Aalto University
School of Engineering

Lecture 5: Finite Volume Methods 2/2

V.Vuorinen
Aalto University School of Engineering
CFD Course, Spring 2019

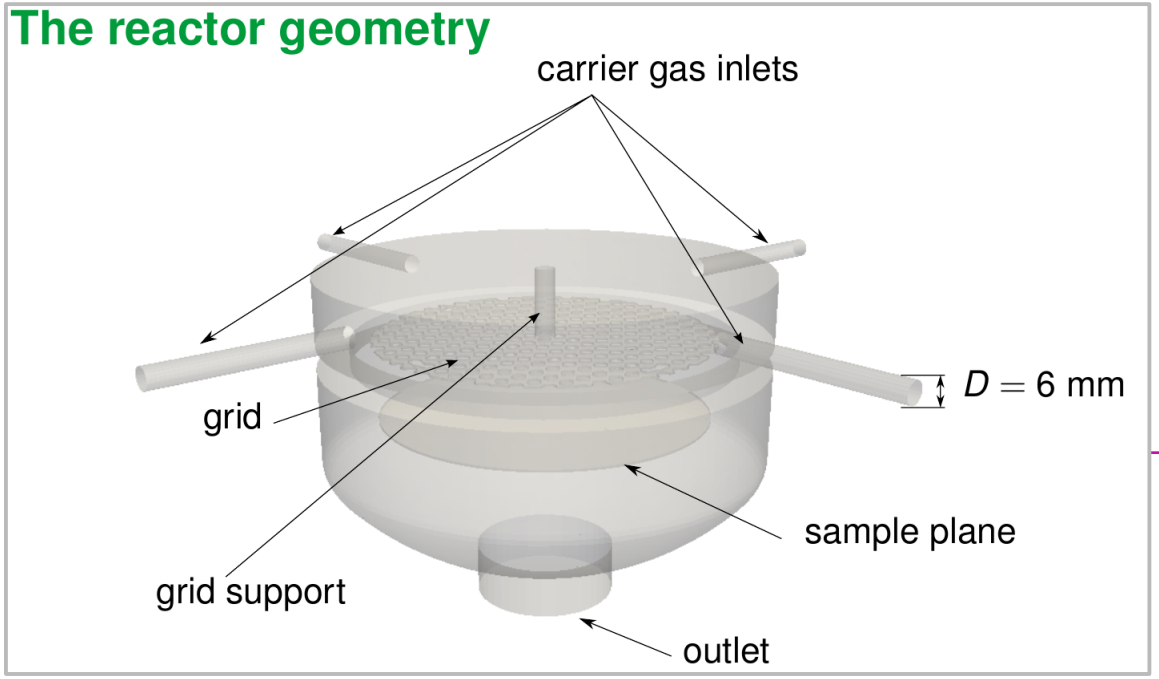
February 11th 2019, Otaniemi
ville.vuorinen@aalto.fi

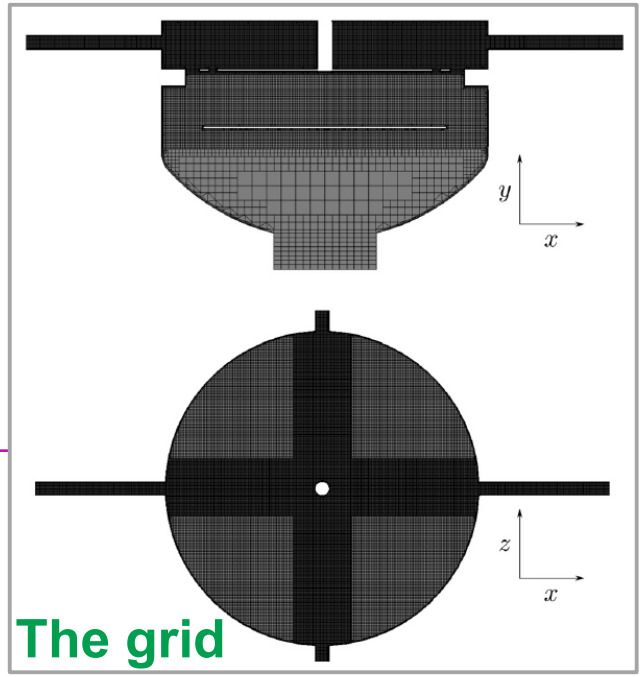
Example: Multi-jet flow reactor

- OpenFOAM open source code
- Finite volume based discretization (2nd order space/time)
- Implicit **L**arge-**E**ddy **S**imulation because flows are low Reynolds number biased. 6.5M cell mesh.
- Iso-thermal, incompressible flow. Surface reactions NOT accounted.
- Reynolds number sweep

P.Peltonen, V.Vuorinen et al., Fluid dynamical aspects of the ALD process Using Scale Resolving Simulations, Journal of Vacuum Technology, (2018)

The reactor geometry



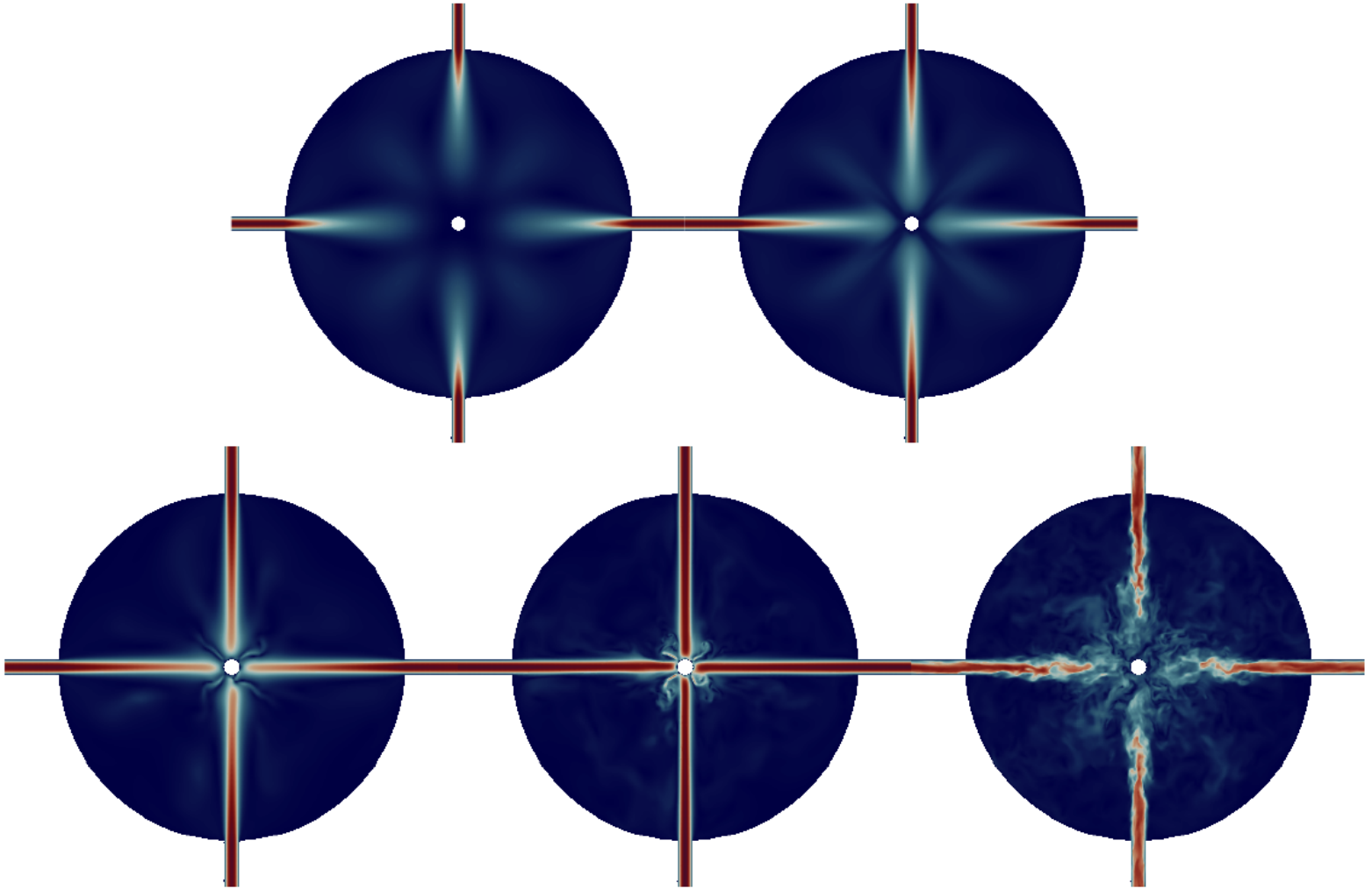


The grid

Overview of flow in the ALD chamber at Re=2400

P.Peltonen, V.Vuorinen et al., Fluid dynamical aspects of the ALD process Using Scale Resolving Simulations, Journal of Vacuum Technology, (2018)

Here: constant flow of carrier gas from each inlet. From Re~30 to high Re~3000



Objectives

- **Overall message:** at a given high quality mesh, discretization of the convection term is probably the most important thing to influence simulation quality.

- Continue with finite volume methods and evaluation of derivatives by FVM
- Centered vs upwind flux evaluation and link to CD2 and UW1 finite difference method

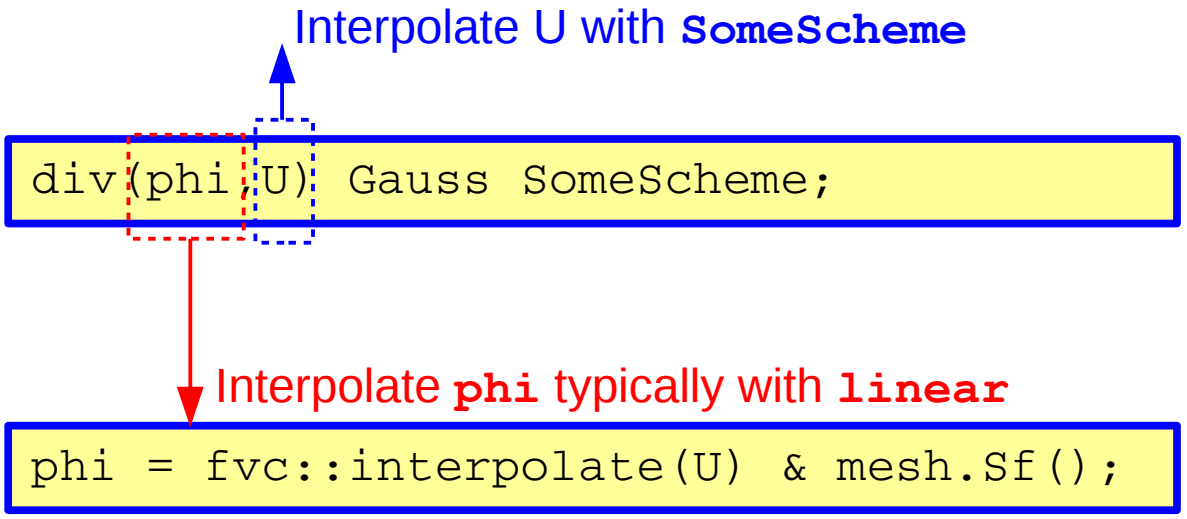
Part 1: How can OpenFOAM user impact the chosen finite volume method discretization?

