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

February 25, 2022

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 \( k \), \( \varepsilon \) and \( \omega \). 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 \( \Delta z \) may vary.

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

pyCALC-LES is a finite volume code. It is fully vectorized (i.e. no for loops). The solution procedure is based on fractional step. Second-order central differencing is used in space and the Crank-Nicolson scheme in time. The discretized equations are solved with Python 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 pressure equation can be solved on the GPU (a speed-up of a factor of ten; overall speed-up of two), see Section C. Also the momentum equations may be solved on the GPU.
## Contents

1 Geometrical details of the grid .................................................. 7
   1.1 Grid ................................................................................. 7
       1.1.1 Nomenclature for the grid ........................................... 7
       1.1.2 Area calculation of control volume faces ....................... 7
       1.1.3 Interpolation ............................................................... 10
   1.2 Gradient ............................................................................. 10

2 Diffusion .................................................................................... 11
   2.1 Unsteady diffusion ............................................................. 12
       2.1.1 Crank-Nicolson ........................................................... 13
   2.2 Convergence criteria .......................................................... 13
   2.3 2D Diffusion ....................................................................... 14
   2.4 3D diffusion ....................................................................... 15

3 Convection – diffusion ................................................................. 16
   3.1 Central Differencing scheme (CDS) ....................................... 16
   3.2 First-order upwind scheme .................................................. 17
   3.3 Hybrid scheme .................................................................... 18
   3.4 QUICK ............................................................................... 18
   3.5 Inlet boundary conditions using source term ......................... 18
   3.6 Wall boundary conditions using source term ......................... 19

4 The Fractional-step method .......................................................... 19
   4.1 Boundary condition for \( \bar{p} \) ............................................... 20

5 Boundary Conditions ................................................................... 21
   5.1 Outlet velocity, small outlet .................................................. 21
   5.2 Outlet velocity, large outlet ................................................... 22
   5.3 Remaining variables ........................................................... 22

6 The Smagorinsky Model ............................................................... 22

7 The WALE model ....................................................................... 23

8 The PANS Model ....................................................................... 23

9 The PITM Model ....................................................................... 24

10 The \( k - \omega \) DES model ............................................................. 24

11 Inlet boundary conditions ............................................................. 25
   11.1 Synthesized turbulence ...................................................... 25
   11.2 Random angles ................................................................. 26
   11.3 Highest wave number ....................................................... 26
   11.4 Smallest wave number ...................................................... 26
   11.5 Divide the wave number range ......................................... 26
   11.6 von Kármán spectrum ........................................................ 26
   11.7 Computing the fluctuations ............................................... 27
   11.8 Introducing time correlation .............................................. 27
12 Procedure to generate anisotropic synthetic fluctuations

13 Flow Chart

14 Modules

14.1 bc_outlet_bc ........................................... 30
14.2 calceps .................................................. 30
14.3 calc_kom ............................................... 30
14.4 calc ...................................................... 30
14.5 calcom .................................................. 30
14.6 calep ..................................................... 30
14.7 calcu ..................................................... 30
14.8 calcv .................................................... 31
14.9 calew .................................................... 31
14.10 coeff ................................................... 31
14.11 compute_facephi ...................................... 31
14.12 compute_fk ............................................. 31
14.13 compute_inlet_fluct .................................. 31
14.14 conv .................................................... 31
14.15 correct_conv .......................................... 31
14.16 fix_omega .............................................. 31
14.17 crank_nicol ............................................ 32
14.18 dphi_dxdy_dzd ........................................ 32
14.19 init ...................................................... 32
14.20 modify_eps, modify_k, modify_d, modify_u, modify_v, modify_w .................................. 32
14.21 modify_case.py ........................................ 32
14.22 modify_init ............................................ 32
14.23 print_indata ........................................... 32
14.24 read_restart_data ..................................... 32
14.25 save_data .............................................. 32
14.26 save.file .............................................. 33
14.27 save_timeaver_data ................................... 33
14.28 save_vtk ................................................ 33
14.29 setup_case.py ......................................... 33
14.30 solve_3d ............................................... 33
14.31 solve_p .................................................. 33
14.32 solve_tdma ............................................. 34
14.33 synt_fluct .............................................. 34
14.34 time_stats .............................................. 34
14.35 update .................................................. 34
14.36 vist_kom ............................................... 34
14.37 vist_pans .............................................. 34
14.38 vist_smag .............................................. 34
14.39 vist_wale .............................................. 34

