boundary patches.
- The following names are consistent among all dictionary files,

Base type	Numerical type	
<i>constant/polyMesh/boundary</i>	<i>0/U</i>	<i>0/p</i>
inlet	inlet	inlet
top	top	top
cylinder	cylinder	cylinder
sym	sym	sym

Boundary conditions and initial conditions

- There is a plethora of boundary conditions implemented in OpenFOAM®.
- You can find the source code of the main numerical boundary conditions in the following directory:
 - `$WM_PROJECT_DIR/src/finiteVolume/fields/`
- The wall boundary conditions for the turbulence models (wall functions), are located in the following directory:
 - `$WM_PROJECT_DIR/src/MomentumTransportModels/momentumTransportModels/derivedFvPatchFields/wallFunctions`
- To find all the boundary conditions implemented in OpenFOAM, go to the directory `$WM_PROJECT_DIR` and type in the terminal,
 - `$> find . -type d -iname *fvPatch*`
 - `$> find . -type d -iname *derivedFv*`
 - `$> find . -type d -iname *pointPatch*`
- To get more information about all the boundary conditions available in OpenFOAM® you can read the Doxygen documentation.
- You can access the documentation online at this link <http://cpp.openfoam.org/v9/>
- To know more about a particular boundary condition, you can use the command `foamInfo`

Boundary conditions and initial conditions

 The *constant/polyMesh/boundary* dictionary

- For a generic case, the file *boundary* is divided as follows

3

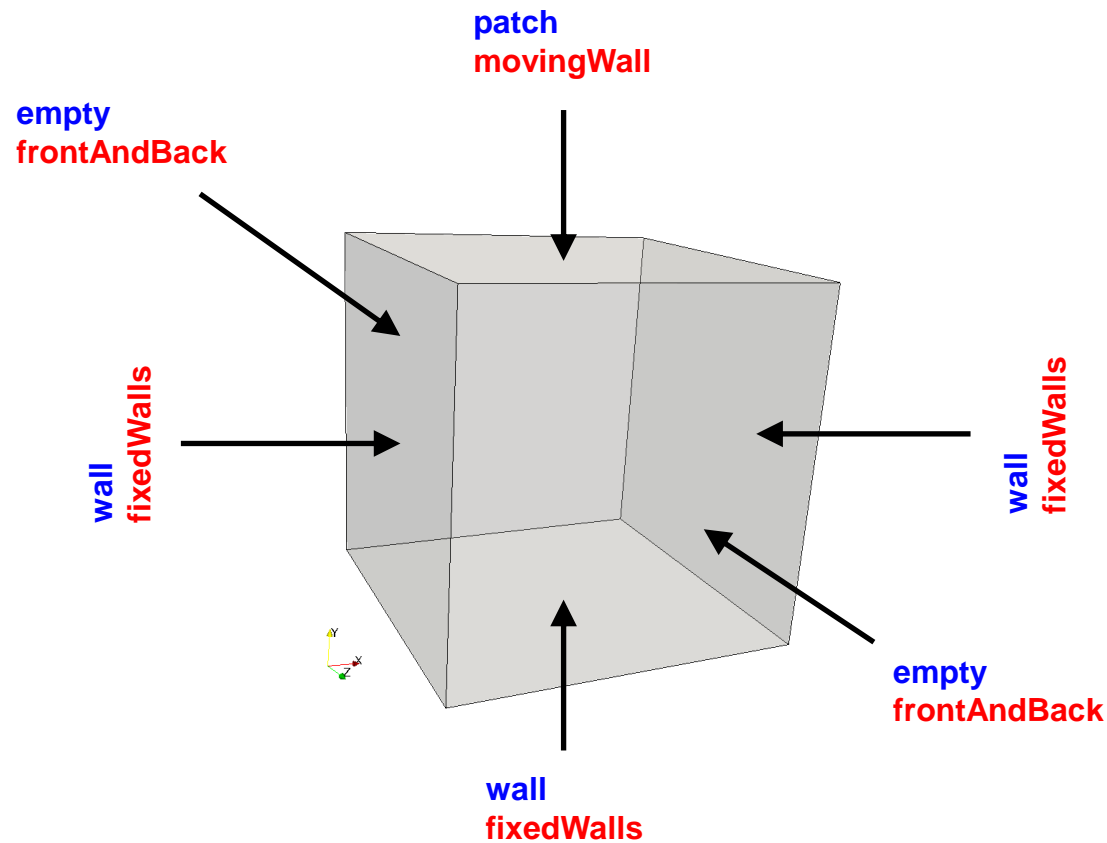
(

```
movingWall
{
    type            patch;
    nFaces          20;
    startFace       760;
}
```

```
fixedWalls
{
    type            wall;
    nFaces          60;
    startFace       780;
}
```

```
frontAndBack
{
    type            empty;
    nFaces          800;
    startFace       840;
}
```

)



Boundary conditions and initial conditions

The *constant/polyMesh/boundary* dictionary

- For a generic case, the file *boundary* is divided as follows

```
3 (
    movingWall {
        type      patch;
        nFaces    20;
        startFace 760;
    }
    fixedWalls {
        type      wall;
        nFaces    60;
        startFace 780;
    }
    frontAndBack {
        type      empty;
        nFaces    800;
        startFace 840;
    }
)
```

Number of surface patches

There must be 3 patches definition.

Name and type of the surface patches

- The name and type of the patch is given by the user.
- You can change the name if you do not like it. Do not use strange symbols or white spaces.
- You can also change the **base type**. For instance, you can change the type of the patch **movingWall** from **patch** to **wall**.
- When converting the mesh from a third-party format, OpenFOAM® will try to recover the information from the original format. But it might happen that it does not recognize the base type and name of the original format. If that is your case, you will need to modify the file manually or using any of the mesh manipulation utilities distributed with OpenFOAM®.

nFaces and startFace keywords

- Unless you know what are you doing, **you do not need to change this information**.



Boundary conditions and initial conditions



The O/U dictionary

- For a generic case, the **numerical** type BC are assigned as follows (**U**),

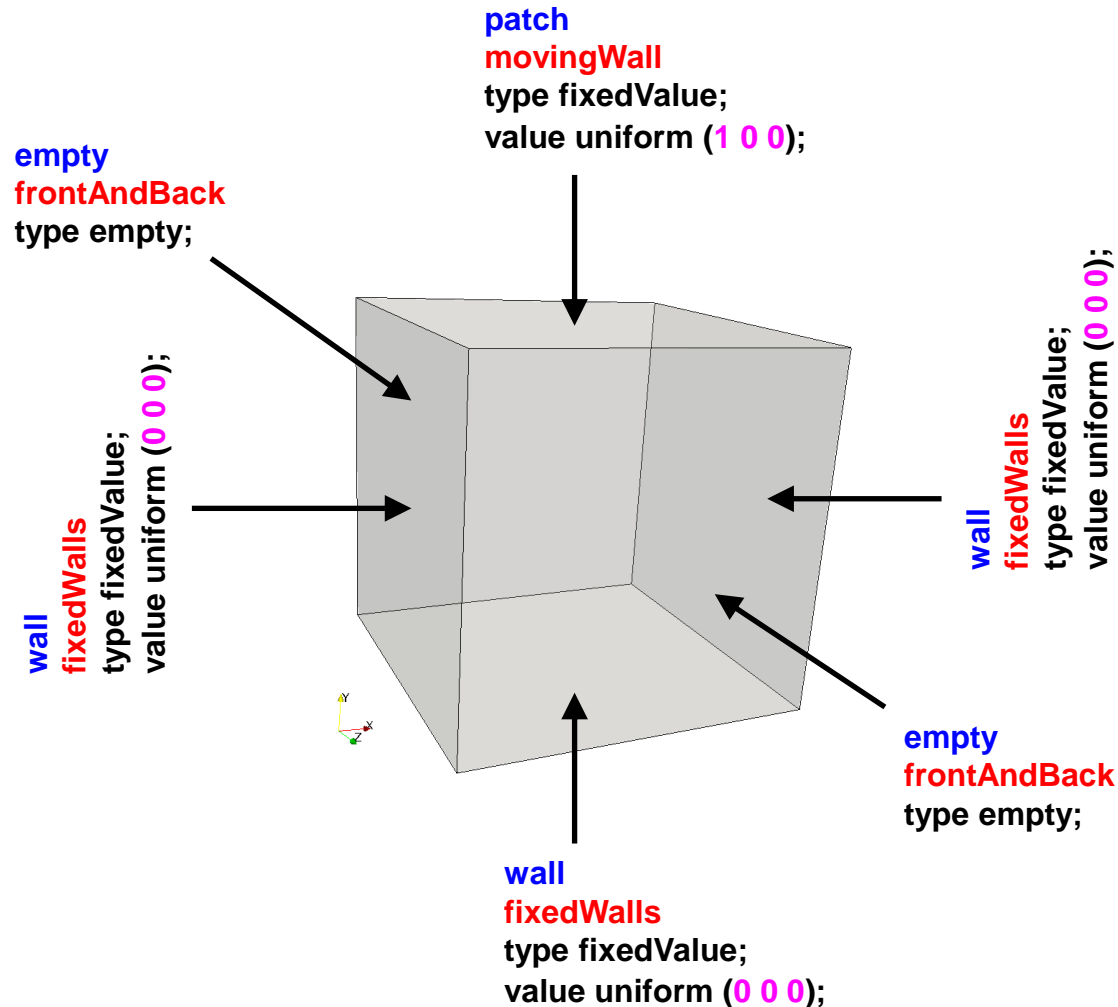
```
dimensions      [0 1 -1 0 0 0 0];

internalField    uniform (0 0 0);

boundaryField
{
    movingWall
    {
        type      fixedValue;
        value      uniform (1 0 0);
    }

    fixedWalls
    {
        type      fixedValue;
        value      uniform (0 0 0);
    }

    frontAndBack
    {
        type      empty;
    }
}
```



Boundary conditions and initial conditions

The $0/p$ dictionary

- For a generic case, the **numerical** type BC are assigned as follows (p),

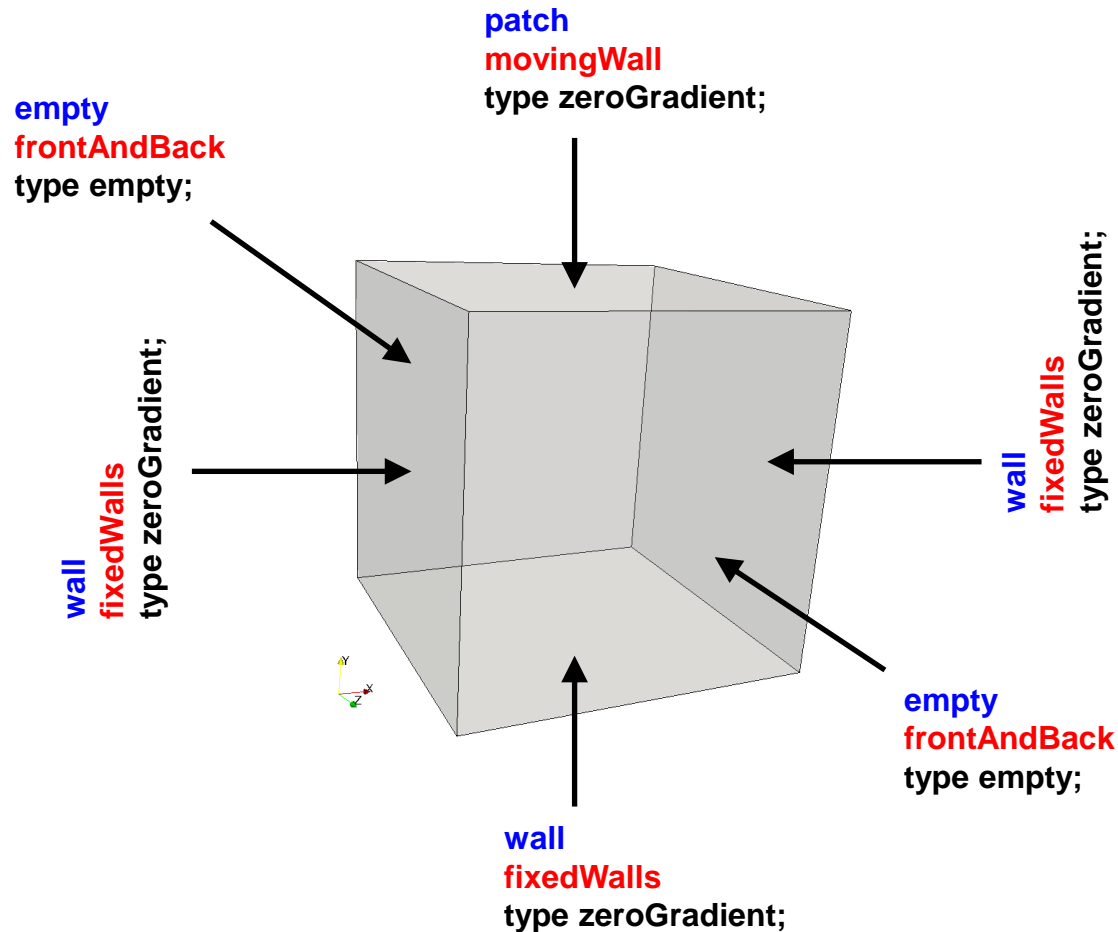
```
dimensions      [0 2 -2 0 0 0 0];

internalField    uniform 0;

boundaryField
{
    movingWall
    {
        type      zeroGradient;
    }

    fixedWalls
    {
        type      zeroGradient;
    }

    frontAndBack
    {
        type      empty;
    }
}
```



Roadmap

- ~~1. Finite Volume Method: A Crash Introduction~~
- ~~2. On the CFL number~~
- ~~3. Linear solvers in OpenFOAM®~~
- ~~4. Pressure-Velocity coupling in OpenFOAM®~~
- ~~5. Unsteady and steady simulations~~
- ~~6. Understanding residuals~~
- ~~7. Boundary and initial conditions~~
- 8. Numerical playground**

Numerical playground

Merry-go-round:

Pure convection of a passive scalar in a vector field – One dimensional tube.