Finite volume discretization: fvSchemes

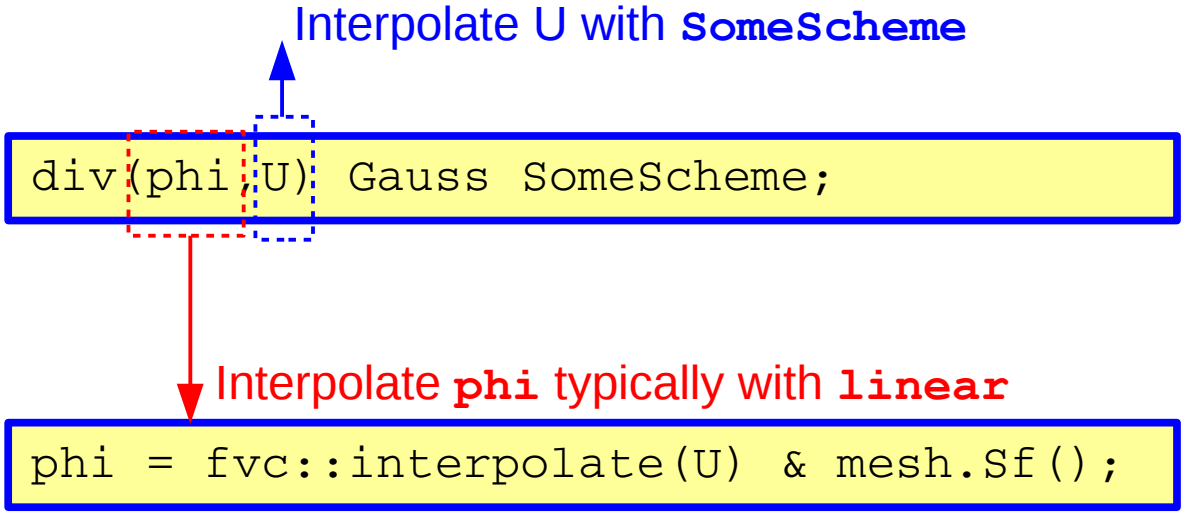
```
ddtSchemes
{
    default          Euler;
}

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

divSchemes
{
    default          none;
    div(phi,U)       Gauss linear;   % linear interpolation for U
    % div(phi,U)     Gauss upwind; % upwind interpolation for U
    % div(phi,U)     Gauss GammaV 0.1; % flux limited interpolation for U
}
```

Try "Gauss hello"

In OpenFOAM the `fvSchemes` definitions and interpolation of convecting flux lead to so-called skew symmetric discretization (convection conserves E_{kin})



In OpenFOAM the `fvSchemes` definitions and interpolation of convecting flux lead to so-called skew symmetric discretization (convection conserves E_{kin})



$$\mathbf{C}_{fvm} = \frac{1}{\Omega}\int_{\Omega} \nabla \cdot (\mathbf{uu}) d\Omega = \frac{1}{\Omega}\int_{\partial\Omega} \mathbf{uu} \cdot \mathbf{n}\, dS$$

$$\approx \frac{1}{\Omega}\sum_{f=1}^{N} \mathbf{u}_f \mathbf{u}_f^c \cdot \mathbf{n}_f\, dS_f,$$

Kinetic energy decays when pouring milk stops because of viscous dissipation – to "capture" the viscous effects correctly one wishes to minimize numerical dissipation (e.g. in interpolation of the `div(phi,U)` term)

The rate of change of total kinetic energy is negative in the absence of energy source

$$\frac{dE_{kin}^{tot}}{dt} = -\chi^{tot} < 0$$

For periodic boundary Conditions (HW3):

$$E_{kin}^{tot} = \int_V \frac{1}{2}\boldsymbol{u}^2 \, dV$$

$$\chi^{tot} = \int_V \nu |\nabla \boldsymbol{u}|^2 \, dV$$

Note: energy decay eqn derived by volume-integrating term

$\boldsymbol{u} \cdot NS$

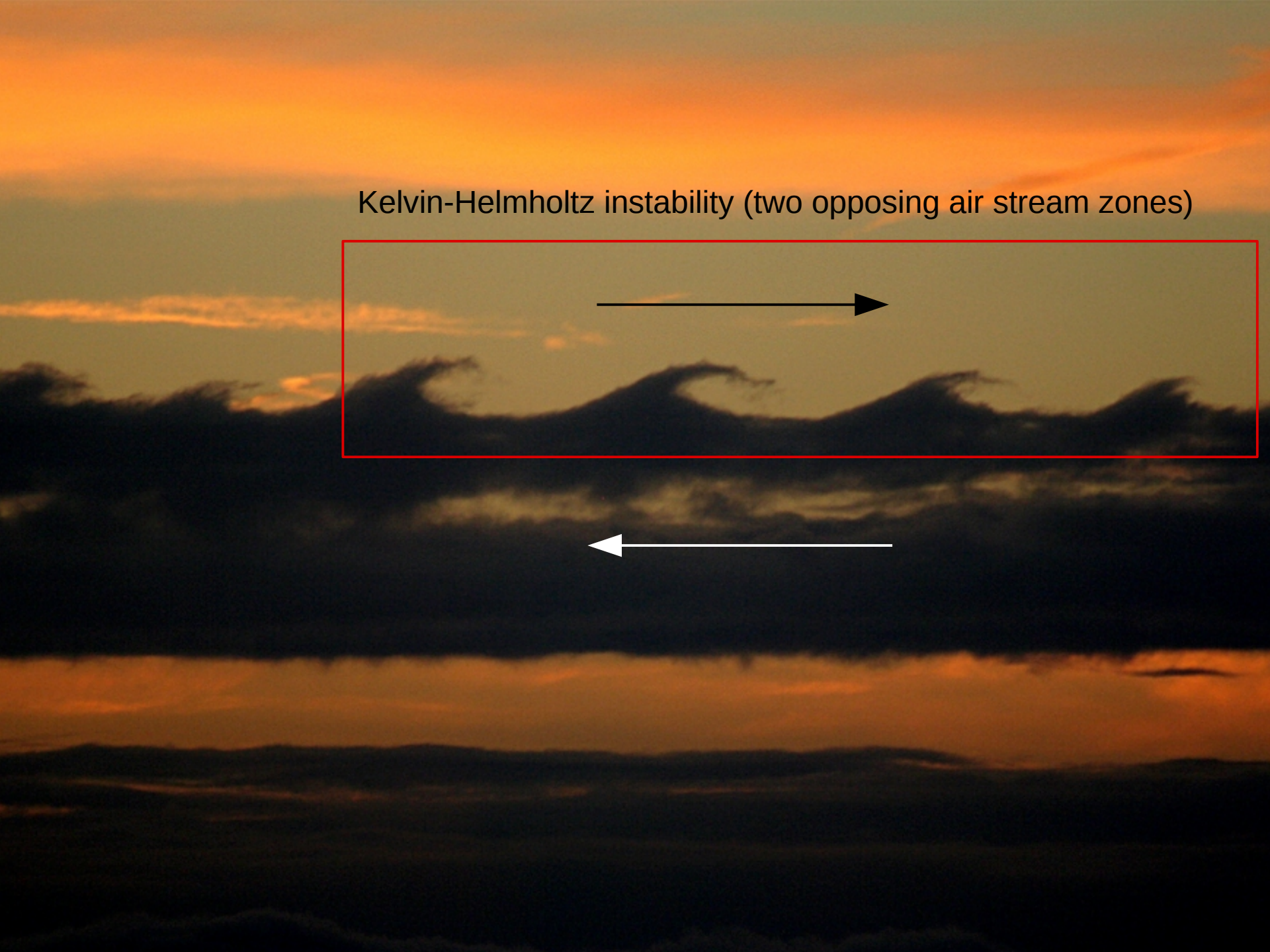
$Work = \boldsymbol{F} \cdot \boldsymbol{ds} = \boldsymbol{F} \cdot \boldsymbol{u}\, dt$

Table 1
Examples of differential operator types in OpenFOAM®. The `fvc` syntax declares the operators to be explicit i.e. to be evaluated with known values [26].

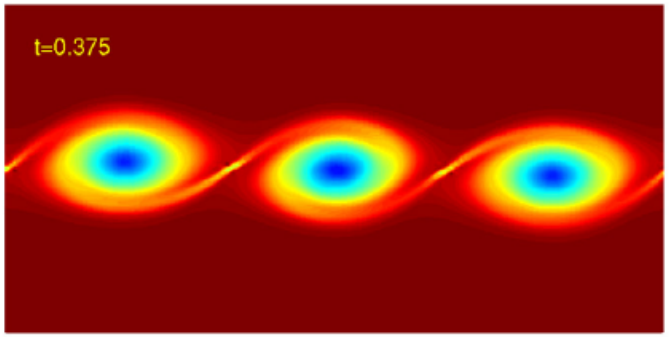
Operator	Syntax	Output for a scalar	Output for a vector
∇	`fvc::grad(F)`	`volVectorField`	`volTensorField`
$\nabla\times$	`fvc::curl(F)`	–	`volVectorField`
Δ	`fvc::laplacian(F)`	`volScalarField`	`volVectorField`

Table 2
Examples of field data types in OpenFOAM® [26].

Field name	Variable	Data type	Variable values defined at
Velocity	`U`	`volVectorField`	Cell center
Pressure	`p`	`volScalarField`	Cell center
Rate of strain	`S`	`volTensorField`	Cell center
Velocity flux	`phi`	`surfaceScalarField`	Cell face center

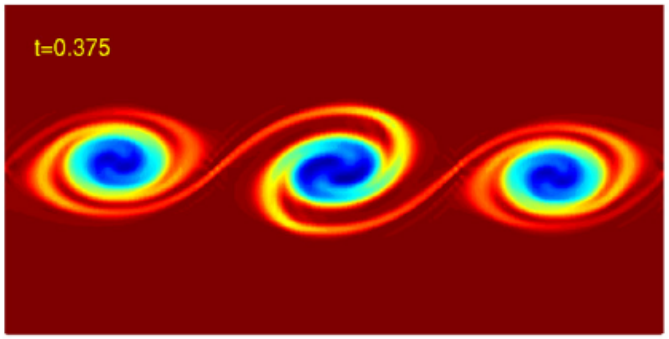

Kelvin-Helmholtz instability (two opposing air stream zones)

Simulation of KH-instability in OpenFOAM (Vuorinen et al. 2012)



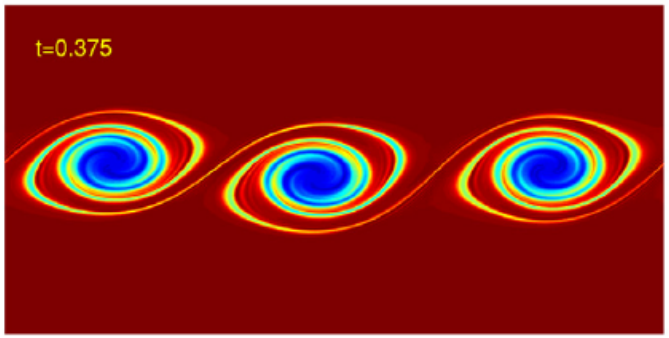
→ Blend of 2nd order (linear) and 1st order upwind interpolation for velocity (128^2) (**`blended`** scheme in OpenFOAM)

→ **Overall space accuracy: 1st order**



→ 2nd order interpolation for velocity with 4th order numerical diffusion (128^2) (~**`linear`** scheme in OpenFOAM)

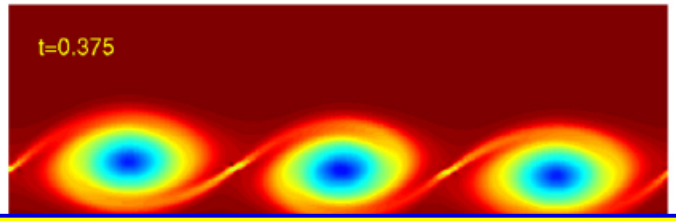
→ **Overall space accuracy: 2nd order**



→ 2nd order interpolation for velocity with 4th order numerical diffusion (512^2) (~**`linear`** scheme in OpenFOAM)

