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

November 18, 2025

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 $k-\omega$ and the $k-\varepsilon$ turbulence models. 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). Three methods for discretizing the convection terms are available, second-order central differencing, a hybrid scheme of first-order upwind and second-order central differencing and the MUSCL scheme. The discretized equations are solved with Pythons sparse matrix solvers (currently `linalg.lgmres` or `linalg.gmres` are used).

The Explicit Algebraic Reynolds Stress (EARSM) is also available in **pyCALC-RANS** . It has been improved using Neural Network [1].

pyCALC-RANS was written starting from the 3D, unsteady LES/DES code **pyCALC-LES** [2].

Contents

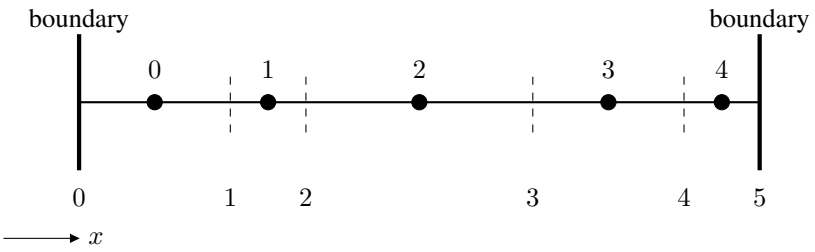


Figure 2.1: `ni=5`. The bullets denote cell centers of the control volumes where the solution vector, ϕ, is stored. They are labeled 0–4. Dashed lines denote control volume faces labeled 0–5.

1 Flow equations

The momentum equations read

$$\frac{\partial \bar{u}_j \bar{u}_i}{\partial x_j} = -\frac{1}{\rho}\frac{\partial \bar{p}}{\partial x_i} + \frac{\partial}{\partial x_j}\left[(\nu + \nu_{eff})\left(\frac{\partial \bar{u}_i}{\partial x_j} + \frac{\partial \bar{u}_j}{\partial x_i}\right)\right] - \frac{\partial(\overline{v_i' v_j'})_r}{\partial x_j} \tag{1.1}$$

When EARSM is used, ν_{eff} is the effective viscosity including the EARSM coefficient β_1, see Eq. 8.9. The last term includes $(\overline{v_i' v_j'})_r$ which is the residual stress tensor in EARSM. The total stress tensor is

$$\overline{v_i' v_j'} = -\nu_{eff}\left(\frac{\partial \bar{v}_i}{\partial x_j} + \frac{\partial \bar{v}_j}{\partial x_i}\right) + (\overline{v_i' v_j'})_r \tag{1.2}$$

When the standard $k-\omega$ is used without EARSM the residual stress vanishes and $\nu_{eff} = \nu_t$.

2 Geometrical details of the grid

2.1 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.

2.1.1 Nomenclature for the grid

Figure 2.1 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. 2.2. 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).

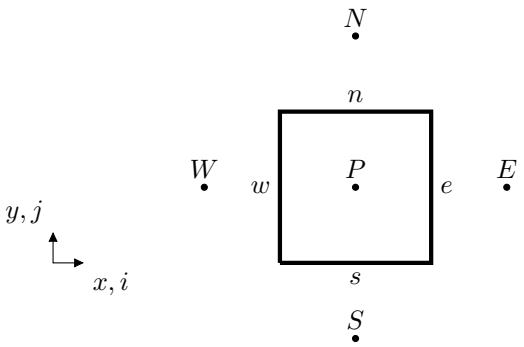


Figure 2.2: Control volume.

2.1.2 Area calculation of control volume faces

The x and y coordinates of the corners of the face in Fig. 2.3 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. 2.3 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

$$a_x = x2d(i, j+1) - x2d(i, j) \tag{2.1}$$

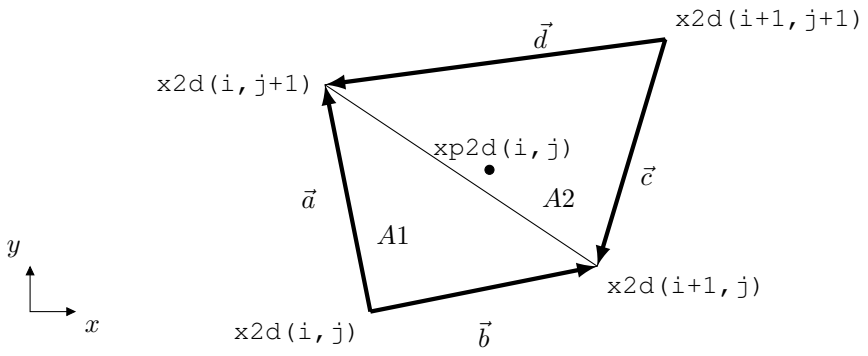


Figure 2.3: Calculation of areas of cell `i,j`.

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

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.

$$A1 = \frac{1}{2}|\vec{a} \times \vec{b}|; \qquad A2 = \frac{1}{2}|\vec{b} \times \vec{c}| \tag{2.2}$$

2.1.3 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

$$\phi_w = f_x \phi_P + (1 - f_x)\phi_W \tag{2.3}$$

where

$$f_x = \frac{|\overrightarrow{Ww}|}{|\overrightarrow{Pw}| + |\overrightarrow{Ww}|} \tag{2.4}$$

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`.

2.2 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

$$\frac{\partial \Phi}{\partial x} = \frac{1}{V}\left(\Phi_e A_{ex} - \Phi_w A_{wx} + \Phi_n A_{nx} - \Phi_s A_{sx}\right) \tag{2.5}$$

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

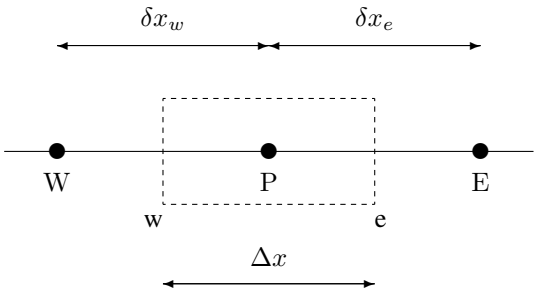


Figure 3.1: 1D control volume. Node P located in the middle of the control volume.

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`.

3 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. 3.1.

$$\int_w^e \left[\frac{d}{dx}\left(\Gamma\frac{d\phi}{dx}\right) + S\right] dx = \left(\Gamma\frac{d\phi}{dx}\right)_e - \left(\Gamma\frac{d\phi}{dx}\right)_w + \bar{S}\Delta x = 0 \tag{3.1}$$

where (see Fig. 3.1):

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.

$$\left(\frac{d\phi}{dx}\right)_e \simeq \frac{\phi_E - \phi_P}{\delta x_e}, \quad \left(\frac{d\phi}{dx}\right)_w \simeq \frac{\phi_P - \phi_W}{\delta x_w}. \tag{3.2}$$

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

$$\Gamma_w = f_x\Gamma_P + (1 - f_x)\Gamma_W, \tag{3.3}$$

Insertion of Eq. 3.2 into Eq. 3.1 gives

$$\begin{aligned} a_P\phi_P &= a_E\phi_E + a_W\phi_W + S_U \\ a_E &= \frac{\Gamma_e}{\delta x_e} \\ a_W &= \frac{\Gamma_w}{\delta x_w} \\ S_U &= \bar{S}\Delta x \\ a_P &= a_E + a_W \end{aligned} \tag{3.4}$$

3.1 Convergence criteria

Compute the residual for Eq. 3.4

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

In Python it corresponds to $|Ax - b|$. Since we want Eq. 3.4 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.

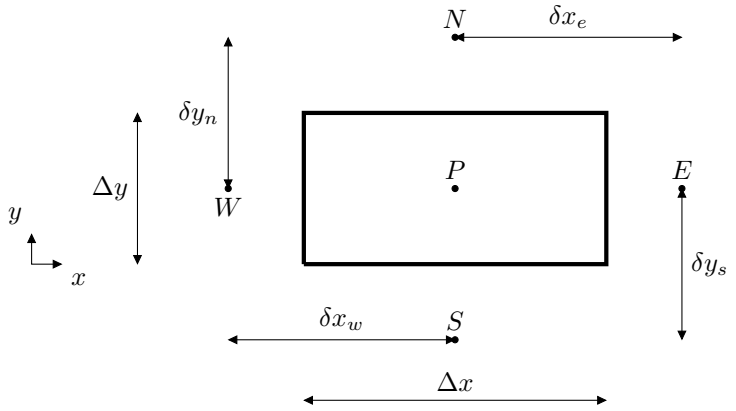


Figure 3.2: 2D control volume.

