pyCALC-RANS: A Python Code for Two-Dimensional Turbulent Steady Flow

Lars Davidson
Div. of Fluid Dynamics
Dept. of Mechanics and Maritime Sciences
Chalmers University of Technology
SE-412 96 Göteborg, Sweden

July 10, 2023

Abstract

This report gives some details on **pyCALC-RANS** and how to use it. It is written in Python (3.8). The code solves the two-dimensional, steady. incompressible momentum equations, the continuity equation and the $k-\omega$ turbulence model. The density is assumed to be constant and equal to one, i.e. $\rho \equiv 1$. The grid may be curvi-linear.

pyCALC-RANS is a finite volume code. It is fully vectorized (i.e. no `for` loops). The solution procedure is based on the pressure-correction method (SIMPLEC). Two metnods for dicretizing the convection terms are available, second-order central differencing and a hybrid scheme of first-order upwind and second-order central differencing. The discretized equations are solved with Pythons sparse matrix solvers (currently `linalg.lgmres` or `linalg.gmres` are used).

pyCALC-RANS was written starting from the 3D, unsteady LES/DES code **pyCALC-LES** [1].

Contents

1 Geometrical details of the grid **5**
1.1 Grid . . . 5
1.1.1 Nomenclature for the grid . . . 5
1.1.2 Area calculation of control volume faces . . . 5
1.1.3 Interpolation . . . 7
1.2 Gradient . . . 7

2 Diffusion **7**
2.1 Convergence criteria . . . 8
2.2 2D Diffusion . . . 9

3 Convection – diffusion **11**
3.1 Central Differencing scheme (CDS) . . . 11
3.2 First-order upwind scheme . . . 12
3.3 Hybrid scheme . . . 13
3.4 Inlet boundary conditions using source term . . . 13
3.5 Wall boundary conditions using source term . . . 14
3.6 Pressure correction equation . . . 14

4 Boundary Conditions **15**
4.1 Outlet velocity, small outlet . . . 15
4.2 Outlet velocity, large outlet . . . 15
4.3 Remaining variables . . . 16

5 The $k - \omega$ model **16**

6 Modules **17**
6.1 bc_outlet_bc . . . 17
6.2 calck . . . 17
6.3 calcom . . . 17
6.4 calcp . . . 17
6.5 calcu . . . 17
6.6 calcv . . . 17
6.7 coeff . . . 17
6.8 compute_face_phi . . . 17
6.9 conv . . . 17
6.10 correct_u_v_p . . . 18
6.11 fix_omega . . . 18
6.12 dphidx, dphidy . . . 18
6.13 init . . . 18
6.14 modify_k, modify_om, modify_u, modify_v . . . 18
6.15 modify_case.py . . . 18
6.16 modify_init . . . 18
6.17 print_indata . . . 18
6.18 read_restart_data . . . 18
6.19 save_data . . . 19
6.20 save.file . . . 19
6.21 save_vtk . . . 19

6.22 setup_case.py 19
6.23 solve_2d 19
6.24 vist_kom 19

7 Lid-driven cavity at $Re = 1000$ 19
7.1 setup_case.py 21
7.1.1 Section 1 21
7.1.2 Section 3 21
7.1.3 Section 4 21
7.1.4 Section 6 21
7.1.5 Section 7 22
7.1.6 Section 8 22
7.1.7 Section 9 22
7.1.8 Section 10 22
7.2 modify_case.py 23
7.2.1 modify_u 23
7.3 Run the code 23

8 Fully-developed channel flow at $Re_\tau = 5\,200$ 24
8.1 setup_case.py 24
8.1.1 Section 1 24
8.1.2 Section 2 24
8.1.3 Section 3 24
8.1.4 Section 4 25
8.1.5 Section 8 25
8.1.6 Section 9 25
8.1.7 Section 10 25
8.2 modify_case.py 26

9 Channel flow (inlet outlet) at $Re_\tau = 5\,200$ 27
9.1 setup_case.py 27
9.1.1 Section 10 27
9.2 modify_case.py 27
9.2.1 modify_init 27
9.2.2 modify_inlet 28
9.2.3 modify_u 28
9.2.4 modify_v 28
9.2.5 modify_k 29
9.2.6 modify_om 29
9.2.7 modify_outlet 29
9.2.8 fix_omega 29

10 RANS of boundary layer flow using $k - \omega$ 30
10.1 setup_case.py 30
10.1.1 Section 1 30
10.1.2 Section 2 30
10.1.3 Section 4 30
10.1.4 Section 5 30
10.1.5 Section 6 31
10.1.6 Section 9 31

10.1.7 Section 10 31
10.2 modify_case.py 32
10.2.1 modify_init 32

11 Channel with a corrugation **33**
11.1 setup_case.py 33
11.1.1 Section 1 33
11.1.2 Section 2 33
11.1.3 Section 4 34
11.1.4 Section 5 34
11.1.5 Section 6 34
11.1.6 Section 9 34
11.1.7 Section 10 34
11.1.8 modify_init 35
11.1.9 modify_init 35
11.1.10 Comparison with LES data 36

A Variables in pyCALC-RANS **36**

B Sparse matrix format in Python **39**
B.1 2D grid, $ni \times nj = (3, 4)$ 39
B.2 2D grid, $ni \times nj = (3, 2)$ 41

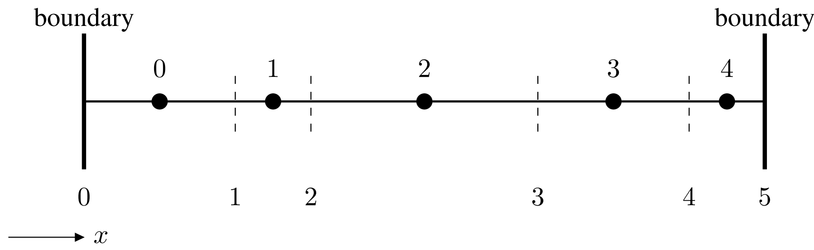


Figure 1.1: 1D grid. `ni=5`. The bullets denote cell centers (and control volume) which are labeled 0–4. Dashed lines denote control volume faces labeled 0–5.

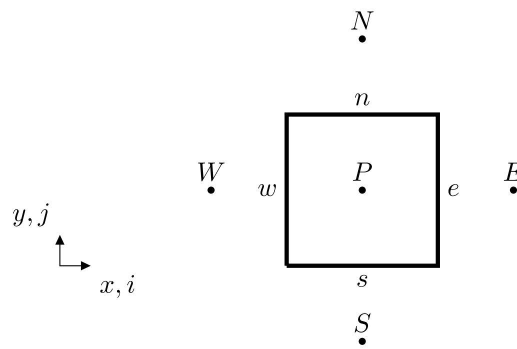


Figure 1.2: Control volume.

1 Geometrical details of the grid

1.1 Grid

The grid (`x2d,y2d`) must be generated by the user. The nodes of the control volume `xp2d,yp2d` are placed at the center of the control volume. In any coordinate direction, lets say ξ, there are `ni+1` control volume faces, and `ni` control volumes. The grid may be curvilinear.

1.1.1 Nomenclature for the grid

Figure 1.1 shows a 1D grid. The first cell is number 0. Note that there are no ghost cells. This means that all Dirichlet boundary conditions must be prescribed using sources.

A schematic 2D control volume grid is shown in Fig. 1.2. Single capital letters define nodes [E(ast), W(est), N(orth) and S(outh)], and single small letters define faces of the control volumes. When a location can not be referred to by a single character, combination of letters are used. The order in which the characters appear is: first east-west (i direction) and then north-south (j direction).

1.1.2 Area calculation of control volume faces

The x and y coordinates of the corners of the face in Fig. 1.3 are given by

```
x2d(i,j),y2d(i,j)
```

```
x2d(i+1,j),y2d(i+1,j)

x2d(i,j+1),y2d(i,j+1)

x2d(i+1,j+1),y2d(i+1,j+1)
```

The vectors $\vec{a}$, $\vec{b}$ and $\vec{c}$ for faces in Fig. 1.3 are set in a manner that the normal vectors point outwards. For the west face they are defined as

$\vec{a}$: from corner (i,j) to (i,j+1)

$\vec{b}$: from corner (i,j) to (i+1,j)

The Cartesian components of $\vec{a}$ and $\vec{b}$ are thus

$$\begin{aligned} a_x &= x2d(i,j+1) - x2d(i,j) \\ a_y &= y2d(i,j+1) - y2d(i,j) \\ b_x &= x2d(i+1,j) - x2d(i,j) \\ b_y &= y2d(i+1,j) - y2d(i,j) \end{aligned} \tag{1.1}$$

Since the grid in the z direction is uniform, it is simple to compute the west and south areas of a control volume. The outwards-pointing vector areas reads

$$\begin{aligned} A_{wx} &= -a_y \Delta z \\ A_{wy} &= a_x \Delta z \\ A_{sx} &= b_y \Delta z \\ A_{sy} &= -b_x \Delta z \end{aligned}$$

which are stored in Python arrays `areawx`, `areawy`, `areasx` and `areasy`.

The area of the control volume in the $x - y$ plane is calculated as the sum of two triangles. The area of the two triangles, $A1$, $A2$, is calculated as the cross product.

$$A1 = \frac{1}{2}|\vec{a} \times \vec{b}|; \qquad A2 = \frac{1}{2}|\vec{b} \times \vec{c}| \tag{1.2}$$

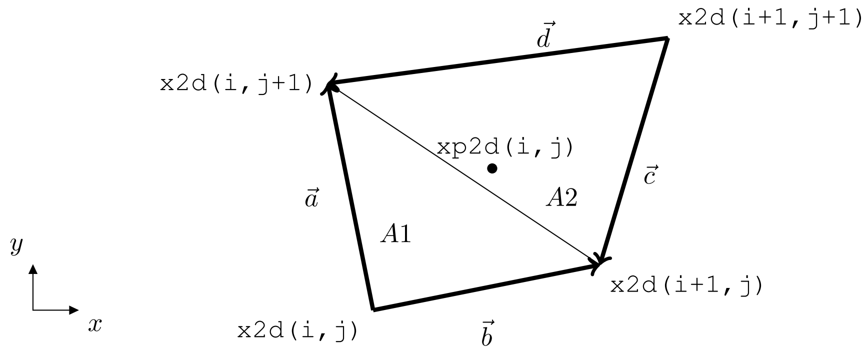


Figure 1.3: Calculation of areas of cell `i,j`.