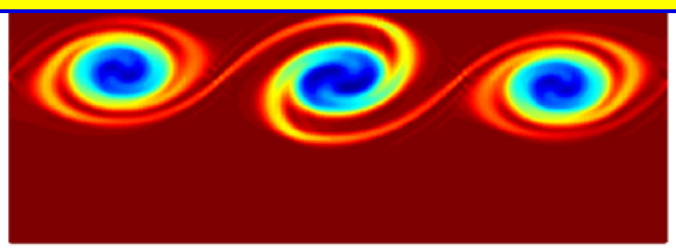
→ **Overall space accuracy: 2nd order**

Simulation of KH-instability in OpenFOAM (Vuorinen et al. 2012)



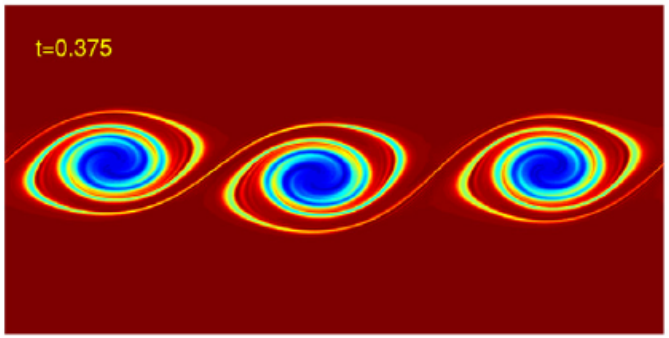
→ Blend of 2nd order (linear) and 1st order upwind interpolation for velocity (128^2) (**`blended`** scheme in OpenFOAM)

Why is the 1st order simulation so diffusive and the 2nd order quite well resolved (fine details seen)?
→ **This will be explained in what follows.**



…der numerical diffusion (128^2) (~**`linear`** scheme in OpenFOAM)

→ **Overall space accuracy: 2nd order**



→ 2nd order interpolation for velocity with 4th order numerical diffusion (512^2) (~**`linear`** scheme in OpenFOAM)
→ **Overall space accuracy: 2nd order**

Part 2: Summary of finite volume method

Recall the Gauss' Theorem as "Generalization of Integration"

- Gauss theorem to convert volume integrals to surface integrals

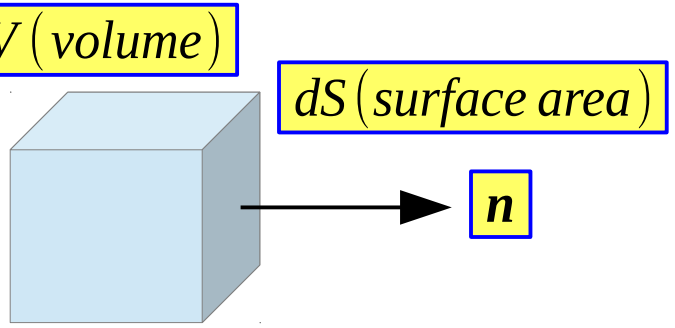
$$\int_V \nabla \cdot \boldsymbol{F}\, dV = \int_S \boldsymbol{F} \cdot \boldsymbol{n}\, dS$$

- Convection terms are of the divergence form

$$\int_V \nabla \cdot (\boldsymbol{u}\phi)\, dV = \int_S (\boldsymbol{u}\phi) \cdot \boldsymbol{n}\, dS$$

- Diffusion terms are also of the divergence form

$$\int_V \nabla \cdot (\nu \nabla \phi)\, dV = \int_S \nabla \phi \cdot \boldsymbol{n}\, dS$$



Observation: Transport equations can be written in flux form so that the rate of change of a quantity within a control volume V depends on the fluxes through the boundary S → **the basic idea of finite volume method.**

The Gauss-theorem forms the basics of the finite volume method – the method in OpenFOAM

Differential form

Conservation form

$$\frac{1}{V}\int_V \boxed{\frac{\partial \phi}{\partial t}+\nabla\cdot(\boldsymbol{u}\phi)=\nabla\cdot(\nu\nabla\phi)} \qquad \boxed{\frac{1}{V}\int_V \frac{\partial \phi}{\partial t}dV+\frac{1}{V}\int_S(\boldsymbol{u}\phi)\cdot\boldsymbol{n}\,dS=\frac{1}{V}\int_S \nu\nabla\phi\cdot\boldsymbol{n}\,dS}$$

$$\frac{1}{V}\int_V \boxed{\frac{\partial \boldsymbol{u}}{\partial t}+\nabla\cdot\boldsymbol{u}\boldsymbol{u}=-\nabla p+\nu\Delta\boldsymbol{u}} \qquad \boxed{\frac{1}{V}\int_V \frac{\partial \boldsymbol{u}}{\partial t}dV+\frac{1}{V}\int_S(\boldsymbol{u}\boldsymbol{u})\cdot\boldsymbol{n}\,dS=-\frac{1}{V}\int_S p\boldsymbol{n}\,dS+\frac{1}{V}\int_S \nu\nabla\boldsymbol{u}\cdot\boldsymbol{n}\,dS}$$

In the finite volume discretization method the fluxes on the cell faces are numerically estimated utilizing interpolation functions. **Integration is replaced by summation:**

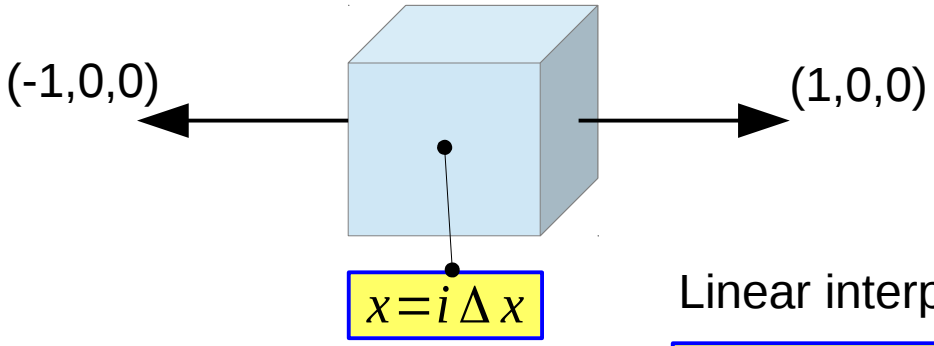
$$\boxed{\frac{1}{V}\int_S(\boldsymbol{u}\phi)\cdot\boldsymbol{n}\,dS\approx\frac{1}{V}\Sigma_{faces}(\boldsymbol{u}\phi)_f\cdot\boldsymbol{n}_f dS_f}$$

$$\boxed{\frac{1}{V}\int_S \nu\nabla\phi\cdot\boldsymbol{n}\,dS\approx\frac{1}{V}\Sigma_{faces}(\nu\nabla\phi)_f\cdot\boldsymbol{n}_f dS_f}$$

Here the subscript *f* refers to evaluation of the quantity at the cell face center.

Example: Discretize Derivative of a Function *u=u(x)* Using the FVM on a Cartesian Grid

$$u_x = \frac{\partial u}{\partial x} \approx \frac{1}{\Delta x \Delta y \Delta z} \int_{\Delta x} u_x \, dx \, dy \, dz = \frac{1}{\Delta x} \int_{\Delta x} u_x \, dx = \frac{1}{\Delta x} \int_{\Delta x} u \, dS \, n_x = \frac{u_{i+1/2} - u_{i-1/2}}{\Delta x}$$



Linear interpolation would give:

$$\frac{\partial u}{\partial x} \approx \frac{u_{i+1/2} - u_{i-1/2}}{\Delta x} = \frac{u_{i+1} - u_{i-1}}{2 \Delta x}$$

Flux interpolation essential in finite volume method

Let us consider convection equation with constant velocity $u > 0$

$$\frac{\partial \phi}{\partial t} + u \frac{\partial \phi}{\partial x} = 0$$

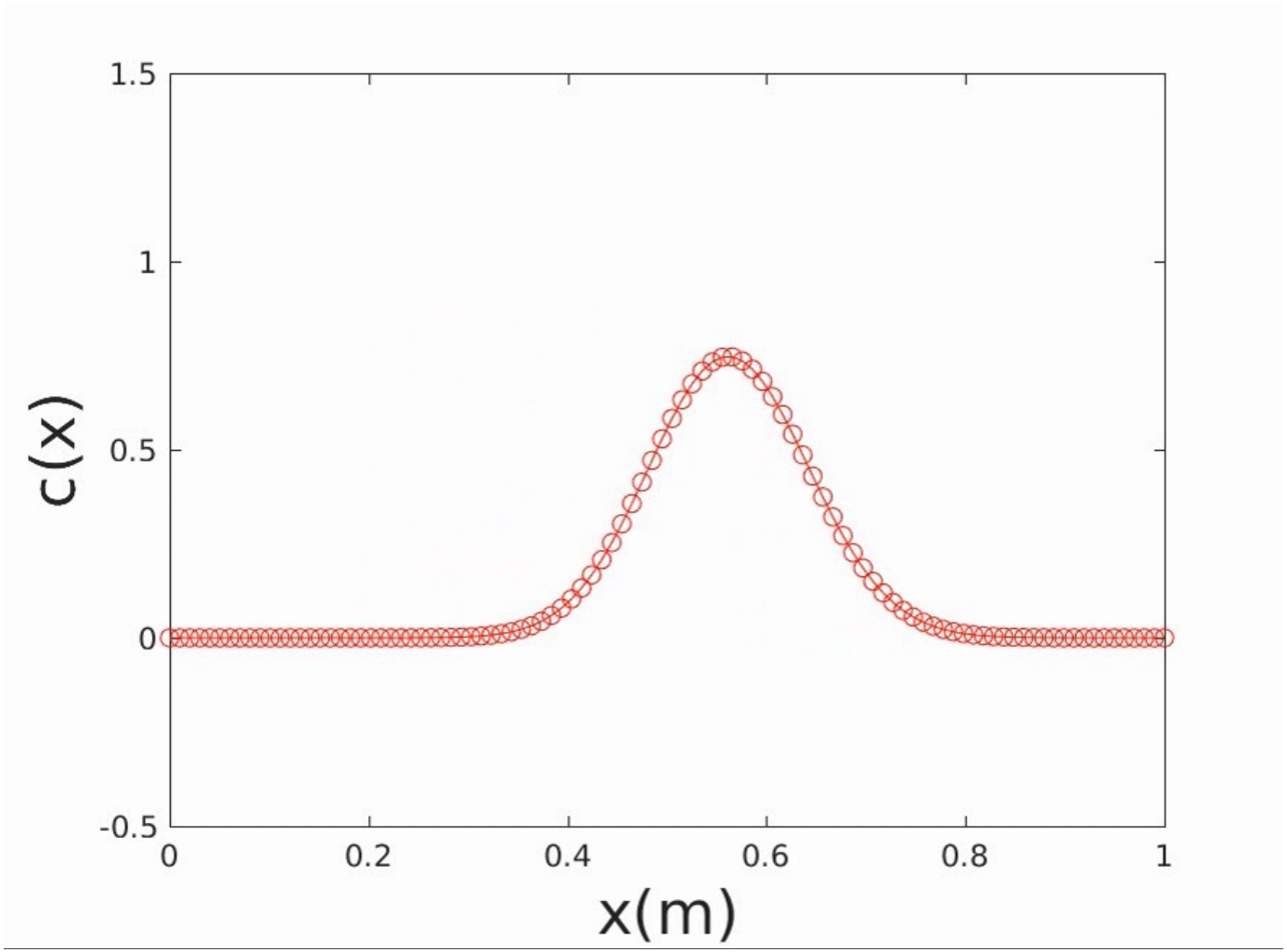
In flux form, the space derivative becomes (see lecture 4):

$$u \frac{\partial \phi}{\partial x} = \frac{1}{\Delta x} \Big/_{x^-}^{x^+} u\phi = u \frac{\phi(x^+) - \phi(x^-)}{\Delta x}$$

$$x^+ = x + \Delta x/2$$

$$x^- = x - \Delta x/2$$

Recall: UW1 for pure convection equation is diffusive



Linear, upwind and blended interpolation in finite volume method

Linear

$$\phi\left(x^{+}\right)=\phi_{linear}=0.5\left(\phi_{i+1}+\phi_{i}\right)$$

$$\frac{\partial \phi}{\partial x}\approx\frac{\phi_{i+1}-\phi_{i-1}}{2\Delta x}$$

Upwind

$$\phi\left(x^{+}\right)=\phi_{upwind}=\phi_{i},\ \text{for}\ \ u>0$$

$$\frac{\partial \phi}{\partial x}\approx\frac{\phi_{i}-\phi_{i-1}}{\Delta x}$$

How about weighting the linear flux and upwind flux differently?

