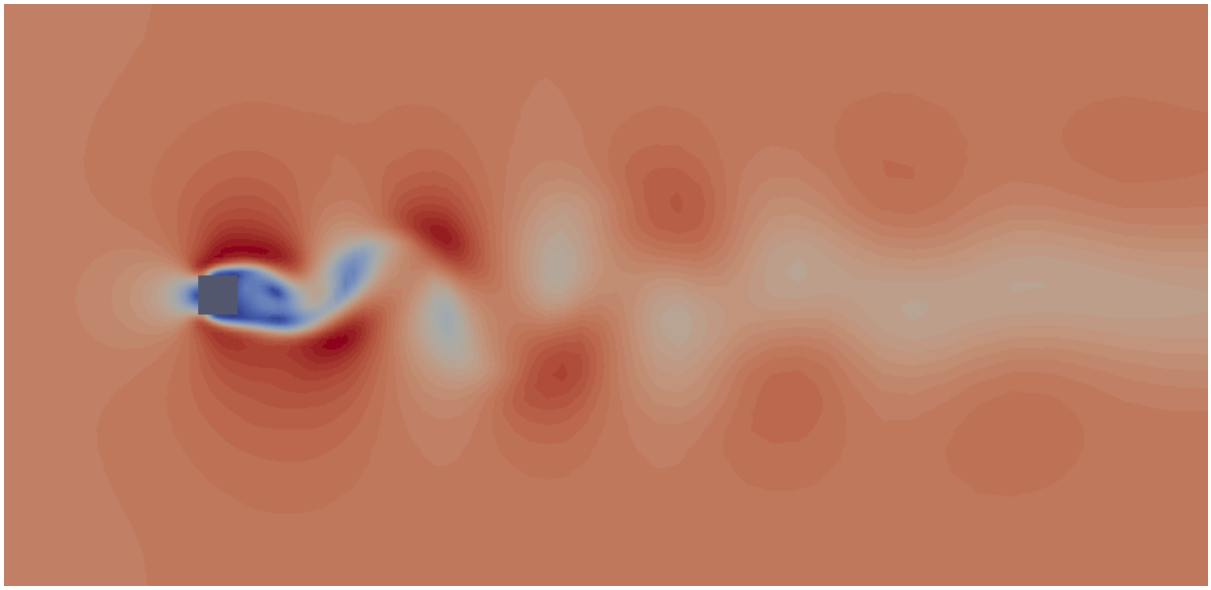


pyCFD Documentation

Release 0.1

Bence Somogyi

June 09, 2014

CHAPTER

ONE

ABOUT THIS DOCUMENT

This documentation is a result of solving the programming example for the lecture *Numerical Methods in Fluid Mechanics and Heat Transfer (LV-Nr.: 321.023 VO, WS 2012/13).*

It was written by Bence Somogyi (matr #: 1231273, somogyi@ivt.tugraz.at (somogyi@ivt.tugraz.at))

CHAPTER

TWO

DESCRIPTION OF THE PROJECT

English translation of the descrition is given as:

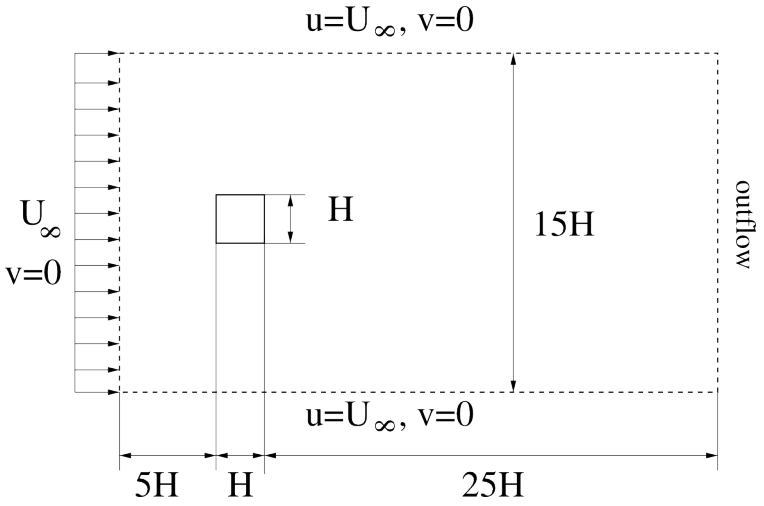
Numerical simulation of laminar flow around an infinitely long cylinder of square cross section (with edge length H) has to be performed. The problem can be described by the two-dimensional laminar incompressible ($\rho = const$) Navier-Stokes equations:

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0$$

$$\frac{\partial u}{\partial t} + u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial x} + \nu\left(\frac{\partial^2 u}{\partial x^2} + \frac{\partial^2 u}{\partial y^2}\right)$$

$$\frac{\partial v}{\partial t} + u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y} = -\frac{1}{\rho}\frac{\partial p}{\partial y} + \nu\left(\frac{\partial^2 v}{\partial x^2} + \frac{\partial^2 v}{\partial y^2}\right)$$

Dimensions of the computational domain and the associated boundary conditions are shown below:



2.1 Tasks

- solution of the equations in a non-dimensional form using H and U_∞ as characteristic values.
- second order finite-volume discretization in space and time (explicit)
- implicit solution of diffusion terms with suitable solution methods for linear systems
- numerical simulation of the flow field for $Re = U_\infty * H/\nu = 200$ to reach a statistcal steady state solution. Continuity shall be ensured via the solution of the Poisson equation. The Poisson equation shall be solved by a suitable dircet method.
- documentation of all the actions and results.

CHAPTER

THREE

ABOUT THE CODE

Prior to solving the example the author has set a list of criterion about what and how the solution should serve learning and self developement. These criterion resulted in the following list of features:

- a general purpose 3D code/library was written (*modindex*)
- colocated grid arrangement was used
- code was written in python in an object oriented fashion
- bottlenecks of the code were re-implemented in C via Cython
- the library supports writing the mathematical operators in the governing equations in a style ispired by OpenFOAM
- the output is provided in the file formats of the vtk library allowing the use of Paraview for post-processing

CHAPTER

FOUR

TESTING THE OPERATORS

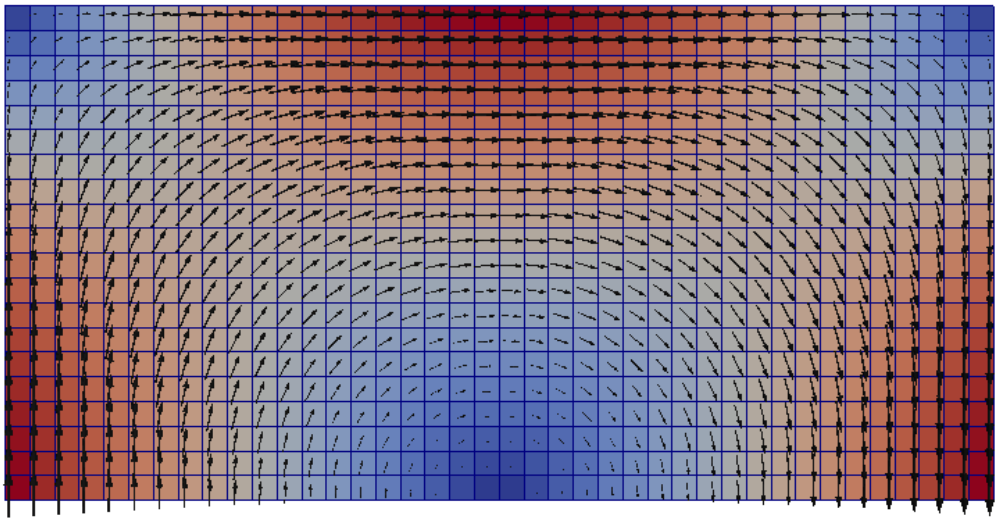
4.1 Convection

The convection problem is described by the following equation:

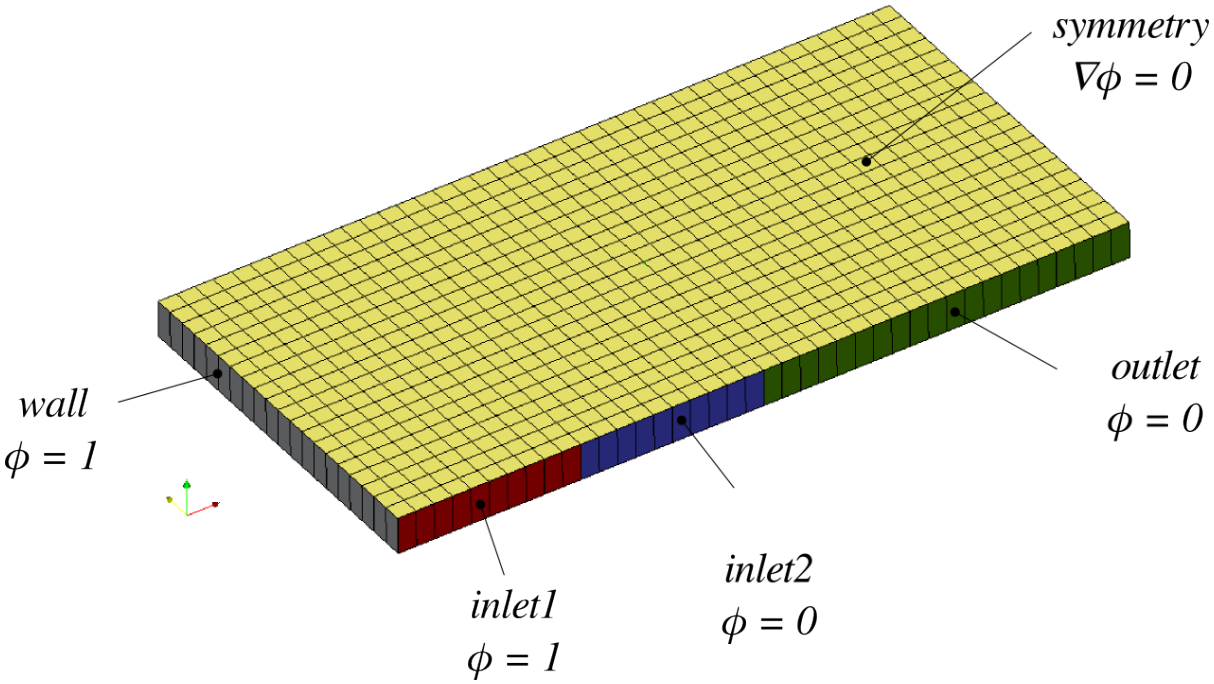
$$\frac{\partial \phi}{\partial t} + \nabla \cdot (\vec{v}\phi) = 0$$

This equation is solved for the Simth-Hutton problem. In this problem a constant velocity field is prescribed which is given by:

$$u = 2y(1 - x^2)$$
$$v = -2x(1 - y^2)$$



The mesh and boundary conditions for ϕ are shown on the next image:



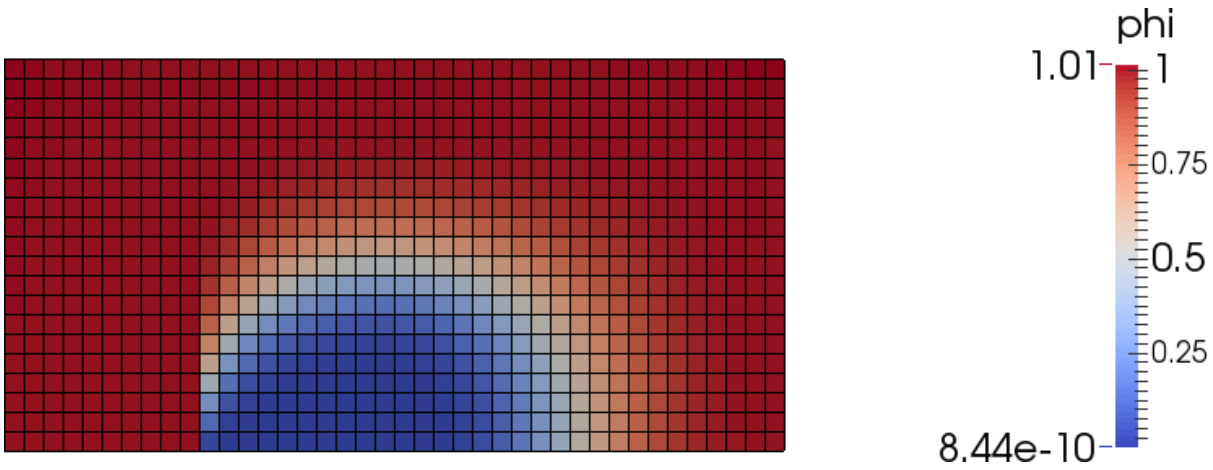
The following parameters were set for the calculation of this problem:

- timestep: 0.05 s
- calculation time: 4 s
- initial value for ϕ: 1

The whole *script for solving the Smith-Hutton problem* (page 32).

Divergence was calculated with `pyCFD_operators.implicit_operators.Divergence` (page 77). The problem was solved with the three available interpolation schemes: upwind (UDS), MINMOD and STOIC.

Solution of the Smith-Hutton problem using UDS:



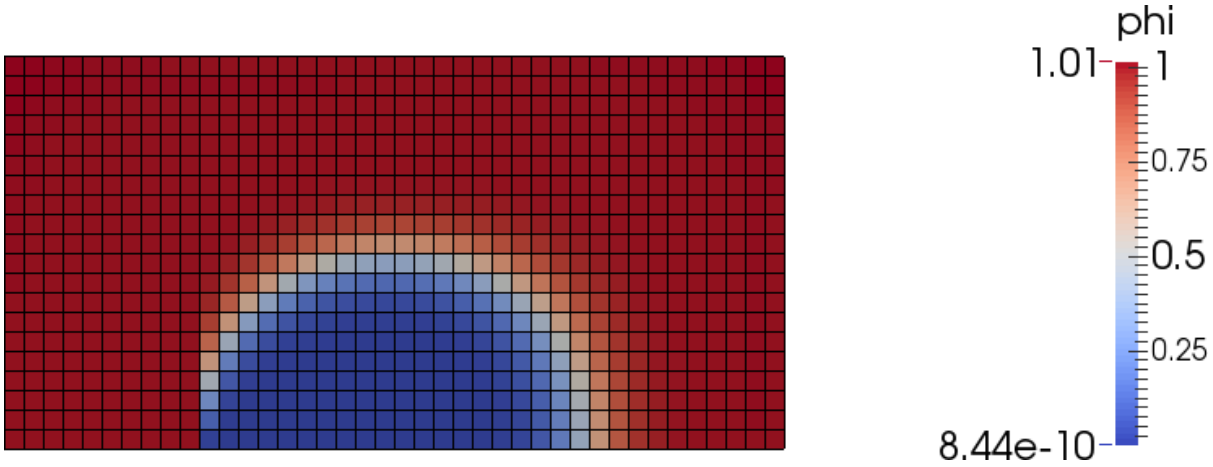
Solution of the Smith-Hutton problem using MINMOD:

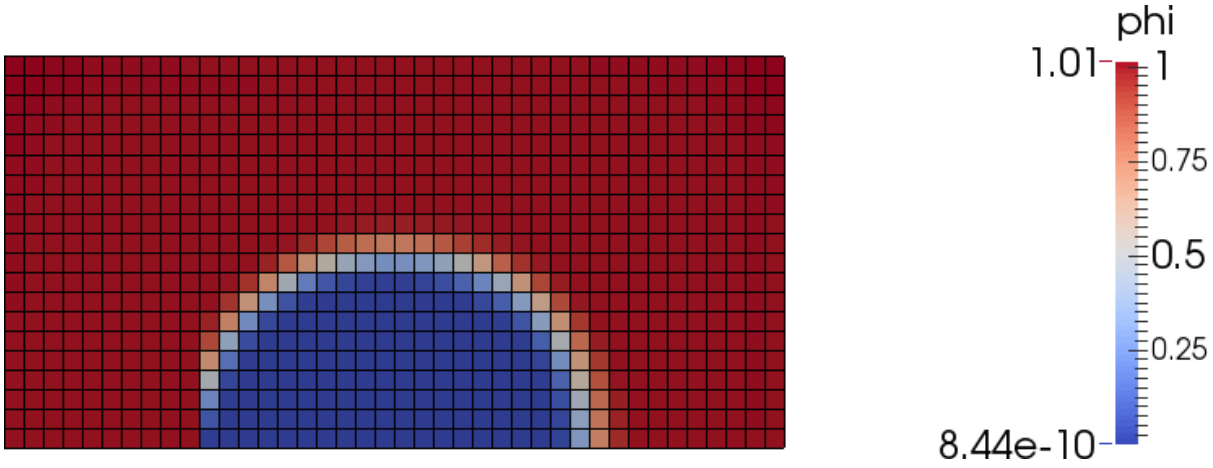
Solution of the Smith-Hutton problem using STOIC:

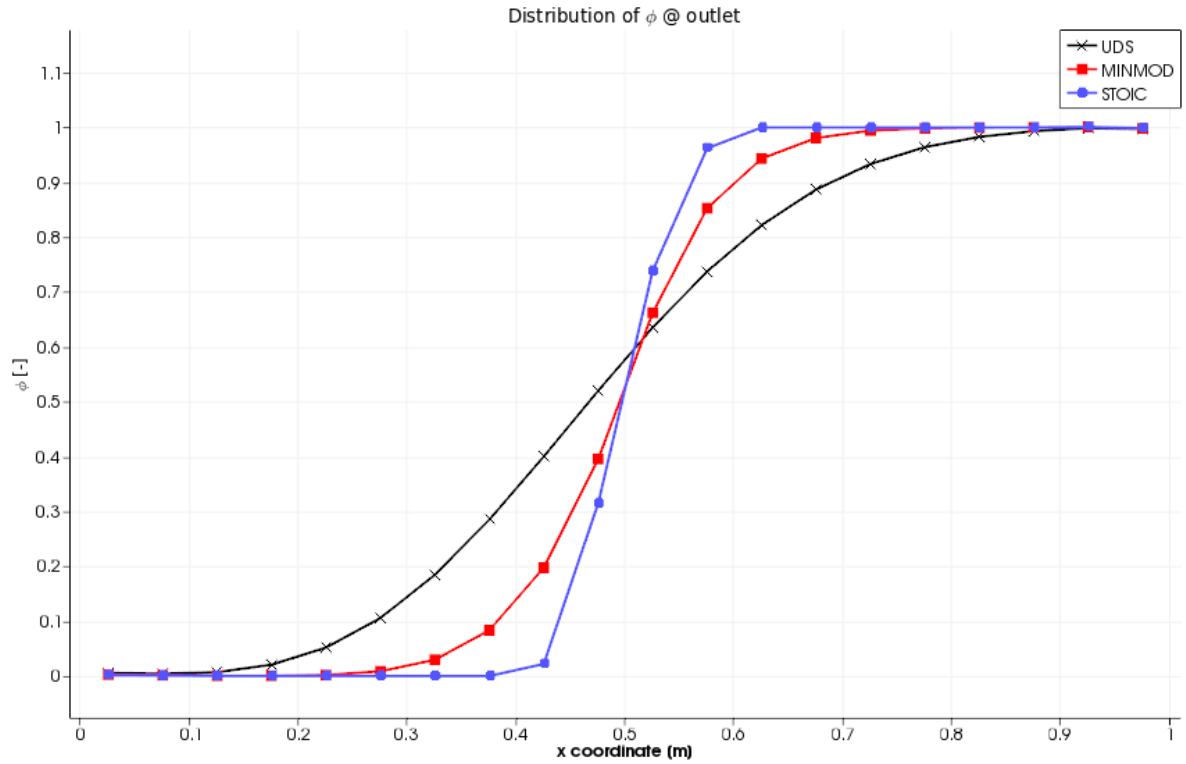
Comparison of distributions of ϕ at the outlet:

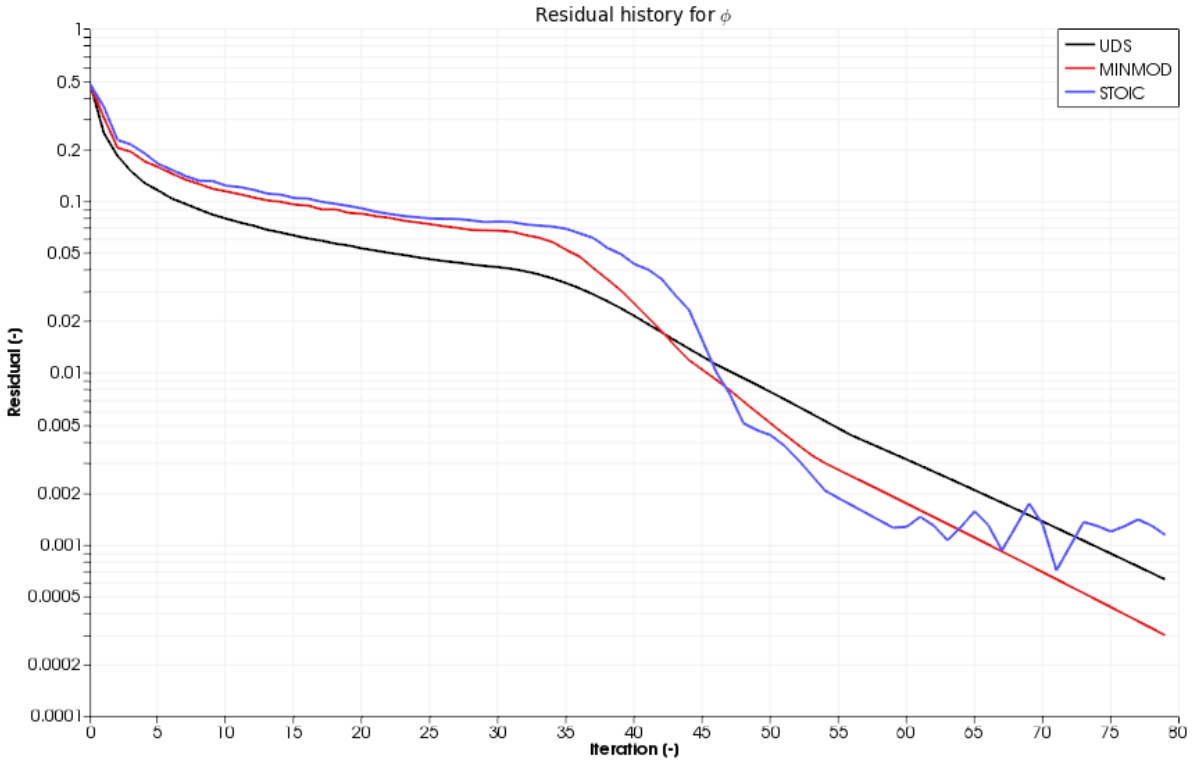
Convergence history for the three cases:

Residual was calculated as $max_i \left(\phi_i^{new} - \phi_i^{old}\right)$, where i=1... # of cells.


phi
1.01
1
0.75
0.5
0.25
8.44e-10


phi
1.01
1
0.75
0.5
0.25
8.44e-10


Distribution of ϕ @ outlet
UDS
MINMOD
STOIC
ϕ [-]
x coordinate (m)



4.2 Diffusion

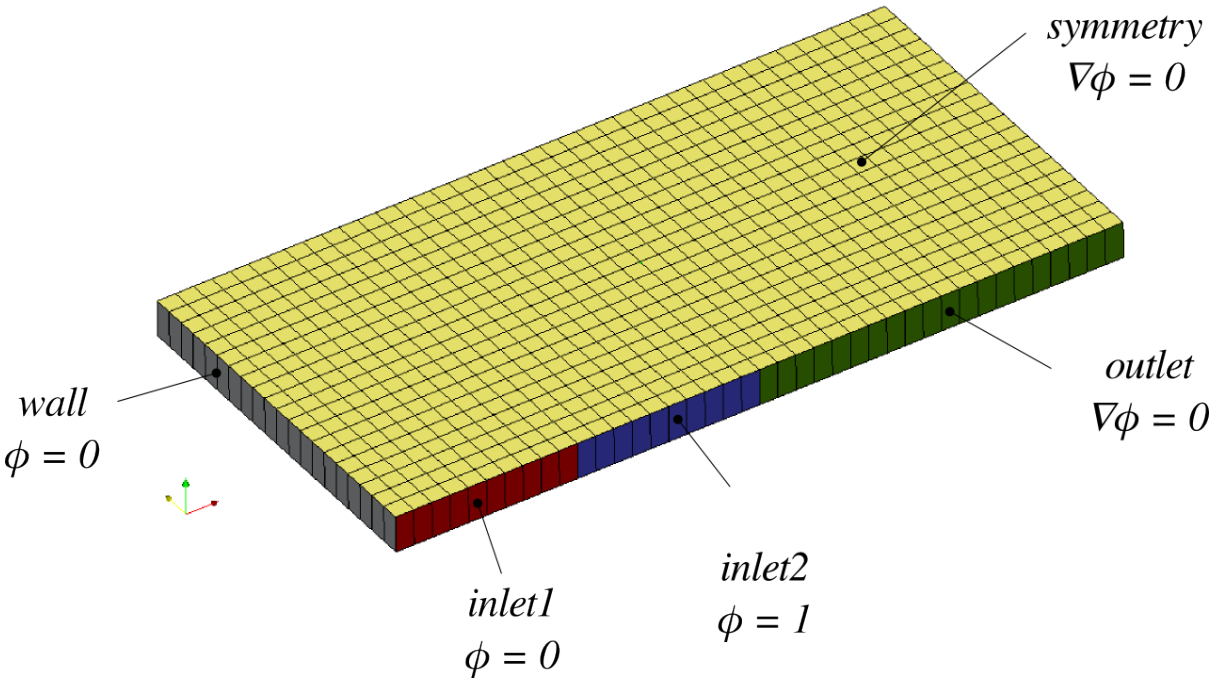
The diffusion problem is described by the following equation:

$$-\nabla \cdot (\Gamma \nabla \phi) = S$$

In this test case $S = 0$ and a transient term is applied that the explicit operator can be also tested. The final equation solved is therefore:

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

The same geometry is used as in the Smith-Hutton problem with different boundary conditions:



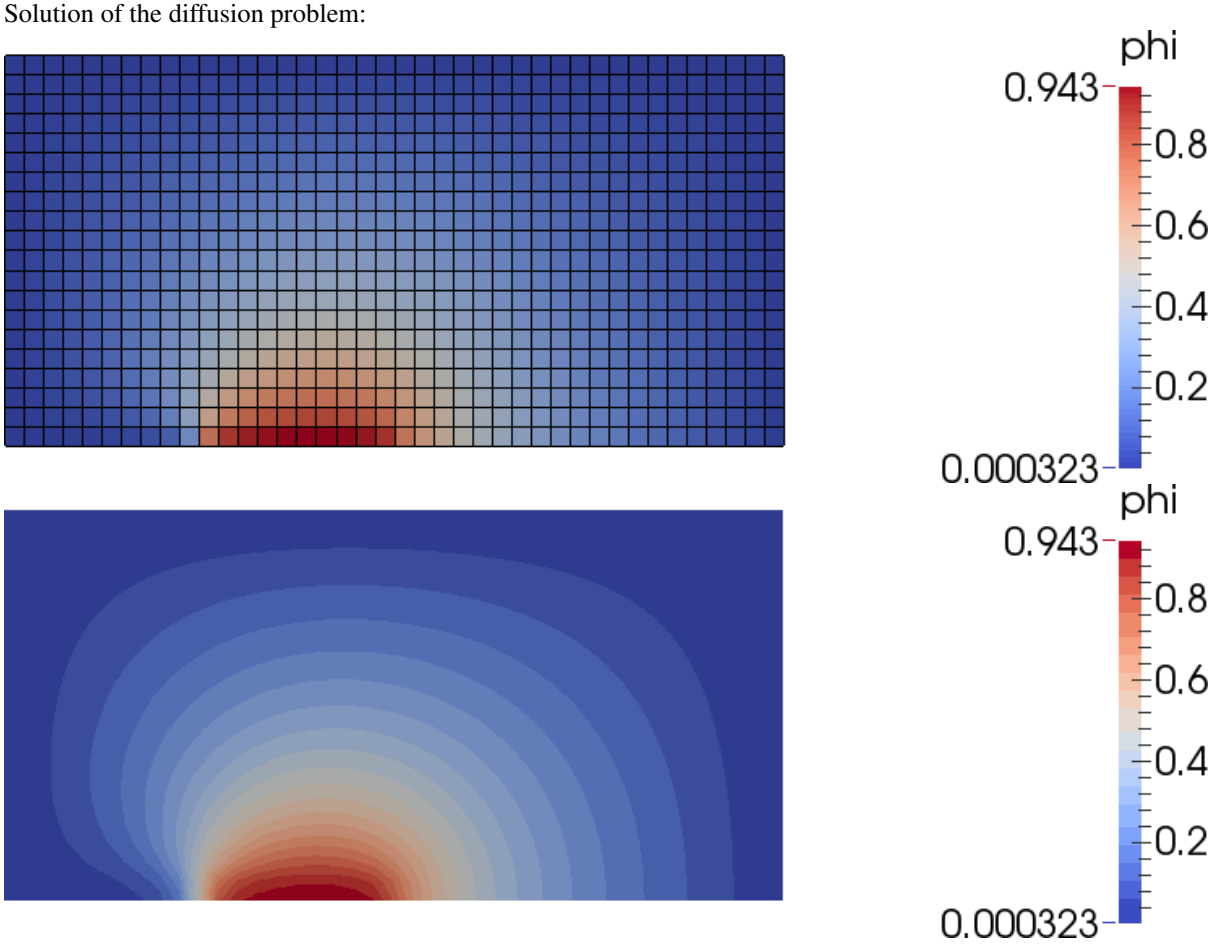
The following parameters were set for the calculation of the problem:

- timestep: 0.1 s
- calculation time: 1 s
- initial value for ϕ: 0

The whole *script for solving the diffusion problem* (page 34).