1.1.3 Interpolation

The nodes where all variables are stored are situated in the center of the control volume. When a variable is needed at a control volume face, linear interpolation is used. The value of the variable ϕ at the west face is

$$\phi_w = f_x \phi_P + (1 - f_x)\phi_W \tag{1.3}$$

where

$$f_x = \frac{|\overrightarrow{Ww}|}{|\overrightarrow{Pw}| + |\overrightarrow{Ww}|} \tag{1.4}$$

where $|\overrightarrow{Pw}|$ is the distance from P (the node) to w (the west face). In **pyCALC-RANS** the interpolation factors (f_x, f_y) are stored in the Python array `fx` and `fy`. The interpolation factor in the z direction is 0.5 since Δz is constant.

All geometrical quantities are computed in the module `init`.

1.2 Gradient

The derivatives of ϕ ($\partial\phi/\partial x_i$) at the cell center are in **pyCALC-RANS** computed as follows. We apply Green's formula to the control volume, i.e.

$$\frac{\partial \Phi}{\partial x} = \frac{1}{V}\int_A \Phi n_x dA, \quad \frac{\partial \Phi}{\partial y} = \frac{1}{V}\int_A \Phi n_y dA$$

where A is the surface enclosing the volume V. For the x component, for example, we get

$$\frac{\partial \Phi}{\partial x} = \frac{1}{V}\left(\Phi_e A_{ex} - \Phi_w A_{wx} + \Phi_n A_{nx} - \Phi_s A_{sx}\right) \tag{1.5}$$

where index w, e, s, n denotes east $(i + 1/2)$, west $(i - 1/2)$, north $(j + 1/2)$ and south $(j - 1/2)$.

The values at the west and south faces of a variable are stored in the Python arrays `u_face_w`, `u_face_s`, `v_face_w`, etc. They are computed in the Python module `compute_face_phi`.

The derivative $\partial\Phi/\partial x$ and $\partial\Phi/\partial x$, are computed in the Python modules `dphidx` and `dphidy`.

2 Diffusion

We start by looking at 1D diffusion for a generic variable, ϕ, with diffusion coefficient Γ

$$\frac{d}{dx}\left(\Gamma\frac{d\phi}{dx}\right) + S = 0.$$

To discretize (i.e. to go from a *continuous* differential equation to an algebraic *discrete* equation) this equation is integrated over a control volume (C.V.), see Fig. 2.1.

$$\int_w^e \left[\frac{d}{dx}\left(\Gamma\frac{d\phi}{dx}\right) + S\right] dx = \left(\Gamma\frac{d\phi}{dx}\right)_e - \left(\Gamma\frac{d\phi}{dx}\right)_w + \bar{S}\Delta x = 0 \tag{2.1}$$

where (see Fig. 2.1):

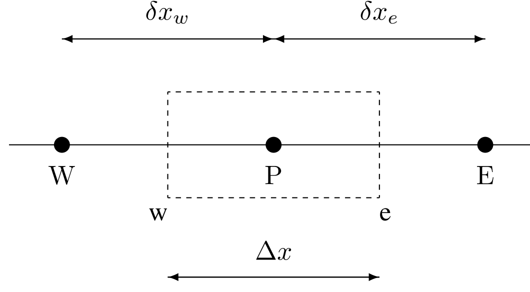


Figure 2.1: 1D control volume. Node P located in the middle of the control volume.

P: an arbitrary node

E, W: its east and west neighbor node, respectively

e, w: the control volume's east and west face, respectively

$\bar{S}$: volume average of S

The variable ϕ and the diffusion coefficient, Γ, are stored at the nodes W, P and E. Now we need the derivatives $d\phi/dx$ at the faces w and e. These are estimated from a straight line connecting the two adjacent nodes, i.e.

$$\left(\frac{d\phi}{dx}\right)_e \simeq \frac{\phi_E - \phi_P}{\delta x_e},\ \left(\frac{d\phi}{dx}\right)_w \simeq \frac{\phi_P - \phi_W}{\delta x_w}. \tag{2.2}$$

The diffusion coefficient, Γ, is also needed at the faces. It is estimated by linear interpolation between the adjacent nodes. For the east face, for example, we obtain

$$\Gamma_w = f_x \Gamma_P + (1 - f_x)\Gamma_W, \tag{2.3}$$

Insertion of Eq. 2.2 into Eq. 2.1 gives

$$\begin{aligned} a_P\phi_P &= a_E\phi_E + a_W\phi_W + S_U \qquad (2.4)\\ a_E &= \frac{\Gamma_e}{\delta x_e} \\ a_W &= \frac{\Gamma_w}{\delta x_w} \\ S_U &= \bar{S}\Delta x \\ a_P &= a_E + a_W \end{aligned}$$

2.1 Convergence criteria

Compute the residual for Eq. 2.4

$$R = \sum_{\text{all cells}} |a_E\phi_E + a_W\phi_W + S_U - a_P\phi_P|$$

In Python it corresponds to $|Ax - b|$. Since we want Eq. 2.4 to be satisfied, the difference of the right-hand side and the left-hand side is a good measure of how well

the equation is satisfied. The residual R is computed using the Python command `np.linalg.norm`. Note that R has the units of the integrated differential equation. For example, for the temperature R has the same dimension as heat transfer rate divided by density, ρ, and specific heat, c_p, i.e. temperature times volume per second $[Km^3/s]$. If $R = 1$, it means that the residual for the computation is 1. This does not tell us anything, since it is problem dependent. We can have a problem where the total heat transfer rate is 1000, and a another where it is only 1. In the former case $R = 1$ means that the solutions can be considered converged, but in the latter case this is not true at all. We realize that we must normalize the residual to be able to judge whether the equation system has converged or not. The criterion for convergence is then

$$\frac{R}{F} \leq \varepsilon$$

where $0.0001 < \varepsilon < 0.01$, and F represents the total flow of ϕ.

Regardless if we solve the continuity equation, the Navier-Stokes equation or the temperature equation, the procedure is the same: F should represent the total flow of the dependent variable.

Continuity equation. F is here the total incoming volume flow $\dot{V}$.

Navier-Stokes equation. The unit is that of a force per unit volume. A suitable value of F is obtained from $F = \dot{V}\bar{u}$ at the inlet.

Temperature equation. F should be the total incoming temperature flow. In a convection-diffusion problem we can take the convective flow at the inlet i.e. $F = \dot{V}T$. In a conduction problem we can integrate the boundary flow, taking the absolute value at each cell, since the sum will be zero in case of internal source. If there are large sources in the computational domain, F could be taken as the sum of all sources.

2.2 2D Diffusion

The two-dimensional diffusion equation for a generic variable ϕ reads

$$\frac{\partial}{\partial x}\left(\Gamma\frac{\partial\phi}{\partial x}\right) + \frac{\partial}{\partial y}\left(\Gamma\frac{\partial\phi}{\partial y}\right) + S = 0. \tag{2.5}$$

In the same way as we did for the 1D case, we integrate over our control volume, but now it's in 2D (see Fig. 2.2, i.e.

$$\int_w^e \int_s^n \left[\frac{\partial}{\partial x}\left(\Gamma\frac{\partial\phi}{\partial x}\right) + \frac{\partial}{\partial y}\left(\Gamma\frac{\partial\phi}{\partial y}\right) + S\right] dxdy = 0.$$

We start by the first term. The integration in x direction is carried out in exactly the same way as in 1D, i.e.

$$\begin{aligned}
\textstyle\int_w^e \quad \int_s^n \left[\frac{\partial}{\partial x}\left(\Gamma\frac{\partial\phi}{\partial x}\right)\right] dxdy &= \int_s^n \left[\left(\Gamma\frac{\partial\phi}{\partial x}\right)_e - \left(\Gamma\frac{\partial\phi}{\partial x}\right)_w\right] dy \\
= \quad \int_s^n \left(\Gamma_e\frac{\phi_E - \phi_P}{\delta x_e} - \Gamma_w\frac{\phi_P - \phi_W}{\delta x_w}\right) dy &
\end{aligned}$$

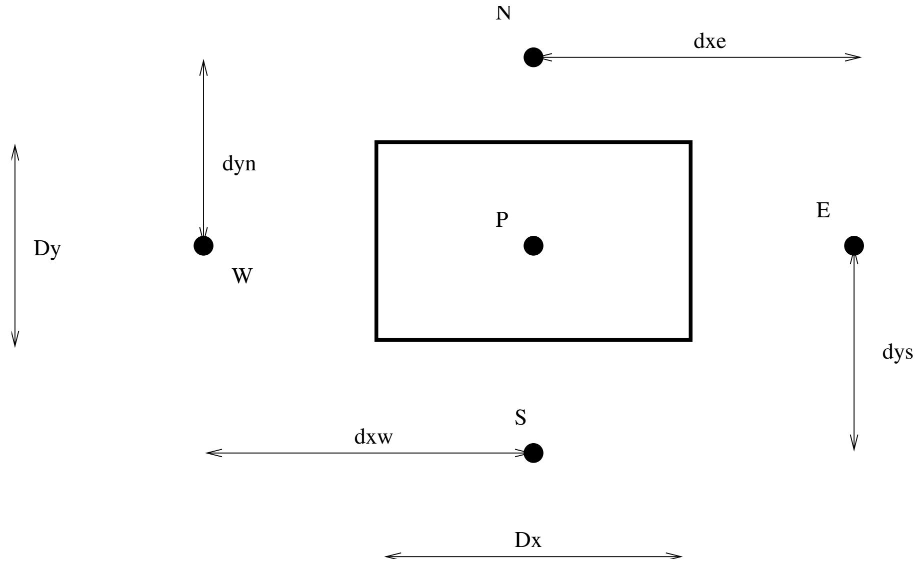


Figure 2.2: 2D control volume.

