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 26, 2024

Abstract

This report gives some details on **pyCALC-LES** and how to use it. It is written in Python (3.8). 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 Cartesian but Δz may vary.

The code is suitable 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. The code can also be used for steady RANS and the time-marching method is then used to reach steady flow.

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` [1] has been found to be very efficient. For users who have an Nvidia graphics card, the entire code runs on the CPU (see Section 24). The `CuPy` library is used and this part was implemented in [2]. On large meshes the speed-up is a factor of 40 on the GPu compared to the CPU.

Contents

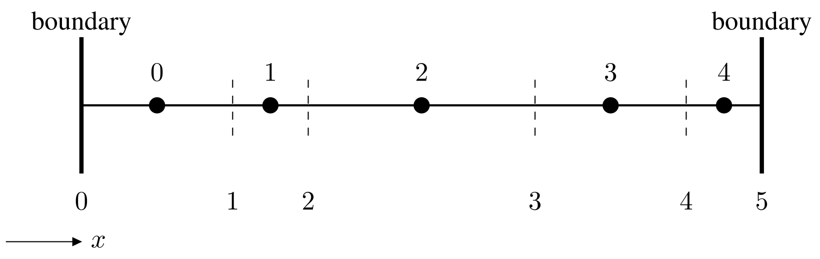


Figure 1.1: 1D grid with five cells (`ni=5`). The bullets denote cell centers (and control volume) which are labeled 0–4. Dashed lines denote control volume faces labeled 0–5.