3.2 2D Diffusion

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

$$\frac{\partial}{\partial x}\left(\Gamma\frac{\partial\phi}{\partial x}\right)+\frac{\partial}{\partial y}\left(\Gamma\frac{\partial\phi}{\partial y}\right)+S=0. \tag{3.5}$$

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. 3.2, 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.

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

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

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

Doing the same for the diffusion term in the y direction in Eq. 3.5 gives

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

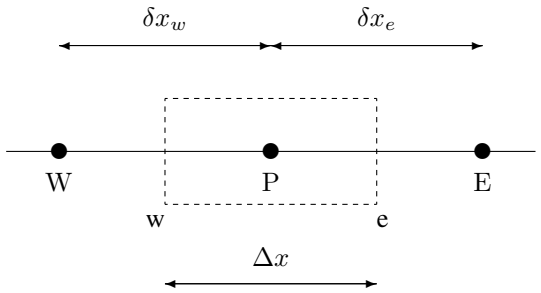


Figure 4.1: 1D control volume. Node P located in the middle of the control volume.

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

Rewriting it as an algebraic equation for ϕ_P, we get

$$\begin{aligned} a_P \phi_P &= a_E \phi_E + a_W \phi_W + a_N \phi_N + a_S \phi_S + S_U \\ a_E &= \frac{\Gamma_e \Delta y}{\delta x_e},\ a_W = \frac{\Gamma_w \Delta y}{\delta x_w},\ a_N = \frac{\Gamma_n \Delta x}{\delta y_n},\ a_S = \frac{\Gamma_s \Delta x}{\delta y_s} \\ S_U &= \bar{S}\Delta x \Delta y,\ a_P = a_E + a_W + a_N + a_S - S_P. \end{aligned} \tag{3.6}$$

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. 3.4).

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

4 Convection – diffusion

The 1D convection-diffusion equation reads

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

We discretize this equation in the same way as the diffusion equation. We start by integrating over the control volume (see Fig. 4.1).

$$\int_w^e \frac{d}{dx}(\bar{u}\phi)\, dx = \int_w^e \left[\frac{d}{dx}\left(\Gamma \frac{d\phi}{dx}\right) + S\right] dx. \tag{4.1}$$

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

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

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

4.1 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

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

where the convecting part, $\bar{u}$, is taken by central differencing, and the convected part, ϕ, is estimated with different differencing schemes. We start by using central differencing for ϕ so that

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

where f_x is the interpolation function (see Eq. 3.3, 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. 4.1 we obtain

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

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.

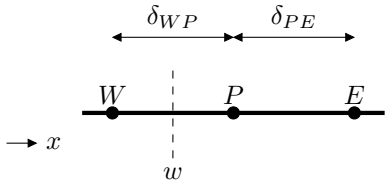


Figure 4.2: Constant mesh spacing. $\bar{u} > 0$.

4.2 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.

4.3 Hybrid scheme

modules: `coeff`

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.

4.4 MUSCL

modules: `coeff_muscl`

MUSCL [3] is a second-order upwind bounded scheme. It uses two nodes upstream and one downstream, see Fig. 4.2. The coefficients $a_W, a_E, \ldots a_H$ are computed with first-order upwind. A correction term is then added to the source term. For the west face, e.g, it reads

$$\begin{aligned} C^+ &= 0.5 + 0.5 \cdot \mathrm{sign(C_w)} \\ C^- &= 0.5 \cdot \mathrm{sign(C_w)} - 0.5 \\ S_U^M &= 0.5 \cdot C_w \cdot (C^+ \cdot \mathrm{minmo(u_P - u_W, u_W - u_{WW})} \\ &- C^- \cdot \mathrm{minmo(u_P - u_W, u_E - u_P)}) \end{aligned} \tag{4.2}$$

where C_w is the convective flux through the west face. The function `minmo` is defined as

$$\mathrm{minmo(a, b)} = \mathrm{sign(a)} \cdot \max(0, \min(|\mathrm{a}|, \mathrm{b} \cdot \mathrm{sign(a)}))$$

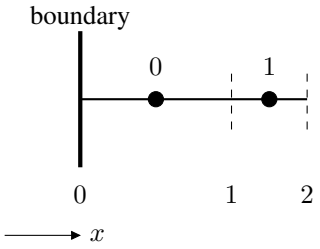


Figure 4.3: 1D grid. Boundary conditions at $x = 0$.

4.5 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. 4.3. Consider only convection. For the $\bar{u}$ equation at cell $i = 0$ we get

$$\begin{aligned} a_P \bar{u}_P &= a_W \bar{u}_W + a_E \bar{u}_E + S_U \\ a_P &= a_W + a_E - S_P, \quad a_W = C_w, \quad a_E = -0.5 C_e \\ C_w &= \bar{u}_W A_w \\ a_P &= C_w - 0.5 C_e \end{aligned} \tag{4.3}$$

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. 4.3 has to be implemented with additional source terms

$$\begin{aligned} a_w &= 0 \\ S^u_{U,add} &= C_w \bar{u}_{in} \\ S^u_{P,add} &= -C_w \end{aligned} \tag{4.4}$$

For $\bar{v}$ it reads

$$\begin{aligned} S^v_{U,add} &= C_w \bar{v}_{in} \\ S^v_{P,add} &= -C_w \end{aligned} \tag{4.5}$$

$$\tag{4.6}$$

The additional term for the diffusion reads

$$\begin{aligned} S^u_{U,add,diff} &= \frac{\nu_{tot} A_w}{\Delta x} \bar{u}_{in} \\ S^v_{U,add,diff} &= \frac{\nu_{tot} A_w}{\Delta x} \bar{v}_{in} \\ S_{P,add,diff} &= -\frac{\nu_{tot} A_w}{\Delta x} \end{aligned} \tag{4.7}$$

where $S_{P,add,diff}$ is the same for $\bar{u}$ and $\bar{v}$. The viscous part of Eq. 4.7 is implemented in module `bc`. The turbulent part and the convective part (Eqs. 4.4 and 4.5) are implemented in `module_u`, `module_v` etc.

4.6 Wall boundary conditions using source term

We use exactly the same procedure as in Section 4.5. At walls, there is no convection and the velocity is zero. Hence we simply use Eq. 4.7 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`.

4.7 Pressure correction equation

The pressure correction equation is obtained by applying the SIMPLEC algorithm [4] 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

$$\dot{m}_e = \dot{m}_e^* + \dot{m}_e', \quad \dot{m}_e' = \left(\vec{A} \cdot \vec{u}'\right)_e = (A_{ex} u_e' + A_{ey} v_e') \tag{4.8}$$

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

$$u' = -\frac{\Delta V}{a_P}\frac{\partial p'}{\partial x}, \quad v' = -\frac{\Delta V}{a_P}\frac{\partial p'}{\partial y}, \tag{4.9}$$

where ΔV_P denotes the volume of the control volume. By introducing Eq. 4.8 into Eq. 4.9 we obtain

$$\dot{m}' = -\left[\frac{\Delta V_P}{a_P}\vec{A} \cdot \nabla p'\right] = -\frac{\Delta V_P}{a_P}\left[\vec{A}_x \frac{\partial p'}{\partial x} + \vec{A}_y \frac{\partial p'}{\partial y}\right] \tag{4.10}$$

Consider, for simplicity, the continuity equation in one dimension

$$\dot{m}_e - \dot{m}_w = 0 \tag{4.11}$$

If $\dot{m} = \dot{m}^* + \dot{m}'$ and Eq. 4.10 are substituted into eq. 4.11 we obtain

$$\left[\frac{\Delta V_P A_x}{a_P}\frac{\partial p'}{\partial x}\right]_w - \left[\frac{\Delta V_P A_x}{a_P}\frac{\partial p'}{\partial x}\right]_e + \dot{m}_e^* - \dot{m}_w^* = 0 \tag{4.12}$$

This is a diffusion equation for the pressure correction p' which is discretized as Eq. 3.6 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

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.

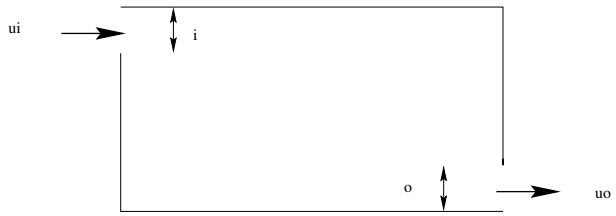


Figure 5.1: Outlet boundary condition. Small outlet

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. 4.8) $\dot{m}_e^*$ 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).