Now integrate in the y direction. We do this by estimating the integral

$$\int_s^n f(y)dy = f_P \Delta y + \mathcal{O}\left((\Delta y)^2\right)$$

(i.e. f is taken at the mid-point P) which is second order accurate, since it is exact if f is a linear function. For our equation we get

$$\int_s^n \left(\Gamma_e \frac{\phi_E - \phi_P}{\delta x_e} - \Gamma_w \frac{\phi_P - \phi_W}{\delta x_w}\right) dy$$
$$= \left(\Gamma_e \frac{\phi_E - \phi_P}{\delta x_e} - \Gamma_w \frac{\phi_P - \phi_W}{\delta x_w}\right) \Delta y$$

Doing the same for the diffusion term in the y direction in Eq. 2.5 gives

$$\left(\Gamma_e \frac{\phi_E - \phi_P}{\delta x_e} - \Gamma_w \frac{\phi_P - \phi_W}{\delta x_w}\right) \Delta y$$
$$+ \left(\Gamma_n \frac{\phi_N - \phi_P}{\delta y_n} - \Gamma_s \frac{\phi_P - \phi_S}{\delta y_s}\right) \Delta x + \bar{S} \Delta x \Delta y = 0$$

Rewriting it as an algebraic equation for ϕ_P, we get

$$\begin{aligned} a_P \phi_P &= a_E \phi_E + a_W \phi_W + a_N \phi_N + a_S \phi_S + S_U \\ a_E &= \frac{\Gamma_e \Delta y}{\delta x_e},\ a_W = \frac{\Gamma_w \Delta y}{\delta x_w},\ a_N = \frac{\Gamma_n \Delta x}{\delta y_n},\ a_S = \frac{\Gamma_s \Delta x}{\delta y_s} \\ S_U &= \bar{S} \Delta x \Delta y,\ a_P = a_E + a_W + a_N + a_S - S_P. \end{aligned} \tag{2.6}$$

In this 2D equation we have introduced the general form of the source term, $S = S_P \Phi + S_U$; this could also be done in the 1D equation (Eq. 2.4).

For more detail on diffusion, see
http://www.tfd.chalmers.se/~lada/comp_fluid_dynamics/lecture_notes.html

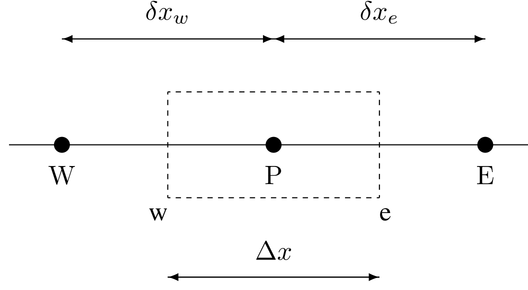


Figure 3.1: 1D control volume. Node P located in the middle of the control volume.

3 Convection – diffusion

The 1D convection-diffusion equation reads

$$\frac{d}{dx}\left(\bar{u}\phi\right) = \frac{d}{dx}\left(\Gamma\frac{d\phi}{dx}\right) + S$$

We discretize this equation in the same way as the diffusion equation. We start by integrating over the control volume (see Fig. 3.1).

$$\int_w^e \frac{d}{dx}\left(\bar{u}\phi\right) dx = \int_w^e \left[\frac{d}{dx}\left(\Gamma\frac{d\phi}{dx}\right) + S\right] dx. \tag{3.1}$$

We start by the convective term (the left-hand side)

$$\int_w^e \frac{d}{dx}\left(\bar{u}\phi\right) dx = \left(\bar{u}\phi\right)_e - \left(\bar{u}\phi\right)_w .$$

We assume the velocity $\bar{u}$ to be known, or, rather, obtained from the solution of the Navier-Stokes equation.

3.1 Central Differencing scheme (CDS)

How to estimate ϕ_e and ϕ_w? The most natural way is to use linear interpolation (central differencing); for the east face this gives

$$\left(\bar{u}\phi\right)_w = \left(\bar{u}\right)_w \phi_w$$

where the convect<u>ing</u> part, $\bar{u}$, is taken by central differencing, and the convect<u>ed</u> part, ϕ, is estimated with different differencing schemes. We start by using central differencing for ϕ so that

$$\left(\bar{u}\phi\right)_w = \left(\bar{u}\right)_w \phi_w, \quad \text{where} \quad \phi_w = f_x\phi_P + (1-f_x)\phi_W$$

where f_x is the interpolation function (see Eq. 2.3, p. 8), and for constant mesh spacing $f_x = 0.5$. Assuming constant equidistant mesh (i.e. $\delta x_w = \delta x_e = \Delta x$) so that $f_x = 0.5$, inserting the discretized diffusion and the convection terms into Eq. 3.1 we obtain

$$\begin{aligned} \left(\bar{u}\right)_e \frac{\phi_E + \phi_P}{2} - \left(\bar{u}\right)_w \frac{\phi_P + \phi_W}{2} &= \\ = \frac{\Gamma_e(\phi_E - \phi_P)}{\delta x_e} - \frac{\Gamma_w(\phi_P - \phi_W)}{\delta x_w} &+ \bar{S}\Delta x \end{aligned}$$

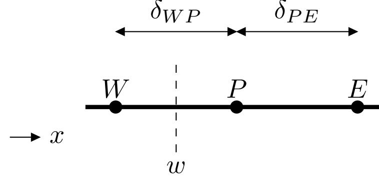


Figure 3.2: Constant mesh spacing. $\bar{u} > 0$.

which can be rearranged as

$$\begin{aligned} a_P\phi_P &= a_E\phi_E + a_W\phi_W + S_U \\ a_E &= \frac{\Gamma_e}{\delta x_e} - \frac{1}{2}(\bar{u})_e,\ a_W = \frac{\Gamma_w}{\delta x_w} + \frac{1}{2}(\bar{u})_w \\ S_U &= \bar{S}\Delta x,\ a_P = \frac{\Gamma_e}{\delta x_e} + \frac{1}{2}(\bar{u})_e + \frac{\Gamma_w}{\delta x_w} - \frac{1}{2}(\bar{u})_w \end{aligned}$$

We want to compute a_P as the sum of its neighbor coefficients to ensure that $a_P \geq a_E + a_W$ which is the requirement to make sure that the iterative solver converges. We can add

$$(\bar{u})_w - (\bar{u})_e = 0$$

(the continuity equation) to a_P so that

$$a_P = a_E + a_W.$$

Central differencing is second-order accurate (easily verified by Taylor expansion), i.e. the error is proportional to $(\Delta x)^2$. This is very important. If the number of cells in one direction is doubled, the error is reduced by a factor of four. By doubling the number of cells, we can verify that the discretization error is small, i.e. the difference between our algebraic, numerical solution and the exact solution of the differential equation.

Central differencing gives negative coefficients when $|Pe| > 2$; this condition is unfortunately satisfied in most of the computational domain in practice. The result is that it is difficult to obtain a convergent solution in steady flow.

3.2 First-order upwind scheme

For turbulent quantities upwind schemes must usually be used in order stabilize the numerical procedure. Furthermore, the source terms in these equations are usually very large which means that an accurate estimation of the convection term is less critical.

In this scheme the face value is estimated as

$$\phi_w = \begin{cases} \phi_W & \text{if } \bar{u}_w \geq 0 \\ \phi_P & \text{otherwise} \end{cases}$$

- first-order accurate
- bounded

The large drawback with this scheme is that it is inaccurate.

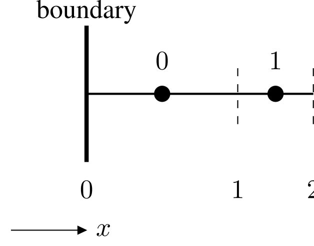


Figure 3.3: 1D grid. Boundary conditions at $x = 0$.

3.3 Hybrid scheme

This scheme is a blend of the central differencing scheme and the first-order upwind scheme. We learned that the central scheme is accurate and stable for $|Pe| \leq 2$. In the Hybrid scheme, the central scheme is used for $|Pe| \leq 2$; otherwise the first-order upwind scheme is used. This scheme is only marginally better than the first-order upwind scheme, as normally $|Pe| > 2$. It should be considered as a first-order scheme.

3.4 Inlet boundary conditions using source term

Since **pyCALC-RANS** does not use any ghost cells or cell centers located at the boundaries, the boundary conditions must be prescribed through source terms. By default, there is no flux through the boundaries and hence Neumann boundary conditions are set by default. Here, we describe how to set Dirichlet boundary conditions.

Consider discretization in a cell, P, adjacent to an inlet, see Fig. 3.3. Consider only convection. For the $\bar{u}$ equation at cell $i = 0$ we get

$$\begin{aligned} a_P \bar{u}_P &= a_W \bar{u}_W + a_E \bar{u}_E + S_U \\ a_P &= a_W + a_E - S_P, \quad a_W = C_w, \quad a_E = -0.5 C_e \\ C_w &= \bar{u}_W A_w \\ a_P &= C_w - 0.5 C_e \end{aligned} \tag{3.2}$$

Note there's no 0.5 in front of C_w since the west node is located *at* the inlet. Since there is no cell west of $i = 0$, Eq. 3.2 has to be implemented with additional source terms

$$\begin{aligned} a_w &= 0 \\ S^u_{U,add} &= C_w \bar{u}_{in} \\ S^u_{P,add} &= -C_w \end{aligned} \tag{3.3}$$

For $\bar{v}$ it reads

$$\begin{aligned} S^v_{U,add} &= C_w \bar{v}_{in} \\ S^v_{P,add} &= -C_w \end{aligned} \tag{3.4}$$

$$\tag{3.5}$$

The additional term for the diffusion reads

$$S^u_{U,add,diff} = \frac{\nu_{tot} A_w}{\Delta x} \bar{u}_{in} \tag{3.6}$$

$$
\begin{aligned}
S^{v}_{U,add,diff} &= \frac{\nu_{tot} A_w}{\Delta x} \bar{v}_{in} \\
S_{P,add,diff} &= -\frac{\nu_{tot} A_w}{\Delta x}
\end{aligned}
$$

where $S_{P,add,diff}$ is the same for $\bar{u}$ and $\bar{v}$. The viscous part of Eq. 3.6 is implemented in module `bc`. The turbulent part and the convective part (Eqs. 3.3 and 3.4) are implemented in `module_u`, `module_v` etc.

3.5 Wall boundary conditions using source term

We use exactly the same procedure as in Section 3.4. At walls, there is no convection and the velocity is zero. Hence we simply use Eq. 3.6 with $\bar{u} = \bar{v} = 0$, i.e. (for west wall)

$$
S_{P,add,diff} = -\frac{\nu A_w}{\Delta x}
$$

Note that we use ν instead of ν_{tot} since the turbulent viscosity is zero at the wall.

This boundary condition is implemented in module `bc`.