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 set by the 1D array `z` (control volume face). 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), W(est), 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)
```

The grid in the $y-z$ direction (see Fig. 1.4), but may be non-equidistant. The z coordinates of the face and the cell center are given by the 1D arrays `z(k)` and `zp(k)`, respectively.

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

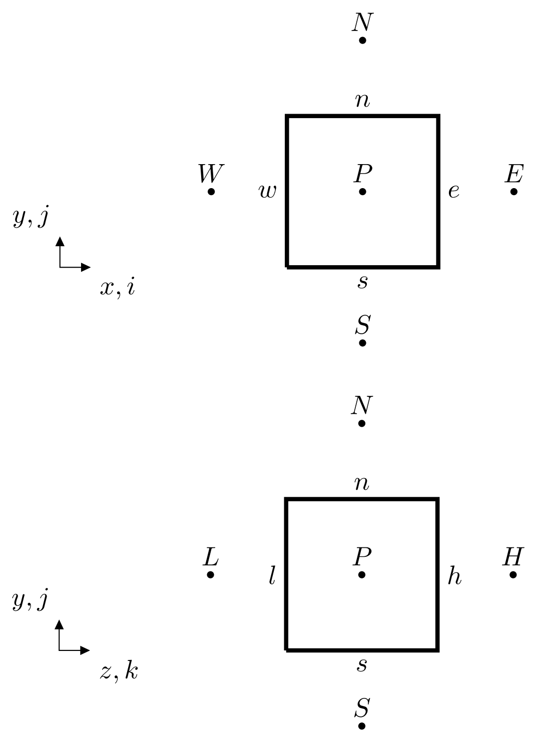


Figure 1.2: Control volume. Top: $x - y$ plane; bottom: $y - z$ plane.

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

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

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

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

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

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

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

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

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

The area for the low face is then obtained as

$$A_z = A1 + A2 \tag{1.3}$$

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`.

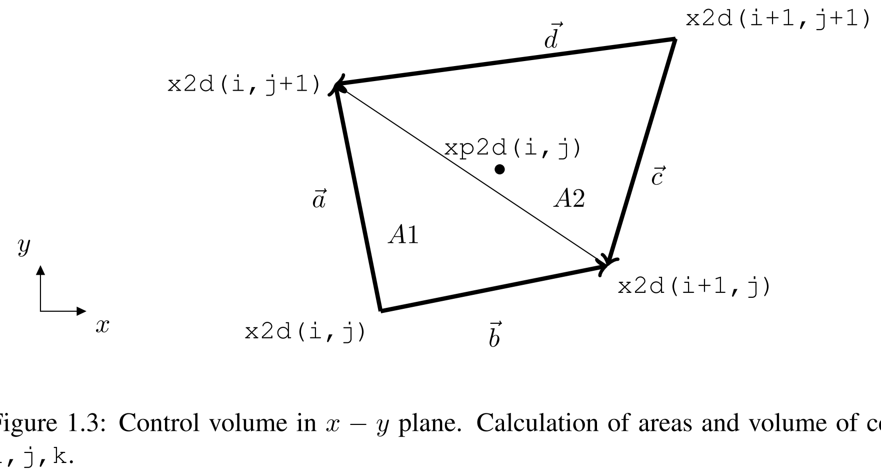


Figure 1.3: Control volume in $x - y$ plane. Calculation of areas and volume of cell `i,j,k`.

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

$$\phi_w = f_x \phi_P + (1 - f_x)\phi_W \tag{1.4}$$

where

$$f_x = \frac{|\overrightarrow{Ww}|}{|\overrightarrow{Pw}| + |\overrightarrow{Ww}|} \tag{1.5}$$

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 stored in the Python array `fz`.

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

$$\frac{\partial \Phi}{\partial x} = \frac{1}{V}\left(\Phi_e A_{ex} - \Phi_w A_{wx} + \Phi_n A_{nx} - \Phi_s A_{sx} + \Phi_h A_{hx} - \Phi_l A_{lx}\right) \tag{1.6}$$

where index e, w, n, s, h, l 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 derivatives $\partial\Phi/\partial x$, $\partial\Phi/\partial x$ and $\partial\Phi/\partial z$, are computed in the Python modules `dphidx`, `dphidy` and `dphidz`, respectively.

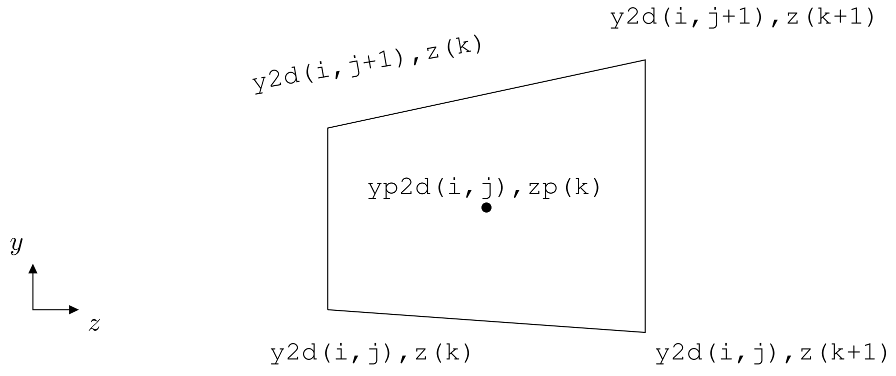


Figure 1.4: Control volume in $y-z$ plane.

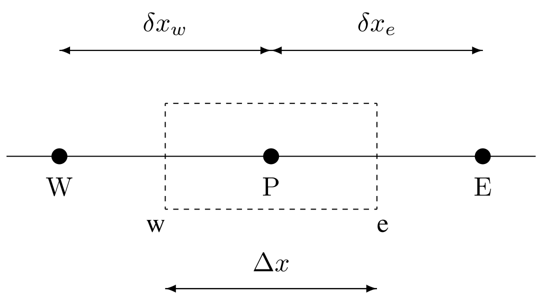


Figure 2.1: 1D control volume. Node P located in the middle of the control volume.

2 Diffusion

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

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

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

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

where (see Fig. 2.1):

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

$$a_P\phi_P\quad=\quad a_E\phi_E+a_W\phi_W+S_U \tag{2.4}$$

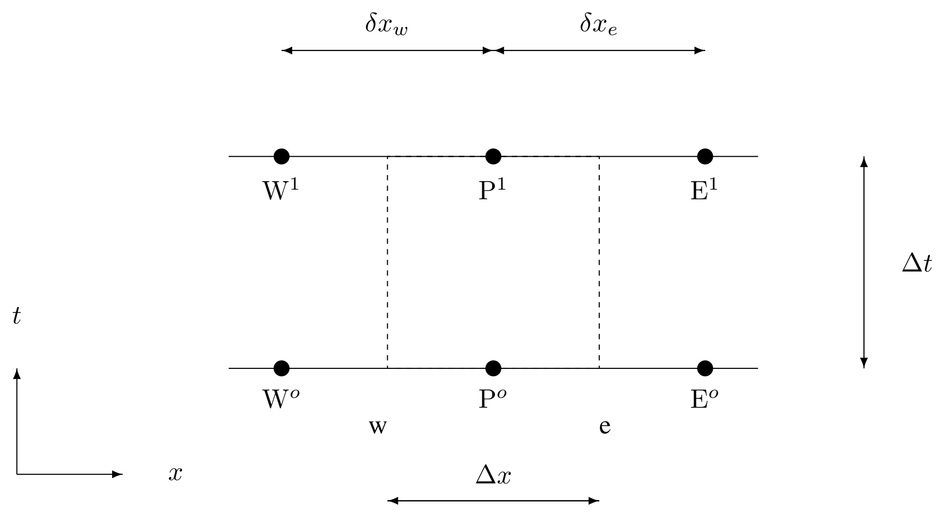


Figure 2.2: Control volume for 1D unsteady diffusion

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

2.1 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

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

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

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

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 in the sparse-matrix solvers except for the continuity equation for which the Python command `xp.linalg.norm` is employed. 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.

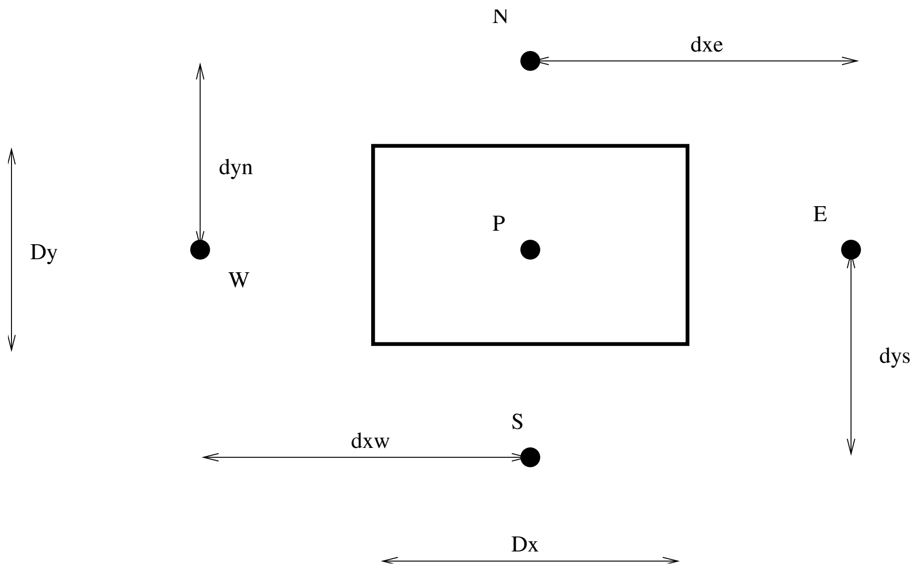


Figure 2.3: 2D control volume.

Continuity equation. F is here the total incoming volume flow $\dot{V}$. The Python variable resnorm_p is used for scaling.

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. The Python variable resnorm_vel is used for scaling.

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.

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

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.

$$\begin{aligned} a_P \phi_P &= a_E \phi_E + a_W \phi_W + a_N \phi_N + a_S \phi_S + a_H \phi_H + a_L \phi_L + S_U \\ a_W &= \frac{\Gamma_w A_w^2}{V_w} \\ a_S &= \frac{\Gamma_s A_s^2}{V_s} \\ a_L &= \frac{\Gamma_l A_l^2}{V_l} \\ a_P &= a_E + a_W + a_N + a_S + a_H + a_L - S_P \end{aligned} \tag{2.8}$$

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

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

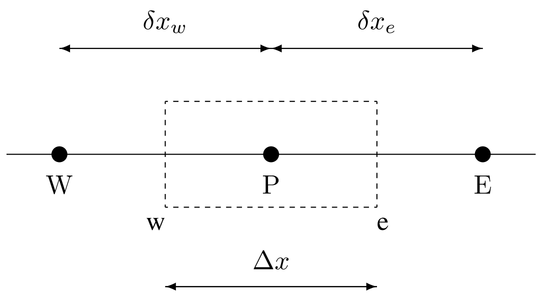


Figure 3.1: 1D control volume. Node P located in the middle of the control volume.

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 <u>different</u> 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. 11), and for constant mesh spacing $f_x = 0.5$. Assuming constant equidistant mesh (i.e. $\delta x_w = \delta x_e = \Delta x$) so that $f_x = 0.5$, inserting the discretized diffusion and the convection terms into Eq. 3.1 we obtain

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

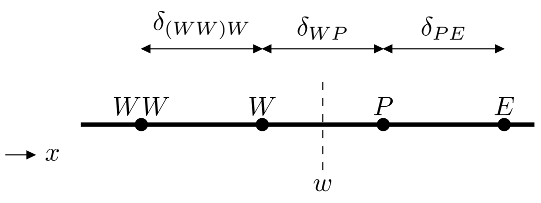


Figure 3.2: Constant mesh spacing. $u_w > 0$.

which can be rearranged as

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

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

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

(the continuity equation) to a_P so that

$$a_P = a_E + a_W.$$

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

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

MUSCL [3] is a second-order upwind bounded scheme. It uses two nodes upstream and one downstream, see Fig. 3.2. The coefficients $a_W, a_E, \ldots a_H$ are computed with first-order upwind. A correction term is then added to the source term. For the west face, e.g, it reads

$$\begin{aligned} C^+ &= 0.5 + 0.5 \cdot \mathrm{sign(C_w)} \\ C^- &= 0.5 \cdot \mathrm{sign(C_w)} - 0.5 \\ S_U^M &= 0.5 \cdot C_w \cdot (C^+ \cdot \mathrm{minmo(u_P - u_W, u_W - u_{WW})} \\ &- C^- \cdot \mathrm{minmo(u_P - u_W, u_E - u_P))} \end{aligned} \tag{3.2}$$

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

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

3.5 Blended CDS and MUSCL

When using the central differencing scheme (CDS), numerical oscillations may appear in inviscid regions. If they are harmful, they can be avoided by using a blend between CDS and MUSCLE. The MUSCL scheme is first used. A blending parameter, $b =$`blend`, is set. If $b = 1$, we use full CDS. This corresponds to *deferred* CDS and is different from the standard CDS because here the left-hand side (i.e. $a_W, a_E, \ldots a_H$) is first-order upwind and the CDS is added on the right-hand side. Hence, deferred CDS ($b = 1$) is more dissipative than CDS.

The blending parameter, b, is first used for reducing the additional source term, S_U^M (see Eq. 3.2) when $b < 1$, i.e.

$$S_U^M \quad = (1 - b) \cdot S_U^M \tag{3.3}$$

Second, it is used for adding the CDS scheme to the right-hand side and subtracting the first-order upwind scheme. For the west face the additional source term reads

$$S_U^{CDS} = b \cdot (a_W^c - a_W^u) \cdot (\alpha \cdot (u_W - u_P) + (1 - \alpha) \cdot ((u_W^o - u_P^o)) \tag{3.4}$$

where superscript c, u and o denote CDS, first-order upwind and previous time step, respectively. The parameter, α, is the time-discretization, see Eq. 2.5; it is 0.5 for Crank-Nicolson.

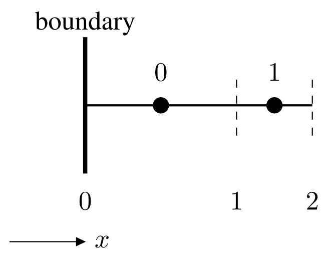


Figure 3.3: 1D grid. Boundary conditions at $x = 0$.

3.6 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. 3.3. Consider only convection. For the $\bar{u}$ equation at cell $i = 0$ we get

$$\begin{aligned}
a_P \bar{u}_P &= a_W \bar{u}_W + a_E \bar{u}_E + S_U \\
a_P &= a_W + a_E - S_P, \quad a_W = C_w, \quad a_E = -0.5C_e \\
C_w &= \bar{u}_W A_w \\
a_P &= C_w - 0.5C_e
\end{aligned} \tag{3.5}$$

Note there's no 0.5 in front of C_w since the west node is located *at* the inlet. Since there is no cell west of $i = 0$, Eq. 3.5 has to be implemented with additional source terms

$$\begin{aligned}
a_W &= 0 \\
S^u_{U,add} &= C_w \bar{u}_{in} \\
S^u_{P,add} &= -C_w
\end{aligned} \tag{3.6}$$

For $\bar{v}$ and w it reads

$$\begin{aligned}
S^v_{U,add} &= C_w \bar{v}_{in} \\
S^v_{P,add} &= -C_w \\
S^w_{U,add} &= C_w \bar{w}_{in} \\
S^w_{P,add} &= -C_w
\end{aligned} \tag{3.7}$$

The additional term for the diffusion reads

$$\begin{aligned}
S^u_{U,add,diff} &= \frac{\nu_{tot} A_w}{\Delta x} \bar{u}_{in} \\
S^v_{U,add,diff} &= \frac{\nu_{tot} A_w}{\Delta x} \bar{v}_{in} \\
S^w_{U,add,diff} &= \frac{\nu_{tot} A_w}{\Delta x} \bar{w}_{in} \\
S_{P,add,diff} &= -\frac{\nu_{tot} A_w}{\Delta x}
\end{aligned} \tag{3.8}$$

where $S_{P,add,diff}$ is the same for $\bar{u}$, $\bar{v}$ and $\bar{w}$. The viscous part of Eq. 3.8 is implemented in module `bc`. The turbulent part and the convective part (Eqs. 3.6 and 3.7) are implemented in `modify_u`, `modify_v` etc.

3.7 Wall boundary conditions using source term

We use exactly the same procedure as in Section 3.6. At walls, there is no convection and the velocity is zero. Hence we simply use Eq. 3.8 with $\bar{u} = \bar{v} = \bar{w} = 0$, i.e. (for west wall)

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

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

This boundary condition is implemented in module `bc`.

4 The Fractional-step method

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

The 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 [4]. The Navier-Stokes equation for the $\bar{u}_i$ velocity reads

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

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

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

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 $1/(\Delta t\alpha)$).

The Poisson equation for $\bar p^{n+1}$ is solved with an efficient algebraic multigrid method, either on the CPU (pyAMg [1]) or on the GPU using pyAMGx [5]. 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 or pyAMGx.
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 and transport equations for turbulent quantities are solved.
8. Next time step.

4.1 Boundary condition for $\bar p$

The Poisson equation for pressure reads (see Eq. 4.4)

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

Integration over the entire flow domain, V, using Gauss divergence law gives

$$\int_S \frac{\partial \bar p}{\partial x_i} n_i dS = \frac{1}{\Delta t\alpha}\int_S \bar v^*_{f,i} n_i dS \tag{4.6}$$

where S denotes the bounding surface of the flow domain and n_i is the unit normal vector of the surface. At the boundaries, the intermediate velocity field, $\bar v^*_{f,i}$, is equal to the physical velocity, i.e. $\bar v^*_{f,i} = \bar v_i$ Hence, the right side of Eq. 4.6 expresses the total mass flow out of the domain. This must – due to global continuity – be zero. A consistent boundary condition for the pressure is then

$$\frac{\partial \bar p}{\partial x_i} n_i = \frac{\partial \bar p}{\partial x_n} = 0 \tag{4.7}$$

at all boundaries (x_n denotes the local direction normal to the boundary).

	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.

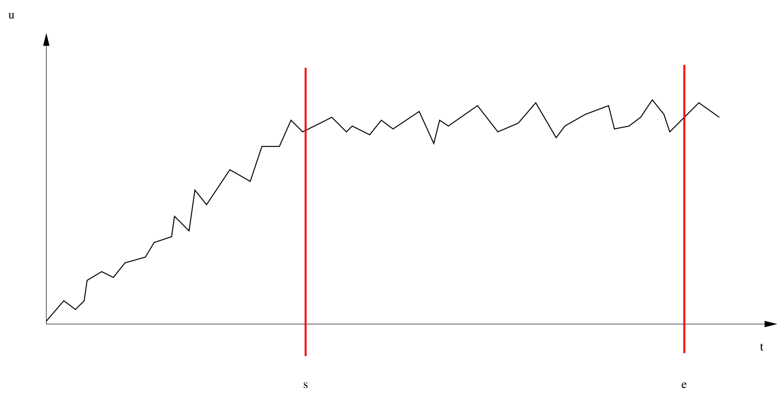


Figure 4.1: Time averaging in LES.

5 Boundary Conditions

5.1 Outlet velocity, small outlet

For *small* outlets, the outlet velocity can be determined from global continuity. As the outlet is small a constant velocity over the whole outlet can be used. The outlet velocity is set as (see Fig. 5.1)

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

5.2 Outlet velocity, large outlet

For *large* outlets the outlet velocity must be allowed to vary over the outlet. The proper boundary condition in this case is $\partial \bar{u}/\partial x = 0$. Hence it is important that the flow near the outlet is fully developed, so that this boundary condition corresponds to the flow conditions. The best way to ensure this is to locate the outlet boundary sufficiently far

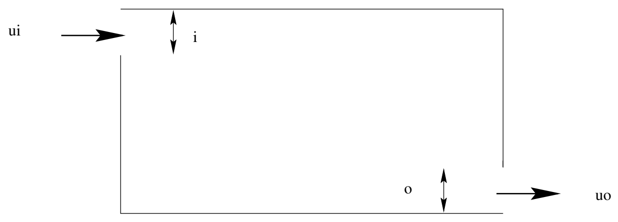


Figure 5.1: Outlet boundary condition. Small outlet

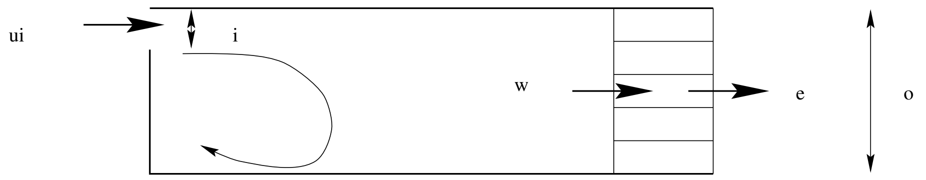


Figure 5.2: Outlet boundary condition. Large outlet.

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 [6]:

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

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 [7] 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, calceps, vist_pans`

The Reynolds number PANS model presented in [8, 9] reads

$$\begin{aligned}
\frac{\partial k}{\partial t} &+ \frac{\partial(k\bar{u}_j)}{\partial x_j} = \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_{ku}}\right)\frac{\partial k}{\partial x_j}\right] + (P^k - \varepsilon) \\
\frac{\partial \varepsilon}{\partial t} &+ \frac{\partial(\varepsilon\bar{u}_j)}{\partial x_j} = \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_{\varepsilon u}}\right)\frac{\partial \varepsilon}{\partial x_j}\right] + C_{\varepsilon 1}P^k\frac{\varepsilon}{k} - C^*_{\varepsilon 2}\frac{\varepsilon^2}{k} \\
\nu_t &= C_\mu f_\mu \frac{k^2}{\varepsilon}, \quad C^*_{\varepsilon 2} = C_{\varepsilon 1} + \frac{f_k}{\underline{f_\varepsilon}}(C_{\varepsilon 2}f_2 - C_{\varepsilon 1}) \\
\sigma_{ku} &\equiv \sigma_k \frac{f^2_k}{\underline{f_\varepsilon}}, \quad \sigma_{\varepsilon u} \equiv \sigma_\varepsilon \frac{f^2_k}{\underline{f_\varepsilon}}, \quad P^k = 2\nu_t\bar{s}_{ij}\bar{s}_{ij} \\
f_2 &= \left[1 - \exp\left(-\frac{y^*}{3.1}\right)\right]^2 \left\{1 - 0.3\exp\left[-\left(\frac{R_t}{6.5}\right)^2\right]\right\} \\
f_\mu &= \left[1 - \exp\left(-\frac{y^*}{14}\right)\right]^2 \left\{1 + \frac{5}{R^{3/4}_t}\exp\left[-\left(\frac{R_t}{200}\right)^2\right]\right\} \\
R_t &= \frac{k^2}{\nu\varepsilon}, \quad y^* = \frac{U_\varepsilon y}{\nu}, \quad U_\varepsilon = (\varepsilon\nu)^{1/4}
\end{aligned} \tag{8.1}$$

The modifications introduced by the PANS modeling as compared to its parent RANS model are underlined. The model constants take the same values as in the AKN model [10], i.e.

$$C_{\varepsilon 1} = 1.5, C_{\varepsilon 2} = 1.9, \sigma_k = 1.4, \sigma_\varepsilon = 1.4, C_\mu = 0.09 \tag{8.2}$$

When the turbulent Prandtl numbers, σ_k and σ_ε (Python variables `prand_k` and `prand_eps`), are set to negative values, $\sigma_{k,u}$ and $\sigma_{\varepsilon,u}$ are computed as in Eq. 8.1 using the absolute values of σ_k and σ_ε. When σ_k and σ_ε are positive, the PITM model is used (see Section 9).

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 [11]

$$\begin{aligned}
f_k &= \max\left[f_{k,min}, \min\left(1, 1 - \frac{\psi - 1}{C_{\varepsilon 2} - C_{\varepsilon 1}}\right)\right] \\
\psi &= \max\left(1, \frac{k^{3/2}/\varepsilon}{C_{DES}\Delta_{max}}\right), \quad \Delta_{max} = \max(\Delta x_1, \Delta x_2, \Delta x_3)
\end{aligned} \tag{8.3}$$

which means that the interface is chosen automatically. The minimum $f_{k,min}$ is stored in the Python variable `fkmin_limit`

At the wall-adjacent cells, ε is not solved but it is fixed as

$$\varepsilon_P = \frac{2\nu k}{y^2} \tag{8.4}$$

where subscript P denotes wall-adjacent cells and y is the distance between the cell center and the wall. This is done in module `fix_eps`.

9 The PITM Model

PITM is an acronym for Partially Integrated Transport Model [12, 13]. It is identical to the PANS model, except that $f_k = 1$ in the diffusion terms of the k and ε equations. Setting `pans=True` and positive values σ_k and σ_ε activates PITM.

10 The $k - \omega$ DES model

modules: `calck_kom, calcom, vist_kom`

The Wilcox $k - \omega$ RANS turbulence model [14] modified for DES reads

$$\begin{aligned}
\frac{\partial k}{\partial t} + \frac{\partial \bar{v}_i k}{\partial x_i} &= P^k - F_{DES} c_\mu k\omega + \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_k}\right)\frac{\partial k}{\partial x_j}\right] \\
\frac{\partial \omega}{\partial t} + \frac{\partial \bar{v}_i \omega}{\partial x_i} &= C_{\omega_1}\frac{\omega}{k}P^k - C_{\omega 2}\omega^2 + \frac{\partial}{\partial x_j}\left[\left(\nu + \frac{\nu_t}{\sigma_\omega}\right)\frac{\partial \omega}{\partial x_j}\right] \\
\nu_t &= \frac{k}{\omega} \\
F_{DES} &= \max\left(\frac{k^{1/2}}{c_\mu \omega \Delta}, 1\right), \quad \Delta = 0.67\max(\Delta x_1, \Delta x_2, \Delta x_2)
\end{aligned} \tag{10.1}$$

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

$$k_w = 0, \quad \omega_w = 10\frac{6\nu}{C_{\omega 2} y^2} \tag{10.2}$$

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

11 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 [15, 16].

There are different ways to create turbulent inlet boundary conditions. One way is to use a pre-cursor DNS or a 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 [17]. 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.

11.1 Synthesized turbulence

module: `synt_fluct`.

The method described below was developed in [18, 19] for creating turbulence for generating noise. It was later further developed for inlet boundary conditions [20, 21, 22, 23].

A turbulent fluctuating velocity fluctuating field (whose average is zero) can be expressed using a Fourier series, see [24]. Let us re-write this formula as

$$\begin{aligned} a_n\cos(nx)+b_n\sin(nx) &= \\ c_n\cos(\alpha_n)\cos(nx)+c_n\sin(\alpha_n)\sin(nx) &= c_n\cos(nx-\alpha_n) \end{aligned} \tag{11.1}$$

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

$$c_n = \left(a_n^2+b_n^2\right)^{1/2}, \quad \alpha_n = \arctan\left(\frac{b_n}{a_n}\right) \tag{11.2}$$

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

$$\mathbf{v}'(\mathbf{x}) = 2\sum_{n=1}^{N}\hat{u}^n\cos(\boldsymbol{\kappa}^n\cdot\mathbf{x}+\psi^n)\boldsymbol{\sigma}^n \tag{11.3}$$

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.

11.2 Random angles

The angles φ^n and θ^n determine the direction of the wavenumber vector $\boldsymbol{\kappa}$, see Eq. 11.3 and Eq. 11.1; α^n denotes the direction of the velocity vector, $\mathbf{v}'$. For more details, see [24].

11.3 Highest wave number

Define the highest wave number based on mesh resolution $\kappa_{max} = 2\pi/(2\Delta)$ (see [24]), 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 don't let it be smaller than `dmin_synt` (which can be set to a fraction of `dz`).

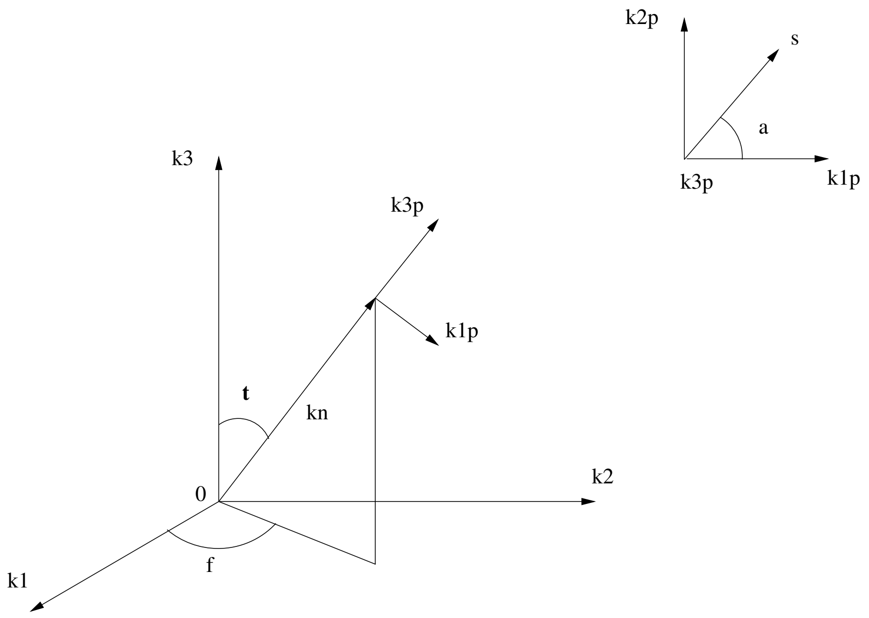


Figure 11.1: The wave-number vector, κ_i^n, and the velocity unit vector, σ_i^n, are orthogonal (in physical space) for each wave number n.

11.4 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 [21, 22, 23] 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.

11.5 Divide the wave number range

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

11.6 von Kármán spectrum

A modified von Kármán spectrum is chosen, see Eq. 11.4 and Fig. 11.2. The amplitude $\hat{u}^n$ of each mode in Eq. 11.3 is then obtained from

$$\begin{aligned}
\hat{u}^n &= (E(\kappa)\Delta\kappa)^{1/2} \\
E(\kappa) &= c_E \frac{u_{rms}^2}{\kappa_e} \frac{(\kappa/\kappa_e)^4}{[1+(\kappa/\kappa_e)^2]^{17/6}} e^{[-2(\kappa/\kappa_\eta)^2]} \\
\kappa &= (\kappa_i\kappa_i)^{1/2}, \quad \kappa_\eta = \varepsilon^{1/4}\nu^{-3/4}
\end{aligned} \tag{11.4}$$

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

$$k = \int_0^\infty E(\kappa)d\kappa \tag{11.5}$$

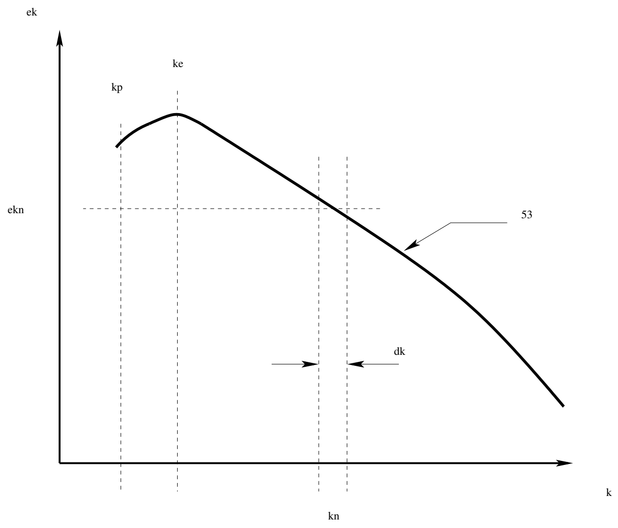


Figure 11.2: Modified von Kármán spectrum

which gives [25]

$$c_E = \frac{4}{\sqrt{\pi}} \frac{\Gamma(17/6)}{\Gamma(1/3)} \simeq 1.453 \tag{11.6}$$

where

$$\Gamma(z) = \int_0^\infty e^{-z'} x^{z-1} dz' \tag{11.7}$$

11.7 Computing the fluctuations

Having $\hat{u}^n$, κ_j^n, σ_i^n and ψ^n, allows the expression in Eq. 11.3 to be computed, i.e.

$$\begin{aligned}
v'_1 &= 2\textstyle\sum_{n=1}^N \hat{u}^n \cos(\beta^n)\sigma_1 \\
v'_2 &= 2\textstyle\sum_{n=1}^N \hat{u}^n \cos(\beta^n)\sigma_2 \\
v'_3 &= 2\textstyle\sum_{n=1}^N \hat{u}^n \cos(\beta^n)\sigma_3 \\
\beta^n &= k_1^n x_1 + k_2^n x_2 + k_3^n x_3 + \psi^n
\end{aligned} \tag{11.8}$$

where $\hat{u}^n$ is computed from Eq. 11.4.

In this way inlet fluctuating velocity fields (v'_1, v'_2, v'_3) are created at the inlet x_2-x_3 plane.

11.8 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 is non-physical. 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

$$(\mathcal{V}'_1)_m \quad = a(\mathcal{V}'_1)_{m-1} + b(v'_1)_m$$

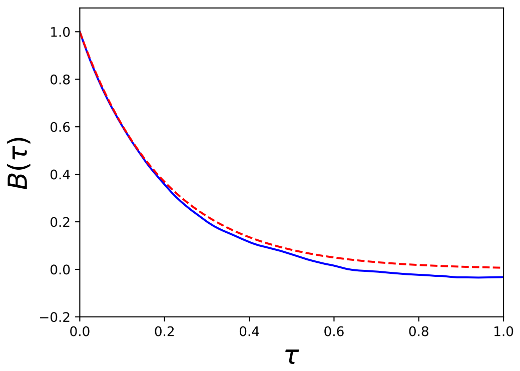


Figure 11.3: Auto correlation, $B(\tau) = \langle v_1'(t)v_1'(t-\tau)\rangle_t$ (averaged over time, t). ━━ : Eq. 11.12; ‒ ‒ : computed from synthetic data, $(\mathcal{V}_1')^m$, see Eq. 11.9.

$$\begin{aligned}(\mathcal{V}_2')_m &= a(\mathcal{V}_2')_{m-1} + b(v_2')_m \\ (\mathcal{V}_3')_m &= a(\mathcal{V}_3')_{m-1} + b(v_3')_m\end{aligned} \tag{11.9}$$

where m denotes the time step number and

$$a = \exp(-\Delta t/T_{int}) \tag{11.10}$$

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

$$b = (1 - a^2)^{0.5} \tag{11.11}$$

which ensures that $\langle \mathcal{V}_1'^2\rangle = \langle v_1'^2\rangle$ ($\langle\cdot\rangle$ denotes averaging). The time correlation of will be equal to

$$\exp(-\hat{t}/T_{int}) \tag{11.12}$$

where $\hat{t}$ is the time separation and thus Eq. 11.9 is a convenient way to prescribe the turbulent time scale of the fluctuations. For more detail, see Section 11.8. 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)

$$\begin{aligned}\bar{v}_1(0, x_2, x_3, t) &= V_{1,in}(x_2) + v_{1,in}'(x_2, x_3, t) \\ \bar{v}_2(0, x_2, x_3, t) &= V_{2,in}(x_2) + v_{2,in}'(x_2, x_3, t) \\ \bar{v}_3(0, x_2, x_3, t) &= V_{3,in}(x_2) + v_{3,in}'(x_2, x_3, t)\end{aligned} \tag{11.13}$$

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. 11.9). 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 [26]

$$V_{1,in}^+ = \begin{cases} x_2^+ & x_2^+ \le 5 \\ -3.05 + 5\ln(x_2^+) & 5 < x_2^+ < 30 \\ \frac{1}{\kappa}\ln(x_2^+) + B & x_2^+ \ge 30 \end{cases} \tag{11.14}$$

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

The method to prescribed fluctuating inlet boundary conditions have been used for channel flow [23], for diffusor flow [16] as well as for the flow over a bump and an axisymmetric hill [27].

The time correlation is implemented in module `modify_inlet`.

12 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 17.

2. After having carried out the pre-cursor RANS simulation, the Reynolds stress tensor is computed using the EARSM model [28].

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.

13 Flow Chart

pyCALC-LES flowchart

14 Modules

14.1 bc_outlet_bc

Neumann outlet boundary conditions are set.

14.2 calceps

Source terms in the ε equation (AKN) 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`.

14.3 calck_kom

Source terms in the k equation (Wilcox model) are computed, see Section 10. When DES is used, C_{DES} is computed (it is stored in `fk3d`). The user can define additional source terms in `modify_k`.

14.4 calck

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

14.5 calcom

Source terms in the ω equation (Wilcox model) are computed, see Section 10. The user can define additional source terms in `modify_om`.

14.6 calcp

Coefficients in the $\bar{p}$ equation (Eq. 4.4). It is a diffusion equation and hence the coefficients $a_w, a_E \dots$ are computed in the same was as in Section 2.4 (with the diffusion coefficient Γ set to $1/(\alpha \Delta t)$).

14.7 calcu

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

14.8 calcv

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

14.9 calcw

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

14.10 coeff

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

14.11 compute_face_phi

Compute the face values of a variable.

14.12 compute_fk

Computes f_k (array `f3kd`) from Eq. 8.3. The user can modify `fk3d` in `modify_fk`.

14.13 compute_inlet_fluct

Compute synthetic fluctuations (see Section 11.1)

14.14 conv

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

14.15 correct_conv

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

14.16 fix_eps, fix_k, fix_omega

This routine may be used for set ε, k and ω in the wall-adjacent *cell center*. For ω, for example, it may be set according to Eq. 10.2 rather than as a wall-boundary condition. Note that it is called just before the solver is called. For fixing ω near a south boundary we use