15 DNS of fully-developed channel flow at \( Re_\tau = 500 \)

15.1 setup_case.py ......................................... 36
15.1.1 Section 1 ............................................ 36
15.1.2 Section 3 ............................................ 36
15.1.3 Section 4 ............................................ 36
<table>
<thead>
<tr>
<th>Section</th>
<th>Content</th>
</tr>
</thead>
<tbody>
<tr>
<td>15.1.4</td>
<td>Section 6</td>
</tr>
<tr>
<td>15.1.5</td>
<td>Section 7</td>
</tr>
<tr>
<td>15.1.6</td>
<td>Section 8</td>
</tr>
<tr>
<td>15.1.7</td>
<td>Section 9</td>
</tr>
<tr>
<td>15.1.8</td>
<td>Section 10</td>
</tr>
<tr>
<td>15.2</td>
<td>modify case.py</td>
</tr>
<tr>
<td>15.2.1</td>
<td>modify_u</td>
</tr>
<tr>
<td>15.3</td>
<td>Run the code</td>
</tr>
<tr>
<td>16</td>
<td>Fully-developed channel flow at $Re_\tau = 5200$ using $k - \omega$ DES</td>
</tr>
<tr>
<td>16.1</td>
<td>setup case.py</td>
</tr>
<tr>
<td>16.1.1</td>
<td>Section 1</td>
</tr>
<tr>
<td>16.1.2</td>
<td>Section 2</td>
</tr>
<tr>
<td>16.1.3</td>
<td>Section 5</td>
</tr>
<tr>
<td>16.1.4</td>
<td>Section 6</td>
</tr>
<tr>
<td>16.1.5</td>
<td>Section 10</td>
</tr>
<tr>
<td>16.2</td>
<td>modify case.py</td>
</tr>
<tr>
<td>17</td>
<td>RANS of channel flow at $Re_\tau = 5200$ using $k - \omega$</td>
</tr>
<tr>
<td>17.1</td>
<td>setup case.py</td>
</tr>
<tr>
<td>17.1.1</td>
<td>Section 1</td>
</tr>
<tr>
<td>17.1.2</td>
<td>Section 2</td>
</tr>
<tr>
<td>17.1.3</td>
<td>Section 3</td>
</tr>
<tr>
<td>17.1.4</td>
<td>Section 8</td>
</tr>
<tr>
<td>17.1.5</td>
<td>Section 10</td>
</tr>
<tr>
<td>17.2</td>
<td>modify case.py</td>
</tr>
<tr>
<td>18</td>
<td>Periodic flow over a 2D hill using PANS</td>
</tr>
<tr>
<td>18.1</td>
<td>setup_case.py</td>
</tr>
<tr>
<td>18.1.1</td>
<td>Section 1</td>
</tr>
<tr>
<td>18.1.2</td>
<td>Section 2</td>
</tr>
<tr>
<td>18.1.3</td>
<td>Section 4</td>
</tr>
<tr>
<td>18.1.4</td>
<td>Section 6</td>
</tr>
<tr>
<td>18.1.5</td>
<td>Section 8</td>
</tr>
<tr>
<td>18.2</td>
<td>modify_case.py</td>
</tr>
<tr>
<td>18.2.1</td>
<td>modify_u</td>
</tr>
<tr>
<td>18.2.2</td>
<td>fix_\epsilon</td>
</tr>
<tr>
<td>19</td>
<td>Synthetic turbulence at inlet: Channel flow at $Re_\tau = 395$</td>
</tr>
<tr>
<td>19.1</td>
<td>setup_case.py</td>
</tr>
<tr>
<td>19.1.1</td>
<td>Section 2</td>
</tr>
<tr>
<td>19.1.2</td>
<td>Section 3</td>
</tr>
<tr>
<td>19.1.3</td>
<td>Section 4</td>
</tr>
<tr>
<td>19.1.4</td>
<td>Section 6</td>
</tr>
<tr>
<td>19.1.5</td>
<td>Section 10</td>
</tr>
<tr>
<td>19.2</td>
<td>modify_case.py</td>
</tr>
<tr>
<td>19.2.1</td>
<td>modify_init</td>
</tr>
<tr>
<td>19.2.2</td>
<td>modify_inlet</td>
</tr>
<tr>
<td>19.2.3</td>
<td>modify_u</td>
</tr>
<tr>
<td>19.2.4</td>
<td>modify_v</td>
</tr>
<tr>
<td>Section</td>
<td>Description</td>
</tr>
<tr>
<td>---</td>
<td>---</td>
</tr>
<tr>
<td>19.2.5</td>
<td>modify_w</td>
</tr>
<tr>
<td>19.2.6</td>
<td>modify_outlet</td>
</tr>
</tbody>
</table>