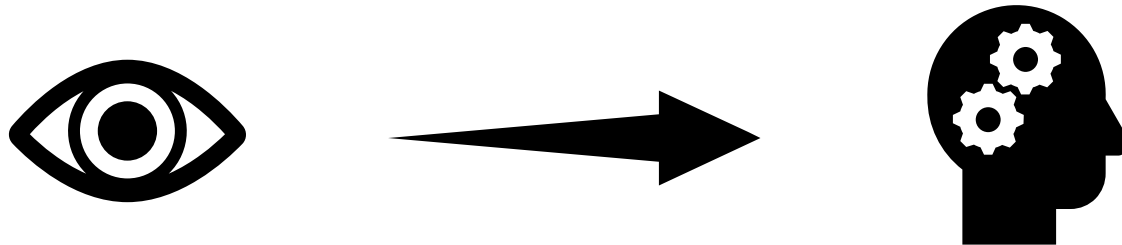


Numerical playground

- This is a visual and mental exercise only.
- You will find this case in the directory

`$PTOFC/101FVM/pureConvection/orthogonal_1d`

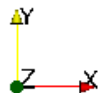
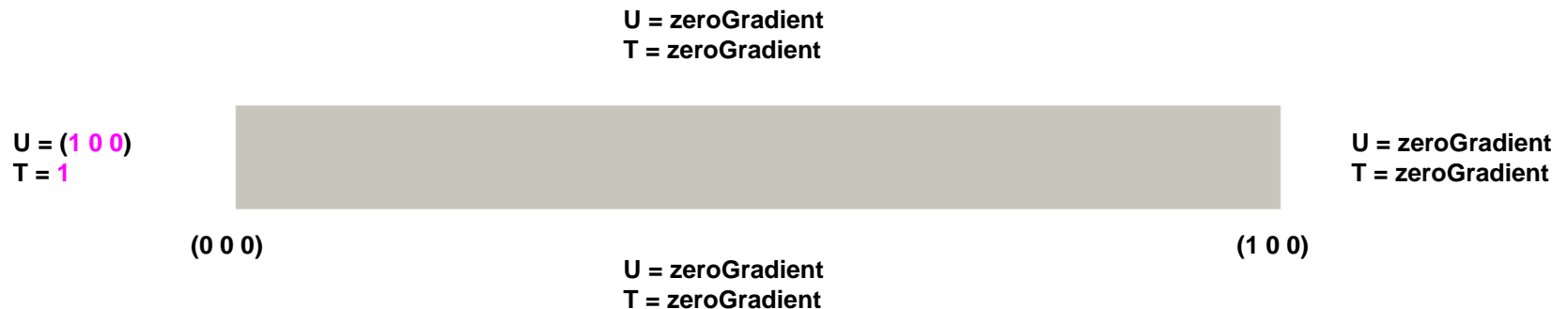
- In this directory, you will also find the *README.FIRST* file with the instructions of how to run the case.
- Hereafter, we will focus our eyes to train our brain.



Numerical playground

Pure convection of a scalar in a vector field – One dimensional tube.

$$\frac{\partial T}{\partial t} + \nabla \cdot (\phi T) - \cancel{\nabla \cdot (\Gamma \nabla T)} = 0$$



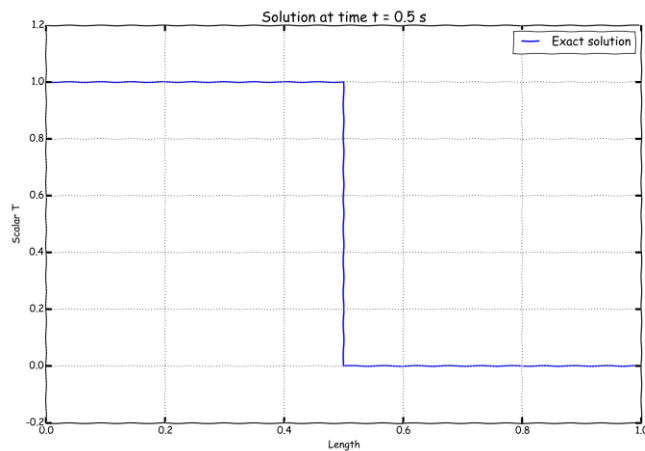
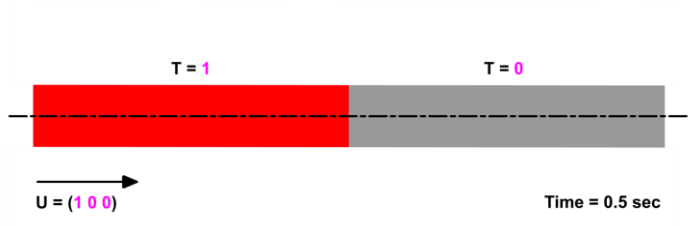
Initial conditions

$$U = (1 \ 0 \ 0)$$

$$T = 0$$

Numerical playground

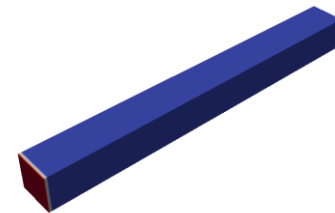
- This problem has an exact solution in the form of a traveling wave.
- We will use this case to study the different discretization schemes implemented in OpenFOAM®.
- In the figure, we show the solution for time = 0.5 s



www.wolfdynamics.com/wiki/pureconvection/xani1.gif

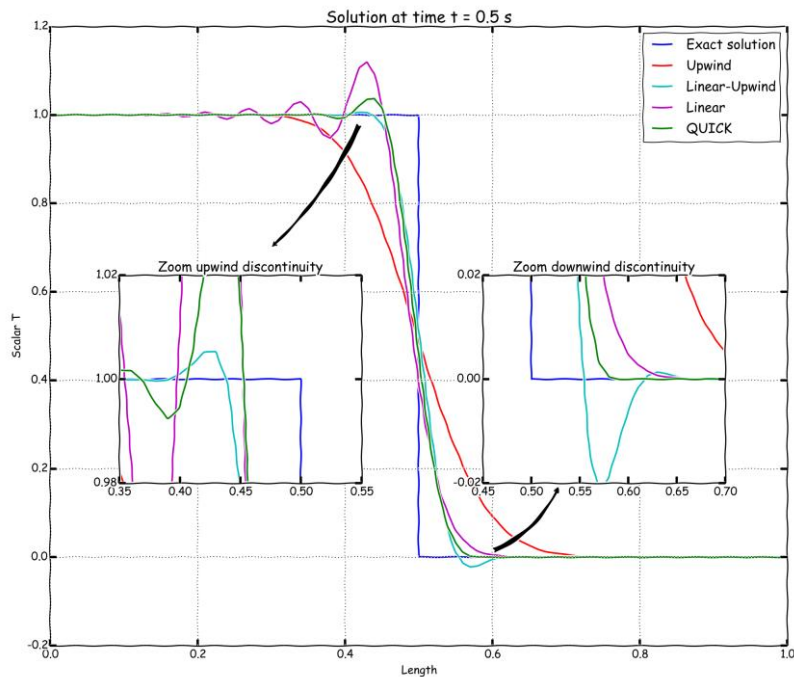


www.wolfdynamics.com/wiki/pureconvection/xani2.gif

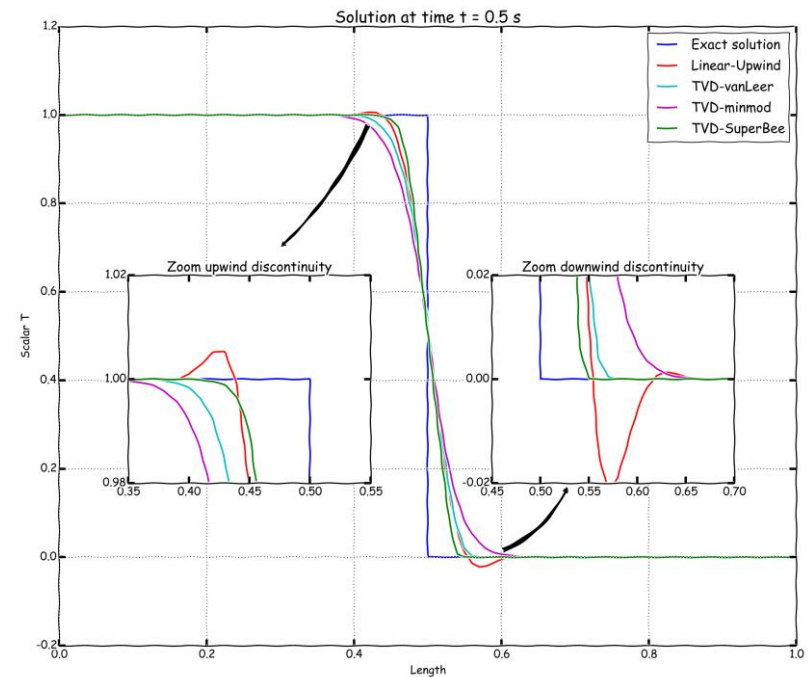


Numerical playground

Comparison of different spatial discretization schemes.
Euler in time – 100 cells – CFL = 0.1
Linear limiter functions on the Sweby diagram.

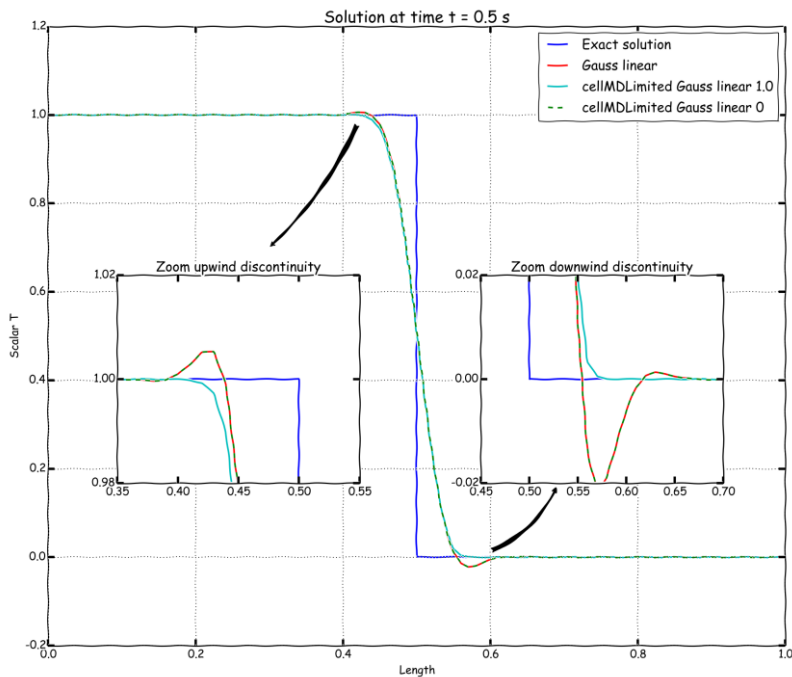


Comparison of different spatial discretization schemes.
Euler in time – 100 cells – CFL = 0.1
Non-linear limiter functions on the Sweby diagram.