Blended

$$\phi\left(x^{+}\right)=\phi_{blended}=(1-\alpha)\,\phi_{linear}+\alpha\,\phi_{upwind},\ 0<\alpha<1$$

Numerical diffusion from the viewpoint of blended scheme

Elaborating from the previous slide, a blended flux is expressed as **linear flux + diffusive flux**:

$$\phi_{blended} = \frac{1}{2}(\phi_{i+1} + \phi_i) - \frac{\alpha}{2}(\phi_{i+1} - \phi_i)$$

ϕ_{linear} $\phi_{diffusive}$

Recall: Fick's law in diffusion

$$J = -k \nabla \phi$$

Thus, the blend of linear and upwind method leads to the following discretization:

$$\frac{\partial \phi}{\partial x} \approx \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} - \frac{\Delta x \alpha}{2} \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2}$$

Recall:

$$\frac{\partial^2 \phi}{\partial x^2} \approx \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2}$$

Numerical diffusion from the viewpoint of blended scheme

$$\phi_{blended} = \frac{1}{2}(\phi_{i+1} + \phi_i) - \frac{\alpha}{2}(\phi_{i+1} - \phi_i)$$

ϕ_{linear} $\phi_{diffusive}$

Thus, the blend of linear and upwind method leads to the following discretization:

$$\frac{\partial \phi}{\partial x} \approx \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} - \frac{\Delta x\, \alpha}{2} \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2}$$

CD2 Diffusive term. Interpretation: strong numerical diffusion

Conclusion 1: linear interpolation is non-diffusive (but dispersive). When upwinding is added, numerical diffusion increases because in discretized equations an extra term – proportional to 2nd derivative - arises. This term acts in numerical simulations similar to **added viscosity/diffusivity**. Such term stabilizes but on the other hand may completely deteriorate the solution. **Solution**: use **flux limiter** instead of blending.

How does the numerical diffusion of upwind-biased scheme (blended) reflect to convection equation solution?

$$\frac{\partial \phi}{\partial t} + u \frac{\partial \phi}{\partial x} = 0$$

Discretizing convection equation with blended scheme can be written:

$$\frac{\partial \phi}{\partial t} = -u \frac{\phi_{i+1} - \phi_{i-1}}{2\Delta x} + |u| \frac{\Delta x \alpha}{2} \frac{\phi_{i+1} - 2\phi_i + \phi_{i-1}}{\Delta x^2}$$

In fact, by adding upwinding by blending CD2 with UW1 corresponds to solving a different equation with CD2:

$$\frac{\partial \phi}{\partial t} + u \frac{\partial \phi}{\partial x} = \kappa \frac{\partial^2 \phi}{\partial x^2}, \text{ where } \kappa = |u| \frac{\Delta x \alpha}{2}$$

Conclusion 2: All numerics changes the physics to some extent. Adding upwinding to convection term add diffusion. The modified equation has now also a numerical diffusion term and the original character of convection (non-diffusive) is modified.

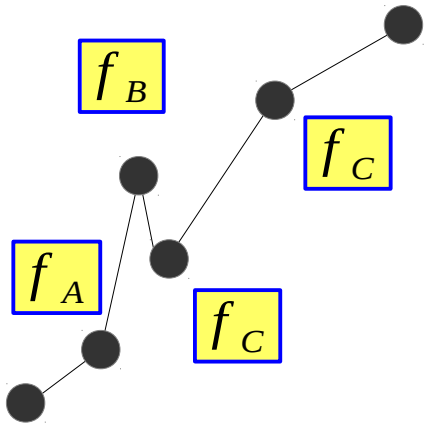
About interpolation and flux limiting

Thus, the blend of linear and upwind method leads to the following discretization:

$$\phi_{\text{flux limited}} = \phi_{linear} + \beta\,\phi_{diffusive}$$

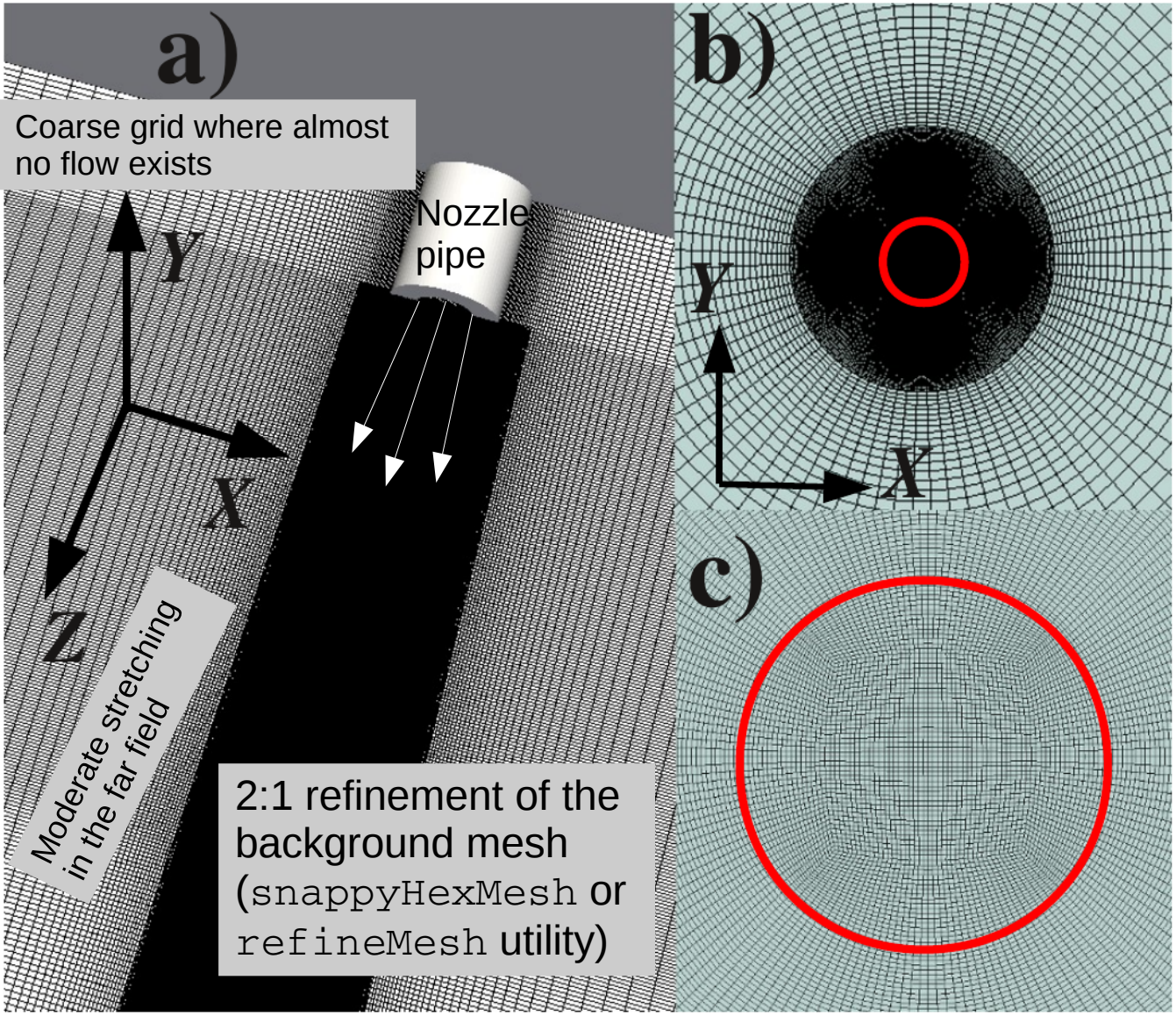
where the switch **β** is a locally defined, time-dependent parameter which can be used to control the numerical diffusion. Often such local diffusive flux is needed in practice to keep simulation stable and running.

FLUX LIMITING is useful interpolation technique close to high oscillating parts: how would you numerically compute flux/derivative around point B?
BENEFIT: more stable and robust simulation.
DRAWBACK: numerical diffusion increases.



Part 3: A word about computational grids

Mesh in gas jet studies: Vuorinen et al. (2013, 2014)



3D

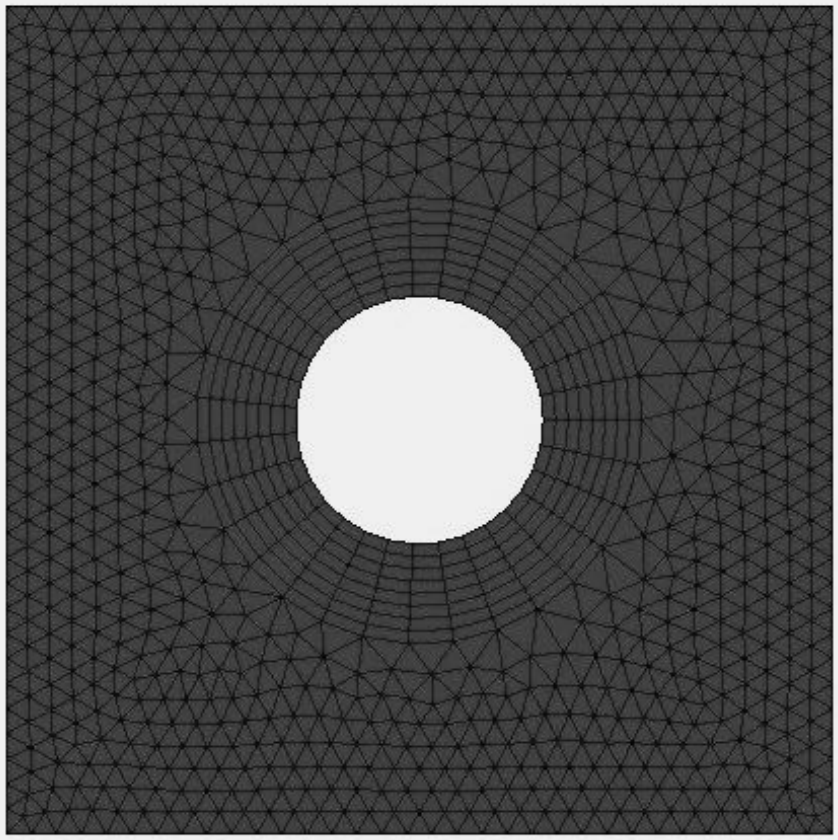
D

Visualization of instantaneous Incoming jet concentration (0 < c < 1)

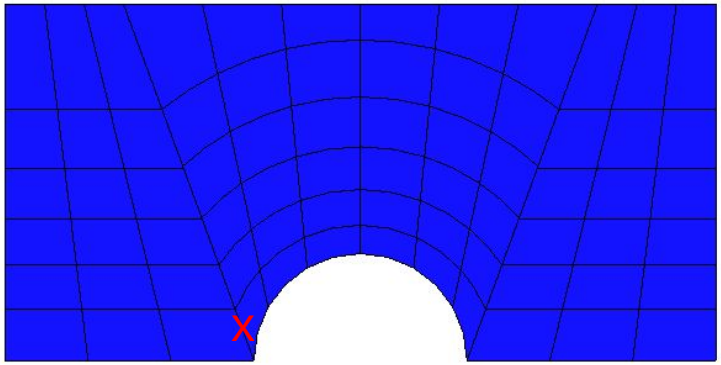
https://www.youtube.com/watch?v=v7yEvi31bkU

Computational grids: structured vs unstructured

Combination of structured mesh (surface conforming)
and unstructured mesh



X = high level of non-orthogonality



Thinking task

On a (really small) 3 by 3 grid:
a) how many cell centroids (P) do we have? (easy)
b) how many cell faces do we have?
c) how many neighbour cells altogether exist for a given cell?
d) how many boundary cell faces do we have?
e) if we interpolate a flux to the cell face, how many interpolants do we get ?
f) is the number larger or smaller than cell centroids?
g) how does the storage space for face vs centroid values differ ?