Diffusion was calculated with `pyCFD_operators.implicit_operators.Laplace` (page 78).

Solution of the diffusion problem:



Convergence history for the diffusion case:

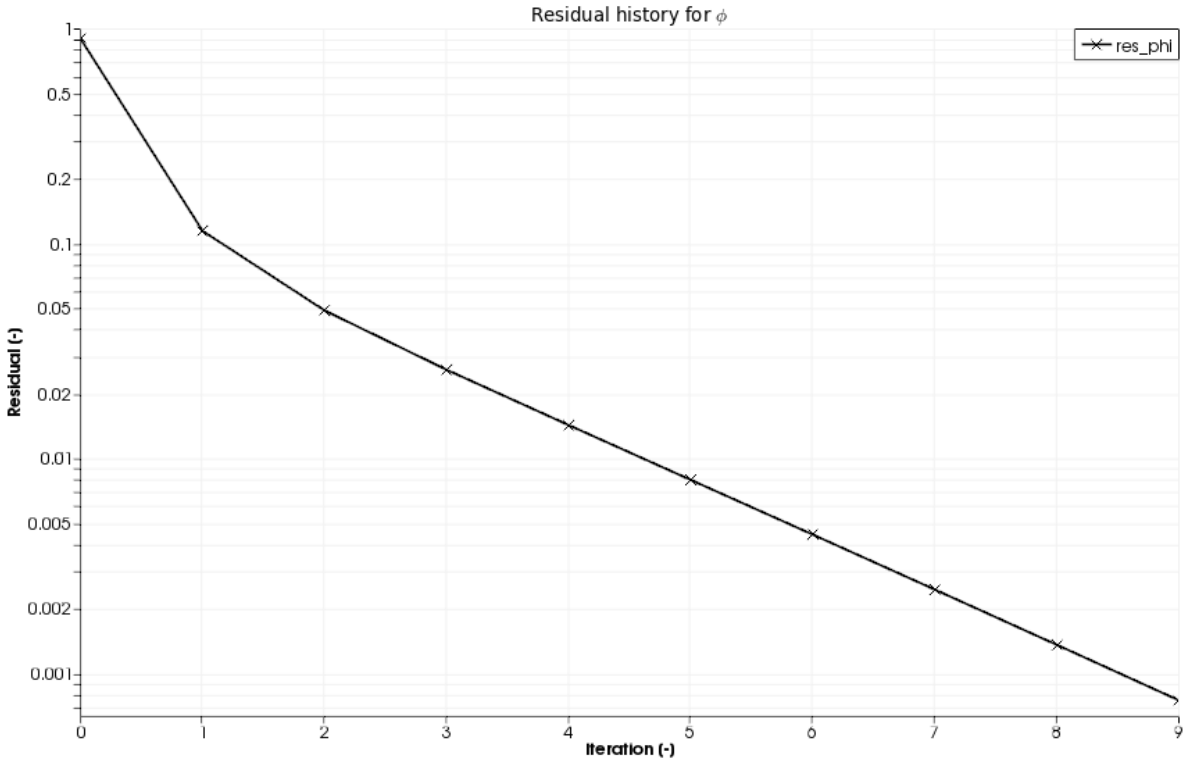
Residual was calculated as $max_i \left(\phi_i^{new} - \phi_i^{old}\right)$, where i=1... # of cells.

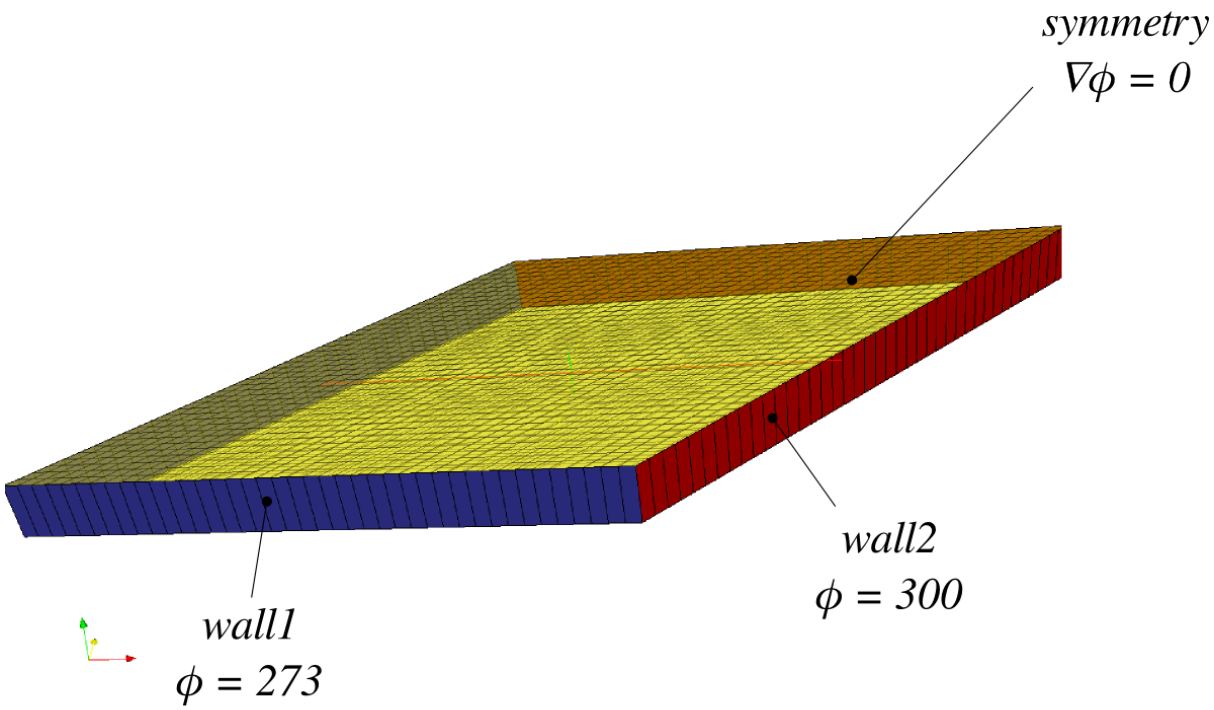
4.3 Diffusion with correction for non conjunctionality

The following diffusion equation is solved:

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

The mesh and boundary conditions for ϕ are shown on the next image:



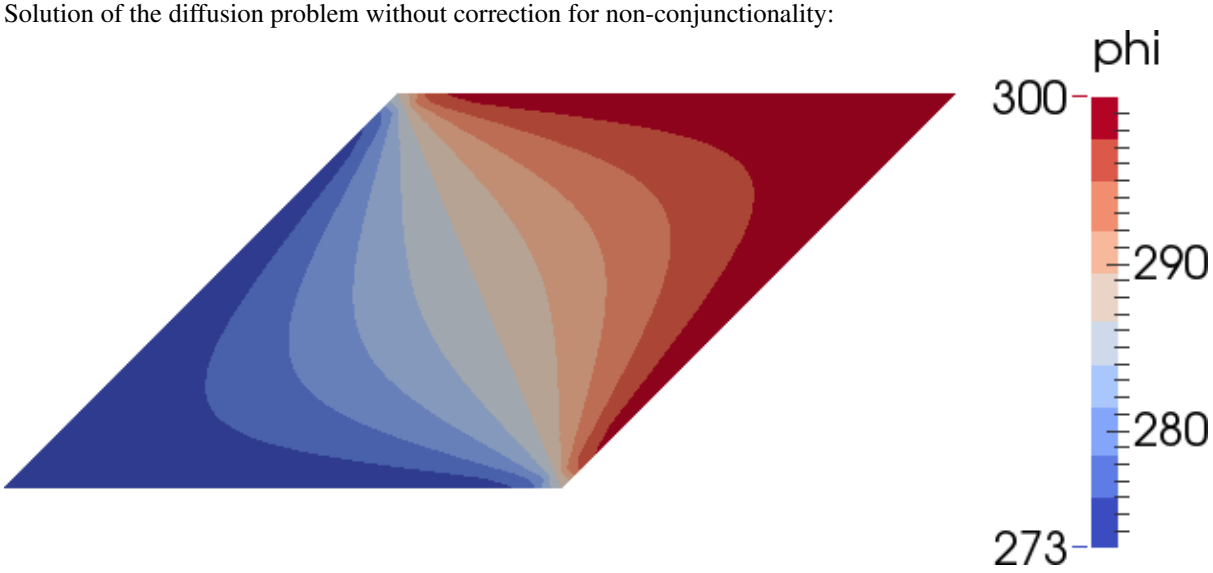


The following parameters were set for the calculation of the problem:

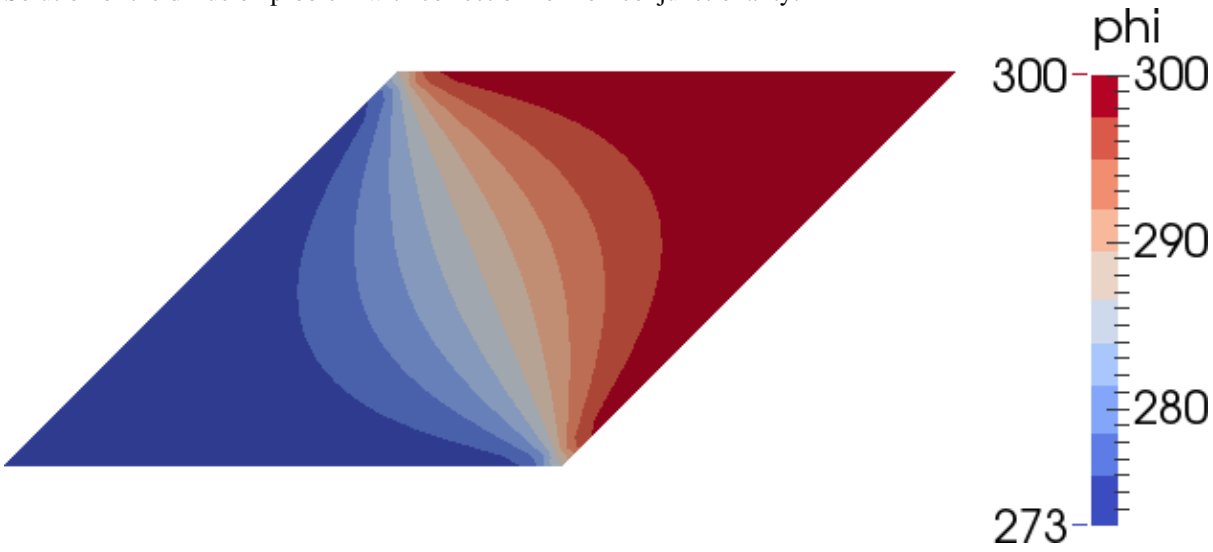
- timestep: 0.1 s
- calculation time: 3 s
- initial value for ϕ: 0
- Γ = 0.01 (has the effect of under relaxation for this case)

The whole *script for solving the diffusion problem in the inclined block* (page 35).

Solution of the diffusion problem without correction for non-conjunctionality:

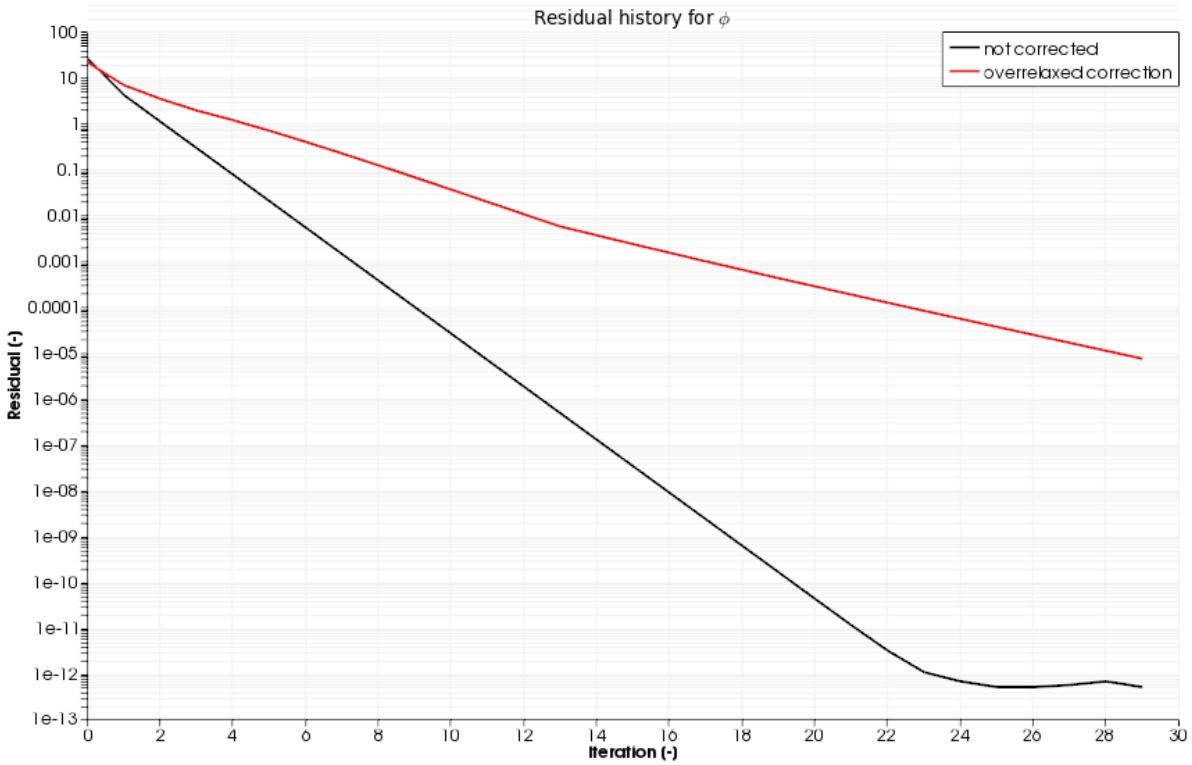


Solution of the diffusion problem with correction for non-conjunctionality:



Convergence history for the diffusion case:

Residual was calculated as $max_i \left(\phi_i^{new} - \phi_i^{old}\right)$, where i=1... # of cells.


Residual history for ϕ
not corrected
overrelaxed correction
Residual (-)
100
10
1
0.1
0.01
0.001
0.0001
1e-05
1e-06
1e-07
1e-08
1e-09
1e-10
1e-11
1e-12
1e-13
0
2
4
6
8
10
12
14
16
18
20
22
24
26
28
30
Iteration (-)

CHAPTER

FIVE

THE NON-DIMENSIONAL EQUATIONS

5.1 Continuity

The continuity equation for incompressible flow in 3D can be written using Einstein's summation as:

$$\frac{\partial u_i}{\partial x_i} = 0 \tag{5.1}$$

As a first step the non dimensional coordinates shall be defined:

$$x_i^* := \frac{x_i}{H} \tag{5.2}$$

, followed by the non dimensional velocities:

$$u_i^* := \frac{u_i}{U_\infty} \tag{5.3}$$

Substituting (5.2) and (5.3) into (5.1) leads to:

$$\frac{U_\infty}{H} \cdot \frac{\partial u_i^*}{\partial x_i^*} = 0 \tag{5.4}$$

, which can be simplified to:

$$\frac{\partial u_i^*}{\partial x_i^*} = 0 \tag{5.5}$$

The continuity equation in vector form:

$$\nabla \vec{v^*} = 0 \tag{5.6}$$

, where

$$\vec{v^*} = \begin{pmatrix} u^* \\ v^* \\ w^* \end{pmatrix} \tag{5.7}$$

5.2 Momentum

5.2.1 x_i component

The xi direction momentum equation for incompressible flow in 3D can be written as:

$$\frac{\partial u_i}{\partial t} + u_j \frac{\partial u_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial p}{\partial x_i} + \nu \left(\frac{\partial^2 u_i}{\partial x_j^2} \right) \tag{5.8}$$

Non-dimensional variables for spatial coordinates are defined in (5.2). New variables shall be defined for non dimensional time and pressure:

$$t^* := \frac{t \cdot U_\infty}{H} \tag{5.9}$$

$$p^* := \frac{p}{\rho \cdot U_\infty^2} \tag{5.10}$$

Substituting (5.2), (5.3), (5.9) and (5.10) into (5.8):

$$\frac{U_\infty \cdot U_\infty}{H} \cdot \frac{\partial u_i^*}{\partial t^*} + \frac{U_\infty \cdot U_\infty}{H} \cdot u_j^* \frac{\partial u_i^*}{\partial x_j^*} =$$
$$-\frac{\rho \cdot U_\infty \cdot U_\infty}{H} \cdot \frac{1}{\rho} \cdot \frac{\partial p^*}{\partial x^*} + \frac{\nu \cdot U_\infty}{H^2} \cdot \left(\frac{\partial^2 u_i^*}{\partial x_j^2} \right)$$

Dividing both sides with $\frac{U_\infty^2}{H}$, simplifying with ρ in the pressure term and substituting $Re = \frac{U_\infty \cdot H}{\nu}$ into the viscous term, we get the non-dimensional form of the momentum equation in the x_i direction:

$$\frac{\partial u_i^*}{\partial t^*} + u_j^* \frac{\partial u_i^*}{\partial x_j^*} = -\frac{\partial p^*}{\partial x^*} + \frac{1}{Re} \frac{\partial^2 u_i^*}{\partial x_j^{*2}} \tag{5.11}$$

5.2.2 3D vector form

Based on (5.11) the momentum equation can be written in vector form:

$$\frac{\partial \vec{v}^*}{\partial t^*} + \vec{v}^* \cdot \nabla \vec{v}^* = -\nabla p^* + \frac{1}{Re} \cdot \Delta \left(\vec{v}^* \right) \tag{5.12}$$

CHAPTER

SIX

THE SIMPLE ALGORITHM

The equations derived in *The non-dimensional equations* (page 12) for momentum:

$$\frac{\partial \vec{v}^*}{\partial t^*} + \vec{v}^* \cdot \nabla \vec{v}^* = -\nabla p^* + \frac{1}{Re} \cdot \Delta \left(\vec{v}^* \right)$$

and continuity:

$$\nabla \vec{v^*} = 0 \tag{6.1}$$

does not contain an equation for pressure. It should be deducted therefore here.

Writing the momentum equations discretized in space and time, using coefficients of the linear equation system:

$$a_C \cdot \vec{v^*_{pr,C}} + \sum_f \left(a_f \cdot \vec{v^*_{pr,f}} \right) = -V_C \cdot \left(\nabla p^*_{pr} \right) + V_C \cdot S_{mom} \tag{6.2}$$

, where the subscript *C* indicates the cell center values, *f* the values at the faces, *a* are the coefficients of the linear equation system and S_{mom} is the source term in the momentum equation. The superscript * is used for the non dimensional fields. Writing (6.3) for $\vec{v^*_{pr,C}}$:

$$\vec{v^*_{pr,C}} = -H \left(\vec{v^*_{pr,f}} \right) - \frac{V_C}{a_C} \cdot \left(\nabla p^*_{pr} \right) + \frac{V_C}{a_C} \cdot S_{mom} \tag{6.3}$$

, where:

$$H \left(\vec{v^*_{pr,f}} \right) = \frac{\sum_f \left(a_f \cdot \vec{v^*_{pr,f}} \right)}{a_C} \tag{6.4}$$

As the solution results in a velocity field which does not divergence free ((6.1) is not fulfilled) this solution is called a predicted velocity field (or the solution of the momentum equation the predictor step). We search for a corrective velocity and corrective pressure field

$$\vec{v^*_{final}} = \vec{v^*_{pr}} + \vec{v^*_{corr}} \tag{6.5}$$

$$p^*_{final} = p^*_{pr} + p^*_{corr} \tag{6.6}$$

, which enforces continuity:

$$\nabla \vec{v^*_{final}} = 0 \tag{6.7}$$

Substituting (6.4), (6.5) and (6.6) into (6.7):

$$\begin{aligned} \nabla \left(\vec{v^*_{pr}} + \vec{v^*_{corr}} \right) = \nabla \left(-H \left(\vec{v^*_{pr,f}} \right) - \frac{V_C}{a_C} \cdot \left(\nabla p^*_{pr} \right) + \frac{V_C}{a_C} \cdot S_{mom} \right) \\ + \nabla \left(-H \left(\vec{v^*_{corr,f}} \right) - \frac{V_C}{a_C} \cdot \left(\nabla p^*_{corr} \right) + \frac{V_C}{a_C} \cdot S_{mom} \right) = 0 \end{aligned} \tag{6.8}$$

The SIMPLE algorithm neglects the corrective terms except for the pressure correction gradient term. (6.8) can now be written for p^*_{corr} with only known terms on the right hand side:

$$\Delta p^*_{corr} = \frac{a_C}{V_C} \nabla \vec{v^*_{pr,C}} \tag{6.9}$$

As $\nabla \vec{v^*_{pr}} = -\nabla \vec{v^*_{corr}}$ the velocity correction becomes:

$$\vec{v^*_{corr}} = -\frac{V_C}{a_C} \nabla p^*_{corr}$$

Cell velocities are corrected according to (6.5):

$$\vec{v^*_{final,C}} = \vec{v^*_{pr,C}} - \frac{V_C}{a_C} \nabla p^*_{corr}$$

Correction of the face are also necessery for providing a divergence free massflux field in the next timestep. The correction is calculated using the cell gradient of p_{corr} interpolated to the faces as:

$$\vec{v^*_{final,f}} = \vec{v^*_{pr,f}} - \left(\frac{V_C}{a_C}\right)_f (\nabla p^*_{corr})_f$$

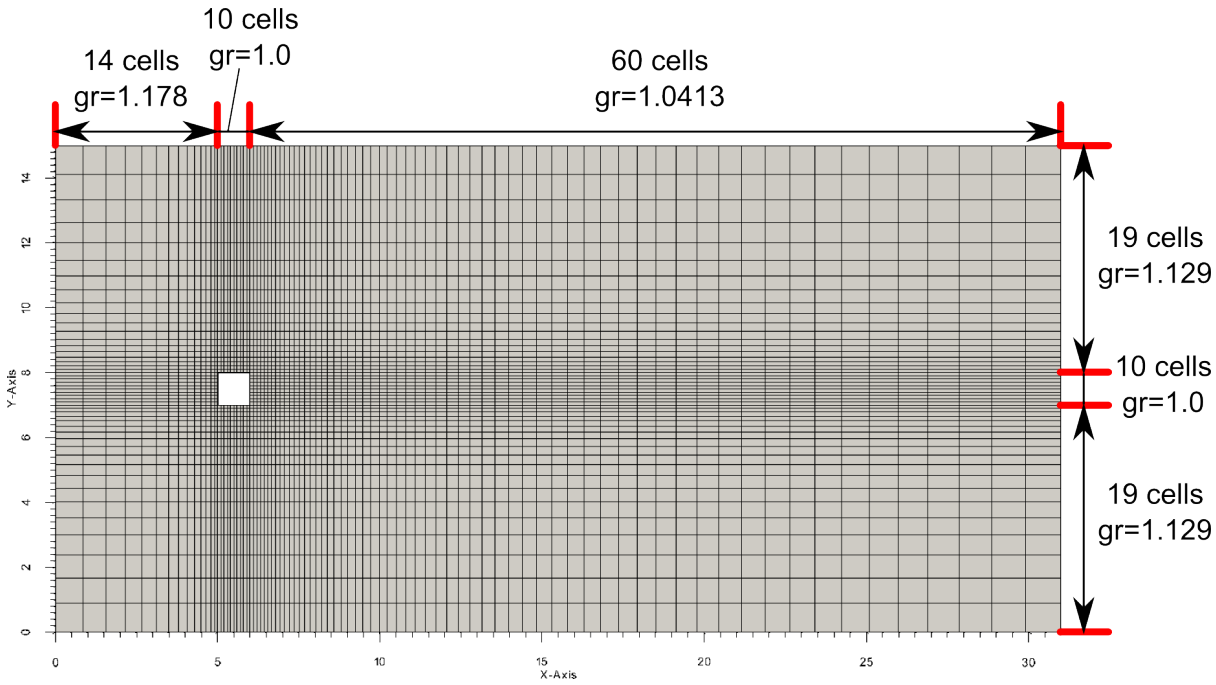
CHAPTER

SEVEN

SOLUTION OF THE SQUARE CYLINDER PROBLEM

7.1 Computational grid

To save computational time the following block structured mesh was created for the square cylinder example with dimensions according to the project description. The mesh consists of 3932 hexahedron cells. The mesh, cell numbers in the blocks and growth rates (*gr*) are shown on the following picture:



7.2 Description of script used for the calculation

(The complete *script for calculating the flow around a square cylinder* (page 37))

7.2.1 1. Reading mesh

The first important step is the loading of the mesh from a subdirectory of the "_MESH" directory. Additionaly cell volumes and face areas are saved as vectors:

```
# create mesh object and read mesh data from MESHES directory
myMesh = readers.FoamMesh("squareCylinderGrad")
cell_volumes = myMesh.get_volumes()[:,0]
face_areas = myMesh.get_areas()
```

7.2.2 2. Creating fields

- Five volume fields are created: pressure (p), pressure correction (p_corr), ρ (rho), $\frac{1}{\rho}$ (one_over_rho) as `pyCFD_fields.fields.ScalarField` (page 49) and velocity (U) as `pyCFD_fields.fields.VectorField` (page 50). For pressure, pressure correction and velocity appropriate boundary conditions are set. The fields U and p represent the non dimensional fields as described in *The non-dimensional equations* (page 12).
- Additionaly the p_old and U_old fields are also created as copies of the original p and U fields. U_old is used in the IMEX scheme and in the Rhie-Chow interpolation, while p_old in the IMEX scheme.
- Pressure and correction pressure are initialized with the value of 0, while the velocity with the vector $(1, 0, 0)$.
- Two fields are created for the current and old values of mass fluxes (m_dot and m_dot_old) with `pyCFD_fields.calculated_fields.MassFlux` (page 47).

```
# create and initialize field U, V and W
U = fields.VectorField(myMesh, "U", [1.,0.,0.])
U.get_patch("inlet"       ).set_patch_uniform([1.,0.,0.], "fixedValue"    )
U.get_patch("side"        ).set_patch_uniform(0., "fixedGradient")
```

```
U.get_patch("outlet"      ).set_patch_uniform(0., "fixedGradient")
U.get_patch("frontAndRear").set_patch_uniform(0., "fixedGradient")
U.load_init_fields("square_cylinder_5s")
U_old = fields.VectorField(myMesh, "U_old")
U_old.copy_field(U_old)

# create and initialize field p
p = fields.ScalarField(myMesh, "p", 0.)
p.get_patch("inlet"       ).set_patch_uniform(0., "fixedGradient")
p.get_patch("side"        ).set_patch_uniform(0., "fixedGradient")
p.get_patch("walls"       ).set_patch_uniform(0., "fixedGradient")
p.get_patch("frontAndRear").set_patch_uniform(0., "fixedGradient")
p.load_init_fields("square_cylinder_5s")
# p_old for IMEX scheme
p_old = fields.ScalarField(myMesh, "p_old")
p_old.copy_field(p)
div_save = fields.ScalarField(myMesh, "div_save", 0.)
rhs = fields.ScalarField(myMesh, "press_RHS", 0.)

# create and initialize field p_corr
p_corr = fields.ScalarField(myMesh, "p_corr", 0.)
p_corr.get_patch("inlet"       ).set_patch_uniform(0., "fixedGradient")
p_corr.get_patch("side"        ).set_patch_uniform(0., "fixedGradient")
p_corr.get_patch("walls"       ).set_patch_uniform(0., "fixedGradient")
p_corr.get_patch("frontAndRear").set_patch_uniform(0., "fixedGradient")
p_corr.load_init_fields("square_cylinder_5s")

# create rho field
rho = fields.ScalarField(myMesh, "rho", 1.)
one_over_rho = fields.ScalarField(myMesh, "1_rho", 1.)

# create massflux field
m_dot = calcfield.MassFlux(U, rho)
m_dot_old = fields.SurfaceScalarField(myMesh, "massFlux_old", )
m_dot_old.copy_field(m_dot)
```

Note: Patches are initialized with *fixedValue* type and 0 value.