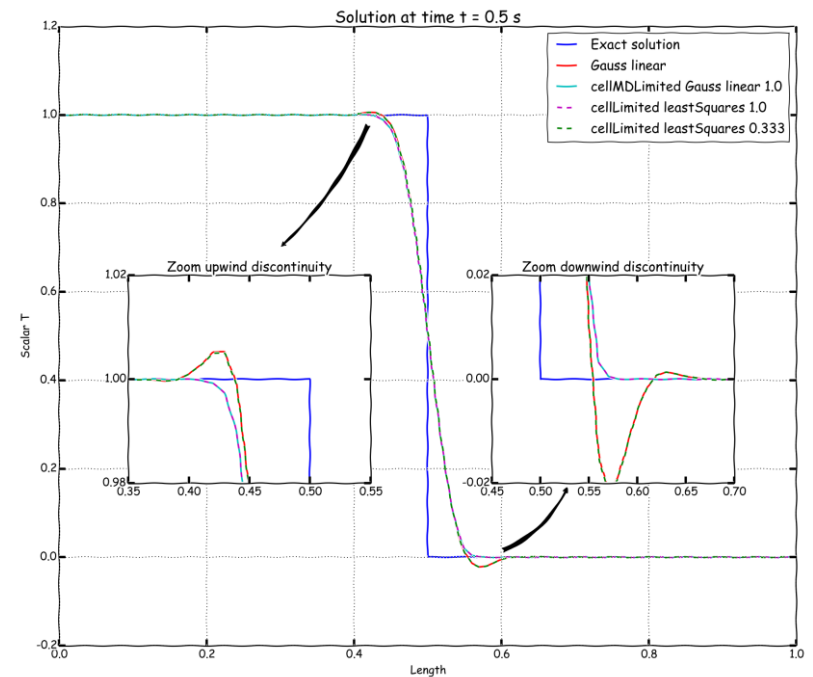


Numerical playground

Comparison of different gradient limiters.
Linear upwind in space – Euler in time – 100 cells –
CFL 0.1



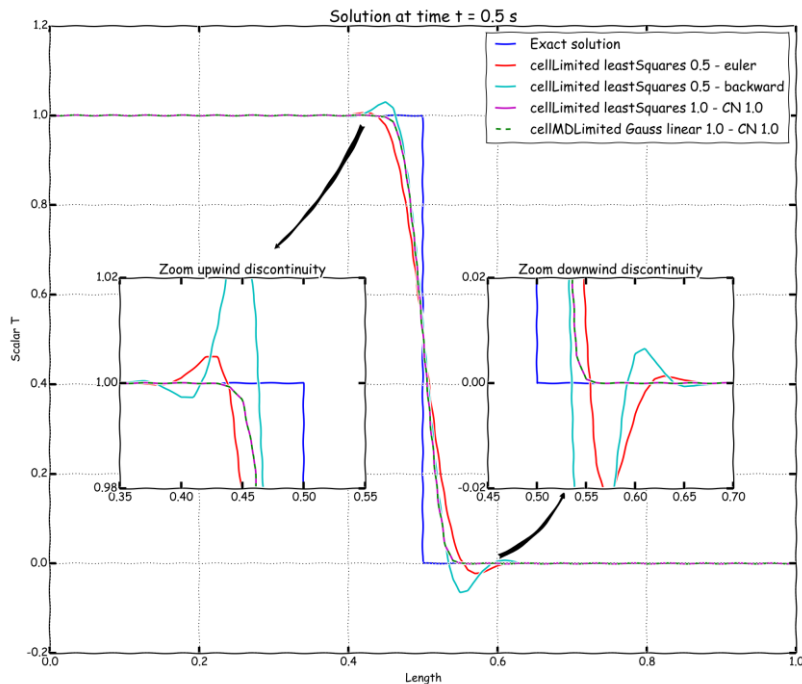
Comparison of different gradient limiters.
Linear upwind in space – Euler in time – 100 cells –
CFL 0.1



Numerical playground

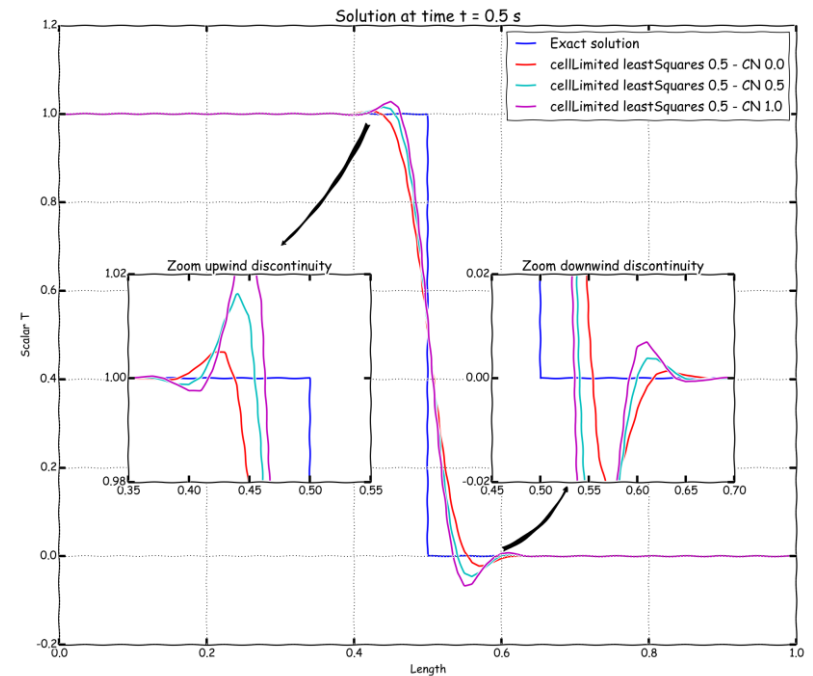
Comparison of different time discretization schemes and gradient limiters.

Linear upwind in space – 100 cells – CFL 0.1



Comparison of Crank Nicolson blending factor using cellLimited leastSquares 0.5 gradient limiter.

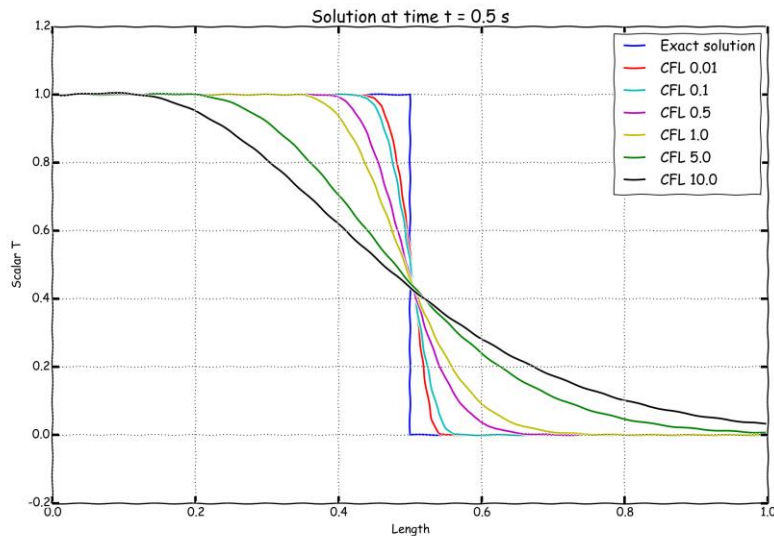
Linear upwind in space – 100 cells – CFL 0.1



Numerical playground

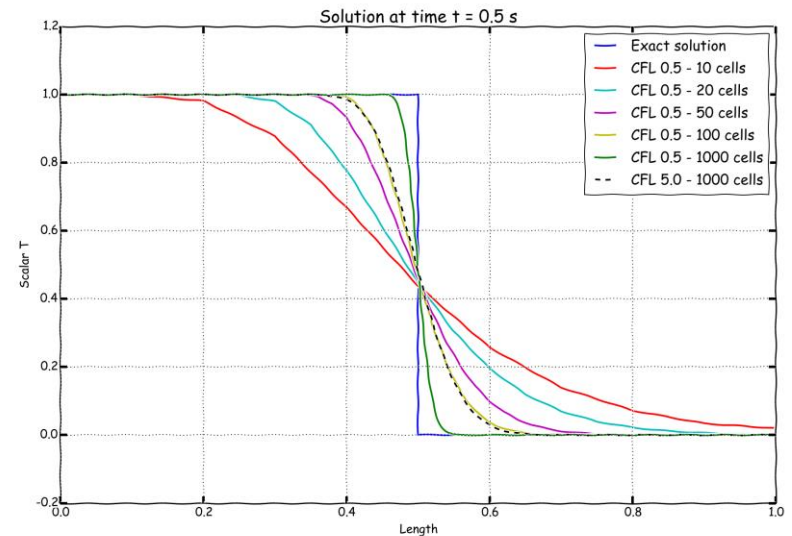
Comparison of different time-step size (different CFL number).

Linear upwind in space – Euler in time – 100 cells



$$CFL = \frac{u\Delta t}{\Delta x}$$

Comparison of different mesh sizes.
Linear upwind in space – Euler in time



Numerical playground

- This case was for your eyes and brain only, but we encourage you to reproduce all the previous results,
 - Use all the time discretization schemes.
 - Use all the spatial discretization schemes.
 - Use all the gradient discretization schemes.
 - Use gradient limiters.
 - Use different mesh resolution.
 - Use different time-steps.
- Sample the solution and compare the results.
- Try to find the best combination of numerical schemes.
- Remember, in the *README.FIRST* file you will find the instructions of how to run the case.



Numerical playground

Exercises

- Which one of the following schemes is useless: **upwind**, **downwind**, or **linear**
- Compare the solution obtained with the following schemes: **upwind**, **linearUpwind**, **MUSCL**, **QUICK**, **cubic**, **UMIST**, **OSHER**, **Minmod**, **vanAlbada**. Are all of them bounded? Are they second order accurate?
- Use the **linearUpwind** method with **Gauss linear**, **Gauss pointLinear** and **leastSquares** for gradient computations, which method is more accurate?
- Imagine that you are using the **linearUpwind** method with no gradient limiters. How will you stabilize the solution if it becomes unbounded?
- When using gradient limiters, what is clipping?
- Use the **linearUpwind** with different gradient limiters. Which method is more unbounded?
- Use the **vanLeer** method with a CFL number of 0.1, 0.9 and 2, did all solutions converge? Are both solutions bounded?
- In the directory **tri_mesh**, you will find the same case setup using a triangular mesh.
 - Run the case and compare the solution with the equivalent setup using the orthogonal mesh.
 - Repeat the same experiments as before and draw your conclusions about which method is better for unstructured meshes.
 - With unstructured meshes, is it possible to get the same accuracy level as for orthogonal meshes?

Numerical playground

Exercises

- The solver `scalarTransportFoam` does not report the CFL number on the screen. How will you compute the CFL number in this case?

(Hint: you can take a look at the post-processing slides or the utilities directory)

- Which one is more diffusive, spatial discretization or time discretization?
- Are all time discretization schemes bounded?
- If you are using the Crank-Nicolson scheme, how will you avoid oscillations?
- Does the solution improve if you reduce the time-step?
- Use the **upwind** scheme and a really fine mesh. Does the accuracy of the solution improve?
- From a numerical point of view, what is the Peclet number? Can it be compared to the Reynolds number?

$$Pe = \frac{LU}{D} = \frac{\text{convection effects}}{\text{diffusion effects}}$$

- If the Peclet number is more than 2, what will happen with your solution if you were using a **linear** scheme?

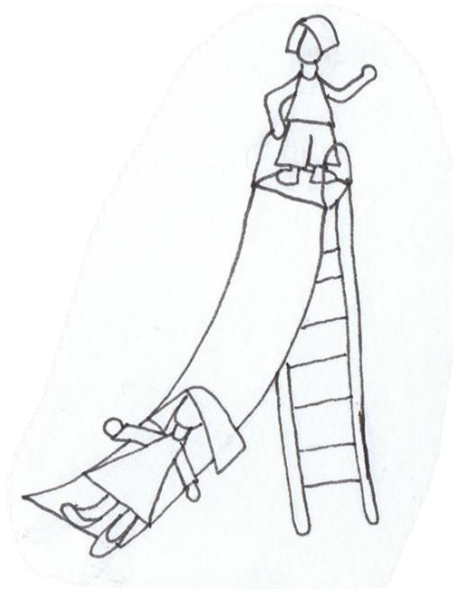
(Hint: to change the Peclet number you will need to change the diffusion coefficient)

- Pure convection problems have analytical solutions. You are asked to design your own tutorial with an analytical solution in 2D or 3D.
- Try to break the solver using a time step less than 0.005 seconds. You are allowed to modify the original mesh and use any combination of discretization schemes.

Numerical playground

Slide:

2D Laplace equation in a square domain.