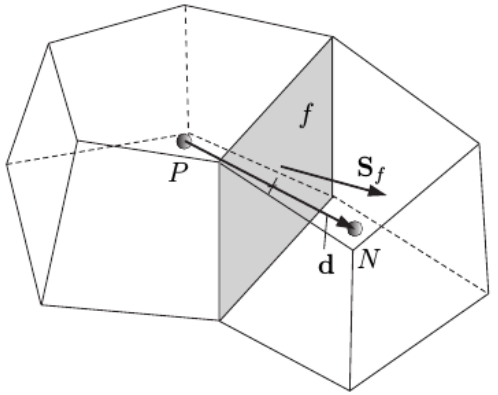
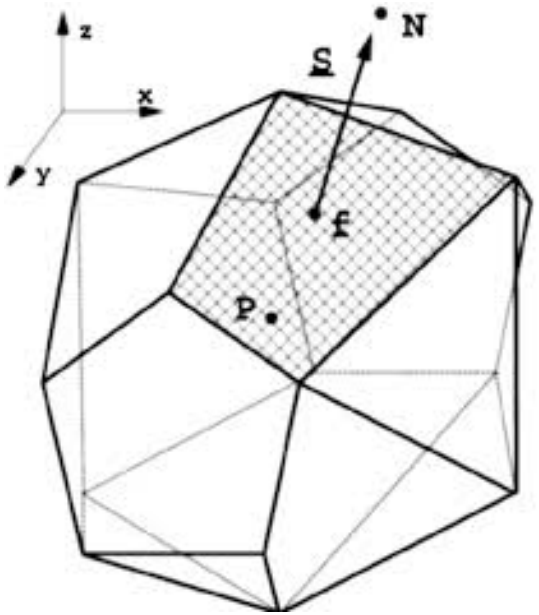


Figure 2.2: Parameters in finite volume discretisation

Thinking task



On a (really small) 3 by 3 grid:
a) how many cell centroids (P) do we have? (easy)
b) how many cell faces do we have?
c) how many neighbour cells altogether exist for a given cell?
d) how many boundary cell faces do we have?
e) what does nof faces tell about nof interpolations?
f) how does the storage space for face vs centroid values differ ?

Answers
a) 9
b) 12 + 12
c) 2,3 or 4
d) 12
e) Number of faces tells about number of interpolations.
f) More face values than centroid values.

Conclusion: special data types are needed to store the data at centroids and faces.

OpenFOAM has unstructured grid topology

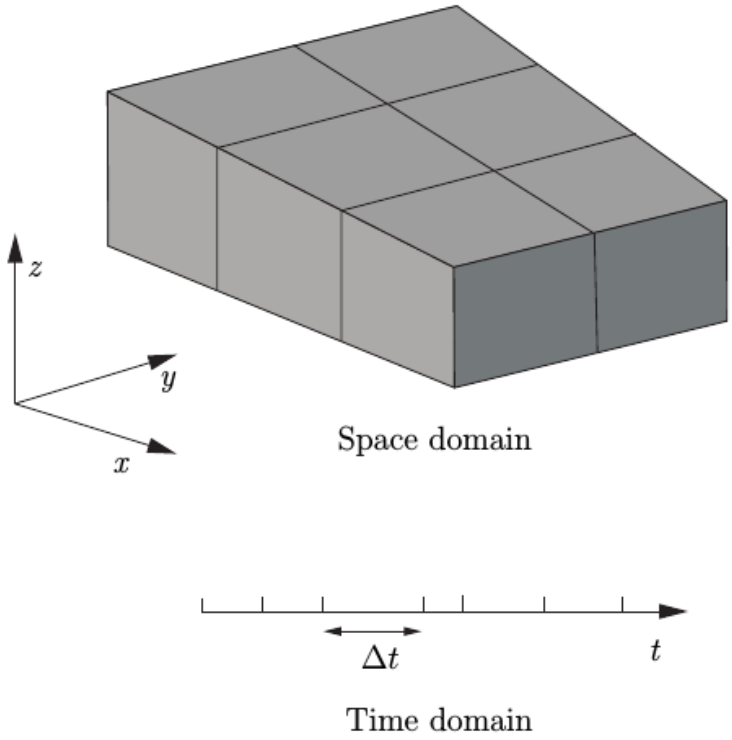


Figure 2.1: Discretisation of the solution domain

Vector and scalar quantities stored in cell centers (point P) while fluxes can be interpolated on cell faces from adjacent points.

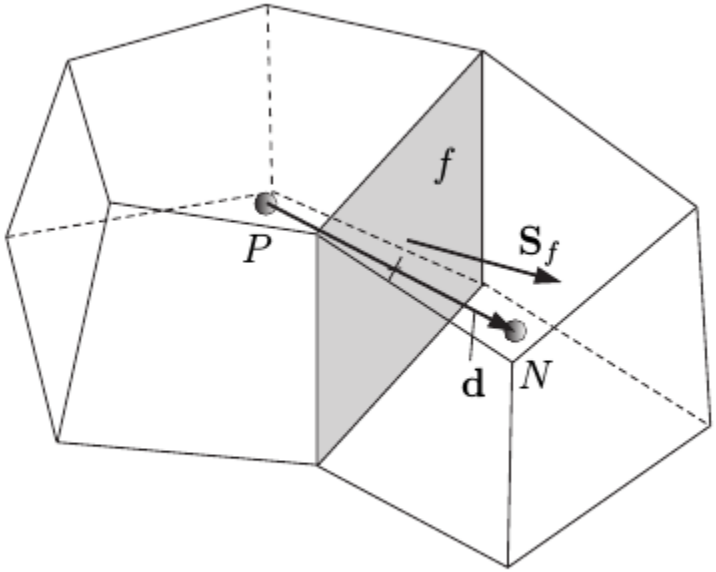


Figure 2.2: Parameters in finite volume discretisation

OpenFOAM has unstructured grid topology

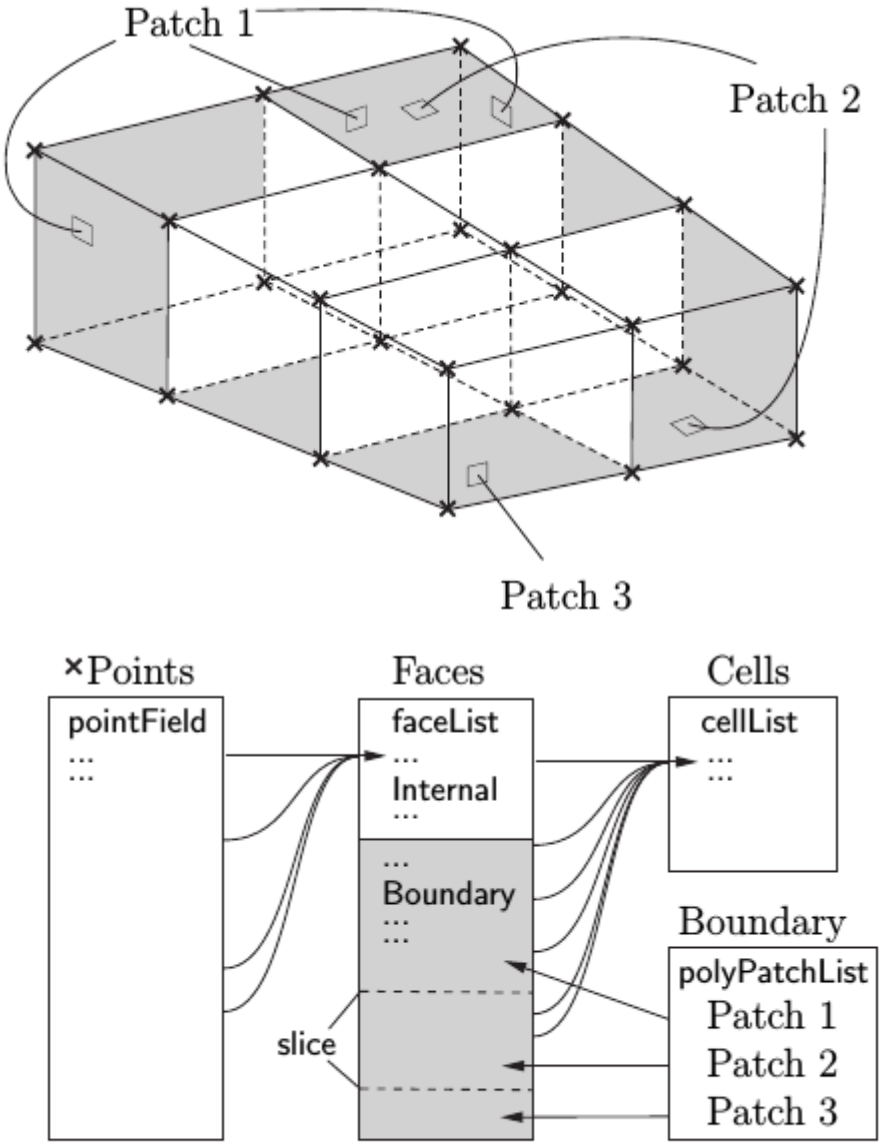


Figure 2.3: Schematic of the basic mesh description used in OpenFOAM

“Unstructured” grid topology implemented with very handy data types

Class	Description	Symbol	Access function
volScalarField	Cell volumes	V	`V()`
surfaceVectorField	Face area vectors	$\mathbf{S}_f$	`Sf()`
surfaceScalarField	Face area magnitudes	$\lvert\mathbf{S}_f\rvert$	`magSf()`
volVectorField	Cell centres	$\mathbf{C}$	`C()`
surfaceVectorField	Face centres	$\mathbf{C}_f$	`Cf()`
surfaceScalarField	Face motion fluxes **	ϕ_g	`phi()`

Table 2.1: fvMesh stored data.