20 Synthetic turbulence at inlet using commutation terms: Channel flow

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>20.1</td>
<td>setup_case.py</td>
</tr>
<tr>
<td>20.1.1</td>
<td>Section 2</td>
</tr>
<tr>
<td>20.1.2</td>
<td>Section 4</td>
</tr>
<tr>
<td>20.1.3</td>
<td>Section 6</td>
</tr>
<tr>
<td>20.1.4</td>
<td>Section 10</td>
</tr>
<tr>
<td>20.2</td>
<td>modify_case.py</td>
</tr>
<tr>
<td>20.2.1</td>
<td>modify_init</td>
</tr>
<tr>
<td>20.2.2</td>
<td>modify_inlet</td>
</tr>
<tr>
<td>20.2.3</td>
<td>modify_k</td>
</tr>
<tr>
<td>20.2.4</td>
<td>modify_om</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>21.1</td>
<td>setup_case.py</td>
</tr>
<tr>
<td>21.1.1</td>
<td>Section 1</td>
</tr>
<tr>
<td>21.1.2</td>
<td>Section 2</td>
</tr>
<tr>
<td>21.1.3</td>
<td>Section 4</td>
</tr>
<tr>
<td>21.1.4</td>
<td>Section 6</td>
</tr>
<tr>
<td>21.1.5</td>
<td>Section 8</td>
</tr>
<tr>
<td>21.1.6</td>
<td>Section 10</td>
</tr>
<tr>
<td>21.2</td>
<td>modify_case.py</td>
</tr>
<tr>
<td>21.2.1</td>
<td>modify_init</td>
</tr>
</tbody>
</table>

22 DES of boundary layer flow with $k-\omega$ model and commutation terms

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>22.1</td>
<td>setup_case.py</td>
</tr>
<tr>
<td>22.1.1</td>
<td>Section 1</td>
</tr>
<tr>
<td>22.1.2</td>
<td>Section 2</td>
</tr>
<tr>
<td>22.1.3</td>
<td>Section 6</td>
</tr>
<tr>
<td>22.1.4</td>
<td>Section 8</td>
</tr>
<tr>
<td>22.1.5</td>
<td>Section 10</td>
</tr>
<tr>
<td>22.2</td>
<td>modify_case.py</td>
</tr>
<tr>
<td>22.2.1</td>
<td>modify_init</td>
</tr>
<tr>
<td>22.2.2</td>
<td>modify_inlet</td>
</tr>
</tbody>
</table>

23 RANS of hump flow using the $k-\omega$ model

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>23.1</td>
<td>setup_case.py</td>
</tr>
<tr>
<td>23.1.1</td>
<td>Section 6</td>
</tr>
</tbody>
</table>

24 DES of hump flow using the $k-\omega$ model

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>24.1</td>
<td>setup_case.py</td>
</tr>
<tr>
<td>24.1.1</td>
<td>Section 6</td>
</tr>
</tbody>
</table>

25 Workshop

<table>
<thead>
<tr>
<th>Section</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>25.1</td>
<td>Channel flow, RANS</td>
</tr>
<tr>
<td>25.1.1</td>
<td>New grid</td>
</tr>
<tr>
<td>25.1.2</td>
<td>Boundary wall conditions on $\omega$</td>
</tr>
<tr>
<td>25.1.3</td>
<td>$k-\varepsilon$ model</td>
</tr>
</tbody>
</table>
1. Geometrical details of the grid

1.1 Grid

The grid \((x_{2d}, y_{2d})\) 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 \(x_{p2d}, y_{p2d}\) are placed at the center of the control volume. In any coordinate direction, lets say \(\xi\), there are \(n_i + 1\) control volume faces, and \(n_i\) control volumes. Note that \((\xi, \eta, z)\) 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

\[
\begin{align*}
x_{2d}(i, j), y_{2d}(i, j) \\
x_{2d}(i+1, j), y_{2d}(i+1, j) \\
x_{2d}(i, j+1), y_{2d}(i, j+1) \\
x_{2d}(i+1, j+1), y_{2d}(i+1, j+1)
\end{align*}
\]

