pyCALC-LES: A Python Code for DNS, LES and Hybrid LES-RANS

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

January 5, 2021

Abstract

This report gives some details on **pyCALC-LES** and how to use it. It is written in Python (3.7). The code solves the incompressible momentum equations, the continuity equation and transport equations for modeled turbulent quantities such as k, ε and ω. The density is assumed to be constant and equal to one, i.e. $\rho \equiv 1$. The transport equations are solved in 3D and the grid may be curvi-linear in the $x-y$ plane. In the z direction the grid is equi-distant.

The code can be used for DNS, LES or DES (hybrid LES-RANS). For LES, the Smagorinsky model and the WALE model are implemented. For DES, a $k-\omega$ DES model and a PANS $k-\varepsilon$ model are implemented.

pyCALC-LES is a finite volume code. It is fully vectorized (i.e. no `for` loops). The solution procedure is based on fractional step. Second-order central differencing is used in space and the Crank-Nicolson scheme in time. The discretized equations are solved with Pythons sparse matrix solvers (currently `linalg.lgmres` or `linalg.gmres` are used). For the pressure Poisson equation, the `pyAMG` [19] has been found to be very efficient.

Contents

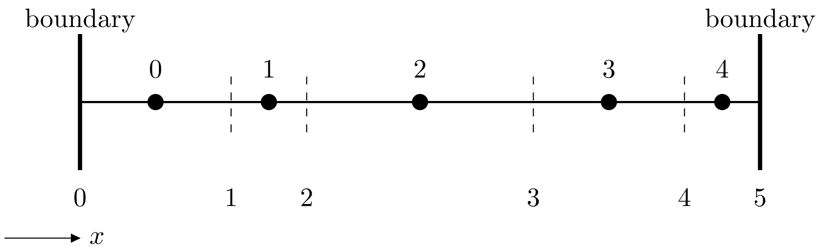


Figure 1.1: 1D grid. `ni=5`. The bullets denote cell centers (and control volume) which are labeled 0–4. Dashed lines denote control volume faces labeled 0–5.