The link between data types and OpenFOAM top-level code syntax?

```
dC  = dt*(fvc::laplacian(nu, C) - fvc::div(phi,C));
```

The link between OpenFOAM top-level code syntax and mathematical eqs?

$$\frac{\partial c}{\partial t}+\nabla\cdot(\boldsymbol{u}c)=\nabla\cdot\nu\nabla c$$

Part 4: Projection Method (Used in Bolund Hill Case) Implementation in OpenFOAM

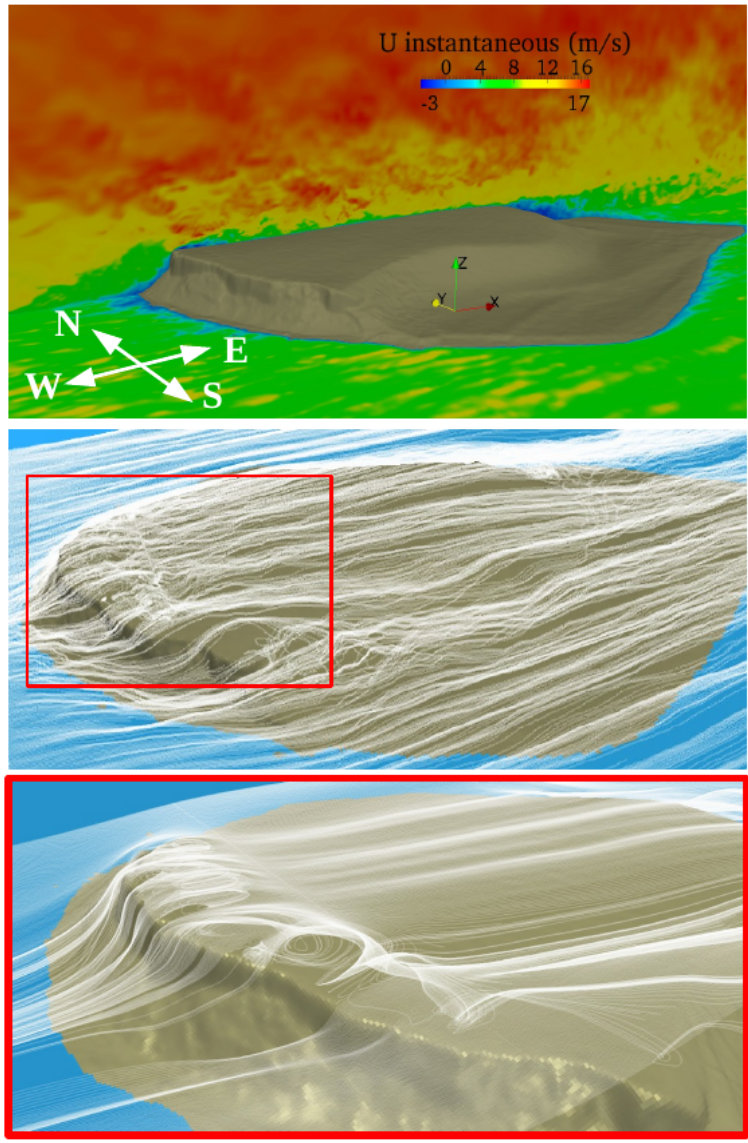
Essentials of OpenFOAM top-level solver `rk4projectionFoam`

```
// the main while loop in rk4projectionFoam.C
#include "CreatePoissonMatrix.H"

while (runTime.run())
  {
      #include "readTimeControls.H"
      #include "setDeltaT.H"
      runTime++;

      Uold = U; Uc = U; dt=runTime.deltaT();     // store old variables

      phi = (fvc::interpolate(U) & mesh.Sf());  // evaluate flux
      dU  = dt*(fvc::laplacian(turbulence->nuEff(), U)- fvc::div(phi,U) );
      Uc  = Uc +a1*dU; U = Uold + 0.5*dU;       // accumulate to Uc
      #include "PressureCorrection.H"           // remove divergence from U

      phi = (fvc::interpolate(U) & mesh.Sf());
      dU  = dt*(fvc::laplacian(turbulence->nuEff(), U) - fvc::div(phi,U) );
      Uc  = Uc +a2*dU; U = Uold + 0.5*dU;
      #include "PressureCorrection.H"

      phi = (fvc::interpolate(U) & mesh.Sf());;
      dU  = dt*(fvc::laplacian(turbulence->nuEff(), U) - fvc::div(phi,U) );
      Uc  = Uc +a3*dU; U = Uold + dU;
      #include "PressureCorrection.H"

      phi = (fvc::interpolate(U) & mesh.Sf());
      dU  = dt*(fvc::laplacian(turbulence->nuEff(), U) - fvc::div(phi,U) );
      Uc  = Uc +a4*dU; U = Uc;                  // set U = Uc
      #include "PressureCorrection.H"           // and remove divergence

      phi = (fvc::interpolate(U) & mesh.Sf());
      turbulence->correct();                    // SGS-model updated

      runTime.write();
    }
```

```
// PressureCorrection.H
U.correctBoundaryConditions();
solve(pEqn==fvc::div(U));
U = U - fvc::grad(p);
U.correctBoundaryConditions();
```

```
// CreatePoissonMatrix.H
fvScalarMatrix pEqn
(
 fvm::laplacian(p)
);
pEqn.setReference(pRefCell, pRef);
```

Vuorinen et al. Computers & Fluids (2014)
Vuorinen et al. Advances in Engineering Software (2015)



Essentials of OpenFOAM top-level solver `rk4projectionFoam`

```
// the main while loop in rk4projectionFoam.C
#include "CreatePoissonMatrix.H"

while (runTime.run())
  {
      #include "readTimeControls.H"
      #include "setDeltaT.H"
      runTime++;

      Uold = U; Uc = U; dt=runTime.deltaT();     // store old variables

      phi = (fvc::interpolate(U) & mesh.Sf());  // evaluate flux
```

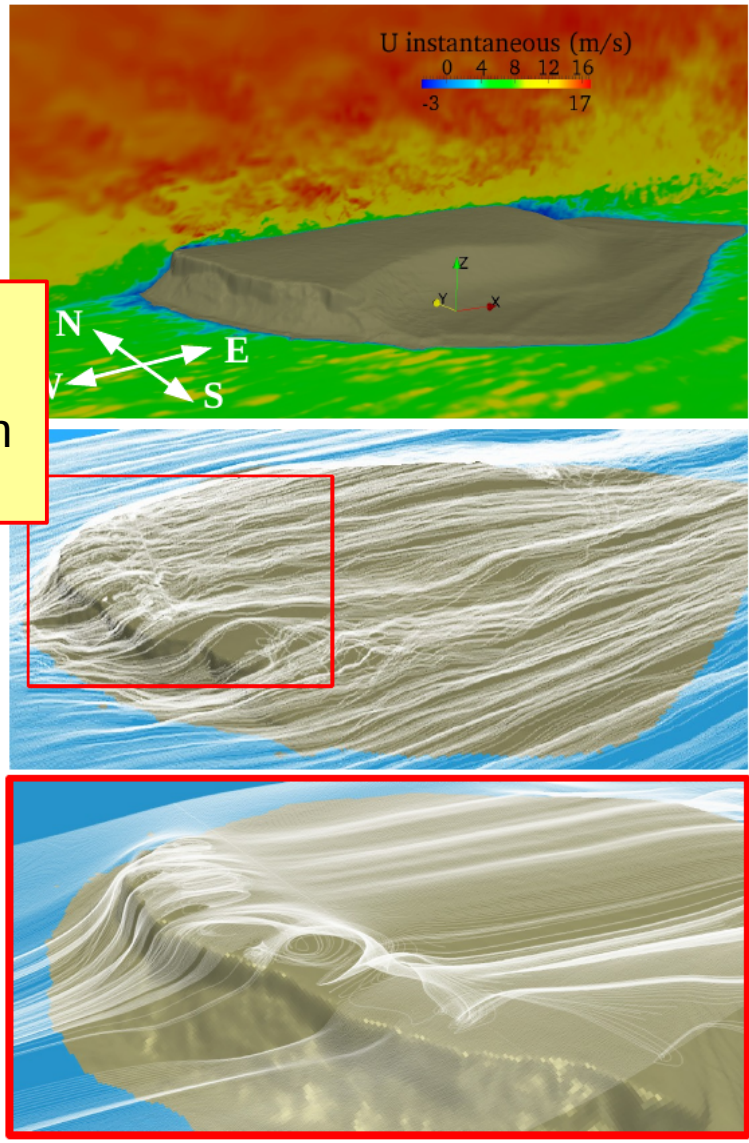
- 4th order explicit Runge-Kutta time integration
- 2nd order explicit finite volumes space
- Projection method (Hirsch) for pressure correction

```
      phi = (fvc::interpolate(U) & mesh.Sf());;
      dU  = dt*(fvc::laplacian(turbulence->nuEff(), U) - fvc::div(phi,U) );
      Uc  = Uc +a3*dU; U = Uold + dU;
      #include "PressureCorrection.H"

      phi = (fvc::interpolate(U) & mesh.Sf());
      dU  = dt*(fvc::laplacian(turbulence->nuEff(), U) - fvc::div(phi,U) );
      Uc  = Uc +a4*dU; U = Uc;                   // set U = Uc
      #include "PressureCorrection.H"            // and remove divergence

      phi = (fvc::interpolate(U) & mesh.Sf());
      turbulence->correct();                     // SGS-model updated

      runTime.write();
    }
```

```
// PressureCorrection.H
U.correctBoundaryConditions();
solve(pEqn==fvc::div(U));
U = U - fvc::grad(p);
U.correctBoundaryConditions();
```

```
// CreatePoissonMatrix.H
fvScalarMatrix pEqn
(
 fvm::laplacian(p)
);
pEqn.setReference(pRefCell, pRef);
```

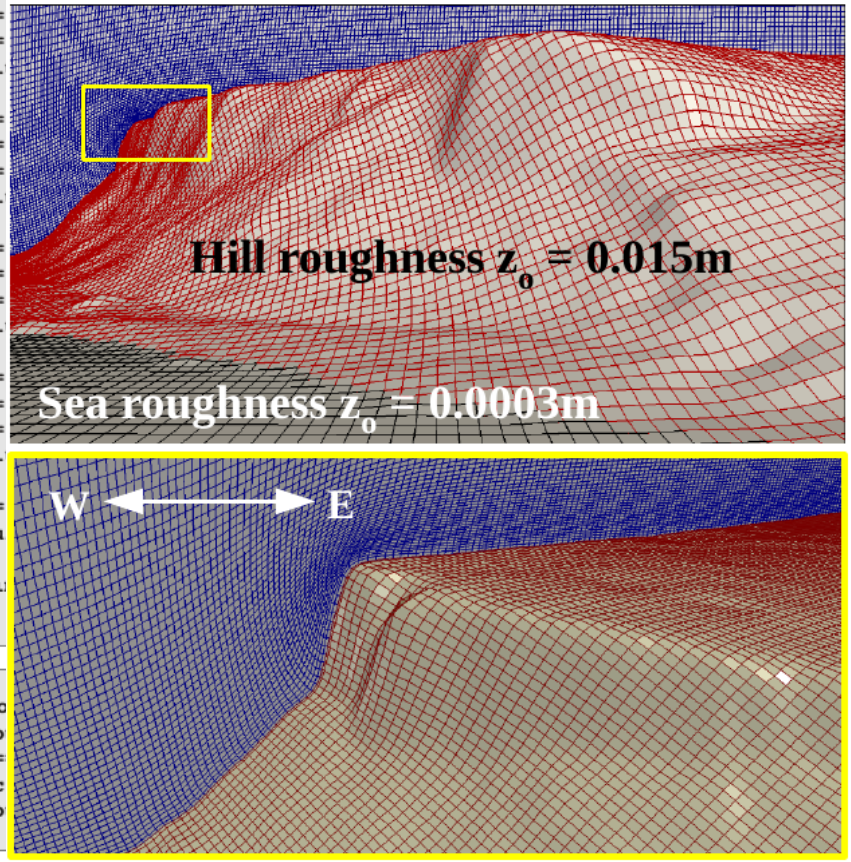
Vuorinen et al. Computers & Fluids (2014)
Vuorinen et al. Advances in Engineering Software (2015)