```
aw3d[:,0,:]=0
ae3d[:,0,:]=0
as3d[:,0,:]=0
an3d[:,0,:]=0
al3d[:,0,:]=0
ah3d[:,0,:]=0
ap3d[:,0,:]=1
su3d[:,0,:]=om_bc_south
```

The discretized equation (Eq. 2.8) then reads

$$\omega_P = S_U$$

which gives ω_P =`om_bc_south` as intended.

14.17 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.

14.18 dphidx, dphidy, dphidz

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

14.19 init

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

14.20 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.

14.21 modify_case.py

This file includes `modify_eps`, `modify_k`, ... `modify_w` and `modify_conv`, `modify_init`, `modify_inlet`, `modify_outlet`, `fix_eps`, `fix_k`, `fix_omega`, `modify_vis` and `modify_fk`.

14.22 modify_init

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

14.23 print_indata

Prints the indata set by the user.

14.24 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.

14.25 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`.

14.26 save.file

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

14.27 save_time_aver_data

This module is called when every `itstep_save` time step 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`

If `save_average_z`=True, the flow fields are averaged also in the z direction.

14.28 save_vtk

The results are stored in VTK format. It is called if `vtk=True`. You must then set the name of the VTK file names, i.e. `vtk_file_name`. If `vtk_movie` is true, the results are saved every `itstep_save` time step may be viewed as a movie.

14.29 setup_case.py

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

14.30 solve_amg

An algebraic multigrid solver.

14.31 solve_amgx

An algebraic multigrid solver on the GPU. The sparse matrix (e.g. Eq. B.4) is uploaded every time this module is called.

14.32 solve_3d

This module can be 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.cgs`, `linalg.cg`, `linalg.gmres`, `linalg.qmr` or `linalg.lgmres`.

14.33 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 time step 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','cg','gm','gmres','fgmres','cgne','cgnr','cr'').

14.34 solve_px

An algebraic multigrid solver on the GPU for the pressure. The sparse matrix (e.g. Eq. B.4) is uploaded once at the first iteration at the first time step. Note that this option requires twice as much memory on the GPU compared to `solve_amgx`.

14.35 solve_tdma

This module can be used for all variables except pressure, $\bar{p}$. It solves the equations exactly in the y direction by treating the x and z directions explicitly. Hence, only the coefficient arrays `as3d, an3d, ap3d` are used in the matrix solver. With theses three arrays, a sparse matrix is created, `A`. The equation system is solved using the Python solver `linalg.spsolve`. This means that the equation is solved by TDMA (Tri-Diagonal-Matrix-Algorithm). It is a combination of exact solution in y direction (TDMA) combined with Jacobi iteration in the other two directions. This solver is efficient at high Reynolds numbers when the diffusion terms near the wall are very large. This solver is activated by setting `solver_vel = 'tdma'` and `solver_turb = 'tdma'`. When the solver `tdma` is employed, the convergence limits are not used. Instead we rely on how many sweeps should be made (`nsweep_vel`, `nsweep_keps` and `nsweep_kom`).

14.36 synt_fluct

The synthetic fluctuations are computed and scaled with `uv_rans`, see Section 11.1

14.37 time_stats

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

14.38 update

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

14.39 vist_kom

The turbulent viscosity is computed using the $k - \omega$ model, see Section 10

14.40 vist_pans

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

14.41 vist_smag

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

14.42 vist_wale

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

15 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*. You can also kook at the **pyCALC-LES** flowchart.

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=xp.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=xp.repeat(yc[None,:], repeats=ni+1, axis=0)

   y2d=xp.append(y2d,nj)
   xp.savetxt('y2d.dat', y2d)
# x grid
   xc = xp.linspace(0, xmax, ni+1)
# make it 2D
   x2d=xp.repeat(xc[:,None], repeats=nj+1, axis=1)
   x2d_org=x2d
   x2d=xp.append(x2d,ni)
   xp.savetxt('x2d.dat', x2d)
```

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

```
   zmax, nk=xp.loadtxt('z.dat')
   nk=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.

15.1 `setup_case.py`

This module consists of 10 sections.

15.1.1 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 for the pressure gradient stabilizes the simulation and makes it possible to use only one global iteration.

15.1.2 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 19.

15.1.3 Section 4

The viscosity is set.

```
viscos=1/500
```

15.1.4 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 18), 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_u=1e-5
    convergence_limit_v=1e-5
    convergence_limit_w=1e-5
    convergence_limit_p=5e-4
```

The convergence limit in the Python solvers is defined as

$$|Ax - b|/|b| < \gamma \tag{15.1}$$

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

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

If your computer has an Nvidia compatible graphics card, you may select to solve the pressure equation on the graphics card. You set

```
solver_p='pyamgx'
```

If you want to solve also the velocity equations on the GPU, you set

```
solver_vel='pyamgx'
```

Note that you must then install CUDA, the AMGX library as well as `pyamgx`, see Section C. The `pyamgx` solver on GPU gives a speed-up of approximately a factor of ten when solving the pressure equation and an overall speed-up factor of approximately two.