1 Geometrical details of the grid

1.1 Grid

The grid (`x2d,y2d`) must be generated by the user. The grid spacing in the third direction is constant and is set by `dz`. 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. Note that (ξ, η, z) must form a right-hand coordinate system. The grid in the $x-y$ plane 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), S(outh), N(orth), S(outh), H(igh) and L(ow)], 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), then north-south (j direction), and finally high-low (k 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)
```

and the z coordinate is constant. 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)

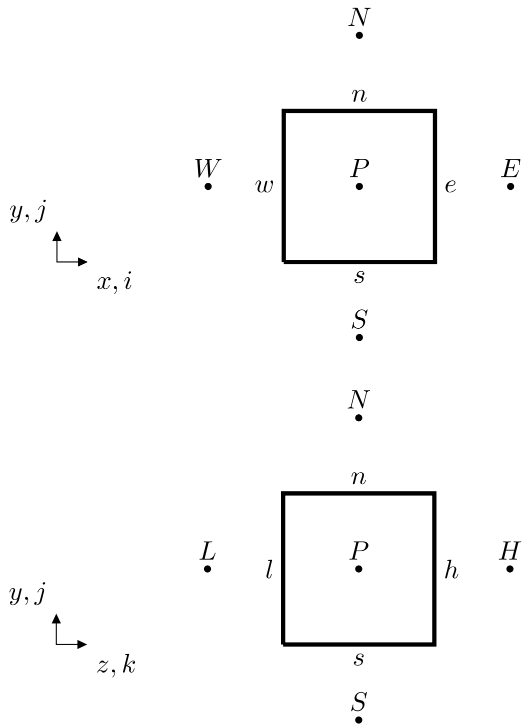


Figure 1.2: Control volume. Top: $x - y$ plane; bottom: $y - z$ plane.

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

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

$$\begin{aligned} a_x &= x2d(i,j+1) - x2d(i,j) && (1.1)\\ a_y &= y2d(i,j+1) - y2d(i,j)\\ b_x &= x2d(i+1,j) - x2d(i,j)\\ b_y &= y2d(i+1,j) - y2d(i,j)\\ & && (1.2) \end{aligned}$$

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

$$\begin{aligned} A_{wx} &= -a_y \Delta z && (1.3)\\ A_{wy} &= a_x \Delta z && (1.4)\\ A_{sx} &= b_y \Delta z && (1.5)\\ A_{sy} &= -b_x \Delta z && (1.6)\\ & && (1.7) \end{aligned}$$

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.

$$A1 = \frac{1}{2}|\vec{a} \times \vec{b}|; \qquad A2 = \frac{1}{2}|\vec{b} \times \vec{c}| \qquad (1.8)$$

The area for the low face is then obtained as

$$A_z = A1 + A2 \qquad (1.9)$$

which is stored in the Python array `areaz`.

The volume of the control volume is computed as $A_z \Delta z$ which is stored in the Python array `vol`.

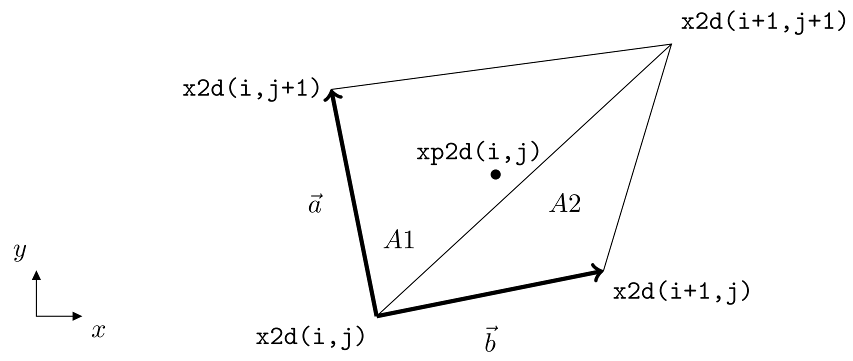


Figure 1.3: Calculation of areas and volume of cell `i,j,k`.

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

$$\phi_w = f_x \phi_P + (1 - f_x)\phi_W \tag{1.10}$$

where

$$f_x = \frac{|\overrightarrow{Ww}|}{|\overrightarrow{Pw}| + |\overrightarrow{Ww}|} \tag{1.11}$$

where $|\overrightarrow{Pw}|$ is the distance from P (the node) to w (the west face). In **pyCALC-LES** 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-LES** computed as follows. We apply Green's formula to the control volume, i.e.

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

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

$$\frac{\partial \Phi}{\partial x} = \frac{1}{V}\left(\Phi_e A_{ex} - \Phi_w A_{wx} + \Phi_n A_{nx} - \Phi_s A_{sx} + \Phi_h A_{hx} - \Phi_l A_{lx}\right) \tag{1.12}$$

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

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

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

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.

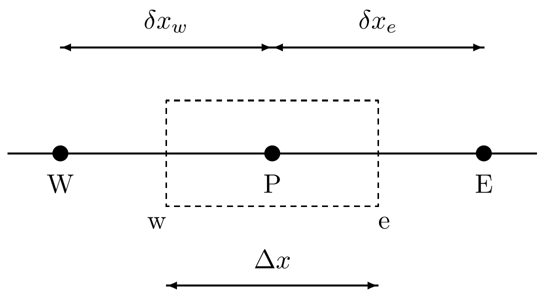


Figure 2.1: 1D control volume. Node P located in the middle of the control volume.

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

where (see Fig. 2.1):

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

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

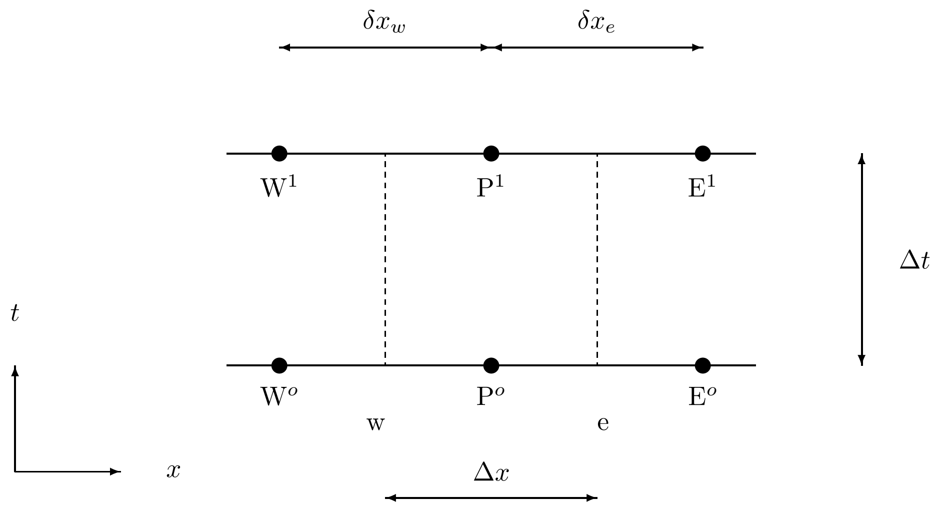


Figure 2.2: Control volume for 1D unsteady diffusion

2.1 Unsteady diffusion

We discretize the unsteady diffusion equation

$$\frac{\partial \phi}{\partial t} = \frac{\partial}{\partial x}\left(\Gamma \frac{\partial \phi}{\partial x}\right)$$

over a 1D control volume (see Fig. 2.2). We integrate in space and time

$$\int_t^{t+\Delta t}\int_w^e \frac{\partial \phi}{\partial t} dx dt = \int_t^{t+\Delta t}\int_w^e \frac{\partial}{\partial x}\left(\Gamma \frac{\partial \phi}{\partial x}\right) dx dt$$

Left-hand side:

$$\int_w^e \left[\underbrace{\phi^1}_{t+\Delta t} - \underbrace{\phi^o}_{t}\right] dx = (\phi_P^1 - \phi_P^o)\Delta x$$

Right-hand side:

$$\int_t^{t+\Delta t}\left[\left(\Gamma \frac{\partial \phi}{\partial x}\right)_e - \left(\Gamma \frac{\partial \phi}{\partial x}\right)_w\right] dt =$$
$$\int_t^{t+\Delta t}\left[\Gamma_e \frac{\phi_E - \phi_P}{\delta x_e} - \Gamma_w \frac{\phi_P - \phi_W}{\delta x_w}\right] dt$$

At what time should ϕ_W, ϕ_P and ϕ_E be taken?

1. Fully implicit: take them at the new time step $t + \Delta t$, i.e. ϕ_W^1, ϕ_P^1 and ϕ_E^1 (first-order accurate).
2. Fully explicit: take them at the old time step t, i.e. ϕ_W^o, ϕ_P^o and ϕ_E^o (first-order accurate).
3. Use central differencing in time (Crank-Nicolson). Second-order accurate. Note that this is what we did in space when integrating the LHS.

2.1.1 Crank-Nicolson

For Crank-Nicolson the interpolation factor in time, α, is equal to 0.5. Below we express the time integration in a general way using α. When $\alpha = 0$, it corresponds to fully explicit and when $\alpha = 1$, it corresponds to fully implicit. We get

$$\begin{aligned}
a_P\phi_P &= \alpha a_E\phi_E + \alpha a_W\phi_W \\
&+ \underbrace{(1-\alpha)(a_E\phi_E^o + a_W\phi_W^o) + (a_P^o - (1-\alpha)(a_E + a_W))\,\phi_P^o}_{S_U} \\
a_E &= \frac{\Gamma_e}{\delta x_e},\ a_W = \frac{\Gamma_w}{\delta x_w},\ a_P^o = \frac{\Delta x}{\Delta t} \\
a_P &= \alpha\,(a_E + a_W) + a_P^o
\end{aligned} \tag{2.5}$$

The Crank-Nicolson scheme ($\alpha = 0.5$) is implicit and unconditionally stable. In practice, however, it is less stable than the fully implicit scheme. Crank-Nicolson in time can be compared with central differencing in space, even though it is much more stable.

2.2 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.

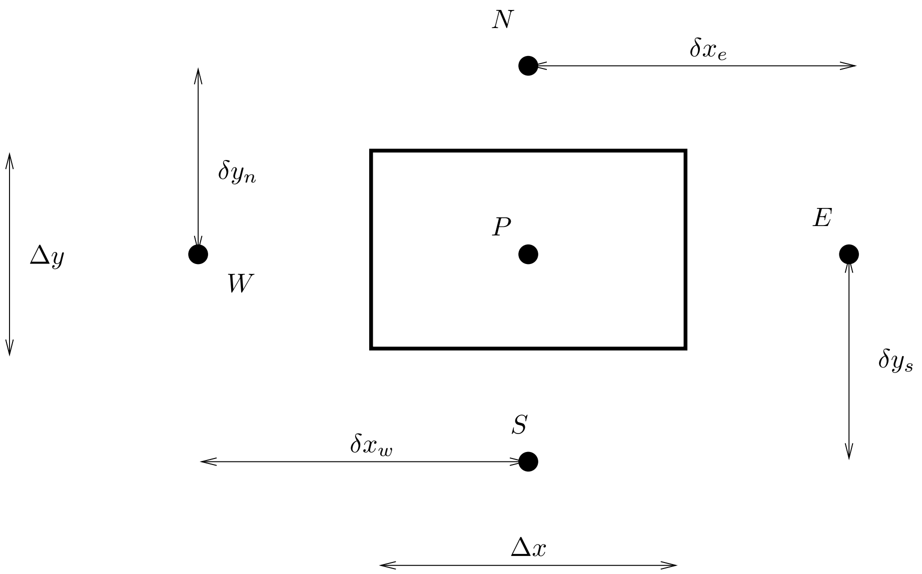


Figure 2.3: 2D control volume.

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.

2.3 2D Diffusion

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

$$\frac{\partial}{\partial x}\left(\Gamma\frac{\partial \phi}{\partial x}\right) + \frac{\partial}{\partial y}\left(\Gamma\frac{\partial \phi}{\partial y}\right) + S = 0. \tag{2.6}$$

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. 2.3, 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.

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

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. 2.6 gives

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

Rewriting it as an algebraic equation for ϕ_P, we get

$$\begin{aligned} a_P\phi_P &= a_E\phi_E + a_W\phi_W + a_N\phi_N + a_S\phi_S + S_U \\ a_E &= \frac{\Gamma_e\Delta y}{\delta x_e},\ a_W = \frac{\Gamma_w \Delta y}{\delta x_w},\ a_N = \frac{\Gamma_n\Delta x}{\delta y_n},\ a_S = \frac{\Gamma_s\Delta x}{\delta y_s} \\ S_U &= \bar{S}\Delta x\Delta y,\ a_P = a_E + a_W + a_N + a_S - S_P. \end{aligned} \tag{2.7}$$

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

2.4 3D diffusion

In **pyCALC-LES** the diffusion coefficients are computed using areas and volume, i.e.

$$\begin{aligned} a_P\phi_P &= a_E\phi_E + a_W\phi_W + a_N\phi_N + a_S\phi_S + a_H\phi_H + a_L\phi_L + S_U \\ a_W &= \frac{\Gamma_w A_w^2}{V_w} \\ a_S &= \frac{\Gamma_s A_s^2}{V_s} \\ a_L &= \frac{\Gamma_l A_l^2}{V_l} \end{aligned}$$

The east, north and high coefficients are computed from a_W, a_S and a_L, respectively, as

$$\begin{aligned} a_{E,i} &= a_{W,i+1} \\ a_{N,j} &= a_{S,j+1} \\ a_{H,k} &= a_{L,k+1} \end{aligned}$$

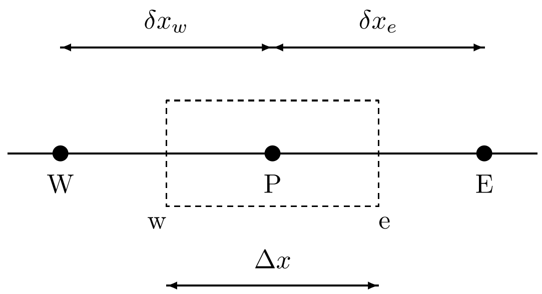


Figure 3.1: 1D control volume. Node P located in the middle of the control volume.

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. 10), and for constant mesh spacing $f_x = 0.5$. Assuming constant equidistant mesh (i.e. $\delta x_w = \delta x_e = \Delta x$)

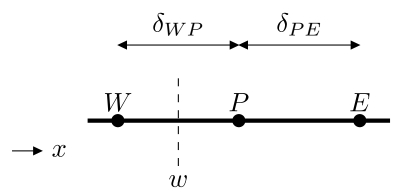


Figure 3.2: Constant mesh spacing. $\bar{u} > 0$.

so that $f_x = 0.5$, inserting the discretized diffusion and the convection terms into Eq. 3.1 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. However, in LES this does usually not pose any problems.

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.

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-LES** 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. 1.1. 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 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 $\bar{v}$ and w it reads

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

The additional term for the diffusion reads

$$\begin{aligned} S^u_{U,add,diff} &= \frac{\nu_{tot}A_w}{\Delta x}\bar{u}_{in} \\ S^v_{U,add,diff} &= \frac{\nu_{tot}A_w}{\Delta x}\bar{v}_{in} \\ S^w_{U,add,diff} &= \frac{\nu_{tot}A_w}{\Delta x}\bar{w}_{in} \\ S_{P,add,diff} &= -\frac{\nu_{tot}A_w}{\Delta x} \end{aligned} \tag{3.5}$$

where $S_{P,add,diff}$ is the same for $\bar{u}$, $\bar{v}$ and $\bar{w}$. The viscous part of Eq. 3.5 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.

4 The Fractional-step method

<u>modules:</u> `calcp, correct_conv`

A numerical method based on an implicit, finite volume method with collocated grid arrangement, central differencing in space, and Crank-Nicolson ($\alpha = 0.5$) in time is briefly described below. An implicit, two-step time-advancement methods is used [3]. The Navier-Stokes equation for the $\bar{u}_i$ velocity reads

$$\frac{\partial \bar{u}_i}{\partial t} + \frac{\partial}{\partial x_j}(\bar{u}_i\bar{u}_j) = -\frac{\partial \bar{p}}{\partial x_i} + \nu\frac{\partial^2 \bar{u}_i}{\partial x_j \partial x_j} - \frac{\partial \tau_{ij}}{\partial x_j} \tag{4.1}$$

The discretized momentum equations read

$$\begin{aligned} \bar{v}_i^{n+1/2} &= \bar{v}_i^n + \Delta t H\left(\bar{v}^n, \bar{v}_i^{n+1/2}\right) \\ &-\alpha\Delta t\frac{\partial \bar{p}^{n+1/2}}{\partial x_i} - (1-\alpha)\Delta t\frac{\partial \bar{p}^n}{\partial x_i} \end{aligned} \tag{4.2}$$

where H includes convective, viscous and SGS terms. In SIMPLE notation this equation reads

$$a_P\bar{v}_i^{n+1/2} = \sum_{nb} a_{nb}\bar{v}^{n+1/2} + S_U - \alpha\frac{\partial \bar{p}^{n+1/2}}{\partial x_i}\Delta V$$

where S_U includes the explicit pressure gradient. The face velocities $\bar{v}_{f,i}^{n+1/2} = 0.5(\bar{v}_{i,J}^{n+1/2} + \bar{v}_{i,J-1}^{n+1/2})$ (note that J denotes node number and i is a tensor index) do not satisfy continuity. Create an intermediate velocity field by subtracting the implicit pressure gradient from Eq. 4.2, i.e.

$$\begin{aligned} \bar{v}_i^* &= \bar{v}_i^n + \Delta t H\left(\bar{v}^n, \bar{v}_i^{n+1/2}\right) - (1-\alpha)\Delta t\frac{\partial \bar{p}^n}{\partial x_i} \\ \Rightarrow \bar{v}_i^* &= \bar{v}_i^{n+1/2} + \alpha\Delta t\frac{\partial \bar{p}^{n+1/2}}{\partial x_i} \end{aligned} \tag{4.3}$$

Take the divergence of Eq. 4.3b and require that $\partial \bar{v}_{f,i}^{n+1/2}/\partial x_i = 0$ so that

$$\frac{\partial^2 \bar{p}^{n+1}}{\partial x_i \partial x_i} = \frac{1}{\Delta t \alpha}\frac{\partial \bar{v}_{f,i}^*}{\partial x_i} \tag{4.4}$$

	RANS	**LES**
Domain	2D or 3D	always 3D
Time domain	steady or unsteady	always unsteady
Space discretization	2nd order upwind	central differencing
Time discretization	1st order	2nd order (e.g. C-N)
Turbulence model	more than two-equations	zero- or one-equation

Table 4.1: Differences between a finite volume RANS and LES code.

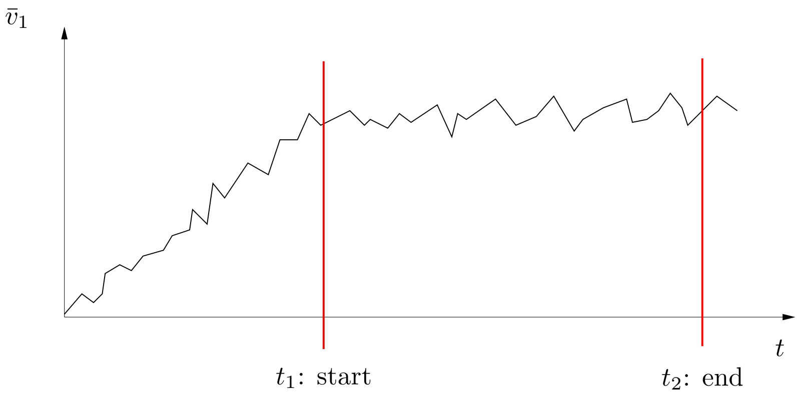


Figure 4.1: Time averaging in LES.

This is a diffusion equation which is discretization in the same way as in Sections 2.3 and 2.4 (the diffusion coefficient Γ is set to one).