Note: The patch *frontAndRear* represents the patches towards the independent third direction (the flow around a square cylinder problem is calculated in 2D, z direction is skipped).

7.2.3 3. Set Re number and timestep

- Reynolds number is set to 200
- Set start time, time step and timesteps for saving .vtu output. During the calculation the following restarts were performed and new settings were applied:
- **1st run upto t*=5**
  timestep:
  - upto t*=00.250: dt*=0.025
  - upto t*=02.000: dt*=0.050

  IMEX: True
- **2nd run upto t*=80.0**
  timestep:
  - upto t*=05.000: dt*= 0.100
  - upto t*=05.100: dt*= 0.005

```
        - upto t*=05.575:  dt*= 0.025
        - upto t*=80.000:  dt*= 0.050
```

IMEX: True

vortex shedding induced in the period: 5.000 < t* < 5.075

```
# set Reynolds number
Re = 200.

# run settings
#start_time =   0.
#stop_time  =  80.
#time_step  = [ (  0.25 , 0.025 ),
#               (  2.   , 0.05  ),
#               (  5.   , 0.1   ),
#               (  6.   , 0.005 ),
#               (  7.   , 0.025 ),
#               (  8.   , 0.05  ),
#               ( 80.   , 0.1   )   ]
#save_step  =   0.5
#vtk_start  =   0
# 2nd run
start_time =   5.
stop_time  =  80.
time_step  = [ (  5.1  , 0.005 ),
               (  5.575, 0.025 ),
               ( 80.   , 0.05  ) ]
save_step  =   0.5
vtk_start  =  10
```

7.2.4 4. Set up solvers for the equations

- The momentum equation is created as pyCFD_operators.generic_equation.GenericVectorEquation (page 74). For solving the linear equation systems the Gauss-Seidel solver (pyCFD_linear_solvers.linear_solvers.gs (page 60)) was applied with an under relaxation factor $\lambda_m = 0.7$
- The pressure correction equation is created as pyCFD_operators.generic_equation.GenericScalarEquatio (page 72). For solving the linear equation systems the direct solver using LU decomposition (pyCFD_linear_solvers.linear_solvers.lu_solver_plu (page 61)) was applied with no under relaxation.
- Under relaxation of the pressure and velocity corrections was set to $\lambda_p = 1.0$ to ensure mass conservation of the transient process.

```
# set up equation for U and p_corr
U_eqn      = generic_equation.GenericVectorEquation(myMesh, U,      "gs"       , 0.7 )
p_corr_eqn = generic_equation.GenericScalarEquation(myMesh, p_corr, "lu_solver", 1.  )
p_under_relaxation = 1.
```

7.2.5 5. Calculate and store coefficient matrix of laplace terms

Coefficient matrix of the laplace terms both in the momentum and pressure correction matrix are not changing during this calculation therefore they are calculated and stored in advance. The matrices are stored using operator objects: pyCFD_operators.implicit_operators.LaplaceVec (page 79) for the momentum equation and pyCFD_operators.implicit_operators.Laplace (page 78) for the pressure correction equation.

```
# build Laplace term coefficient matrices now as they will not change
U_lapl = implicit_operators.LaplaceVec(U, 1./Re)
p_lapl = implicit_operators.Laplace(p_corr)
```

7.2.6 6. Calculate and store LU decomposition for the pressure equation

The LU solver can use the pre-calculated lower and upper triangle matrices, therefore these are calculated and stored in advance. The matrices P,L and U are stored in *p_corr_eq* of type pyCFD_operators.generic_equation.GenericScalarEquation (page 72) in the variables *p*, *l* and *u*. LU decomposition is calculated in pyCFD_linear_solvers.linear_solvers.lu_decomp (page 60).

```
# calculate or load PLU decomposition of p_corr_eqn
loadPLU = True
if start_time == 0. and loadPLU == False:
    print "\ncalculating lu decomposition for pressure..."
    tlu = time.time()
    if Config_.__sparse__:
        p_corr_eqn.p, p_corr_eqn.l, p_corr_eqn.u = mylin.lu_decomp(p_lapl.A.todense())
    else:
        p_corr_eqn.p, p_corr_eqn.l, p_corr_eqn.u = mylin.lu_decomp(p_lapl.A)
```

7.2.7 7. Start time iteration loop

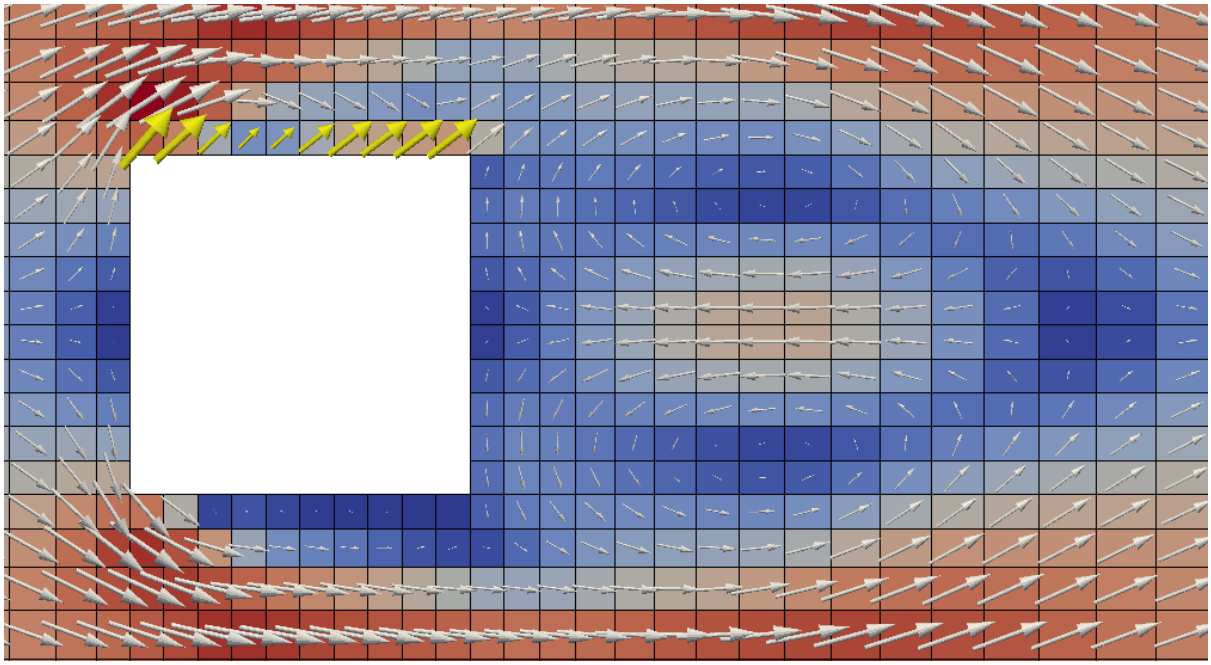
7.1 Update boundary conditions and mass fluxes

- boundary conditions are updated (update values at fixedGradient type boundaries)
- coefficient matrix elements are reset to 0
- right hand side values are reset to 0
- mass flux is re-calculated

```
# update boundary conditions, reset matrices and right hand sides, update m_dot
U_eqn.reset()
p_corr_eqn.reset()
p.update_boundary_values()
```

7.2 Induce vortex shedding at a specified time

Vortex shedding induced with applying artificial velocity values over the square cylinder. These vectors are highlighted with yellow color on the image below (the image shows the velocity distribution after the iteration step t*=5.500):



Artificial velocity values were applied between 5.000 < t* < 5.075 using pyCFD_fields.initialization.init_cell_values_in_box (page 50). The initialization value was $(1, 1, 0)$.

```
# induce vortex shedding with a perturbation above the cylinder
if time_ > 5. and time_ < 5.075:
    init.init_cell_values_in_box(U, numpy.array([5., 8., 0.]),
```

7.3 Assemble momentum predictor equation using a 2nd order IMEX scheme

The momentum equation is solved in the form as previously written in *3D vector form* (page 13):

$$\frac{\partial \vec{v}^*}{\partial t^*} + \vec{v}^* \cdot \nabla \vec{v}^* = -\nabla p^* + \frac{1}{Re} \cdot \Delta \left(\vec{v}^* \right)$$

- According to Ascher et al 1995 (Ascher, Ruth, Wetton: *IMPLICIT-EXPLICIT METHODS FOR TIME-DEPENDENT PARTIAL DIFFERENTIAL EQUATIONS*, SIAM J. NUMER. ANAL., Vol. 32, No. 3, pp. 797–823, June 1995) a second order accurate IMplicit-EXplicit scheme is constructed using the explicit Adams-Bashforth scheme for the convection (C) and gradient (G) terms and the implicit Crank-Nicholson scheme for the diffusion term (D).

$$\frac{U^{n+1} - U^n}{\delta t} + \frac{3}{2} C\left(U^n\right) - \frac{1}{2} C\left(U^{n-1}\right) - \frac{\nu}{2} D\left(U^{n+1}\right) - \frac{\nu}{2} D\left(U^n\right) + \frac{3}{2}\frac{1}{\rho} G\left(p^n\right) - \frac{1}{2}\frac{1}{\rho} G\left(p^{n-1}\right) = 0$$

- The time derivative term $\frac{U^{n+1} - U^n}{\delta t}$ is represented by the Euler operator: pyCFD_operators.implicit_operators.DdtEulerVec (page 77) .

```
U_eqn += implicit_operators.DdtEulerVec(U_eqn, myLoop.dt)
```

- A pyCFD_fields.fields.ScalarField (page 49) (*Dt*) is created in which the implicit conribution of the transient term is saved. This field is used later in the Rhie-Chow interpolation.

```
Dt = fields.ScalarField(myMesh, "Dt_time")
Dt.initialize_cell_with_vector(U_eqn.diag())
```

- The term $C\left(U^n\right)$ is represented by pyCFD_operators.explicit_operators.DivergenceVec (page 70) using the pyCFD_fields.calculated_fields.HRSFaceValue.MINMOD (page 47) scheme for calculating the surface values of U.

```
U_eqn += explicit_operators.DivergenceVec(U,     m_dot,     "MINMOD") * 1.5
```

- The term $C\left(U^{n-1}\right)$ is represented by pyCFD_operators.explicit_operators.DivergenceVec (page 70) using the pyCFD_fields.calculated_fields.HRSFaceValue.MINMOD (page 47) scheme for calculating the surface values of U. This operator is applied using the old velocity field *U_old* and the old mass flux field *m_dot_old*.

```
U_eqn -= explicit_operators.DivergenceVec(U_old, m_dot_old, "MINMOD") * 0.5
```

- The term $D\left(U^{n+1}\right)$ is represented by the pyCFD_operators.implicit_operators.LaplaceVec (page 79) operator. The coefficient matrix of this term is not calculated, it is added from the previously saved variable *U_lapl* (see *5. Calculate and store coefficient matrix of laplace terms* (page 19)).

```
U_eqn -= implicit_operators.LaplaceVec(U, 1./Re, "", False) * 0.5
U_eqn.A = U_eqn.A - U_lapl.A * 0.5
```

- The term $D\left(U^{n}\right)$ is represented by the pyCFD_operators.explicit_operators.LaplaceVec (page 72) operator.

```
U_eqn -= explicit_operators.LaplaceVec(U, 1./Re) * 0.5
```

- The pressure gradient terms are represented by the explicit gradient operator pyCFD_operators.explicit_operators.Gradient (page 71) with the factor $\frac{1}{\rho}$ (*one_over_rho*). The term $G\left(p^{n-1}\right)$ is applied for the old pressure field *p_old*.

```
U_eqn += explicit_operators.Gradient(p,     one_over_rho) * 1.5
U_eqn -= explicit_operators.Gradient(p_old, one_over_rho) * 0.5
```

The whole code section:

```
if Config_.__IMEX__ == True:
    ## PREDICTOR IMEX
    ## convection, gradient: Adams-Bashfort
    ## diffusion:            Crank-Nicolson
    # assemble momentum equation
    U_eqn += implicit_operators.DdtEulerVec(U_eqn, myLoop.dt)
    Dt = fields.ScalarField(myMesh, "Dt_time")
    Dt.initialize_cell_with_vector(U_eqn.diag())
    U_eqn += explicit_operators.DivergenceVec(U,     m_dot,     "MINMOD") * 1.5
    U_eqn -= explicit_operators.DivergenceVec(U_old, m_dot_old, "MINMOD") * 0.5
    # use pre-calculated coefficient matrix for the laplace operator
    U_eqn -= implicit_operators.LaplaceVec(U, 1./Re, "", False) * 0.5
    U_eqn.A = U_eqn.A - U_lapl.A * 0.5
    U_eqn -= explicit_operators.LaplaceVec(U, 1./Re) * 0.5
    U_eqn += explicit_operators.Gradient(p,     one_over_rho) * 1.5
```

7.4 Solve momentum equation and check divergence

- The momentum is solved, old field values are overwritten.
- The mass flux *m_dot* is updated.
- The old mass flux *m_dot_old* is overwritten

```
# solve momentum equation
U_eqn.solve()

#overwrite old m_dot field befor Rhie-Chow interpolation
m_dot_old.copy_field(m_dot)

# calculate predicted massflux field
m_dot = calcfield.MassFlux(U, rho)
```

7.5 Correct $\dot{m}$ (*m_dot*) with Rhie-Chow interpolation

- According to Darwish et al 2000 (Darwish, Moukalled: *The Rhie-Chow Interpolation Revisited*, Int. J. Numer. Meth. Fluids 2000; 00:1-6) the Rhie- Chow interpolation is applied for $\dot{m}$. The Rhie-Chow interpolated face velocity can be written as:

$$u_f = \overline{u} - \overline{\frac{V}{a^t + a^{CD}}} \left(\left(\frac{\partial p}{\partial x} \right)_f - \overline{\frac{\partial p}{\partial x}} \right) + \overline{\frac{a^t}{a^t + a^{CD}}} \left(u_f^o - \overline{u^o} \right)$$

  , where the overbar $\overline{<>}$ is the linear interpolated value of <> at the face, the f index $<>_f$ refers to old values at the face or values calculated directly at the face, V is the cell volume, a^t is the transient contribution in the main diagonal of the coefficient matrix of the momentum equation, a^{CD} is the contribution from the convection and diffusion terms, u^o is the old velocity value.

- As $\rho = 1$, $\dot{m}$ can be directly corrected with the above equation from the face velocity.
- The term $\frac{V}{a^t + a^{CD}}$ is renamed to D. The face interpolate of D is called D_f. Interpolation is done using `pyCFD_fields.calculated_fields.LinearFaceValue` (page 47).
- The face pressure gradient $\left(\frac{\partial p}{\partial x} \right)_f$ is calculated by `pyCFD_fields.calculated_fields.GaussFaceGradient` (page 46). In the structured grid used its expression simplifies to:

$$\frac{\phi_N - \phi_O}{|\vec{d_{ON}}|} \vec{e_{ON}}$$

  It can be seen that this face gradient is calculated using a small stencil which includes only the two cells next to the face.

- The interpolated cell pressure gradient term $\overline{\frac{\partial p}{\partial x}}$ is the linear interpolate (`pyCFD_fields.calculated_fields.LinearFaceValue` (page 47)) of `pyCFD_fields.calculated_fields.GaussCellGradient` (page 45). It is calculated through the steps:
  - $\phi_{f'} = g_O * \phi_O + g_N * \phi_N$
  - $\nabla \phi_O = \frac{1}{V_O} \sum_{nb} \phi_{f'} \vec{S_f}$
  - $\nabla \phi_f = g_O * (\nabla \phi)_O + g_N (\nabla \phi)_N$

  It can be seen that the interpolated face gradient is calculated using a wider stencil which includes also neighbour cell values of the cells next to the face.

- The term $\frac{a^t}{a^t + a^{CD}}$ is calculated from *Dt* (*7.3 Assemble momentum predictor equation using a 2nd order IMEX scheme* (page 21)) by dividing with the main diagonal of the momentum equation's coefficient matrix. *Dt* is then interpolated to the cell faces by `pyCFD_fields.calculated_fields.LinearFaceValue` (page 47).

```
## CORRECTOR
# correct m_dot with Rhie-Chow interpolation
# skip this only after a restart!
if time_ != myLoop.times[0] and time_ != 0.:
    D = fields.ScalarField(myMesh, "D")
    D.initialize_cell_with_vector(cell_volumes/U_eqn.diag())
    D_f = calcfield.LinearFaceValue(D)
    grad_p = calcfield.GaussCellGradient(p)
    grad_p_lin_f = calcfield.LinearFaceValue(grad_p)
    grad_p_f = calcfield.GaussFaceGradient(p)
    m_dot.A -= D_f.A * (grad_p_f.dot_Sf()[:,0] - grad_p_lin_f.dot_Sf()[:,0])
    m_dot_old_lin = calcfield.MassFlux(U_old, rho)
    Dt.V /= U_eqn.diag()
    Dt_f = calcfield.LinearFaceValue(Dt)
```

7.6 Assemble pressure correction equation

The pressure correction equation is solved in the form as previously written in *The SIMPLE algorithm* (page 14):

$$\Delta p^*_{corr} = \frac{a_C}{V_C} \nabla \vec{v^*_{pr,C}}$$

, where $\vec{v^*_{pr,C}}$ is the predicted velocity field after the solution of the momentum equations (see *7.3 Assemble momentum predictor equation using a 2nd order IMEX scheme* (page 21) and *7.4 Solve momentum equation and check divergence* (page 22)).

- The term Δp^*_{corr} is represented by the implicit laplace operator `pyCFD_operators.implicit_operators.Laplace` (page 78). The coefficient matrix of this term is not calculated, it is added from the previously saved variable *p_lapl* (see *5. Calculate and store coefficient matrix of laplace terms* (page 19)).

```
p_corr_eqn += implicit_operators.Laplace(p_corr, 1., "", False)
p_corr_eqn.A = p_corr_eqn.A + p_lapl.A
```

- The divergence term is taken into account by the explicit divergence operator `pyCFD_operators.explicit_operators.Divergence` (page 70).

```
p_corr_eqn -= explicit_operators.Divergence(None, m_dot, "")
p_corr_eqn.b[:,0] *= U_eqn.diag() / cell_volumes
```

The whole code section:

```
# assemble pressure-correction equation
# use pre-calculated coefficient matrix for the laplace operator
p_corr_eqn += implicit_operators.Laplace(p_corr, 1., "", False)
p_corr_eqn.A = p_corr_eqn.A + p_lapl.A
p_corr_eqn -= explicit_operators.Divergence(None, m_dot, "")
p_corr_eqn.b[:,0] *= U_eqn.diag() / cell_volumes
```

7.7 Solve pressure correction equation and correct fields

- The pressure correction equation is solved.
- The pressure field is updated using

$$p^*_{final} = p^*_{pr} + p^*_{corr} * \lambda_p$$

- The velocity field is updated using

$$\vec{v^*_{corr,C}} = -\frac{V_C}{a_C} \nabla p^*_{corr}$$

  and

$$\vec{v^*_{final,C}} = \vec{v^*_{pr,C}} + \vec{v^*_{corr,C}} * \lambda_p$$

- The face velocity field is updated using

$$\vec{v^*_{final,f}} = \vec{v^*_{pr,f}} - \left(\frac{V_C}{a_C}\right)_f (\nabla p^*_{corr})_f$$

```
p_corr_eqn.b[:,0] *= U_eqn.diag() / cell_volumes
# solve pressure-correction equation
p_corr_eqn.solve()

#overwrite p_old field befor pressure correction
```

```
if Config_.__IMEX__ == True:
    p_old.copy_field(p)

# correct pressure
p.V += p_corr.V * p_under_relaxation
# correct velocity components
p_corr.update_boundary_values()
grad_p_corr = calcfield.GaussCellGradient(p_corr)
U.V[:,0] -= grad_p_corr.V[:,0] * cell_volumes/U_eqn.diag() * p_under_relaxation
U.V[:,1] -= grad_p_corr.V[:,1] * cell_volumes/U_eqn.diag() * p_under_relaxation
U.V[:,2] -= grad_p_corr.V[:,2] * cell_volumes/U_eqn.diag() * p_under_relaxation
U.update_boundary_values()
# correct mass flux with Rhie-Chow interpolation
if time_ != myLoop.times[0] and time_ != 0.:
    grad_p_corr_f = calcfield.GaussFaceGradient(p_corr)
    grad_p_corr_f_Sf = grad_p_corr_f.dot_abs_Sf()
    for face_ in myMesh.faces:
        if face_.isBnd == True:
            continue
```

7.2.8 8. Monitoring the calculation

During the calculation the following values are stored for monitoring purposes:

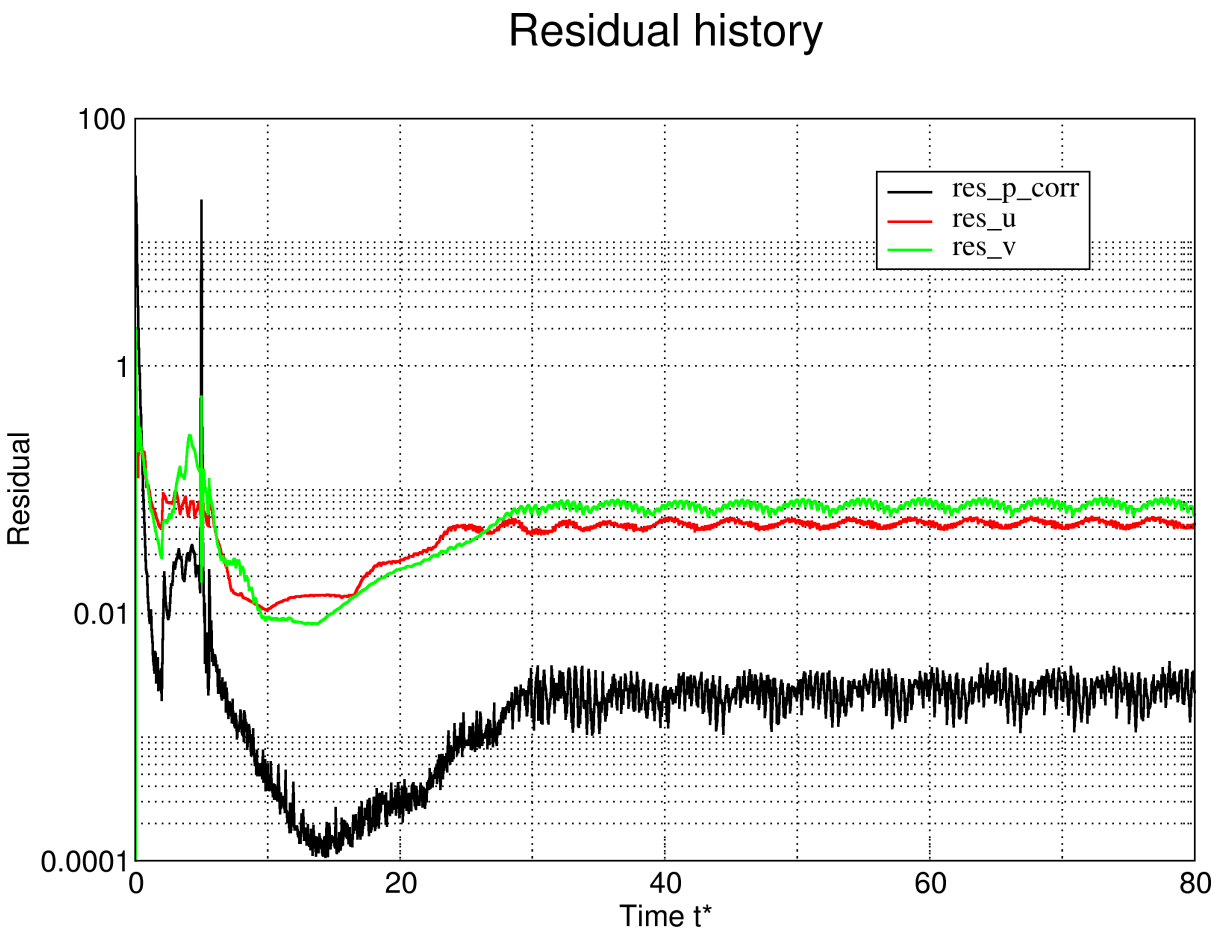
- absolute residuals of the equations beeing solved
- absolute value of divergence after the predictor step and after the corretor step
- drag coefficient of the square cylinder with pyCFD_monitors.monitors.cd (page 69)
- mass balance through the boundaries with pyCFD_monitors.monitors.globalMass (page 70)

```
# monitor remaining divergence
div_ = calcfield.Divergence(None, m_dot, "")
print "corrector divergence: "+str(max(abs(div_.V)))
p_corr_eqn.append_to_monitor(max(abs(div_.V)), "divU_corr")

# monitor drag coefficient
current_cd = monitors.cd(p, 1., 1., "walls", numpy.array([1., 0., 0.]), 0.1)
print "cd of cylinder: "+str(current_cd)
p_corr_eqn.append_to_monitor(current_cd, "cd")

# monitor global mass ballance
mass_ballance = monitors.globalMass(m_dot)
print "global mass ballance: "+str(mass_ballance)
```