5 Boundary Conditions

5.1 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. 5.1)

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

5.2 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. 5.2), 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. 5.2)

1. Set $\bar{u}_e = \bar{u}_w$ for all nodes (i.e. for $j = 0$ to 4, see Fig. 5.2);
2. In order to speed up convergence, enforce global continuity.
   - Inlet volume flow: $\dot{V}_{in} = \sum_{inlet} \bar{u}_{in} \Delta y$

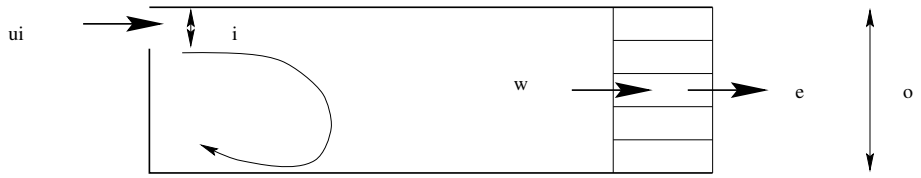


Figure 5.2: Outlet boundary condition. Large outlet.

- 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`.

5.3 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'`.

6 The $k - \omega$ model

modules: `calck_kom, calcom, vist_kom`

The Wilcox $k - \omega$ turbulence model reads [5]

$$\begin{aligned} \frac{\partial \bar{v}_j k}{\partial x_j} &= P^k + \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_j}\right] - C_\mu k\omega \\ \frac{\partial \bar{v}_j \omega}{\partial x_j} &= C_{\omega_1}\frac{\omega}{k}P^k + \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_\omega}\right)\frac{\partial \omega}{\partial x_j}\right] - C_{\omega 2}\omega^2 \\ \nu_t &= \frac{k}{\omega} \end{aligned} \tag{6.1}$$

The standard coefficients are used, i.e. $C_{\omega 1} = 5/9$, $C_{\omega 2} = 3/40$, $\sigma_k = \sigma_\omega = 2$ and $C_\mu = 0.09$. When EARSM is used, the production term is computed as

$$P^k = -\overline{v_i' v_j'}\frac{\partial \bar{v}_i}{\partial x_j} \tag{6.2}$$

and the dissipation in the EARSM reads

$$\varepsilon = C_\mu k\omega \tag{6.3}$$

In the $k - \omega$ model (without EARSM), the production term is computed as

$$P^k = \nu_t\left(\frac{\partial \bar{v}_i}{\partial x_j} + \frac{\partial \bar{v}_j}{\partial x_i}\right)\frac{\partial \bar{v}_i}{\partial x_j} \tag{6.4}$$

The wall boundary conditions are

$$k_w = 0, \quad \omega_w = 10\frac{6\nu}{C_{\omega 2} y^2} \tag{6.5}$$

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

$$\omega_P = \frac{6\nu}{C_{\omega 2} y^2} \tag{6.6}$$

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

7 The $k-\varepsilon$ model

modules: `calc_keps, calcepd, vist_keps`

The AKN low-Reynolds number reads [6]

$$\begin{aligned}
\frac{\partial k}{\partial t} + \frac{\partial \bar{v}_j k}{\partial x_j} &= \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_j}\right] + P_k - \varepsilon \\
\frac{\partial \varepsilon}{\partial t} + \frac{\partial \bar{v}_j \varepsilon}{\partial x_j} &= \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_\varepsilon}\right)\frac{\partial \varepsilon}{\partial x_j}\right] + C_{\varepsilon 1} P_k \frac{\varepsilon}{k} - C_{\varepsilon 2} f_2 \frac{\varepsilon^2}{k} \\
\nu_t &= C_\mu f_\mu \frac{k^2}{\varepsilon}, \quad P_k = \nu_t \left(\frac{\partial \bar{v}_i}{\partial x_j} + \frac{\partial \bar{v}_j}{\partial x_i}\right)\frac{\partial \bar{v}_i}{\partial x_j} \\
C_{\varepsilon 1} &= 1.5, \quad C_{\varepsilon 2} = 1.9, \quad C_\mu = 0.09, \quad \sigma_k = 1.4, \quad \sigma_\varepsilon = 1.4
\end{aligned} \tag{7.1}$$

where k and ε denote the modeled turbulent kinetic energy and its dissipation, respectively. The damping functions are defined as

$$\begin{aligned}
f_2 &= \left[1 - \exp\left(-\frac{y^*}{3.1}\right)\right]^2 \left\{1 - 0.3 \exp\left[-\left(\frac{R_t}{6.5}\right)^2\right]\right\} \\
f_\mu &= \left[1 - \exp\left(-\frac{y^*}{14}\right)\right]^2 \left\{1 + \frac{5}{R_t^{3/4}} \exp\left[-\left(\frac{R_t}{200}\right)^2\right]\right\} \\
y^* &= \frac{u_\varepsilon d}{\nu}, \quad u_\varepsilon = (\varepsilon\nu))^{1/4}, \quad Re_t = \frac{k^2}{\nu\varepsilon}
\end{aligned}$$

where d denotes the distance to the wall. The wall boundary condition is implemented by setting ε at the wall-adjacent cells as

$$\varepsilon = 2\nu \frac{k}{d^2} \tag{7.2}$$

8 The EARSM

modules: `calc_earsm`

The Algebraic Stress Model (ASM) [7] with the LRR pressure-strain model [8] reads [9]

$$\begin{aligned}
\left(c_1 - 1 + P^k/\varepsilon\right) a_{ij} &= -\frac{8}{15}\bar{s}_{ij} + \frac{7c_2 + 1}{11}(a_{ik}\bar{\Omega}_{kj} - \bar{\Omega}_{ik}a_{kj}) \\
&- \frac{5 - 9c_2}{11}\left(a_{ik}\bar{s}_{kj} + \bar{s}_{ik}a_{kj} - \frac{2}{3}a_{mn}\bar{s}_{nm}\delta_{ij}\right) \\
a_{ij} = \frac{\overline{v_i' v_j'}}{k} - \frac{2}{3}\delta_{ij}, \quad \bar{s}_{ij} &= \frac{1}{2}\left(\frac{\partial \bar{v}_i}{\partial x_j} + \frac{\partial \bar{v}_j}{\partial x_i}\right), \quad \bar{\Omega}_{ij} = \frac{1}{2}\left(\frac{\partial \bar{v}_i}{\partial x_j} - \frac{\partial \bar{v}_j}{\partial x_i}\right)
\end{aligned} \tag{8.1}$$

Note that the last term in Eq. 8.1 is zero if c_2 is set to $5/9$ [10]. Equation 8.1 can be written as [11]

$$\begin{aligned} Na_{ij} &= -A_1\bar{s}^*_{ij} + (a_{ik}\bar{\Omega}^*_{kj} - \bar{\Omega}^*_{ik}a_{kj}) - A_2\left(\bar{s}^*_{ik}a_{kj} + a_{ik}\bar{s}^*_{kj} - \frac{2}{3}\delta_{ij}\bar{s}^*_{mn}a_{nm}\right) \\ \bar{s}^*_{ij} &= \frac{k}{\varepsilon}\bar{s}_{ij}, \quad \bar{\Omega}^*_{ij} = \frac{k}{\varepsilon}\bar{\Omega}_{ij} \end{aligned} \tag{8.2}$$

where

$$A_1 = 1.54, \quad A_2 = 0.37, \quad A_3 = 1.45, \quad A_4 = 2.89 \tag{8.3}$$

In order to get an explicit form of Eq. 8.2, Girimaji [12, 13] and Wallin & Johansson [11, 14], formulated a_{ij} in terms of the strain-rate tensor ($\bar{s}_{ij}$) and the vorticity tensor ($\bar{\Omega}_{ij}$). In 2D, it reads [15]

$$a_{ij} = \beta_1\bar{s}^*_{ij} + \beta_2\left(\bar{s}^*_{ik}\bar{s}^*_{kj} - \frac{1}{3}\bar{s}^*_{mn}\bar{s}^*_{nm}\delta_{ij}\right) + \beta_4(\bar{s}^*_{ik}\bar{\Omega}^*_{kj} - \bar{\Omega}^*_{ik}\bar{s}^*_{kj}) \tag{8.4}$$