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 \(z_p(k)\), 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.
1.1. Grid

\( \vec{a} \): 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

\[
\begin{align*}
a_x &= x2d(i, j+1) - x2d(i, j) \\
a_y &= y2d(i, j+1) - y2d(i, j) \\
b_x &= x2d(i+1, j) - x2d(i, j) \\
b_y &= y2d(i+1, j) - y2d(i, j)
\end{align*}
\]

(1.1)

(1.2)

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

\[
\begin{align*}
A_{wx} &= -a_y \Delta z \\
A_{wy} &= a_x \Delta z \\
A_{sx} &= b_y \Delta z \\
A_{sy} &= -b_x \Delta z
\end{align*}
\]

(1.3)

(1.4)

(1.5)

(1.6)

which are stored in Python arrays \texttt{areawx, areawy, areasx} and \texttt{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, \( A_1, A_2 \), is calculated as the cross product.

\[
A_1 = \frac{1}{2} |\vec{a} \times \vec{b}|; \quad A_2 = \frac{1}{2} |\vec{b} \times \vec{c}|
\]

(1.8)

The area for the low face is then obtained as

\[
A_z = A_1 + A_2
\]

(1.9)

which is stored in the Python array \texttt{areaz}.

The volume of the control volume is computed as \( A_z \Delta z \) which is stored in the Python array \texttt{vol}.

![Control volume in x - y plane. Calculation of areas and volume of cell \( i, j,k \).](image-url)
1.2. Gradient

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 \( \phi \) at the west face is

\[
\phi_w = f_x \phi_P + (1 - f_x) \phi_W
\]

where

\[
f_x = \frac{|\overrightarrow{PW}|}{|\overrightarrow{PW}| + |\overrightarrow{Ww}|}
\]

where \( |\overrightarrow{PW}| \) is the distance from \( P \) (the node) to \( w \) (the west face). In \texttt{pyCALC-LES} the interpolation factors \( (f_x, f_y) \) are stored in the Python array \( f_x \) and \( f_y \). The interpolation factor in the \( z \) direction is 0.5 since \( \Delta z \) is constant.

All geometrical quantities are computed in the module \texttt{init}.

1.2 Gradient

The derivatives of \( \phi \) \( (\partial \phi / \partial x) \) at the cell center are in \texttt{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

\[
\frac{\partial \Phi}{\partial x} = \frac{1}{V} (\Phi_e A_{ex} - \Phi_w A_{wx} + \Phi_s A_{sx} + \Phi_n A_{nx} + \Phi_h A_{hx} - \Phi_l A_{lx})
\]

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

The values at the west, south and low faces of a variable are stored in the Python arrays \( \text{uface}_w, \text{uface}_s, \text{uface}_l, \text{vface}_w, \text{vface}_s, \text{vface}_l, \text{wface}_w, \text{wface}_s, \text{wface}_l \), etc. They are computed in the Python module \texttt{compute_face_phi}.

The derivative \( \partial \Phi / \partial x, \partial \Phi / \partial y \) and \( \partial \Phi / \partial z \), are computed in the Python modules \texttt{dphidx}, \texttt{dphidy} and \texttt{dphidz}, respectively.

![Figure 1.4: Control volume in y – z plane.](image-url)
2. Diffusion

We start by looking at 1D diffusion for a generic variable, $\phi$, with diffusion coefficient $\Gamma$

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

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.

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

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 $\phi$ and the diffusion coefficient, $\Gamma$, 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.

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

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

$$\Gamma_w = f_s \Gamma_P + (1 - f_s) \Gamma_W, \quad (2.3)$$

Insertion of Eq. 2.2 into Eq. 2.1 gives

$$a_P \phi_P = a_E \phi_E + a_W \phi_W + \bar{S} \Delta x \quad (2.4)$$
2.1 Unsteady diffusion

We discretize the unsteady diffusion equation

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

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

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

Left-hand side:

$$\int_w^{e} \left[ \phi^t_{t+\Delta t} - \phi^o_t \right] dx = (\phi^P_t - \phi^o_P) \Delta x$$

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 \(\phi_W\), \(\phi_P\) and \(\phi_E\) be taken?
2.2. Convergence criteria

1. Fully implicit: take them at the new time step \( t + \Delta t \), i.e. \( \phi_{W}, \phi_{P}^{1} \) and \( \phi_{E}^{1} \) (first-order accurate).