7.3 Results

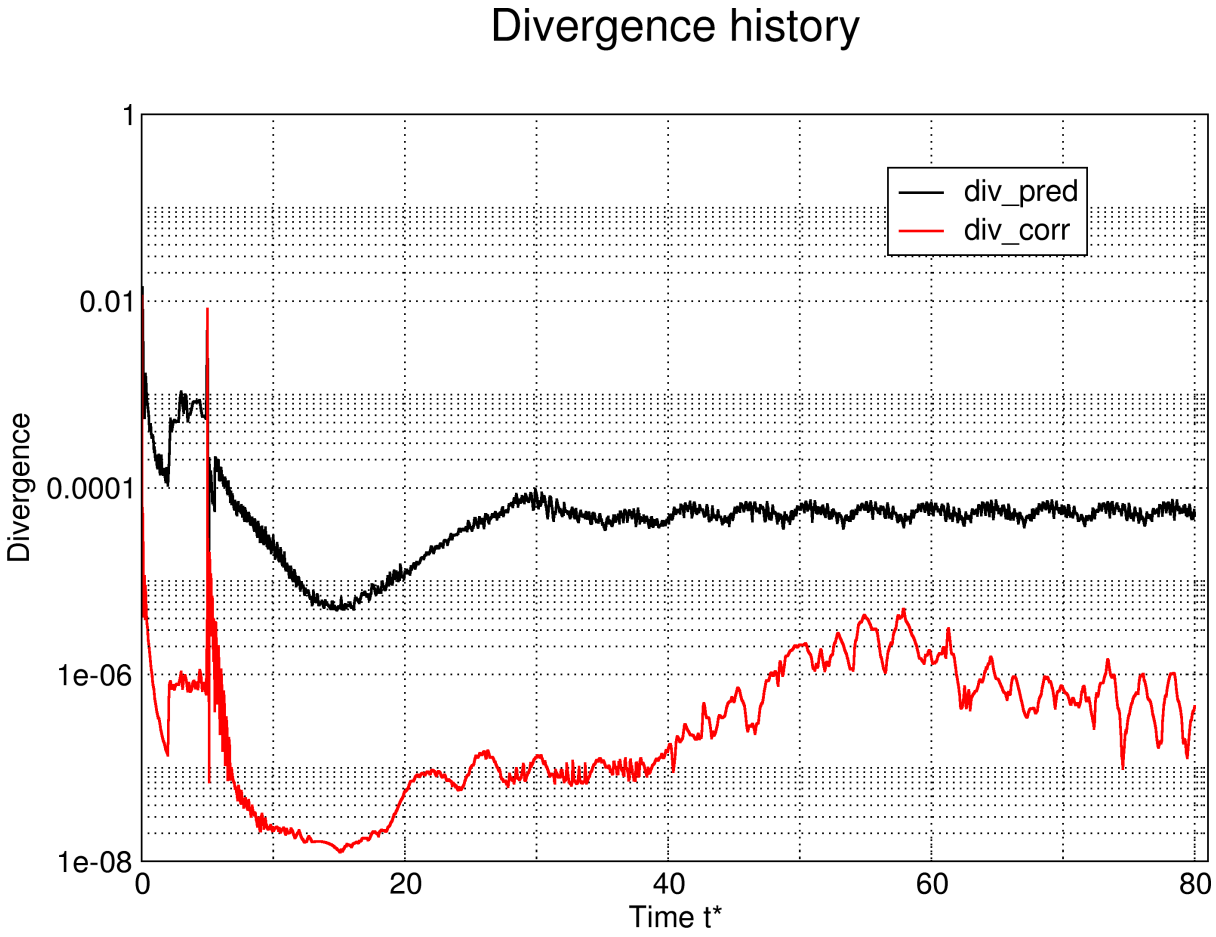
7.3.1 Calculation history

Residual history for the whole calculation is shown on the next image:

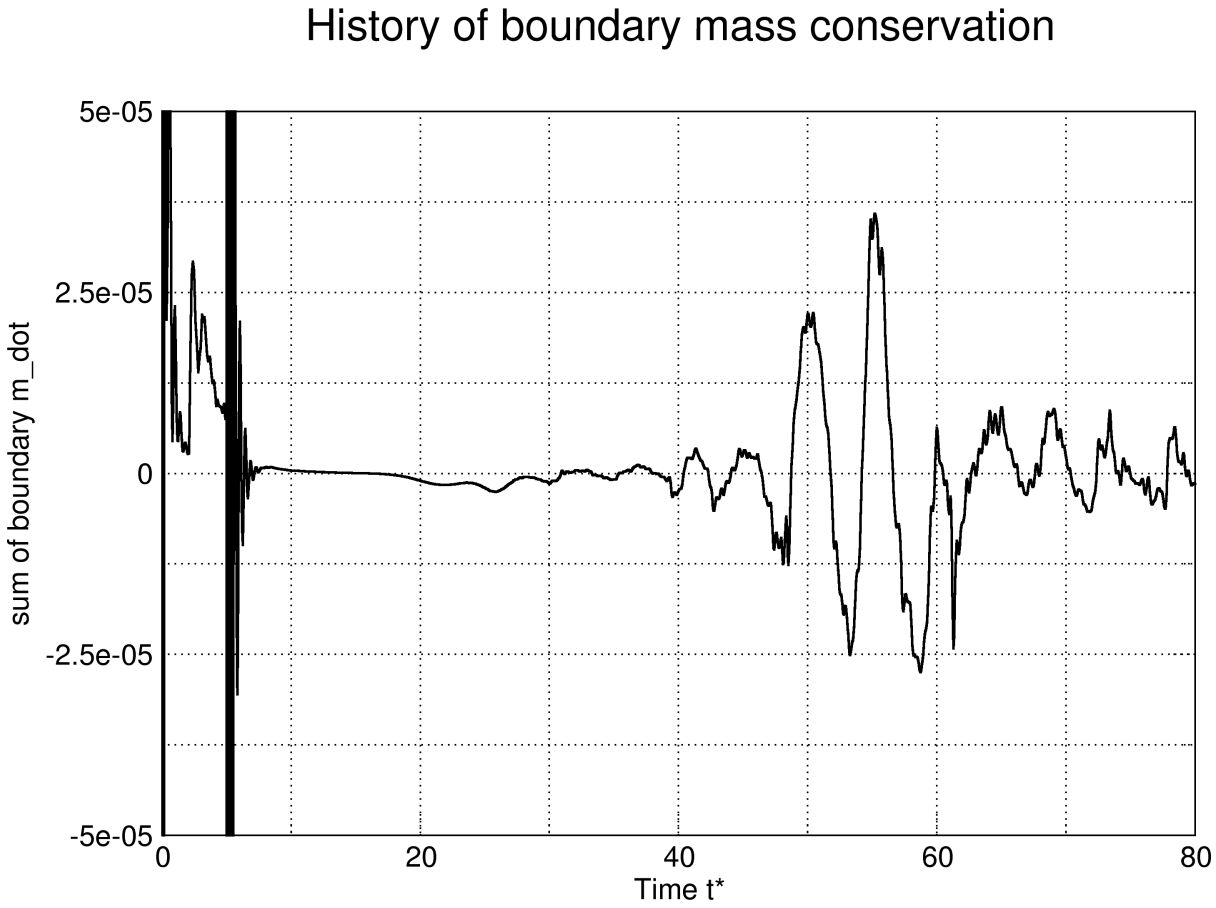


The high transients visible in the charts are the results of a restart which was necessary to migrate data between different machines. The restart history correcponds to the settings described in *3. Set Re number and timestep* (page 18).

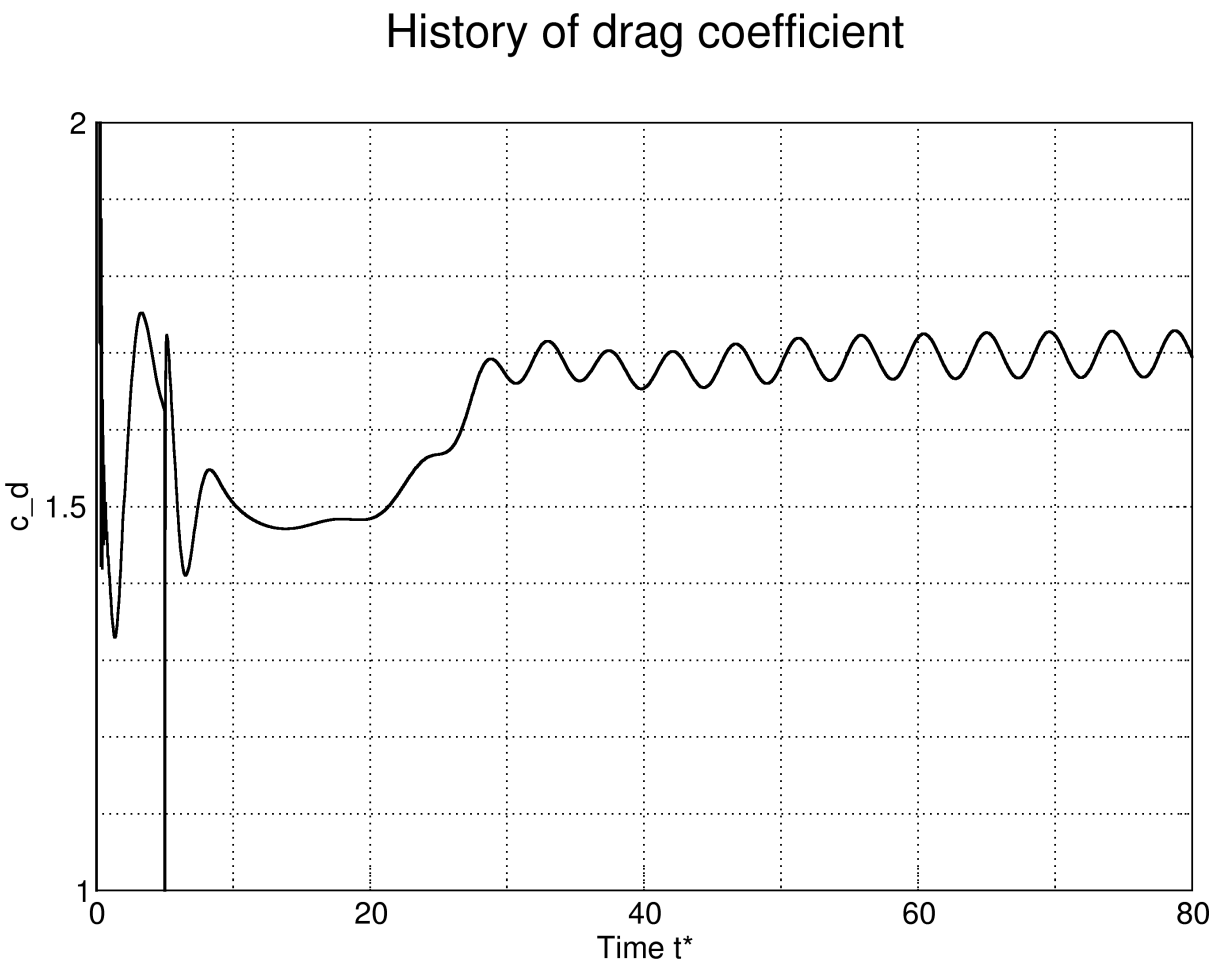
The next image shows the history of divergence:



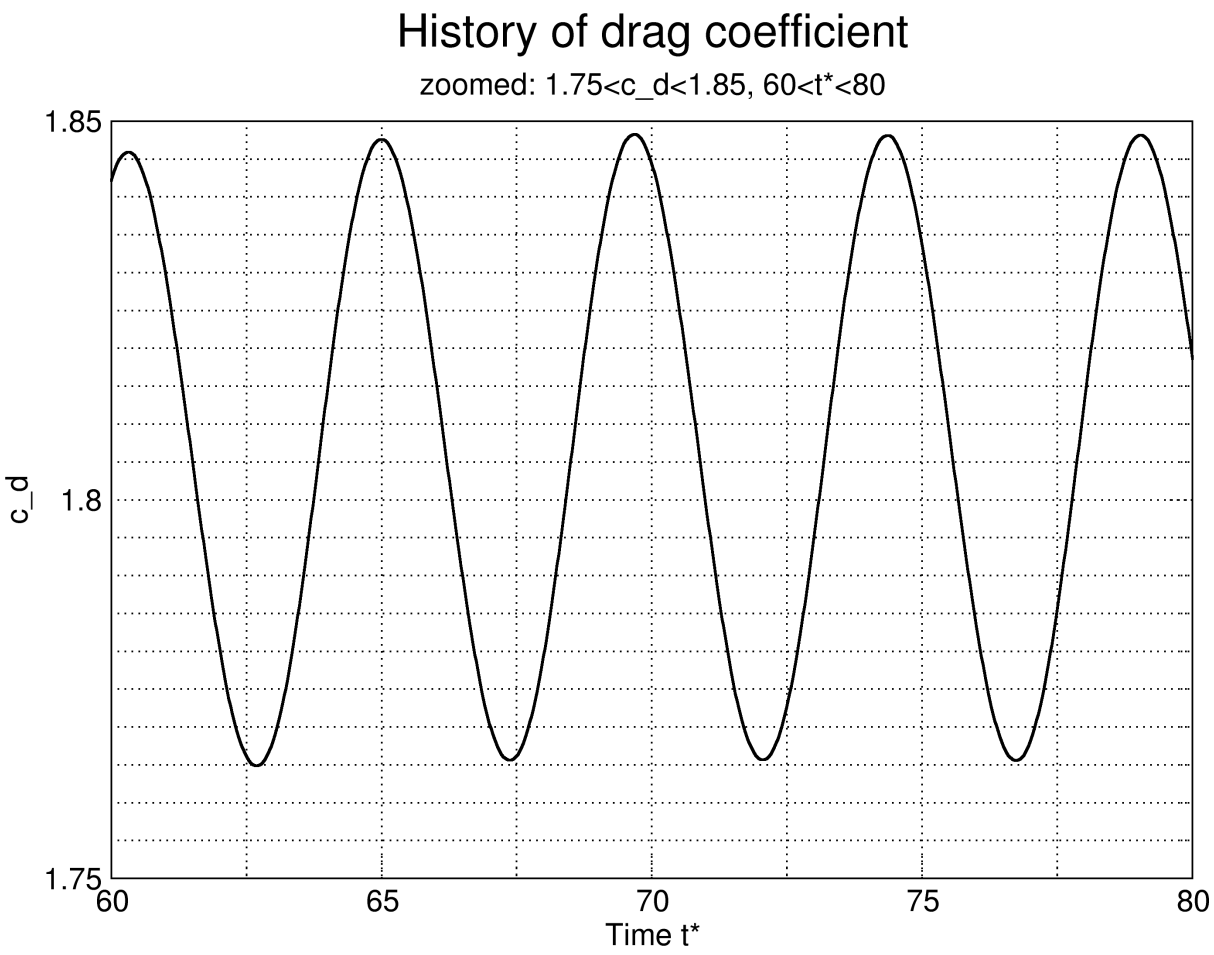
The history of boundary mass conservation:



Finally the history of drag coefficient:

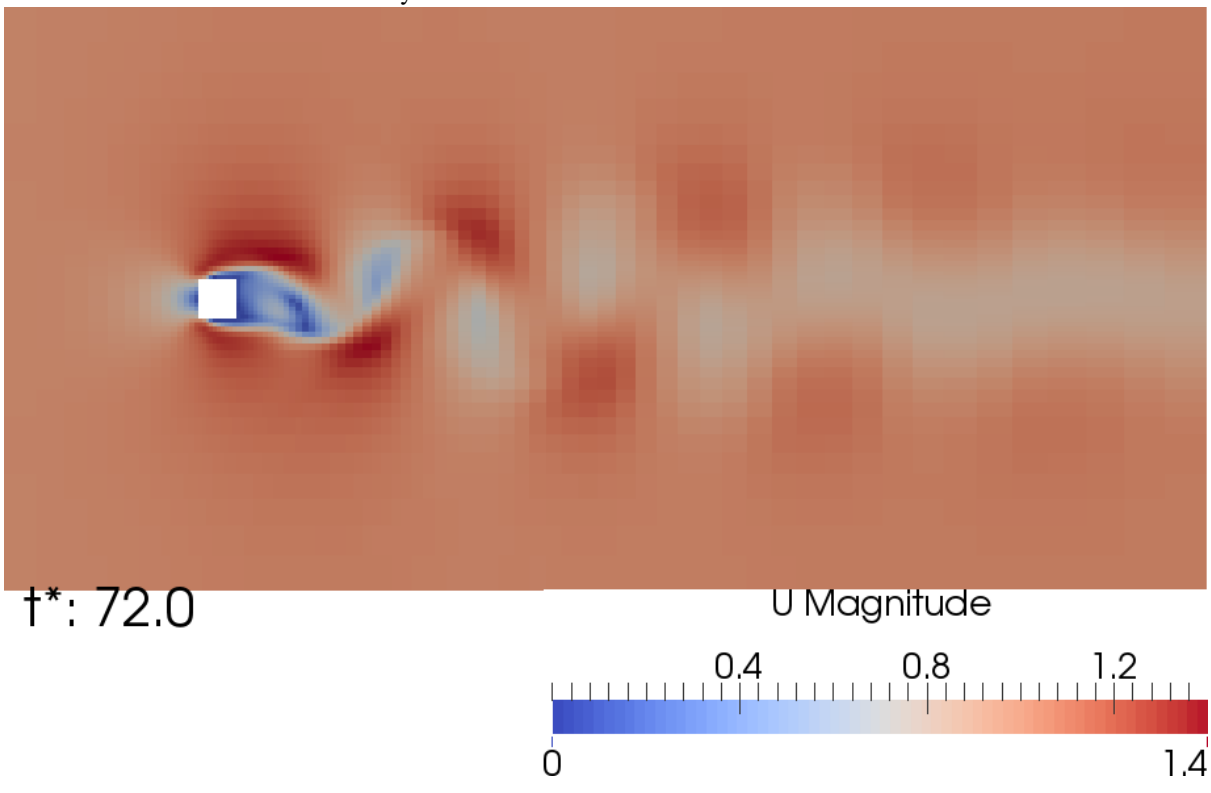


The zoom of drag coefficient histor shows that the last three periods of drag coefficient history are not changing therefore a statistical convergence is found:

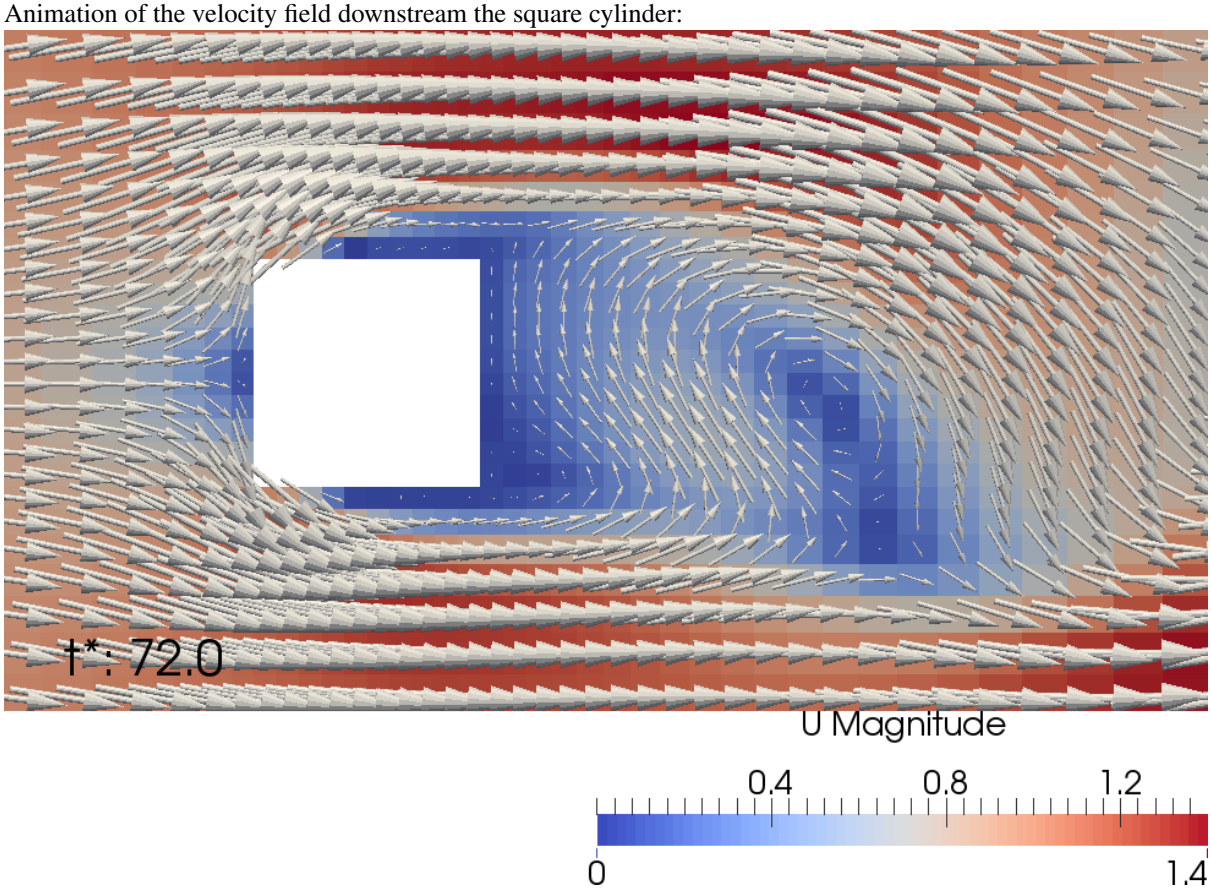


7.3.2 Flow field

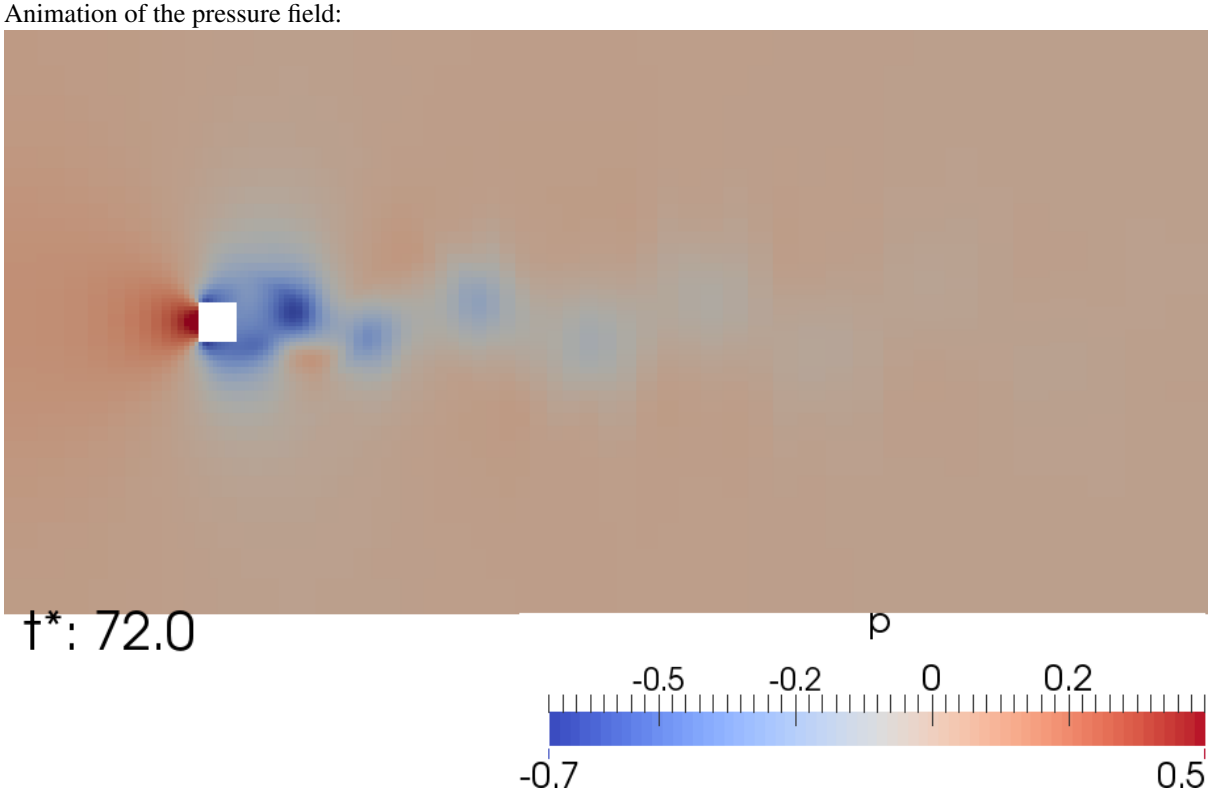
The next animation shows the velocity distribution in the domain:



Animation of the velocity field downstream the square cylinder:



Animation of the pressure field:



CHAPTER

EIGHT

APPENDIX 1: TEST CODES

8.1 script for solving the Smith-Hutton problem

```
"""
2D test case for the Smith-Hutton problem
"""
__author__ = "Bence Somogyi"
__copyright__ = "Copyright 2014"
__version__ = "0.1"
__maintainer__ = "Bence Somogyi"
__email__ = "bencesomogyi@ivt.tugraz.at"
__status__ = "Prototype"

# "global variables"
import pyCFD_config.config as Config_
Config_.__FIELDDIR__        = '_FIELD_FILES/'
Config_.__OUTDIR__          = '_OUTPUT/'
Config_.__OUTITERDIR__      = '_OUTPUT/ITERATIONS/'
Config_.__OUTITERDIRREL__   = 'ITERATIONS/'

import numpy
import pyCFD_mesh.readers as readers
import pyCFD_output.output as output
import pyCFD_fields.fields as fields
import pyCFD_fields.calculated_fields as calcfield
import pyCFD_calculation.time_loop as time_loop
import pyCFD_operators.generic_equation as generic_equation
#import pyCFD_operators.explicit_operators as explicit_operators
import pyCFD_operators.implicit_operators as implicit_operators

# create mesh object and read mesh data from MESHES directory
myMesh = readers.FoamMesh("smithhutton")

# create and initialize field U
U = fields.VectorField(myMesh, "U")

for i_cell in range(len(U.V)):
    cell_x = myMesh.cells[i_cell].C[0]
    cell_y = myMesh.cells[i_cell].C[1]
    U.V[i_cell][0] =  2. * cell_y * (1. - pow(cell_x,2))
    U.V[i_cell][1] = -2. * cell_x * (1. - pow(cell_y,2))

for patch_ in myMesh.patches:
    patch_length = len(patch_.faces)
    temp_vector = numpy.zeros((patch_length,3))
    for face_i,face_ in enumerate(patch_.faces):
        face_x = face_.C[0]
        face_y = face_.C[1]
        temp_vector[face_i][0] =  2. * face_y * (1. - pow(face_x,2))
```

```
        temp_vector[face_i][1] = -2. * face_x * (1. - pow(face_y,2))
    U.get_patch(patch_.name).set_patch_distributed(temp_vector, "fixedValue")

U.update_boundary_values()

# create rho field
rho = fields.ScalarField(myMesh, "rho", 1.0)

# create massflux field
m_dot = calcfield.MassFlux(U, rho)

# create scalar field with boundary conditions
phi = fields.ScalarField(myMesh, "phi", 1.0)
phi.get_patch("inlet1"      ).set_patch_uniform(1., "fixedValue"   )
phi.get_patch("outlet"      ).set_patch_uniform(0., "fixedGradient")
phi.get_patch("wall"        ).set_patch_uniform(0., "fixedGradient")
phi.get_patch("frontAndBack").set_patch_uniform(0., "fixedGradient")

# create time loop
save_fields = [U, phi]

start_time = 0.
stop_time  = 4.
time_step  = 0.05
save_step  = 1.

myLoop = time_loop.TimeLoop(save_fields, start_time, stop_time, time_step)
myLoop.uniform_save_times(save_step)

# set up equation for phi
phi_eqn = generic_equation.GenericScalarEquation(myMesh, phi, "bicg")

# save initial condition
if start_time == 0.:
    # clear output dirs
    output.clean_output_dirs()
    myLoop.save_current(0)

# iteration loop
for time_ in myLoop.times:
    myLoop.time = time_
    myLoop.print_time()

    phi_eqn.reset()

    phi_eqn += implicit_operators.DdtEuler(phi_eqn, myLoop.dt)
#    phi_eqn += implicit_operators.Divergence(phi, m_dot, "UDS")
#    phi_eqn += implicit_operators.Divergence(phi, m_dot, "MINMOD")
    phi_eqn += implicit_operators.Divergence(phi, m_dot, "STOIC")

    phi_eqn.solve()

    myLoop.save_time()

phi_eqn.save_residual(myLoop.times,"phi")

print ""
print "testConvection_SmithHutton_UDS FINSHED"
```

8.2 script for solving the diffusion problem

```
"""
2D test case diffusion using the Smith-Hutton mesh
"""
__author__ = "Bence Somogyi"
__copyright__ = "Copyright 2014"
__version__ = "0.1"
__maintainer__ = "Bence Somogyi"
__email__ = "bencesomogyi@ivt.tugraz.at"
__status__ = "Prototype"

# "global variables"
import pyCFD_config.config as Config_
Config_.__FIELDDIR__        = '_FIELD_FILES/'
Config_.__OUTDIR__          = '_OUTPUT/'
Config_.__OUTITERDIR__      = '_OUTPUT/ITERATIONS/'
Config_.__OUTITERDIRREL__   = 'ITERATIONS/'

import pyCFD_mesh.readers as readers
import pyCFD_output.output as output
import pyCFD_fields.fields as fields
#import pyCFD_fields.calculated_fields as calcfield
import pyCFD_calculation.time_loop as time_loop
import pyCFD_operators.generic_equation as generic_equation
#import pyCFD_operators.explicit_operators as explicit_operators
import pyCFD_operators.implicit_operators as implicit_operators

# clear output dirs
output.clean_output_dirs()

# create mesh object and read mesh data from MESHES directory
myMesh = readers.FoamMesh("smithhutton")

# create scalar field with boundary conditions
phi = fields.ScalarField(myMesh, "phi", 0.)
phi.get_patch("inlet2"      ).set_patch_uniform(1., "fixedValue"   )
phi.get_patch("wall"        ).set_patch_uniform(0., "fixedValue"   )
phi.get_patch("outlet"      ).set_patch_uniform(0., "fixedGradient")
phi.get_patch("frontAndBack").set_patch_uniform(0., "fixedGradient")

# create time loop
save_fields = [phi]

start_time = 0.
stop_time  = 1.
time_step  = 0.1
save_step  = 0.1

myLoop = time_loop.TimeLoop(save_fields, start_time, stop_time, time_step)
myLoop.uniform_save_times(save_step)

# set up equation for phi
phi_eqn = generic_equation.GenericScalarEquation(myMesh, phi, "bicg")

# save initial condition
myLoop.save_current(0)

# iteration loop
for time_ in myLoop.times:
    myLoop.time = time_
    myLoop.print_time()
```

```
    phi_eqn.reset()

    phi_eqn += implicit_operators.DdtEuler(phi_eqn, myLoop.dt)
    phi_eqn -= implicit_operators.Laplace(phi)

    phi_eqn.solve()

    myLoop.save_time()

phi_eqn.save_residual(myLoop.times)

print ""
print "testDiffusion_SmithHutton FINSHED"
```

8.3 script for solving the diffusion problem in the inclined block

```
"""
2D test case diffusion in an inclined block
"""
__author__ = "Bence Somogyi"
__copyright__ = "Copyright 2014"
__version__ = "0.1"
__maintainer__ = "Bence Somogyi"
__email__ = "bencesomogyi@ivt.tugraz.at"
__status__ = "Prototype"

# "global variables"
import pyCFD_config.config as Config_
Config_.__FIELDDIR__        = '_FIELD_FILES/'
Config_.__OUTDIR__          = '_OUTPUT/'
Config_.__OUTITERDIR__      = '_OUTPUT/ITERATIONS/'
Config_.__OUTITERDIRREL__   = 'ITERATIONS/'

import pyCFD_mesh.readers as readers
import pyCFD_output.output as output
import pyCFD_fields.fields as fields
#import pyCFD_fields.calculated_fields as calcfield
import pyCFD_calculation.time_loop as time_loop
import pyCFD_operators.generic_equation as generic_equation
#import pyCFD_operators.explicit_operators as explicit_operators
import pyCFD_operators.implicit_operators as implicit_operators

# clear output dirs
output.clean_output_dirs()

# create mesh object and read mesh data from MESHES directory
myMesh = readers.FoamMesh("inclinedBlock")

# create scalar field with boundary conditions
phi = fields.ScalarField(myMesh, "phi", 273.)
phi.get_patch("phi0"        ).set_patch_uniform(273., "fixedValue"    )
phi.get_patch("phi1"        ).set_patch_uniform(300., "fixedValue"    )
phi.get_patch("frontAndRear").set_patch_uniform(0., "fixedGradient")

# create time loop
save_fields = [phi]

start_time = 0.
stop_time  = 3.
time_step  = 0.1
```

```
save_step  = 1.

myLoop = time_loop.TimeLoop(save_fields, start_time, stop_time, time_step)
myLoop.uniform_save_times(save_step)

# set up equation for phi
phi_eqn = generic_equation.GenericScalarEquation(myMesh, phi, "bicg")

# save initial condition
myLoop.save_current(0)

# iteration loop
for time_ in myLoop.times:
    myLoop.time = time_
    myLoop.print_time()

    phi_eqn.reset()

    phi_eqn += implicit_operators.DdtEuler(phi_eqn, myLoop.dt)
    phi_eqn -= implicit_operators.Laplace(phi, 0.01)
    #phi_eqn -= implicit_operators.Laplace(phi, 0.01, "OVERRELAXED")

    phi_eqn.solve()

    myLoop.save_time()

phi_eqn.save_residual(myLoop.times)

print ""
print "testDiffusion_SmithHutton_explicit FINSHED"
```

CHAPTER

NINE

APPENDIX 2: SOLUTION CODE

9.1 script for calculating the flow around a square cylinder

```
"""
2D flow around square cylinder
"""
__author__ = "Bence Somogyi"
__copyright__ = "Copyright 2014"
__version__ = "0.1"
__maintainer__ = "Bence Somogyi"
__email__ = "bencesomogyi@ivt.tugraz.at"
__status__ = "Prototype"

import time
t00 = time.time()

# "global variables"
import pyCFD_config.config as Config_
Config_.__sparse__          = True
Config_.__FIELDDIR__        = '_FIELD_FILES/'
Config_.__OUTDIR__          = '_OUTPUT/'
Config_.__OUTITERDIR__      = '_OUTPUT/ITERATIONS/'
Config_.__OUTITERDIRREL__   = 'ITERATIONS/'
Config_.__IMEX__            = True
Config_.__runSettingsFile__ = "squareCylinderNonDim.conf"

import numpy
import pyCFD_mesh.readers as readers
import pyCFD_output.output as output
import pyCFD_fields.fields as fields
import pyCFD_fields.calculated_fields as calcfield
import pyCFD_calculation.time_loop as time_loop
import pyCFD_operators.generic_equation as generic_equation
import pyCFD_operators.explicit_operators as explicit_operators
import pyCFD_operators.implicit_operators as implicit_operators
import pyCFD_monitors.monitors as monitors
import pyCFD_linear_solvers.linear_solvers as mylin
import pyCFD_fields.initialization as init

# create mesh object and read mesh data from MESHES directory
myMesh = readers.FoamMesh("squareCylinderGrad")
cell_volumes = myMesh.get_volumes()[:,0]
face_areas = myMesh.get_areas()

# create and initialize field U, V and W
U = fields.VectorField(myMesh, "U", [1.,0.,0.])
U.get_patch("inlet"       ).set_patch_uniform([1.,0.,0.], "fixedValue"   )
U.get_patch("side"        ).set_patch_uniform(0., "fixedGradient")
U.get_patch("outlet"      ).set_patch_uniform(0., "fixedGradient")
```

```
U.get_patch("frontAndRear").set_patch_uniform(0., "fixedGradient")
U.load_init_fields("square_cylinder_5s")
U_old = fields.VectorField(myMesh, "U_old")
U_old.copy_field(U_old)

# create and initialize field p
p = fields.ScalarField(myMesh, "p", 0.)
p.get_patch("inlet"       ).set_patch_uniform(0., "fixedGradient")
p.get_patch("side"        ).set_patch_uniform(0., "fixedGradient")
p.get_patch("walls"       ).set_patch_uniform(0., "fixedGradient")
p.get_patch("frontAndRear").set_patch_uniform(0., "fixedGradient")
p.load_init_fields("square_cylinder_5s")
# p_old for IMEX scheme
p_old = fields.ScalarField(myMesh, "p_old")
p_old.copy_field(p)
div_save = fields.ScalarField(myMesh, "div_save", 0.)
rhs = fields.ScalarField(myMesh, "press_RHS", 0.)

# create and initialize field p_corr
p_corr = fields.ScalarField(myMesh, "p_corr", 0.)
p_corr.get_patch("inlet"       ).set_patch_uniform(0., "fixedGradient")
p_corr.get_patch("side"        ).set_patch_uniform(0., "fixedGradient")
p_corr.get_patch("walls"       ).set_patch_uniform(0., "fixedGradient")
p_corr.get_patch("frontAndRear").set_patch_uniform(0., "fixedGradient")
p_corr.load_init_fields("square_cylinder_5s")

# create rho field
rho = fields.ScalarField(myMesh, "rho", 1.)
one_over_rho = fields.ScalarField(myMesh, "1_rho", 1.)

# create massflux field
m_dot = calcfield.MassFlux(U, rho)
m_dot_old = fields.SurfaceScalarField(myMesh, "massFlux_old", )
m_dot_old.copy_field(m_dot)

# set Reynolds number
Re = 200.

# create time loop
save_fields = [U, p, p_corr]

# run settings
#start_time =   0.
#stop_time  =  80.
#time_step  = [ (  0.25 , 0.025 ),
#               (  2.   , 0.05  ),
#               (  5.   , 0.1   ),
#               (  6.   , 0.005 ),
#               (  7.   , 0.025 ),
#               (  8.   , 0.05  ),
#               ( 80.   , 0.1   )   ]
#save_step  =   0.5
#vtk_start  =   0
# 2nd run
start_time =   5.
stop_time  =  80.
time_step  = [ (  5.1  , 0.005 ),
               (  5.575, 0.025 ),
               ( 80.   , 0.05  ) ]
save_step  =   0.5
vtk_start  =  10

myLoop = time_loop.TimeLoop(save_fields, start_time, stop_time, time_step)
```

```
myLoop.uniform_save_times(save_step)
myLoop.vtk_start = vtk_start

# set up equation for U and p_corr
U_eqn      = generic_equation.GenericVectorEquation(myMesh, U,       "gs"       , 0.7 )
p_corr_eqn = generic_equation.GenericScalarEquation(myMesh, p_corr,  "lu_solver", 1.  )
p_under_relaxation = 1.

# save initial condition
if start_time == 0.:
    # clear output dirs
    output.clean_output_dirs()
    myLoop.save_current(0)

# prepare monitors
# linear solver residuals are added by default
# divergence
p_corr_eqn.add_monitor("divU_pred")
p_corr_eqn.add_monitor("divU_corr")
# global mass balance
p_corr_eqn.add_monitor("global_mass_cons")
# drag coefficient
p_corr_eqn.add_monitor("cd")
if start_time == 0.:
    U_eqn.save_residual_header("U")
    p_corr_eqn.save_residual_header("p")

# build Laplace term coefficient matrices now as they will not change
U_lapl = implicit_operators.LaplaceVec(U, 1./Re)
p_lapl = implicit_operators.Laplace(p_corr)

# calculate or load PLU decomposition of p_corr_eqn
loadPLU = True
if start_time == 0. and loadPLU == False:
    print "\ncalculating lu decomposition for pressure..."
    tlu = time.time()
    if Config_.__sparse__:
        p_corr_eqn.p, p_corr_eqn.l, p_corr_eqn.u = mylin.lu_decomp(p_lapl.A.todense())
    else:
        p_corr_eqn.p, p_corr_eqn.l, p_corr_eqn.u = mylin.lu_decomp(p_lapl.A)
    print "DONE in "+str(time.time()-tlu)+" s"
    print "saving plu matrices..."
    p_corr_eqn.save_plu()
    print "DONE"
else: # loadPLU == True:
    print "\nloading lu decomposition for pressure..."
    p_corr_eqn.load_plu("square_cylinder_PLU")
    print "DONE"

##################
# iteration loop #
##################
for time_ in myLoop.times:
    myLoop.time = time_
    myLoop.dt = myLoop.find_variable_dt()
    myLoop.print_time()

    # update boundary conditions, reset matrices and right hand sides, update m_dot
    U_eqn.reset()
    p_corr_eqn.reset()
    p.update_boundary_values()

    # induce vortex shedding with a perturbation above the cylinder
```

```
if time_ > 5. and time_ < 5.075:
    init.init_cell_values_in_box(U, numpy.array([5., 8., 0.]),
                                 numpy.array([6., 8.1, 0.1]), [1., 1., 0.])

if Config_.__IMEX__ == False:
    ## PREDICTOR EULER
    # assemble momentum equation
    U_eqn += implicit_operators.DdtEulerVec(U_eqn, myLoop.dt)
    Dt = fields.ScalarField(myMesh, "Dt_time")
    Dt.initialize_cell_with_vector(U_eqn.diag())
    U_eqn += explicit_operators.DivergenceVec(U, m_dot, "UDS")
    # use pre-calculated coefficient matrix for the laplace operator
    U_eqn -= implicit_operators.LaplaceVec(U, 1./Re, "", False)
    U_eqn.A = U_eqn.A - U_lapl.A
    U_eqn += explicit_operators.Gradient(p, one_over_rho)

if Config_.__IMEX__ == True:
    ## PREDICTOR IMEX
    ## convection, gradient: Adams-Bashfort
    ## diffusion:            Crank-Nicolson
    # assemble momentum equation
    U_eqn += implicit_operators.DdtEulerVec(U_eqn, myLoop.dt)
    Dt = fields.ScalarField(myMesh, "Dt_time")
    Dt.initialize_cell_with_vector(U_eqn.diag())
    U_eqn += explicit_operators.DivergenceVec(U,     m_dot,     "MINMOD") * 1.5
    U_eqn -= explicit_operators.DivergenceVec(U_old, m_dot_old, "MINMOD") * 0.5
    # use pre-calculated coefficient matrix for the laplace operator
    U_eqn -= implicit_operators.LaplaceVec(U, 1./Re, "", False) * 0.5
    U_eqn.A = U_eqn.A - U_lapl.A * 0.5
    U_eqn -= explicit_operators.LaplaceVec(U, 1./Re) * 0.5
    U_eqn += explicit_operators.Gradient(p,     one_over_rho) * 1.5
    U_eqn -= explicit_operators.Gradient(p_old, one_over_rho) * 0.5

# solve momentum equation
U_eqn.solve()

#overwrite old m_dot field befor Rhie-Chow interpolation
m_dot_old.copy_field(m_dot)

# calculate predicted massflux field
m_dot = calcfield.MassFlux(U, rho)

## CORRECTOR
# correct m_dot with Rhie-Chow interpolation
# skip this only after a restart!
if time_ != myLoop.times[0] and time_ != 0.:
    D = fields.ScalarField(myMesh, "D")
    D.initialize_cell_with_vector(cell_volumes/U_eqn.diag())
    D_f = calcfield.LinearFaceValue(D)
    grad_p = calcfield.GaussCellGradient(p)
    grad_p_lin_f = calcfield.LinearFaceValue(grad_p)
    grad_p_f = calcfield.GaussFaceGradient(p)
    m_dot.A -= D_f.A * (grad_p_f.dot_Sf()[:,0] - grad_p_lin_f.dot_Sf()[:,0])
    m_dot_old_lin = calcfield.MassFlux(U_old, rho)
    Dt.V /= U_eqn.diag()
    Dt_f = calcfield.LinearFaceValue(Dt)
    m_dot.A += Dt_f.A * (m_dot_old.A - m_dot_old_lin.A)

# monitor predictor divergence
div = calcfield.Divergence(None, m_dot, "")
print "predictor divergence: "+str(max(abs(div.V)))
p_corr_eqn.append_to_monitor(max(abs(div.V)), "divU_pred")
```

```
    # assemble pressure-correction equation
    # use pre-calculated coefficient matrix for the laplace operator
    p_corr_eqn += implicit_operators.Laplace(p_corr, 1., "", False)
    p_corr_eqn.A = p_corr_eqn.A + p_lapl.A
    p_corr_eqn -= explicit_operators.Divergence(None, m_dot, "")
    p_corr_eqn.b[:,0] *= U_eqn.diag() / cell_volumes
    # solve pressure-correction equation
    p_corr_eqn.solve()

    #overwrite p_old field befor pressure correction
    if Config_.__IMEX__ == True:
        p_old.copy_field(p)

    # correct pressure
    p.V += p_corr.V * p_under_relaxation
    # correct velocity components
    p_corr.update_boundary_values()
    grad_p_corr = calcfield.GaussCellGradient(p_corr)
    U.V[:,0] -= grad_p_corr.V[:,0] * cell_volumes/U_eqn.diag() * p_under_relaxation
    U.V[:,1] -= grad_p_corr.V[:,1] * cell_volumes/U_eqn.diag() * p_under_relaxation
    U.V[:,2] -= grad_p_corr.V[:,2] * cell_volumes/U_eqn.diag() * p_under_relaxation
    U.update_boundary_values()
    # correct mass flux with Rhie-Chow interpolation
    if time_ != myLoop.times[0] and time_ != 0.:
        grad_p_corr_f = calcfield.GaussFaceGradient(p_corr)
        grad_p_corr_f_Sf = grad_p_corr_f.dot_abs_Sf()
        for face_ in myMesh.faces:
            if face_.isBnd == True:
                continue
            m_dot.A[face_.id] -= D_f.A[face_.id] * (grad_p_corr_f_Sf[face_.id,0])

    # overwrite old velo field
    U_old.copy_field(U)

    # monitor remaining divergence
    div_ = calcfield.Divergence(None, m_dot, "")
    print "corrector divergence: "+str(max(abs(div_.V)))
    p_corr_eqn.append_to_monitor(max(abs(div_.V)), "divU_corr")

    # monitor drag coefficient
    current_cd = monitors.cd(p, 1., 1., "walls", numpy.array([1., 0., 0.]), 0.1)
    print "cd of cylinder: "+str(current_cd)
    p_corr_eqn.append_to_monitor(current_cd, "cd")

    # monitor global mass ballance
    mass_ballance = monitors.globalMass(m_dot)
    print "global mass ballance: "+str(mass_ballance)
    p_corr_eqn.append_to_monitor(mass_ballance, "global_mass_cons")

    # monitor CFL number
    CFL_ = monitors.CFLMag(U, 0.1, myLoop.dt)
    print "CFL: "+str(CFL_)

    # save results and residual data of time step
    myLoop.save_time()
    p_corr_eqn.append_current_residual(myLoop.time, "p")
    U_eqn.append_current_residual(myLoop.time, "U")

#########################
# end of iteration loop #
#########################

print "\nfinished square cylinder example in "+str(time.time()-t00)+" s"
```

CHAPTER

TEN

APPENDIX 3: LIBRARY DOCUMENTATION

10.1 pyCFD_VTK_tools Package

10.1.1 vtkTools Module

This module provides functions to write ascii VTK or VTU files that can be later loaded for postprocessing in Paraview.

pyCFD_VTK_tools.vtkTools.**save_pvd_collection**(*times_*, *save_path*, *data_path*, *file_name*, *standalone=False*)

function to PVD with time information and references to the saved VTU files

Parameters

- **times** (*list of float*) – list of timesteps to save
- **save_path** (*string*) – location to save PVD
- **data_path** (*string*) – location where VTU files are stored
- **standalone** – default: False, if True time information is skipped, all VTU files from the target directory are saved in the PVD

pyCFD_VTK_tools.vtkTools.**save_vtu_face**(*pointCoords*, *fileNameWithPath*)

function to write VTU file of a face

Parameters

- **pointCoords** – list of vertex coordinates
- **fileNameWithPath** (*string*) – location and file name to save

pyCFD_VTK_tools.vtkTools.**save_vtu_objects**(*objList*, *fileNameWithPath*, *fieldList=None*, *fatherMesh=None*, *writeVector=False*)

function to write VTK file with objects (faces/cells) given in objList

Parameters

- **objList** (pyCFD_mesh.mesh_object.MeshObject (page 66)) – list of mesh objects
- **fileNameWithPath** (*string*) – location and file name to save
- **fieldList** (pyCFD_fields.fields.VolumeField (page 50)) – default None, list of volume fields to save
- **fatherMesh** (pyCFD_mesh.generic_mesh.GenericMesh (page 65)) – default: None, mesh objects' owner mesh
- **writeVector** (*bool*) – default: False, if vector fields should be written

pyCFD_VTK_tools.vtkTools.**save_vtu_vector**(*pointCoords*, *vectCoords*, *fileNameWithPath*)

function to write VTU file of a vector at a point

Parameters

- **pointCoords** (*float*) – list of vertex coordinates
- **vectCoords** (*float*) – list of vector coordinates
- **fileNameWithPath** (*string*) – location and file name to save

pyCFD_VTK_tools.vtkTools.**save_vtu_vector_field**(*objList*, *vectList*, *fileNameWithPath*)
function to write VTU with vector fields

Parameters

- **objList** (pyCFD_mesh.mesh_object.MeshObject (page 66)) – list of mesh objects
- **vectList** (pyCFD_fields.fields.VectorField (page 50)) – list of volume vector fields to save
- **fileNameWithPath** (*string*) – location and file name to save

10.2 pyCFD_calculation Package

10.2.1 time_loop Module

module for time loops

class pyCFD_calculation.time_loop.**TimeLoop**(*field_list*, *start_time*, *stop_time*, *dt*)
basic class for time loops

default constructor for the TimeLoop class

Parameters

- **field_list** (*pyCFD_fields.fields.VolumeField* (page 50)) – list of volume fields
- **start_time** (*float*) – physical time of first time step
- **stop_time** (*float*) – physical time of first time step
- **dt** (*float or list*) – time step (fixed or list)

Note: syntax for changing dt:

dt = [(up_to_0, dt_0), (up_to_1, dt_1)]

e.g.:

dt = [(3.5, 0.005), (20.0, 0.01)]

last dt value is used if calculating further.

---

dt = None
current timestep

fields = None
list of fields

find_variable_dt()
function find list of time steps from a given lookup table

lastTime = None
last timestep

print_time()
print iteration counter, physical time and current time step to screen

```
>>>
================================================
Timestep 81: t = 3.405s | dt = 0.005
================================================
```

saveTimes = None
list of timesteps where fields are saved

save_current(*iter_*)
save current timestep

save_current_fields_as(*iter_*, *dir_name*)
save current fields into directory

save_time()
save timestep if given in save times

startTime = None
first timestep

time = None
current time

times = None
all timesteps

uniform_save_times(*save_dt*, *save_start=0.0*)
set up save times with uniform save steps

Parameters

- **save_dt** (*float*) – save step
- **save_start** (*float*) – default: 0.0, first timestep to save

vtk_start = None
Write vtk file vtk_start+1. Useful when restarting

10.3 pyCFD_config Package

10.3.1 config Module

module for setting global variables

- use sparse the scipy.sparse module instead of numpy arrays
__sparse__ = False
- directory for saving field files
__FIELDDIR__ = “”
- directory of output files
__OUTDIR__ = “”
- directory to save vtk files of iterations
__OUTITERDIR__ = “”
- relative directory of vtk files within the output directory
__OUTITERDIRREL__ = “”
- use IMEX schame
__IMEX__ = False

10.4 pyCFD_fields Package

10.4.1 calculated_fields Module

module for calculated variable fields

class pyCFD_fields.calculated_fields.**Divergence**(*volume_field*, *massflux_field*, *type_*)

Bases: pyCFD_fields.fields.ScalarField (page 49)

returns a scalar field with the divergence of a field

It uses the explicit operator class pyCFD_operators.explicit_operators.Divergence (page 70) and feeds the right hand side values to the return field's cell values.