By inserting Eq. 8.4 in Eq. 8.2, Girimaji [12, 13] and Wallin & Johansson [11] derived an explicit form which in 2D reads [11] (a detailed derivation is given in [9])

$$\beta_1 = -\frac{A_1 N}{Q}, \quad \beta_2 = 2\frac{A_1 A_2}{Q}, \quad \beta_4 = -\frac{A_1}{Q}, \quad Q = N^2 - 2II_\Omega - \frac{2}{3}A_2^2 II_S \tag{8.5}$$

where N is given by the cubic equation

$$\begin{aligned} N^3 &- A_3N^2 - \left(\left(A_1A_4 + \frac{2}{3}A_2^2\right)II_S + 2II_\Omega\right)N + 2A_3\left(\frac{1}{3}A_2^2 II_S + II_\Omega\right) = 0 \\ II_S &= \bar{s}^*_{mn}\bar{s}^*_{nm}, \quad II_\Omega = \bar{\Omega}^*_{mn}\bar{\Omega}^*_{nm}. \end{aligned} \tag{8.6}$$

Equation 8.6 can be solved analytically. The analytical solution for the positive root reads [11]

$$N = \begin{cases} \frac{A_3}{3} + \left(P_1 + \sqrt{P_2}\right)^{1/3} + \text{sign}\left(P_1 - \sqrt{P_2}\right)\left|P_1 - \sqrt{P_2}\right|^{1/3}, P_2 \geq 0 \\ \frac{A_3}{3} + 2\left(P_1^2 - P_2\right)^{1/6}\cos\left[\frac{1}{3}\arccos\left(\frac{P_1}{\sqrt{P_1^2 - P_2}}\right)\right], P_2 < 0 \end{cases} \tag{8.7}$$

where

$$\begin{aligned} P_1 &= \left(\frac{A_3}{27} + \left(\frac{A_1A_4}{6} - \frac{2}{9}A_2^2\right)II_S - \frac{2}{3}II_\Omega\right)A_3 \\ P_2 &= P_1^2 - \left(\frac{A_3}{9} + \left(\frac{A_1A_4}{3} + \frac{2}{9}A_2^2\right)II_S + \frac{2}{3}II_\Omega\right)^3 \end{aligned}$$

The Reynolds stress tensor including only the first term in Eq. 8.4 reads (see Eq. 8.1)

$$\overline{v_i'v_j'} = \beta_1 k\bar{s}^*_{ij} + \frac{2}{3}k\delta_{ij} = \beta_1\frac{k^2}{\varepsilon}\bar{s}_{ij} + \frac{2}{3}k\delta_{ij} = -\nu_{eff}\bar{s}_{ij} + \frac{2}{3}k\delta_{ij} \tag{8.8}$$

where

$$\nu_{eff} = -0.5\beta_1\frac{k^2}{\varepsilon} \tag{8.9}$$

is the effective viscosity and $\varepsilon = C_\mu k\omega$, see Eq. 6.3. Equation 8.8 corresponds to the Boussinesq assumption with $\beta_1 = -2C_\mu$. The discretized momentum equation on matrix form reads

$$AW = b \tag{8.10}$$

where W is $\bar{v}_1$ or $\bar{v}_2$. The term including the effective viscosity, ν_{eff}, in Eq. 8.8 is included in A which greatly improves the numerical stability of the CFD code.

9 The Neural Network (NN) model

Here I present how to use NN to improve the EARSM. More detail can be found in [1] (see also README in the main folder)).

Instead of computing β_1, β_2 and β_4 from Eqs. 8.5 and 8.6, I will in the present work make them functions of some input parameter(s) (to be determined) using Neural Network (NN). The process can be depicted as:

1. Choose input parameter(s) involving e.g. the velocity gradient, the shear stress, the dissipation, the wall distance which should all be non-dimensional.
2. The output (target) parameters are β_1, β_2, β_4.
3. Train the NN model in fully-developed channel flow.
4. Use the NN model to compute β_1, β_2, β_4 in the EARSM (k and ω predicted with the $k-\omega$ model) in the **pyCALC-RANS** CFD code.

In fully-developed channel flow the EARSM (Eq. 8.4) reads:

$$\begin{aligned} a_{11} &= \frac{1}{12}\left(\frac{\partial \bar{v}_1^*}{\partial y}\right)^2(\beta_2 - 6\beta_4), \quad a_{22} = \frac{1}{12}\left(\frac{\partial \bar{v}_1^*}{\partial y}\right)^2(\beta_2 + 6\beta_4) \\ a_{33} &= -\frac{2\beta_2}{12}\left(\frac{\partial \bar{v}_1^*}{\partial y}\right)^2, \quad a_{12} = \frac{\beta_1}{2}\frac{\partial \bar{v}_1^*}{\partial y}, \quad \frac{\partial \bar{v}_1^*}{\partial y} = \frac{k}{\varepsilon}\frac{\partial \bar{v}_1}{\partial x_2} \end{aligned} \tag{9.1}$$

From the relations above, I get the targets for the NN model

$$\beta_1 = \frac{2a_{12}}{\frac{\partial \bar{v}_1^*}{\partial y}}, \quad \beta_2 = \frac{6(a_{11}+a_{22})}{\left(\frac{\partial \bar{v}_1^*}{\partial y}\right)^2}, \quad \beta_4 = \frac{a_{22}-a_{11}}{\left(\frac{\partial \bar{v}_1^*}{\partial y}\right)^2} \tag{9.2}$$

a_{ij}, $k/\varepsilon \equiv (\omega C_\mu)^{-1}$ and $\frac{\partial \bar{v}_1}{\partial x_2}$ are computed from DNS data of channel flow. Note that a_{33} is defined by $a_{ii} = 0$.

The output parameters of the NN model are β_1, β_2 and β_4. What input parameters should be used? The NN model should be applicable at difference Reynolds numbers so it should be a good idea to choose input parameters which also are Reynolds number independent. The NN model will be used in the CFD code and validated against DNS data in channel flow at $Re_\tau = 2\,000$ [16], $Re_\tau = 5\,200$ [17] $Re_\tau = 10\,000$ [18] and flat-plate boundary-layer flow $Re_\tau = 5\,500$ [19].

I choose the production term together with y^+ as input parameters. I scale the two input parameters using `MinMaxScaler()` so that they are in the range $[0, 1]$

I use the NN in Python's `pytorch`. Figure 9.1 shows the NN model schematically. The optimizer is set as

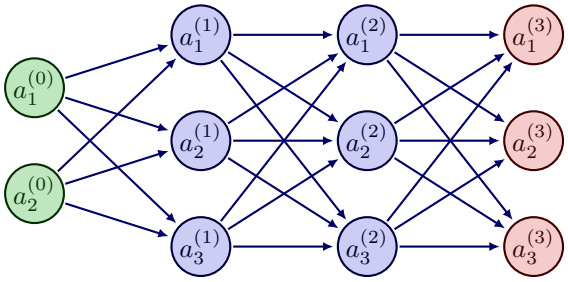


Figure 9.1: The Neural Network with two inputs variables, $a_1^{(0)} = y^+$ and $a_2^{(0)} = P^+$ and three output variables, $a_1^{(3)} = \beta_1$, $a_2^{(3)} = \beta_2$ and $a_3^{(3)} = \beta_4$. There are three neurons and two hidden layers in this figure; in the simulations I use 50 neurons.

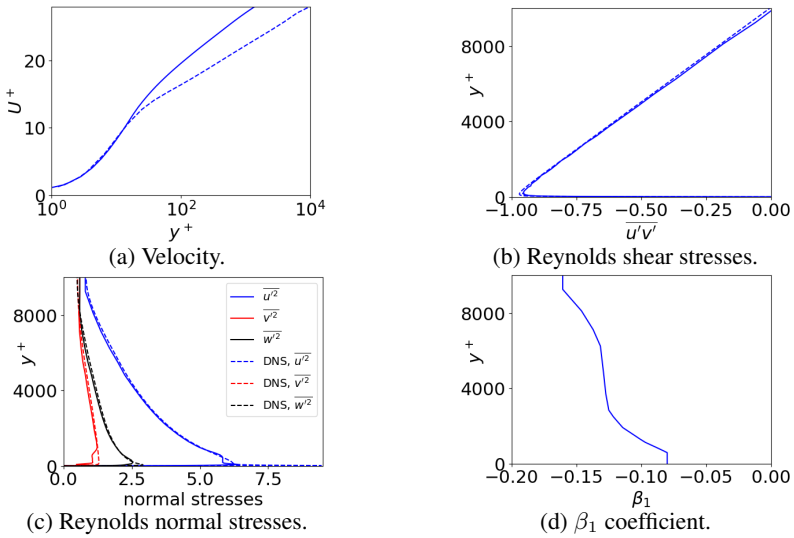
```
optimizer = torch.optim.SGD(neural_net.parameters(), lr=l_rate)
```

Since I use y^+ as input variable, I must train the NN model at the largest Reynolds number (which has the largest y^+ value), i.e. channel flow at $Re_\tau = 10\,000$. I exclude data in the viscous sublayer ($y^+ \leq 5$) because the gradient of β_2 and β_4 (see Eq. 9.2) are very large near the wall. I also exclude data near the center ($y^+ > 9\,800$) where $\frac{\partial \bar{v}_1^*}{\partial y}$ is very small. The turbulence is negligible in both these regions. I train on 80% of the data (approximately 800 randomly chosen data points) and test on (predict) the remaining 20%.