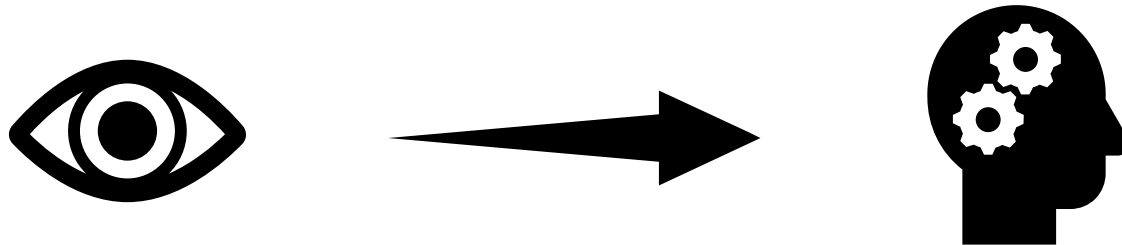


Numerical playground

- This is a visual and mental exercise only.
- You will find this case in the directory

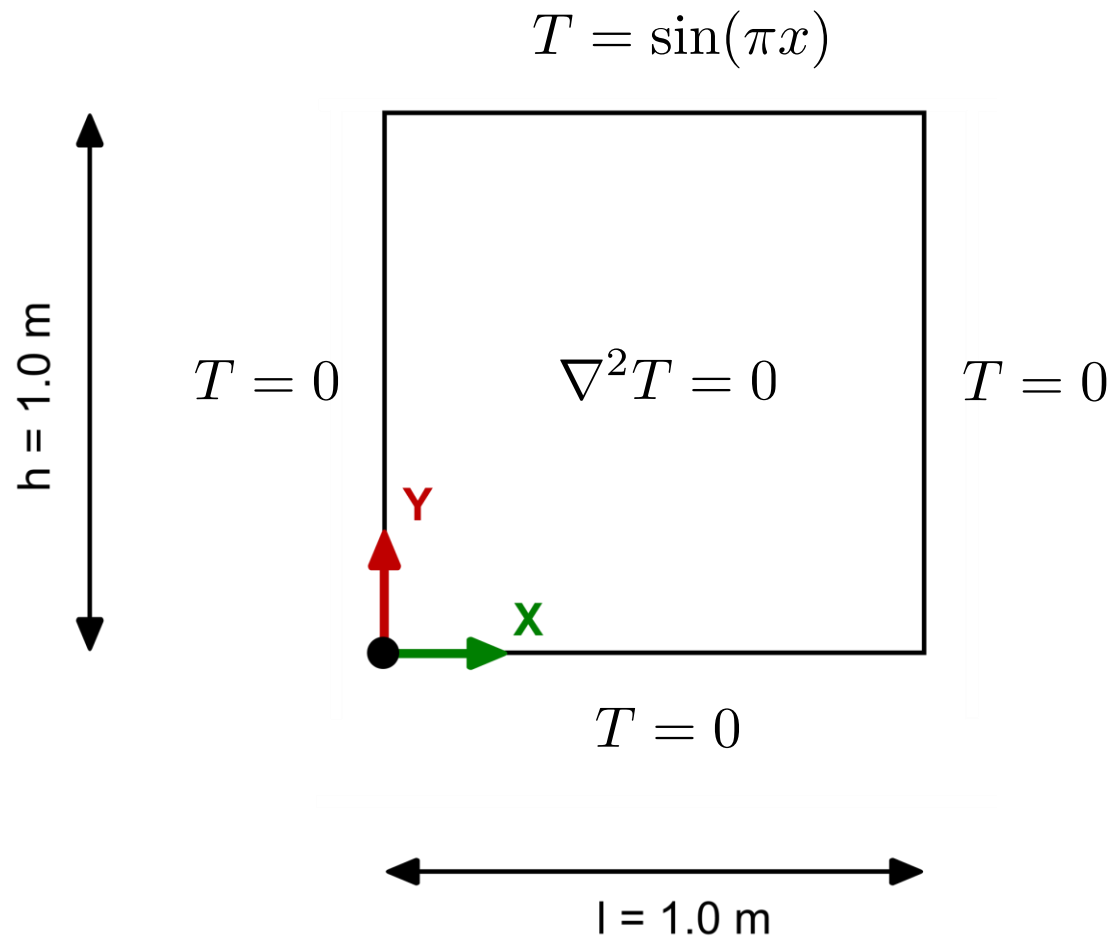
`$PTOFC/101FVM/laplace`

- In this directory, you will also find the *README.FIRST* file with the instructions of how to run the case.
- Hereafter, we will focus our eyes to train our brain.



Numerical playground

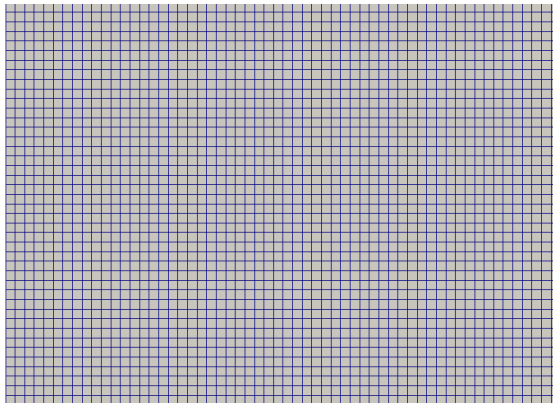
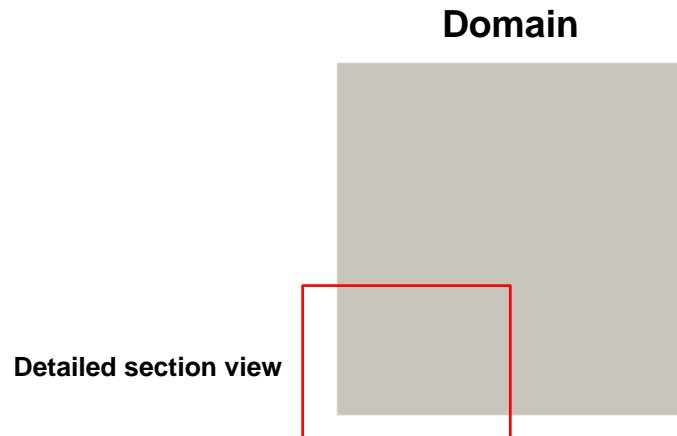
2D Laplace equation in a square domain



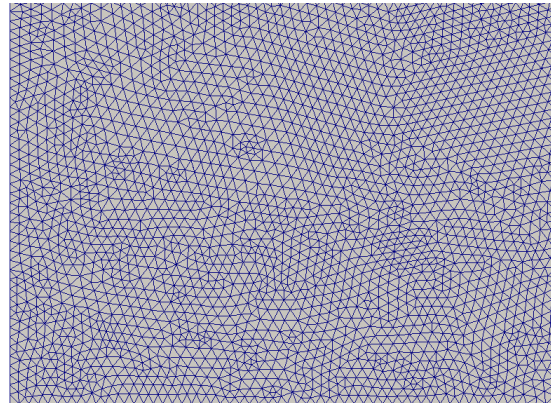
Numerical playground

2D Laplace equation in a square domain

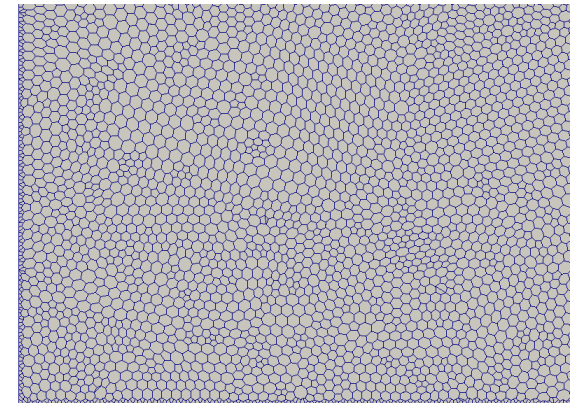
- This case consist of one domain and three different element types.



Hexahedral mesh



Triangular mesh

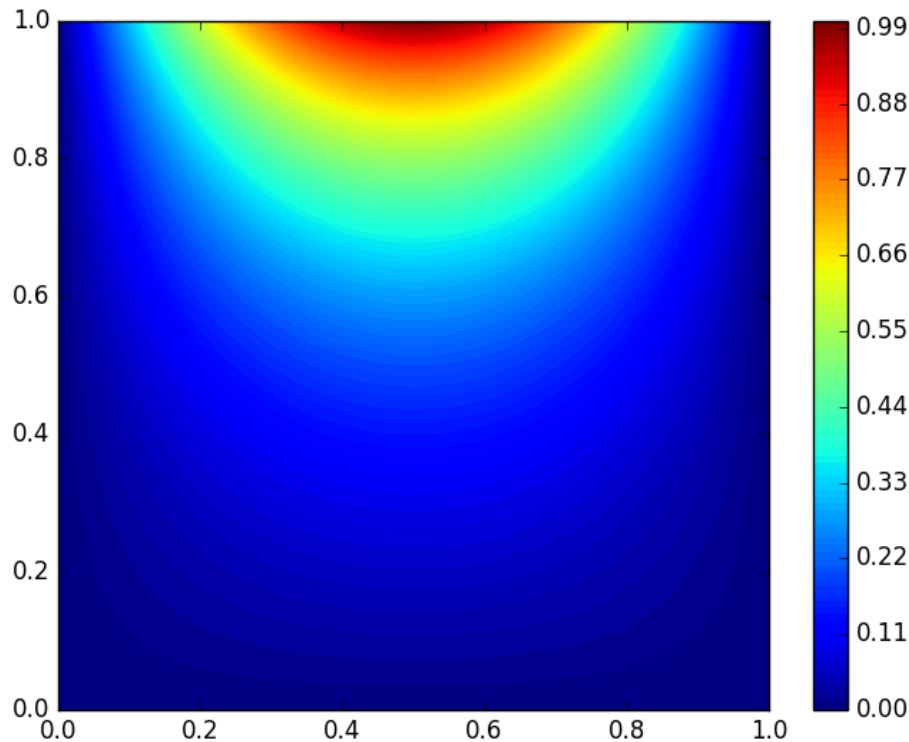


Polyhedral mesh

Numerical playground

2D Laplace equation in a square domain

- We will study the influence of the element type on the gradients computation.
- We will also study the influence of the **gradSchemes** method and **laplacianSchemes** method on the solution.

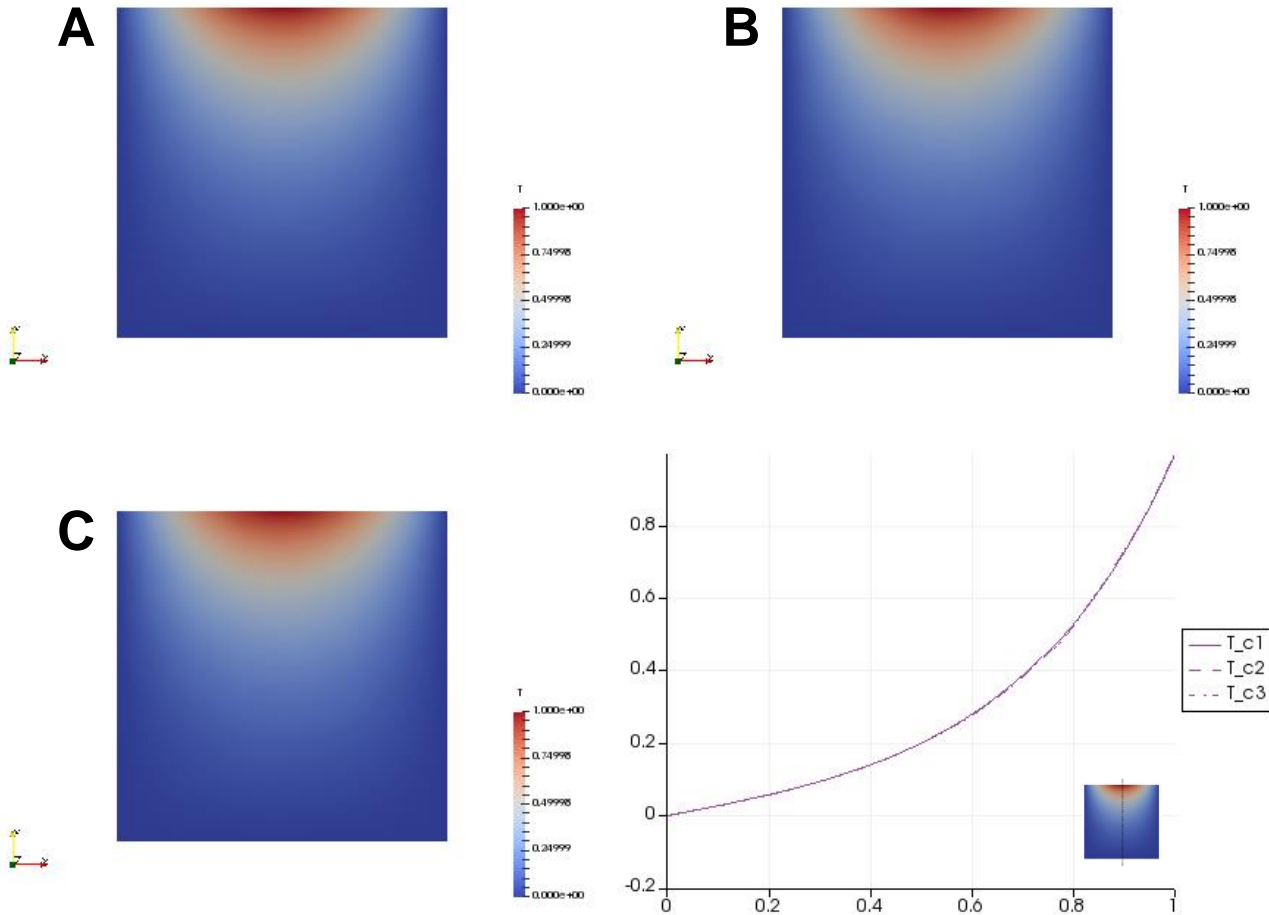


This problem has the following analytical solution:

$$T(x, y) = \frac{\sin(\pi x) \times \sinh(\pi y)}{\sinh(\pi)}$$

Numerical playground

2D Laplace equation in a square domain



gradSchemes:
Gauss linear

laplacianSchemes:
Gauss linear orthogonal

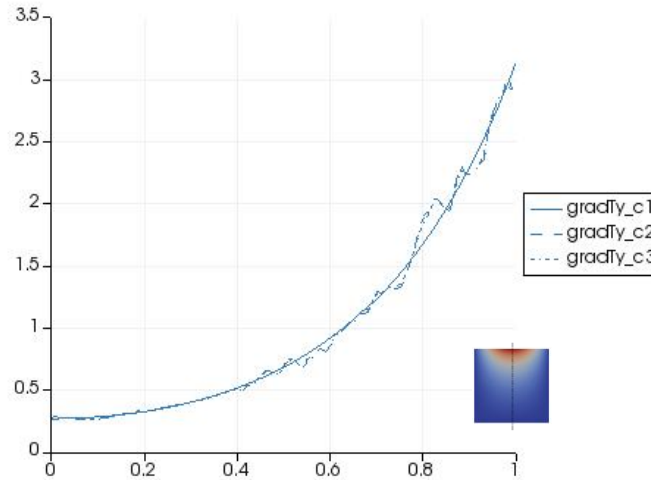
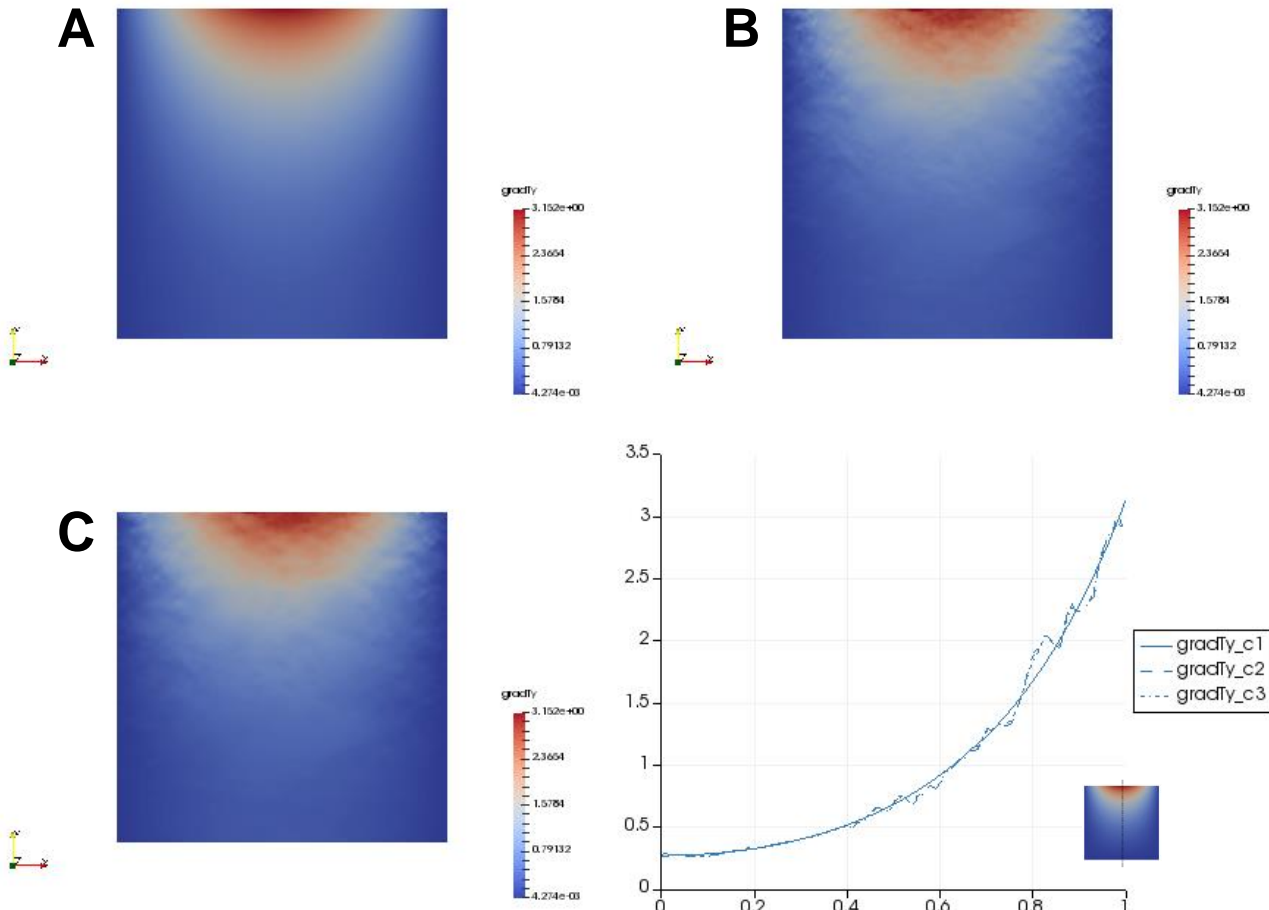
A. Hexahedral mesh
B. Triangular mesh
C. Polyhedral mesh

- This is the actual solution.
- Each mesh gives basically the same solution.
- However, when we look at the information behind the field T, we will see a different outcome.
- Precisely, we will take a look at the gradients.

T field

Numerical playground

2D Laplace equation in a square domain



gradSchemes:
Gauss linear

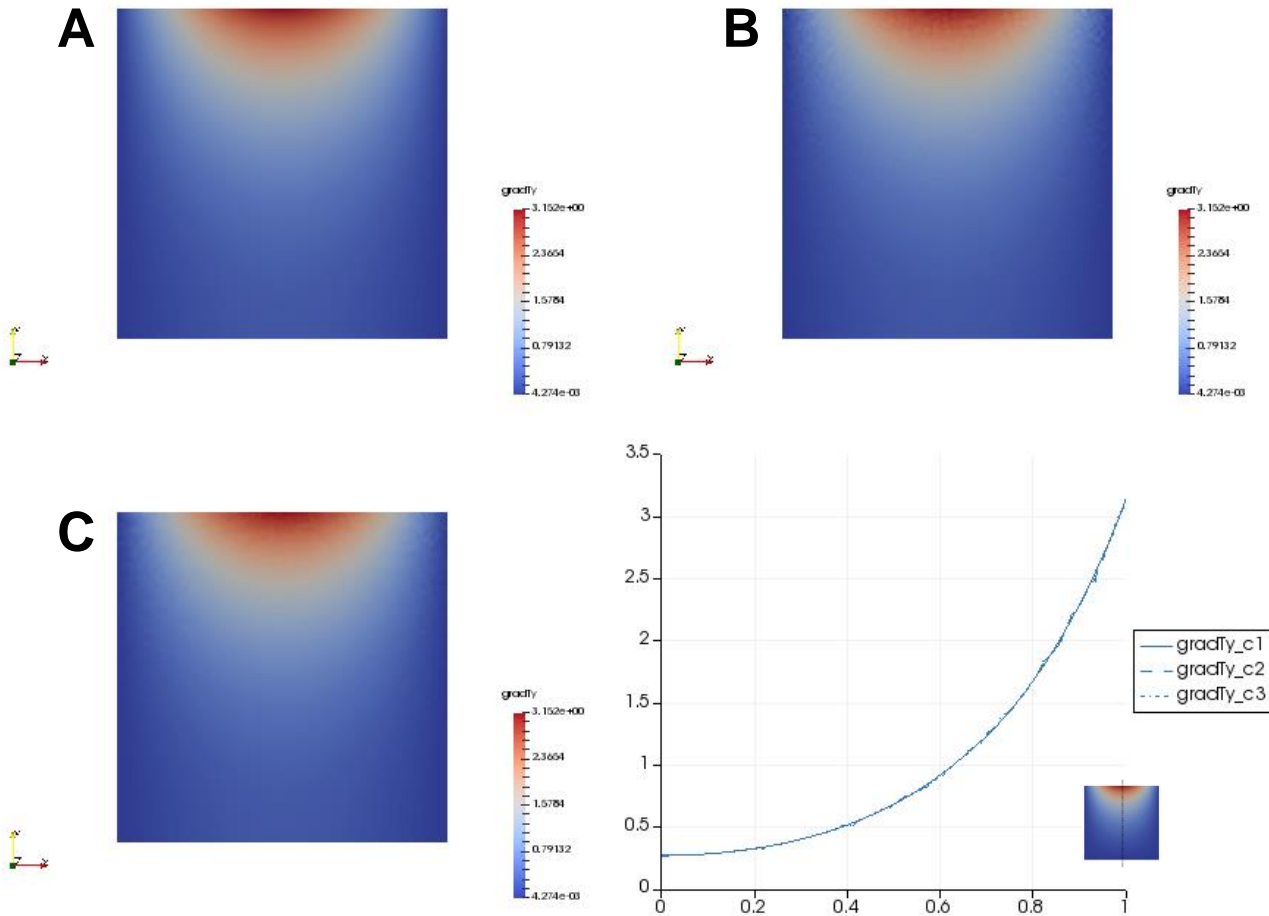
laplacianSchemes:
Gauss linear orthogonal

A. Hexahedral mesh
B. Triangular mesh
C. Polyhedral mesh

- This is not the actual solution. This is the gradient of the field T used to compute the solution.
- The outcome is different for each mesh.
- Behind doors, the gradients need to be computed accurately.
- For the method used in this case, the gradients on the unstructured meshes are noisy.

Numerical playground

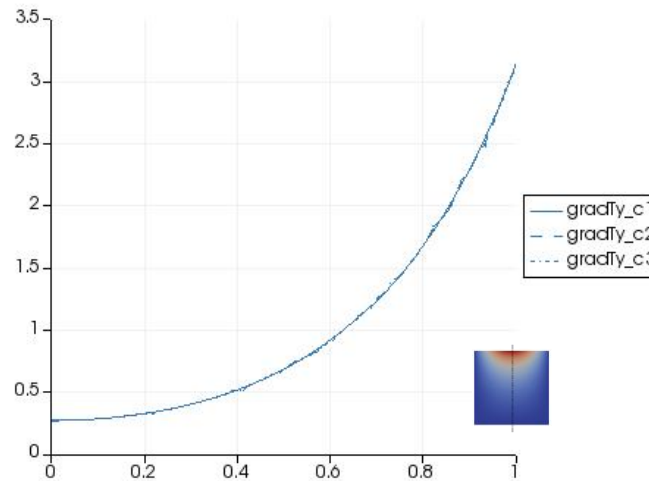
2D Laplace equation in a square domain



gradSchemes:
Gauss linear

laplacianSchemes:
Gauss linear limited 1

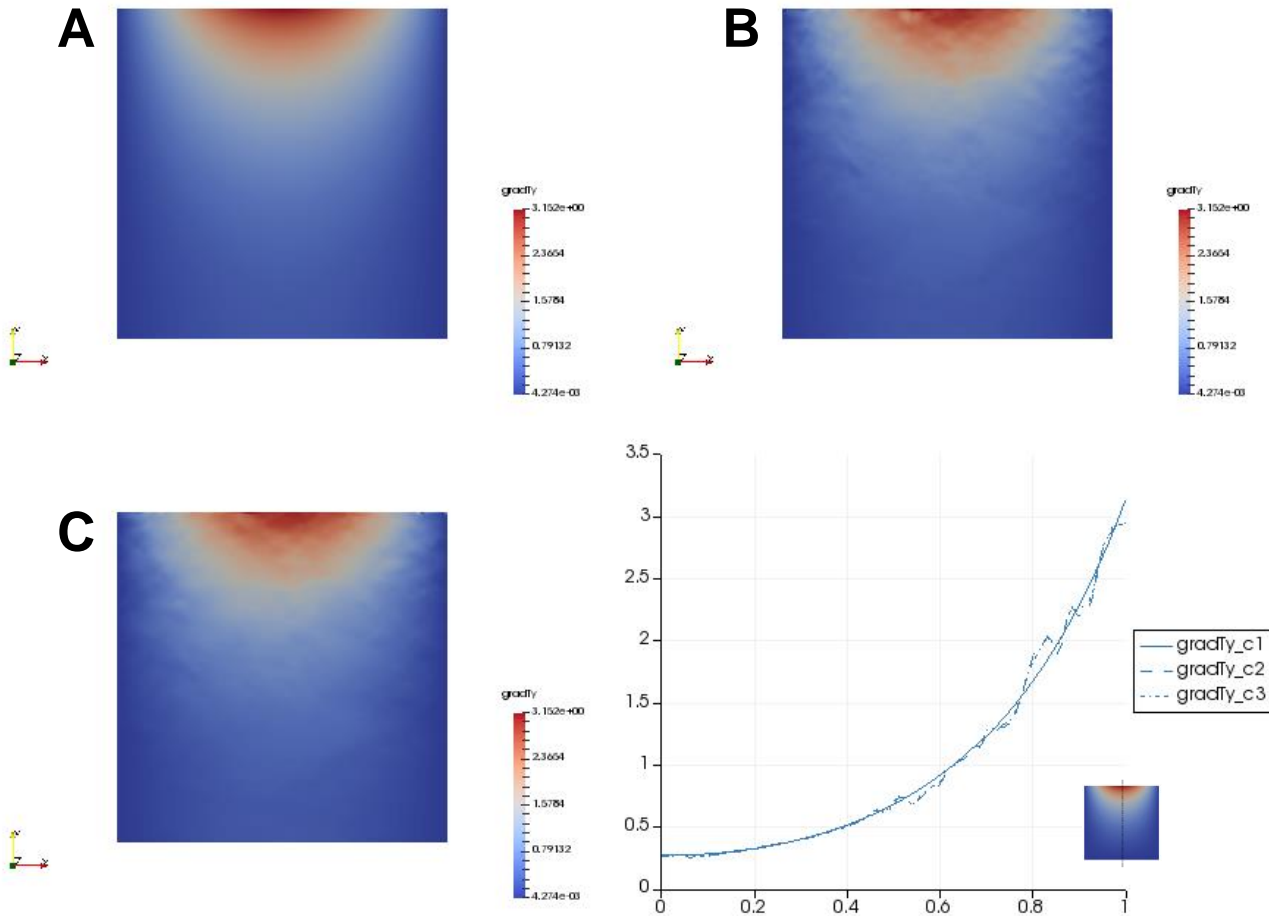
A. Hexahedral mesh
B. Triangular mesh
C. Polyhedral mesh



- This is not the actual solution. This is the gradient of the field T used to compute the solution.
- The outcome is different for each mesh.
- Behind doors, the gradients need to be computed accurately.
- By adjusting the numerics, we can smooth the gradients.
- All meshes show similar gradients.