The Poisson equation for $\bar{p}^{n+1}$ is solved with an efficient algebraic multigrid method, pyAMg [19]. The face velocities are corrected as

$$\bar{v}_{f,i}^{n+1} = \bar{v}_{f,i}^{*} - \alpha\Delta t \frac{\partial \bar{p}^{n+1}}{\partial x_i} \tag{4.5}$$

1. Solve the discretized filtered Navier-Stokes equation, Eq. 4.3, for $\bar{v}_1$, $\bar{v}_2$ and $\bar{v}_3$.
2. Create an intermediate velocity field $\bar{v}_i^*$ from Eq. 4.3.
3. Use linear interpolation to obtain the intermediate velocity field, $\bar{v}_{f,i}$, at the face
4. The Poisson equation (Eq. 4.4) is solved with pyAMG.
5. Compute the face velocities (which satisfy continuity) from the pressure and the intermediate face velocity from Eq. 4.5
6. Step 1 to 4 is performed till convergence (normally one iteration) is reached.
7. The turbulent viscosity is computed.
8. Next time step.

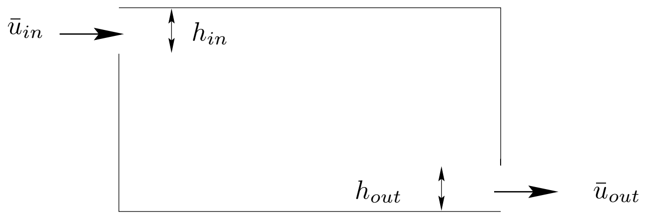


Figure 5.1: Outlet boundary condition. Small outlet

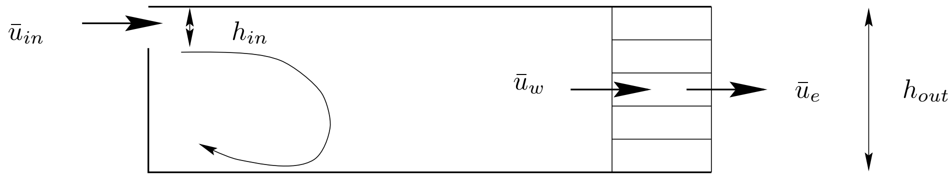


Figure 5.2: Outlet boundary condition. Large outlet.

5 Boundary Conditions

5.1 Outlet velocity, small outlet

For *small* outlets, the outlet velocity can be determined from global continuity. As the inlet 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: $\dot{V}_{in} = \sum_{inlet} \bar{u}_{in} \Delta y$
   - 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 Smagorinsky Model

module: `vist_les`

The simplest model is the Smagorinsky model [20]:

$$\begin{aligned} \tau_{ij} - \frac{1}{3}\delta_{ij}\tau_{kk} &= -\nu_{sgs}\left(\frac{\partial \bar{v}_i}{\partial x_j} + \frac{\partial \bar{v}_j}{\partial x_i}\right) = -2\nu_{sgs}\bar{s}_{ij} \\ \nu_{sgs} &= (C_S\Delta)^2\sqrt{2\bar{s}_{ij}\bar{s}_{ij}} \equiv (C_S\Delta)^2|\bar{s}| \end{aligned} \tag{6.1}$$

and the filter-width is taken as the local grid size

$$\Delta = (\Delta V_{ijk})^{1/3} \tag{6.2}$$

Near the wall, the SGS viscosity becomes quite large since the velocity gradient is very large at the wall. A convenient way to dampen the SGS viscosity near the wall is simply to use the RANS length scale as an upper limit, i.e.

$$\Delta = \min\left\{(\Delta V_{ijk})^{1/3}, \kappa n\right\} \tag{6.3}$$

where n is the distance to the nearest wall. C_S is set to 0.1 (in **pyCALC-LES** it is set by `cmu`).

7 The WALE model

module: `vist_wale`

The WALE model by [18] reads

$$\begin{aligned} g_{ij} &= \frac{\partial \bar{v}_i}{\partial x_j},\; g^2_{ij} = g_{ik}g_{kj} \\ \bar{s}^d_{ij} &= \frac{1}{2}\left(g^2_{ij} + g^2_{ji}\right) - \frac{1}{3}\delta_{ij}g^2_{kk} \\ \nu_{sgs} &= (C_m\Delta)^2\frac{\left(\bar{s}^d_{ij}\bar{s}^d_{ij}\right)^{3/2}}{(\bar{s}_{ij}\bar{s}_{ij})^{5/2} + \left(\bar{s}^d_{ij}\bar{s}^d_{ij}\right)^{5/4}} \end{aligned} \tag{7.1}$$

with $C_m = 0.325$ which corresponds to $C_s = 0.1$.

8 The PANS Model

module: `calck_ls, calceps_ls, vist_pans`

The low-Reynolds number (LRN) turbulence model is taken from [17]. Incorporated in the PANS formulation it reads

$$\begin{aligned} \frac{\partial k}{\partial t} + \frac{\partial}{\partial x_j}(\bar{v}_j k) &= \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_{ku}}\right)\frac{\partial k}{\partial x_j}\right] + P_k + P_{k_{tr}} - \varepsilon - D \\ \frac{\partial \varepsilon}{\partial t} + \frac{\partial}{\partial x_j}(\bar{v}_j \varepsilon) &= \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_{\varepsilon u}}\right)\frac{\partial \varepsilon}{\partial x_j}\right] + C_{\varepsilon 1}P_k\frac{\varepsilon}{k} - C^*_{\varepsilon 2}\frac{\varepsilon^2}{k} + E + S_{Yap} \end{aligned} \tag{8.1}$$

$$
\begin{aligned}
\nu_t &= C_\mu f_\mu \frac{k^2}{\varepsilon}, P_k = 2\nu_t \bar{s}_{ij}\bar{s}_{ij}, \bar{s}_{ij} = \frac{1}{2}\left(\frac{\partial \bar{v}_i}{\partial x_j} + \frac{\partial \bar{v}_j}{\partial x_i}\right) \\
C^*_{\varepsilon 2} &= C_{\varepsilon 1} + \frac{f_k}{f_\varepsilon}(C_{\varepsilon 2} f_2 - C_{\varepsilon 1}), \sigma_{ku} \equiv \sigma_k \frac{f_k^2}{f_\varepsilon}, \sigma_{\varepsilon u} \equiv \sigma_\varepsilon \frac{f_k^2}{f_\varepsilon} \\
\sigma_k &= 1.0, \sigma_\varepsilon = 1.31, C_{\varepsilon 1} = 1.44, C_{\varepsilon 2} = 1.82, C_\mu = 0.09, f_\varepsilon = 1
\end{aligned}
\tag{8.2}
$$

The wall boundary conditions are

$$
k_w = 0, \quad \varepsilon_w = 0 \tag{8.3}
$$

The term $P_{k_{tr}}$ in Eq. 8.1 is an additional term which is non-zero in the inlet-adjacent cells because $\partial f_k/\partial x_1 \neq 0$ inlet. The damping functions and the additional terms are defined as

$$
\begin{aligned}
f_2 &= 1 - \exp\left(R_T^2\right) \\
f_\mu &= \exp\left[\frac{-3.4}{\left(1 + R_T/50\right)^2}\right] \\
E &= 2\nu\nu_t \left(\frac{\partial^2 \bar{v}}{\partial x_2^2}\right)^2 \\
D &= 2\nu \left(\frac{\partial\sqrt{k}}{\partial x_2}\right)^2 \\
S_{Yap} &= 083(T_1 - 1)T_1^2 \frac{\varepsilon^2}{k}, \quad T_1 = \frac{k^{3/2}}{2.6 d\varepsilon}
\end{aligned}
$$

where d is the wall distance. The last term is the Yap term [24] whose role is to drive the turbulent lengthscale $k^{3/2}/\varepsilon$ to its equilibrium value of $2.6d$. This term is set to zero when the equations are in LES mode (i.e. when $f_k < 1$).

The function f_ε, the ratio of the modeled to the total dissipation, is set to one since the turbulent Reynolds number is high. f_k is computed as [12]

$$
\begin{aligned}
f_k &= \max\left[0, \min\left(1, 1 - \frac{\psi - 1}{C_{\varepsilon 2} - C_{\varepsilon 1}}\right)\right] \\
\psi &= \max\left(1, \frac{k^{3/2}/\varepsilon}{C_{DES}\Delta_{max}}\right), \quad \Delta_{max} = \max(\Delta x_1, \Delta x_2, \Delta x_3)
\end{aligned}
\tag{8.4}
$$

which means that the interface is chosen automatically.

9 The $k-\omega$ DES model

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

The Wilcox $k-\omega$ RANS turbulence model [23] modified for DES reads

$$
\begin{aligned}
\frac{\partial k}{\partial t} + \frac{\partial \bar{v}_i k}{\partial x_i} &= P^k - F_{DES} c_\mu k\omega + \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_j}\right] \\
\frac{\partial \omega}{\partial t} + \frac{\partial \bar{v}_i \omega}{\partial x_i} &= C_{\omega_1}\frac{\omega}{k}P^k - C_{\omega 2}\omega^2 + \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_\omega}\right)\frac{\partial \omega}{\partial x_j}\right] \\
\nu_t &= \frac{k}{\omega}
\end{aligned}
\tag{9.1}
$$

$$F_{DES} = \max\left(\frac{k^{1/2}}{c_\mu \omega \Delta}, 1\right), \quad \Delta = 0.67\max(\Delta x_1, \Delta x_2, \Delta x_2)$$

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

$$k_w = 0, \quad \omega_w = 10\frac{6\nu}{C_{\omega 2} y^2} \tag{9.2}$$

where y is the wall distance between the wall-adjacent cell center and the wall.

10 Inlet boundary conditions

In RANS it is sufficient to supply profiles of the mean quantities such as velocity and temperature plus the turbulent quantities (e.g. k and ε). However, in unsteady simulations (LES, URANS, DES ...) the time history of the velocity and temperature need to be prescribed; the time history corresponds to turbulent, resolved fluctuations. In some flows it is critical to prescribe reasonable turbulent fluctuations, but in many flows it seems to be sufficient to prescribe constant (in time) profiles [7, 8].

There are different ways to create turbulent inlet boundary conditions. One way is to use a pre-cursor DNS or well resolved LES of channel flow. This method is limited to fairly low Reynolds numbers and it is difficult (or impossible) to re-scale the DNS fluctuations to higher Reynolds numbers.

Another method based partly on synthesized fluctuations is the vortex method [16]. It is based on a superposition of coherent eddies where each eddy is described by a shape function that is localized in space. The eddies are generated randomly in the inflow plane and then convected through it. The method is able to reproduce first and second-order statistics as well as two-point correlations.

A third method is to take resolved fluctuations at a plane downstream of the inlet plane, re-scale them and use them as inlet fluctuations.

Below we present a method of generating synthesized inlet fluctuations.

10.1 Synthesized turbulence

module: `synt_fluct`.

The method described below was developed in [1, 2] for creating turbulence for generating noise. It was later further developed for inlet boundary conditions [11, 4, 6, 5].

A turbulent fluctuating velocity fluctuating field (whose average is zero) can be expressed using a Fourier series, see [10]. Let us re-write this formula as

$$\begin{aligned} a_n\cos(nx) + b_n\sin(nx) &= \\ c_n\cos(\alpha_n)\cos(nx) + c_n\sin(\alpha_n)\sin(nx) &= c_n\cos(nx-\alpha_n) \end{aligned} \tag{10.1}$$

where $a_n = c_n\cos(\alpha)$, $b_n = c_n\sin(\alpha_n)$. The new coefficient, c_n, and the phase angle, α_n, are related to a_n and b_n as

$$c_n = \left(a_n^2 + b_n^2\right)^{1/2}, \quad \alpha_n = \arctan\left(\frac{b_n}{a_n}\right) \tag{10.2}$$

A general form for a turbulent velocity field can thus be written as

$$\mathbf{v}'(\mathbf{x}) = 2\sum_{n=1}^{N} \hat{u}^n \cos(\boldsymbol{\kappa}^n \cdot \mathbf{x} + \psi^n)\boldsymbol{\sigma}^n \tag{10.3}$$

where $\hat{u}^n$, ψ^n and σ_i^n are the amplitude, phase and direction of Fourier mode n. The synthesized turbulence at one time step is generated as follows.

10.2 Random angles

The angles φ^n and θ^n determine the direction of the wavenumber vector $\boldsymbol{\kappa}$, see Eq. 10.3 and Eq. 10.1; α^n denotes the direction of the velocity vector, $\mathbf{v}'$. For more details, see [10].

10.3 Highest wave number

Define the highest wave number based on mesh resolution $\kappa_{max} = 2\pi/(2\Delta)$ (see [10]), where Δ is the grid spacing. Often the smallest grid spacing near the wall is too small, and then a slightly larger values may be chosen. Here we simply set $\kappa_{max} = 2\pi/(\Delta z)$.

10.4 Smallest wave number

Define the smallest wave number from $\kappa_1 = \kappa_e/p$, where $\kappa_e = \alpha 9\pi/(55L_t)$, $\alpha = 1.453$. The turbulent length scale, L_t, may be estimated in the same way as in RANS simulations, i.e. $L_t \propto \delta$ where δ denotes the inlet boundary layer thickness. In [4, 6, 5] it was found that $L_t \simeq 0.1\delta_{in}$ is suitable. Here we usually use $L_t \simeq 0.2\delta_{in}$.

Factor p should be larger than one to make the largest scales larger than those corresponding to κ_e. A value $p = 2$ is suitable.

10.5 Divide the wave number range

Divide the wavenumber space, $\kappa_{max} - \kappa_1$, into N modes, equally large, of size $\Delta\kappa$.

10.6 von Kármán spectrum

A modified von Kármán spectrum is chosen, see Eq. 10.4 and Fig. 10.2. The amplitude $\hat{u}^n$ of each mode in Eq. 10.3 is then obtained from

$$\begin{aligned}
\hat{u}^n &= (E(\kappa)\Delta\kappa)^{1/2} \\
E(\kappa) &= c_E \frac{u_{rms}^2}{\kappa_e} \frac{(\kappa/\kappa_e)^4}{[1+(\kappa/\kappa_e)^2]^{17/6}} e^{[-2(\kappa/\kappa_\eta)^2]} \\
\kappa &= (\kappa_i\kappa_i)^{1/2}, \quad \kappa_\eta = \varepsilon^{1/4}\nu^{-3/4}
\end{aligned} \tag{10.4}$$

The coefficient c_E is obtained by integrating the energy spectrum over all wavenumbers to get the turbulent kinetic energy, i.e.

$$k = \int_0^\infty E(\kappa)d\kappa \tag{10.5}$$

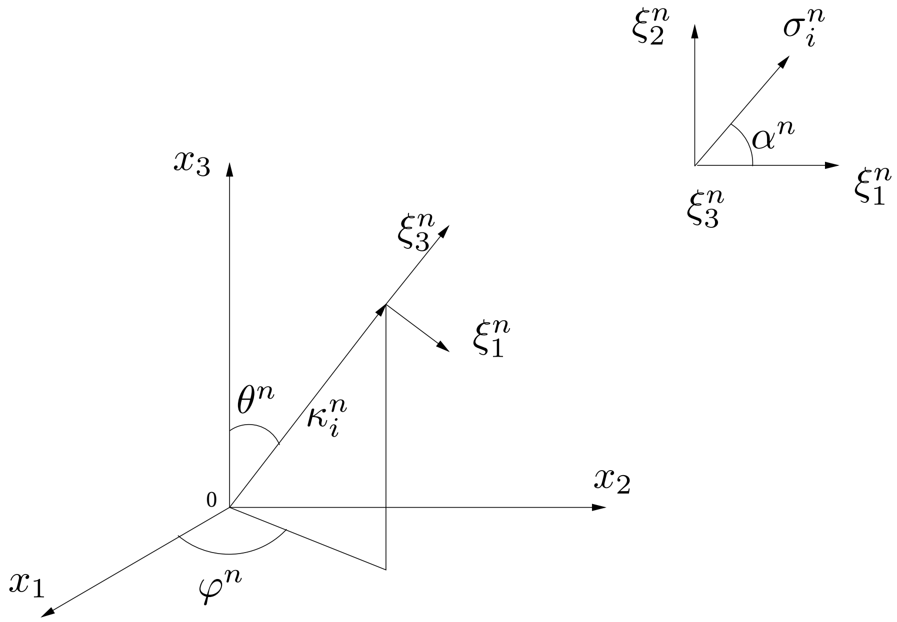