Constructor

Parameters

- **volume_field** (*pyCFD_fields.fields.VolumeField* (page 50)) – scalar field to calculate the divergence for
- **massflux_field** (*pyCFD_fields.fields.SurfaceScalarField* (page 49)) – massflux field which transports the volume_field over the faces
- **type** (*string*) – scheme to calculate the face value of transporting velocity

class pyCFD_fields.calculated_fields.**GaussCellGradient**(*volume_field*, *nonConjIters_=0*)

Bases: pyCFD_fields.fields.Field (page 48)

Calculate cell gradient of a scalar field using the Gauss theorem resulting in a vector field:

$\int_V \nabla\phi dV = \oint_A \phi_f d\vec{A} = \sum_f \phi_f \vec{A}$

Iterations are performed to correct non-conjunctionality.

Iteration:

1.Face values are guessed assuming that cells are conjunctional:
$\phi_{f'} = g_O * \phi_O + g_N * \phi_N$
, the subscripts stand for Owner and Neighbour. Interpolation weights
g_O and g_N are calculated in
pyCFD_mesh.face.Face.update_gradient_weights() (page 63)

2.From guessed face values the volume gradient is calculated:
$\nabla\phi_O = \frac{1}{V_O} \sum_{nb} \phi_{f'} \vec{S_f}$

3.Update face gradient from cell gradient:
$\nabla\phi_f = g_O * (\nabla\phi)_O + g_N (\nabla\phi)_N$

4.Update the face value using a correction from the $\phi_{f'}$ value
$\phi_f = \phi_{f'} + \nabla\phi_f \cdot (\vec{r_f} - \vec{r_{f'}})$

5.Update $\nabla\phi_O$:
$\nabla\phi_O = \frac{1}{V_O} \sum_{nb} \phi_f \vec{S_f}$

6.Repeat from step 3

Treatment of boundary conditions:

At boundaries the value ϕ_f is known, therefore it is not calculated.

Constructor

Parameters

- **volume_field** (pyCFD_fields.fields.ScalarField (page 49)) – scalar field to calculate the gradient for

- **nonConjIters** (*int*) – default: 0, number of non-conjunctional iteration steps

Returns a field with cell gradient values

Return type pyCFD_fields.fields.VectorField (page 50)

class pyCFD_fields.calculated_fields.**GaussFaceGradient**(*volume_field*, *nonConjIters_=0*)

Bases: pyCFD_fields.fields.Field (page 48)

Calculate face gradient of a scalar using the iterative loop of the GaussCellGradient (page 45) class but resulting in a surface vector field with a stencil reduced to the neighbour cells only.

Reduction of stencil:

$\nabla\phi_f = \overline{\nabla\phi_f} - \left(\overline{\nabla\phi_f} \cdot e_{\vec{ON}}\right) e_{\vec{ON}} + \frac{\phi_N - \phi_O}{|d_{\vec{ON}}|} e_{\vec{ON}}$

, where O refers to the owner cell, while N to the neighbour cell. $d_{\vec{ON}}$ is the vector pointing from O's centroid to N's centroid and $e_{\vec{ON}}$ is the unit vector in the same direction.

Constructor

Parameters **volume_field** (pyCFD_fields.fields.ScalarField (page 49)) – scalar field to calculate the gradient for

Returns a field including gradient face values

Return type pyCFD_fields.fields.SurfaceVectorField (page 49)

class pyCFD_fields.calculated_fields.**HRSFaceValue**(*volume_field*, *massflux_field*, *scheme_*)

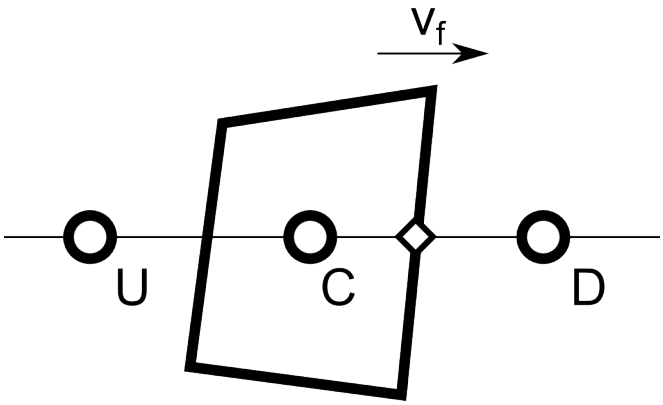
Bases: pyCFD_fields.fields.Field (page 48)

High Resolution interpolation of volume scalar fields resulting in surface field

Interpolation:

•cell gradient of the volume scalar field is calculated using GaussCellGradient (page 45)

•based on the face massflux central and downwind directions are decided



•upwind value is calculated based on the cell gradient and distance of C and D:
$\phi_U = \phi_D - \nabla\phi_P \cdot 2d_{\vec{CD}}$

•normalized ϕ is calculated:
$\tilde{\phi_C} = \frac{\phi_C - \phi_D}{\phi_D - \phi_U}$

•normalized face ϕ is calculated by according function (for UDS $\tilde{\phi_f} = \tilde{\phi_C}$).
Available functions:

–STOIC() (page 47)

–MINMOD() (page 47)

•from the normalized face value face value is calculated:
$\phi_f = \tilde{\phi_f}\,(\phi_D - \phi_U) + \phi_U$

Constructor

Parameters

- **volume_field** (*pyCFD_fields.fields.ScalarField* (page 49)) – scalar field to calculate face values for
- **massflux_field** (*pyCFD_fields.fields.SurfaceScalarField* (page 49)) – surface field with massflux values

Returns a field including face values from STOIC scheme

Return type pyCFD_fields.fields.Field

MINMOD (*phi_tilda_c*)
the MINMOD high resolution scheme

•for $0 \leq \tilde{\phi_C} \leq 0.5$
$\tilde{\phi_f} = 3/2 * \tilde{\phi_C}$

•for $0.5 < \tilde{\phi_C} \leq 1$
$\tilde{\phi_f} = 0.5 + 0.5 * \tilde{\phi_C}$

•$\tilde{\phi_f} = \tilde{\phi_C}$ elsewhere

STOIC (*phi_tilda_c*)
the STOIC high resolution scheme

•for $0 \leq \tilde{\phi_C} \leq 0.2$
$\tilde{\phi_f} = 3 * \tilde{\phi_C}$

•for $0.2 < \tilde{\phi_C} \leq 0.5$
$\tilde{\phi_f} = 0.5 + 0.5 * \tilde{\phi_C}$

•for $0.5 < \tilde{\phi_C} \leq 5/6$
$\tilde{\phi_f} = 3/8 + 0.75 * \tilde{\phi_C}$

•for $5/6 < \tilde{\phi_C} \leq 1$
$\tilde{\phi_f} = 1$

•$\tilde{\phi_f} = \tilde{\phi_C}$ elsewhere

class pyCFD_fields.calculated_fields.**LinearFaceValue**(*volume_field*)
Bases: pyCFD_fields.fields.Field (page 48)

linear interpolation of volume fields resulting in a surface field

Interpolation:

Linear interpolation is calculated using weight factors for the neighbour cell values:

$$\phi_f = g_O * \phi_O + g_N * \phi_N$$

, boundary values are taken from the volue field at the boundary.

Interpolation weights are calculated in pyCFD_mesh.face.Face.update_weights() (page 64).

Constructor

Parameters **volume_field** (*pyCFD_fields.fields.ScalarField* (page 49)) – scalar field to calculate face values for

Returns a field with face values from linear interpolation

Return type pyCFD_fields.fields.Field

class pyCFD_fields.calculated_fields.**MassFlux**(*velo_field*, *rho_field*)
Bases: pyCFD_fields.fields.SurfaceScalarField (page 49)

Calculate the massflux surface field by:
$\dot{m} = \rho_f * \vec{u_f} \cdot \vec{S_f}$
, where ρ_f and $\vec{u_f}$ are the values at the face

from linear interpolation using the LinearFaceValue (page 47) class.

Constructor

Parameters

- **velo_field** (*pyCFD_fields.fields.VectorField* (page 50)) – vector field with velocity values
- **rho_field** (*pyCFD_fields.fields.ScalarField* (page 49)) – scalar field with density values

Returns a surface field with the mass flux values

Return type pyCFD_fields.fields.SurfaceScalarField

class pyCFD_fields.calculated_fields.**UpwindFaceValue**(*volume_field*, *massflux_field*)
Bases: pyCFD_fields.fields.Field (page 48)

upwind interpolation of volume scalar fields resulting in a surface field

Interpolation:

Interpolation is decided based on the value of massflux_field at the faces:

•if the massflux is negative the neighbour cell value is taken

•otherwise the owner cell value is taken

Constructor

Parameters

- **volume_field** (*pyCFD_fields.fields.ScalarField* (page 49)) – scalar field to calculate face values for
- **massflux_field** (*pyCFD_fields.fields.SurfaceScalarField* (page 49)) – surface field with massflux values

Returns a field including face values from upwind interpolation

Return type pyCFD_fields.fields.Field

10.4.2 fields Module

module for abstract variable fields

class pyCFD_fields.fields.**Field**
basic class for fields

A = None
surface values

dot_Sf()
return the dot products of face vector values with face normals as scalar vector

dot_abs_Sf()
return the dot products of face vector values with absolute face normals as scalar vector

father = None
reference to owner mesh object

get_patch (*patch_name*)
return a reference to the field patch with matching name

Parameters **patch_name** (*str*) – patch name of geometric patch (without '__fieldName')

name = None
string for field name

patches = None
references to field patches

type = None
string for field type: scalar or vector

class pyCFD_fields.fields.**ScalarField** (*mesh_*, *field_name*, *init_value=0.0*)
Bases: pyCFD_fields.fields.VolumeField (page 50)

class for scalar fields

Constructor

Parameters

- **mesh** (pyCFD_mesh.generic_mesh.GenericMesh (page 65)) – owner mesh object
- **field_name** (*str*) – name of resulting field
- **init_value** (*float*) – default: 0, initial value

initialize_cell_with_vector (*values*)
set field cell values to the values given as input

initialize_face_with_vector (*values*)
set field face values to the values given as input

patches = None
reference to patches of field

class pyCFD_fields.fields.**SurfaceScalarField** (*mesh_*, *field_name*, *init_value=0.0*)
Bases: pyCFD_fields.fields.Field (page 48)

class for surface scalar fields

Constructor

Parameters

- **mesh** (pyCFD_mesh.generic_mesh.GenericMesh (page 65)) – owner mesh object
- **field_name** (*str*) – name of resulting field
- **init_value** (*float*) – default: 0, initial value

copy_field (*other_field*)

initialize_with_vector (*values*)
set field face values to the values given as input

class pyCFD_fields.fields.**SurfaceVectorField** (*mesh_*, *field_name*, *init_value=array([0., 0., 0.])*)
Bases: pyCFD_fields.fields.Field (page 48)

class for surface vector fields

Constructor

Parameters

- **mesh** (pyCFD_mesh.generic_mesh.GenericMesh (page 65)) – owner mesh object
- **field_name** (*str*) – name of resulting field
- **init_value** (*numpy.array*) – default: numpy.zeros(3), initial value

copy_field(*other_field*)

initialize_with_vector(*values*)
set field face values to the values given as input

class pyCFD_fields.fields.**VectorField**(*mesh_*, *field_name*, *init_value=array([0., 0., 0.])*)
Bases: pyCFD_fields.fields.VolumeField (page 50)

class for vector fields

Constructor

Parameters

- **mesh** (pyCFD_mesh.generic_mesh.GenericMesh (page 65)) – owner mesh object
- **field_name** (*str*) – name of resulting field
- **init_value** (*numpy.array*) – default: numpy.zeros(3), initial value

get_component_as_scalar_field(*component_*)
return one component of a volume vector field as a volume scalar field

patches = None
reference to patches of field

class pyCFD_fields.fields.**VolumeField**
Bases: pyCFD_fields.fields.Field (page 48)

class for volume fields

V = None
cell values

copy_field(*other_field*)

load_init_fields(*name_=''*)
load initial field values and boundary conditions from saved .npy files

Parameters **name** (*str*) – name of directory to load the files from located in pyCFD_config.config.__FIELDDIR__

update_boundary_values()
update boundary values

10.4.3 initialization Module

module for field initialization methods

pyCFD_fields.initialization.**init_cell_values_in_box**(*field*, *box_min*, *box_max*, *init_value*)
Re-initialize field values within a box selection

Parameters

- **field** (pyCFD_fields.fields.ScalarField (page 49)) – a scalar field
- **box_min** (*numpy.array*) – lower bounding coordinates of the box
- **box_max** (*numpy.array*) – higher bounding coordinates of the box
- **init_value** (*float*) – desired initial value

pyCFD_fields.initialization.**init_linear_scalar_sphere_distribution**(*field*, *center_value*, *side_value*, *center_coords*, *side_distance*)

Initialize a field with a spherical linear distribution. Cell and face centers are calculated based on distance from distribution center

Parameters

- **field** (pyCFD_fields.fields.ScalarField (page 49)) – a scalar field
- **center_value** (*float*) – value at the center of the distribution
- **side_value** (*float*) – value at the side of the distribution
- **center_coords** (*numpy.array*) – coordinates of the center of the distribution
- **side_distance** (*float*) – distance fro center to side

10.5 pyCFD_general.cython_boost_linux2 package

10.5.1 pyCFD_general.cyt...cy_general module

```
"""
cython module for boosting general tasks
"""
__author__ = "Bence Somogyi"
__copyright__ = "Copyright 2014"
__version__ = "0.1"
__maintainer__ = "Bence Somogyi"
__email__ = "bencesomogyi@ivt.tugraz.at"
__status__ = "Prototype"
# from __future__ import division
# cimport numpy
# import numpy
# cython: embedsignature=True
cimport cython

DTYPE = float
ctypedef float DTYPE_t

@cython.boundscheck(False)

def list_index(list myList, DTYPE_t value):
    """
    find index of defined value in a list

    :param myList: list of values
    :type myList:  float
    :param value:  value to search the index for
    :typevalue:    float

    .. note:
        cython code to compile c library
    """
    cdef unsigned int n = len(myList)
    for i in xrange(n):
        if myList[i] == value:
            return i
```

10.6 pyCFD_general package

10.6.1 Subpackages

10.6.2 Module contents

10.7 pyCFD_geometric_tools.cython_boost_linux2 package

10.7.1 pyCFD_geometric_tools.c...cy_geometric_tools module

cython module for calculations in the geomTools module

```
"""
cython module for calculations in the geomTools module
"""
__author__ = "Bence Somogyi"
__copyright__ = "Copyright 2014"
__version__ = "0.1"
__maintainer__ = "Bence Somogyi"
__email__ = "bencesomogyi@ivt.tugraz.at"
__status__ = "Prototype"
# from __future__ import division
cimport numpy
import numpy
cimport cython

DTYPE = numpy.float
ctypedef numpy.float_t DTYPE_t

cdef extern from "math.h":
    float sqrt (float x)
    float abs (float x)

@cython.boundscheck(False)
# @cython.cdivision(True)

def cy_triangle_areas(numpy.ndarray[DTYPE_t, ndim=2] area_vector):
    """
    calculate triangle area from area vector

    :param area_vector: array of area vector
    :type area_vector:  float
    :return:            area
    :rtype:             float

    .. note:
        cython code to compile c library
    """
    cdef unsigned int n = len(area_vector)
    cdef numpy.ndarray[DTYPE_t, ndim=1] triangle_areas = numpy.zeros(n)
    for i in xrange(n):
        triangle_areas[i] = sqrt(area_vector[i,0]*area_vector[i,0] + area_vector[i,1]*area_vector
    return triangle_areas

def cy_area_vector(numpy.ndarray[DTYPE_t, ndim=2] centroid, numpy.ndarray[DTYPE_t, ndim=2] coords_
    """
    calculate triangle area vector from coordinates of 3 vertices

    :param centroid: coordinate array of node 1
    :type centroid:  float
```

```
    :param coords_a: coordinate array of node 2
    :type coords_a:  float
    :param coords_b: coordinate array of node 3
    :type coords_b:  float
    :return:         array of area vector
    :rtype:          float

    .. note:
        cython code to compile c library
    """
    cdef unsigned int n = len(centroid)
    cdef numpy.ndarray[DTYPE_t, ndim=2] area_vector = numpy.zeros((n,3))
    cdef numpy.ndarray[DTYPE_t, ndim=1] vec_a = numpy.zeros(3)
    cdef numpy.ndarray[DTYPE_t, ndim=1] vec_b = numpy.zeros(3)
    for i in xrange(n):
        vec_a[0] = centroid[i,0] - coords_a[i,0]
        vec_a[1] = centroid[i,1] - coords_a[i,1]
        vec_a[2] = centroid[i,2] - coords_a[i,2]
        vec_b[0] = centroid[i,0] - coords_b[i,0]
        vec_b[1] = centroid[i,1] - coords_b[i,1]
        vec_b[2] = centroid[i,2] - coords_b[i,2]
        area_vector[i,0] = ((vec_a[1] * vec_b[2]) - (vec_a[2] * vec_b[1])) * 0.5
        area_vector[i,1] = ((vec_a[2] * vec_b[0]) - (vec_a[0] * vec_b[2])) * 0.5
        area_vector[i,2] = ((vec_a[0] * vec_b[1]) - (vec_a[1] * vec_b[0])) * 0.5
    return area_vector

def cy_value_close(DTYPE_t value, DTYPE_t compare):
    """
    cython implementation of numpy.allclose()

    Returns True if two values are equal within a tolerance.

    The tolerance value is positive, typically a very small number. The
    relative difference (rtol * abs(compare)) and the absolute difference atol
    are added together to compare against the absolute difference between value
    and compare.

    **Tolerances**

    * rtol = 1e-05

    * atol = 1e-08

    :param value:   value to be compared
    :type value:    float
    :param compare: value to compare with
    :type compare:  float

    .. note:
        cython code to compile c library
    """
    cdef DTYPE_t rtol = 1e-05
    cdef DTYPE_t atol = 1e-08
    if abs(value - compare) <= (atol + rtol * abs(compare)):
        return True
    return False

def cy_calc_cos(numpy.ndarray[DTYPE_t, ndim=1] areaVect, numpy.ndarray[DTYPE_t, ndim=1] vectElmFa
    """
    calculate cosine between face are vector and the vector between the cell
    centroid and the face centroid to decide if area vector points inside or
    outside the cell
```

```
    :param areaVect:    coordinate array of the area vector
    :type areaVect:     float
    :param vectElmFace: coordinate array of the vector between cell and face centroids
    :type vectElmFace:  float
    :return:            cos value between the vectors
    :rtype:             float

    .. note:
        cython code to compile c library
    """
    cdef DTYPE_t areaVect_norm = sqrt(areaVect[0]*areaVect[0] + areaVect[1]*areaVect[1] + areaVect
    cdef DTYPE_t vectElmFace_norm = sqrt(vectElmFace[0]*vectElmFace[0] + vectElmFace[1]*vectElmFa
    cdef DTYPE_t cos_value = ( areaVect[0]*vectElmFace[0] + areaVect[1]*vectElmFace[1] + areaVect
    return cos_value
```

10.8 pyCFD_geometric_tools package

10.8.1 pyCFD_geometric_tools.geomTools module

module for geometric calculations as face areas, cell volumes and centroid coordinates

pyCFD_geometric_tools.geomTools.**face_area_vect_vert**(*vert_list*, *centroid_of_element*)

calculate the face normal area vector of a face with respect to the centroid of an element: face normal points outwards

Depending of the number of faces `triangle_area_vect_vert()` (page 57) or func:*quadrangle_area_vect_vert* is called. Other faces are not yet supported.

Parameters

- **vertList** (`pyCFD_mesh.vertex.Vertex` (page 68)) – list of vertex objects
- **centroidOfElement** (*numpy.array*) – array with cell centroid coordinate

Returns array of face area vector

Type numpy.array

pyCFD_geometric_tools.geomTools.**hexahedronCentroid**(*ndCoordList*)

calculate coordinates of the centroid of a hexahedron from list of coordinates of its vertices

Parameters **ndCoordList** (*numpy.array*) – list of arrays with node coordinates

Returns array with centroid coordinate

Type numpy.array

pyCFD_geometric_tools.geomTools.**hexahedronVolume**(*ndCoordList*, *centroid*)

calculate the volume of a hexahedron from the coordinates of its vertices and its centroid

The hexaherdon is subdivided into tetrahedons using the centroid of the hexahedron. Sub-tetrahedron volumes are calculated with `tetrahedronVolume()` (page 57) and summed up.

Parameters

- **ndCoordList** (*numpy.array*) – list of arrays with node coordinates
- **centroid** (*numpy.array*) – array with hexahedron centroid

Returns volume of tetrahedron

Type float

pyCFD_geometric_tools.geomTools.**hexahedron_centroid_vert**(*vertList*)

calculate coordinates of the centroid of a hexahedron from a list of vertex objects

Parameters **vertList** (pyCFD_mesh.vertex.Vertex (page 68)) – list of vertex objects

Returns array with centroid coordinate

Type numpy.array

pyCFD_geometric_tools.geomTools.**hexahedron_volume_vert**(*vertList*)
calculate the volume of a hexahedron from its vertices

The hexahedron is subdivided into tetrahedons. Sub-tetrahedron volumes are calculated with tetrahedron_volume_vert() (page 57) and summed up.

Parameters **vertList** (pyCFD_mesh.vertex.Vertex (page 68)) – list of vertex objects

Returns volume of hexahedron

Type float

pyCFD_geometric_tools.geomTools.**polygon_centroid_and_area**(*vertex_list*)
calculate the centroid and area of a polygon from the list of its vertices

Calculation steps:

•polygon is subdivided into triangles using the geometric centroid of the polygon

•centers of the sub-triangles and their area is calculated

•polygon centroid is calculated as surface weighted average of the sub triangles

•polygon area is calculated as the sum of sub triangle areas

Parameters **vertex_list** (pyCFD_mesh.vertex.Vertex (page 68)) – list of vertex objects defining the face

Returns dictionary [weighted centroid, area]

Return type float

pyCFD_geometric_tools.geomTools.**polyhedron_centroid_and_volume**(*cell*, *face_list*)
calculate the centroid and volume of a polygon from the list of its faces

Calculation steps

•polyhedra is subdivided into pyramids using the geometric centroid of the polyhedra. Therefore the cell itself is needed as input with its vertices updated.

•centers of the sub-pyramids and their volume is calculated

•polyhedra centroid is calculated as volume weighted average of the sub pyramids

•polyhedra volume is calculated as the sum of sub pyramids volumes

Parameters

- **cell** (*pyCFD_mesh.cell.Cell* (page 61)) – incomplete cell object
- **face_list** (*dict*) – list of face objects defining the cell

Returns dictionary [weighted centroid, volume]

Return type dict

pyCFD_geometric_tools.geomTools.**prism_centroid_vert**(*vertList*)
calculate coordinates of the centroid of a prism from a list of vertex objects

Parameters **vertList** (pyCFD_mesh.vertex.Vertex (page 68)) – list of vertex objects

Returns array with centroid coordinate

Type numpy.array

pyCFD_geometric_tools.geomTools.**prism_volume_vert**(*vertList*)
calculate the volume of a prism from its vertices

The prism is subdivided into tetrahedons. Sub-tetrahedron volumes are calculated with tetrahedron_volume_vert() (page 57) and summed up.

Parameters **vertList** (pyCFD_mesh.vertex.Vertex (page 68)) – list of vertex objects

Returns volume of prism

Type float

pyCFD_geometric_tools.geomTools.**quadrangleAreaVect**(*ndCoordList*, *centroidOfElement*)
calculate the face normal area vector of a quadrangle with respect to the centroid of an element: face normal points outwards

Parameters

- **ndCoordList** (*numpy.array*) – list of arrays with node coordinates
- **centroidOfElement** (*numpy.array*) – array with cell centroid coordinate

Returns array of face area vector

Type numpy.array

pyCFD_geometric_tools.geomTools.**quadrangleCentroid**(*ndCoordList*)
calculate coordinates of the centroid of a 3D quadrangle from list of coordinates of its vertices

Parameters **ndCoordList** (*numpy.array*) – list of arrays with node coordinates

Returns array with centroid coordinate

Type numpy.array

pyCFD_geometric_tools.geomTools.**quadrangle_area_vect_face**(*face_*, *centroid_of_element*)
calculate the face normal area vector of a quadrangle with respect to the centroid of an element: face normal points outwards

Parameters

- **face** (pyCFD_mesh.face.Face (page 62)) – a face object
- **centroidOfElement** (*numpy.array*) – array with cell centroid coordinate

Returns array of face area vector

Type numpy.array

pyCFD_geometric_tools.geomTools.**quadrangle_area_vect_vert**(*vertList*, *centroidOfElement*)
calculate the face normal area vector of a quadrangle with respect to the centroid of an element: face normal points outwards

Parameters

- **vertList** (pyCFD_mesh.vertex.Vertex (page 68)) – list of vertex objects
- **centroidOfElement** (*numpy.array*) – array with cell centroid coordinate

Returns array of face area vector

Type numpy.array

pyCFD_geometric_tools.geomTools.**quadrangle_centroid_vert**(*vertList*)
calculate coordinates of the centroid of a 3D quadrangle from a list of vertex objects

Parameters **vertList** (pyCFD_mesh.vertex.Vertex (page 68)) – list of vertex objects

Returns array with centroid coordinate

Type numpy.array

pyCFD_geometric_tools.geomTools.**tetrahedronCentroid**(*ndCoordList*)
calculate coordinates of the centroid of a tetrahedron from list of coordinates of its vertices

Parameters **ndCoordList** (*numpy.array*) – list of arrays with node coordinates

Returns array with centroid coordinate

Type numpy.array

pyCFD_geometric_tools.geomTools.**tetrahedronVolume**(*ndCoordList*)
the volume of a tetrahedron from the coordinates of its vertices

Parameters **ndCoordList** (*numpy.array*) – list of arrays with node coordinates

Returns volume of tetrahedron

Type float

pyCFD_geometric_tools.geomTools.**tetrahedron_centroid_vert**(*vertList*)
calculate coordinates of the centroid of a tetrahedron from a list of vertex objects

Parameters **vertList** (pyCFD_mesh.vertex.Vertex (page 68)) – list of vertex objects

Returns array with centroid coordinate

Type numpy.array

pyCFD_geometric_tools.geomTools.**tetrahedron_volume_vert**(*vertList*)
calculate the volume of a tetrahedron from its vertices

Parameters **vertList** (pyCFD_mesh.vertex.Vertex (page 68)) – list of vertex objects

Returns volume of tetrahedron

Type float

pyCFD_geometric_tools.geomTools.**triangleAreaVect**(*ndCoordList*, *centroidOfElement*)
calculate the face normal area vector of a triangle with respect to the centroid of an element: face normal points outwards

Parameters

- **ndCoordList** (*numpy.array*) – list of arrays with node coordinates
- **centroidOfElement** (*numpy.array*) – array with cell centroid coordinate

Returns array of face area vector

Type numpy.array

pyCFD_geometric_tools.geomTools.**triangleCentroid**(*ndCoordList*)
calculate coordinates of the centroid of a 3D triangle from list of coordinates of its vertices

Parameters **ndCoordList** (*numpy.array*) – list of arrays with node coordinates

Returns array with centroid coordinate

Type numpy.array

pyCFD_geometric_tools.geomTools.**triangle_area_vect_face**(*face_*, *centroid_of_element*)
calculate the face normal area vector of a triangle with respect to the centroid of an element: face normal points outwards