Numerical playground

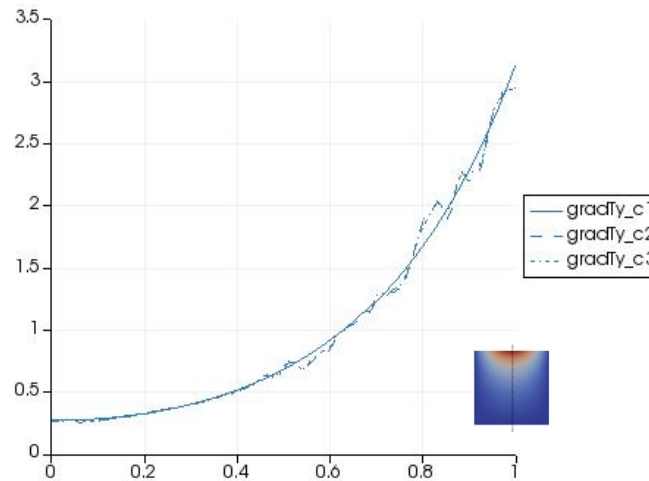
2D Laplace equation in a square domain



gradSchemes:
Gauss leastSquares

laplacianSchemes:
Gauss linear orthogonal

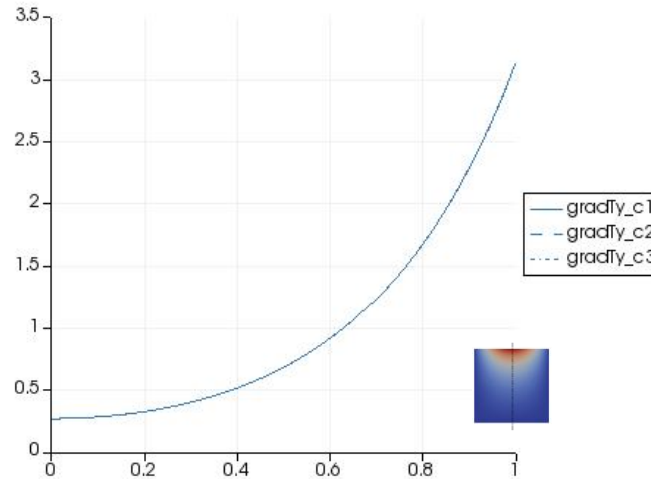
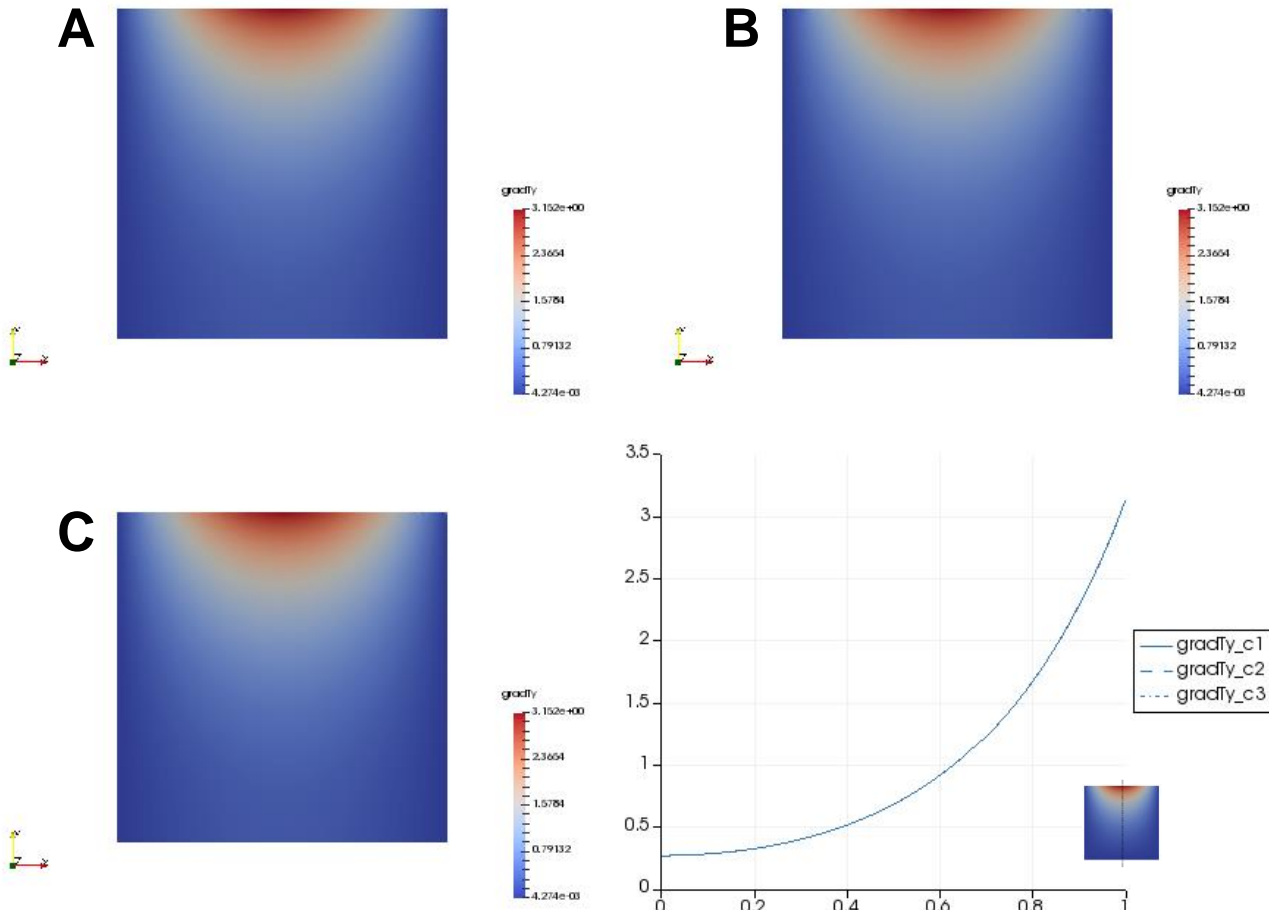
A. Hexahedral mesh
B. Triangular mesh
C. Polyhedral mesh



- This is not the actual solution. This is the gradient of the field T used to compute the solution.
- The outcome is different for each mesh.
- Behind doors, the gradients need to be computed accurately.
- For the method used in this case, the gradients on the unstructured meshes are noisy.

Numerical playground

2D Laplace equation in a square domain



gradSchemes:
Gauss leastSquares

laplacianSchemes:
Gauss linear limited 1

A. Hexahedral mesh
B. Triangular mesh
C. Polyhedral mesh

- This is not the actual solution. This is the gradient of the field T used to compute the solution.
- The outcome is different for each mesh.
- Behind doors, the gradients need to be computed accurately.
- By adjusting the numerics, we can smooth the gradients.
- All meshes show similar gradients.

Numerical playground

- This case was for your eyes and brain only, but we encourage you to reproduce all the previous results.
- In the subdirectory `c1` you will find the hexahedral mesh, in the subdirectory `c2` you will find the triangular mesh, and in the subdirectory `c3` you will find the polyhedral mesh.
- Use the script `runallcases.sh` to run all the cases automatically.
- When launching `paraFoam` it will give you a warning, accept the default option (yes).
- In `paraFoam`, go to the **File** menu and select **Load State**. Load the state located in the directory **paraview** (`state1.pvsm`).
- In the window that pops out, give the location of the `*.foam` files inside each subdirectory (`c1/c1.foam`, `c2/c2.foam`, and `c3/c3.foam`).
- The file `state1.pvsm` will load a preconfigured state with all the solutions.
- If you are interested in running the cases individually, enter the subdirectory and follow the instructions in the `README.FIRST` file.



Numerical playground

Exercises

- Run the case using all gradient discretization schemes available. Which scheme gives the best results?
- According to the previous results, which element type is the best one? Do you think that the choice of the element type is problem dependent (e.g., direction of the flow)?
- Use the **leastSquares** method for gradient discretization, and the **corrected** and **uncorrected** method for Laplacian discretization. Do you get the same results in all the meshes? How can you improve the results?
(Hint: look at the corrections)
- Does it make sense to do more non-orthogonal corrections using the **uncorrected** method?
- Run a case only 1 iteration. Do you get a converged solution? Is there a difference between 1 and 100 iterations? Compare the solutions.
- Use a different interpolation method for the diffusion coefficient. Do you get the same results?
- Try to break the solver (this is a difficult task in this case). You are allowed to modify the original mesh and use any combination of discretization schemes.

Numerical playground

Swing:

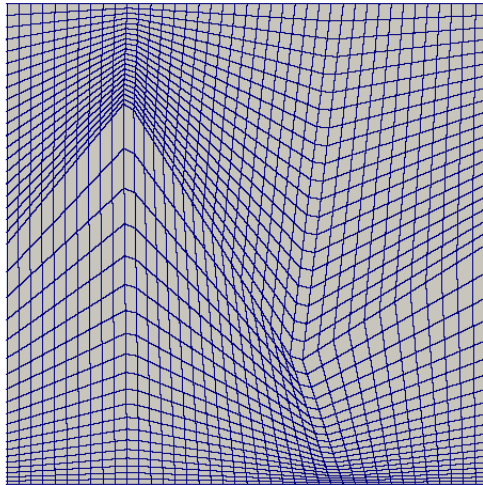
Flow in a lid-driven square cavity – $Re = 100$

Effect of grading and non-orthogonality on the accuracy of the solution



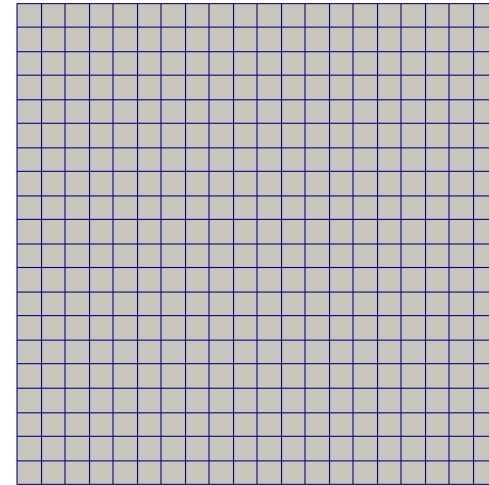
Numerical playground

Flow in a lid-driven square cavity – $Re = 100$ Non-orthogonal mesh vs. orthogonal mesh



Non-orthogonal mesh

The overall quality of this mesh is good (in terms of non-orthogonality and skewness),
but by no standard this is a good mesh.



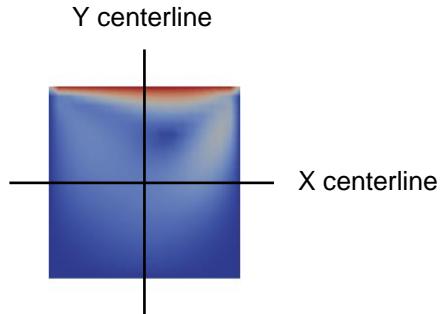
Orthogonal mesh

This is a perfect mesh

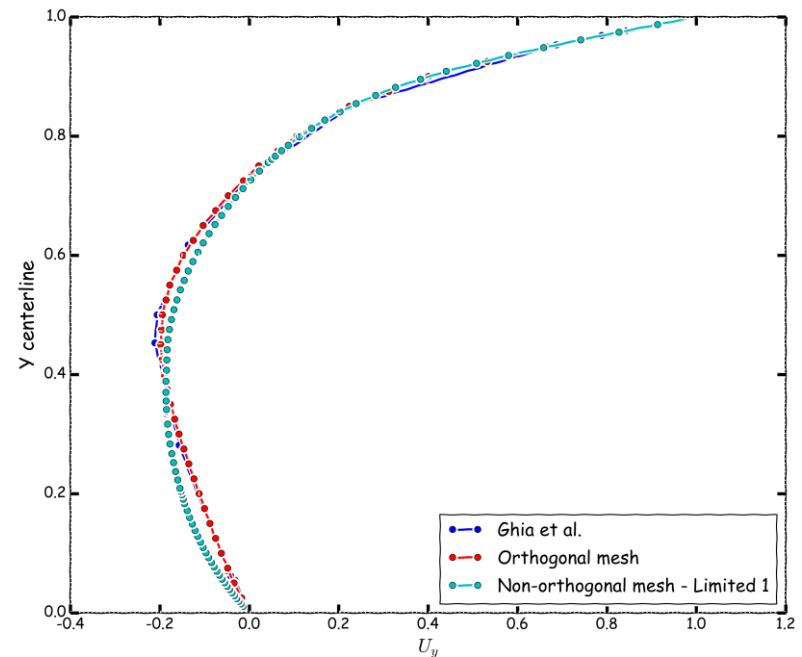
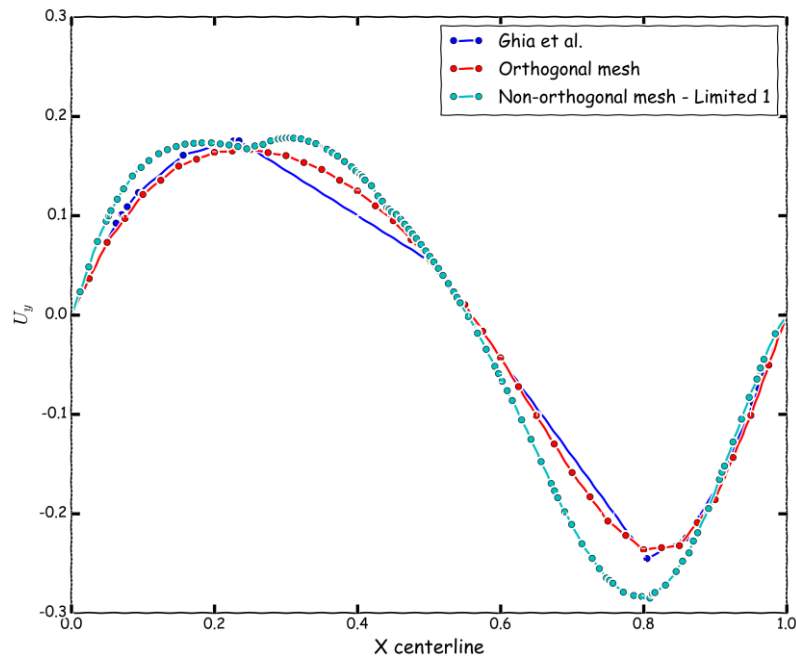
- Often people refer to these non-orthogonal meshes as Kershaw distorted meshes.
- We will use this case to learn how to adjust the numerical schemes according to mesh non-orthogonality and grading.