3.6 Pressure correction equation

The pressure correction equation is obtained by applying the SIMPLEC algorithm [2] on the non-staggered grid. The mass flux $\dot{m}$ is divided into one old value, $\dot{m}^*$, and another correction value, $\dot{m}'$. The mass flux correction at the east face can be calculated by

$$
\dot{m}_e = \dot{m}^*_e + \dot{m}'_e, \quad \dot{m}'_e = \left(\vec{A} \cdot \vec{u}'\right)_e = (A_{ex} u'_e + A_{ey} v'_e) \tag{3.7}
$$

where u' and v' are the correction velocities. The velocity components are related to the pressure gradient

$$
u' = -\frac{\Delta V}{a_P}\frac{\partial p'}{\partial x}, \quad v' = -\frac{\Delta V}{a_P}\frac{\partial p'}{\partial y}, \tag{3.8}
$$

where V_P denotes the volume of the control volume. By introducing Eq. 3.7 into Eq. 3.8 we obtain

$$
\dot{m}' = -\left[\frac{\Delta V_P}{a_P}\vec{A} \cdot \nabla p'\right] = -\frac{\Delta V_P}{a_P}\left[\vec{A}_x \frac{\partial p'}{\partial x} + \vec{A}_y \frac{\partial p'}{\partial y}\right] \tag{3.9}
$$

Consider, for simplicity, the continuity equation in one dimension

$$
\dot{m}_e - \dot{m}_w = 0 \tag{3.10}
$$

If $\dot{m} = \dot{m}^* + \dot{m}'$ and Eq. 3.9 are substituted into eq. 3.10 we obtain

$$
\left[\frac{\Delta V_P A_x}{a_P}\frac{\partial p'}{\partial x}\right]_w - \left[\frac{\Delta V_P A_x}{a_P}\frac{\partial p'}{\partial x}\right]_e + \dot{m}^*_e - \dot{m}^*_w = 0 \tag{3.11}
$$

This is a diffusion equation for the pressure correction p' which is discretized as Eq. 2.6 by replacing Φ by p' and setting $\Gamma = \Delta V_P/a_P$. The boundary conditions at all boundaries is homogeneous Neumann, i.e. $\partial p'/\partial x = 0$ at west and east boundaries and $\partial p'/\partial y = 0$ at south and north boundaries.

Given the boundary conditions for the flow to be predicted, the solution proceeds as follows

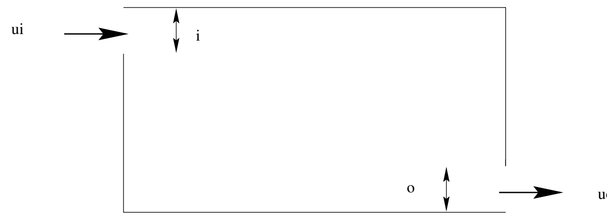


Figure 4.1: Outlet boundary condition. Small outlet

1. Assign initial values (usually 10^{-10}) to the variable fields $\bar{u}^*$, $\bar{v}^*$, $\bar{p}^*$ and turbulence quantities k and ω.
2. Solve the $\bar{u}$-momentum equation by first calculating the coefficients and sources, then imposing the $\bar{u}$-velocity boundary conditions followed by application of the Python solver.
3. Point 2 is repeated for $\bar{v}$
4. Solve the pressure-correction equation by first calculating the coefficients and sources, then imposing the pressure-correction boundary conditions followed by application of the Python solver.
5. Correct the velocity fields $\bar{u}^*$, $\bar{v}^*$ and mass fluxes (see Eq. 3.7) $\dot{m}_e^*$ and $\dot{m}_n^*$ with u', v'.
6. Correct the pressure field $\bar{p}^*$ with p' to give the correct pressure field $\bar{p}$.
7. Solve additional equations such as k, ω, T etc.
8. Go to step 2 and repeat step 2 to 7 until convergence.

You can find more details about discretization and the pressure correction method in `lecture notes` (Chapter 2-9).

4 Boundary Conditions

4.1 Outlet velocity, small outlet

For *small* outlets, the outlet velocity can be determined from global continuity. As the outlet is small a constant velocity over the whole outlet can be used. The outlet velocity is set as (see Fig. 4.1)

$$\bar{u}_{in}h_{in} = \bar{u}_{out}h_{out} \;\Rightarrow \bar{u}_{out} = \bar{u}_{in}h_{in}/h_{out}$$

4.2 Outlet velocity, large outlet

For *large* outlets the outlet velocity must be allowed to vary over the outlet. The proper boundary condition in this case is $\partial\bar{u}/\partial x = 0$. Hence it is important that the flow near the outlet is fully developed, so that this boundary condition corresponds to the flow conditions. The best way to ensure this is to locate the outlet boundary sufficiently far downstream. If we have a recirculation region in the domain (see Fig. 4.2), the outlet should be located sufficiently far downstream of this region so that $\partial\bar{u}/\partial x \simeq 0$.

The outlet boundary condition is implemented as follows (see Fig. 4.2)

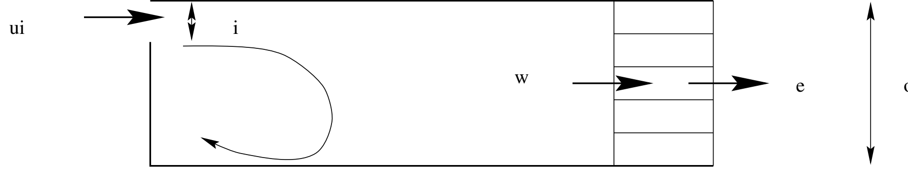


Figure 4.2: Outlet boundary condition. Large outlet.

1. Set $\bar{u}_e = \bar{u}_w$ for all nodes (i.e. for $j = 0$ to 4, see Fig. 4.2);
2. In order to speed up convergence, enforce global continuity.
   - Inlet volume flow: $\dot{V}_{in} = \sum_{inlet} \bar{u}_{in}\Delta y$
   - Outlet volume flow: $\dot{V}_{out} = \sum_{outlet} \bar{u}_{out}\Delta y$
   - Compute correction velocity: $\bar{u}_{corr} = (\dot{V}_{in} - \dot{V}_{out})/(A_{out})$, where $A_{out} = \sum_{outlet} \Delta y$.
   - Correct $\bar{u}_e$ so that global continuity (i.e. $\dot{V}_{in} = \dot{V}_{out}$) is satisfied: $\bar{u}_e^{new} = \bar{u}_e + \bar{u}_{corr}$

This boundary condition is implemented in module `modify_outlet`.

4.3 Remaining variables

Set $\partial\Phi/\partial x = 0$, and implement it through $\Phi_{ni} = \Phi_{ni-1}$ each iteration. This is done in module `compute_face_phi` if `phi_bc_east_type ='n'`.

5 The $k - \omega$ model

<u>modules:</u> `calck_kom, calcom, vist_kom`

The Wilcox $k-\omega$ RANS turbulence model [3] reads

$$\begin{aligned}
\frac{\partial k}{\partial t} + \frac{\partial \bar{v}_i k}{\partial x_i} &= P^k - C_\mu k\omega + \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_j}\right] \\
\frac{\partial \omega}{\partial t} + \frac{\partial \bar{v}_i \omega}{\partial x_i} &= C_{\omega_1}\frac{\omega}{k}P^k - C_{\omega 2}\omega^2 + \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_\omega}\right)\frac{\partial \omega}{\partial x_j}\right] \qquad (5.1) \\
\nu_t &= \frac{k}{\omega}
\end{aligned}$$

(5.2)

where $c_\mu = 0.09$, $c_{\omega_1} = 5/9$, $c_{\omega_2} = 3/40$, $\sigma_k = 0.5 = \sigma_\omega = 2.0$. The wall boundary conditions are

$$k_w = 0, \quad \omega_w = 10\frac{6\nu}{C_{\omega 2} y^2} \qquad (5.3)$$

where y is the wall distance between the wall-adjacent cell center and the wall. Sometimes we prescribe this boundary condition by setting ω in the control volume adjacent to the wall. Then we omit the factor of 10 so that

$$\omega_P = \frac{6\nu}{C_{\omega 2} y^2} \qquad (5.4)$$

where index P denotes the cell P adjacent to the wall. where y is the wall distance between the wall-adjacent cell center and the wall. Sometimes we prescribe this

6 Modules

6.1 bc_outlet_bc

Neumann outlet boundary conditions are set.

6.2 calck

Source terms in the k equation (Wilcox model) are computed, see Section 5. The user can define additional source terms in `modify_k`.

6.3 calcom

Source terms in the ω equation (Wilcox model) are computed, see Section 5. The user can define additional source terms in `modify_om`.

6.4 calcp

Coefficients in the p' equation, see Section 3.6.

6.5 calcu

Source terms in the $\bar{u}$ equation are computed. The user can define additional source terms in `modify_u`.

6.6 calcv

Source terms in the $\bar{v}$ equation are computed. The user can define additional source terms in `modify_v`.

6.7 coeff

The coefficient a_W, a_E, a_S, a_N are computed. There are two different discretization schemes: central differencing scheme (CDS) and the hybrid scheme (first-order upwind and CDS).

6.8 compute_face_phi

Compute the face values of a variable.

6.9 conv

Compute the convection as a vector product $\mathbf{v} \cdot \mathbf{A}$ at the west and south faces (stored in arrays `convw` and `convn` and `convl`. Note that they are defined as the volume flow going into the control volume.

6.10 correct_u_v_p

After the presssure correction has been solved, the convections `convw` and `convs` (which are defined at the control volume faces) and the velocities, `u2d` and `v2d` and pressure, `p2d` are corrected so as to satisfy continuity.

6.11 fix_omega

This routine may be used for fix ω in the wall-adjacent *cell center* according to Eq. 5.4 rather than as a wall-boundary condition (Eq. 5.3). Note that it is called just before the solver is called. For fixing ω near a south boundary we use

```
aw2d[:,0]=0
ae2d[:,0]=0
as2d[:,0]=0
an2d[:,0]=0
al2d[:,0]=0
su2d[:,0]=om_bc_south
```

6.12 dphidx, dphidy

The derivative in x or y direction are computed, see Section 1.2.

6.13 init

Geometric quantities such as areas, volume, interpolation factors etc are computed.

6.14 modify_k, modify_om, modify_u, modify_v

The sources `su2d` and `sp2d` can be modified for the k, ω, $\bar{u}$ and $\bar{v}$ equations.

6.15 modify_case.py

This file includes `modify_k`, ...`modify_omega` and `modify_conv`, `modify_init`, `modify_inlet`, `modify_outlet`, `fix_omega` and `modify_vis`.

6.16 modify_init

The user can set initial fields. If `restart=True`, these fields are over-written with the fields from the restart file.

6.17 print_indata

Prints the indata set by the user.

6.18 read_restart_data

This module is called when `restart=True`. Initial fields from files

- `u2d_saved.npy, v2d_saved.npy, p2d_saved.npy, k2d_saved.npy, om2d_saved.npy`

are read from a previous simulation.