15.1.5 Section 7

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

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

15.1.6 Section 8

We use 15000 time steps. Time-averaging starts after 7500 time steps. The time steps is set to $0.5\Delta x/U_{in}$ where U_{in} is an estimated bulk velocity. The instantaneous and time-averaged fields are saved to disk every 2000 time steps. When time-averaging, we use every 10^{th} time step.

```
ntstep=15000
uin=20
dt=0.5*(x2d[1,0]-x2d[0,0])*xp.ones(ntstep)/uin
itstep_start=7500
itstep_save=2000
itstep_stats=10
```

We don't want to store data on VTK format. Hence

```
vtk=False
```

15.1.7 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]
```

15.1.8 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=xp.zeros((ni,nk))
u_bc_north=xp.zeros((ni,nk))
v_bc_south=xp.zeros((ni,nk))
v_bc_north=xp.zeros((ni,nk))
w_bc_south=xp.zeros((ni,nk))
w_bc_north=xp.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

15.2 `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`)

15.2.1 **`modify_u`**

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

```
su3d= su3d+vol
```

15.3 Run the code

The bash script `run-python` is used which reads

```
#!/bin/bash
# delete first line
sed  '/setup_case()/d' setup_case.py > temp_file
# add new first line plus global declarations
cat  ~/pythons-rans-code/global-GPU temp_file modify_case.py ~/pythons-rans-coc
# find setting of 'gpu'
'grep' gpu setup_case.py > gpu_value;
# remove leading white space
sed -i 's/^[ \t]*//' gpu_value;
# add value of 'gpu' at first line
cat  gpu_value temp_file1  > exec-pyCALC-LES-GPU.py;
/usr/bin/time -a  -o out ~/anaconda3/bin/python -u  exec-pyCALC-LES-GPU.py > ou
```

You must change the last line if your Python installation is not in `/anaconda3/bin/`. If you have not installed the Python module `pyamgx` (see Section C) you must comment line 11 in **pyCALC-LES**.

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 &
```

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

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

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

```
grep umax out
```

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

```
grep warn out
```

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

```
grep history out
```

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

Plot the results using the script `pl_uvw_DNS.py`

16 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 15.

16.1 `setup_case.py`

16.1.1 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
```

16.1.2 Section 2

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

```
kom_des = True
jl0=0
```

The variable `jl0`is set to zero which means that the location LES-RANS interface is automatically computed (if we want to prescribe the j line of the interface, we set it to a negative value). 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
```

16.1.3 Section 5

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

```
urfvis=0.5
```

16.1.4 Section 6

The `tdma` solver is chosen for k and ω and the number of sweep is set to one.

```
solver_turb='tdma'
nsweep_kom=1
```

16.1.5 Section 10

The wall-boundary conditions for k and ω are set as $k = 0$ and ω as below (see Eq. 10.2).

```
# boundary conditions for k
   k_bc_south=xp.zeros((ni,nk))
   k_bc_north=xp.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=6*viscos/c_omega_2/dist2_s

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

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

16.2 modify_case.py

No changes are made compared to Section 15.

Note that no initial conditions are set here. The default ones are used which are set where all variables are initialized, i.e.

```
u3d=xp.zeros((ni,nj,nk))
v3d=xp.zeros((ni,nj,nk))
w3d=xp.zeros((ni,nj,nk))
k3d=xp.ones((ni,nj,nk))*1
eps3d=xp.ones((ni,nj,nk))*1
om3d=xp.ones((ni,nj,nk))*1
```

17 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 15, 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 16.

17.1 `setup_case.py`

17.1.1 Section 1

Since we will simulated a time-marching flow towards steady conditions we choose the hybrid scheme for the velocities and set fully implicit time integration for the velocities, i.e.

```
scheme='h'
acrank_conv=1
```

17.1.2 Section 2

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

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

17.1.3 Section 3

We don't start from a previous solution.

```
restart = False
```

17.1.4 Section 8

We increase the time step.

```
dt=4*(x2d[1,0]-x2d[0,0])*xp.ones(ntstep)/uin
```

and we use 1000 and time average during the last 100 time steps

```
ntstep=1000
itstep_start=ntstep-100
```

17.1.5 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_low_type='n'
u_bc_high='n'
v_bc_low_type='n'
v_bc_high='n'
w_bc_low_type='d'
w_bc_high='d'
p_bc_low_type='n'
p_bc_high='n'
k_bc_low_type='n'
k_bc_high='n'
om_bc_low_type='n'
om_bc_high='n'
```

17.2 `modify_case.py`

No changes are made compared to Section 16.

18 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 AKN as the baseline RANS model.

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 16.

18.1 `setup_case.py`

18.1.1 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
```

18.1.2 Section 2

The PANS model is selected

```
pans = True
```

18.1.3 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
```

18.1.4 Section 6

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

```
min_iter=2
```

For the turbulent quantities we use the `tdma` solved and set the number of sweeps to one.

```
solver_turb='tdma'
nsweep_keps=1
```

18.1.5 Section 8

Number of time steps is set to 15000 and time averaging starts after 7500 time steps. The time step 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])*xp.ones(ntstep)/uin
itstep_start=7500
```

18.2 `modify_case.py`

18.2.1 `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 [29].

First, compute the viscous forces at the walls,

```
taus=xp.sum(viscos*as_bound*u3d[:,0,:])
taun=xp.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=xp.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=xp.sum(vol)
uin=xp.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
```

18.2.2 `fix_eps`

Here we set the wall boundary conditions on ε according to Eq. 8.4

```
# south wall
   aw3d[:,0,:]=0
   ae3d[:,0,:]=0
   as3d[:,0,:]=0
   an3d[:,0,:]=0
   al3d[:,0,:]=0
   ah3d[:,0,:]=0
   ap_max=xp.max(ap3d)
   ap3d[:,0,:]=ap_max
   su3d[:,0,:]=ap_max*2*viscos*k3d[:,0,:]/dist3d[:,0,:]**2

# north wall
   aw3d[:,-1,:]=0
   ae3d[:,-1,:]=0
   as3d[:,-1,:]=0
   an3d[:,-1,:]=0
   al3d[:,-1,:]=0
   ah3d[:,-1,:]=0
   ap_max=xp.max(ap3d)
   ap3d[:,-1,:]=ap_max
   su3d[:,-1,:]=ap_max*2*viscos*k3d[:,-1,:]/dist3d[:,-1,:]**2
```

Run the code and then plot the results using the script `plot_hill.py`.

19 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 17. Synthetic fluctuations are superimposed on the mean flow.

Go into the directory `channel-395-inlet-ni96` (or something similar). 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 16.

19.1 `setup_case.py`

19.1.1 Section 2

We choose the WALE turbulence model

```
   wale = True
```

19.1.2 Section 3

No restart.

```
restart = False
```

19.1.3 Section 4

Reynolds number $Re_\tau = 395$

```
viscos=1/395
```

19.1.4 Section 6

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

```
amg_relax='default'
```

19.1.5 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 length scale of the synthetic fluctuations to $L_t = 0.2$ (see Section 11.4) and the number of modes (see Section 11.5) to 1200.

```
L_t_synt=0.2
nmodes_synt=1200
```

The Reynolds stress tensor of the generated time-averaged anisotropic fluctuations is equal to the prescribed Reynolds stress tensor, see Item 2 in Section 12. 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=int(nj/2)
```

19.2 `modify_case.py`

19.2.1 `modify_init`

Here we set initial conditions. We use the 1D RANS data, see Section 17 (the y, u, k, ω and $\overline{v_1'v_2'}$ profiles are stored on disk in `pl_uvw.py`). We read $\bar{u}$

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

19.2.2 modify_inlet

Inlet boundary conditions are set here. At the first time step, we read the 1D RANS solution for $\bar{u}$ and $\overline{v_1' v_2'}$

```
   if itstep == 0:
      y_u_k_om=xp.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=xp.repeat(u_rans[:,None], repeats=nk, axis=1)

      uv_rans=xp.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. 11.8.

```
      zp = xp.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=xp.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. 11.13) and compute a and b (see Eqs. 11.10 and 11.11).

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

For time step higher than zero, we call `synt_fluct`, correct u' and make the time filtering in Eq. 11.13

```
   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=xp.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 result from the `modify_inlet`

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

19.2.3 **modify_u**

Add the inlet convective flow to source terms

```
   su3d[0,:,:]= su3d[0,:,:]+xp.maximum(convw[0,:,:],0)*u_bc_west
   sp3d[0,:,:]= sp3d[0,:,:]-xp.maximum(convw[0,:,:],0)
   vist=vis3d[0,:,:]-viscos
   sp3d[0,:,:]=sp3d[0,:,:]-vist*aw_bound
   su3d[0,:,:]=su3d[0,:,:]+vist*aw_bound*u_bc_west
```

We take max of `convw` because large negative synthetic fluctuations sometimes make $\bar{u}$ negative near the walls. Note that the viscous diffusive part is added in module `bc`.

19.2.4 **modify_v**

Same as in `modify_u`

```
   su3d[0,:,:]= su3d[0,:,:]+xp.maximum(convw[0,:,:],0)*v_bc_west
   sp3d[0,:,:]= sp3d[0,:,:]-xp.maximum(convw[0,:,:],0)
   vist=vis3d[0,:,:]-viscos
   sp3d[0,:,:]=sp3d[0,:,:]-vist*aw_bound
   su3d[0,:,:]=su3d[0,:,:]+vist*aw_bound*v_bc_west
```

19.2.5 **modify_w**

Same as in `modify_u`

```
   su3d[0,:,:]= su3d[0,:,:]+xp.maximum(convw[0,:,:],0)*w_bc_west
   sp3d[0,:,:]= sp3d[0,:,:]-xp.maximum(convw[0,:,:],0)
   vist=vis3d[0,:,:]-viscos
   sp3d[0,:,:]=sp3d[0,:,:]-vist*aw_bound
   su3d[0,:,:]=su3d[0,:,:]+vist*aw_bound*w_bc_west
```

19.2.6 **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=xp.sum(convw[0,:,:])
   flow_out=xp.sum(convw[-1,:,:])
   area_out=xp.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`.

20 Synthetic turbulence at inlet using commutation terms: Channel flow

Here we will simulate the flow in a channel at $Re_\tau = 5,\ 200$. We use the $k - \omega$ DES turbulence model. The grid in the y and z direction is used as in Section 16. The number of cells and extent in the x direction are 96 and 9 (constant grid spacing), respectively.

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

Below, we highlight the differences compared to Section 19.

20.1 `setup_case.py`

20.1.1 Section 2

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

```
kom_des = True
```

The interface is automatically computed

```
jl0 = 0
```

20.1.2 Section 4

The Reynolds number is set to 5 200.

```
viscos=1/5200
```

20.1.3 Section 6

For the turbulent quantities we use the `gmres` solved and set the number of sweeps to 50.

```
solver_turb='gmres'
nsweep_kom=50
```

20.1.4 Section 10

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

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

   k_bc_south_type='d'
   k_bc_north_type='d'

# boundary conditions for omega
   om_bc_south=xp.zeros((ni,nk))
   om_bc_north=xp.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=xp.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=xp.repeat(om_bc_north[:,None], repeats=nk, axis=1)
```

20.2 modify_case.py

20.2.1 modify_init

Here we set initial conditions. We use the 1D RANS data, see Section 17. 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=xp.loadtxt('y_u_k_om_uv_5200_nj96.txt')
   u_rans=data[:,1]
# make it 2D
   u_rans=xp.repeat(u_rans[:,None], repeats=nk, axis=1)

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

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

# set inlet field in entre domain
```

```
   u3d=xp.repeat(u_rans[None,:,:], repeats=ni, axis=0)
   k3d=0.2*xp.repeat(k_rans[None,:,:], repeats=ni, axis=0)
   om3d=k3d**0.5/(0.01*delta_max)

   vis3d=k3d/om3d+viscos
```

20.2.2 modify_inlet

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

```
   if itstep == 0:
      y_u_k_om=xp.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=xp.repeat(u_rans[:,None], repeats=nk, axis=1)
      k_rans=y_u_k_om[:,2]
# make it 2D
      k_rans=xp.repeat(k_rans[:,None], repeats=nk, axis=1)
      eps_rans=y_u_k_om[:,3]
# make it 2D
      eps_rans=xp.repeat(eps_rans[:,None], repeats=nk, axis=1)
      uv_rans=xp.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.

20.2.3 modify_k

We need to add inlet boundary conditions.

```
   su3d[0,:,:]= su3d[0,:,:]+xp.maximum(convw[0,:,:],0)*k_bc_west
   sp3d[0,:,:]= sp3d[0,:,:]-xp.maximum(convw[0,:,:],0)
   vist=vis3d[0,:,:]-viscos
   su3d[0,:,:]=su3d[0,:,:]+vist*aw_bound*k_bc_west
   sp3d[0,:,:]=sp3d[0,:,:]-vist*aw_bound
```

We prescribe RANS inlet conditions on k and ω. Hence, we must make sure that they are turned into values relevant to LES. This is done by adding commutation terms [30, 31]. It is implemented as:

```
   delt_i1=0.09**(-0.25)*k_bc_west**0.5/om_bc_west
   delt_i2=vol[0,:,:]**0.333333
   flux_k_RANS=xp.maximum(u_bc_west,0)*k3d[0,:,:]
   vis_smag= (0.1 *delt_i2)**2*gen[0,:,:]**0.5
```

```
rk_smag=(vis_smag/delt_i2)**2
flux_k_LES=u3d[0,:,:]*rk_smag
delt_LES=delt_i2
delt_RANS=delt_i1
dx=x2d[1,0]-x2d[0,0]
comm_term=(flux_k_LES-flux_k_RANS)/(delt_LES-delt_RANS)*(delt_i2-delt_i1)/dx
sp3d[0,:,:]=sp3d[0,:,:]+xp.minimum(comm_term,0.)*vol[0,:,:]/k3d[0,:,:]
```

20.2.4 modify_om

Inlet boundary conditions

```
su3d[0,:,:]= su3d[0,:,:]+xp.maximum(convw[0,:,:],0)*om_bc_west
sp3d[0,:,:]= sp3d[0,:,:]-xp.maximum(convw[0,:,:],0)
vist=vis3d[0,:,:]-viscos
su3d[0,:,:]=su3d[0,:,:]+vist*aw_bound*om_bc_west
sp3d[0,:,:]=sp3d[0,:,:]-vist*aw_bound
```

and the commutation term

```
prod_extra=-om3d[0,:,:]/k3d[0,:,:]*comm_term
su3d[0,:,:]=su3d[0,:,:]+xp.maximum(prod_extra,0.)*vol[0,:,:]
```

21 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 $\Delta x = 0.03$ and then we stretch it by 0.5%. 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 `1.0, 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 17.

21.1 setup_case.py

21.1.1 Section 1

Hybrid discretization is set for all variables.

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

21.1.2 Section 2

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

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

21.1.3 Section 4

The viscosity is set.

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

21.1.4 Section 6

The `tdma` solver is chosen for k and ω.

```
solver_turb='tdma'
nsweep_kom=1
```

Recall that the number of sweeps should be set to low value since no convergence criteria is used for TDMA.

21.1.5 Section 8

The number of time steps is set to 1000 and the results are time averaged the last 100 time steps (the solution will be steady). A rather large time step is chosen (we are not concerned about time accuracy since we time march to steady state).

```
ntstep=400
uin=1
dt=4*(x2d[1,0]-x2d[0,0])*xp.ones(ntstep)/uin
itstep_start=ntstep-10
```

21.1.6 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_low_type='n'
u_bc_high='n'
v_bc_low_type='n'
v_bc_high='n'
w_bc_low_type='n'
w_bc_high='n'
k_bc_low_type='n'
k_bc_high='n'
om_bc_low_type='n'
om_bc_high='n'
```

Inlet boundary conditions are $\bar{u} = 1$ and $\omega = 1$. For the first 10 cells adjacent to the wall $k = 0.01$ and further out we set $k = 10^{-5}$.