Numerical playground

LaplacianSchemes orthogonal – Non-orthogonal corrections disabled

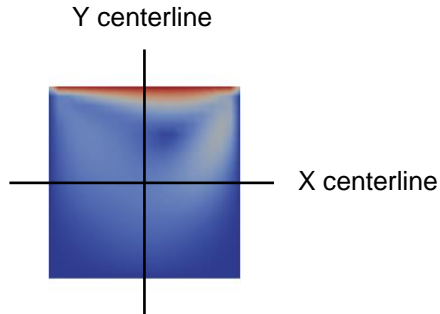


- And as CFD is not only about pretty colors, we should also validate the results

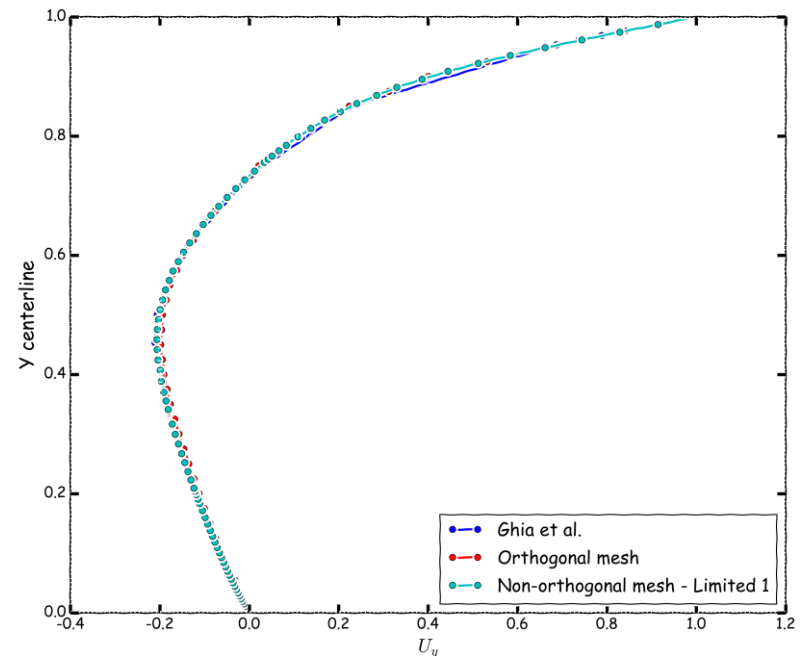
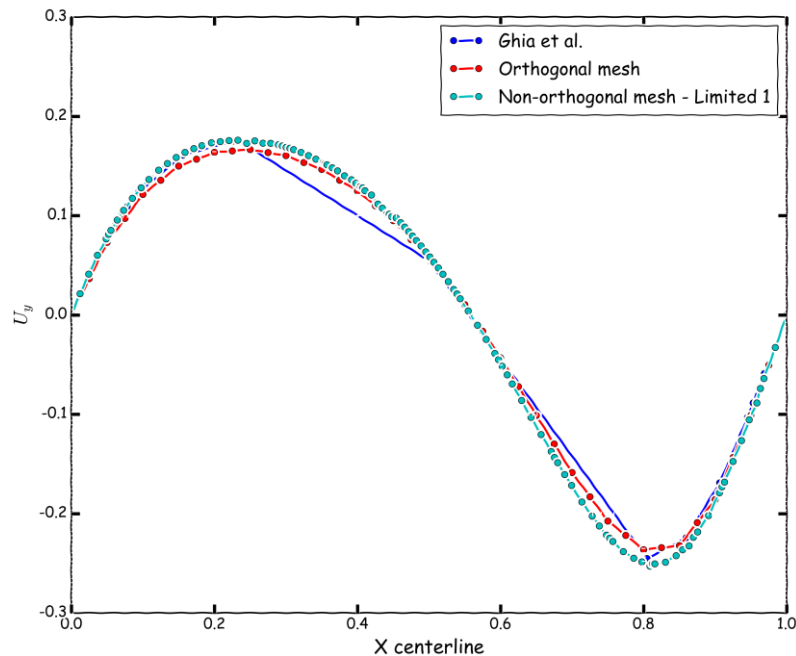


Numerical playground

Laplacian Schemes orthogonal – Non-orthogonal corrections enabled



- And as CFD is not only about pretty colors, we should also validate the results



Numerical playground

How to adjust the numerical method to deal with non-orthogonality

```
ddtSchemes
{
    default            backward;
}

gradSchemes
{
    default            Gauss linear;
    //default          Gauss skewCorrected linear;
    //default          cellMDLimited Gauss linear 1;
    grad(p)            Gauss linear;
}

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

laplacianSchemes
{
    default            Gauss linear orthogonal;
    //default          Gauss linear limited 1;
    //default          Gauss skewCorrected linear limited 1;
}

interpolationSchemes
{
    //default          skewCorrected linear;
    default            linear;
}

snGradSchemes
{
    default            orthogonal;
    //default          limited 1;
}
```

- In the dictionary *fvSchemes* we can enable non-orthogonal corrections.
- Non-orthogonal corrections are chosen using the keywords **laplacianSchemes** and **snGradSchemes**.
- These are the **laplacianSchemes** and **snGradSchemes** schemes that you will use most of the times:
 - **orthogonal**: second order accurate, bounded on perfect meshes, without non-orthogonal corrections.
 - **corrected**: second order accurate, bounded depending on the quality of the mesh, with non-orthogonal corrections.
 - **limited ψ** : second order accurate, bounded depending on the quality of the mesh, with non-orthogonal corrections.
 - **uncorrected**: second order accurate, without non-orthogonal corrections. Stable but more diffusive than limited and corrected.

Numerical playground

How to adjust the numerical method to deal with non-orthogonality

```
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          1;
    nNonOrthogonalCorrectors 0;
    pRefCell             0;
    pRefValue            0;
}
```

- Additionally, in the dictionary *fvSolution* we need to define the number of **PISO** corrections (**nCorrectors**) and non-orthogonal corrections (**nNonOrthogonalCorrectors**).
- You need to do at least one **PISO** correction. Increasing the number of **PISO** correctors will improve the stability and accuracy of the solution at a higher computational cost.
- For orthogonal meshes, 1 **PISO** correction is ok. But as most of the time you will deal with non-orthogonal meshes, doing 2 **PISO** corrections is a good choice.
- If you are using a method with non-orthogonal corrections (**corrected** or **limited 1-0.5**), you need to define the number of non-orthogonal corrections (**nNonOrthogonalCorrectors**).
- If you use 0 **nNonOrthogonalCorrectors**, you are computing the initial approximation using central differences (accurate but unstable), with no explicit correction.
- To take into account the non-orthogonality of the mesh, you will need to increase the number of corrections (you get better approximations using the previous correction).
- Usually 2 **nNonOrthogonalCorrectors** is ok.

Numerical playground

- We will now illustrate a few of the discretization schemes available in OpenFOAM® using a model case.
- We will use the lid-driven square cavity case to study the effect of grading and non-orthogonality on the accuracy of the solution
- This case is located in the directory:

`$PTOFC/101FVM/nonorthoCavity/`

- In the case directory, you will find a few scripts with the extension `.sh`, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
 - `$> sh run_solver`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, and type the commands in the terminal. In this way, you will get used with the command line interface and OpenFOAM commands.
- If you are already comfortable with OpenFOAM, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

Numerical playground

What are we going to do?

- This is the same case as the one we used during the first tutorial session.
- The only difference is that we have modified the mesh a little bit in order to add grading and non-orthogonality.
- After generating the mesh, we will use the utility `checkMesh` to control the quality of the mesh. Is it a good mesh?
- We will use this case to learn how to adjust the numerical schemes according to mesh non-orthogonality and grading.
- After finding the numerical solution we will do some sampling and plotting.

Running the case

- You will find this tutorial in the directory `$PTOFC/101FVM/nonorthoCavity`
- In the terminal window type:

1. `$> foamCleanTutorials`
2. `$> blockMesh -dict system/blockMeshDict.0`
3. `$> checkMesh`
4. `$> pisoFoam | log.solver`
5. `$> postProcess -func sampleDict -latestTime`
6. `$> gnuplot gnuplot/gnuplot_script`
7. `$> paraFoam`

Numerical playground

To run the case, follow these steps

- First run the case using the original dictionaries. Did it crash right?
- Now change the **laplacianSchemes** and **snGradSchemes** to **limited 1**. It crashed again but this time it ran a few more time-steps, right?
- Now increase the number of **nNonOrthogonalCorrectors** to 2. It crashed again but it is running more time-steps, right?
- Now increase the number of **PISO** corrections to 2 (**nCorrectors**). Did it run?
- Basically we enabled non-orthogonal corrections, we computed better approximations of the gradients, and we increased the number of **PISO** corrections to get better predictions of the field variables (**U** and **p**).
- Now set the number of **nNonOrthogonalCorrectors** to 0. Did it crash right? This is telling us that the mesh is sensitive to the gradients.
- Now change the **laplacianSchemes** and **snGradSchemes** to **limited 0** (uncorrected). In this case we are not using non-orthogonal corrections, therefore there is no need to increase the value of **nNonOrthogonalCorrectors**.
- We are using a method that uses a wider stencil to compute the Laplacian, this method is more stable but a little bit more diffusive. Did it run?
- At this point, compare the solution obtained with corrected and uncorrected schemes. Which one is more diffusive?

Numerical playground

- When it comes to **laplacianSchemes** and **snGradSchemes** this is how we proceed most of the times (a robust setup),