6.19 save_data

This module is called when `save=True`. The

- $\bar{u}$, $\bar{v}$, $\bar{p}$, k and ω fields

are stored in the files

- `u2d_saved.npy, v2d_saved.npy, p2d_saved.npy, k2d_saved.npy, om2d_saved.npy`.

6.20 save.file

This is file, not a module. It is read every second time step. It should include a integer '0' or '1'. If it's '1', the module `save_data` is called. The object is to be able to save data during a long simulation,

6.21 save_vtk

The results are stored in VTK format. It is called if `vtk=True`. You must then set the name of the VTK file names, i.e. `vtk_file_name`.

6.22 setup_case.py

In this module the user sets up the case (time step, turbulence model, turbulence constants, type of boundary condition, solver, convergence criteria, etc)

6.23 solve_2d

This module can be used for all variables except pressure, $\bar{p}$. With the coefficient arrays `aw2d, ae2d, as2d, ...` a sparse matrix is created, `A`. The equation system is solved using the Python solver `linalg.lgmres` or `linalg.gmres`.

6.24 vist_kom

The turbulent viscosity is computed using the $k-\omega$ model, see Section 5

7 Lid-driven cavity at $Re = 1000$

To follow the execution of **pyCALC-RANS** , it is recommended to start reading at the line *the execution of the code starts here*. To find where the solution procedure starts, look for the line *start of global iteration process*. You can also kook at the **pyCALC-RANS** flowchart.

The lid-driven cavity is shown in Fig. 7.1 with the grid. The top wall is moving. The boundary conditions are $u = v = 0$ an all boundaries (walls) except the top wall for which $U_{wall} = 1$. The length of all side is one, i.e. $L = 1$. The Reynolds number, $Re_L = U_{wall}L/\nu = 1000$.

The grid is created using the script `generate-lid-grid.py`. The number of cells is set to $ni = nj = 60$. The grid is stretched by 5% from all four walls.

```
   import numpy as np
   import sys
   ni=60
   nj=ni
   yfac=1.05 # stretching
   viscos=1/1000
   dy=0.1
   ymax=2
   yc=np.zeros(nj+1)
   yc[0]=0.
   for j in range(1,int(nj/2)+1):
       yc[j]=yc[j-1]+dy
       dy=yfac*dy

   ymax_scale=yc[int(nj/2)]

# cell faces
   for j in range(1,int(nj/2)+1):
   yc[j]=yc[j]/ymax_scale
   yc[nj-j+1]=ymax-yc[j-1]

   yc[int(nj/2)]=1

# make side=1
   yc=yc/yc[-1]

# make it 2D
   y2d=np.repeat(yc[None,:], repeats=ni+1, axis=0)

   y2d=np.append(y2d,nj)
   np.savetxt('y2d.dat', y2d)
# x grid
   xc = np.linspace(0, xmax, ni+1)
# make it 2D
```

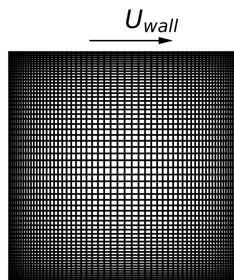


Figure 7.1: Lid-driven cavity with grid. Top wall is moving.

```
x2d=np.repeat(xc[:,None], repeats=nj+1, axis=1)
x2d_org=x2d
x2d=np.append(x2d,ni)
np.savetxt('x2d.dat', x2d)
```

The case is defined in modules `setup_case` and `modify_case`. They are located in a directory with the name `channel` (or something similar). Enter this directory.

7.1 setup_case.py

This module consists of 10 sections.

7.1.1 Section 1

I choose the hybrid scheme for convection

```
scheme='h'
```

7.1.2 Section 3

I will not initial conditions from a previous simulation (`restart=False`) and I also save the new results to disk (`save=True`) which can be used as initial condition for next simulation.

```
restart =False
save= True
```

7.1.3 Section 4

The viscosity is set.

```
viscos=1/1000
```

7.1.4 Section 6

The maximum number of global iterations is set to 500.

The AMG solver s chosen for the pressure correction and the convergence level in the AMG solver is set to $5 \cdot 10^{-2}$. Note that this is a *relative* limit, i.e. ratio of final to initial L2 norm.

The 'lgmres' sparse matrix solver in Python is set for $\bar{u}$ and $\bar{v}$. The maximum number of iterations is set to 50 and the convergence level to 10^{-6}. The global convergence limit, `sormax`, is set to 10^{-5} and the maximal number of gloabl iterations to 1000.

```
maxit=1000
sormax=1e-5
amg_relax='default'
solver_vel='lgmres'
nsweep_vel=50
convergence_limit_u=1e-5
convergence_limit_v=1e-5
convergence_limit_w=1e-5
convergence_limit_p=5e-4
```

The convergence limit in the Python solvers is defined as

$$|Ax - b|/|b| < \gamma \tag{7.1}$$

where γ is the convergence limit. The norm of, for example f, is computed as (L2 norm)

$$|f| = \left[\sum_{\text{all cells } i} f_i^2 \right]^{1/2}$$

7.1.5 Section 7

The flow during the iterations and time steps is monitored in cell $(i, j) = (10, 10)$.

```
imon=10
jmon=10
```

7.1.6 Section 8

I don't want to store data on VTK format (if you do, you can visualize the flow with the open-source post-processing tool ParaView). Hence

```
vtk=False
```

7.1.7 Section 9

The residual of the momentum equation and the continuity equation are normalized by `resnorm_vel` and `resnorm_p` which are set to

```
uin=1
resnorm_p=uin*zmax*y2d[1,-1]
resnorm_vel=uin**2*zmax*y2d[1,-1]
```

7.1.8 Section 10

The boundary conditions are set here. All boundaries are defined as no-slip walls (Dirichlet)

```
u_bc_south_type='d'
u_bc_north_type='d'
v_bc_south_type='d'
v_bc_north_type='d'
```

and the value for all variables is set to zero for all except the top (north) wall where $U_{wall} = 1$, i.e.

```
u_bc_south=np.zeros((ni,nk))
u_bc_north=np.ones((ni,nk))
v_bc_south=np.zeros((ni,nk))
v_bc_north=np.zeros((ni,nk))
```

7.2 modify_case.py

Initial condition and additional boundary conditions – mostly implicit – are set in this file. It includes a module which are called for every flow field variable, i.e. modify_u, modify_v, modify_p, modify_k and modify_om. It includes also modules for modifying initial boundary conditions (modify_init), convections (modify_conv), inlet (modify_inlet) and outlet boundary conditions (modify_outlet). There is also a module fix_omega which is used for setting ω according to Eq. 5.4. The only thing I add in modify_case.py for this flow is to monitor how the u velocity changes when the flow goes toward convergence.

7.2.1 modify_u

I plot the values of u in six points for every iteration

```
global file1

if iter == 0:
   print('file1 opened')
   l1=[iter,u2d[ni-5,5],u2d[ni-5,10],u2d[ni-5,20],\
        u2d[ni-5,30],u2d[ni-5,40],u2d[ni-5,50]]
   np.savetxt('u-iter-history.dat', l1, newline=" ")
   file1=open('u-iter-history.dat','a') #append
else:
   print('file1 printed')
   file1.write("\n")
   l1=[iter,u2d[ni-5,5],u2d[ni-5,10],u2d[ni-5,20],\
        u2d[ni-5,30],u2d[ni-5,40],u2d[ni-5,50]]
   np.savetxt(file1, l1, newline=" ")
```

This monitoring is used as an extra check that the flow has converged, i.e. I want to make sure that u has stopped changing during the solution process.

7.3 Run the code

The bash script run-python is used which reads

```
#!/bin/bash
# delete forst line
sed  '/setup_case()/d' setup_case.py > temp_file
# add new first line plus global declarations
cat  ../global temp_file  modify_case.py  \
../pyCALC-RANS.py  > exec-pyCALC-RANS.py;
/usr/bin/time -a  -o out  ~/anaconda3/bin/python -u  exec-pyCALC-RANS.py > out
```

This script simply collects all Pythons files in one file and the global declarations (which gives all modules access to the global variables) into the file exec-pyCALC-RANS.py and then executes it. Now run the code with the command

```
run-python &
```

If you're using Windows the script work in an Ubuntu terminal window. However, of you prefer to run the code in your Windows environment, you can simply run the executable (which resides in every folder `boundary-layer-laminar`, `channel-2000` . . .)

The input data is written to the file `out`. In this file you also find convergence history etc. To check the convergence history type

```
grep 'max res' out
```

The code also writes out maximum values of some variables (in order to detect if the simulation is diverging). Check this by

```
grep umax out
```

If the Python sparse matrix solved does not converge, a warning is written. Check this with

```
grep warn out
```

You can check that the Python sparse matrix reduces the residuals. Type

```
grep history out
```

You see three lines per time step, i.e. the residuals for $\bar{u}$, $\bar{v}$ and p' equation.

Plot the results using the script `pl_uvw_lid.py`.

8 Fully-developed channel flow at $Re_\tau = 5\,200$

You find `setup_case.py` and `modify_case.py` in a directory with the name `channel-5200` (or something similar). Go into this directory.

I generate a new grid. I use the same Python script as in Section 7 but I set one cells, `ni=1` in the x direction, `xmax=1` and set the stretching factor in the y direction to 1.15. The grid is created using the script `generate-channel-grid.py`.