Figure 10.1: The wave-number vector, κ_i^n, and the velocity unit vector, σ_i^n, are orthogonal (in physical space) for each wave number n.

which gives [14]

$$c_E = \frac{4}{\sqrt{\pi}} \frac{\Gamma(17/6)}{\Gamma(1/3)} \simeq 1.453 \tag{10.6}$$

where

$$\Gamma(z) = \int_0^\infty e^{-z'} x^{z-1} dz' \tag{10.7}$$

10.7 Computing the fluctuations

Having $\hat{u}^n$, κ_j^n, σ_i^n and ψ^n, allows the expression in Eq. 10.3 to be computed, i.e.

$$\begin{aligned}
v_1' &= 2\sum_{n=1}^N \hat{u}^n \cos(\beta^n)\sigma_1 \\
v_2' &= 2\sum_{n=1}^N \hat{u}^n \cos(\beta^n)\sigma_2 \\
v_3' &= 2\sum_{n=1}^N \hat{u}^n \cos(\beta^n)\sigma_3 \\
\beta^n &= k_1^n x_1 + k_2^n x_2 + k_3^n x_3 + \psi^n
\end{aligned} \tag{10.8}$$

where $\hat{u}^n$ is computed from Eq. 10.4.

In this way inlet fluctuating velocity fields (v_1', v_2', v_3') are created at the inlet $x_2 - x_3$ plane.

10.8 Introducing time correlation

A fluctuating velocity field is generated each time step as described above. They are independent of each other and their time correlation will thus be zero. This

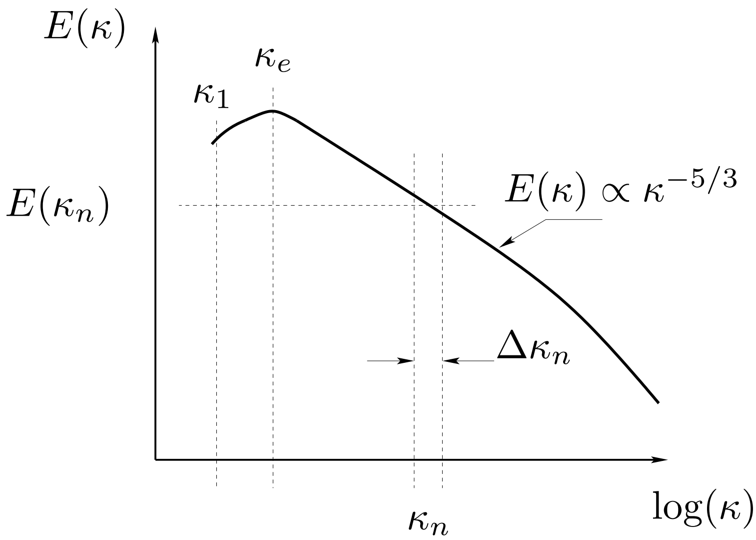


Figure 10.2: Modified von Kármán spectrum

is nonphysical. To create correlation in time, new fluctuating velocity fields, $\mathcal{V}'_1$, $\mathcal{V}'_2$, $\mathcal{V}'_3$, are computed based on an asymmetric time filter

$$\begin{aligned}(\mathcal{V}'_1)_m &= a(\mathcal{V}'_1)_{m-1} + b(v'_1)_m \\ (\mathcal{V}'_2)_m &= a(\mathcal{V}'_2)_{m-1} + b(v'_2)_m \\ (\mathcal{V}'_3)_m &= a(\mathcal{V}'_3)_{m-1} + b(v'_3)_m \end{aligned} \tag{10.9}$$

where m denotes the time step number and

$$a = \exp(-\Delta t/T_{int}) \tag{10.10}$$

where Δt and T_{int} denote the computational time step and the integral time scale, respectively. The integral time scale is here set at $T_{ont} = L_{int}/u_\tau$. The second coefficient is taken as

$$b = (1 - a^2)^{0.5} \tag{10.11}$$

which ensures that $\langle \mathcal{V}'^2_1 \rangle = \langle v'^2_1 \rangle$ ($\langle \cdot \rangle$ denotes averaging). The time correlation of will be equal to

$$\exp(-\hat{t}/T_{int}) \tag{10.12}$$

where $\hat{t}$ is the time separation and thus Eq. 10.9 is a convenient way to prescribe the turbulent time scale of the fluctuations. For more detail, see Section 10.8. The inlet boundary conditions are prescribed as (we assume that the inlet is located at $x_1 = 0$ and that the mean velocity is constant in the spanwise direction, x_3)

$$\begin{aligned}\bar{v}_1(0, x_2, x_3, t) &= V_{1,in}(x_2) + v'_{1,in}(x_2, x_3, t) \\ \bar{v}_2(0, x_2, x_3, t) &= V_{2,in}(x_2) + v'_{2,in}(x_2, x_3, t) \\ \bar{v}_3(0, x_2, x_3, t) &= V_{3,in}(x_2) + v'_{3,in}(x_2, x_3, t) \end{aligned} \tag{10.13}$$

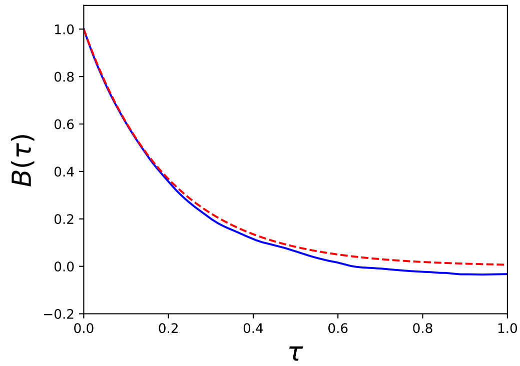


Figure 10.3: Auto correlation, $B(\tau) = \langle v_1'(t)v_1'(t-\tau)\rangle_t$ (averaged over time, t). —— : Eq. 10.12; - - : computed from synthetic data, $(\mathcal{V}_1')^m$, see Eq. 10.9.

where $v_{1,in}' = (\mathcal{V}_1')_m$, $v_{2,in}' = (\mathcal{V}_2')_m$ and $v_{3,in}' = (\mathcal{V}_3')_m$ (see Eq. 10.9). The mean inlet profiles, $V_{1,in}$, $V_{2,in}$, $V_{3,in}$, are either taken from experimental data, a RANS solution or from the law of the wall; for example, if $V_{2,in} = V_{3,in} = 0$ we can estimate $V_{1,in}$ as [22]

$$V_{1,in}^+ = \begin{cases} x_2^+ & x_2^+ \leq 5 \\ -3.05 + 5\ln(x_2^+) & 5 < x_2^+ < 30 \\ \frac{1}{\kappa}\ln(x_2^+) + B & x_2^+ \geq 30 \end{cases} \tag{10.14}$$

where $\kappa = 0.4$ and $B = 5.2$.

The method to prescribed fluctuating inlet boundary conditions have been used for channel flow [5], for diffusor flow [8] as well as for the flow over a bump and an axisymmetric hill [9].

The time correlation is implemented in module `modify_inlet`.

11 Procedure to generate anisotropic synthetic fluctuations

The methodology is as follows:

1. A pre-cursor RANS simulation is made using a RANS model, see Section 16.

2. After having carried out the pre-cursor RANS simulation, the Reynolds stress tensor is computed using the EARSM model [21].

3. The Reynolds stress tensor is used as input for generating the anisotropic synthetic fluctuations. The integral length scale, L_{int}, need to be prescribed; it can be set to $0.1\delta < L_{int} < 0.3\delta$, where δ denotes half-channel width.

4. Since the method of synthetic turbulence fluctuations assumes homogeneous turbulence, we can only use the Reynolds stress tensor in one point. We need to choose a relevant location for the Reynolds stress tensor. In a turbulent boundary layer, the Reynolds shear stress is by far the most important stress component. Hence, the Reynolds stress tensor is taken at the location where the magnitude of the turbulent shear stress is largest.

5. Finally, the synthetic fluctuations are scaled with $\left(|\overline{u'v'}|/|\overline{u'v'}|_{max}\right)^{1/2}_{RANS}$, which is taken from the 1D RANS simulation.
   This is done in module `modify inlet`.

The only constant used when generating these synthetic simulations is the prescribed integral length scale.

12 Flow Chart

To view flowchart, click on the link below and then Packages/pyCALC-LES/def main()

pyCALC-LES flowchart

13 Modules

13.1 bc_outlet_bc

Neumann outlet boundary conditions are set.

13.2 calceps_ls

Source terms in the ε equation (Launder & Sharma model) are computed, see Section 8. When PANS is used, `pans=True`, f_k is computed in module `compute_fk`. Otherwise it is set to one. The user can define additional source terms in `modify_eps`.

13.3 calck_kom

Source terms in the k equation (Wilcox model) are computed, see Section 9. When DES is used, C_{DES} is computed (it is stored in `fk3d`). The user can define additional source terms in `modify_k`.

13.4 calck_ls

Source terms in the k equation (Launder & Sharma model) are computed, see Section 8. The user can define additional source terms in `modify_k`.

13.5 calcom

Source terms in the ω equation (Wilcox model) are computed, see Section 9. The user can define additional source terms in `modify_om`.

13.6 calcp

Coefficients in the $\bar{p}$ equation (Eq. 4.4). It is a diffusion equation and hence the coefficients $a_w, a_E \ldots$ are computed in the same was as in Section 2.4 (with the diffusion coefficient Γ set to one).

13.7 calcu

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

13.8 calcv

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

13.9 calcw

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

13.10 coeff

The coefficient $a_w, a_E, a_S, a_N, a_L, a_H$ are computed. There are three different discretization schemes: central differencing scheme (CDS) first-order upwind and the hybrid scheme (first-order upwind and CDS)

13.11 compute_face_phi

Compute the face values of a variable.

13.12 compute_fk

Computes f_k (array `f3kd`) from Eq. 8.4.

13.13 compute_inlet_fluct

Compute synthetic fluctuations (see Section 10.1)

13.14 conv

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

13.15 correct_conv

After the Poisson equation for pressure has been solved, the convections `convwconvs` and `convl` (which are defined at the control volume faces) are corrected so as to satisfy continuity, see Eq. 4.5.

13.16 crank_nicol

Modification of the coefficients $a_w, a_E, \ldots$ due to time integration of the convective and diffusion made, see Section 2.1.1.

13.17 dphidx, dphidy, dphidz

The derivative on x_1, x_2 or x_3 direction are computed, see Section 1.2.

13.18 init

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

13.19 modify_eps, modify_k, modify_om, modify_u, modify_v, modify_w

The sources `su3d` and `sp3d` can be modified for the ε, k, ω, $\bar{u}$, $\bar{v}$ and $\bar{w}$ equations.

13.20 File modify_case.py

This file includes `modify_eps`, `modify_k`, ... `modify_w`.

13.21 modify_init

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

13.22 print_indata

Prints the indata set by the user.

13.23 read_restart_data

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

- `u3d_saved.npy, v3d_saved.npy, w3d_saved.npy, p3d_saved.npy, k3d_saved.npy, eps3d_saved.npy, om3d_saved.npy`

are read from a previous simulation.

13.24 save_data

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

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

are stored in the files

- `u3d_saved.npy, v3d_saved.npy, w3d_saved.npy, p3d_saved.npy, k3d_saved.npy, eps3d_saved.npy, om3d_saved.npy`.

13.25 save_time_aver_data

This module is called when every `itstep_save` timestep when `itstep` $\geq$ `itstep_start`. Time-averaged data of the

- $\bar{u}$, $\bar{v}$, $\bar{w}$, $\bar{p}$, k, f_k, ε, ω, $\nu_t + \nu$, $\bar{u}^2$, $\bar{v}^2$, $\bar{w}^2$ and $\bar{u}\bar{v}$

are stored in the files

- `u_averaged, v_averaged, w_averaged, p_averaged, k_averaged, fk_averaged, k_averaged, om_averaged, vis_averaged, eps_averaged, k3d_averaged, uu_stress, vv_stress, ww_stress, uv_stress`

13.26 File setup_case.py

In this file the user sets up the case (timestep, turbulence model, turbulence constants, type of boundary condition, convergence criteria, etc)

13.27 solve_3d

This module is used for all variables except pressure, $\bar{p}$. With the coefficient arrays `aw3d, ae3d, as3d, ...` a sparse matrix is created, `A`. The equation system is solved using the Python solver `linalg.lgmres` or `linalg.gmres`.

13.28 solve_p

This module is used for the pressure, $\bar{p}$. With the coefficient arrays `aw3d, ae3d, as3d, ...` a sparse matrix is created, `Ap`. At the first timestep and iteration, the multigrid hierarchy is constructed using `pyamg.ruge_stuben_solver` (recall that the coefficient arrays `aw3d, ae3d, as3d, ...` do not change since they are defined by geometrical quantities). The equation system is solved using the pyAMG solver `Ap.solve`. The user can choose relaxation method at each MG level with the variable `amg_relax` ('default' or 'cg').

13.29 synt_fluct

The synthetic fluctuations are computed and scaled with `uv_rans`, see Section 10.1

13.30 time_stats

Time-averaged quantities are created such as time-averaged velocities, pressure, resolved stresses etc. This module is called every `itstep_stats` timestep when `itstep` $\geq$ `itstep_start`.

13.31 update

At the end of each timestep, all variables are updated, i.e. `u3d_old=u3d`, `v3d_old=v3d`, etc.

13.32 vist_kom

The turbulent viscosity is computed using the $k - \omega$ model, see Section 9

13.33 vist_pans

The turbulent viscosity is computed using the Launder-Sharma $k-\varepsilon$ model, see Section 8.

13.34 vist_smag

The turbulent viscosity is computed using the Smagorinsky model, see Section 6.

13.35 vist_wale

The turbulent viscosity is computed using the WALE model, see Section 7.

14 DNS of fully-developed channel flow at $Re_\tau = 500$

To follow the execution of **pyCALC-LES** , it is recommended to start reading at the line *the execution of the code starts here*. To find where the time stepping starts, look for the line *start of time stepping*.

The grid is created using the script `generate-channel-grid.py`. The number of cells is set to $ni = nj = 96$. The extent of the grid in x and y direction is 3.2 and 2 respectively. The grid is stretched by 9% from both walls.