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

snGradSchemes
{
    default      limited 1;
}
```

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

- This method works fine for meshes with non-orthogonality less than 75.
- If the non-orthogonality is more than 75, you should consider using **limited 0.5**, and increasing **nCorrectors** and **nNonOrthogonalCorrectors**.
- When the non-orthogonality is more than 85, the best solution is to redo the mesh.

Numerical playground

Exercises

- Using the non-orthogonal mesh and the original dictionaries, try to run the solver reducing the time-step. Do you get a solution at all?
- Try to get a solution using the method **limited 1** and two **nNonOrthogonalCorrectors** (leave **nCorrectors** equal to 1).

(Hint: try to reduce the time-step)

- If you managed to get a solution using the previous numerical scheme. How long did it take to get the solution? Use the robust setup, clock the time and compare with the previous case. Which one is faster? Do you get the same solution?
- Instead of using the non-orthogonal mesh, use a mesh with grading toward all edges. How will you stabilize the solution?

(Hint: take a look at the blockMesh slides in order to add grading to the mesh)

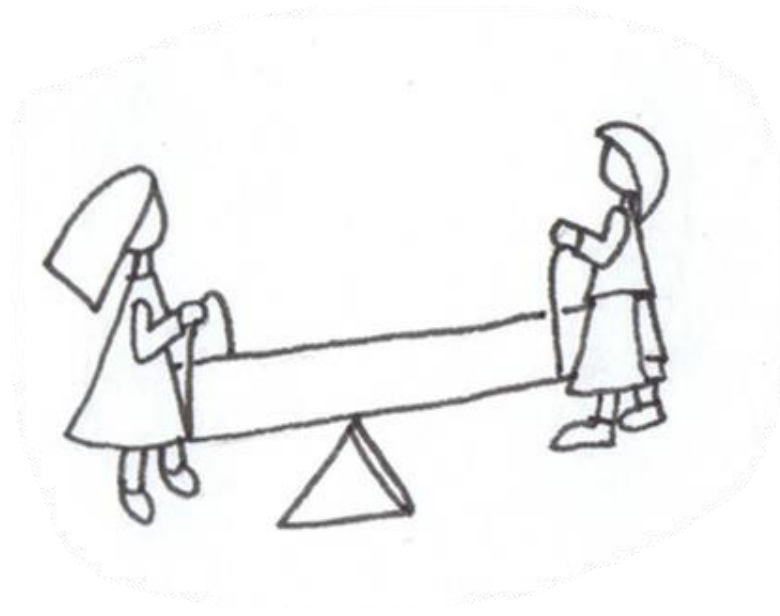
- Try to get a solution using a time-step of 0.05 seconds. Use the original discretization schemes for the gradient and convective terms.

(Hint: increase nCorrectors and nNonOrthogonalCorrectors)

- Using the uniform orthogonal mesh and a robust numerics, determined the largest CFL you can use. Is the solution still accurate? What about the clock-speed?
- Try to break the solver and interpret the output screen. You are allowed to modify the original mesh and use any combination of discretization schemes.

Numerical playground

**Seesaw:
Sod's shock tube.**



Numerical playground

Sod's shock tube

- This case has an analytical solution and plenty of experimental data.
- This is an extreme test case used to test solvers.
- Every single commercial and open-source solver use this case for validation of the numerical schemes.
- The governing equation of this test case are the Euler equations.

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

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

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

$$p = \rho R_g T$$

Numerical playground

High Purity Photolysis Shock Tube (NASA Tube)



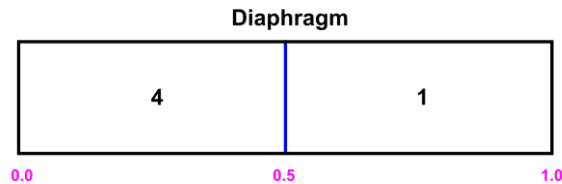
Shock tube. The driver section, including vacuum pumps, controls, and helium driver gas.

Photo credit: Stanford University. http://hanson.stanford.edu/index.php?loc=facilities_nasa

Copyright on the images is held by the contributors. Apart from Fair Use, permission must be sought for any other purpose.

Numerical playground

Sod's shock tube



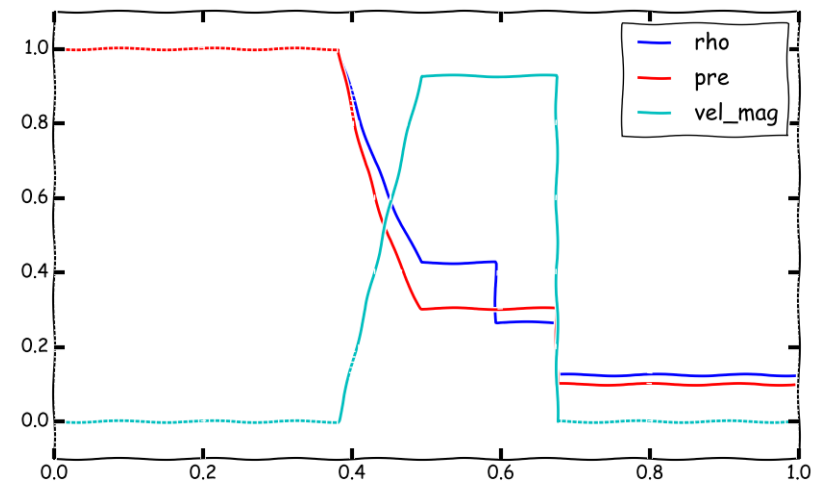
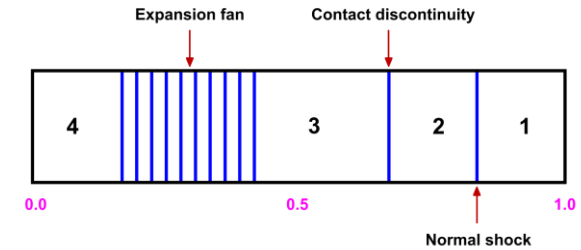
All walls are slip

$$U_4 = U_1 = 0$$

$$p_4 = 1, \quad p_1 = 0.1$$

$$T_4 = 0.00348, \quad T_1 = 0.00278$$

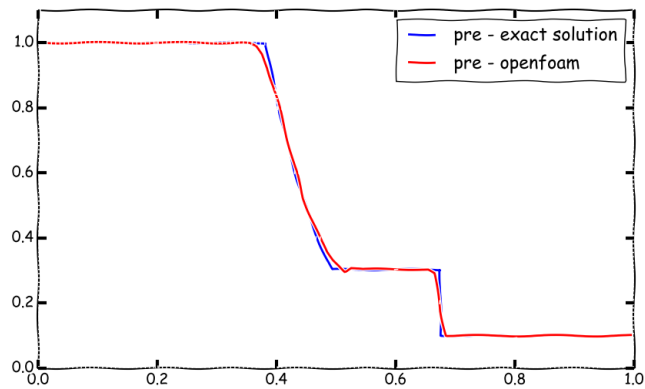
Boundary conditions and initial conditions



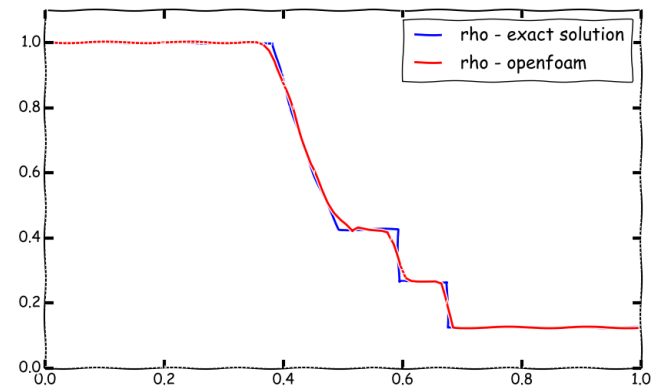
Analytical solution

Numerical playground

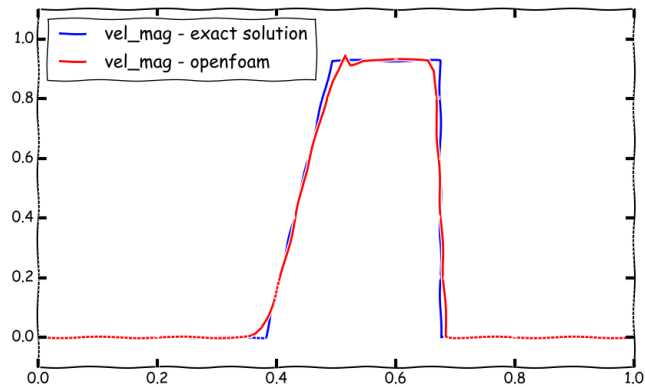
Sod's shock tube



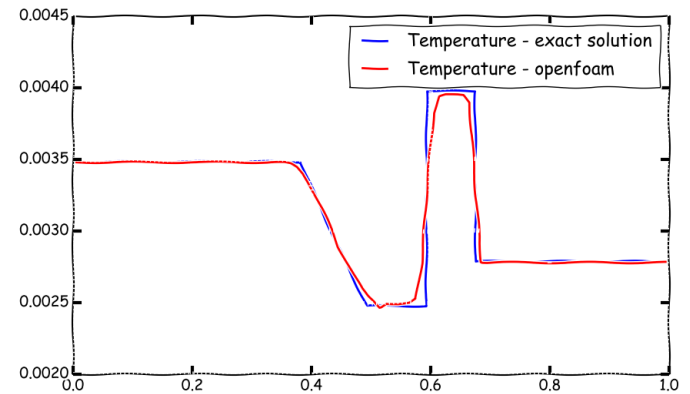
Pressure field



Density field



Velocity magnitude field



Temperature field

Numerical playground

- We will now illustrate a few of the discretization schemes available in OpenFOAM® using a severe model case.
- We will use the Sod's shock tube case.
- This case is located in the directory:

\$PTOFC/101FVM/shockTube/

- In the case directory, you will find a few scripts with the extension .sh, namely, `run_all.sh`, `run_mesh.sh`, `run_sampling.sh`, `run_solver.sh`, and so on.
- These scripts can be used to run the case automatically by typing in the terminal, for example,
 - `$> sh run_solver`
- These scripts are human-readable, and we highly recommend you open them, get familiar with the steps, and type the commands in the terminal. In this way, you will get used with the command line interface and OpenFOAM commands.
- If you are already comfortable with OpenFOAM, run the cases automatically using these scripts.
- In the case directory, you will also find the `README.FIRST` file. In this file, you will find some additional comments.

Numerical playground

What are we going to do?

- Now is your turn.
- You are asked to select the best discretization scheme for the physics involve.
- Remember the following concepts: accuracy, stability and boundedness.
- We will compare your numerical solution with the analytical solution.
- At this point, we are very familiar with the numerical schemes. It is up to you to choose the best setup.
- You can start using the original dictionaries.
- To find the numerical solution we will use the solver `rhoPimpleFoam`.
- `rhoPimpleFoam` is a transient solver for laminar or turbulent flow of a compressible gas.
- After finding the numerical solution we will do some sampling.
- At the end, we will do some plotting (using gnuplot or Python) and scientific visualization.

Numerical playground

Running the case

- You will find this tutorial in the directory `$PTOFC/101FVM/schockTube`
- In the terminal window type:

```
1.  | $> foamCleanTutorials
2.  | $> blockMesh
3.  | $> checkMesh
4.  | $> rm -rf 0
5.  | $> cp -r 0_org 0
6.  | $> setFields
7.  | $> rhoPimpleFoam | tee log.solver
8.  | $> postProcess -func sampleDict -latestTime
9.  | $> paraFoam
```

- To plot the analytical solution against the numerical solution, go to the directory `python` and run the Python script.
- In the terminal window type:

```
1.  | $> python3 python/sodshocktube.py
```

- The Python script will save four .png files with the solution. Feel free to explore and adapt the Python script to your needs.
- Python (version 3) must be installed in order to use the script

Numerical playground

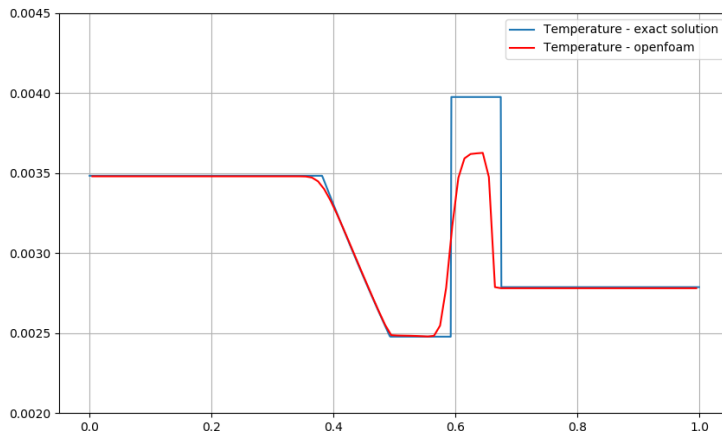
Running the case

- If you used the values proposed in the dictionaries, the solution diverged, right? Try to get the case working.

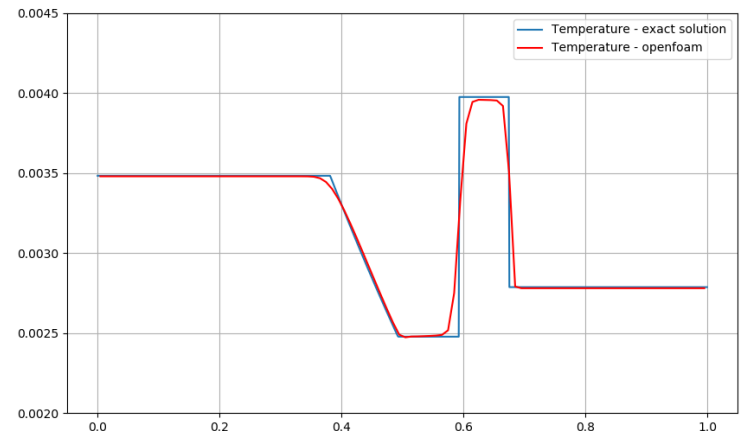
Hint: look at the gradient limiters.

- By adjusting the gradient limiters, the case will run, but the final solution is not very accurate. How can you increase the accuracy of the solution?

Hint: look at the PIMPLE corrections.



Not so accurate solution



Accurate solution

Numerical playground

Exercises

- Using the proposed case setup, try to get an accurate solution by reducing the time-step or refining the mesh. Did you succeed in getting an accurate solution?
- Run the case using different time discretization schemes.
- Run the case using different gradient discretization schemes.
- Run the case using different convective discretization schemes for the term $\text{div}(\mathbf{\phi}, \mathbf{U})$.
- Run the case using different convective discretization schemes for the terms $\text{div}(\mathbf{\phi}, \mathbf{e})$ and $\text{div}(\mathbf{\phi}, \mathbf{K})$. What are the variables \mathbf{e} and \mathbf{K} ?
- Extend the case to 2D and 3D. Do you get the same solution?
- Try to run a 2D case using a triangular mesh and adjust the numerical scheme to get an accurate and stable solution.
- Try to run the 1D case using an explicit solver. For the same CFL number, do you have the same time step size as for the implicit solver?

(Hint: look for the solver with the word Central)

- Try to break the solver (this is extremely easy in this case). You are allowed to modify the original mesh and use any combination of discretization schemes.