```
    u_bc_west=xp.ones((nj,nk))
    k_bc_west=xp.ones((nj,nk))*1e-2
    k_bc_west[10:,:]=1e-10
    om_bc_west=xp.ones((nj,nk))
```

The wall boundary condition of ω is multiplied by a factor of 10

```
    om_bc_south=10*6*viscos/0.075/dist2_s
```

This – of course – increases the cell center value and makes it closer to the correct value in Eq. 10.2.

21.2 `modify_case.py`

21.2.1 `modify_init`

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

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

    vis3d=k3d/om3d+viscos
```

Run the code and plot the results with `plot_inlet_bound.py` Looking at the time histories of $\bar{u}$, we find that we should maybe run more time steps to really reach steady state.

Now we will use these results as mean inlet boundary conditions in Section 23. 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`.

22 RANS of hump flow using the AKN $k - \varepsilon$ model

The grid is shown in Fig. 22.1. This grid was used in [32].

The inlet boundary condition are taken from a 2D boundary layer. It could have been taken from the results in Section 21 if the extent of the streamwise domain were increased (the inlet momentum Reynolds number should be $Re_\theta = 6\,300$).

The flow is 2D, so we use only two cells in the z direction. The `z.dat` file reads

```
    0.2  2
```

22.1 `setup_case.py`

22.1.1 Section 2

The AKN $k - \varepsilon$ model is used

```
    keps=True
```

The TDMA solver using two sweeps is found to be most efficient for the velocities and k and εall equations

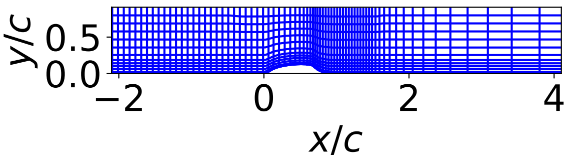


Figure 22.1: Hump flow. The grid. Every 8^{th} grid line is shown.

22.1.2 Section 6

```
solver_vel='tdma'
solver_turb='tdma'
nsweep_vel=2
nsweep_keps=2
```

The `pyamg` solver also works well. 2000 time steps are made using $\Delta t = 0.001$ and time averaging the last 100 time steps

22.1.3 Section 8

```
ntstep=2000
uin=1
dt=0.001*xp.ones(ntstep)
itstep_start=ntstep-100
```

22.2 modify_case.py

22.2.1 modify_inlet

The inlet boundary conditions are taken and interpolated from a RANS boundary-layer simulation using the $k-\omega$ model.

```
   if itstep == 0:
      y_u_k_om=xp.loadtxt('y_u_v_k_om_uv_hump.dat')
      y_rans_in=y_u_k_om[:,0]
      u_rans_in=y_u_k_om[:,1]
      v_rans_in=y_u_k_om[:,2]
      k_rans_in=y_u_k_om[:,3]
      om_rans_in=y_u_k_om[:,4]
      uv_rans_in=y_u_k_om[:,5]
      eps_rans_in=0.09*k_rans_in*om_rans_in

      y_rans=yp2d[0,:]
      u_rans=xp.interp(y_rans, y_rans_in, u_rans_in)
# make it 2D
      u_rans=xp.repeat(u_rans[:,None], repeats=nk, axis=1)
      k_rans=xp.interp(y_rans, y_rans_in, k_rans_in)
# make it 2D
      k_rans=xp.repeat(k_rans[:,None], repeats=nk, axis=1)
      eps_rans=xp.interp(y_rans, y_rans_in, eps_rans_in)
# make it 2D
      eps_rans=xp.repeat(eps_rans[:,None], repeats=nk, axis=1)

      uv_rans=xp.interp(y_rans, y_rans_in, uv_rans_in)

      k_bc_west=k_rans
      eps_bc_west=eps_rans
      u_bc_west=u_rans
```

22.2.2 `fix_eps`

The values of ε are set fo the wall-adjacent cells.

```
# south wall
   aw3d[:,0,:]=0
   ae3d[:,0,:]=0
   as3d[:,0,:]=0
   an3d[:,0,:]=0
   al3d[:,0,:]=0
   ah3d[:,0,:]=0
   ap_max=xp.max(ap3d)
   ap3d[:,0,:]=ap_max
   su3d[:,0,:]=ap_max*2*viscos*k3d[:,0,:]/dist3d[:,0,:]**2
```

Run the code. Check how much time it takes to solve each equation with the command

```
grep time out
```

The output on a Dell Alienware x17 R1 laptop is

```
time one iteration: 7.28e-01
time u: 1.35e-01
time v: 1.26e-01
time w: 1.26e-01
time p: 5.83e-02
time k: 1.39e-01
time eps: 1.34e-01
```

22.3 Using the GPU to solve the equations

Above we use the solvers on the CPU. If you have a Nvidia graphics card, you can solve the equations on the GPU. In `setup_case.py`, Section 6, you should then set

```
   solver_vel='pyamgx'
   solver_turb='pyamgx'
   solver_vel='pyamgx'
   solver_p='pyamgx'
```

Run the code. Then type

```
grep time out
```

The output on a Dell Alienware x17 R1 laptop is

```
time one iteration: 2.91e-01
time u: 4.61e-02
time v: 2.06e-02
time w: 2.01e-02
time p: 8.42e-02
time k: 6.11e-02
time eps: 5.49e-02
```

We find that the CPU time is reduced by a factor of 2.5. We can actually make it even faster. By setting

```
   solver_p='pyamgx_p'
```

the code does uploads the coefficient matrix (see e.g. Eq. B.1) of the pressure equations only once (it depends only on geomeetry). The output on a Dell Alienware x17 R1 laptop is

```
time one iteration: 2.58e-01
time u: 4.46e-02
time v: 1.89e-02
time w: 1.84e-02
time p: 6.04e-02
time k: 5.87e-02
time eps: 5.47e-02
```

We get further speed-up of approximately 12%. The disadvantage is that this option requires more memory on the GPU since the CPU then must store to sparse matrices all the time.

23 IDDES of hump flow using the $k - \varepsilon$ model

This setup was used for simulations presented in [32].

The same grid is used as in Section 22 except that the flow is now three dimensional. 32 cells are used in the z direction and the `z.dat` file reads

```
   0.2  32
```

The grid is shown in Fig. 22.1. It happens to give good results, but that's probably fortuitous; in order to accurately resolve large-scale turbulence it should probably be refined upstream the hump and in the outlet region for $x > 2$. It is fairly easy to do this with a Python script. This has been done in [33].

The settings are very similar to those in Section 22 except that we now use central differencing and synthetic inlet fluctuations.

You find this case in the directory
`hump-IDDES-ni-583-go4hybrid-mesh-vel-tdma-STG-dt-0.003/`

The results are presented in [32].

23.1 setup_case.py

All equationsa are solved in the GPU

23.1.1 Section 6

```
   solver_vel='pyamgx'
   solver_turb='pyamgx'
   solver_p='pyamgx_p'
```

23.2 modify_case.py

23.2.1 modify_inlet

Initial data are taken from the RANS simulation in Section 22.

```
# start from RANS
   itstep,dummy1,dummy2=xp.load('../hump-2D-RANS-AKN-go4hybrid-mesh-inlet-at-x-
   u2d=xp.load('../hump-2D-RANS-AKN-go4hybrid-mesh-inlet-at-x-m2.1/u_averaged.n
   v2d=xp.load('../hump-2D-RANS-AKN-go4hybrid-mesh-inlet-at-x-m2.1/v_averaged.n
   p2d=xp.load('../hump-2D-RANS-AKN-go4hybrid-mesh-inlet-at-x-m2.1/p_averaged.n
   k2d=xp.load('../hump-2D-RANS-AKN-go4hybrid-mesh-inlet-at-x-m2.1/k_averaged.n
   eps2d=xp.load('../hump-2D-RANS-AKN-go4hybrid-mesh-inlet-at-x-m2.1/eps_averag


   vismax=xp.max(0.09*k2d**2/eps2d)
   print('vismax',vismax/viscos)

# change from RANS to LES
   k2d=0.4*k2d

   vismax_les=xp.max(0.09*k2d**2/eps2d)
   print('vismax_les',vismax_les/viscos)


# make 2d to 3d
   u3d=xp.repeat(u2d[:,:,None], repeats=nk, axis=2)
   v3d=xp.repeat(v2d[:,:,None], repeats=nk, axis=2)
   p3d=xp.repeat(p2d[:,:,None], repeats=nk, axis=2)
   k3d=xp.repeat(k2d[:,:,None], repeats=nk, axis=2)
   eps3d=xp.repeat(eps2d[:,:,None], repeats=nk, axis=2)
   w3d=xp.zeros((ni,nj,nk))
```

23.2.2 modify_fk

The IDDES model is implemented here

```
  global f_e_mean, f_b_mean, f_dt_mean

   l_dist=0.15*dist3d
   l_max=0.15*delta_max
   dy=xp.diff(y2d[1:,:],axis=1)
```

```
# make it 3d
   dy=xp.repeat(dy[:,:,None],repeats=nk,axis=2)

   l_temp=xp.maximum(l_dist,l_max)
   l_temp=xp.maximum(l_temp,dy)
   l_iddes=xp.minimum(l_temp,delta_max)
#  l_iddes=xp.minimum(xp.maximum(l_dist,l_max,dy),delta_max)

   ueps=(eps3d*viscos)**0.25
   ystar=ueps*dist3d/viscos
   rt=k3d**2/eps3d/viscos
   fdampf2=((1.-xp.exp(-ystar/3.1))**2)*(1.-0.3*xp.exp(-(rt/6.5)**2))
   fmu=((1.-xp.exp(-ystar/14.))**2)*(1.+5./rt**0.75*xp.exp(-(rt/200.)**2))
   fmu=xp.minimum(fmu,1.)

   psi=xp.minimum(10,(fdampf2*fmu)**(-0.75))

   l_c=psi*cdes*l_iddes  #eq. 9

   vist=vis3d-viscos
   denom=kappa**2*dist3d**2*gen**0.5

   r_dt=vist/denom  #eq. 22
   r_dl=viscos/denom  #eq. 23

   f_t=xp.tanh((c_t**2*r_dt)**3)
   f_l=xp.tanh((c_l**2*r_dl)**10)

   f_e2=1.-xp.maximum(f_t,f_l) #eq. 19

  alpha=0.25-dist3d/delta_max

   f_e1=xp.where(alpha <= 0,2*xp.exp(-9*alpha**2),2*xp.exp(-11.09*alpha**2))

   f_b=  xp.minimum(2.*xp.exp(-9*alpha**2),1.)

   f_dt=1.-xp.tanh((8.*r_dt)**3)

   f_e=xp.maximum(f_e1-1.,0.)*psi*f_e2

   f_d=xp.maximum((1.-f_dt),f_b)

   l_u=k3d**1.5/eps3d

#  l_tilde=f_d*(1+f_e)*l_u+(1-f_d)*l_c
   l_tilde=f_d*l_u+(1-f_d)*l_c

   fk3d=l_u/l_tilde

   # fk3d=xp.minimum(fk3d,c_eps2/c_eps_1) # limie on psi
```

23.2.3 modify_k

A commutation term is added to the k equation.

```
   global u2prim_i0,v2prim_i0,w2prim_i0,umean_i0

   if iter == 0:
# set running-averaged inlet values to zero
      if itstep == 0:
         u2prim_i0=xp.zeros(nj)
         v2prim_i0=xp.zeros(nj)
         w2prim_i0=xp.zeros(nj)
         umean_i0=xp.zeros(nj)

# time average
      u2prim_i0=u2prim_i0+xp.mean(u3d[0,:,:]**2,axis=1)
      v2prim_i0=v2prim_i0+xp.mean(v3d[0,:,:]**2,axis=1)
      w2prim_i0=w2prim_i0+xp.mean(w3d[0,:,:]**2,axis=1)
      umean_i0=umean_i0+xp.mean(u3d[0,:,:],axis=1)

   comm_term=xp.zeros((nj,nk))
   umean=umean_i0/(itstep+1)
   u2prim=u2prim_i0/(itstep+1)-umean**2
   v2prim=v2prim_i0/(itstep+1)
   w2prim=w2prim_i0/(itstep+1)
   k_tot=0.5*(u2prim+v2prim+w2prim)+xp.mean(k3d[0,:,:],axis=1)
# make it 2D
   k_tot=xp.repeat(k_tot[:,None], repeats=nk, axis=1)

   psi_small=fk3d[0,:,:]
   term1=xp.maximum((c_eps_2-c_eps_1*psi_small)/(c_eps_2-c_eps_1),1e-10)
   fk2d_from_psi=term1**0.333

   dfk_dx=u3d[0,:,:]*(fk2d_from_psi-1)/(x2d[1,1]-x2d[0,1])
# commutation term
   comm_term_pans=k_tot*dfk_dx
   comm_min_pans=xp.min(comm_term_pans)
   pk_max=xp.max((vis3d-viscos)*gen)

   u2prim_max=xp.max(u2prim)
   v2prim_max=xp.max(v2prim)
   w2prim_max=xp.max(w2prim)

   print(f"\n{'comm_min_pans: '} {comm_min_pans:.2e}, {'pk_max: '}{pk_max:.2e},
   sp3d[0,:,:]=sp3d[0,:,:]+xp.minimum(comm_term_pans,0)*vol[0,:,:]/k3d[0,:,:]
```

Run the code and look at the CPU time by typing

```
grep time out
```

The output on a Dell Alienware x17 R1 laptop is

```
time one iteration: 7.18e+00
time u: 1.33e+00
time v: 1.06e+00
time w: 1.05e+00
time p: 1.17e+00
time k: 1.46e+00
time eps: 1.35e+00
```

23.3 Using the GPU to solve the equations

Above we use the solvers on the CPU. If you have a Nvidia graphics card, you can solve the equations on the GPU. In `setup_case.py`, Section 6, you should then set

```
   solver_vel='pyamgx'
   solver_turb='pyamgx'
   solver_vel='pyamgx'
   solver_p='pyamgx_p'
```

Run the code and look at the CPU time by typing

```
grep time out
```

The output on a Dell Alienware x17 R1 laptop is

```
time one iteration: 4.58e+00
time u: 8.57e-01
time v: 2.30e-01
time w: 2.15e-01
time p: 8.21e-01
time k: 1.02e+00
time eps: 9.23e-01
```

We find that solving the equations on the GPU gives a speed-up of approximately 1.6.

24 IDDES of hump flow running 100% on the GPU

In order to run the case in Section 23 we must make a few modifications in `setup_case` and `modify_case`.

24.1 `setup_case.py`

We introduce a new Section at the beginning where we tell the code to run on the GPU.

24.1.1 Section 0

```
   gpu = True
```

24.1.2 Section 6

The pyAMGX solver which is available on the GPU is very efficient. Hence we choose to use this for all equations.

```
solver_vel='pyamgx'
solver_turb='pyamgx'
solver_p='pyamgx_p'
```

Note that I chose `pyamgx_p` for the pressure. That means that the matrix $(a_W, \ldots A_H)$ is computed only once. This option increases the memory requirement.