9.1 The NN model incorporated in the CFD solver

I save the NN model developed in Section 9 to disk and then a load it into **pyCALC-RANS** . I include the NN model as follows:

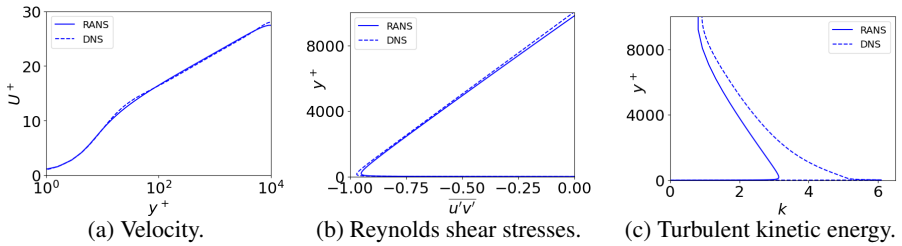
1. Load the NN model in module `calc_earsm`.
2. Solve $\bar{v}_1$, $\bar{v}_2$ and P' equations. The Reynolds stresses $\overline{v_1'^2}$, $\overline{v_2'^2}$, $\overline{v_1' v_2'}$ in the $\bar{v}_1$ and $\bar{v}_2$ equations (see Eq. 1.1) are taken from the previous iteration.
3. Compute β_1, β_2, β_4 using the NN model. Limits are set on both input and output parameters corresponding to min and max values during the training process.
4. Compute the anisotropic Reynolds stresses (a_{11}, a_{22}, a_{12}) using the β coefficients, see Eq. 9.1.
5. Compute the Reynolds stresses $\overline{v_1'^2} = ka_{11} + \frac{2}{3}k$, $\overline{v_2'^2} = ka_{22} + \frac{2}{3}k$, $\overline{v_1' v_2'} = ka_{12}$.
6. Solve the k and ω equations. The Reynolds stresses are used in the production term, see Eq. 6.2. In fully-developed channel flow $\overline{v_1'^2}$ and $\overline{v_2'^2}$ have no effect since $\partial \bar{v}_1 / \partial x_1 = \partial \bar{v}_2 / \partial x_2 = 0$, but in flat-plate boundary layer flow they have a small effect.
7. End of iteration. Repeat from Item 2 until convergence (1000s of iterations).



(a) Velocity. (b) Reynolds shear stresses.

(c) Reynolds normal stresses. (d) β_1 coefficient.

Figure 9.2: The NN model incorporated in the CFD code. $Re_\tau = 10\,000$. NN model trained on DNS data. Dashed lines: DNS data [18]



(a) Velocity. (b) Reynolds shear stresses. (c) Turbulent kinetic energy.

Figure 9.3: Channel flow at $Re_\tau = 10\,000$ using Wilcox $k-\omega$ model. DNS data [18]

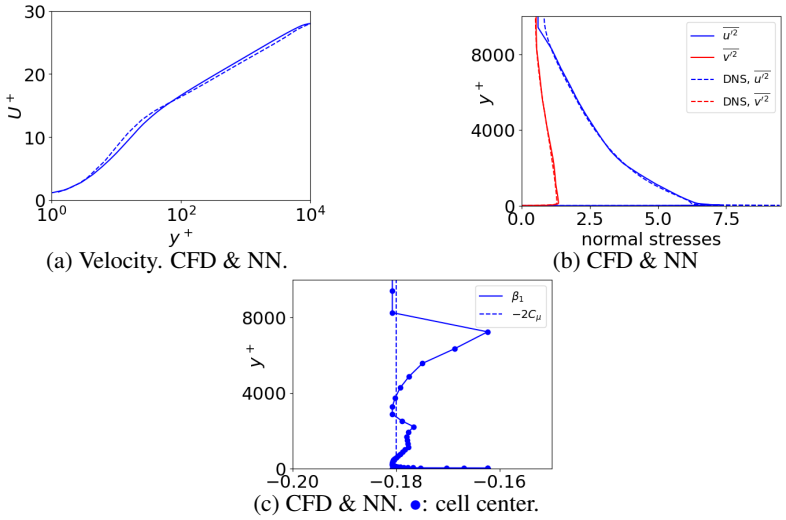
9.2 Channel flow with the NN model trained on $k-\omega$ and DNS data

It was found in [1] than when the NN model is trained using DNS data I get poor results, see Fig. 9.2. The reason is that the stress-strain relation and the turbulent kinetic energy are not the same in DNS and $k-\omega$ predictions. Hence the target data are taken both from DNS ($\overline{v_1'^2}$ and $\overline{v_2'^2}$) and a $k-\omega$ simulation ($\frac{\partial \bar{v}_1}{\partial x_2}$, $\overline{v_1' v_2'}$, k, $\varepsilon = C_\mu k\omega$). In this way the strain-stress relation and the turbulent kinetic energy are the same in the training process as in the CFD-NN predictions.

Instead of training the NN model on DNS data, I will train on data taken both from DNS and the $k-\omega$ simulation shown in Fig. 9.3. The fact that this may be necessary was noted in [20]. I will use the following data:

- Input: P^k and y^+ from $k-\omega$ prediction
- Target: β_1, β_2 and β_4 computed from $\underbrace{\overline{v_1'^2}, \overline{v_2'^2}}_{DNS}$ and $\underbrace{\overline{v_1' v_2'}, k, \varepsilon}_{k-\omega}$

This will ensure that the relation between the shear stress and the velocity gradient as well as the turbulent kinetic energy are the same in the training process as in the CFD



(a) Velocity. CFD & NN.

(b) CFD & NN

(c) CFD & NN. •: cell center.

Figure 9.4: CFD predictions with EARSM-NN model. The NN model is trained on DNS and $k-\omega$ data. Channel flow, $Re_\tau = 10\,000$. Dashed lines: DNS.

simulation. Note that

$$k_{DNS} = 0.5\left(\overline{v_1'^2}_{DNS} + \overline{v_2'^2}_{DNS} + \overline{v_3'^2}_{DNS}\right)$$

is not equal to the turbulent kinetic energy, $k_{k-\omega}$, predicted by the $k-\omega$ model. The spanwise normal stress, $\overline{v_3'^2}$, predicted by the NN model will adapt in order to satisfy

$$k_{k-\omega} = 0.5\left(\overline{v_1'^2}_{DNS} + \overline{v_2'^2}_{DNS} + \overline{v_3'^2}_{DNS}\right) \tag{9.3}$$

Hence, $\overline{v_3'^2}$ will be incorrectly predicted by the NN model (it even goes negative near the wall). Thus, the proposed EARSM-NN model is applicable only in two-dimension flows. In order to make the model applicable in three dimension, a new $k-\omega$ (or $k-\varepsilon$) model must be developed which satisfies $k_{DNS} = k_{k-\omega}$.

In [1] I show that the EARSM-NN gives good results also for channel flow at $Re_\tau = 2\,000$ and $Re_\tau = 2\,000$ as well as flat-plate boundary layer. Description of the Python script for creating the NN model as well as `setup_case` and `modify_case` are found in Section 17.

In Section 16 I present the Python script for creating the EARSM-NN model and in Section 17 I show how to make channel flow simulations using the EARSM-NN model.

10 Modules

10.1 bc_outlet_bc

Neumann outlet boundary conditions are set.

10.2 calc_earsm

The EARSM (see Section 8) is implemented in this module. Note that the module is empty in the main script, `pyCALC-RANS.py`, and that `calc_earsm.py` resides in the relevant subdirectories.

10.3 calck

Source terms in the k equation (Wilcox model) are computed, see Section 6. The user can define additional source terms in `modify_k`.

10.4 calcom

Source terms in the ω equation (Wilcox model) are computed, see Section 6. The user can define additional source terms in `modify_om`.

10.5 calcp

Coefficients in the p' equation, see Section 4.7.

10.6 calcu

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

10.7 calcv

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

10.8 coeff

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

10.9 compute_face_phi

Compute the face values of a variable.

10.10 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`. Note that they are defined as the volume flow going into the control volume.

10.11 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.