2. Fully explicit: take them at the old time step \( t \), i.e. \( \phi_{W}^{0}, \phi_{P}^{0} \) and \( \phi_{E}^{0} \) (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, \( \alpha \), is equal to 0.5. Below we express the time integration in a general way using \( \alpha \). When \( \alpha = 0 \), it corresponds to fully explicit and when \( \alpha = 1 \), it corresponds to fully implicit. We get

\[
a_{P}\phi_{P} = \alpha a_{E}\phi_{E} + \alpha a_{W}\phi_{W} + (1 - \alpha)(a_{E}\phi_{E}^{0} + a_{W}\phi_{W}^{0}) + (a_{P}^{0} - (1 - \alpha)(a_{E} + a_{W}))\phi_{P}^{0} + (1 - \alpha)(a_{E} + a_{W})\phi_{P}^{o}
\]

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

2.2 Convergence criteria

Compute the residual for Eq. 2.4

\[
R = \sum_{\text{all cells}} |a_{E}\phi_{E} + a_{W}\phi_{W} + S_{U} - a_{P}\phi_{P}|
\]

In Python it corresponds to \( |Ax - b| \). Since we want Eq. 2.4 to be satisfied, the difference of the right-hand side and the left-hand side is a good measure of how well the equation is satisfied. The residual \( R \) is computed using the Python command `np.linalg.norm`. Note that \( R \) has the units of the integrated differential equation. For example, for the temperature \( R \) has the same dimension as heat transfer rate divided by density, \( \rho \), and specific heat, \( c_{p} \), i.e. temperature times volume per second \( [K m^{3}/s] \). 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 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 \( \phi \).

Regardless if we solve the continuity equation, the Navier-Stokes equation or the temperature equation, the procedure is the same: \( F \) should represent the total flow of the dependent variable.
Continuity equation. $F$ is here the total incoming volume flow $\dot{V}$.

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

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

2.3 2D Diffusion

The two-dimensional diffusion equation for a generic variable $\phi$ reads

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

(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.

$$
\int_{w}^{e} \int_{s}^{n} \left[ \frac{\partial}{\partial x} \left( \Gamma \frac{\partial \phi}{\partial x} \right) \right] dxdy = \int_{s}^{n} \left[ \left( \Gamma \frac{\partial \phi}{\partial x} \right)_e - \left( \Gamma \frac{\partial \phi}{\partial x} \right)_w \right] dy
$$

$$
= \int_{s}^{n} \left( \Gamma \frac{\phi_E - \phi_P}{\delta x_e} - \Gamma \frac{\phi_P - \phi_W}{\delta x_w} \right) dy
$$
Now integrate in the $y$ direction. We do this by estimating the integral
\[
\int_y^n f(y)dy = f_P \Delta y + O((\Delta y)^2)
\]
(i.e. $f$ is taken at the mid-point $P$) which is second order accurate, since it is exact if $f$ is a linear function. For our equation we get
\[
\int_y^n (\Gamma_e \phi_E - \phi_P - \phi_W) \delta x_e - (\Gamma_w \phi_W - \phi_P) \delta x_w \Delta y
\]
Doing the same for the diffusion term in the $y$ direction in Eq. 2.6 gives
\[
\left(\Gamma_e \frac{\phi_E - \phi_P}{\delta x_e} - \Gamma_w \frac{\phi_P - \phi_W}{\delta x_w}\right) \Delta y + \left(\Gamma_n \frac{\phi_N - \phi_P}{\delta y_n} - \Gamma_s \frac{\phi_P - \phi_S}{\delta y_s}\right) \Delta x + S \Delta x \Delta y = 0
\]
Rewriting it as an algebraic equation for $\phi_P$, we get
\[
a_P \phi_P = a_E \phi_E + a_W \phi_W + a_N \phi_N + a_S \phi_S + S
\]
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.
\[
a_P \phi_P = a_E \phi_E + a_W \phi_W + a_N \phi_N + a_S \phi_S + a_H \phi_H + a_L \phi_L + S
\]
\[
a_E = \frac{\Gamma_e \Delta y}{\delta x_e}, \ a_W = \frac{\Gamma_w \Delta y}{\delta x_w}, \ a_N = \frac{\Gamma_n \Delta x}{\delta y_n}, \ a_S = \frac{\Gamma_s \Delta x}{\delta y_s}
\]
\[
S_U = \bar{S} \Delta x \Delta y, \ a_P = a_E + a_W + a_N + a_S - \bar{S}.
\]