Parameters

- **face** (pyCFD_mesh.face.Face (page 62)) – a face object
- **centroidOfElement** (*numpy.array*) – array with cell centroid coordinate

Returns array of face area vector

Type numpy.array

pyCFD_geometric_tools.geomTools.**triangle_area_vect_vert**(*vertList*, *centroidOfElement*)
calculate the face normal area vector of a triangle with respect to the centroid of an element: face normal points outwards

Parameters

- **vertList** (pyCFD_mesh.vertex.Vertex (page 68)) – list of vertex objects
- **centroidOfElement** (*numpy.array*) – array with cell centroid coordinate

Returns array of face area vector

Type numpy.array

pyCFD_geometric_tools.geomTools.**triangle_centroid_vert**(*vertList*)
calculate coordinates of the centroid of a 3D triangle from a list of vertex objects

Parameters **vertList** (pyCFD_mesh.vertex.Vertex (page 68)) – list of vertex objects

Returns array with centroid coordinate

Type numpy.array

10.9 pyCFD_linear_solvers.cython_boost_linux2 package

10.9.1 pyCFD_linear_solvers.c...cy_linear_solvers module

```
# from __future__ import division
import numpy
cimport numpy
cimport cython
from libc.math cimport abs

DTYPE = numpy.float
ctypedef numpy.float_t DTYPE_t

#cdef extern from "math.h":
#    float abs (float x)

@cython.boundscheck(False)
# @cython.cdivision(True)

def gs_sparse_loop(numpy.ndarray[int, ndim=1] row_indices,
                   numpy.ndarray[int, ndim=1] column_indices,
                   numpy.ndarray[DTYPE_t, ndim=1] matrix_values,
                   numpy.ndarray[DTYPE_t, ndim=1] diagonal_values,
                   numpy.ndarray[DTYPE_t, ndim=1] b,
                   numpy.ndarray[DTYPE_t, ndim=1] x0,
                   int max_iter,
                   DTYPE_t tol,
                   DTYPE_t delta_x                                ):
    """
    iterative solver for linear system of equations using Gauss-Seidel
    iterations. Two sub-iterations are performed in each iteration steps: once
    starting from front, once starting from rear.

    :param row_indices:     row indices of the non zero sparse matrix elements
    :type row_indices:      numpy.array
    :param column_indices:  column indices of the non zero sparse matrix elements
    :type column_indices:   numpy.array
    :param matrix_values:   non zero matrix coefficients
    :type matrix_values:    numpy.array
```

```
    :param diagonal_values: diagonal values of the sparse matrix
    :type diagonal_values:  numpy.array
    :param b:               right hand side of the equation system
    :type b:                numpy.array
    :param x0:              initial condition
    :type x0:               numpy.array
    :param max_iter:        maximum number of iterations
    :type max_iter:         int
    :return:                solution vector of the equation system
    :rtype:                 numpy.array
    """
    cdef unsigned int n = len(x0)
    cdef numpy.ndarray[DTYPE_t, ndim=1] x = x0
    cdef DTYPE_t sum_
    cdef DTYPE_t new_x = 0.
    cdef int iter_ = 0
    while iter_ < max_iter:
        delta_x = 0.
        for i in xrange(n):
            sum_ = 0.0
            j_for_i = (numpy.where(row_indices==i))[0]
            for j_ in j_for_i:
                j = column_indices[j_]
                if j != i:
                    sum_ = sum_ + matrix_values[j_]*x[j]
            new_x = 1./diagonal_values[i] * (b[i] - sum_)
            if abs(new_x - x[i]) > delta_x:
                delta_x = abs(new_x - x[i])
            x[i] = new_x
        if tol > delta_x:
            break
        delta_x = 0.
        for i in xrange(n-1,-1,-1):
            sum_ = 0.0
            j_for_i = (numpy.where(row_indices==i))[0]
            for j_ in j_for_i:
                j = column_indices[j_]
                if j != i:
                    sum_ = sum_ + matrix_values[j_]*x[j]
            new_x = 1./diagonal_values[i] * (b[i] - sum_)
            if abs(new_x - x[i]) > delta_x:
                delta_x = abs(new_x - x[i])
            x[i] = new_x
        if tol > delta_x:
            break
        iter_ += 1
    return x, iter_, delta_x

def lu_solver_backward_loop(numpy.ndarray[DTYPE_t, ndim=1] b_,
                            numpy.ndarray[DTYPE_t, ndim=2] l_,
                            int n                           ):
    """
    backward substitution loop for the lu direct solver

    :param b_: right hand side
    :type b_:  numpy.array
    :param l_: lower triangular matrix
    :type l_:  numpy.array
    :param n:  length of the unknown vector
    :type n:   int
    :return:   intermediate solution
    :rtype:    numpy.array
    """
```

```
    cdef numpy.ndarray[DTYPE_t, ndim=1] y = numpy.zeros(n, DTYPE)
    cdef DTYPE_t sum_
    for i in xrange(0,n):
        sum_ = 0.
        if i != 0:
            for j in range(0,i):
                sum_ += l_[i,j] * y[j]
        y[i] = (b_[i] - sum_) / l_[i,i]
    return y

def lu_solver_forward_loop(numpy.ndarray[DTYPE_t, ndim=1] b_,
                           numpy.ndarray[DTYPE_t, ndim=1] y,
                           numpy.ndarray[DTYPE_t, ndim=2] u_,
                           int n                             ):
    """
    forward substitution loop for the lu direct solver

    :param b_: right hand side
    :type b_:  numpy.array
    :param y:  intermediate solution
    :type y:   numpy.array
    :param n:  length of the unknown vector
    :type n:   int
    :return:   final solution
    :rtype:    numpy.array
    """
    cdef numpy.ndarray[DTYPE_t, ndim=1] x = numpy.zeros(n, DTYPE)
    cdef DTYPE_t sum_
    for i in range(n-1,-1,-1):
        sum = 0.
        if i != n-1:
            for j in range(n-1,i-1,-1):
                sum += u_[i,j] * x[j]
        x[i] = (y[i] - sum) / u_[i,i]
    return x
```

10.10 pyCFD_linear_solvers package

10.10.1 pyCFD_linear_solvers.linear_solvers module

module for linear equation solvers

pyCFD_linear_solvers.linear_solvers.**gs**(*A*, *b*, *x0*, *tol*, *max_iter*)

iterative solver for linear system of equations using Gauss-Seidel iterations. Two sub-iterations are performed in each iteration steps: once starting from front, once starting from rear.

The sparse solution is implemented in cython.

Parameters

- **A** (*numpy.array*) – coefficient matrix
- **b** (*numpy.array*) – right hand side
- **x0** (*numpy.array*) – initial condition vector
- **tol** (*float*) – absolute tolerance between two iterations
- **max_iter** (*int*) – maximum number of iterations

Returns solution vector of the equation system

Return type numpy.array

pyCFD_linear_solvers.linear_solvers.**lu_decomp**(*A*)
calculate LU decomposition of dense matrix A with pivoting

Parameters **A** (*numpy.array*) – array to be decomposed

Returns permutation matrix, lower triangular matrix, upper triangular matrix

Return type numpy.array

pyCFD_linear_solvers.linear_solvers.**lu_decomp_sparse**(*A*)
calculate LU decomposition of sparse matrix A with pivoting

Parameters **A** (*scipy.sparse matrix*) – array to be decomposed

Returns permutation matrix, lower triangular matrix, upper triangular matrix

Return type scipy.sparse matrix

pyCFD_linear_solvers.linear_solvers.**lu_solver**(*A*, *b*)
direct solver for linear system of equations using LU decomposition and backward and forward substituion.

Calculation steps for the equation system $A\phi = b$:

•the coefficient matrix is decomposed: $A = PLU$

•forward substitution to solve for y: $Ly = Pb$

•backward substitution to solve for x: $Ux = y$

Parameters

- **A** (*numpy.array*) – coefficient matrix
- **b** (*numpy.array*) – right hand side

Returns solution vector of the equation system

Return type numpy.array

pyCFD_linear_solvers.linear_solvers.**lu_solver_plu**(*p_*, *l_*, *u_*, *b*)
direct solver for linear system of equations using an existing LU decomposition of the coefficient matrix A. Result is obtained via backward and forward substituion. The substitution loops are implemented as cython functions in the pyCFD_linear_solvers.cy_boost_linux2.cy_linear_solvers: lu_solver_backward_loop and lu_solver_forward_loop.

Parameters

- **p** (*numpy.array*) – permutation matrix
- **l** (*numpy.array*) – lower triangular matrix
- **u** (*numpy.array*) – upper triangular matrix
- **b** (*numpy.array*) – right hand side

Returns solution vector of the equation system

Return type numpy.array

10.11 pyCFD_mesh package

10.11.1 pyCFD_mesh.cell module

module for mesh cells

class pyCFD_mesh.cell.**Cell**(*obj_list*)
Bases: pyCFD_mesh.mesh_object.MeshObject (page 66)

class for cells

Constructor

Parameters **obj_list** (pyCFD_mesh.vertex.Vertex (page 68) or pyCFD_mesh.face.Face (page 62)) – list of objects defining the cell

C = None
cell centroid coordinates

V = None
cell volume [m3]

create_cell_from_faces(*face_list*)
create cell from faces

Parameters **vertexList** (pyCFD_mesh.face.Face (page 62)) – list of faces

create_cell_from_vertices(*vertexList*)
create cell from vertices

Parameters **vertexList** (pyCFD_mesh.vertex.Vertex (page 68)) – list of vertices

faces = None
list faces of the cell

id = None
cell id

print_face_ids()
print the ids of the cell's faces.

print_vertex_ids()
print the ids of the cell's vertices.

vertices = None
list of vertices defining the cell

pyCFD_mesh.cell.**set_father_of_cell_faces**(*cell*)
set father for all faces of a cell

Parameters **cell** (Cell (page 61)) – a cell object

10.11.2 pyCFD_mesh.face module

module for cell faces

class pyCFD_mesh.face.**Face**(*vertexList*)
Bases: pyCFD_mesh.mesh_object.MeshObject (page 66)

class for cell faces

Constructor

Parameters **vertexList** (pyCFD_mesh.vertex.Vertex (page 68)) – list of vertices defining the face

A = None
face areas [m2]

C = None
face centroid

Sf = None
face area vectors [m2]

bndId = None
boundary ID of face

cells = None
list of connected cells

ffToF = None
is the shortest vector starting from the line between owner and neighbour pointing to the face centroid

find_neighbour_vertex_on_face(*vertex_*)
returns a list of vertex objects which are the neighbours of reference vertex on the same face

Parameters **vertex** (pyCFD_mesh.vertex.Vertex (page 68)) – reference vertex

Returns list vertex objects

Return type pyCFD_mesh.vertex.Vertex (page 68)

get_Sf(*cell_*)
return face area vector pointing out from one of the cells that own the face

Parameters **cell** (pyCFD_mesh.cell.Cell (page 61)) – reference cell

Returns array of face area vector

Return type numpy.array

get_Sf_sign(*cell_*)
return

- 1.0 if cell is the owner
- -1.0 if cell is the neighbour

Parameters **cell** (pyCFD_mesh.cell.Cell (page 61)) – reference cell

Returns 1.0 or -1.0

Return type float

get_cell_ids()
return the id of the cell which owns the face

Returns id of the owner cell

Return type int

get_vertex_ids()
return the ids of the face's vertices.

Returns list of vertex indices

Return type int

gradWeights = None
weights for Gauss gradient correction [w_owner, w_neigbour]

id = None
face id

inPatchId = None
face id in patch face list

isBnd = None
bool for boundary faces

print_vertex_ids()
print the ids of the face's vertices.

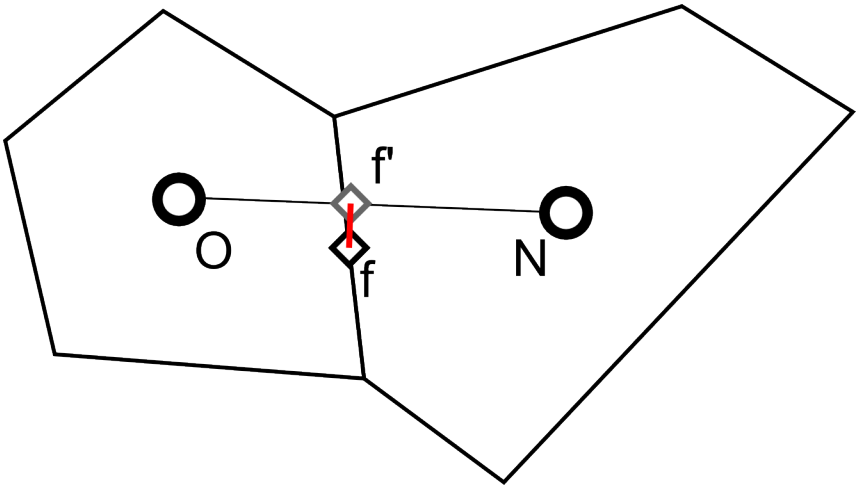
update_Sf()
update surface vectors

update_gradient_weights()

update non-conjunctional interpolation location to be used in Gauss gradient iterations

The vector $r_{f'}^{\vec{}}$ belongs to the face and is calculated by:

$$r_{f'}^{\vec{}} = r_O^{\vec{}} + \frac{r_{Of}^{\vec{}} \cdot r_{ON}^{\vec{}}}{r_{ON}^{\vec{}} \cdot r_{ON}^{\vec{}}} (r_O^{\vec{}} - r_N^{\vec{}})$$



, $r_{ff'}^{\vec{}}$ is chosen to be perpendicular to $r_{ON}^{\vec{}}$.

Interpolation weights based on this fictious point f' are calculated the following way:

- for the owner:

  $g_O = \frac{|r_N^{\vec{}} - r_{f'}^{\vec{}}|}{|r_N^{\vec{}} - r_O^{\vec{}}|}$

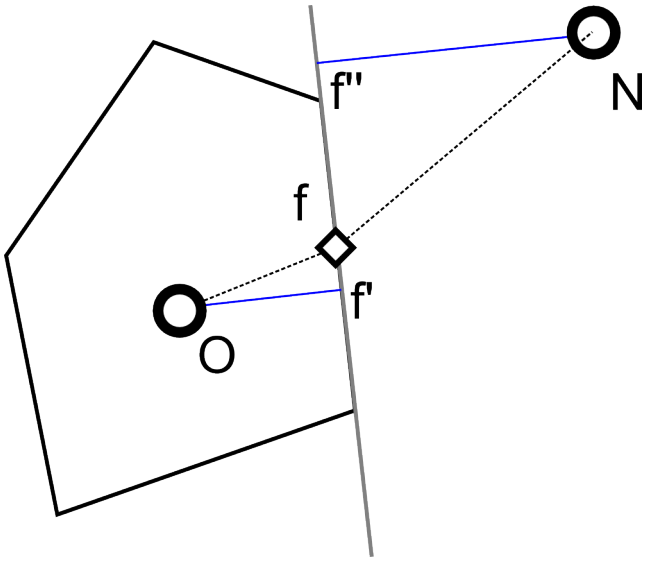
- for the neighbour:

  $g_N = 1 - g_O$

update_weights()

update linear interpolation weights

Calculation of the weights is done according to their normal distance from the face:



- for the neighbour cell

  $g_N = \frac{d_{Of}^{\vec{}} \cdot e_f^{\vec{}}}{d_{Of}^{\vec{}} \cdot e_f^{\vec{}} + d_{fC}^{\vec{}} \cdot e_f^{\vec{}}}$

- for the owner cell

  $g_O = 1 - g_N$

, where $\vec{e_f}$ is the face normal unit vector $\vec{e_f} = \frac{\vec{S_f}}{\|\vec{S_f}\|}$

vertices = None
list of vertices defining the face

weights = None
weights for linear interpolation [w_owner, w_neigbour]

pyCFD_mesh.face.**are_faces_equal**(*face1*, *face2*)
check if two face objects are the same

Parameters

- **face1** (pyCFD_mesh.face.Face (page 62)) – first face to compare
- **face2** (pyCFD_mesh.face.Face (page 62)) – second face to compare

Returns True or False

Return type bool

10.11.3 pyCFD_mesh.generic_mesh module

module for generic meshes

class pyCFD_mesh.generic_mesh.**GenericMesh**
basic class for meshes

cells = None
List of cells in the mesh

faces = None
List of independent faces

fields = None
List of fields

get_areas()
return areas of all faces within the mesh as vector

Returns array with face areas

Return type numpy.array

get_patch(*patch_name*)
return the patch with matching name

Parameters **patch_name** (*string*) – name of patch

Returns geometric patch object

Return type pyCFD_mesh.patch.Patch (page 67)

get_volumes()
return volumes of all cells within the mesh as vector

Returns array with cell volumes

Return type numpy.array

patchNames = None
List of patch names

patches = None
List of patches

plot_mesh_data()
plots basic mesh data

•number of vertices

- number of faces
  - number of boundary faces
  - number of internal faces
- number of cells
- number of patches
  - list of patches

vertices = None
List of all vertices

10.11.4 pyCFD_mesh.mesh module

class pyCFD_mesh.mesh.**Mesh**(*cellList=None*, *faceList=None*)
Bases: pyCFD_mesh.generic_mesh.GenericMesh (page 65)

class for meshes defined by existing cells or faces

Constructor

Parameters

- **cellList** (pyCFD_mesh.cell.Cell (page 61)) – default: None, list of vertices defining the face
- **faceList** (pyCFD_mesh.cell.Cell (page 61)) – default: None, list of vertices defining the face

10.11.5 pyCFD_mesh.mesh_object module

module for mesh objects (cells/faces)

class pyCFD_mesh.mesh_object.**MeshObject**
abstract class for mesh objects with father and vertices

father = None
reference to father object

vertices = None
list of vertices defining the object

pyCFD_mesh.mesh_object.**are_mesh_objects_equal**(*obj1*, *obj2*)
compare mesh cells or faces for equality

10.11.6 pyCFD_mesh.patch module

module for patches

class pyCFD_mesh.patch.**FieldPatch**(*field*, *geometric_patch*)
class for field patches

Constructor

The field patch is initialized with fixedValue type and with face values of 0.0.

Parameters

- **field** (pyCFD_fields.fields.VolumeField (page 50)) – reference to field object
- **geometric_patch** (Patch (page 67)) – reference to geometric patch

father = None
reference to father geometric patch object

field = None
reference to owner field

find_face_id_in_patch (*face_*)
return face id within the patch

Parameters **face** (pyCFD_mesh.face.face) – face object to find

Returns face id in patch

Return type int

set_patch_distributed (*boundary_values*, *boundary_type*)
set up patch with non-uniform boundary values

set_patch_uniform (*boundary_value*, *boundary_type*)
set up patch with uniform boundary values

type = None
patch type: 'fixedValue' or 'fixedGradient'

class pyCFD_mesh.patch.**Patch** (*face_list*, *patch_name*, *check_independent=False*)
class for geometric patches

Constructor

Parameters

- **face_list** (pyCFD_mesh.face.Face (page 62)) – list of faces in the patch
- **patch_name** (*string*) – name to use for the resulting patch
- **check_independent** (*bool*) – default: False, whether to check if faces are independent

faces = None
list of faces in the patch

ids = None
list of face ids in the patch

name = None
name of patch

10.11.7 pyCFD_mesh.readers module

module for reading existing meshes

class pyCFD_mesh.readers.**FoamMesh** (*dir=''*)
Bases: pyCFD_mesh.generic_mesh.GenericMesh (page 65)

A class for reading OpenFOAM meshes. The reader supports hex, tetra and wedge cells. Mesh data is read from _MESH/dir/cells.msh and _MESH/dir/boundary.msh.

Constructor

Parameters **dir** (*string*) – subdir within the _MESH directory

FLAG_verbose = None
true/false to plot extra info when constructing the mesh > for debugging

faces = None
list of face objects

vertices = None
list of vertex objects

class pyCFD_mesh.readers.**MSHMesh**
Bases: pyCFD_mesh.generic_mesh.GenericMesh (page 65)

A class for reading gmsh meshes. The reader supports hex, tetra and wedge cells. Mesh data is read from _MESH/cells.msh and _MESH/boundary.msh.

FLAG_verbose = None
true/false to plot extra info when constructing the mesh > for debugging

cells = None
list of cell objects

faces = None
list of independent face objects

get_boundary_info()
read boundary information from _MESH/bounday.msh file

meshFile = None
Name of mesh file loaded.

patchNames = None
list of patch names

patches = None
list of patch objects

vertices = None
list of vertex objects

10.11.8 pyCFD_mesh.sub_mesh module

class pyCFD_mesh.sub_mesh.**SubMesh**(*cellList=None*, *faceList=None*)
Bases: pyCFD_mesh.generic_mesh.GenericMesh (page 65)

Class for submeshes with a given list of cells and faces (e.g only the internal faces of a mesh)

constructor for the SubMesh class

10.11.9 pyCFD_mesh.vertex module

class pyCFD_mesh.vertex.**Vertex**(*X=0.0*, *Y=0.0*, *Z=0.0*)
class for 3D vertices

Constructor

Parameters

- **X** (*float*) – default: 0.0, x coordinate of the vertex
- **Y** (*float*) – default: 0.0, y coordinate of the vertex
- **Z** (*float*) – default: 0.0, z coordinate of the vertex

X = None
vertex X coordinate

Y = None
vertex Y coordinate

Z = None
vertex Z coordinate

cells = None
reference to connected cells

coords = None
vector of vertex coordinates

faces = None
reference to connected faces

father = None
reference to father object

get_cell_ids()

get_coords()
return coordinates as numpy array

id = None
vertex id

print_coordinates()

setX(*newX*)
setter for X coordinate of vertex

setY(*newY*)
setter for Y coordinate of vertex

setZ(*newZ*)
setter for Z coordinate of vertex

pyCFD_mesh.vertex.**are_vertices_equal**(*vertex1*, *vertex2*)

pyCFD_mesh.vertex.**get_independent_vertices**(*vertex_list*)
return the list of independent vertices

Parameters **vertex_list** (*dict*) – list of vertices

Returns list of independent vertices

Return type dict

pyCFD_mesh.vertex.**get_list_of_ids**(*vertex_list*)
return the id / list of ids for a vertex / list of vertices

10.12 pyCFD_monitors package

10.12.1 pyCFD_monitors.monitors module

module for calculating monitoring quantities

pyCFD_monitors.monitors.**CFL**(*velocity_field*, *length_scale*, *dt*)

pyCFD_monitors.monitors.**CFLMag**(*velocity_field*, *length_scale*, *dt*)

pyCFD_monitors.monitors.**cd**(*scalar_field*, *referece_velocity*, *rho*, *patch_name*, *flow_direction*, *mesh_width=1.0*)
calculate drag coefficient on a 2D mesh:

$c_d = \frac{\sum_f \phi_f \left(\vec{A} \cdot e_{f\vec{low}}\right)}{0.5 \rho v_{ref}^2}$

Parameters

- **scalar_field** (pyCFD_fields.fields.ScalarField (page 49)) – field to monitor: ϕ
- **reference_velocity** (*float*) – reference velocity: v_{ref}
- **rho** (*float*) – reference density: ρ
- **patch_name** (*string*) – patch on which integration should be performed

- **flow_direction** (*numpy.array*) – array of reference flow direction: $e_{\vec{flow}}$
- **mesh_width** (*float*) – default: 1.0, mesh width normal to the flow

pyCFD_monitors.monitors.**globalMass**(*massflux_field*)

10.13 pyCFD_operators package

10.13.1 pyCFD_operators.explicit_operators module

module for explicit operators

class pyCFD_operators.explicit_operators.**Divergence**(*volume_field*, *massflux_field*, *type_*)

Bases: pyCFD_operators.generic_operator.GenericScalarOperator (page 76)

Explicit divergence operator for pyCFD_fields.fields.ScalarField (page 49) using Picard iteration with known mass fluxes. Divergence is calculated using the Gauss theorem:

$$\int_V \nabla\left(\rho\vec{v}\phi\right) dV = \oint_A \left(\rho\vec{v}\phi\right)\cdot d\vec{A} = \sum_f \dot{m}_f * \phi_f$$

The explicit divergence operator returns the explicit contributions of the divergence of a scalar field to the right hand side of the linear equation system of the scalar field:

- $A_{ij} = 0$
- $A_{ii} = 0$
- $b_i = \sum_f \dot{m}_f * \phi_f$
- i,j = 1 ... # of cells

Note: $\dot{m}_f$ is a pyCFD_fields.fields.SurfaceScalarField (page 49) value multiplied with 1 if divergence is calculated for the owner cell and -1 for the neighbour cell.

Note: ϕ_f is a pyCFD_fields.fields.SurfaceScalarField (page 49) calculated with an interpolation scheme

---

Note: boundary face values should be updated before the operator is applied

---

Constructor

Parameters

- **volume_field** (*pyCFD_fields.fields.VolumeField or None*) – volume field to calculate the divergence for. If None than mass divergence is calculated.
- **massflux_field** (*pyCFD_fields.fields.SurfaceScalarField* (page 49)) – surface field with massflux values
- **type** (*string*) – type of scheme to calculate the face values with. Available types are in pyCFD_fields.calculated_fields.HRSFaceValue (page 46) and pyCFD_fields.calculated_fields.UpwindFaceValue (page 48).

class pyCFD_operators.explicit_operators.**DivergenceVec**(*volume_field*, *massflux_field*, *type_*)

Bases: pyCFD_operators.generic_operator.GenericVectorOperator (page 76)

Explicit divergence operator for pyCFD_fields.fields.VectorField (page 50) using Picard iteration with known mass fluxes. The operator calls

pyCFD_operators.explicit_operators.Divergence (page 70) for all the components of the field.