```
import numpy as np
import sys
   ni=96
   nj=96
   yfac=1.09 # stretching
   ymax=2
   xmax=3.2
   viscos=1/500
   dy=0.1
   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 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)
   np.savetxt('x2d.dat', x2d)
```

The grid in the z direction is read from file `z.dat`

```
zmax, nk=np.loadtxt('z.dat')
nk=np.int(nk)
dz=zmax/nk
```

and the `z.dat` file reads

```
1.6 96
```

The case is defined in modules `setup_case` and `modify_case`. They are located in a directory with the name `channel-500` (or something similar). Enter this directory.

14.1 `setup_case`

This module consists of 10 sections.

14.1.1 Section 1

We choose the central-differencing scheme for convection

```
scheme='c'
```

We use Crank-Nicolson for time discretization of the convection terms and for pressure we use fully implicit

```
acrank=1.0  # for pressure gradient
acrank_conv=0.5  # for convection-diffusion
```

The fully implicit discretization stabilizes the simulation and makes it possible to use only one global iteration.

14.1.2 Section 3

We take initial conditions from a previous simulation (`restart=True`) and we also save the new results to disk (`save=True`) which can be used as initial condition for next simulation.

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

The restart file used as initial condition may be created as in Section 18.

14.1.3 Section 4

The viscosity is set.

```
viscos=1/500
```

14.1.4 Section 6

The maximum number of global iterations is set to 5. We allow the solver to do only one iteration (`min_iter=1`). For the hill flow (see Section 17), the code diverges when `min_iter=1` and we must then force the solver to do at least two iterations.

The default relaxation method is chosen for the AMG solver for pressure and the convergence level in the AMG solver is set to $5 \cdot 10^{-4}$.

The 'lgmres' sparse matrix solver in Python is set for $\bar{u}$, $\bar{v}$ and $\bar{w}$.

In the Python solver for the velocities, the maximum number of iterations is set to 50 and the convergence level to 10^{-5}.

```
maxit=5
min_iter=1
sormax=1e-3
amg_relax='default'
solver_vel='lgmres'
nsweep_vel=50
convergence_limit_vel=1e-5
convergence_limit_p=5e-4
```

The convergence limit in the Python solvers is defined as

$$|Ax - b|/|b| < \gamma \tag{14.1}$$

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

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

14.1.5 Section 7

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

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

14.1.6 Section 8

We use 15000 timesteps. Time-averaging starts after 7500 timesteps. The timestep is set to $0.25\Delta x/U_{in}$ where U_{in} is an estimated bulk velocity. The instantaneous and time-averaged fields are saved to disk every 2000 timesteps. When time-averaging, we use every 10^{th} timestep.

```
ntstep=15000
uin=20
dt=0.25*(x2d[1,0]-x2d[0,0])*np.ones(ntstep)/uin
itstep_start=7500
itstep_save=2000
itstep_stats=10
```

14.1.7 Section 9

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

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

14.1.8 Section 10

The boundary conditions are set here. We have cyclic boundary conditions in x and z directions and hence

```
cyclic_x = True
cyclic_z = True
```

The south and north boundaries we define as walls (Dirichlet)

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

and the value for all variables is set to zero

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

Note that we don't need to set and type boundary conditions for west, east and high/low boundaries since they are defined by the cyclic boundary conditions

14.2 modify_case.py

Initial condition and additional boundary conditions are set in this file. It includes a module which are called for every flowfield variable, i.e. `modify_u`, `modify_v`, `modify_w`, `modify_p`, `modify_k`, `modify_eps` 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`)

14.2.1 modify_u

The only boundary conditions we need to set is the prescribed driving pressure gradient in the $\bar{u}$ equation.

```
su3d= su3d+vol
```

14.3 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 ../synt_fluct.py \
../pyCALC-LES.py  > exec-pyCALC-LES.py;
/usr/bin/time -oa out python -u  exec-pyCALC-LES.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-LES.py` and then executes it. Now run the code with the command

```
run-python &
```

Convergence history etc are written to the file `out`. 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
```

We can also check that the Python sparse matrix reduces the residuals. Check this with

```
grep resid out
```

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

Plot the results using the script `pl_uvw.py`

15 Fully-developed channel flow at $Re_\tau = 5\,200$ using $k-\omega$ DES

You find `setup_case.py` and `modify_case.py` in a directory with the name `channel_5200-k-omega-DES` (or something similar). Go into this directory. The grid is generated with the script `generate-channel-grid.py`. It is stretched by 15% in the y direction and the extent in the x direction is set to 3.14 with 32 cells.

32 cells are also used in the z direction with and extent of 1.6. The `z.dat` reads

```
1.6 32
```

Here we comment only on differences compared to the DNS flow in Section 14.

15.1 `setup_case`

15.1.1 Section 1

We choose the first-order upwind scheme for the k and ε equations.

```
scheme_turb='u'
```

We use also first-order time discretization for k and ω

```
acrank_conv_kom=1
```

15.1.2 Section 2

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

```
kom_des = True
```

The turbulence constants are set to

```
cmu=0.09
c_omega_1= 5./9.
c_omega_2=3./40.
prand_omega=2.0
prand_k=2.0
```

15.1.3 Section 5

The under-relaxation factor for turbulent viscosity is set to 0.5.

```
urfvis=0.5
```

15.1.4 Section 6

The convergence criteria for the Python solver is set for k and ω and the lgmres solver is selected.

```
convergence_limit_k=1e-4
convergence_limit_om=1e-8
solver_turb='lgmres'
```

Note that a rather strict limit is set for ω which seems to be necessary. The reason may be that the norm of the right-side, $|b| \equiv |S_U|$ (see Eq. 14.1) is large near the walls which makes the error $|Ax - b|/|b|$ small. An alternative is to use the gmres solver (which is selected in Section 20.

15.1.5 Section 10

The wall-boundary conditions for k and ω are set as $k = 0$ and ω as below (see Eq. 9.2).

```
# boundary conditions for k
   k_bc_south=np.zeros((ni,nk))
   k_bc_north=np.zeros((ni,nk))

   k_bc_south_type='d'
   k_bc_north_type='d'

# boundary conditions for omega
   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/c_omega_2/dist2_s

# make it 2D
   om_bc_south=np.repeat(om_bc_south[:,None], repeats=nk, axis=1)

   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/c_omega_2/dist2_n

# make it 2D
   om_bc_north=np.repeat(om_bc_north[:,None], repeats=nk, axis=1)

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

15.2 modify_case.py

No changes are made compared to Section 14.

16 RANS of channel flow at $Re_\tau = 5\,200$ using $k-\omega$

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

We generate a new grid. We take the same grid in the y direction as in Section 14, but in the x direction we set three cells, `ni=3`, and `xmax=1` (this is the minimum number of cells we can use when `cyclic_x=True`). In the z direction we set domain size to one and use two cells; the `z.dat` is modified to `1, 2`. The grid is created using the script `generate-channel-grid.py`.

Here we comment only on differences compared to the DES flow in Section 15.

16.1 `setup_case`

16.1.1 Section 1

Since we will simulated a time-marching flow towards steady conditions we choose the hybrid scheme for the velocities.

```
scheme='h'
```

16.1.2 Section 2

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

```
kom = True
kom_des = False
```

16.1.3 Section 3

We don't start from a previous solution.

```
restart = False
```

16.1.4 Section 8

we increase the timestep.

```
dt=0.9*(x2d[1,0]-x2d[0,0])*np.ones(ntstep)/uin
```

and we use 6000 and time average during the last 100 timesteps

```
ntstep=6000
itstep_start=5900
```

16.1.5 Section 10

We do not use cyclic boundary conditions in the z direction.

```
cyclic_z=False
```

In the z direction we set Neumann boundary condition for all variables except $\bar{w}$ (which is set to zero) .

```
u_bc_z_type='n'
v_bc_z_type='n'
w_bc_z_type='d'
k_bc_z_type='n'
om_bc_z_type='n'
w_bc_z=0
```

16.2 modify_case.py

No changes are made compared to Section 15.

17 Periodic flow over a 2D hill using PANS

In this section we present the flow over many 2D hills. We define the case as one hill with periodic boundary conditions in x. The flow is also periodic on the z direction. The PANS model (see Section 8) is used together with the Launder-Sharma model as the baseline RANS model. The Yap correction is used in the RANS region.

The test case is presented at Erfoctac. The mesh has 160×80 cells in the $x - y$ plane and 32 cells in the z direction with $x_{max} = 4.5$.

Below we comment only on differences compared to the DNS flow in Section 15.

17.1 setup_case

17.1.1 Section 1

We use the hybrid spatial discretization scheme and the first-order time discretization for k and ε

```
scheme='h'
acrank_conv_keps=1
```

17.1.2 Section 2

The PANS model is selected

```
pans = True
```

17.1.3 Section 4

The Reynolds number is set to $Re = 10500$ based on the bulk velocity (equal to one) and the height of the channel at the hill crest (equal to one).

```
viscos=1/10500
```

17.1.4 Section 6

For this flow we must do at least two global iterations. If not, the solution diverges.

```
min_iter=2
```

The convergence level for the Python sparse-matrix solver for the ε equation is set to 10^{-4}.

```
convergence_limit_eps=1e-4
```

17.1.5 Section 8

Number of timesteps is set to 15000 and time averaging starts after 7500 timesteps. The timestep is set to $0.2\Delta x/U_{in}$ where U_{in} is the bulk velocity the hill crest.