10.12 fix_omega

This routine may be used for fix ω in the wall-adjacent *cell center* according to Eq. 6.6 rather than as a wall-boundary condition (Eq. 6.5). 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
```

10.13 dphidx, dphidy

The derivative in x or y direction are computed, see Section 2.2.

10.14 init

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

10.15 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.

10.16 modify_case.py

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

10.17 modify_init

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

10.18 print_indata

Prints the indata set by the user.

10.19 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.

10.20 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`.

10.21 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`.

10.22 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)

10.23 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 a sparse matrix Python solver, e.g. `linalg.lgmres` or `linalg.gmres`.

10.24 vist_kom

The turbulent viscosity is computed using the $k-\omega$ model, see Section 6

11 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 look at the **pyCALC-RANS** flowchart.

The lid-driven cavity is shown in Fig. 11.1 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 case is defined in modules `setup_case` and `modify_case`. They are located in a directory with the name `lid`. Enter this directory.

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

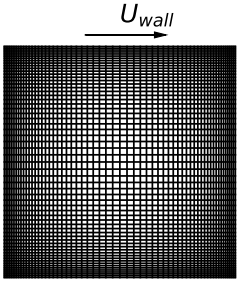


Figure 11.1: Lid-driven cavity with grid. Top wall is moving.

```
np.savetxt('x2d.dat', x2d)
```

11.1 setup_case.py

This module consists of 10 sections.

11.1.1 Section 1

I choose the hybrid scheme for convection

```
scheme='h'
```

11.1.2 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
```

11.1.3 Section 4

The viscosity is set.

```
viscos=1/1000
```

11.1.4 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 relative convergence limit in the Python solvers is defined as

$$|Ax - b|/|b| < \gamma \tag{11.1}$$

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

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

11.1.5 Section 7

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

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

11.1.6 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
```

11.1.7 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]
```

11.1.8 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))
```

11.2 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. 6.6. 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.

11.2.1 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.

11.3 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`.

You find some information also in the README file in the main folder.

12 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 11 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`.

12.1 `setup_case.py`

12.1.1 Section 1

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

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

12.1.2 Section 2

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

```
kom = True
```

12.1.3 Section 3

I don't start from a previous solution.

```
restart = False
```

12.1.4 Section 4

The viscosity is set.

```
viscos=1/5200
```

12.1.5 Section 8

The direct solver is chosen for all variables

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

12.1.6 Section 9

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

```
uin=20
```

12.1.7 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. 6.6

```
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
```

12.2 `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 13.

13 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 12.

I generate a new grid. I use the same Python script as in Section 12 but I set 30 cells, `ni=30` in the x direction and `xmax=15`.

13.1 `setup_case.py`

13.1.1 Section 6

The `lgmres` solver is chosen

```
solver_vel='lgmres'
solver_pp='lgmres'
solver_turb='lgmres'

nsweep_vel=50
nsweep_pp=50
nsweep_turb=50
convergence_limit_u=1e-6
convergence_limit_v=1e-6
convergence_limit_k=-1e-6
convergence_limit_om=-1e-6
convergence_limit_pp=5e-4
```

Note that absolute convergence criteria (i.e they are set negative) are used for k and ω, i.e.

$$|Ax - b| < \gamma \tag{13.1}$$

The reason is that it has been found difficult to set relative convergence criteria (especially for ω), possibly because of the large source terms which are used for defining the relative convergence criteria, see Eq. 11.1.

13.1.2 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 12. For ω, I use Eq. 6.6

```
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 12 set ω *at* the wall.

13.2 modify_case.py

13.2.1 modify_init

Here I set initial b.c. I load the data from the results in Section 12. 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 12. 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)
```

13.2.2 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
```

13.2.3 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`.

13.2.4 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`.

13.2.5 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`.

13.2.6 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`.

13.2.7 modify_outlet

Outlet b.c. are set according to Section 5.2

```
# 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
```

13.2.8 fix_omega

Here I set ω at the first interior cell according to Eq. 6.6. 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`.

14 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`.

14.1 setup_case.py

14.1.1 Section 1

Hybrid discretization is set for all variables.

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

14.1.2 Section 2

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

```
kom = True
```

14.1.3 Section 4

The viscosity is set.

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

14.1.4 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
```

14.1.5 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-6
convergence_limit_pp=5e-2
```

Absolute convergence level is used for k and ω , see Eq. 13.1.

14.1.6 Section 9

The scaling velocity for the residuals is set to one

```
uin=1
```

14.1.7 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. 6.5

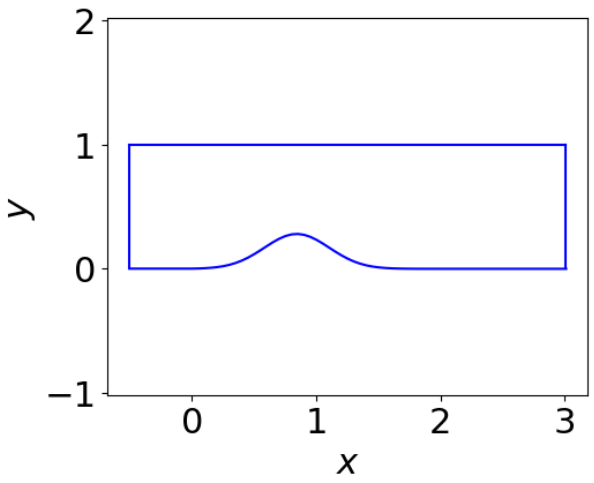


Figure 15.1: Domain of channel with a hill

```
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
```

14.2 modify_case.py

14.2.1 modify_init

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

```
# 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
```

15 Channel with a hill

This flow is setup in the directory `large-wave`.

DNS of this flow can be found in Assignment 1 in the course MTF271 Turbulence Modeling. The height of the hill is 0.28. The Reynolds number based on the bulk flow velocity and channel height is $Re_b = u_b H/\nu \simeq 36\,000$.

15.1 setup_case.py

15.1.1 Section 1

Hybrid discretization is set for all variables.

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

15.1.2 Section 2

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

```
kom = True
```

15.1.3 Section 4

The viscosity is set.

```
viscos=1/500
```

15.1.4 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
```

15.1.5 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='lgmres'
```

The convergence limit in the Python solvers is set to 10^{-8} for $\bar{u}$ and $\bar{v}$ and -10^{-6} (i.e. absolute) for 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 $1 \cdot 10^{-6}$ and the maximum number of iterations is set to 10000.

```
sormax=1e-6
maxit=10000
```

Note that when the convergence limit of $\bar{u}$ and $\bar{v}$ is set to $1 \cdot 10^{-6}$, the global convergence is stuck at $2 \cdot 10^{-5}$.

15.1.6 Section 9

The scaling velocity for the residuals is set to one

```
uin=1
```

15.1.7 Section 10

I set Dirichlet at the inlet (west) and Neumann at the outlet (reast).

```
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 south boundary is a wall (Dirichlet) and the north wall is a slip boundary (Neumann)

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

v_bc_south_type='d'
v_bc_north_type='d'

k_bc_south_type='d'
u_bc_north_type='n'

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

The inlet profiles of $\bar{u}$, $\bar{v}$, k and ω are set in `modify_case`.

15.1.8 `modify_init`

Inlet boundary condition from DNS (fully developed channel flow at $Re_\tau = 500$) are used to set initial condition.

```
# set inlet field in entre domain
   data = np.loadtxt('yp-u-k-omega.dat')
   yin = data[:,0]
   uin = data[:,1]
   kin = data[:,2]
   omin = data[:,3]
   uin_interp=np.interp(yp2d[0,:], yin, uin)
   kin_interp=np.interp(yp2d[0,:], yin, kin)
   omin_interp=np.interp(yp2d[0,:], yin, omin)
```

```
# set initial field in entire domain
   y0=y2d[0,-1]-y2d[0,0]
   for i in range(0,ni):
      yi=y2d[i,-1]-y2d[i,0]
      u2d[i,:]=uin_interp*y0/yi
      k2d[i,:]=kin_interp
      om2d[i,:]=omin_interp

   vis2d=k2d/om2d+viscos
```

15.1.9 `modify_init`

Load inlet data created with DNS.

```
# read inlet data. DNS data found in my eBook, Assignment 1 in the course MTF27
   data = np.loadtxt('yp-u-k-omega.dat')
   yin = data[:,0]
   uin = data[:,1]
   kin = data[:,2]
   omin = data[:,3]
   uin_interp=np.interp(yp2d[0,:], yin, uin)
   kin_interp=np.interp(yp2d[0,:], yin, kin)
   omin_interp=np.interp(yp2d[0,:], yin, omin)

# interpolate to the CFD grid

   u_bc_west = uin_interp
   k_bc_west = kin_interp
   om_bc_west = omin_interp
```

16 An improved EARSM using Neural Network (EARSM-NN)

The Neural Network (NN) model is created in the directory `NN`. Look at the Python script `NN.py`. Go to line 148 ('The neural network modules: end'). There I load data from a $k-\omega$ simulation and DNS data.