The east, north and high coefficients are computed from $a_W$, $a_S$ and $a_L$, respectively, as
\[
a_{E,i} = a_{W,i+1}
\]
\[
a_{N,j} = a_{S,j+1}
\]
\[
a_{H,k} = a_{L,k+1}
\]
3 Convection – diffusion

The 1D convection-diffusion equation reads

\[
\frac{d}{dx}(\bar{u}\phi) = \frac{d}{dx}(\Gamma \frac{d\phi}{dx}) + S
\]

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

\[
\int_{w}^{e} \frac{d}{dx}(\bar{u}\phi) \, dx = \int_{w}^{e} \left[ \frac{d}{dx}(\Gamma \frac{d\phi}{dx}) + S \right] \, dx.
\] (3.1)

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

\[
\int_{w}^{e} \frac{d}{dx}(\bar{u}\phi) \, dx = (\bar{u}\phi)_{e} - (\bar{u}\phi)_{w}.
\]

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

3.1 Central Differencing scheme (CDS)

How to estimate \( \phi_{e} \) and \( \phi_{w} \)? The most natural way is to use linear interpolation (central differencing); for the east face this gives

\[
(\bar{u}\phi)_{w} = (\bar{u})_{w} \phi_{w}
\]

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

\[
(\bar{u}\phi)_{w} = (\bar{u})_{w} \phi_{w}, \quad \text{where} \quad \phi_{w} = f_{x}\phi_{P} + (1 - f_{x})\phi_{W}
\]

where \( f_{x} \) is the interpolation function (see Eq. 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

\[
\frac{(\bar{u})_{e} \phi_{E} + \phi_{P}}{2} - (\bar{u})_{w} \frac{\phi_{P} + \phi_{W}}{2} = \Gamma_{e} \frac{(\phi_{E} - \phi_{P})}{\delta x_{e}} - \Gamma_{w} \frac{(\phi_{P} - \phi_{W})}{\delta x_{w}} + S \Delta x
\]
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.

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 \(|P_c| > 2\); 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.
3.3 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 QUICK

QUICK [2] is an upwind biased scheme. It fits a quadratic polynomial to three nodes (two upstream and one downstream, see Fig. 3.2)

$$\Phi_w = \begin{cases} 
-\frac{1}{8}\Phi_W + \frac{3}{4}\Phi_P + \frac{3}{8}\Phi_E & \text{if } u_w \geq 0 \\
\frac{3}{8}\Phi_W + \frac{1}{4}\Phi_P - \frac{1}{8}\Phi_E & \text{if } u_w < 0
\end{cases}$$

The scheme gives a third-order discretization error (in 1D). It is stable due to its upwind character, but it is unbounded, i.e. it can occur that $\Phi_e > \max \{\Phi_W, \Phi_P, \Phi_E\}$.

3.5 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

$$a_P\bar{u}_P = a_W\bar{u}_W + a_E\bar{u}_E + S_U$$

$$a_P = a_W + a_E - S_P, \quad a_W = C_w, \quad a_E = -0.5C_e$$

$$C_w = \bar{u}_W A_w$$

$$a_P = C_w - 0.5C_e$$

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

$$a_w = 0$$

$$S_{U,add}^u = C_w\bar{u}_{in}$$

$$S_{P,add}^u = -C_w$$

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

![1D grid. Boundary conditions at x = 0.](image-url)
3.6. Wall boundary conditions using source term

For \( \bar{v} \) and \( w \) it reads

\[
\begin{align*}
S_{V,\text{add}}^v &= C_w \bar{v}_{\text{in}} \\
S_{P,\text{add}}^v &= -C_w \\
S_{U,\text{add}}^w &= C_w \bar{w}_{\text{in}} \\
S_{P,\text{add}}^w &= -C_w
\end{align*}
\]