25 Developing boundary layer

Here we simulate developing flow boundary layer flow along a flat plate at inlet Reynolds number based on the momentum thickness, $Re_\theta = U_{free}\theta/\nu = 2\,550$. The data presented below are averaged in z direction and time.

The inlet boundary-layer thickness is $\delta_{in} \sim 0.78$. The far-field mean velocity is one, i.e. $U_{free} = 1$, and the time step is 0.043. The mean inlet profiles are taken from a 2D RANS solution at $Re_\theta = 2\,500$. Synthetic fluctuations [34, 35] are superimposed to the RANS velocity profile.

The mesh has $700 \times 90 \times 64$ cells (x, y, z). The domain size is $87 \times 2.9 \times 1.6$. $\Delta x^+_{in} = 110$, $\Delta z^+_{in} = 31$ and $y^+_{in} = 0.5$ for the center of the wall-adjacent cell. A stretching of 10% is used in the wall-normal direction but Δy is not allowed to be larger than 0.05.

You find the case in directory
`boundary-layer-IDDES-ni-700-nk128-2zmax-synt-commut-with-fk-psi-eps-neumann`

The settings in `setup_case` and `modify_case` are almost identical to those in Section 23. The main difference is that here I use Neumann indet b.c. on ε instead of adding a commutation term. This has a negligible influence on the predicted results.

26 Workshop

In this section you will get familiar to use and modify the **pyCALC-LES** code. We start by doing some simple RANS simulations. Note that you should **not** use any `for` loops because in Python they are very slow. An exception may be the grid generator and plotting scripts in which the CPU time is not an issue.

26.1 Getting started

Unpack the `pyCALC-LES-course-june-202x.tgz` (`x=2, 3, ...`). Go into the directory `pyCALC-LES-course-june-202x.tgz`.

First, you must make sure that you have installed the required Python packages. I recommend that you install Python in your home directory using Anaconda.

Run the Python script `pyamgx_solve_random_matrix.py`. This script will test if you have installed the required sparse-matrix solvers, the algebraic multigrid solver `pyamg`, and algebraic multigrid solver on the GPU`pyamgx`. Do the test by typing

```
run pyamgx_solve_random_matrix.py
```

If everything works you will see

```
************************ gmRES solver works
************************ pyAMG solver works
************************ pyAMGx solver works
```

Probably, neither `pytamg` not `pytamgx` are installed on your system. On Ubuntu, I installed `pyamg` with the command

```
conda install -c anaconda pyamg
```

For installation of `pyamgx`, see Section C.

26.2 Channel flow, RANS

Go to the directory `channel-5200-k-omega-RANS` (or something similar). Here RANS simulations of fully-developed channel flow will be studied. Look at `setup_case.py` and `modify_case.py`; the input is briefly described in Section 17. Plot the results using the script `pl_uvw_RANS.py`. I'm using the Python interface `ipython`. So I would type

```
ipython
```

and then

```
run pl_uvw_RANS.py
```

If you like the vi editor – as I do – then you can from `ipython` edit the script using the command

```
!vim pl_uvw_RANS.py
```

Below I give some examples of how you can modify this flow. You may do all or only a few. The object is that you should get familiar with the code and do some fast simulations. Create a new directory (below the directory where **pyCALC-LES** resides). Copy all files from `channel-5200-k-omega-RANS` into this new directory.

26.2.1 New grid

The grid is generated using the script `generate-channel-grid.py`. 96 cells are used in the y direction with a stretching of 15%. It gives a y^+ value of approximately 0.5 for the wall-adjacent cell center. Modify the number of cell and/or the stretching and look at the influence. You execute the grid script by typing

```
python generate-channel-grid.py
```

Now a new grid is generated (it is written to `x2d.dat` and `y2d.dat`) which is read by **pyCALC-LES**. Now run **pyCALC-LES** by typing

```
run-pyton &
```

This is a bash script which simply puts the four Python scripts `setup_case.py`, `modify_case.py`, `../pyCALC-LESp.py` and `../synt_flucy.py` (together with the declarations of global variables in file `../globals`) into one file called `exec-pyCALC-LES.py` and then runs this file. In Section 15.3, you find some useful information on how to extract convergence history etc from the output file `out`.

26.2.2 Boundary wall conditions on ω

The wall boundary conditions on ω are set in Section 10 in `setup_case.py` according to Eq. 10.2. This is not entirely correct, because it prescribes ω *at* the wall, whereas it should be prescribed at the cell center. With the present boundary condition, the value of ω at the cell center will be too small. Try to compensate this by increasing the value of ω at the wall by a factor of `fact=10`.

When you edit the code you may do it in two ways. Either you edit `setup_case.py` and then execute the code with the `run-python` script. Or you edit the file `exec-pyCALC-LES.py` directly. If you do it with `ipython` you type

```
ipython
```

and then

```
!vim exec-pyCALC-LES.py
run exec-pyCALC-LES.py
```

You can insert breaks in the code by inserting the command `sys.exit()`.

Now, do you get the correct value of ω at the wall-adjacent cell center? Or is it still too small? If so, increase `fact`.

Another way is to prescribe ω at the cell center using sources S_P and S_U. In standard SIMPLE finite volume methods, this is usually done setting S_U and S_P to large values, i.e

$$S_P = -10^{10}, \quad S_U = 10^{10}\omega_{wall}$$

where ω_{wall} is the wall boundary condition. However, this option does not work in **pyCALC-LES** (the simulations diverges rapidly), probably because the advanced solvers do not tolerate the resulting large condition number of the solution matrix.

Instead, at the wall-adjacent cells, we simply set all coefficients, $a_w, a_E, \ldots a_H$ to zero, $a_P = 1$ and $S_u = \omega_{wall}$. You do this in the module `fix_omega` in file `modify_case.py`

```
def fix_omega():
   aw3d[:,0,:]=0
   ae3d[:,0,:]=0
   as3d[:,0,:]=0
   an3d[:,0,:]=0
   al3d[:,0,:]=0
   ah3d[:,0,:]=0
   ap3d[:,0,:]=1
   su3d[:,0,:]=om_bc_south
```

Setting $a_p = 1$ may not be optimal, since this value may be much larger/smaller than a_p at other cells. It's probably better to set

$$a_{P,max} = \max(a_P)$$

where $\max$ is taken over all cells and $S_u = a_{P,max}\omega_{wall}$; this approach makes the condition number of the coefficient matrix smaller.

Note, that the procedure of setting the coefficients $a_W, a_E, \ldots$ cannot by done in `modify_om`, since the a_P and S_U are modified in module `crank_nicol` after leaving `modify_om`. You must use the module `fix_omega` (in file `modify_case.py`) which is called just before the solver is called. Implement the boundary condition (i.e. setting ω_{wall} as the wall-adjacent cell value) and find out how large the effect is on the results.

26.2.3 $k-\varepsilon$ model

Now simulate the same flow with the AKN $k-\varepsilon$ model. You set `keps=True` and `kom=False`. You need to set the wall boundary for ε according to Eq. 8.4. Do that in module `fix_eps`. You can look at the `modify_case` file for the hill flow, see Section 18.

The default initial values are set in the main code for k and ε, i.e. $k = \varepsilon = 1$.

Check the convergence by typing

```
grep 'max resi' out
```

You find that the simulation diverges. Change solver to TDMA and set the number of sweeps to one

```
    solver_vel='tdma'
    solver_turb='tdma'
    nsweep_vel=1
    nsweep_keps=1
```

Run the code. Check convergence. Plot the results. They don't look too good, do they? If you look at the time histories you see there are large oscillations. Decrease the time step (Section 8 in `setup_case.py`) by a factor of four and make a corresponding increase in number of time steps. Run again and you find it looks better.

How do the results compare with $k-\omega$ model? Try different grids. Is the $k-\varepsilon$ more or less sensitive to the near-wall refinement than the $k-\omega$ model?

26.3 Boundary layer flow, RANS

Read Section 21 carefully. This is a developing boundary layer flow. At the inlet, $\bar{u} = 1$, $\omega = 1$ and $k = 10^{-2}$ near the wall (first 10 cells) and $k = 10^{-10}$ in the outer region. This flow case can be used for creating mean inlet profiles for the DES simulations in Section 23 (but you need to increase `ni`). Neumann boundary conditions are used at the free (north) boundary for k and ω. Do some sensitivity checks.

- Is the flow sensitive to the inlet values of k and ω?
- The TDMA solved is used for k and ω.
  - Check the CPU time by typing
    `grep itera out`
    which gives the CPU time per iteration. If you type
    `grep time out`
    you get the CPU time for each variable (per iteration)
  - What happens if you use the LGMRES solver? Remember to set

```
nsweep_kom=50
```

Check maximum turbulent viscosity by typing
`grep vismax out`

- What happens if you set Dirichlet boundary at the free boundary (the $k-\omega$ model is known to be sensitive to free-stream values of ω)
- The ω_{wall} value is set to $10\omega_{wall}$, See Section 10 in `setup_case.py` What happens if you fix is to ω_{wall} in the center of the cell (as you did in Section 26.2.2).

26.4 Channel flow, inlet-outlet, $Re_\tau = 395$

Now we will – finally – do some LES. The setup of this flow is given in Section 19. Read this section carefully, look at the file `out` and plot the results. Now create a new directory and copy all files.

In order to make the simulations quicker, you can make the domain smaller and use shorter integration times. You can also choose to make simulations only in the lower half of the channel using a symmetry boundary condition at the upper (north) boundary. Note that by doing this we modify the physics, but the influence will probably be limited to the region near the upper boundary.

So, let's change the domain and generate a new grid with extent $x = [0, 4]$ and $y = [0, 1]$ with $ni = 44$ and $nj = 40$. The `generate-channel-grid.py` mirrors the grid in the upper half; remove that part (since we want to create a grid only for the lower half). Modify the script `generate-channel-grid.py` accordingly.

Next, we need to change the boundary conditions at the upper (north) boundary from Dirichlet to Neumann (note that it should be changed for all variables except one). You do that in file `setup_case.py` in Section 10.

In the full channel (i.e. $y_{max} = 2$ in Section 19), the inlet shear stress profile created by the synthetic fluctuations is negative in the lower (south) half and positive in the upper (north) half. We change the sign of the inlet shear stress in the upper half by switching the sign of v' in Eq. 11.8, see Section 19.1.5. In this case, we compute the flow only in the lower half of the channel and hence we set `jmirror_synt=0` (also in Sectioon 10).

In file `modify_case.py` you should look for `modify_init` and `modify_inlet`. Here the variables `y_rans`, `u_rans` and `uv_rans` are used. The length of the loaded vectors are that of the full channel. But now we must use only the values in the lower half of the channel, e.g.

```
y_rans=y_u_k_om[0:nj,0]
u_rans=y_u_k_om[0:nj,1]
uv_rans=y_u_k_om[0:nj,4]
```

Finally, reduce the start of time integration and number of time steps to

```
itstep_start=2000
ntstep=6000
```

Also, if you have managed to install `pyamgx`, you can solve set

```
solver_p='pyamgx_p'
solver_vel='pyamgx'
solver_turb='pyamgx'
```

Now run the code. On my Dell Alienware x17 R1 laptop the simulation takes 10 minutes. By loosening the convergence limits in the Pythons solvers (e.g. 10^{-4} for velocities and $5 \cdot 10^{-3}$) you can make the simulation even faster. Plot the results (you'll find that you must make some modifications of the plot script) and compare with the original results. The most critical quantities are the friction velocity and the resolved shear stress. The profiles of the resolved stresses are non-smooth because of too short a time averaging. Increase `ntstep` and `itstep_start` if you prefer smoother profiles.

Now investigate how sensitive the flow is to various parameters.

- The number of synthetic modes is set to `nmodes_synt=1200`. What happens if you increase or decrease it? What about the CPU time?
- The SGS viscosity is plotted. You find that $\nu_{sgs}/\nu \simeq 1$. We use the WALE model. What happens if you switch to DNS?
- The integral turbulent length scale of the synthetic fluctuations is set to `L_t_synt=0.2`. What happens if you increase/decrease it? Do you get the same effect as in [23]?
- Can you increase the time step? If so, you can reduce the integration time. Is the CPU time/time step the same for the larger time step (type `grep time` at the prompter)? Can you loosen the convergence criteria?
- The integral turbulent timescale of the synthetic fluctuations is set to L_t/u_τ (see `tturb=L_t_synt` in `modify_inlet`). Note that this value gives `a_synt=0.994` and `b_synt=0.108` (see file `out`) which correspond to a and b in Eqs. 11.10 and 11.11 (hence only a small contribution from the "new" fluctuation in the time filter, Eq. 11.9). What happens if you increase/decrease the integral timescale?
- The eigenvalues and the eigenvectors for the synthetic fluctuations are read in module `synt_fluct`. It reads the files `a_synt_inlet.dat` and `R_synt_inlet.dat`. The eigenvalues and the eigenvectors have been computed using a Reynolds stress tensor created with EARSM and a 1D RANS simulation. They were computed using the script `compute_a_and_R-from-earsm.py`. Try another Reynolds stress tensor (e.g. from DNS). This task in **optional**.
- Change any other parameters. For example, you can make more changes in the synthetic fluctuation generator (file `../synt_fluct.py`).

26.5 Channel flow, inlet-outlet with heat transfer, $Re_\tau = 395$

Now we can add a new transport equation: a temperature equation. If you're more interested in the $k - \omega$ DES turbulence model, skip this section. You can make a LES simulation or if you want to faster simulation, then use RANS. In either way, copy the files in Section 26.4. For RANS, you skip the inlet fluctuations and you can reduce the number of cells to two in the z direction, and you can choose, e.g., the $k - \omega$ model. Or you can even do laminar flow (don't forget to reduce the Reynolds number).