```
DNS_mean=np.loadtxt('y_u_k_eps_uv_channel-10000-k-omega.txt')
y_DNS=DNS_mean[:,0];
yplus_DNS= y_DNS/viscos
u_DNS=DNS_mean[:,1];
k_DNS=DNS_mean[:,2];
eps_DNS=DNS_mean[:,3]*viscos;
uv_DNS=DNS_mean[:,4];
dudy_DNS= np.gradient(u_DNS,yplus_DNS)
pk_DNS = -uv_DNS*dudy_DNS
tau_DNS=np.maximum(k_DNS/eps_DNS,6*(1/eps_DNS)**0.5)
```

```
dudy_DNS_org = np.copy(dudy_DNS)

DNS_mean=np.genfromtxt("P10k.txt",comments="%")
y_DNS_org=DNS_mean[:,0];
uu_DNS=DNS_mean[:,3]**2;
vv_DNS=DNS_mean[:,4]**2;
ww_DNS=DNS_mean[:,5]**2;
```

The DNS grid has 1051 cells and the $k-\omega$ grid has 55. I interpolate the DNS data to the $k-\omega$ grid

```
uu_DNS = np.interp(y_DNS, y_DNS_org, uu_DNS)
vv_DNS = np.interp(y_DNS, y_DNS_org, vv_DNS)
ww_DNS = np.interp(y_DNS, y_DNS_org, ww_DNS)
```

I exclude data points near the wall and near the center

```
index_choose=np.nonzero((yplus_DNS>5)&(yplus_DNS<9200))
```

The reason is that otherwise the NN training process does not converge because the gradients of the β coefficients are very large near the wall and the velocity gradient gets very small near the center.

Next, the β coefficients (the targets) are computed

```
# compute anisotropic Reynolds stresses
a11_DNS=uu_DNS/k_DNS-0.66666
a22_DNS=vv_DNS/k_DNS-0.66666
a33_DNS=ww_DNS/k_DNS-0.66666
a12_DNS=uv_DNS/k_DNS

# Array for storing b1, b2, b4
b1_DNS=2*a12_DNS/tau_DNS/dudy_DNS  # b1
b2_DNS=6*(a11_DNS+a22_DNS)/tau_DNS**2/dudy_DNS**2  # b2
b4_DNS=(a22_DNS-a11_DNS)/tau_DNS**2/dudy_DNS**2  # b4
```

and are then put into to the output matrix `y`

```
c = np.array([b1_DNS,b2_DNS,b4_DNS])
# transpose the target vector to make it a column vector
y = c.transpose()
```

Then the input parameters are scaled, reshaped and put into the general influence parameters matrix `X`, i.e.

```
pk_DNS_scaled = pk_DNS
# re-shape
pk_DNS_scaled = pk_DNS_scaled.reshape(-1,1)
yplus_DNS= yplus_DNS.reshape(-1,1)

# use standard scaler
scaler_pk = MinMaxScaler()
scaler_yplus = MinMaxScaler()
X=np.zeros((len(dudy_DNS),2))
X[:,0] = scaler_pk.fit_transform(pk_DNS_scaled)[:,0]
X[:,1] = scaler_yplus.fit_transform(yplus_DNS)[:,0]
```

I set the learning rate and the number of epochs (i.e. iterations)

```
learning_rate = 0.01 # loss = 8.8e-5,  max error = 0.025
my_batch_size = 1
epochs = 20000
```

Then the training process starts

```
############################ training starts here ########################
```

and ends

```
############################ training ends here ########################
```

In the next step, I test (i.e. predict) the β coefficients using the trained EARSM-NN model

```
preds = neural_net(X_test_tensor)

#transform from tensor to numpy
c_NN = preds.detach().numpy()

b1=c_NN[:,0]
b2=c_NN[:,1]
b4=c_NN[:,2]
```

Finally, I compute the Reynolds stresses

```
#
# compute the anisotropic stresses and Reynolds stresses using b1, b2 and from
a_11 = tau_DNS_test**2*dudy_DNS_test**2/12*(b2-6*b4)
uu_NN = (a_11+0.6666)*k_DNS_test

a_22 = tau_DNS_test**2*dudy_DNS_test**2/12*(b2+6*b4)
vv_NN = (a_22+0.6666)*k_DNS_test

a_33 = -tau_DNS_test**2*dudy_DNS_test**2/6*b2
ww_NN = (a_33+0.6666)*k_DNS_test

a_12 = b1*tau_DNS_test*dudy_DNS_test/2
uv_NN = a_12*k_DNS_test
```

17 Fully-developed channel flow using the EARSM-NN model

Here I setup a simulation of fully-developed channel flow the $Re_\tau 10\,000$. Go to the directory directory `channel-10000-earsm-NN`.

17.1 setup_case.py

17.1.1 Section 2

The EARSM is chosen along with the underlying $k - \omega$ model

```
kom = True
earsm = True
```

17.1.2 Section 10

The boundary conditions are set for $\overline{v_1'^2}$, $\overline{v_2'^2}$ and $\overline{v_1' v_2'}$ are set. Although I don't solve any transport equation for these quantities I do take the gradient of them and hence I must set boundary conditions.

```
# boundary conditions for uu
   uu_bc_west=np.zeros(nj)
   uu_bc_east=np.zeros(nj)
   uu_bc_south=np.zeros(ni)
   uu_bc_north=np.zeros(ni)

   uu_bc_west_type='n'
   uu_bc_east_type='n'
   uu_bc_south_type='d'
   uu_bc_north_type='d'

# boundary conditions for uv
   uv_bc_west=np.zeros(nj)
   uv_bc_east=np.zeros(nj)
   uv_bc_south=np.zeros(ni)
   uv_bc_north=np.zeros(ni)

   uv_bc_west_type='n'
   uv_bc_east_type='n'
   uv_bc_south_type='d'
   uv_bc_north_type='d'

# boundary conditions for vv
   vv_bc_west=np.zeros(nj)
   vv_bc_east=np.zeros(nj)
   vv_bc_south=np.zeros(ni)
   vv_bc_north=np.zeros(ni)

   vv_bc_west_type='n'
   vv_bc_east_type='n'
   vv_bc_south_type='d'
   vv_bc_north_type='d'
```

17.2 modify_case.py

This file is very similar to that in the directory `channel-5200/`

17.3 run-python

As I mentioned in Section 10.2, the module `calc_earsm` in the main script, `pyCALC-RANS.py`, is empty. You find one `calc_earsm` module in the file `calc_earsm.py` in this directory. It is inserted into the executable using a modified `run-python` which reads

```
#!/bin/bash
# delete first line
sed  '/setup_case()/d' setup_case.py>temp_file
# rename empty calc_earsm
sed 's/def calc_earsm/def calc_earsm_old/' ../pyCALC-RANS.py >temp_file1

# add new first line plus global declarations and calc_earsm
cat ../global temp_file modify_case.py calc_earsm.py \
temp_file1>exec-pyCALC-RANS.py;
../../anaconda3/bin/python3.8 -u  exec-pyCALC-RANS.py > out
```

In the `bash` script above, I rename the empty `calc_earsm` module to `calc_earsm_old` and then I add the local `calc_earsm` into `exec-pyCALC-RANS.py`.

I start the simulations by – as usual – by typing