The additional term for the diffusion reads

\[
\begin{align*}
S_{U,\text{add,} \text{diff}}^w &= \frac{\nu_{\text{tot}} A_w}{\Delta x} \bar{u}_{\text{in}} \\
S_{U,\text{add,} \text{diff}}^v &= \frac{\nu_{\text{tot}} A_w}{\Delta x} \bar{v}_{\text{in}} \\
S_{U,\text{add,} \text{diff}}^w &= \frac{\nu_{\text{tot}} A_w}{\Delta x} \bar{w}_{\text{in}} \\
S_{P,\text{add,} \text{diff}} &= -\frac{\nu_{\text{tot}} A_w}{\Delta x}
\end{align*}
\]

where \( S_{P,\text{add,} \text{diff}} \) is the same for \( \bar{u}, \bar{v} \) and \( \bar{w} \). The viscous part of Eq. 3.5 is implemented in module bc. The turbulent part and the convective part (Eqs. 3.3 and 3.4) are implemented in module \( u \), module \( v \) etc.

3.6 Wall boundary conditions using source term

We use exactly the same procedure as in Section 3.5. At walls, there is no convection and the velocity is zero. Hence we simply use Eq. 3.5 with \( \bar{u} = \bar{v} = \bar{w} = 0 \), i.e. (for west wall)

\[
S_{P,\text{add,} \text{diff}} = -\frac{\nu A_w}{\Delta x}
\]

Note that we use \( \nu \) instead of \( \nu_{\text{tot}} \) since the turbulent viscosity is zero at the wall.

This boundary condition is implemented in module bc.

4 The Fractional-step method

modules: calcp, correct_conv

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

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

The discretized momentum equations read

\[
\begin{align*}
\bar{v}_i^{n+1/2} &= \bar{v}_i^n + \Delta t H \left( \bar{v}_i^n, \bar{v}_i^{n+1/2} \right) \\
-\alpha \Delta t \frac{\partial \bar{p}^{n+1/2}}{\partial x_i} - (1-\alpha) \Delta t \frac{\partial \bar{p}^n}{\partial x_i}
\end{align*}
\]
4.1 Boundary condition for $\bar{p}$

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.

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

(4.3)

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

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

(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 $\Gamma$ is set to one).

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

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

(4.5)

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

4.1 Boundary condition for $\bar{p}$

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

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

<table>
<thead>
<tr>
<th></th>
<th>RANS</th>
<th>LES</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>Domain</strong></td>
<td>2D or 3D</td>
<td>always 3D</td>
</tr>
<tr>
<td><strong>Time domain</strong></td>
<td>steady or unsteady</td>
<td>always unsteady</td>
</tr>
<tr>
<td><strong>Space discretization</strong></td>
<td>2nd order upwind</td>
<td>central differencing</td>
</tr>
<tr>
<td><strong>Time discretization</strong></td>
<td>1st order</td>
<td>2nd order (e.g. C-N)</td>
</tr>
<tr>
<td><strong>Turbulence model</strong></td>
<td>more than two-equations</td>
<td>zero- or one-equation</td>
</tr>
</tbody>
</table>

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

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

$$
\int_S \frac{\partial \bar{p}}{\partial x_i} n_i dS = \frac{1}{\Delta t \alpha} \int_S \bar{v}^*_f i n_i dS
$$

(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

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

(4.7)

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

5 Boundary Conditions

5.1 Outlet velocity, small outlet

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

$$
\bar{u}_in h_{in} = \bar{u}_out h_{out} \Rightarrow \bar{u}_out = \bar{u}_in h_{in}/h_{out}
$$
5.2 Outlet velocity, large outlet

For large outlets the outlet velocity must be allowed to vary over the outlet. The proper boundary condition in this case is $\partial \bar{u}/\partial x = 0$. Hence it is important that the flow near the outlet is fully developed, so that this boundary condition corresponds to the flow conditions. The best way to ensure this is to locate the outlet boundary sufficiently far downstream. If we have a recirculation region in the domain (see Fig. 5.2), the outlet should be located sufficiently far downstream of this region so that $\partial \bar{u}/\partial x \simeq 0$.