```
ntstep=15000
uin=1
dt=0.2*(x2d[1,0]-x2d[0,0])*np.ones(ntstep)/uin
itstep_start=7500
```

17.2 modify_case.py

17.2.1 modify_u

We compute the driving pressure gradient from a balance of all forces on the surfaces, i.e. wall shear stresses and pressure force. For more details, see Section 3.5 in Irannezhad [15].

First, compute the viscous forces at the walls,

```
taus=np.sum(viscos*as_bound*u3d[:,0,:])
taun=np.sum(viscos*an_bound*u3d[:,-1,:])
```

Next, compute the force in the x direction due to pressure on the lower wall and the total force.

```
sumps=np.sum(p3d[:,0,:]*areasx[:,0,:])
total_forces=taus+taun+sumps
```

Compute the total volume of the domain and the bulk velocity at the hill crest. The target bulk velocity is one.

```
sumvol=np.sum(vol)
uin=np.sum(convw[0,:,:])/(y2d[0,-1]-y2d[0,0])/zmax
```

Finally, compute the required driving pressure gradient, `beta`, and add it as a volume source (in the $\bar{u}$ equation).

```
beta=total_forces/sumvol
su3d=su3d+beta*vol
```

17.2.2 modify_eps

Here we add the Yap term.

```
   global f
#  add Yap term
# Yap =0.83*(k^1/5/(eps*cl*yb) -1)*(k^1/5/(eps(cl*yb)^2*eps^2/k
# su= 0.83*(k^1/5/(eps*cl*yb)*(k^1/5/(eps(cl*yb)^3*eps^2/k = 0.83*t1^3 *eps^2/k
# sp=-0.83*                   (k^1/5/(eps(cl*yb)^2*eps/k   = 0.83*t1^2 *eps  /k

   cl=2.6
   t1=k3d**1.5/(eps3d*cl*dist3d)
# include the term only in the RANS region where fk=1
   f=np.where(fk3d >= 1,1,0)
   su3d=su3d+0.83*t1**3*eps3d**2/k3d*vol*f
   sp3d=sp3d-0.83*t1**2*eps3d/k3d*vol*f
```

The `f` array is set to zero when $f_k > 1$ (i.e. LES mode).
Run the code and then plot the results using the script `plot_hill.py`.

18 Synthetic turbulence at inlet: Channel flow at $Re_\tau = 395$

Here we will simulate the flow in a channel at $Re_\tau = 395$. At the inlet, we prescribe mean flow velocity obtained from a 1D RANS simulation with the $k-\omega$ model, see Section 16. Synthetic fluctuations are superimposed on the mean flow. To create the anisotropy, we need the eigenvalues and the eigenvectors of a Reynolds stress tensor which is taken from the EARSM model. The Reynolds stress tensor is taken at the cell where $|\overline{v_1'v_2'}|$ is maximum. The eigenvectors and the eigenvalues are created with the script `compute_a_and_R-from-earsm.py`. This script generates two files, `R.dat` which includes the eigenvectors and `a.dat` which includes the eigenvalues. The two files are read in module `synt_fluct`. Finally, the synthetic fluctuations are scaled with the shear stress from the 1D RANS simulation.

Below, we highlight the differences compared to Section 15.

18.1 setup_case

18.1.1 Section 2

We choose the WALE turbulence model

```
wale = True
```

18.1.2 Section 3

No restart.

```
restart = False
```

18.1.3 Section 4

Reynolds number $Re_\tau = 395$

```
viscos=1/395
```

18.1.4 Section 6

We choose the default relaxation method for the AMG solver of the Poisson pressure equation.

```
amg_relax='default'
```

18.1.5 Section 10

We will use inlet-outlet boundary conditions. Hence, no cyclic boundary conditions in the x direction.

```
cyclic_x = False
```

We will use synthetic fluctuations at the inlet. We set the lengthscale of the synthetic fluctuations to $L_t = 0.2$ (see Section 10.4) and the number of modes (see Section 10.5) to 150.

```
L_t_synt=0.2
nmodes_synt=150
```

The Reynold stress tensor of the generated time-averaged anisotropic fluctuations is equal to the prescribed Reynolds stress tensor, see Item 2 in Section 11. In this case, it gives a negative shear stress which is correct for the lower half of the channel. But for the upper half of the channel it should be positive, This is fixed by switching the sign of the synthetic fluctuation in the y direction. The variable `jmirror_synt` tells **pyCALC-LES** where to switch sign. We want to switch sign for $j > nj/2$ and hence we set

```
jmirror_synt=np.int(nj/2)
```

18.2 modify_case.py

18.2.1 modify_init

Here we set initial conditions. We use the 1D RANS data, see Section 16. We read $\bar{u}$

```
   data=np.loadtxt('y_u_k_om_uv_395.dat')
   u_rans=data[:,1]
# make it 2D
   u_rans=np.repeat(u_rans[:,None], repeats=nk, axis=1)
# set inlet field in entre domain
   u3d=np.repeat(u_rans[None,:,:], repeats=ni, axis=0)
```

18.2.2 modify_inlet

Inlet boundary conditions are set here. At the first timestep, we read the 1D RANS solution for $\bar{u}$ and $\overline{v'_1 v'_2}$

```
   if itstep == 0:
      y_u_k_om=np.loadtxt('y_u_k_om_uv_395.dat')
      y_rans=y_u_k_om[:,0]
      u_rans=y_u_k_om[:,1]
# make it 2D
      u_rans=np.repeat(u_rans[:,None], repeats=nk, axis=1)

      uv_rans=np.abs(y_u_k_om[:,4])
```

A grid in the z direction is created and we call `synt_fluct` to generate the synthetic fluctuations, see Eq. 10.8.

```
      zp = np.linspace(0, zmax, nk)
      usynt,vsynt,wsynt=synt_fluct(nmodes_synt,itstep,L_t_synt,y_rans,zp,\
         uv_rans,viscos,jmirror_synt)
```

We want to make sure that the average of the streamwise fluctuation is zero, i.e. $\langle u' \rangle = 0$, Hence we subtract its mean

```
      uinc=np.sum(usynt*areaw[0,:,:])/(y2d[0,-1]-y2d[0,0])/zmax
      usynt=usynt-uinc
```

Next, we set the initial fields of $\mathcal{V}'_3$, $\mathcal{V}'_2$ and $\mathcal{V}'_3$ (see Eq. 10.13) and compute a and b (see Eqs. 10.10 and 10.11).

```
      usynt_inlet=usynt
      vsynt_inlet=vsynt
      wsynt_inlet=wsynt
# tturb from ustar=1
      tturb=L_t_synt/1
      a_synt=np.exp(-dt[itstep]/tturb)
      b_synt=(1.-a_synt**2)**0.5
```

For timestep higher than zero, we call `synt_fluct`, correct u' and make the time filtering in Eq. 10.13

```
   usynt,vsynt,wsynt=synt_fluct(nmodes_synt,itstep,L_t_synt,y_rans,zp,\
      uv_rans,viscos,jmirror_synt)
# correct usynt so that it is = 0 (easier to converge the p solver)
   uinc=np.sum(usynt*areaw[0,:,:])/(y2d[0,-1]-y2d[0,0])/zmax
   usynt=usynt-uinc
   usynt_inlet=a_synt*usynt_inlet+b_synt*usynt
   vsynt_inlet=a_synt*vsynt_inlet+b_synt*vsynt
   wsynt_inlet=a_synt*wsynt_inlet+b_synt*wsynt
```

Finally, we superimpose the synthetic fluctuations to the mean flow and store the inlet fields in `u_bc_west`, `v_bc_west` and `w_bc_west` which are returned as a results from the `modify_inlet`

```
u_bc_west=u_rans+usynt_inlet
v_bc_west=vsynt_inlet
w_bc_west=wsynt_inlet
```

18.2.3 modify_u

Add the inlet convective flow to source terms

```
su3d[0,:,:]= su3d[0,:,:]+convw[0,:,:]*u_bc_west
sp3d[0,:,:]= sp3d[0,:,:]-convw[0,:,:]
vist=vis3d[0,:,:]-viscos
sp3d[0,:,:]=sp3d[0,:,:]-vist*aw_bound
su3d[0,:,:]=su3d[0,:,:]+vist*aw_bound*u_bc_west
```

Note that the viscous diffusive part is added in module `bc`.

18.2.4 modify_v

Same as in `modify_u`

```
su3d[0,:,:]= su3d[0,:,:]+convw[0,:,:]*v_bc_west
sp3d[0,:,:]= sp3d[0,:,:]-convw[0,:,:]
vist=vis3d[0,:,:]-viscos
sp3d[0,:,:]=sp3d[0,:,:]-vist*aw_bound
su3d[0,:,:]=su3d[0,:,:]+vist*aw_bound*v_bc_west
```

18.2.5 modify_w

Same as in `modify_u`

```
su3d[0,:,:]= su3d[0,:,:]+convw[0,:,:]*w_bc_west
sp3d[0,:,:]= sp3d[0,:,:]-convw[0,:,:]
vist=vis3d[0,:,:]-viscos
sp3d[0,:,:]=sp3d[0,:,:]-vist*aw_bound
su3d[0,:,:]=su3d[0,:,:]+vist*aw_bound*w_bc_west
```

18.2.6 modify_outlet

This outlet boundary condition is described in Section 5.2. First, compute inlet and outlet volume flow as well as the outlet area.

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

Next, compare global inflow and outflow, compute a corrective velocity, `uinc` and correct the convective fluxes so that global balance is satisfied.

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

Note that Neumann boundary conditions are set for $\bar{u}$, $\bar{v}$, ... since

```
phi_bc_east_type='n'
```

for all variables.

Run the code and plot the results with the script `plot_inlet`.

19 Synthetic turbulence at inlet: Channel flow at $Re_\tau = 5200$

Here we will simulate the flow in a channel at $Re_\tau = 5200$. We use the $k-\omega$ DES turbulence model. The grid in the y and z direction is used as in Section 15. The number of cells and extent in the x direction are 96 and 9 (constant grid spacing), respectively.

Below, we highlight the differences compared to Section 18.

19.1 `setup_case`

19.1.1 Section 2

We select the $k-\omega$ DES model.

```
kom_des = True
```

The interface is automatically computed

```
jl0 = 0
```

19.1.2 Section 4

The Reynolds number is set to 5200.

```
viscos=1/5200
```

19.1.3 Section 6

The convergence levels in the Python sparse-matrix solver are set.

```
convergence_limit_k=1e-4
convergence_limit_om=1e-8
```

19.1.4 Section 10

The boundary conditions for k and ω at the walls are set.

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

   k_bc_south_type='d'
   k_bc_north_type='d'

# boundary conditions for omega
```

```
   om_bc_south=np.zeros((ni,nk))
   om_bc_north=np.zeros((ni,nk))

   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

# make it 2D
   om_bc_south=np.repeat(om_bc_south[:,None], repeats=nk, axis=1)

   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

# make it 2D
   om_bc_north=np.repeat(om_bc_north[:,None], repeats=nk, axis=1)
```

19.2 modify_case.py

19.2.1 modify_init

Here we set initial conditions. We use the 1D RANS data, see Section 16. We read $\bar{u}$, k and ω. k_{init} is set to 20% of the RANS value and ω_{iniy} is set to $k_{init}^{1/2}/(0.01\Delta_{max})$.

```
   data=np.loadtxt('y_u_k_om_uv_5200_nj96.txt')
   u_rans=data[:,1]
# make it 2D
   u_rans=np.repeat(u_rans[:,None], repeats=nk, axis=1)

   k_rans=data[:,2]
# make it 2D
   k_rans=np.repeat(k_rans[:,None], repeats=nk, axis=1)

   om_rans=data[:,3]
# make it 2D
   om_rans=np.repeat(om_rans[:,None], repeats=nk, axis=1)

# set inlet field in entre domain
   u3d=np.repeat(u_rans[None,:,:], repeats=ni, axis=0)
   k3d=0.2*np.repeat(k_rans[None,:,:], repeats=ni, axis=0)
   om3d=k3d**0.5/(0.01*delta_max)

   vis3d=k3d/om3d+viscos
```

19.2.2 modify_inlet

Here we set inlet boundary conditions. At the first timestep, we read mean inlet data from a 1D RANS solution

```
   if itstep == 0:
      y_u_k_om=np.loadtxt('y_u_k_om_uv_5200_nj96.txt')
      y_rans=y_u_k_om[:,0]
      u_rans=y_u_k_om[:,1]
# make it 2D
      u_rans=np.repeat(u_rans[:,None], repeats=nk, axis=1)
      k_rans=y_u_k_om[:,2]
# make it 2D
      k_rans=np.repeat(k_rans[:,None], repeats=nk, axis=1)
      eps_rans=y_u_k_om[:,3]
# make it 2D
      eps_rans=np.repeat(eps_rans[:,None], repeats=nk, axis=1)
      uv_rans=np.abs(y_u_k_om[:,4])

# store k and omega
      k_bc_west=k_rans
      om_bc_west=om_rans
```

Compared to Section we store also k and ω in `k_bc_west` and `om_bc_west`.