Constructor

Parameters

- **volume_field** (*pyCFD_fields.fields.VolumeField* (page 50)) – volume field to calculate the divergence for
- **massflux_field** (*pyCFD_fields.fields.SurfaceScalarField* (page 49)) – surface field with massflux values
- **type** (*string*) – type of scheme to calculate the face values with. Available types are in pyCFD_fields.calculated_fields.HRSFaceValue .

class pyCFD_operators.explicit_operators.**Gradient**(*volume_scalar_field*, *multipl_field=None*)

Bases: pyCFD_operators.generic_operator.GenericVectorOperator (page 76)

Explicit gradient operator for pyCFD_fields.fields.VectorField (page 50).

$$\int_V \Gamma * \nabla\phi dV = \Gamma * \nabla\phi * \Delta V$$

The explicit gradient operator returns the explicit contributions of the gradient of a scalar field to the right hand sides of the linear equation systems of a vector field:

- $A_{ij} = 0$
- $A_{ii} = 0$
- $b_{i,x} = \Gamma_i * (\nabla\phi)_i \cdot \vec{i} * \Delta V_i$
- $b_{i,y} = \Gamma_i * (\nabla\phi)_i \cdot \vec{j} * \Delta V_i$
- $b_{i,z} = \Gamma_i * (\nabla\phi)_i \cdot \vec{k} * \Delta V_i$
- i,j = 1 ... # of cells

Note: $\nabla\phi$ is calculated in pyCFD_fields.calculated_fields.GaussCellGradient (page 45)

Constructor

Parameters

- **volume_scalar_field** (*pyCFD_fields.fields.ScalarField* (page 49)) – volume field to calculate the gradient for
- **multipl_field** (*pyCFD_fields.fields.ScalarField* (page 49)) – default: None, field to multiply the gradient with (e.g. $\frac{1}{\rho}$)

class pyCFD_operators.explicit_operators.**Laplace**(*volume_field*, *gamma_=1.0*)

Bases: pyCFD_operators.generic_operator.GenericScalarOperator (page 76)

Explicit laplace operator for pyCFD_fields.fields.ScalarField (page 49).

$$\begin{aligned}\int_V \nabla\left(\Gamma\nabla\phi\right) dV &= \oint_A \left(\Gamma\nabla\phi\right)_f \cdot d\vec{A} \\ &= \sum_f \left(\Gamma\nabla\phi\right)_f \cdot \vec{A}\end{aligned}$$

The explicit laplace operator returns the explicit contributions of the laplace of a scalar field to the right hand side of the linear equation system of the scalar field:

- $A_{ij} = 0$
- $A_{ii} = 0$

•$b_i = \sum_f (\Gamma \nabla \phi)_f \cdot \vec{A}$

•i,j = 1 ... # of cells

Note: Gradient at fixed value boundaries is calculated as:

$\nabla \phi = \frac{\phi_b - \phi_O}{d_{Ob}}$

Note: Gradient at fixed gradient boundaries directly applied as:

$\nabla \phi = \nabla \phi_b$

Constructor

Parameters

- **volume_field** (*pyCFD_fields.fields.VolumeField* (page 50)) – volume field to calculate the divergence for
- **gamma** (*Float*) – default: 1.0, constant conductivity

class pyCFD_operators.explicit_operators.**LaplaceVec**(*volume_field*, *gamma_=1.0*)

Bases: pyCFD_operators.generic_operator.GenericVectorOperator (page 76)

Explicit laplace operator for pyCFD_fields.fields.VectorField (page 50). The operator calls pyCFD_operators.explicit_operators.Laplace (page 71) for all the components of the field.

Constructor

Parameters

- **volume_field** (*pyCFD_fields.fields.VolumeField* (page 50)) – volume field to calculate the divergence for
- **gamma** (*Float*) – constant conductivity

10.13.2 pyCFD_operators.generic_equation module

module for generic scalar and vector equations

class pyCFD_operators.generic_equation.**GenericScalarEquation**(*mesh_*, *volume_field*, *solver_type*, *under_relax=1.0*)

Bases: pyCFD_operators.generic_operator.GenericScalarOperator (page 76)

basic class for scalar equations

Available solver types:

•conjugate gradient: numpy.linalg.cg - python built-in

•biconjugate gradient: numpy.linalg.bicg - python built-in

•Gauss-Seidel: pyCFD_linear_solvers.linear_solvers.gs (page 60) - own implementation

•LU solver: pyCFD_linear_solvers.linear_solvers.lu_solver_plu (page 61) - own implementation

Constructor

Parameters

- **mesh** (pyCFD_mesh.generic_mesh.GenericMesh (page 65)) – mesh object
- **volume_field** (pyCFD_fields.fields.ScalarField (page 49)) – variable scalar field

- **solver_type** (*string*) – solver type to be used
- **under_relax** (*float*) – default: 1.0, explicit underrelaxation factor

add_monitor (*name_*)
add a new monitoring quantity to the residuals

Parameters **name** (*string*) – name of new monitoring value

append_current_residual (*time*, *name=''*)
append current residuals and monitoring values to the CSV file

saved file is residuals_<field_name>.csv

append_to_monitor (*value_*, *name_*)
append new value to an existing monitoring quantity

Param value_: new value

Parameters **name** (*string*) – name of existing monitoring quantity

diag ()
return matrix main diagonal as vector

field = None
reference to field

fix_cell_value (*cell_index*)
modify the equations coefficient matrix to fix the value in a cell. Coefficient in the main diagonal is set to 1.0 and other coeffiecient in the cells row are set to 0.0.

Parameters **cell_index** (*int*) – index of the cell where value should be fixed

l = None
lower triangluation matrix for lu solver

load_plu (*dir_name=''*)
save p, l and u matrices to pyCFD_config.config (page 44).__FIELDDIR__

monitor_names = None
list of monitor names

monitors = None
list of monitor values

p = None
permutation matrix for lu solver

relax ()
Relax the solution of solve().

reset ()
Reset coefficient matrix and RHS, update boundary face values and update x_old with current field.

residuals = None
list of residual values

save_plu ()
save p, l and u matrices to pyCFD_config.config (page 44).__FIELDDIR__

save_residual (*times*, *name=''*)
save all stored residuals and other monitoring quantities to pyCFD_config.config (page 44).__OUTDIR__ at the end of the calculation as CSV file

saved file is residuals_<field_name>.csv

save_residual_header (*name=''*)
save only headers of residuals and monitoring values to pyCFD_config.config (page 44).__OUTDIR__ as CSV file

saved file is residuals_<field_name>.csv

solve (*x_0=None*, *tol=1e-05*, *max_iter=100*, *exchange_zero=1e-16*)
Solve the equation *A * x = b* using an linear equation solver. After solving old unknown vectors and volume field values are overwritten.

solver = None
string with iterative solver type

times = None
list of time steps

u = None
upper triangluation matrix for lu solver

under_relax = None
under relaxation for the solution

write_A (*name_=''*)
save the equations coefficient matrix to pyCFD_config.config (page 44).__OUTDIR__ as DAT file.

write_b (*name_=''*)
save the equations right hand side to pyCFD_config.config (page 44).__OUTDIR__ as DAT file.

x = None
vector of unknowns

x_old = None
field vector from previsus iteration/time step

x_old_old = None
field vector from previsus previous iteration/time step

class pyCFD_operators.generic_equation.**GenericVectorEquation** (*mesh_*, *volume_field*, *solver_type*, *under_relax=1.0*)
Bases: pyCFD_operators.generic_operator.GenericVectorOperator (page 76)

basic class for vector equations

Available solver types:

- conjugate gradient: numpy.linalg.cg - python built-in
- biconjugate gradient: numpy.linalg.bicg - python built-in
- Gauss-Seidel: pyCFD_linear_solvers.linear_solvers.gs (page 60) - own implementation
- LU solver: pyCFD_linear_solvers.linear_solvers.lu_solver_plu (page 61) - own implementation

Constructor

Parameters

- **mesh** (pyCFD_mesh.generic_mesh.GenericMesh (page 65)) – mesh object
- **volume_field** (pyCFD_fields.fields.ScalarField (page 49)) – variable scalar field
- **solver_type** (*string*) – solver type to be used
- **under_relax** (*float*) – default: 1.0, explicit underrelaxation factor

append_current_residual (*time*, *name=''*)
append current residuals and monitoring values to the CSV file

saved file is residuals_<field_name>.csv

diag()
return matrix main diagonal as vector

field = None
reference to field

reset()
Reset coefficient matrix and RHS, update boundary face values and update x_old with current field.

residualsX = None
list of residual values for x component

residualsY = None
list of residual values for x component

residualsZ = None
list of residual values for x component

save_residual(*times*, *name=''*)
save all stored residuals and other monitoring quantities to pyCFD_config.config (page 44).__OUTDIR__ at the end of the calculation as CSV file

saved file is residuals_<field_name>.csv

save_residual_header(*name=''*)
save only headers of residuals and monitoring values to pyCFD_config.config (page 44).__OUTDIR__ as CSV file

saved file is residuals_<field_name>.csv

solve(*x_0=None*, *tol=1e-05*, *max_iter=100*, *exchange_zero=1e-16*)
Solve the equation *A * x = b* using an iterative solver. After solving old unknown vectors and volume field values are overwritten.

solver = None
string with iterative solver type - "cg"/"bicg"

times = None
list of time steps

under_relax = None
under relaxation for the solution

write_A(*name_=''*)
save the equations coefficient matrix to pyCFD_config.config (page 44).__OUTDIR__ as DAT file.

write_b(*name_=''*)
save the component equations right hand sides to pyCFD_config.config (page 44).__OUTDIR__ as DAT files.

xX = None
vector of unknowns of x component

xX_old = None
field vector of x component from previsus iteration/time step

xX_old_old = None
field vector of x component from previsus previous iteration/time step

xY = None
vector of unknowns of y component

xY_old = None
field vector of y component from previsus iteration/time step

xY_old_old = None
field vector of y component from previsus previous iteration/time step

xZ = None
vector of unknowns of z component

xZ_old = None
field vector of z component from previsus iteration/time step

xZ_old_old = None
field vector of z component from previsus previous iteration/time step

10.13.3 pyCFD_operators.generic_operator module

generic module for scalar and vector operators

class pyCFD_operators.generic_operator.**GenericScalarOperator**(*mesh_*)
basic class for scalar operators

b = None
vector of right hand side values

father = None
reference to mesh

fix_cell_value(*cell_index*)
modify the equations coefficient matrix to fix the value in a cell. Coefficient in the main diagonal is set to 1.0 and other coeffiecient in the cells row are set to 0.0.

Parameters **cell_index** (*int*) – index of the cell where value should be fixed

class pyCFD_operators.generic_operator.**GenericVectorOperator**(*mesh_*)
basic class for vector operators

bX = None
vector of right hand side values for the x component

bY = None
vector of right hand side values for the y component

bZ = None
vector of right hand side values for the z component

father = None
reference to mesh

10.13.4 pyCFD_operators.implicit_operators module

generic module for implicit operators

class pyCFD_operators.implicit_operators.**DdtEuler**(*equation*, *dt*)
Bases: pyCFD_operators.generic_operator.GenericScalarOperator (page 76)

Euler transient operator for pyCFD_fields.fields.ScalarField (page 49):

$$\int_V \frac{\partial \phi}{\partial t} dV = \frac{\phi^{n+1} - \phi^n}{\Delta t} \Delta V$$

The operator returns the followng contributions:

- $A_{ij} = 0$
- $A_{ii} = \frac{\Delta V_i}{\Delta T}$
- $b_i = \phi_i^n \frac{\Delta V_i}{\Delta T}$
- i,j = 1 ... # of cells

Constructor

Parameters

- **equation** (pyCFD_operators.generic_equation.GenericScalarEquation (page 72)) – field equation of a pyCFD_fields.fields.ScalarField (page 49) to calculate the time derivative for
- **dt** (*float*) – time step

class pyCFD_operators.implicit_operators.**DdtEulerVec**(*equation*, *dt*)

Bases: pyCFD_operators.generic_operator.GenericVectorOperator (page 76)

Euler transient operator for pyCFD_fields.fields.VectorField (page 50).The operator calls pyCFD_operators.implicit_operators.DdtEuler (page 76) for all the components of the field.

Constructor

Parameters

- **volume_field** (pyCFD_operators.generic_equation.GenericVectorEquation (page 74)) – field equation of a pyCFD_fields.fields.VectorField (page 50) to calculate the time derivative for
- **dt** (*float*) – time step

class pyCFD_operators.implicit_operators.**Divergence**(*volume_field*, *massflux_field*, *scheme_*)

Bases: pyCFD_operators.generic_operator.GenericScalarOperator (page 76)

Implicit divergence operator for pyCFD_fields.fields.ScalarField (page 49) using Picard iteration with known mass fluxes. Divergence is calculated using the Gauss theorem:

$$\int_V \nabla\left(\rho\vec{v}\phi\right) dV = \oint_A \left(\rho\vec{v}\phi\right)\cdot\vec{A} = \sum_f \dot{m}_f * \phi_f$$

The operator creates the coefficient matrix of divergence using the UDS scheme for calculating the face value ϕ_f. High resolution schemes are applied as deferred correction :

$$\sum_f \dot{m}_f * \phi_f^{HRS,new} = \sum_f \dot{m}_f * \phi_f^{UDS,new} + \sum_f \dot{m}_f * \left(\phi_f^{HRS,old} - \phi_f^{UDS,old}\right)$$

The resulting matrix coefficients and right hand side:

- $A_{ij} = -max\left(-\dot{m}_{f,ij}, 0\right)$
- $A_{ii} = \sum_j max\left(\dot{m}_{f,ij}, 0\right)$
- $b_i = -\sum_j \dot{m}_{f,ij}\left(\phi_f^{HRS,old} - \phi_f^{UDS,old}\right)$
- i,j = 1 ... # of cells

, where f is the face between cell i and cell j.

Constructor

Parameters

- **volume_field** (pyCFD_fields.fields.ScalarField (page 49)) – volume field to calculate the divergence for
- **massflux_field** (pyCFD_fields.fields.SurfaceScalarField (page 49)) – surface field with massflux values
- **type** (*string*) – type of scheme to calculate the face values with. Available types are in pyCFD_fields.calculated_fields.HRSFaceValue (page 46) and pyCFD_fields.calculated_fields.UpwindFaceValue (page 48).

class pyCFD_operators.implicit_operators.**DivergenceVec**(*volume_field*, *massflux_field*, *scheme_*)

Bases: pyCFD_operators.generic_operator.GenericVectorOperator (page 76)

Implicit divergence operator for pyCFD_fields.fields.VectorField (page 50) using Picard iteration with known mass fluxes. Divergence is calculated using the Gauss theorem. The operator calls pyCFD_operators.implicit_operators.Divergence (page 77) for all the components of the field.

Constructor

Parameters

- **volume_field** (pyCFD_fields.fields.VectorField (page 50)) – volume field to calculate the divergence for
- **massflux_field** (pyCFD_fields.fields.SurfaceScalarField (page 49)) – surface field with massflux values
- **type** (*string*) – type of scheme to calculate the face values with. Available types are in pyCFD_fields.calculated_fields.HRSFaceValue (page 46) and pyCFD_fields.calculated_fields.UpwindFaceValue (page 48).

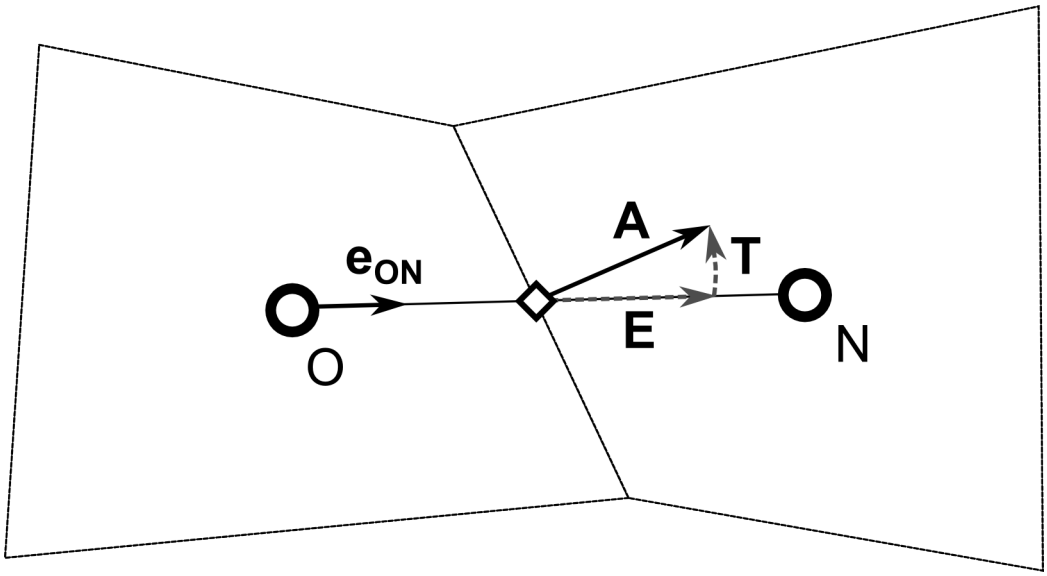
class pyCFD_operators.implicit_operators.**Laplace**(*volume_field*, *gamma_=1.0*, *non_ortho_corr=''*, *calc_matrix=True*)

Bases: pyCFD_operators.generic_operator.GenericScalarOperator (page 76)

Implicit laplace operator for pyCFD_fields.fields.ScalarField (page 49).

$$\int_V \nabla \left(\Gamma \nabla \phi\right) dV = \oint_A \left(\Gamma \nabla \phi\right) \cdot d\vec{A}$$
$$= \sum_f \left(\Gamma \nabla \phi\right) \cdot \vec{A}$$

In an arbitrary mesh accuracy of the gradient is calculation is affected by mesh non-orthogonality:



In the mesh above the vector $d_{\vec{ON}}$ or its unit vector $e_{\vec{ON}}$ is not parallel to the face normal vector $\vec{A}$. The face normal vector can be written as a sum of $\vec{E}$ (parallel to $e_{\vec{ON}}$) and the difference $\vec{T}$.

Applying a non-orthogonal correction for the face normal gradients it can be rewritten as a sum of orthogonal and non-orthogonal contribution:

$$\nabla\phi \cdot \vec{A} = \nabla\phi \cdot \vec{E} + \nabla\phi \cdot \vec{T}$$
$$= E\frac{\phi_N - \phi_O}{d_{ON}} + \nabla\phi \cdot \vec{T}$$

The orthogonal part is treated explicitly, while the non-orthogonal part with a deferred correction. This treatment results in the following coefficient matrix and right hand side:

- $A_{ij} = \Gamma_{f,ij} \frac{E_{f,ij}}{d_{ON,ij}}$
- $A_{ii} = -\sum_j A_{ij}$
- $b_i = -\sum_f \left((\nabla \phi)_f^{old} \cdot \vec{T}_f \right)$
- i,j = 1 ... # of cells

, where f is the face between cell i and cell j.

Note: the term $\nabla (\phi)_f^{old}$ is calculated from the known phi field using pyCFD_fields.calculated_fields.GaussFaceGradient (page 46).

Note: 3 non orthogonal correction types are implemented: "minimal correction" , "orthogonal correction" and "over relaxed correction" from which the "overrelaxed correction" is the one that is tested.

Note: At fixed value boundaries the coefficient matrix and right hand side change the following way:

- $A_{ij} = 0$
- $b_i - = \Gamma_{f,ij} \frac{E_{f,ij}}{d_{ON,ij}} * \phi_b$

Note: At fixed gradient boundaries the coefficient matrix and right hand side change the following way:

- $A_{ij} = 0$
- $b_i - = \Gamma_{f,ij} \nabla \phi_b$

Constructor

Parameters

- **volume_field** (pyCFD_fields.fields.ScalarField (page 49)) – volume field to calculate the divergence for
- **gamma** (*Float*) – default: 1.0, constant conductivity
- **non_ortho_corr** (*String*) – non orthogonal correction type
- **calc_matrix** (*bool*) – default: Ture, switch to decide if coefficient matrix should be calculated again

class pyCFD_operators.implicit_operators.**LaplaceVec**(*volume_field*, *gamma_=1.0*, *non_ortho_corr=''*, *calc_matrix=True*)

Bases: pyCFD_operators.generic_operator.GenericVectorOperator (page 76)

Implicit laplace operator for pyCFD_fields.fields.VectorField (page 50). The operator calls pyCFD_operators.implicit_operators.Laplace (page 78) for all the components of the field.

constructor for Divergence operator

Parameters

- **volume_field** (*pyCFD_fields.fields.VolumeField* (page 50)) – volume field to calculate the divergence for
- **gamma** (*String*) – constant conductivity
- **gamnon_ortho_corr** – non orthogonal correction type

10.13.5 pyCFD_operators.vector_operations module

module for fast vector operations

pyCFD_operators.vector_operations.**VecCross**(*left*, *right*)
array cross product re-implementation for vectors

pyCFD_operators.vector_operations.**VecDot**(*left*, *right*)
array dot product re-implementation for vectors

10.14 pyCFD_output package

10.14.1 pyCFD_output.output module

module of functions for writing data files

pyCFD_output.output.**clean_output_dirs**()
delete files from pyCFD_config.config (page 44).__OUTDIR__ and pyCFD_config.config (page 44).__OUTITERDIR__

pyCFD_output.output.**write_csv_file**(*headers*, *vectors*, *file_name_with_path*)

pyCFD_output.output.**write_mesh_faces**(*mesh_*)
write .vtu file with all the faces in the pyCFD_config.config (page 44).__OUTITERDIR__ directory

Parameters **mesh** (pyCFD_mesh.generic_emsh.GenericMesh) – a mesh object

pyCFD_output.output.**write_mesh_faces_with_field**(*surface_field_list*, *name_=''*)
write .vtu file with all the faces and a list of surface field in the pyCFD_config.config (page 44).__OUTITERDIR__ directory

Parameters **mesh** (pyCFD_mesh.generic_emsh.GenericMesh) – a mesh object

pyCFD_output.output.**write_mesh_file**(*Mesh*)
write .vtu file containing all the cells in the pyCFD_config.config (page 44).__OUTITERDIR__.

Parameters **mesh** (pyCFD_mesh.generic_emsh.GenericMesh) – mesh object

pyCFD_output.output.**write_mesh_file_with_fields**(*field_list*, *name_=''*)
write .vtu file containing all the cells and field values in the pyCFD_config.config (page 44).__OUTITERDIR__ directory

Parameters

- **fieldList** (pyCFD_fields.fields.VolumeField (page 50)) – a list of fields to include in the output. *father* of fields defines the mesh
- **name** (*string*) – filename to use

pyCFD_output.output.**write_mesh_internal_faces**(*mesh_*)
write .vtu file with all the internal faces in the pyCFD_config.config (page 44).__OUTITERDIR__ directory

Parameters **mesh** (pyCFD_mesh.generic_emsh.GenericMesh) – a mesh object

pyCFD_output.output.**write_mesh_internal_faces_with_field**(*surface_field_list*)
write .vtu file with all the internal faces in the pyCFD_config.config (page 44).__OUTITERDIR__ directory

Parameters **mesh** (pyCFD_mesh.generic_emsh.GenericMesh) – a mesh object

pyCFD_output.output.**write_patch_file_with_fields**(*mesh_*, *patch_name*, *volume_field_list*)
write a patch mesh with field values into separate a .vtu file in the pyCFD_config.config (page 44).__OUTITERDIR__ directory

Parameters

- **mesh** (pyCFD_mesh.generic_emsh.GenericMesh) – a mesh object
- **patch_name** (*string*) – name of geometric patch
- **volume_field_list** (pyCFD_fields.fields.VolumeField (page 50)) – list of volume fields

pyCFD_output.output.**write_patch_files**(*mesh_*)
write separate .vtu files containing patch faces for all the patches in the pyCFD_config.config (page 44).__OUTITERDIR__

Parameters **mesh** (pyCFD_mesh.generic_emsh.GenericMesh) – mesh object

pyCFD_output.output.**write_patch_files_with_fields**(*mesh_*, *volume_field_list*)
write patch meshes with field values into separate .vtu files in the pyCFD_config.config (page 44).__OUTITERDIR__ directory

Parameters

- **mesh** (pyCFD_mesh.generic_emsh.GenericMesh) – a mesh object
- **volume_field_list** (pyCFD_fields.fields.VolumeField (page 50)) – list of volume fields

pyCFD_output.output.**write_pvd_collection**(*times_*)
write a .pvd file with links to all the .vtu files including time information in the pyCFD_config.config (page 44).__OUTDIR__ directory

Parameters **times** (*float*) – array of timesteps to include

pyCFD_output.output.**write_row_to_csv_file**(*data_row*, *file_name_with_path*, *write_or_append='w'*)

pyCFD_output.output.**write_standalone_pvd_collection**()
write a .pvd file with links to all the .vtu files without time information in the pyCFD_config.config (page 44).__OUTDIR__ directory. All .vtu files in the pyCFD_config.config (page 44).__OUTITERDIR__ directory will be included.

10.14.2 pyCFD_output.output_other module

pyCFD_output.output_other.**clean_output_dirs**()

pyCFD_output.output_other.**write_csv_file**(*headers*, *vectors*, *file_name_with_path*)

pyCFD_output.output_other.**write_mesh_faces**(*mesh_*)
write .vtu file with all the faces (internal + boundary)

Parameters **mesh** (*pyCFD_mesh.generic_mesh.GenericMesh* (page 65)) – a mesh object (GenericMesh or derived)

pyCFD_output.output_other.**write_mesh_faces_with_field**(*surface_field_list*, *name_=''*)
write .vtu file with all the faces

Parameters **mesh** (*pyCFD_mesh.generic_mesh.GenericMesh* (page 65)) – a mesh object (GenericMesh or derived)

pyCFD_output.output_other.**write_mesh_file**(*Mesh*)
write .vtu file containing all the cells in the _OUTPUT directory

Parameters **mesh** (*pyCFD_mesh.generic_emsh.GenericMesh*) – mesh object

pyCFD_output.output_other.**write_mesh_file_with_fields**(*field_list*, *name_=''*)
write .vtu file containing all the cells and field values in the _OUTPUT directory

Parameters

- **fieldList** (*pyCFD_fields.fields.VolumeField* (page 50)) – a list of fields to include in the output. *father* of fields defines the mesh
- **name** (*string*) – filename to use

pyCFD_output.output_other.**write_mesh_internal_faces**(*mesh_*)
write .vtu file with all the internal faces

Parameters **mesh** (*pyCFD_mesh.generic_mesh.GenericMesh* (page 65)) – a mesh object (GenericMesh or derived)

pyCFD_output.output_other.**write_mesh_internal_faces_with_field**(*surface_field_list*)
write .vtu file with all the internal faces

Parameters **mesh** (*pyCFD_mesh.generic_mesh.GenericMesh* (page 65)) – a mesh object (GenericMesh or derived)

pyCFD_output.output_other.**write_patch_files**(*mesh_*)
write separate .vtu files containing patch faces for all the patches in the _OUTPUT directory

Parameters **mesh** (*pyCFD_mesh.generic_emsh.GenericMesh*) – mesh object

pyCFD_output.output_other.**write_patch_files_with_fields**(*mesh_*, *volume_field_list*)
write patch meshes with field values into separate .vtu files

Parameters

- **mesh** (*pyCFD_mesh.generic_mesh.GenericMesh* (page 65)) – a mesh object (GenericMesh or derived)
- **volume_field_list** (*pyCFD_fields.fields.VolumeField* (page 50)) – list of volume fields

Output separate .vtu files containing patch faces for all the patches in the _OUTPUT directory

pyCFD_output.output_other.**write_pvd_collection**(*times_*)

pyCFD_output.output_other.**write_row_to_csv_file**(*data_row*, *file_name_with_path*, *write_or_append='w'*)

pyCFD_output.output_other.**write_standalone_pvd_collection**()

10.15 pyCFD_test_functions package

10.15.1 pyCFD_test_functions.testFunctions module

module of functions for testing mesh data

pyCFD_test_functions.testFunctions.**print_mesh_data**(*meshObject*)
print minimal info about mesh

pyCFD_test_functions.testFunctions.**write_element_face_vector_files**(*MSHMeshObject*)

pyCFD_test_functions.testFunctions.**write_face_files**(*MSHMeshObject*)

pyCFD_test_functions.testFunctions.**write_internal_face_ffToF_vector_files**(*mesh_*)

pyCFD_test_functions.testFunctions.**write_owner_to_neighbour_vectors**(*mesh_*)

pyCFD_test_functions.testFunctions.**write_patch_faces**(*MSHMeshObject*)

p