When we add a new transport equation, it means that you have to make changes in the main code. i.e. `pyCALC-LES.py`. I suggest that you copy that file into a new name, e.g. `pyCALC-LES-heat.py`. Then you need to change the `run-python` file so that it 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-heat.py  > exec-pyCALC-LES.py;
~/anaconda3/bin/python  -u  exec-pyCALC-LES.py > out
```

Now you need to define many new variables in file `globals` such as
`t_bc_east, t_bc_east_type, t_bc_north, t_bc_north_type, t_bc_south, t_bc_south_type, t_bc_west, t_bc_west_type, t_bc_high, t_bc_low, t_bc_high_type, t_bc_low_type`.

You need to initialize temperature (search for the string `u3d=xp.one` in `pyCALC-LES-heat.py`) with the command

```
t3d=xp.ones((ni,nj,nk))*1e-20
```

Then you need to create a new routine for temperature, `calct`. You need to call `coeff`, `bc` .... You can, for example, copy the lines used for `v3d` (search for the string `calcv` in `pyCALC-LES-heat.py`). You need to define a viscous Prandtl number (`prand` is a turbulent one). You can add one parameter (e.g. `prand_visc` in the call to `coeff`; don't forget to add `prand_visc` to the file `global`).

You must also create a `modify_t` in file `modify_case.py`.

Now, set boundary conditions and try it out! (it will most likely not work right away). I suggest that you use $T = 0$ at the inlet. Then set some Dirichlet b.c. at the wall. Next, you may set some internal heat source in, for example, the cells `(i,j,:)=(5,10,:)`. You do this with the command in `modify_t`

```
su3d[5,10,:]= su3d[5,10,:]+ss*vol[5,10,:]
```

where `ss` is the source per unit volume.

26.5.1 Adding buoyancy

Maybe you want to add buoyancy. We choose the vertical direction as y. That means that we should add the buoyancy term to the $\bar{v}$ momentum equation which reads

$$g\beta(T - T_{ref}) \tag{26.1}$$

see, e.g., Section 11.1 in [24]. β is the thermal expansion coefficient and g is the gravitational acceleration which are set to $1/273$ and $g = 0.81$, respectively. We set the reference temperature to zero, i.e. $T_{ref} = 0$. Now you simply add Eq. 26.1 to `su3d` in module `modify_v` (don't forget to multiply by volume).

26.6 RANS of channel flow at $Re_\tau = 5\,200$: $k-\omega$ and wall functions

Here we will implement wall functions and make RANS simulations of fully developed channel flow. Copy all files used in Section 17. When wall functions are used we place the wall-adjacent cell centers in the log-region, i.e. approximately at $30 \leq y^+ \leq 200$. So we start by generating a new grid using `generate-channel-grid.py`. Set `nj=50` and make all Δy equal. You can achieve this by setting the stretching factor to one, i.e. `yfac=1`. The wall boundary conditions for $\bar{u}$, k and ε are given in Section 11.14.1 in [24]. They can be summarized as

$\bar{u}$: set the wall shear stress as $\tau_w = \rho u_\tau^2$ (recall that $\rho = 1$). The log-law reads

$$\frac{\bar{u}}{u_\tau} = \frac{1}{\kappa} \ln\left(\frac{E u_\tau y}{\nu}\right) \tag{26.2}$$

where $E = 0$ and $\kappa = 0.41$.

k: set k at the wall-adjacent cells as $k_P = C_\mu^{-1/2} u_\tau^2$

ω: set ω at the wall-adjacent cells as $\omega_P = C_\mu^{-1/2} u_\tau/(\kappa y_{wall})$, see Eg. 3.27 in [36]

Here are some tips.

- The wall force (wall shear stress times area), $\tau_w A_s$, should always be in the opposite direction to the local $\bar{u}$ velocity. Hence, it is best to add $\tau_w A_s/|\bar{u}|$ to `sp3d`. Since the wall boundary condition is implemented as a force, there should be no diffusion from the wall via `as3d` and `an3d`. Hence, set Neumann boundary conditions for $\bar{u}$.

- When setting the wall-adjacent ω according to the expression above, use the module `fix_omega` in file `modify_case.py`.

- Add a new module `fix_k` for setting k. Add a call to `fix_k` in the main iteration loop of **pyCALC-LES** in a similar way as the calls to `fix_eps` and `fix_omega`

- The expression for u_τ in the log-law (Eq. 26.2) is implicit, Hence, compute u_τ from the log-law in an iterative way (you could make 3–5 iterations using u_τ from the previous global iteration as initial value).

- Print u_τ at every time step; it is a good check to see if it's correctly computed. It should go to one (it takes at least 1000 time steps).

- Finally, when you plot the results using `pl_uvw.py`. The friction velocity is here computed as

$$u_\tau = \left(\nu \frac{\partial \bar{u}}{\partial y}\bigg|_{wall}\right)^{1/2}$$

Now you should compute is from the wall functions (you can compute it from k)

26.7 Channel flow, inlet-outlet, $Re_\tau = 5\,200$

Here we will make simulations with inlet-outlet boundary conditions using a $k - \omega$ DES turbulence model. Create a new directory and copy the files from the case in Section 20. Make the same modifications as in Section 26.4. Run the code, plot and compare with the results in Section 20.

26.7.1 Neumann boundary condition on k

The discretized commutation term in the k equation is in effect a negative convection term [30]. Hence, we should get the same results if we omit the commutation term in the k equation and change the inlet Dirichlet boundary condition on k to Neumann (cf Figs. 6 and 9 in [30]). Make the changes, run the code and compare the results with those in Section 26.7.

26.7.2 No commutation terms

- What happens if you keep Dirichlet inlet boundary conditions on k and ω and omit the commutation terms?
- What happens if omit the commutation terms and use Neumann inlet boundary conditions on both k ans ω?

26.7.3 No commutation terms in URANS region

As discussed in [30], the commutation terms should maybe not be used in the URANS region. First, find out where the switch between URANS and LES occurs. Then, make a simulation where you use the commutation terms only in the LES region. Run the code. How do the results compare with those in Section 26.7?

26.8 Channel flow, inlet-outlet, $Re_\tau = 5\,200$, using wall functions

Implement wall functions in the same way as Section 26.6. Copy all files from Section 26.7. Modify the grid, `setup_case` and `modify_case` in the same way as in Section 26.6.

If you would do turbulent, atmospheric boundary layer, you would use a similar wall functions but instead of the friction velocity we use roughness length, see, e.g., Eq. 14 in [37]

26.9 Channel flow, fully developed, $Re_\tau = 5\,200$

Now we'll replace the inlet-outlet boundary conditions with cyclic boundary conditions. This will be the same flow as in Section 16 but now we compute the flow only in the lower (south) half of the channel. Copy the files from Section 26.7. In Section 10 in `setup_case.py`, set

```
cyclic_x = True
```

This means that the results from 26.7 will be used as initial conditions stored in files `u3d_saved.npy`, v3d_saved.npy. ...om3d_saved.npy. You don't need to change `u_bc_west_type`, `u_bc_east_type` ...`om_bc_east_type`.

We must have reasonably good initial condition. A good way it to use the results in Section 26.7 as initial condition. Hence, simply set

```
restart = True
```

in Section 3 in `setup_case.py`.

Remove all initial, inlet and outlet conditions in `modify_case.py`. Then add the driving pressure gradient source term in `modify_u`

```
    su3d = su3d +vol
```

Run the code. It may take some time for the flow to get fully developed. When you plot the results, check how large u_τ is (or τ_w). It should be equal to one (because τ_w must balance the pressure gradient, see Section "Force balance, channel flow" in [24]). If it is 5% too small or too large, run the code again (i.e. run another `ntstep` time step). How do the results compare with those in Section 16?

26.9.1 Wall boundary condition of ω

In Section 26.2.2 you investigated the sensitivity of the flow to the wall boundary condition of ω. You compared three different boundary conditions.

1. Equation 10.2 (this is what you used in Section 26.9)
2. Multiply Eq. 10.2 by a factor of 10.
3. Set Eq. 10.2 in the cell center by using the module `fix_omega`.

Make two new runs where you apply the two last options. Are the results much affected? For option 1 and 2, how much do the computed ω values differ from the correct value in Eq. 10.2?

26.9.2 RANS-LES Interface

Check where the RANS-LES interface is located (it is stored in variable `fk3d` which is computed in module `compute_fk`). The interface is defined as the location where `fk3d` gets larger than one.

1. Investigate the sensitivity to the location of the interface by forcing it to a certain cell layer of constant j_{l0}. This is done by setting the `jl0` to a negative value, i.e. `xp.abs(jl0)`=j_{l0}.
2. The LES length scale is Δ, see Eq. 10.1. Replace Δ by the IDDES length scale, Δ_{dw}, see Section 23 or Eq. 8 in [31]. Note that you must not use any `for` loops. Run the code and compare with the results obtained in Section 26.9.

26.9.3 Change turbulence model

Up to now, you have used the standard Wilcox $k-\varepsilon$ model. Now switch to the $k-\omega$ model used in [31]

27 Machine Learning for improving wall functions

Here we will use Machine Learning (ML) for improving wall functions. Start by reading my report [38].

The codes in two directories are used.

27.1 Directory 1

The name of the directory is
`channel-5200-IDDES-96-86-96-ML-aver-xz-database-3cells-15000-timesteps`.

Here IDDES is used to make fully-developed channel flow at $re_\tau = 5\,200$, see Section 2 & 3 in [38]. Files with time series of independent samples at nine cells are created in module `modify_u`.

Next, `svr` is used to create a ML wall-function model in file
`svr-C-10-eps-0.001-low-re-yplus-inst-first-uplus-output-cell-1-9-local-300-samples.py`

It creates three files (which is the ML model):

- `model-low-re-svr-C-10-eps-0.001-yplus-inst-uplus-output-first-cell-1-9-local-cells-300-samples.bin`
- `model-low-ustar-svr-C-10-eps-0.001-yplus-inst-uplus-output-first-cell-1-9-local-cell_scaler-yplus-300-samples.bin`
- `min-max-model-low-re-svr-C-10-eps-0.001-yplus-inst-uplus-output-first-cell-1-9-local-cells-loca-300-samplesl.txt`

27.2 Directory 2

The name of the directory is
`channel-16000-IDDES-wall-functions-nj92-ML-dy-from-database-ni-96-svr-C-10-eps-0.001-low-re-IDDES-yplus-inst-uplus-output-cell-1-9-300-samples`

Here, the wall-function ML model is used to predict fully-developed channel flow at $Re_\tau - 16\,000$. The ML model created in Section 27.1 is loaded in module `fix_k`. The friction velocity, u_τ, is predicted with the ML wall-function model. It is then used to set $k = C_\mu^{-1/2} u_\tau^2$ in the wall-adjacent modes. The friction velocity is also used

- in module `fix_eps`: set $\varepsilon = \frac{u_\tau^3}{\kappa \delta y}$ in the wall-adjacent modes
- in module `modify_u`: set $\tau_w = u_\tau^2$ at the wall in the wall-adjacent modes

A Variables in pyCALC-LES

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_cycle`: type of cycle in the pyAMG solver for the pressure equation ('V', 'W', 'F', 'AMLI')

`amg_cycle_phi`: type of cycle in the pyAMG solver for all equations except the pressure equation ('V', 'W', 'F', 'AMLI')

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

`amg_relax_phi`: relation method in pyAMG for all equations except the pressure equation: '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_S 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)

`blend`: blending between central differencing (CDS) and MUSCL. `blend=1` is full CDS, see Section 3.5

`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_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 (max(limit,limit$\cdot$ norm(su3d)); if negative: abs(limit)) `convergence_limit_om, convergence_limit_k, convergence_limit_om:` convergence limit in Python solver for ε, k, ω (max(limit,limit$\cdot$ norm(su3d)); if negative: abs(limit))

`convergence_limit_p:` convergence limit in Python solver for $\bar{p}$ (relative limit); when the `pyamgx` solver is used for any variable, this variable is used as the convergence criterium

`convergence_limit_u:` convergence limit in Python solver for $\bar{u}$ (max(limit,limit$\cdot$ norm(su3d)); if negative: abs(limit))

`convergence_limit_v:` convergence limit in Python solver for $\bar{v}$ (max(limit,limit$\cdot$ norm(su3d)); if negative: abs(limit))

`convergence_limit_w:` convergence limit in Python solver for $\bar{w}$ (max(limit,limit$\cdot$ norm(su3d)); if negative: abs(limit))

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

`dist3d:` smallest distance to south or north wall

`dmin_synt:` the length defining the maximum wavenumber in the synthetic fluctuations, see Section 11.3

`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 time step

`dt:` time step

`dz3d:` grid spacing in the z direction (3D array)

`dz:` grid spacing in the z direction (1D array)

`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_high, eps_bc_low:` boundary values of ε at east, north, south, west and high/low boundary. Default: 0

`eps_bc_east_type, eps_bc_north_type, eps_bc_south_type, eps_bc_west_type:` see below

`eps_bc_high_type, eps_bc_low_type:` type of b.c. for ε ('d'=Dirichlet, 'n'=Neumann' or '2'=$\partial^2\varepsilon/\partial n^2 = 0$). Default: Neumann

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

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

`fkmin_limit`: minimum f_k in PANS and PITM, see Eq. 8.3

`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)

`gpu`: if TRUE, all simulations are made on the GPU; if FALSE, all simulations are made on the CPU

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

`iter`: current global iteration

`itstep`: current time step

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

`itstep_start`: time averaging starts

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

`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 at 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_high, k_bc_low`: boundary values of k at east, south, west, north, and high/low boundary. Default: 0

`k_bc_east_type, k_bc_north_type, k_bc_south_type, k_bc_west_type`: see below

`k_bc_high_type, k_bc_low_type`: type of b.c. for k ('d'=Dirichlet, 'n'=Neumann' or '2'=$\partial^2 k/\partial n^2 = 0$). Default: Dirichlet

`keps`: the AKN $k-\varepsilon$ 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`: length scale of the synthetic fluctuations, see Eq. 11.4

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

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

`nmodes_synt`: number of modes when generating synthetic fluctuations `norm_order`: order of norm when computing residual for $\bar{u}$, $\bar{v}$, $\bar{w}$, k, ε and ω. Default: 2

`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 ω 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 time steps

`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_high, om_bc_low`: boundary values of ω at east, north, south, west and high/low boundary. Default: 0

`om_bc_east_type, om_bc_north_type, om_bc_south_type, om_bc_west_type`: see below

`om_bc_high_type, om_bc_low_type`: type of b.c. for ω ('d'=Dirichlet, 'n'=Neumann' or '2'=$\partial^2\omega/\partial n^2 = 0$). Default: Dirichlet

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

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

`p_bc_east, p_bc_north, p_bc_south, p_bc_west, p_bc_high, p_bc_low` boundary values of $\bar{p}$ at east, north, south, west, and high/low boundary. Default: 0

`p_bc_east_type, p_bc_north_type, p_bc_south_type, p_bc_west_type`: see below

`p_bc_high_type, p_bc_low_type`: type of b.c. for $\bar{p}$ ('d'=Dirichlet, 'n'=Neumann' or '2'=$\partial^2 p/\partial n^2 = 0$). Default: Neumann

`pans`: PANS (based on $k - \varepsilon$) or PITM is used. PANS is used when `prand_k` and `prand_eps` are positive, otherwise PITM

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

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

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

`restart`: a restart from a previous simulaton is made, see Section 14.24

`save`: the $\bar{u}$, $\bar{v}$ ... fields are saved to disk, see Section 14.25

`save_average_z`: when averaging flow variables in time, average also in z direction. Default: True

`scheme`: discretization scheme for the $\bar{u}$, $\bar{v}$ and $\bar{w}$ equation. 'c'=central, 'h'=hybrid, 'u'=upwind or 'm'=MUSCL, see Section 14.10

`scheme_turb`: discretization scheme for k, ε and ω. 'c'=central, 'h'=hybrid, 'u'=upwind, see Section 14.10

`smag`: the Smagorinsky model is used

`solver_p`: pyAMG solver for $\bar{p}$. solver_p='pyamgx' means that tke $\bar{p}$ equation is solved on the GPU. The coefficient matrix, A (see Eqs.B.4 – B.4), is uploaded to the GPU every iteration; solver_p='pyamgx_p' means that the matrix, A, is uploaded only once. This option is faster but requires twice as much GPU memory.

`solver_turb`: Python sparse matrix or pyAMG solver for k, ε and ω. solver_turb='pyamg', 'pyamgx' (solved on the GPU), 'gmres', 'lgmres', 'cgnr', 'cgne', 'fgmres', 'bicgstab', 'tdma'

`solver_vel`: Python sparse matrix or pyAMG solver for $\bar{u}$, $\bar{v}$ and $\bar{w}$. solver_vel='pyamg', 'pyamgx' (GPU), 'gmres', 'lgmres', 'cgnr', 'cgne', 'fgmres', 'bicgstab', 'tdma'

`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, $\langle\bar{u}\rangle$

`u_bc_east, u_bc_north, u_bc_south, u_bc_west, u_bc_high, u_bc_low`: boundary values of $\bar{u}$ at east, north, south, west, and high/low boundary. Default: 0

`u_bc_east_type, u_bc_north_type, u_bc_south_type, u_bc_west_type`: see below

`u_bc_high_type, u_bc_high_low`: type of b.c. for $\bar{u}$ ('d'=Dirichlet, 'n'=Neumann' or '2'=$\partial^2 u/\partial n^2 = 0$). Default: Dirichlet

`urfvis`: under-relaxation factor for turbulent viscosity

`usynt_inlet`: synthetic inlet fluctuation in the x direction, $(\mathcal{V}'_1)_m$, see 11.9

`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, $\langle\bar{v}\rangle$

`v_bc_east, v_bc_north, v_bc_south, v_bc_west, v_bc_east, v_bc_high, v_bc_low`: boundary values of $\bar{v}$ at east, north, south, west and high/low boundary. Default: 0

`v_bc_east_type, v_bc_north_type, v_bc_south_type, v_bc_west_type`: see below

`v_bc_high_type, v_bc_low_type`: type of b.c. for $\bar{v}$ ('d'=Dirichlet, 'n'=Neumann' or '2'=$\partial^2 v/\partial n^2 = 0$). Default: Dirichlet

`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 11.9

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

`vtk_file_name`: file name of VTK output files

`vtk_movie`: if TRUE, save results every `itstep_save` time step in VTK format

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

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

`w3d_mean`: time-averaged $\bar{w}$ velocity, $\langle\bar{w}\rangle$

`w_bc_east, w_bc_north, w_bc_south, w_bc_west, w_bc_low, w_bc_high`: boundary values of $\bar{w}$ at east, north, south, west, and high/low boundary. Default: 0

`w_bc_east_type, w_bc_north_type, w_bc_south_type, w_bc_west_type`: see below

`w_bc_high_type, w_bc_low_type`: type of b.c. for $\bar{w}$ ('d'=Dirichlet, 'n'=Neumann' or '2'=$\partial^2 w/\partial n^2 = 0$). Default: Dirichlet

`wale`: the WALE model is used

`wsynt_inlet`: synthetic inlet fluctuation in the z direction, $(\mathcal{V}_3')_m$, see 11.9

`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 of the center a control volume, see Fig. 1.3

`z`: the z coordinate of the face of a control volume, see Fig. 1.4

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

`zp`: the z coordinate of the center of a control volume, see Fig. 1.4

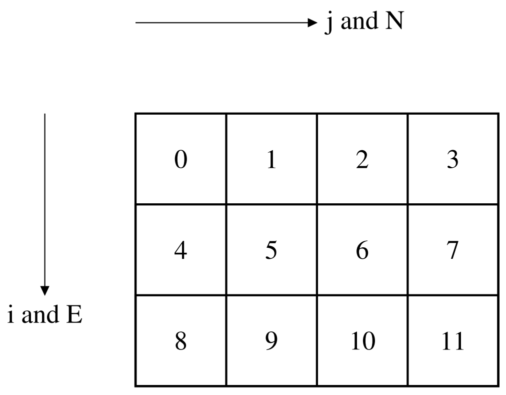
B Sparse matrix format in Python

pyCALC-LES uses the sparse solvers available in Python. The coefficients $a_W, a_E, a_S, a_N, a_L, a_H, a_P, S_u$ must be converted to Python's sparse matrix format. Hence, there are seven 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 solvers `linalg.lgmres`, `linalg.gmres`, `linalg.cgnr`, `linalg.fgmres`, `linalg.bicgstab` or the algebraic multigrid solver pyAMG [1] may be used for all variables. For the pressure, pyAMG is always used.

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

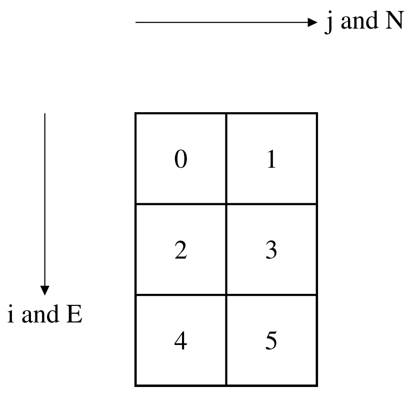
B.1 2D grid, $ni \times nj = (3, 4)$



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

Figure B.1: Matrix for 2D flow. $ni \times nj = (3, 4)$. Cyclic in x. The coefficients due to cyclic boundary conditions are colored in blue.

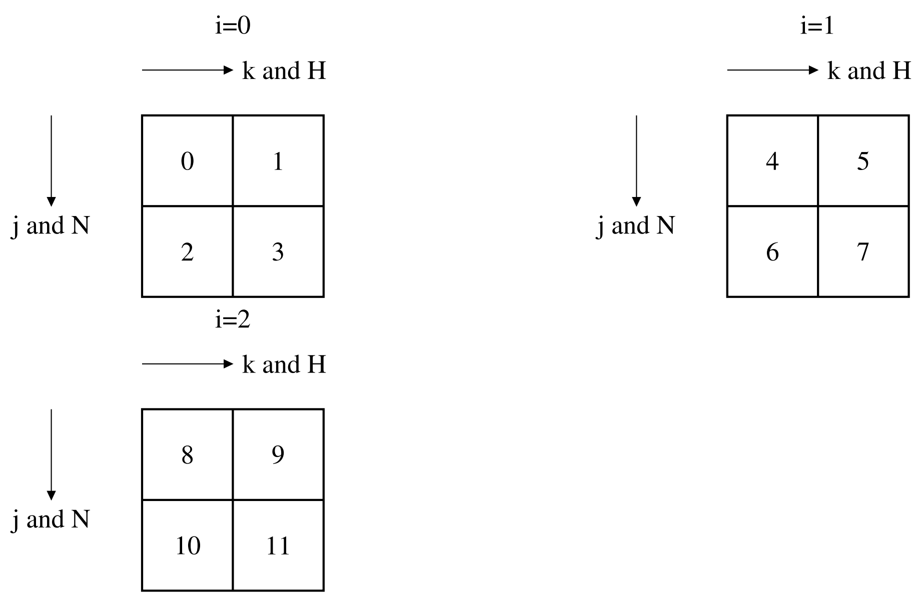
B.2 2D grid, $ni \times nj = (3, 2)$



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

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

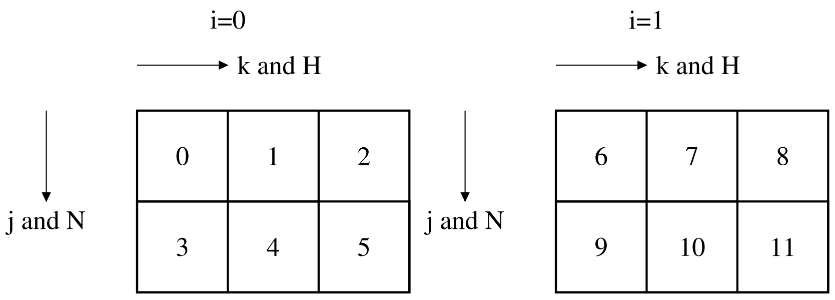
B.3 3D grid, $ni \times nj \times nk = (3, 2, 2)$, cyclic in x,i



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

Figure B.3: Matrix, A, 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.

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



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

Figure B.4: Matrix, A, 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.

C Using pyAMGx on GPU

If you don't do the installation described below, you must de-activate pyAMGx by commenting the line which imports pyAMGX at the top of `pyCALC-LES.py`, i.e.