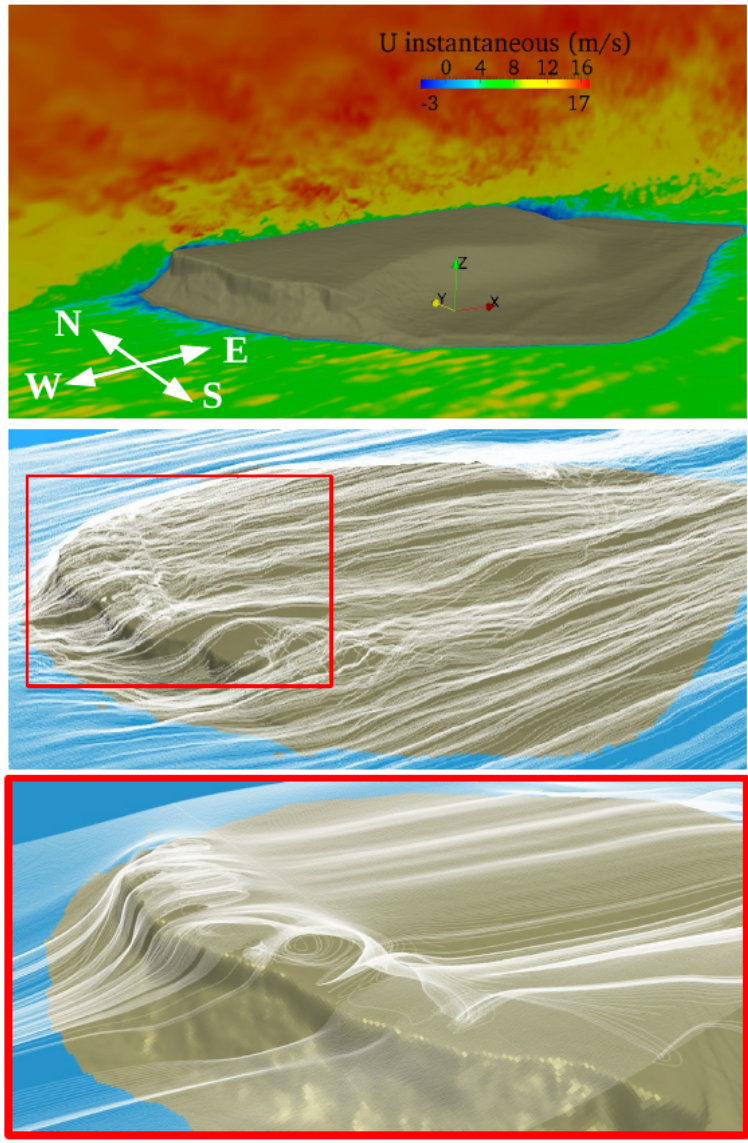
Essentials of OpenFOAM top-level solver `rk4projectionFoam`

```
// the main while loop in rk4projectionFoam.C
#include "CreatePoissonMatrix.H"

while (runTime.run())
  {
      #include "readTimeControls.H"
      #include "setDeltaT.H"
      runTime++;

      Uold = U; Uc = U; dt=runTime.deltaT();    // store old variables

      phi = (fvc::interpolate(U) & mesh.Sf());  // evaluate flux
      dU  =
      Uc  =
      #incl

      phi =
      dU  =
      Uc  =
      #incl

      phi =
      dU  =
      Uc  =
      #incl

      phi =
      dU  =
      Uc  =
      #incl

      phi =
      turbu

      runTi
    }
```

```
// PressureCo
U.correctBo
solve(pEqn=
U = U - fvc
U.correctBo
```



Vuorinen et al. Computers & Fluids (2014)
Vuorinen et al. Advances in Engineering Software (2015)



Essentials of OpenFOAM top-level solver `rk4projectionFoam`

File name: rk4projectionFoam.C

```
// the main while loop in rk4projectionFoam.C
#include "CreatePoissonMatrix.H"

while (runTime.run())
  {
      #include "readTimeControls.H"
      #include "setDeltaT.H"
      runTime++;

      Uold = U; Uc = U; dt=runTime.deltaT();     // store old variables

      phi = (fvc::interpolate(U) & mesh.Sf());  // evaluate flux
      dU  = dt*(fvc::laplacian(turbulence->nuEff(), U)- fvc::div(phi,U) );
      Uc  = Uc +a1*dU; U = Uold + 0.5*dU;       // accumulate to Uc
      #include "PressureCorrection.H"           // remove divergence from U

      phi = (fvc::interpolate(U) & mesh.Sf());
      dU  = dt*(fvc::laplacian(turbulence->nuEff(), U) - fvc::div(phi,U) );
      Uc  = Uc +a2*dU; U = Uold + 0.5*dU;
      #include "PressureCorrection.H"

      phi = (fvc::interpolate(U) & mesh.Sf());;
      dU  = dt*(fvc::laplacian(turbulence->nuEff(), U) - fvc::div(phi,U) );
      Uc  = Uc +a3*dU; U = Uold + dU;
      #include "PressureCorrection.H"

      phi = (fvc::interpolate(U) & mesh.Sf());
      dU  = dt*(fvc::laplacian(turbulence->nuEff(), U) - fvc::div(phi,U) );
      Uc  = Uc +a4*dU; U = Uc;                  // set U = Uc
      #include "PressureCorrection.H"           // and remove divergence

      phi = (fvc::interpolate(U) & mesh.Sf());
      turbulence->correct();                    // SGS-model updated

      runTime.write();
    }
```

```
// PressureCorrection.H
U.correctBoundaryConditions();
solve(pEqn==fvc::div(U));
U = U - fvc::grad(p);
U.correctBoundaryConditions();
```

File name: PressureCorrection.C

```
// CreatePoissonMatrix.H
fvScalarMatrix pEqn
(
 fvm::laplacian(p)
);
pEqn.setReference(pRefCell, pRef);
```

File name: CreatePoissonMatrix.C

Vuorinen et al. Computers & Fluids (2014)
Vuorinen et al. Advances in Engineering Software (2015)

- E.g. `pimpleFoam` or `icoFoam` are “top-level” solvers
- A top-level solver (e.g. `rk4projectionFoam`) would mean: storing `rk4projectionFoam.C` + some other files in folder `/rk4projectionFoam/` and compiling the solver into executable with `wmake`
- Part of top-level solver code can be appended to header files (.H extension)
- Important: configure the `/Make/` folder accordingly

Essentials of OpenFOAM top-level solver `rk4projectionFoam`

Vuorinen et al. Computers & Fluids (2014)
Vuorinen et al. Advances in Engineering Software (2015)

```
// the main while loop in rk4projectionFoam.C
#include "CreatePoissonMatrix.H"

while (runTime.run())
  {
      #include "read
      #include "setD
      runTime++;

      Uold = U; Uc = U; dt=runTime.deltaT();     // store old variables

      phi = fvc::interpolate(U) & mesh.Sf();
      dU = dt*(fvc::laplacian(nu, U) - fvc::div(phi,U));
      #include "PressureCorrection.H"            // remove divergence from U

      phi = (fvc::interpolate(U) & mesh.Sf());
      dU  = dt*(fvc::laplacian(turbulence->nuEff(), U) - fvc::div(phi,U) );
      Uc  = Uc +a2*dU; U = Uold + 0.5*dU;
      #include "PressureCorrection.H"

      phi = (fvc::interpolate(U) & mesh.Sf());;
      dU  = dt*(fvc::laplacian(turbulence->nuEff(), U) - fvc::div(phi,U) );
      Uc  = Uc +a3*dU; U = Uold + dU;
      #include "PressureCorrection.H"

      phi = (fvc::interpolate(U) & mesh.Sf());
      dU  = dt*(fvc::laplacian(turbulence->nuEff(), U) - fvc::div(phi,U) );
      Uc  = Uc +a4*dU; U = Uc;                   // set U = Uc
      #include "PressureCorrection.H"            // and remove divergence

      phi = (fvc::interpolate(U) & mesh.Sf());
      turbulence->correct();                     // SGS-model updated

      runTime.write();
    }
```

```
phi = fvc::interpolate(U) & mesh.Sf();
dU = dt*(fvc::laplacian(nu, U) - fvc::div(phi,U));
```

```
// PressureCorrection.H
U.correctBoundaryConditions();
solve(pEqn==fvc::div(U));
U = U - fvc::grad(p);
U.correctBoundaryConditions();
```

```
// CreatePoissonMatrix.H
fvScalarMatrix pEqn
(
 fvm::laplacian(p)
);
pEqn.setReference(pRefCell, pRef);
```

Table 1
Examples of differential operator types in OpenFOAM®. The `fvc` syntax declares the operators to be explicit i.e. to be evaluated with known values [26].

Operator	Syntax	Output for a scalar	Output for a vector
∇	`fvc::grad(F)`	`volVectorField`	`volTensorField`
$\nabla\times$	`fvc::curl(F)`	–	`volVectorField`
Δ	`fvc::laplacian(F)`	`volScalarField`	`volVectorField`

Table 2
Examples of field data types in OpenFOAM® [26].

Field name	Variable	Data type	Variable values defined at
Velocity	`U`	`volVectorField`	Cell center
Pressure	`p`	`volScalarField`	Cell center
Rate of strain	`S`	`volTensorField`	Cell center
Velocity flux	`phi`	`surfaceScalarField`	Cell face center

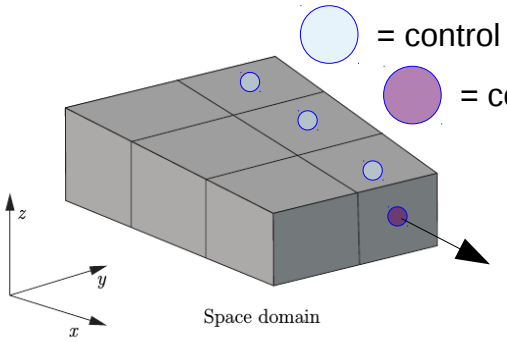
FOAM – OpenFOAM provides extremely handy data types for storing data – Snapshots from `CreateFields.H` showing how fields `p/U` can be created

```
volScalarField p
  (
      IOobject
      (
         "p",
         runTime.timeName(),
         mesh,
         IOobject::MUST_READ,
         IOobject::NO_WRITE
      ),
      p
  );
```

```
volVectorField U
 (
     IOobject
     (  "U",
        runTime.timeName(),
        mesh,
        IOobject::MUST_READ,
        IOobject::AUTO_WRITE
        ),
        mesh
   );
```

Vuorinen et al. Computers & Fluids (2014)

Let us interpret the following lines from math viewpoint



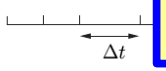


Figure 2.1: Discretisation of the solution domain

`dU` is a **`volVectorField - "vol"`** because it is defined at cell volume centers similar to **`U`**

```
dU = dt*(fvc::laplacian(nu, U) - fvc::div(phi,U));
```

`phi` is a **`surfaceScalarField - "surface"`** because it is defined at cell surfaces

```
phi = fvc::interpolate(U) & mesh.Sf();
```

$$\mathbf{C}_{fvm} = \frac{1}{\Omega}\int_{\Omega} \nabla \cdot (\mathbf{uu}) d\Omega = \frac{1}{\Omega}\int_{\partial\Omega} \mathbf{uu} \cdot \mathbf{n}\, dS$$

$$\approx \frac{1}{\Omega}\sum_{f=1}^{N} \mathbf{u}_f \mathbf{u}_f^c \cdot \mathbf{n}_f\, dS_f,$$