8.1 setup_case.py

8.1.1 Section 1

I choose the hybrid scheme for both velocities and k and ω

```
scheme='h'
scheme_turb='h'
```

8.1.2 Section 2

I choose the $k - \omega$ RANS model.

```
kom = True
```

8.1.3 Section 3

I don't start from a previous solution.

```
restart = False
```

8.1.4 Section 4

The viscosity is set.

```
viscos=1/5200
```

8.1.5 Section 8

The direct solver is chosen for all variables

```
solver_vel='direct'
solver_turb='direct'
solver_pp='direct'
```

8.1.6 Section 9

For estimating scaling of the residuals, I set u_{in}, i.s.

```
uin=20
```

8.1.7 Section 10

This is a fully developed channel flow for which $v_2 = \partial u/\partial x = 0$. Hence, I set homogeneous Neumann boundary conditions for all variables in the x direction

```
u_bc_west_type='n'
u_bc_east_type='n'

v_bc_west_type='n'
v_bc_east_type='n'

k_bc_west_type='n'
k_bc_east_type='n'

om_bc_west_type='n'
om_bc_east_type='n'
```

The north and south boundaries are walls for which I set Dirichlet (no-slip)

```
u_bc_south_type='d'
u_bc_north_type='d'

v_bc_south_type='d'
v_bc_north_type='d'

k_bc_south_type='d'
u_bc_north_type='d'

om_bc_south_type='d'
om_bc_north_type='d'
```

The values are set to zero for $\bar{u}$, $\bar{v}$ and k, i.e.

```
u_bc_south=np.zeros(ni)
u_bc_north=np.zeros(ni)

v_bc_south=np.zeros(ni)
v_bc_north=np.zeros(ni)

k_bc_south=np.zeros(ni)
k_bc_north=np.zeros(ni)
```

For ω, I use Eq. 5.4

```
xwall_s=0.5*(x2d[0:-1,0]+x2d[1:,0])
ywall_s=0.5*(y2d[0:-1,0]+y2d[1:,0])
dist2_s=(yp2d[:,0]-ywall_s)**2+(xp2d[:,0]-xwall_s)**2
om_bc_south=6*viscos/0.075/dist2_s

xwall_n=0.5*(x2d[0:-1,-1]+x2d[1:,-1])
ywall_n=0.5*(y2d[0:-1,-1]+y2d[1:,-1])
dist2_n=(yp2d[:,-1]-ywall_n)**2+(xp2d[:,-1]-xwall_n)**2
om_bc_north=6*viscos/0.075/dist2_n
```

8.2 **modify_case.py**

I set the driving volume force to one

```
su2d=su2d+vol
```

A force balance force the entire channel gives

$$\underbrace{L \cdot 2h}_{\text{volume}} - \underbrace{L \cdot \tau_w}_{\text{two wall shear stresses}} = 0$$

where h and L denote half channel height and length of channel, respectively. I get that the wall shear stress, τ_w, must be equal to one. Hence, I know that when the $\bar{u}$ momentum equation has converged, then $\tau_w = 1$ at both walls. Let's use that as a check of convergence has been obtained.

```
tauw_south=viscos*np.sum(as_bound*u2d[:,0])/x2d[-1,0]
tauw_north=viscos*np.sum(an_bound*u2d[:,-1])/x2d[-1,0]

print(f"{'tau wall, south: '} {tauw_south:.3f},\
{'  tau wall, north: '} {tauw_north:.3f}")
```

Plot the results using the script `pl_uvw-channel.py`. In this script I save y, u, k, ω and $\overline{v_1'v_2'}$ in the file

`y_u_k_om_uv_5200-RANS-code.txt`

These data will be used for prescribing inlet b.c. in Section 9.

9 Channel flow (inlet outlet) at $Re_\tau = 5\,200$

You find `setup_case.py` and `modify_case.py` in a directory with the name `channel-5200-inlet` (or something similar). Go into this directory.

In this section I comment only on differences compared to the case in Section 8.

I generate a new grid. I use the same Python script as in Section 8 but I set 30 cells, `ni=30` in the x direction and `xmax=15`.

9.1 `setup_case.py`

9.1.1 Section 10

This is an inlet-outlet flow. Hence, I set Dirichlet b.c. at the inlet.

```
u_bc_west_type='d'
v_bc_west_type='d'
k_bc_west_type='d'
om_bc_west_type='d'
```

The b.c. at the east, south and north boundaries are the same as in Section 8. For ω, I use Eq. 5.4

```
xwall_s=0.5*(x2d[0:-1,0]+x2d[1:,0])
ywall_s=0.5*(y2d[0:-1,0]+y2d[1:,0])
dist2_s=(yp2d[:,0]-ywall_s)**2+(xp2d[:,0]-xwall_s)**2
om_bc_south=10*6*viscos/0.075/dist2_s

xwall_n=0.5*(x2d[0:-1,-1]+x2d[1:,-1])
ywall_n=0.5*(y2d[0:-1,-1]+y2d[1:,-1])
dist2_n=(yp2d[:,-1]-ywall_n)**2+(xp2d[:,-1]-xwall_n)**2
om_bc_north=10*6*viscos/0.075/dist2_n
```

Note that in this case I fix ω at the wall-adjacent cells whereas I in Section 8 set ω *at* the wall.

9.2 `modify_case.py`

9.2.1 `modify_init`

Here I set initial b.c. I load the data from the results in Section 8. I interpolate the data to the grid. Note that this is not really necessary since the grid is the same in this case as in Section 8. But it allows us to modify the grid.

```
data=np.loadtxt('y_u_k_om_uv_5200-RANS-code.txt')

y_rans_in=data[:,0]
u_rans_in=data[:,1]
k_rans_in=data[:,2]
om_rans_in=data[:,3]
uv_rans_in=data[:,4]
```

```
   y_rans=yp2d[0,:]

   u_rans=np.interp(y_rans, y_rans_in, u_rans_in)
   k_rans=np.interp(y_rans, y_rans_in, k_rans_in)
   om_rans=np.interp(y_rans, y_rans_in, om_rans_in)
   uv_rans=np.interp(y_rans, y_rans_in, uv_rans_in)

# set inlet field in entre domain
   u3d=np.repeat(u_rans[None,:], repeats=ni, axis=0)
   k3d=np.repeat(k_rans[None,:], repeats=ni, axis=0)
   om3d=np.repeat(om_rans[None,:], repeats=ni, axis=0)
```

9.2.2 modify_inlet

Here I set inlet b.c. I load the same data as in `modify_init`. Then I assign the data to the arrays which hold the b.c., i.e. `u_bc_west=u_rans`, `k_bc_west` and `om_bc_west`.

```
   data=np.loadtxt('y_u_k_om_uv_5200-RANS-code.txt')

   y_rans_in=data[:,0]
   u_rans_in=data[:,1]
   k_rans_in=data[:,2]
   om_rans_in=data[:,3]
   uv_rans_in=data[:,4]

   y_rans=yp2d[0,:]

   u_rans=np.interp(y_rans, y_rans_in, u_rans_in)
   k_rans=np.interp(y_rans, y_rans_in, k_rans_in)
   om_rans=np.interp(y_rans, y_rans_in,om_rans_in)

   u_bc_west=u_rans
   k_bc_west=k_rans
   om_bc_west=om_rans
```

9.2.3 modify_u

No volume source is used. The turbulent diffusion is added at the inlet

```
   su2d[0,:]= su2d[0,:]+convw[0,:]*u_bc_west
   sp2d[0,:]= sp2d[0,:]-convw[0,:]
   vist=vis2d[0,:,]-viscos
   su2d[0,:]=su2d[0,:]+vist*aw_bound*u_bc_west
   sp2d[0,:]=sp2d[0,:]-vist*aw_bound
```

The viscous diffusion is added in module `bc`.

9.2.4 modify_v

The turbulent diffusion is added at the inlet

```
su2d[0,:]= su2d[0,:]+convw[0,:]*v_bc_west
sp2d[0,:]= sp2d[0,:]-convw[0,:]
vist=vis2d[0,:,]-viscos
su2d[0,:]=su2d[0,:]+vist*aw_bound*v_bc_west
sp2d[0,:]=sp2d[0,:]-vist*aw_bound
```

The viscous diffusion is added in module bc.

9.2.5 modify_k

The turbulent diffusion is added at the inlet

```
su2d[0,:]= su2d[0,:]+convw[0,:]*k_bc_west
sp2d[0,:]= sp2d[0,:]-convw[0,:]
vist=vis2d[0,:,]-viscos
su2d[0,:]=su2d[0,:]+vist*aw_bound*k_bc_west
sp2d[0,:]=sp2d[0,:]-vist*aw_bound
```

The viscous diffusion is added in module bc.

9.2.6 modify_om

The turbulent diffusion is added at the inlet

```
su2d[0,:]= su2d[0,:]+convw[0,:]*om_bc_west
sp2d[0,:]= sp2d[0,:]-convw[0,:]
vist=vis2d[0,:,]-viscos
su2d[0,:]=su2d[0,:]+vist*aw_bound*om_bc_west
sp2d[0,:]=sp2d[0,:]-vist*aw_bound
```

The viscous diffusion is added in module bc.

9.2.7 modify_outlet

Outlet b.c. are set according to Section 4.2

```
# inlet
   flow_in=np.sum(convw[0,:])
   flow_out=np.sum(convw[-1,:])
   area_out=np.sum(areaw[-1,:])

   uinc=(flow_in-flow_out)/area_out
   ares=areaw[-1,:]
   convw[-1,:]=convw[-1,:]+uinc*ares
```

9.2.8 fix_omega

Here I set ω at the first interior cell according to Eq. 5.4. I do that by setting all coefficients to zero except a_P which is set to one

```
aw2d[:,0]=0
ae2d[:,0]=0
as2d[:,0]=0
an2d[:,0]=0
ap2d[:,0]=1
su2d[:,0]=om_bc_south
```

`om_bc_south` was set in the boundary-condition part in `setup_case.py`.

10 RANS of boundary layer flow using $k-\omega$

You find `setup_case.py` and `modify_case.py` in a directory with the name `boundary-layer-RANS-kom` (or something similar). Go into this directory.

I generate a new grid. The first cell is set to $\Delta t = 7.83 \cdot 10^{-4}$. I stretch the grid in the y direction by 10% but limit the cell size to $\Delta y_{max} = 0.05$. The number of cells is set to `nj=90`. In the x direction, the first cells is set to $\Delta x = 0.03$ and then I stretch it by 0.5%. I set the number of cells to `ni=300`. In the z direction I set the number of cells to two and the extent to one, i.e. the `z.dat` is modified to `1.0, 2`. The grid is created using the script `generate-bound-layer-grid.py`.

10.1 `setup_case.py`

10.1.1 Section 1

Hybrid discretization is set for all variables.

```
scheme='h'  #hybrid
scheme_turb='h'
```

10.1.2 Section 2

The $k-\omega$ RANS model is selected.

```
kom = True
```

10.1.3 Section 4

The viscosity is set.

```
viscos=3.57E-5
```

10.1.4 Section 5

I set under-relation factor of 0.5 for all variables except for p'

```
urfvis=0.5
urf_vel=0.5
urf_k=0.5
urf_omega=0.5
urf_p=1.0
```

10.1.5 Section 6

The `lgmres` solver is chosen for the velocities, the `pyamg` for p' and `gmres` for k and ω.

```
solver_vel='lgmres'
solver_pp='pyamg'
solver_turb='gmres'
```

The convergence limit in the Python solvers is set to 10^{-6} for all variables except p' for which the (relative) limit is set to 0.05

```
convergence_limit_u=1e-6
convergence_limit_v=1e-6
convergence_limit_k=1e-6
convergence_limit_om=1e-8
convergence_limit_pp=5e-2
```

Note that the convergence level for ω must be set to a lower level (10^{-8}) than for the other variables (10^{-6}); if not, the ω equation does not converge and the skin friction is over-predicted by 10% at $x = x_{max}$. To verify that all equations do converge, check that with the command (see Section 7.3)

```
grep 'max res' out
```

The global convergence limit is set to $5 \cdot 10^{-5}$ and the maximum number of iterations is set to 2000

```
sormax=5e-5
maxit=2000
```

If you want to decrease that level, you must decrease the convergence limit in the Python solvers (`convergence_limit_u`, ...) or switch to the direct solver.