20 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.

We generate a new grid. The first cell is set to $\Delta t = 7.83 \cdot 10^{-4}$. We 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 $\Delta x = 0.05$ and then we stretch it by 1%. We set the number of cells to `ni=300`. In the z direction we set the number of cells to two and the extent to one, i.e. the `z.dat` is modified to `1, 2`. The grid is created using the script `generate-bound-layer-grid.py`.

Here we comment only on differences compared to the DES flow in Section 16.

20.1 setup_case

20.1.1 Section 1

Hybrid discretization is set for all variables.

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

20.1.2 Section 2

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

```
kom = True
kom_des = False
```

20.1.3 Section 4

The viscosity is set.

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

20.1.4 Section 6

The gmres solver is chosen for k and ω and the conjugated solver is selected for relaxation in the AMG solver.

```
solver_turb='gmres'
amg_relax='cg'
```

It is found that when the lgmres solver is used for k ans ω, the ω equation stops to converge after approximately 100 timesteps (the residuals are constant, i.e. the lgrres solver does not reduce the residuals of ω). Also, the solution diverges if the default relaxation solver is used in the AMG solver.

20.1.5 Section 8

The number of timesteps is set to 2000 and the results are time averaged the last 100 timesteps (the solution will be steady). A rather large timestep is chosen (we are not concerned about time accuracy since we time march to steady state).

```
ntstep=2000
uin=1
dt=4*(x2d[1,0]-x2d[0,0])*np.ones(ntstep)/uin
itstep_start=1900
```

20.1.6 Section 10

We do not use cyclic boundary conditions in the x and z directions.

```
cyclic_x = False
cyclic_z = False
```

At the north boundary we set Neumann boundary condition for all variables except $\bar{v}$ (which is set to zero) .

```
u_bc_north_type='n'
v_bc_north_type='d'
w_bc_north_type='n'
k_bc_north_type='n'
om_bc_north_type='n'
```

We use Neumann boundary condition in the z directions for all variables except $\bar{w}$ (which is set to zero) .

```
u_bc_z_type='n'
v_bc_z_type='n'
w_bc_z=0
k_bc_z_type='n'
om_bc_z_type='n'
```

20.2 modify_case.py

20.2.1 modify_init

Initial condition: set $\bar{u} = 1$, $k = 0.001$ and $\omega = 1$.

```
u3d=np.ones((ni,nj,nk))
k3d=np.ones((ni,nj,nk))*0.001
om3d=np.ones((ni,nj,nk))

vis3d=k3d/om3d+viscos
```

20.2.2 modify_inlet

Inlet boundary conditions same as initial conditions.

```
k_bc_west=np.ones((nj,nk))*1e-3
u_bc_west=np.ones((nj,nk))
om_bc_west=np.ones((nj,nk))
```

Run the code and plot the results with `plot_inlet_bound.py`

Now we will use these results as mean inlet boundary conditions in Section 21. Look at the script `create-inlet-rans-profiles.py`. Here we extract $\bar{u}$, $\bar{v}$, k, ω and $\overline{v_1'v_2'}$ at cells `ni-10`. The data are stored in file `y_u_v_k_om_uv_re-theta-2500.txt`.

21 DES of boundary layer flow using the $k - \omega$ model

Here we will do DES of a developing boundary layer. First, we create the mesh. The script `generate-bound-layer-grid.py` is used. The mesh in the y direction is the same as in Section 20. The first grid size in the x direction is set as $\Delta x = 0.1\delta_{in}$ with $\delta_{in} = 0.86$. This boundary layer thickness is found from the plot file in Section 20. the stretching of the grid in the x direction is set to 0.3%. The grid in the x direction is defined as `nk=32` and `zmax=0.9`; thus the `z.dat` file reads

```
0.9 32
```

Only the changes compared to Section 20 will commented below

21.1 setup_case

21.1.1 Section 1

We choose the central-differencing scheme for convection and Crank-Nicolson for time discretization.

```
scheme='c'
acrank_conv=0.5
```

21.1.2 Section 2

Choose $k-\omega$ DES model.

```
kom_des = True
kom = False
```

21.1.3 Section 6

Choose $k-\omega$ DES model. Set the number of timesteps to 150000 and average over the last 7500. Reduce the timestep (compared to Section 20).

```
ntstep=15000
uin=1
dt=0.25*(x2d[1,0]-x2d[0,0])*np.ones(ntstep)/uin
itstep_start=7500
```

21.1.4 Section 10

Use cyclic boundary conditions in the z direction

```
cyclic_z = True
```

21.2 modify_case.py

21.2.1 modify_init

Initial condition are set in the same way as in Section 19. Note that here, the bulk velocity is one, and hence $u_\tau \neq 1$. The RANS profiles (stored in file y_u_v_k_om_uv_re-theta-2500.txt) are taken from the RANS simulation, see Section 20.2.2.

```
  data=np.loadtxt('y_u_v_k_om_uv_re-theta-2500.txt')

   u_rans=data[:,1]
# make it 2D
   u_rans=np.repeat(u_rans[:,None], repeats=nk, axis=1)


   k_rans=data[:,3]
# make it 2D
   k_rans=np.repeat(k_rans[:,None], repeats=nk, axis=1)
```

```
   om_rans=data[:,4]
# make it 2D
   om_rans=np.repeat(om_rans[:,None], repeats=nk, axis=1)

# set inlet field in entre domain
   u3d=np.repeat(u_rans[None,:,:], repeats=ni, axis=0)
   k3d=0.2*np.repeat(k_rans[None,:,:], repeats=ni, axis=0)
   om3d=k3d**0.5/(0.01*delta_max)

   vis3d=k3d/om3d+viscos
```

21.2.2 modify_inlet

The inlet boundary conditions are set as in Section 19.2.2 except that we read data from another file, i.e.

```
y_u_k_om=np.loadtxt('y_u_v_k_om_uv_re-theta-2500.txt')
```

21.2.3 modify_u, modify_v, modify_w, modify_k, modify_om

The inlet boundary conditions are set in exactly the same way as in Section 19.2

Nomenclature

`acrank`: time integration scheme for pressure (1: fully implicit)

`acrank_conv`: time integration scheme for convection and diffusion in $\bar{u}$, $\bar{v}$ and $\bar{w}$ equations (1: fully implicit)

`acrank_conv_keps`: time integration scheme for convection and diffusion in k and ε equations (1: fully implicit)

`acrank_conv_kom`: time integration scheme for convection and diffusion in k and ω equations (1: fully implicit)

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

`amg_relax`: relation method in pyAMG: 'default','cg','gm','gmres','fgmres','cgne','cgnr','cr'

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

`apo3d`: a_P^o, see Eq. 2.5

`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

`areaz:` high and low area of control volume

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

`aw3d,ae3d,as3d,an3d,al3d,ah3d,ap3d:` discretization coefficients, a_W, a_E, a_S, a_N, a_L, a_H, a_P

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

`az_bound:` a_H and a_L coefficient for diffusion for high and low boundary (without viscosity)

`c_eps_1:` $C_{\varepsilon 1}$ coefficient in the $k-\varepsilon$ model

`c_eps_2:` $C_{\varepsilon 2}$ coefficient in the $k-\varepsilon$ model

`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_eps_1,c_eps_2:` turbulence model constants, C_μ, $C_{\varepsilon 1}$, $C_{\varepsilon 2}$

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

`convergence_limit_eps, 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_vel:` convergence limit in Python solver for $\bar{u}$, $\bar{v}$ and $\bar{w}$

`convw,convs,convl:` convection through west, south and low face

`cyclic_x:` cyclic boundary conditions in x direction

`cyclic_z:` cyclic boundary conditions in z direction

`delta_max:` $\max(\Delta x, \Delta y, \Delta z)$

`dist2d:` distance to the nearest wall (used in PANS)

`dist3d:` smallest distance to south or north wall

`dpdx_old, dpdy_old, dpdz_old:` pressure derivatives, $\partial\bar{p}/\partial x$, $\partial\bar{p}/\partial y$, $\partial\bar{p}/\partial z$ at old timestep

`dt:` timestep

`dz:` grid spacing in the z direction (the grid is equi-distant)

`eps3d:` modeled dissipation of turbulent kinetic energy, ε

`eps3d_mean:` time-averaged dissipation of turbulent kinetic energy, $\langle\varepsilon\rangle$

`eps_bc_east, eps_bc_north, eps_bc_south, eps_bc_west, eps_bc_z:` boundary values of ε at east, south, west, north, and high/low boundary `eps_bc_east_type, eps_bc_north_type, eps_bc_south_type,eps_bc_west_type, eps_bc_z_type:` type of b.c. for ε ('d'=Dirichlet or 'n'=Neumann')

`fk3d:` f_k, used in PANS and as F_{DEs} in $k-\omega$ DES

`fk3d_mean:` time-averaged f_k, $\langle f_k \rangle$

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

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

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

`iter:` current global iteration

`itstep:` current timestep

`itstep_save:` instantaneous and time-averaged field are saved on disk every `itstep_save` timestep

`itstep_start:` time averaging starts

`itstep_stats:` time averaging is done every `itstep_stats` timestep

`itstep_stats_counter:` counter for how many samples are used for time averaging

`jl0:` when `jl0 < 0`, the LES-RANS interface in the $k-\omega$ DES model is fixed cell `np.abs(jl0)`

`jmirror_synt:` the sign of the v synthetic are changed for nodes `j` $\geq$ `jmirror` (in module `synt_fluct`)

`k3d:` modeled turbulent kinetic energy, k

`k3d_mean:` time-averaged modeled turbulent kinetic energy, $\langle k \rangle$

`k_bc_east, k_bc_south, k_bc_west, k_bc_north, k_bc_z:` boundary values of k at east, south, west, north, and high/low boundary

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

`keps:` the Launder-Sharma $k-\varepsilon$ model is used (RANS)

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

`kom_des:` the DES Wilcox $k-\omega$ model is used

`L_t_synt:` lengthscale of the synthetic fluctuations, see Eq. 10.4

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

`maxit:` maximum number of global iterations at each timestep

`ni,nj,nk:` number of cell centers (including boundaries) in i, j and k direction

`ni,nj,nk:` number of cells in i, j and k direction `nmodes_synt`: number of modes when generating synthetic fluctuations

`nsweep_keps`: maximum number of iterations in the Python solver when solving the k and ε equations in solver called in `solve_3d`

`nsweep_kom`: maximum number of iterations in the Python solver when solving the k and $\bar{\omega}$ equations in solver called in `solve_3d`

`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_3d`

`ntstep`: number of timesteps

`om3d`: specific dissipation of turbulent kinetic energy, ω

`om3d_mean`: time-averaged modeled specific dissipation of turbulent kinetic energy, $\langle \omega \rangle$

`om_bc_east, om_bc_north, om_bc_south, om_bc_west, om_bc_z`: boundary values of ω at east, south, west, north, and high/low boundary

`om_bc_east_type, om_bc_north_type, om_bc_south_type, om_bc_west_type, om_bc_z_type`: type of b.c. for ω ('d'=Dirichlet or 'n'=Neumann')

`p3d`: pressure, $\bar{p}$

`p3d_mean`: time-averaged pressure, $\bar{p}$

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

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

`pans`: PANS (based on $k - \varepsilon$) is used

`prand_eps`: σ_ε, turbulent Prandtl number in the ε equation

`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_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 13.23

`save`: the $\bar{u}$, $\bar{v}$ ... fields are saved to disk, see Section 13.24

`scheme`: discretization scheme for the $\bar{u}$, $\bar{v}$ and $\bar{w}$ equation. 'c'=central, 'h'=hybrid, 'u'=upwind, see Section 13.10

`scheme_turb`: discretization scheme for k, ε and ω. 'c'=central, 'h'=hybrid, 'u'=upwind, see Section 13.10

`smag`: the Smagorinsky model is used

`solver_turb`: Python sparse matrix solver called in module `solve_3d` for k, ε and ω. solver_turb='gmres','lgmres','cgnr','cgne','fgmres'