About interpolation and flux limiting

$$\mathbf{C}_{fvm} = \frac{1}{\Omega}\int_{\Omega} \nabla \cdot (\mathbf{uu}) d\Omega = \frac{1}{\Omega}\int_{\partial\Omega} \mathbf{uu}\cdot\mathbf{n}\, dS$$

$$\approx \frac{1}{\Omega}\sum_{f=1}^{N} \mathbf{u}_f \mathbf{u}_f^c \cdot \mathbf{n}_f\, dS_f,$$

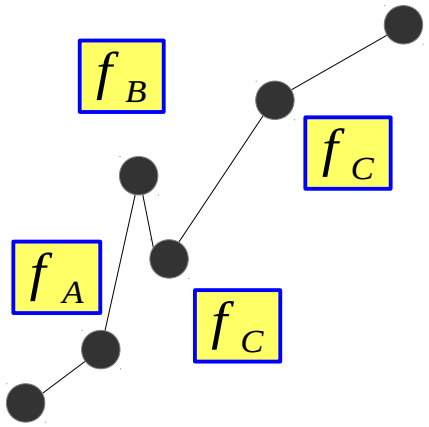
This interpolation can be carried out in various ways:

`linear`
`GammaV`
`limitedLinearV`
`vanLeer`

FLUX LIMITERS

This interpolation is very commonly carried out by:

`linear`

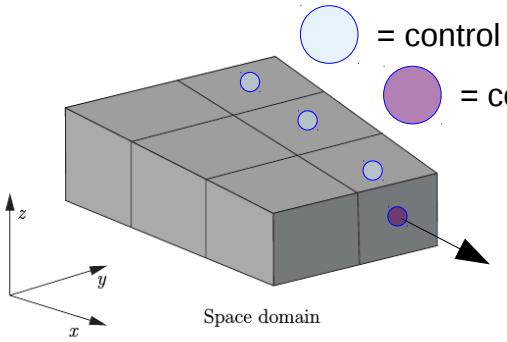


Aalto University
School of Engineering

FLUX LIMITING is useful interpolation technique close to high oscillating parts: how would you numerically compute flux/derivative around point B?
BENEFIT: more stable and robust simulation!

Vuorinen et al. Computers & Fluids (2014)

Let us interpret the following lines from math viewpoint



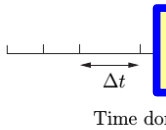


Figure 2.1: Discretisation of the solution domain

`dU` is a **`volVectorField - "vol"`** because it is defined at cell volume centers similar to **`U`**

```
dU = dt*(fvc::laplacian(nu, U) - fvc::div(phi,U));
```

`phi` is a **`surfaceScalarField - "surface"`** because it is defined at cell surfaces

```
phi = fvc::interpolate(U) & mesh.Sf();
```

Outer normal vectors multiplied by surface area of face in the mesh

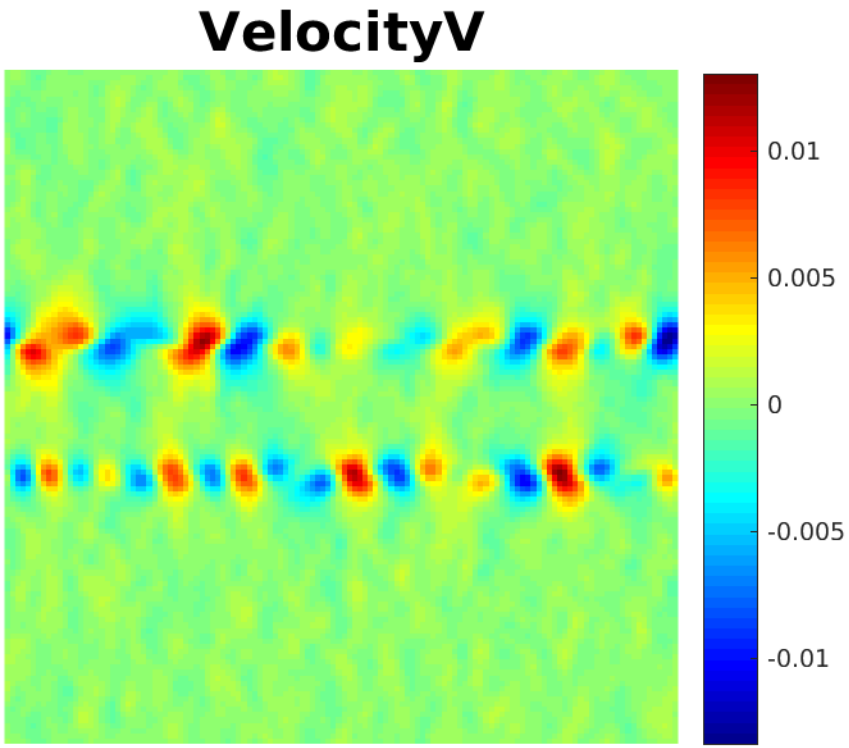
dot product

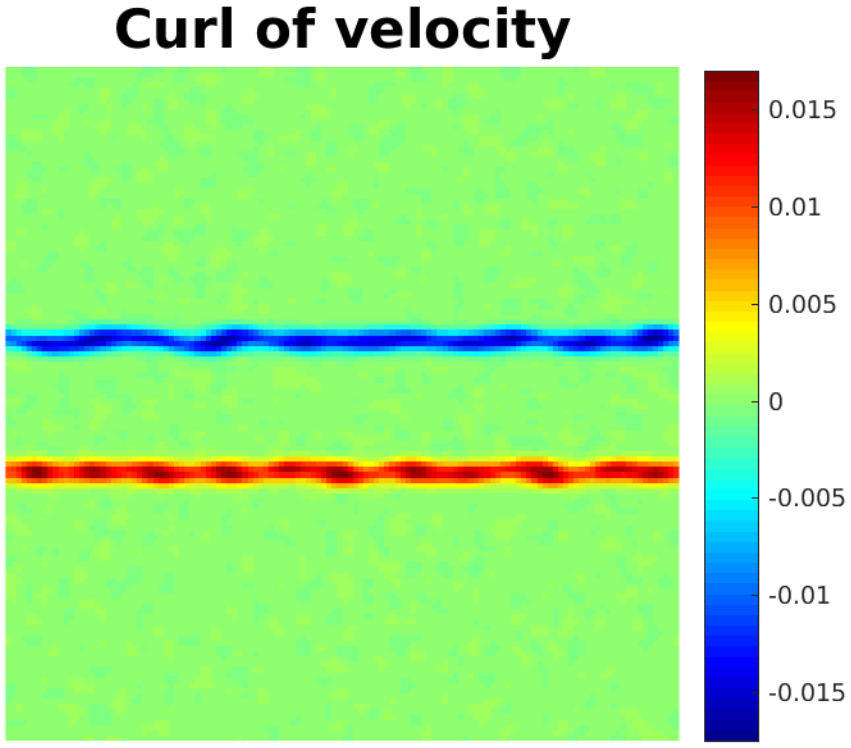
$$\mathbf{C}_{fvm} = \frac{1}{\Omega}\int_{\Omega} \nabla \cdot (\mathbf{uu}) d\Omega = \frac{1}{\Omega}\int_{\partial\Omega} \mathbf{uu} \cdot \mathbf{n}\, dS$$

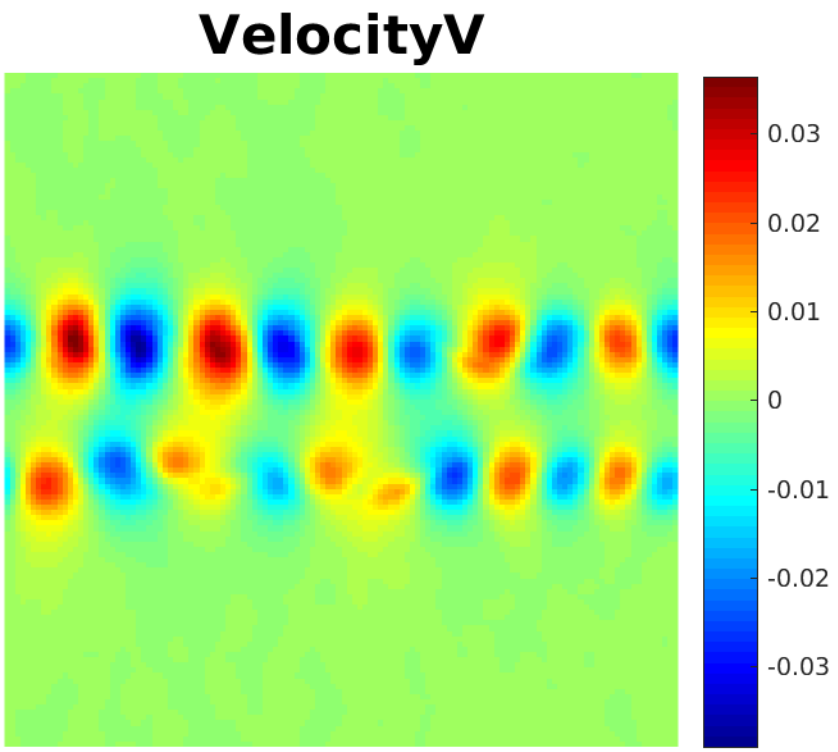
$$\approx \frac{1}{\Omega}\sum_{f=1}^{N} \mathbf{u}_f \mathbf{u}_f^c \cdot \mathbf{n}_f\, dS_f,$$

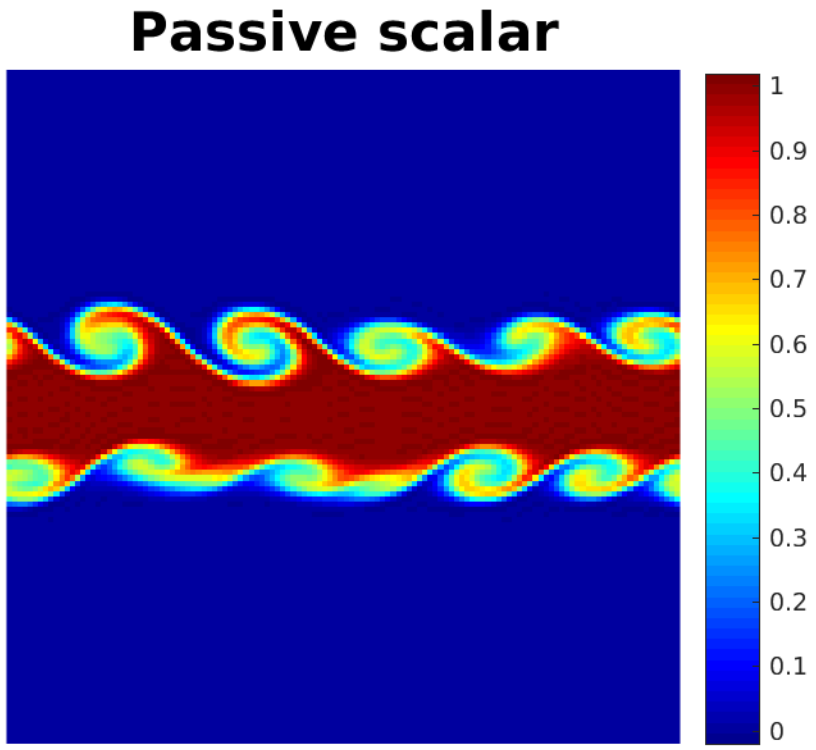
Part 5: A word about HW3

Some random noise added in the beginning to y-component









https://www.youtube.com/watch?v=ELaZ2x42dkU

Thank you for your attention!