10.1.6 Section 9

The scaling velocity for the residuals is set to one

```
uin=1
```

10.1.7 Section 10

I set Dirichlet at the inlet (west) and homogeneous at the outlet (east)

```
u_bc_west_type='d'
u_bc_east_type='n'

v_bc_west_type='d'
v_bc_east_type='n'

k_bc_west_type='d'
k_bc_east_type='n'

om_bc_west_type='d'
om_bc_east_type='n'
```

The values at the inlet are set as $\bar{u} = 1$, $\bar{v} = 0$ and $\omega = 1$

```
u_bc_west=np.ones(nj)
v_bc_west=np.zeros(nj)
om_bc_west=np.ones(nj)
```

For the turbulent kinetic energy, I set $k = 10^{-5}$ outside the boundary layer and $k = 10^{-2}$ in the ten inner cells

```
k_bc_west=np.ones(nj)*1e-2
k_bc_west[10:]=1e-5
```

The north and south boundaries are walls for which I set Dirichlet (no-slip)

```
u_bc_south_type='d'
u_bc_north_type='d'

v_bc_south_type='d'
v_bc_north_type='d'

k_bc_south_type='d'
u_bc_north_type='d'

om_bc_south_type='d'
om_bc_north_type='d'
```

The values are set to zero for $\bar{u}$, $\bar{v}$ and k, i.e.

```
u_bc_south=np.zeros(ni)
u_bc_north=np.zeros(ni)

v_bc_south=np.zeros(ni)
v_bc_north=np.zeros(ni)

k_bc_south=np.zeros(ni)
k_bc_north=np.zeros(ni)
```

For ω, I use Eq. 5.3

```
xwall_s=0.5*(x2d[0:-1,0]+x2d[1:,0])
ywall_s=0.5*(y2d[0:-1,0]+y2d[1:,0])
dist2_s=(yp2d[:,0]-ywall_s)**2+(xp2d[:,0]-xwall_s)**2
om_bc_south=10*6*viscos/0.075/dist2_s

xwall_n=0.5*(x2d[0:-1,-1]+x2d[1:,-1])
ywall_n=0.5*(y2d[0:-1,-1]+y2d[1:,-1])
dist2_n=(yp2d[:,-1]-ywall_n)**2+(xp2d[:,-1]-xwall_n)**2
om_bc_north=10*6*viscos/0.075/dist2_n
```

10.2 modify_case.py

10.2.1 modify_init

Initial condition: set $\bar{u}$, k and $\omega =$ from inlet boundary conditions..

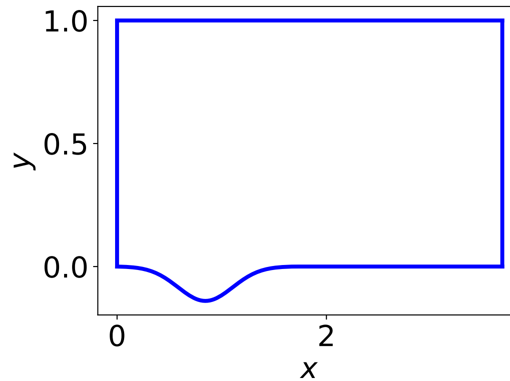


Figure 11.1: Domain of corrugated channel

```
# set inlet field in entre domain
   u3d=np.repeat(u_bc_west[None,:,:], repeats=ni, axis=0)
   k3d=np.repeat(k_bc_west[None,:,:], repeats=ni, axis=0)
   om3d=np.repeat(om_bc_west[None,:,:], repeats=ni, axis=0)

   vis3d=k3d/om3d+viscos
```

11 Channel with a corrugation

This flow was simulated with Large Eddy Simulation in [4]. The height of the corrugation is $H_1 = 0.13$ (It wrongly states $H_1 = 0.2$ in the paper), see Fig. 11.1. Periodic boundary conditions used in the x direction. The bulk flow at the last and right end of the domain is set one by adjusting the driving pressure gradient. The Reynolds number based on the bulk flow velocity and channel height is $Re_b = u_b H/\nu = 10\,000$.

11.1 `setup_case.py`

11.1.1 Section 1

Hybrid discretization is set for all variables.

```
scheme='h'  #hybrid
scheme_turb='h'
```

11.1.2 Section 2

The $k - \omega$ RANS model is selected.

```
kom = True
```

11.1.3 Section 4

The viscosity is set.

```
viscos=1/10000
```

11.1.4 Section 5

I set under-relation factor of 0.5 for all variables except for p'

```
urfvis=0.5
urf_vel=0.5
urf_k=0.5
urf_omega=0.5
urf_p=1.0
```

11.1.5 Section 6

The `lgmres` solver is chosen for the velocities, the `pyamg` for p' and `gmres` for k and ω.

```
solver_vel='lgmres'
solver_pp='pyamg'
solver_turb='gmres'
```

The convergence limit in the Python solvers is set to 10^{-8} for $\bar{u}$ and $\bar{v}$ and 10^{-6} for all k and ω and $5 \cdot 10^{-6}$ for p'.

```
convergence_limit_u=1e-8
convergence_limit_v=1e-8
convergence_limit_k=1e-6
convergence_limit_om=1e-6
convergence_limit_pp=5e-6
```

The global convergence limit is set to $5 \cdot 10^{-9}$ and the maximum number of iterations is set to 50000

```
sormax=1e-9
maxit=50000
```

11.1.6 Section 9

The scaling velocity for the residuals is set to one

```
uin=1
```

11.1.7 Section 10

I set periodic boundary conditions in the x direction

```
cycli_x=True
```

The north and south boundaries are walls for which I set Dirichlet (no-slip)

```
u_bc_south_type='d'
u_bc_north_type='d'

v_bc_south_type='d'
v_bc_north_type='d'

k_bc_south_type='d'
u_bc_north_type='d'

om_bc_south_type='d'
om_bc_north_type='d'
```

The values are set to zero for $\bar{u}$, $\bar{v}$ and k, i.e.

```
u_bc_south=np.zeros(ni)
u_bc_north=np.zeros(ni)

v_bc_south=np.zeros(ni)
v_bc_north=np.zeros(ni)

k_bc_south=np.zeros(ni)
k_bc_north=np.zeros(ni)
```

For ω, I use Eq. 5.3

```
xwall_s=0.5*(x2d[0:-1,0]+x2d[1:,0])
ywall_s=0.5*(y2d[0:-1,0]+y2d[1:,0])
dist2_s=(yp2d[:,0]-ywall_s)**2+(xp2d[:,0]-xwall_s)**2
om_bc_south=10*6*viscos/0.075/dist2_s

xwall_n=0.5*(x2d[0:-1,-1]+x2d[1:,-1])
ywall_n=0.5*(y2d[0:-1,-1]+y2d[1:,-1])
dist2_n=(yp2d[:,-1]-ywall_n)**2+(xp2d[:,-1]-xwall_n)**2
om_bc_north=10*6*viscos/0.075/dist2_n
```

11.1.8 **modify_init**

Initial condition: set $\bar{u}$, k and $\omega =$ from inlet boundary conditions..

```
u2d=np.ones((ni,nj))
k2d=0.1*np.ones((ni,nj))
L=0.1
om2d=k2d**0.5/L

vis2d=k2d/om2d+viscos
```

11.1.9 **modify_init**

Set a driving pressure gradient so that the bulk velocituy at$x = 0$ is equal to one.

```
global flowin,xpforce

 if iter == 0:
    xpforce=np.ones(maxit)*0.012
    flowin=np.ones(maxit)

 flow=np.sum(convw[0,:])

 flowin[iter]=flow
 flowin_target=np.sum(areaw[-1,:])

 if iter > 1:

   factor=1.e-5/0.0005
   xpforce[iter]=xpforce[iter-1]+factor*\
      (flowin_target+flowin[iter-1]-2.*flowin[iter])
   if abs(xpforce[iter]-xpforce[iter-1]) < 0.000001:
      xpforce[iter]=xpforce[iter-1]

 dpdx=-xpforce[iter]
 su2d=su2d-dpdx*vol
```

11.1.10 Comparison with LES data

You find LES data of the results in [4] here which can be used for evaluating the RANS datra.

A Variables in pyCALC-RANS

Nomenclature

`ae_bound`: a_E coefficient for diffusion for east boundary (without viscosity)

`an_bound`: a_N coefficient for diffusion for north boundary (without viscosity)

`areas`: south area

`areasx`: x component of south area of control volume

`areasy`: y component of south area of control volume

`areaw`: west area of control volume

`areawx`: x component of west area of control volume

`areawy`: y component of west area of control volume

`as_bound`: a_S coefficient for diffusion for south boundary (without viscosity)

`aw2d,ae2d,as2d,an2d,ap2d`: discretization coefficients, a_W, a_E, a_S, a_N, a_P

`aw_bound`: a_W coefficient for diffusion for west boundary (without viscosity)

`c_omega_1`: $C_{\omega 1}$ coefficient in the $k-\omega$ model

`c_omega_2`: $C_{\omega 2}$ coefficient in the $k-\omega$ model

`cmu`: C_μ coefficient in the $k-\varepsilon$ model, the $k-\omega$ model and C_S coefficient in the Smagorinsky model

`convergence_limit_k, convergence_limit_om`: convergence limit in Python solver for ε, k, ω

`convergence_limit_p`: convergence limit in Python solver for $\bar{p}$

`convergence_limit_u`: convergence limit in Python solver for $\bar{u}$

`convergence_limit_v`: convergence limit in Python solver for $\bar{v}$

`convw,convs`: convection through west and south

`cyclic_x`: periodic boundary conditions in th x direction

`fx,fy`: f_x, f_y, the interpolation function in i and j direction

`gen`: P^k excluding the turbulent viscosity (used in the k, ε and ω equations)

`imon,jmon`: print time history of variables for this node

`iter`: current global iteration

`k2d`: modeled turbulent kinetic energy, k

`k_bc_east, k_bc_south, k_bc_west, k_bc_north`: boundary values of k at east, south, west, north boundary

`k_bc_east_type, k_bc_north_type, k_bc_south_type, k_bc_pest_type`: type of b.c. for k ('d'=Dirichlet, 'n'=Neumann')

`kom`: the Wilcox $k-\omega$ model is used (RANS)

`maxit`: maximum number of global iterations (solving $\bar{u}$, $\bar{v}$, $\bar{w}$, $\bar{p}$, ...)