```
#import pyamgx
```

pyAMGx is a Python interface to the NVIDIA AMGX library. pyAMGx can be used to construct complex solvers and preconditioners to solve sparse sparse linear systems on the GPU. pyAMGx has been tested only on Linux, though it should be possible to install on Windows as well.
Your computer must have a (compatible) nVidea graphics card. You can check which graphics card you have with the Linux command

```
lspci
```

Look for the line starting with `lUSB controller:`.
Start by getting the nVidia CUDA toolkit. In Ubuntu, type

```
sudo apt install nvidia-cuda-toolkit
```

You may also have to install drivers with the command

```
sudo ubuntu-drivers autoinstall
```

After installation you can check the installation

```
nvcc -version
```

You need to install the AMGX library. Instructions are found at here.
When installing AMGX, I encountered a couple of problems:

- I had to install `gcc-9` and `g++-9` as
  1. `sudo apt install gcc-9`
  2. `sudo apt install g++-9`
- I don't have MPI. Hence, I must use the command `cmake -D CMAKE_NO_MPI="TRUE" ../`
- When running `cmake ../` I had to change the file `../examples/CMakeLists.txt`. I replaced the line

  ```
  include_directories("${CMAKE_CURRENT_SOURCE_DIR}/../include" \\
  "${CMAKE_CUDA_TOOLKIT_INCLUDE_DIRECTORIES}")
  ```

  by

  ```
  include_directories("../include" "/usr/lib/cuda/")
  ```

On Ubuntu, I had to download and install Nvidia drivers. I did that by using Ubuntu's **Software updater**. Then I chose Settings/Additional Drivers/ and ticked 'Using NVIDIA driver metapackage ...'.
Now download `pyAMGx` and install it. You find instructions here.
On my Ubuntu 23.4, I had to set

```
export C_INCLUDE_PATH=$AMGX_DIR/include:$C_INCLUDE_PATH
export LD_LIBRARY_PATH=$AMGX_DIR/build:$LD_LIBRARY_PATH
export CPLUS_INCLUDE_PATH=$C_INCLUDE_PATH
export LIBRARY_PATH=$LD_LIBRARY_PATH
```

before installing pyamgx
To select the `pyamgx` solver in **pyCALC-LES**, set

```
solver_p='pyamgx_p'
solver_vel='pyamgx'
solver_turb'pyamgx'
```

in `setup_case`.

References

[1] 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.

[2] J. Hansson. Implementing GPU acceleration into the pyCALC-LES code using CuPy. Phd course report, Division of Fluid Dynamics, Department of Mechanics and Maritime Sciences, Chalmers University of Technology, Göteborg, Sweden, 2023.

[3] B. P. Leonard. A stable and accurate convective modelling based on quadratic upstream interpolation. *Computational Methods in Applied Mechanical Engineering*, 19:59–98, 1979.

[4] 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.

[5] A. Srinath. pyamgx – GPU accelerated multigrid library for Python, 2018.

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

[7] 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.

[8] J. Ma, S.-H. Peng, L. Davidson, and F. Wang. A low Reynolds number partially-averaged Navier-Stokes model for turbulence. In *8th International ERCOFTAC Symposium on Engineering Turbulence, Modelling and Measurements*, Marseille, France, 9-11 June, 2010.

[9] J. Ma, S.-H. Peng, L. Davidson, and F. Wang. A low Reynolds number variant of Partially-Averaged Navier-Stokes model for turbulence. *International Journal of Heat and Fluid Flow*, 32(3):652–669, 2011. doi: 10.1016/j.ijheatfluidflow.2011.02.001. URL `http://dx.doi.org/10.1016/j.ijheatfluidflow.2011.02.001`. 10.1016/j.ijheatfluidflow.2011.02.001.

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

[11] 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`.

[12] R. Schiestel and A. Dejoan. Towards a new partially integrated transport model for coarse grid and unsteady turbulent flow simulations. *Theoretical and Computational Fluid Dynamics*, 18(6):443–468, 2005. URL `https://doi.org/10.1007/s00162-s̄004-s̄0155-s̄z`.

[13] B. Chaouat and R. Schiestel. A new partially integrated transport model for subgrid-scale stresses and dissipation rate for turbulent developing flows. *Physics of Fluids*, 17(065106), 2005.

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

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

[16] 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.

[17] 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.

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

[19] 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.

[20] 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.

[21] 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.

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

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

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

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

[26] 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.

[27] 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.

[28] 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.

[29] 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.

[30] L. Davidson. Two-equation hybrid RANS-LES models: A novel way to treat k and ω at inlets and at embedded interfaces. *Journal of Turbulence*, 18(4):291–315, 2017. doi: 10.1080/14685248.2017.1281417. URL `http://dx.doi.org/10.1080/14685248.2017.1281417`.

[31] S. Arvidson, L. Davidson, and S.-H. Peng. Interface methods for grey-area mitigation in turbulence-resolving hybrid RANS-LES. *International Journal of Heat and Fluid Flow*, 73:236–257, 2018.

[32] L. Davidson and C. Friess. Detached eddy simulations: Analysis of a limit on the dissipation term for reducing spectral energy transfer at cut-off. In *13th International ERCOFTAC Symposium on Engineering Turbulence Modelling and Measurements (ETMM13), Rhodes/Digital, Greece 15-17 September*, 2021.

[33] L. Davidson and C. Friess. Detached eddy simulations: Analysis of a limit on the dissipation term for reducing spectral energy transfer at cut-off (in review). *International Journal of Heat and Fluid Flow*, 2022.

[34] M. Shur, P.R. Spalart, M.K. Strelets, and A.K. Travin. Synthetic turbulence generators for RANS-LES interfaces in zonal simulations of aerodynamic and aeroacoustic problems. *Flow, Turbulence and Combustion*, 93:69–92, 2014.

[35] M. Carlsson, L. Davidson, S.-H. Peng, and S. Arvidson. Investigation of turbulence injection methods in large eddy simulation using a compressible flow solver. In *AIAA Science and Technology Forum and Exposition, AIAA SciTech Forum*, 2022.

[36] L. Davidson. An introduction to turbulence models. Technical Report 97/2, Dept. of Thermo and Fluid Dynamics, Chalmers University of Technology, Gothenburg, 1997.

[37] Bastian Nebenführ and Lars Davidson. Large-eddy simulation study of thermally stratified canopy flow. *Boundary-Layer Meteorology*, pages 1–24, 2015. ISSN 0006-8314. doi: 10.1007/s10546-s̄015-s̄0025-s̄9. URL `http://dx.doi.org/10.1007/s10546-s̄015-s̄0025-s̄9`.

[38] L. Davidson. Using Machine Learning for formulating new wall functions for Large Eddy Simulation: A second attempt. Technical report, Division of Fluid Dynamics, Dept. of Mechanics and Maritime Sciences, Chalmers University of Technology, Gothenburg, 2022.