`solver_vel`: Python sparse matrix solver called in module `solve_3d` for $\bar{u}$, $\bar{v}$ and $\bar{w}$. solver_vel='gmres','lgmres','cgnr','cgne','fgmres'

`sormax`: convergence criteria in outer iteration loop

`sp3d,su3d`: discretization source terms, S_p, S_U

`u3d`: $\bar{u}$ velocity

`u3d_mean`: time-averaged $\bar{u}$ velocity

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

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

`urfvis`: under-relaxation factor for turbulent viscosity

`usynt_inlet`: synthetic inlet fluctuation in the x direction, $(\mathcal{V}'_1)_m$, see 10.9

`uu3d_stress`: time-averaged resolved stress, $\langle \overline{v_1'^2} \rangle$

`uv3d_stress`: time-averaged resolved stress, $\langle \overline{v_1' v_2'} \rangle$

`v3d`: $\bar{v}$ velocity

`v3d_mean`: time-averaged $\bar{v}$ velocity

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

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

`vis3d`: total viscosity, $\nu + \nu_t$

`vis3d_mean`: time-averaged total viscosity, $\langle \nu_t + \nu \rangle$

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

`vol`: volume of a control volume

`vsynt_inlet`: synthetic inlet fluctuation in the y direction, $(\mathcal{V}'_2)_m$, see 10.9

`vv3d_stress`: time-averaged resolved stress, $\langle \overline{v_2'^2} \rangle$

`w3d`: $\bar{w}$ velocity

`w3d_mean`: time-averaged $\bar{w}$ velocity `w_bc_east, w_bc_north, w_bc_south, w_bc_west, w_bc_z`: boundary values of $\bar{w}$ at east, south, west, north, and high/low boundary

`w_bc_east_type, w_bc_north_type, w_bc_south_type, w_bc_west_type, w_bc_z_type`: type of b.c. for $\bar{w}$ ('d'=Dirichlet or 'n'=Neumann')

`wale`: the WALE model is used

`wsynt_inlet`: synthetic inlet fluctuation in the z direction, $(\mathcal{V}'_3)_m$, see 10.9

`ww3d_stress`: time-averaged resolved stress, $\langle \overline{v_3'^2} \rangle$

`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 the center a control volume, see Fig. 1.3

`zmax`: extent of the computational domain in the z direction

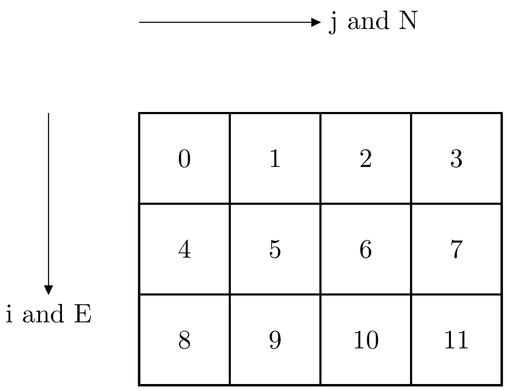
A Matrix solver and sparse matrix format in Python

pyCALC-LES uses the sparse solver available in Python. Hence the coefficients $a_W, a_E, a_S, a_N, a_L, a_H, a_O, S_u$ must be converted to Python's sparse matrix format. In 3D there will be 7 diagonals. When cyclic boundary conditions are used (`cyclic_x` and/or `cyclic_z`), there will be two additional diagonals for each cyclic boundary condition. This means that the cyclic boundary conditions are treated implicitly.

The Python solved `linalg.lgmres` is used for all variables except the pressure for which the algebraic multigrid solver pyAMG [19] is used.

Below, the full coefficient matrix is shown for a couple of cases with and without cyclic boundary conditions..

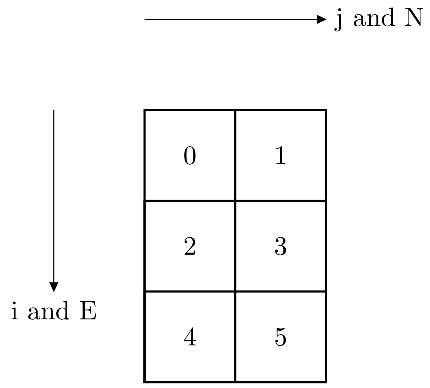
A.1 2D grid, $ni \times nj = (3, 4)$



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

Matrix for 2D flow. $ni \times nj = (3, 4)$. Cyclic in x. The coefficients due to cyclic boundary conditions are colored in blue.

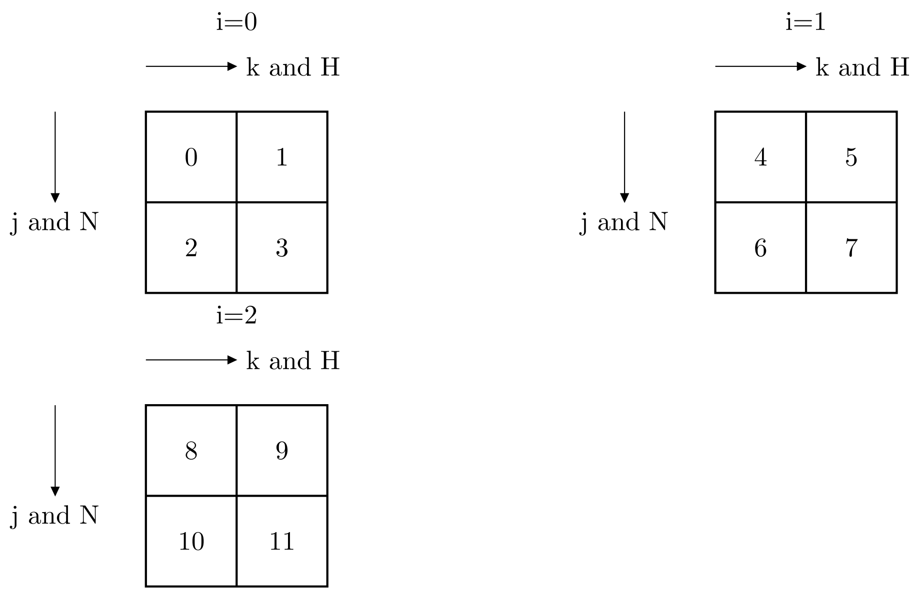
A.2 2D grid, $ni \times nj = (3,2)$



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

Matrix for 2D flow. $ni \times nj = (3,2)$. Cyclic in x. The coefficients due to cyclic boundary conditions are colored in blue.

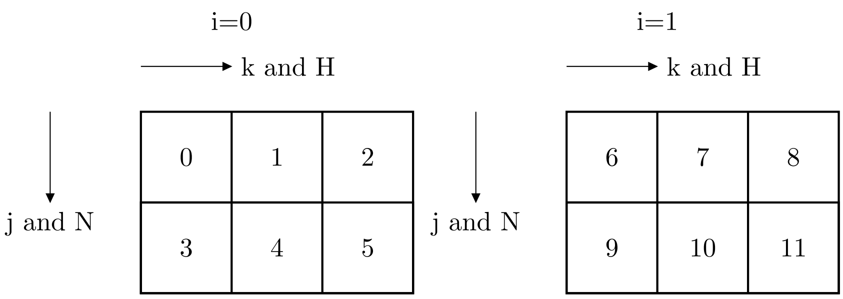
A.3 3D grid, $ni \times nj \times nk = (3,2,2)$, cyclic in x,i



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

Matrix for 3D flow. $ni \times nj \times nk = (3,2,2)$. Cyclic in x. The coefficients due to cyclic boundary conditions are colored in blue.

A.4 3D grid, $ni \times nj \times nk = (2,2,3)$, cyclic in z,k



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

Matrix for 3D flow. $ni \times nj \times nk = (2,2,3)$. Cyclic in and z. The coefficients due to cyclic boundary conditions are colored in blue.

References

[1] M. Billson. *Computational Techniques for Turbulence Generated Noise.* PhD thesis, Dept. of Thermo and Fluid Dynamics, Chalmers University of Technology, Göteborg, Sweden, 2004.

[2] M. Billson, L.-E. Eriksson, and L. Davidson. Jet noise prediction using stochastic turbulence modeling. AIAA paper 2003-3282, 9th AIAA/CEAS Aeroacoustics Conference, 2003.

[3] L. Davidson. LES of recirculating flow without any homogeneous direction: A dynamic one-equation subgrid model. In K. Hanjalić and T. W. J. Peeters, editors, *2nd Int. Symp. on Turbulence Heat and Mass Transfer*, pages 481–490, Delft, 1997. Delft University Press.

[4] L. Davidson. Hybrid LES-RANS: Inlet boundary conditions. In B. Skallerud and H. I. Andersson, editors, *3rd National Conference on Computational Mechanics – MekIT'05 (invited paper)*, pages 7–22, Trondheim, Norway, 2005.

[5] L. Davidson. Using isotropic synthetic fluctuations as inlet boundary conditions for unsteady simulations. *Advances and Applications in Fluid Mechanics*, 1(1):1–35, 2007.

[6] L. Davidson. Hybrid LES-RANS: Inlet boundary conditions for flows with recirculation. In *Second Symposium on Hybrid RANS-LES Methods*, Corfu island, Greece, 2007.

[7] L. Davidson. Inlet boundary conditions for embedded LES. In *First CEAS European Air and Space Conference*, 10-13 September, Berlin, 2007.

[8] L. Davidson. Hybrid LES-RANS: Inlet boundary conditions for flows including recirculation. In *5th International Symposium on Turbulence and Shear Flow Phenomena*, volume 2, pages 689–694, 27-29 August, Munich, Germany, 2007.

[9] L. Davidson. HYBRID LES-RANS: Inlet boundary conditions for flows with recirculation. In *Advances in Hybrid RANS-LES Modelling*, volume 97 of *Notes on Numerical Fluid Mechanics and Multidisciplinary Design*, pages 55–66. Springer Verlag, 2008.

[10] L. Davidson. Fluid mechanics, turbulent flow and turbulence modeling. eBook, Division of Fluid Dynamics, Dept. of Applied Mechanics, Chalmers University of Technology, Gothenburg, http://www.tfd.chalmers.se/~lada/postscript_files/solids-and-fluids_turbulent-flow_turbulence-modelling.pdf, 2014.

[11] L. Davidson and M. Billson. Hybrid LES/RANS using synthesized turbulent fluctuations for forcing in the interface region. *International Journal of Heat and Fluid Flow*, 27(6):1028–1042, 2006.

[12] L. Davidson and C. Friess. A new formulation of f_k for the PANS model. *Journal of Turbulence*, pages 1–15, 2019. doi: 10.1080/14685248.2019.1641605. URL http://dx.doi.org/10.1080/14685248.2019.1641605.

[13] P. Emvin. *The Full Multigrid Method Applied to Turbulent Flow in Ventilated Enclosures Using Structured and Unstructured Grids*. PhD thesis, Dept. of Thermo and Fluid Dynamics, Chalmers University of Technology, Göteborg, 1997.

[14] J. O. Hinze. *Turbulence*. McGraw-Hill, New York, 2nd edition, 1975.

[15] M. Irannezhad. DNS of channel flow with finite difference method on a staggered grid. Msc thesis, Division of Fluid Dynamics, Department of Applied Mechanics, Chalmers University of Technology, Göteborg, Sweden, 2006.

[16] N. Jarrin, S. Benhamadouche, D. Laurence, and R. Prosser. A synthetic-eddy-method for generating inflow conditions for large-eddy simulations. *International Journal of Heat and Fluid Flow*, 27(4):585–593, 2006.

[17] B. E. Launder and B. T. Sharma. Application of the energy dissipation model of turbulence to the calculation of flow near a spinning disc. *Lett. Heat and Mass Transfer*, 1:131–138, 1974.

[18] F. Nicoud and F. Ducros. Subgrid-scale stress modelling based on the square of the velocity gradient tensor. *Flow, Turbulence and Combustion*, 62(3):183–200, 1999.

[19] 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.

[20] J. Smagorinsky. General circulation experiments with the primitive equations. *Monthly Weather Review*, 91:99–165, 1963.

[21] 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.

[22] J. R. Welty, C. E. Wicks, and R. E. Wilson. *Fundamentals of Momentum, Heat, and Mass Transfer*. John Wiley & Sons, New York, 3 edition, 1984.

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

[24] C. Yap. *Turbulent heat and momentum transfer in recirculating and impinging flows*. PhD thesis, UMIST, Manchester, UK, 1987.