`ni,nj`: number of cell centers in i and j direction

`nsweep_kom`: maximum number of iterations in the Python solver when solving the k and ω equations in solver called in `solve_2d`

`nsweep_vel`: maximum number of iterations in the Python solver when solving the $\bar{u}$, $\bar{v}$ and w equations in solver called in `solve_2d`

`om2d`: specific dissipation of turbulent kinetic energy, ω

`om_bc_east, om_bc_north, om_bc_south, om_bc_west`: boundary values of ω at east, north, south, west boundary

`om_bc_east_type, om_bc_north_type, om_bc_south_type, om_bc_omest_type`: type of b.c. for ω

`p2d`: pressure, $\bar{p}$

`p_bc_east, p_bc_north, p_bc_south, p_bc_west` boundary values of $\bar{p}$ at east, north, south, west boundary

`p_bc_east_type, p_bc_north_type, p_bc_south_type, p_bc_pest_type:` type of b.c. for $\bar{p}$ ('d'=Dirichlet, 'n'=Neumann')

`prand_k:` σ_k, turbulent Prandtl number in the k equation

`prand_omega:` σ_ω, turbulent Prandtl number in the ω equation

`residual_p:` residual for the continuity equation

`residual_u:` residual for the $\bar{u}$ equation

`residual_v:` residual for the $\bar{v}$ equation

`resnorm_p:` the residual of the continuity equation is normalised by this quantity

`resnorm_vel:` the residuals of $\bar{u}$, $\bar{v}$ and $\bar{w}$ are normalised by this quantity

`restart:` a restart from a previous simulaton is made, see Section 6.18

`save:` the $\bar{u}$, $\bar{v}$ ... fields are saved to disk, see Section 6.19

`scheme:` discretization scheme for the $\bar{u}$, $\bar{v}$ and $\bar{w}$ equation. 'c'=central, 'h'=hybrid, 'u'=upwind, see Section 6.7

`scheme_turb:` discretization scheme for k, ε and ω. 'c'=central, 'h'=hybrid, 'u'=upwind, see Section 6.7

`solver_turb:` Python sparse matrix or pyAMG solver for k, ε and ω. solver_turb='pyamg', 'gmres', 'lgmres', 'cgs', 'cg'

`solver_vel:` Python sparse matrix or pyAMG solver for $\bar{u}$, $\bar{v}$ and $\bar{w}$. solver_vel='pyamg', 'gmres', 'lgmres', 'cgs', 'cg'

`sormax:` convergence criteria in outer iteration loop

`sp2d, su2d:` discretization source terms, S_p, S_U

`u2d:` $\bar{u}$ velocity

`u_bc_east, u_bc_north, u_bc_south, u_bc_west:` boundary values of $\bar{u}$ at east, north, south, west boundary

`u_bc_east_type, u_bc_north_type, u_bc_south_type, u_bc_uest_type:` type of b.c. for $\bar{u}$ ('d'=Dirichlet, 'n'=Neumann')

`urfvis:` under-relaxation factor for turbulent viscosity

`v2d:` $\bar{v}$ velocity

`v_bc_east, v_bc_north, v_bc_south, v_bc_west:` boundary values of $\bar{v}$ at east, north, south, west boundary

`v_bc_east_type, v_bc_north_type, v_bc_south_type, v_bc_vest_type:` type of b.c. for $\bar{v}$ ('d'=Dirichlet, 'n'=Neumann')

`vis2d`: total viscosity, $\nu + \nu_t$

`viscos`: viscosity, ν. Note that $\nu = \mu$ since $\rho = 1$.

`vol`: volume of a control volume

`vtk`: if TRUE, save results in VTK format

`x2d`: the x coordinate of a corner of a control volume, see Fig. 1.3

`xp2d`: the x coordinate of the center of a control volume, see Fig. 1.3

`y2d`: the y coordinate of a corner of a control volume, see Fig. 1.3

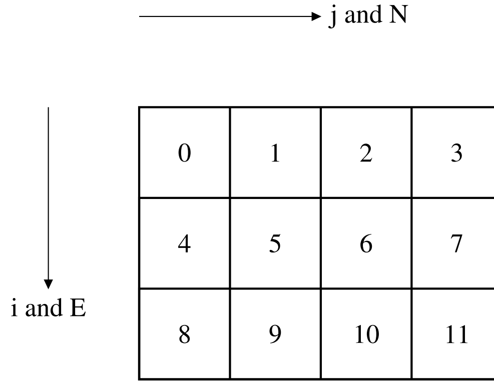
`yp2d`: the y coordinate of the center a control volume, see Fig. 1.3

B Sparse matrix format in Python

pyCALC-RANS uses the sparse solvers available in Python. The coefficients $a_W, a_E, a_S, a_N, a_P, S_u$ must be converted to Python's sparse matrix format. Hence, there are five diagonals.

The Python solvers `linalg.lgmres`, `linalg.gmres`, `linalg.cgs`, `linalg.gs`, or the algebraic multigrid solver pyAMG [5] may be used for all variables.

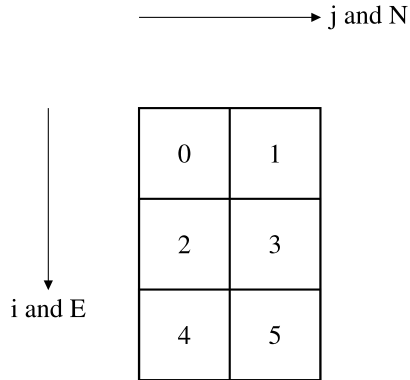
B.1 2D grid, $ni \times nj = (3, 4)$



$$
\left[
\begin{array}{rrrrrrrrrrrrr}
 & C0 & C1 & C2 & C3 & C4 & C5 & C6 & C7 & C8 & C9 & C10 & C11 \\
L0: & a_{P,0} & -a_{N,0} & 0 & 0 & -a_{E,0} & 0 & 0 & 0 & & & & \\
L1: & -a_{S,1} & a_{P,1} & -a_{N,1} & 0 & 0 & -a_{E,1} & 0 & 0 & & & & \\
L2: & 0 & -a_{S,2} & a_{P,2} & -a_{N,2} & 0 & 0 & -a_{E,2} & 0 & 0 & & & \\
L3: & 0 & 0 & -a_{S,3} & a_{P,3} & 0 & 0 & 0 & -a_{E,3} & 0 & 0 & & \\
L4: & -a_{W,4} & 0 & 0 & 0 & a_{P,4} & -a_{N,4} & 0 & 0 & -a_{E,4} & 0 & 0 & \\
L5: & 0 & -a_{W,5} & 0 & 0 & -a_{S,5} & a_{P,5} & -a_{N,5} & 0 & 0 & -a_{E,5} & 0 & 0 \\
L6: & 0 & 0 & -a_{W,6} & 0 & & -a_{S,6} & -a_{P,6} & -a_{N,6} & 0 & 0 & -a_{E,6} & 0 \\
L7: & 0 & 0 & 0 & -a_{W,7} & 0 & 0 & -a_{S,7} & -a_{P,7} & 0 & 0 & 0 & -a_{E,7} \\
L8: & 0 & 0 & 0 & 0 & -a_{W,8} & 0 & 0 & 0 & a_{P,8} & -a_{N,8} & 0 & \\
L9: & 0 & 0 & 0 & 0 & 0 & -a_{W,9} & 0 & 0 & -a_{S,9} & a_{P,9} & -a_{N,9} & 0 \\
L10: & 0 & 0 & 0 & 0 & 0 & 0 & -a_{W,10} & 0 & 0 & -a_{S,10} & a_{P,10} & -a_{N,10} \\
L11: & 0 & 0 & 0 & 0 & 0 & 0 & 0 & -a_{W,11} & 0 & 0 & -a_{S,11} & a_{P,11}
\end{array}
\right]
$$

Matrix for 2D flow. $ni \times nj = (3, 4)$.

B.2 2D grid, $ni \times nj = (3,2)$



$$
\left[\begin{array}{lrrrrrr}
 & C0 & C1 & C2 & C3 & C4 & C5 \\
L0: & a_{P,0} & -a_{N,0} & -a_{E,0} & 00 & 0 & \\
L1: & -a_{S,1} & a_{P,1} & 0 & -a_{E,1} & 0 & 0 \\
L2: & -a_{W,2} & -a_{S,2} & a_{P,2} & -a_{N,2} & -a_{E,2} & 0 \\
L3: & 0 & -a_{W,3} & -a_{S,3} & a_{P,3} & 0 & 0a_{E,3} \\
L4: & 0 & 0 & -a_{W,4} & 0 & a_{P,4} & -a_{N,4} \\
L5: & 0 & 0 & 0 & -a_{W,5} & 0 & a_{P,5}
\end{array}\right]
$$

Matrix for 2D flow. $ni \times nj = (3,2)$.

References

[1] L. Davidson. pyCALC-LES: a Python code for DNS, LES and Hybrid LES-RANS. Division of Fluid Dynamics, Dept. of Mechanics and Maritime Sciences, Chalmers University of Technology, Gothenburg, 2021.

[2] S. V. Patankar. *Numerical Heat Transfer and Fluid Flow*. McGraw-Hill, New York, 1980.

[3] D. C. Wilcox. Reassessment of the scale-determining equation. *AIAA Journal*, 26 (11):1299–1310, 1988.

[4] M. Mirzaei, L. Davidson, A. Sohankar, and F. Innings. The effect of corrugation on heat transfer and pressure drop in channel flow with different Prandtl numbers. *International Journal of Heat and Mass Transfer*, 66:164–176, 2013. doi: 10.1016/j.ijheatmasstransfer.2013.06.047.

[5] L. N. Olson and J. B. Schroder. PyAMG: Algebraic multigrid solvers in Python v4.0, 2018. URL https://github.com/pyamg/pyamg. Release 4.0.