```
run-python
```

in the terminal. Then I plot the results by typing

```
python pl_uvw-channel
```

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 in Python solver for k (max(limit,limit· norm(su3d)); if negative, the residuals are reduced by abs(limit)

`convergence_limit_p`: convergence limit in Python solver for $\bar{p}$ (max(limit,limit· norm(su3d)); if negative: abs(limit))

`convergence_limit_u`: convergence limit in Python solver for k (max(limit,limit· norm(su3d)); if negative, the residuals are reduced by abs(limit)

`convergence_limit_v`: convergence limit in Python solver for k (max(limit,limit· norm(su3d)); if negative, the residuals are reduced by abs(limit)

`convw,convs`: convection through west and south

`cyclic_x`: periodic boundary conditions in th x direction

`earsm`: the EARSM model is used

`eps_bc_east, eps_bc_north, eps_bc_south, eps_bc_west`: boundary values of ε at east, north, south, west boundary

`eps_bc_east_type, eps_bc_north_type, eps_bc_south_type, eps_bc_omest_type`: type of b.c. for ε

`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

`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_turb:` maximum number of iterations in the Python solver when solving the k, ε 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')

`pinn:` PINN is used [21] (see the README file in the main folder)

`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 10.19

`save`: the $\bar{u}$, $\bar{v}$ ... fields are saved to disk, see Section 10.20

`scheme`: discretization scheme for the $\bar{u}$, $\bar{v}$ and $\bar{w}$ equation. 'c'=central, 'h'=hybrid, 'm'=MUSCL, see Section 10.8

`scheme_turb`: discretization scheme for k, ε and ω. 'c'=central, 'h'=hybrid, 'm'=MUSCL, see Section 10.8

`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

`uu2d`: $\overline{v_1'^2} - 2k/3$, anisotropic Reynolds normal stress (when `earsm = True`)

`uu_bc_east`, `uu_bc_north`, `uu_bc_south`, `uu_bc_west`: boundary values of $\overline{v_1'^2} - 2k/3$ at east, north, south, west boundary (when `earsm = True`)

`uv2d`: $\overline{v_1' v_2'}$, anisotropic Reynolds shear stress (when `earsm = True`)

`uv_bc_east`, `uv_bc_north`, `uv_bc_south`, `uv_bc_west`: boundary values of $\overline{v_1' v_2'}$ at east, north, south, west boundary (when `earsm = True`)

`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

`vv2d`: $\overline{v_2'^2} - 2k/3$, anisotropic Reynolds normal stress (when `earsm = True`)

`vv_bc_east, vv_bc_north, vv_bc_south, vv_bc_west`: boundary values of $\overline{v_2'^2} - 2k/3$ at east, north, south, west boundary (when `earsm = True`)

`ww2d`: $\overline{v_3'^2} - 2k/3$, anisotropic Reynolds normal stress (when `earsm = True`)

`ww_bc_east, ww_bc_north, ww_bc_south, ww_bc_west`: boundary values of $\overline{v_3'^2} - 2k/3$ at east, north, south, west boundary (when `earsm = True`)

`x2d`: the x coordinate of a corner of a control volume, see Fig. 2.3

`xp2d`: the x coordinate of the center of a control volume, see Fig. 2.3

`y2d`: the y coordinate of a corner of a control volume, see Fig. 2.3

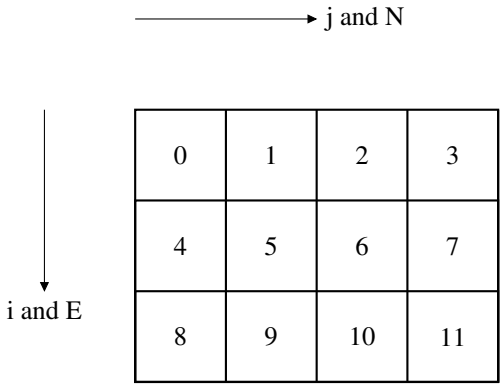
`yp2d`: the y coordinate of the center a control volume, see Fig. 2.3

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 [22] may be used for all variables.

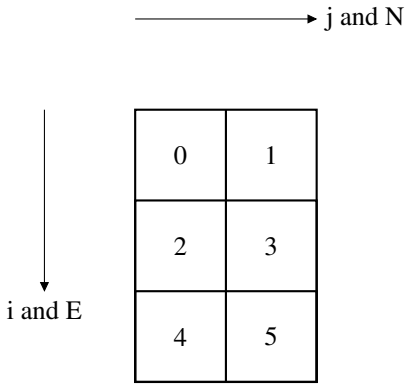
B.1 2D grid, $ni \times nj = (3, 4)$



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

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

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



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

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

References

[1] Lars Davidson. Using neural network for improving an explicit algebraic stress model in 2D flow. In James C. Tyacke and Nagabhushana Rao Vadlamani, editors, *Proceedings of the Cambridge Unsteady Flow Symposium 2024*, pages 37–53, Cham, 2025. Springer Nature Switzerland. ISBN 978-3-031-69035-8. URL https://www.tfd.chalmers.se/~lada/pyCALC-sRANS.html.

[2] 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. URL https://www.tfd.chalmers.se/~lada/postscript_files/py-scalc-sles.pdf.

[3] B. van Leer. Towards the ultimate conservative difference scheme. v. a second-order sequel to godunov's method. *Journal of Computational Physics*, 32(1):101–136, 1979. ISSN 0021-9991. doi: https://doi.org/10.1016/0021-s9991(79)90145-s1. URL https://www.sciencedirect.com/science/article/pii/0021999179901451.

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

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

[6] K. Abe, T. Kondoh, and Y. Nagano. A new turbulence model for predicting fluid flow and heat transfer in separating and reattaching flows - 1. Flow field calculations. *Int. J. Heat Mass Transfer*, 37(1):139–151, 1994.

[7] W. Rodi. A new algebraic relation for calculating the Reynolds stresses. *ZAMM*, 56:T219–T221, 1976.

[8] B. E. Launder, G. J. Reece, and W. Rodi. Progress in the development of a Reynolds-stress turbulence closure. *Journal of Fluid Mechanics*, 68(3):537–566, 1975.

[9] L. Davidson. Fluid mechanics, turbulent flow and turbulence modeling. eBook, Division of Fluid Dynamics, Dept. of Mechanics and Maritime Sciences, Chalmers University of Technology, Gothenburg, 2021.

[10] D. B. Taulbee. An improved algebraic Reynolds stress model and corresponding nonlinear stress model. *Physics of Fluids A*, 4:2555–2561, 1992.

[11] S. Wallin and A.V. Johansson. A new explicit algebraic Reynolds stress model for incompressible and compressible turbulent flows. *Journal of Fluid Mechanics*, 403:89–132, 2000.

[12] S. S. Girimaji. Fully-explicit and self-consistent algebraic Reynolds stress model. *Theoretical and Computational Fluid Dynamics*, 8:387–402, 1996.

[13] S.S. Girimaji. Fully-explicit and self-consistent algebraic Reynolds stress model. ICASE Rep. 95-82, Institute for Computer Applications in Science and Engineering NASA Langley Research Center, Hampton, VA, USA, 1995.

[14] S. Wallin and A.V. Johansson. A new explicit algebraic Reynolds stress turbulence model including an improved near-wall treatment. In C.-J. Chen, C. Shih, J. Lienau, and R. J. Kung, editors, *Proc. Flow Modeling and Turbulence Measurements VI, Tallahassee F. L.*, pages 399–406. Balkema, 1996.

[15] S. B. Pope. A more general effective-viscosity hypothesis. *Journal of Fluid Mechanics*, 472:331–340, 1975.

[16] S. Hoyas and J. Jimenez. Reynolds number effects on the reynolds-stress budgets in turbulent channels,http://torroja.dmt.upm.es/channels/. *Physics of Fluids A*, 20 (101511), 2008. doi: http://dx.doi.org/10.1063/1.3005862.

[17] M. Lee and R. D. Moser. Direct numerical simulation of turbulent channel flow up to $Re_\tau \approx 5200$. *Journal of Fluid Mechanics*, 774:395–415, 2015. doi: 10.1017/jfm.2015.268. URL https://doi.org/10.1017/jfm.2015.268.

[18] Sergio Hoyas, Martin Oberlack, Francisco Alcántara-Ávila, Stefanie V. Kraheberger, and Jonathan Laux. Wall turbulence at high friction Reynolds numbers. *Phys. Rev. Fluids*, 7:014602, Jan 2022. doi: 10.1103/PhysRevFluids.7.014602. URL https://link.aps.org/doi/10.1103/PhysRevFluids.7.014602.

[19] J.A. Sillero, J. Jimenez, and R.D. Moser. One-point statistics for turbulent wall-bounded flows at Reynolds numbers up to $\delta^+ \simeq 2000$. *Physics of Fluids*, 25 (105102), 2014. doi: https://doi.org/10.1063/1.4823831. URL https://doi.org/10.1063/1.4823831.

[20] Karthik Duraisamy. Perspectives on machine learning-augmented reynolds-averaged and large eddy simulation models of turbulence. *Physical Review Fluids*, 6, 05 2021. doi: 10.1103/PhysRevFluids.6.050504.

[21] L. Davidson. Using Physical Informed Neural Network (PINN) to improve a $k - \omega$ turbulence model. In *15th International ERCOFTAC Symposium on Engineering Turbulence Modelling and Measurements (ETMM15), Dubrovnik on 22-24 September*, 2025. URL https://www.tfd.chalmers.se/~lada/Using-sPhysical-sInformed-sNeural-sNetwork-sPINN-simprove-sa-sk-somega-stur html.

[22] 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.