The outlet boundary condition is implemented as follows (see Fig. 5.2)

1. Set $\bar{u}_e = \bar{u}_w$ for all nodes (i.e. for $j = 0$ to $4$, see Fig. 5.2);
2. In order to speed up convergence, enforce global continuity.
   - Inlet volume flow: $\dot{V}_{in} = \sum_{inlet} \bar{u}_{in} \Delta y$
   - Outlet volume flow: $\dot{V}_{out} = \sum_{outlet} \bar{u}_{out} \Delta y$
   - Compute correction velocity: $\bar{u}_{corr} = (\dot{V}_{in} - \dot{V}_{out})/(A_{out})$, where $A_{out} = \sum_{outlet} \Delta y$.
   - Correct $\bar{u}_e$ so that global continuity (i.e. $\dot{V}_{in} = \dot{V}_{out}$) is satisfied: $\bar{u}_{e,new} = \bar{u}_e + \bar{u}_{corr}$

This boundary condition is implemented in module modify_outlet.

5.3 Remaining variables

Set $\partial \Phi/\partial x = 0$, and implement it through $\Phi_{ni} = \Phi_{n-1}$ 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 \[4\]:

\[
\tau_{ij} = \frac{1}{3} \delta_{ij} \tau_{kk} = - \nu_{sgs} \left( \frac{\partial u_i}{\partial x_j} + \frac{\partial u_j}{\partial x_i} \right) = - 2 \nu_{sgs} \delta_{ij}
\]

\[
\nu_{sgs} = \left( C_S \Delta \right)^2 \sqrt{2 S_{ij} S_{ij}} \equiv \left( C_S \Delta \right)^2 |\phi|
\] (6.1)

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

\[
\Delta = \left( \Delta V_{ijk} \right)^{1/3}
\] (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.

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

where \(n\) is the distance to the nearest wall. \(C_S\) is set to 0.1 (in \texttt{pyCALC-LES} it is set by \texttt{cmu}).

7 The WALE model

\texttt{module: vist\_wale}

The WALE model by \[5\] reads

\[
g_{ij} = \frac{\partial \bar{v}_i}{\partial x_j}, \quad g_{ij}^2 = g_{ik} g_{kj}
\]

\[
\delta_{ij}^d = \frac{1}{2} \left( g_{ij}^2 + g_{ji}^2 \right) - \frac{1}{3} \delta_{ij} g_{kk}^d
\]

\[
\nu_{sgs} = \left( C_m \Delta \right)^2 \frac{(\delta_{ij}^d \delta_{ij}^d)^{3/2}}{\left( \delta_{ij} \delta_{ij} \right)^{7/2} + (\delta_{ij}^d \delta_{ij}^d)^{5/2}}
\] (7.1)

with \(C_m = 0.325\) which corresponds to \(C_a = 0.1\).

8 The PANS Model

\texttt{module: calck, calceps, vist\_pans}

The Reynolds number PANS model presented in \[6, 7\] reads

\[
\frac{\partial k}{\partial t} + \frac{\partial (k \bar{u}_j)}{\partial x_j} = \frac{\partial}{\partial x_j} \left[ \left( \nu + \frac{\nu_t}{\sigma_{ku}} \right) \frac{\partial k}{\partial x_j} \right] + (P^k - \varepsilon)
\]

\[
\frac{\partial \varepsilon}{\partial t} + \frac{\partial (\varepsilon \bar{u}_j)}{\partial x_j} = \frac{\partial}{\partial x_j} \left[ \left( \nu + \frac{\nu_t}{\sigma_{\varepsilon u}} \right) \frac{\partial \varepsilon}{\partial x_j} \right] + C_{\varepsilon 1} \frac{P^k}{k} - C_{\varepsilon 2} \frac{\varepsilon^2}{k}
\]

\[
\nu_t = C_p f_p \frac{k^2}{\varepsilon}, \quad C_{\varepsilon 2} = C_{\varepsilon 1} + \frac{f_k}{f_k} (C_{\varepsilon 2} f_2 - C_{\varepsilon 1})
\]

\[
\sigma_{ku} \equiv \sigma_k \frac{f_k}{f_k}, \quad \sigma_{\varepsilon u} \equiv \sigma_{\varepsilon} \frac{f_k}{f_k}
\]

\[
f_2 = \left[ 1 - \exp \left( - \frac{y^*}{\delta} \right) \right]^2 \left[ 1 - 0.3 \exp \left[ - \left( \frac{R_e}{6.5} \right)^2 \right] \right]
\] (8.1)
9. The PITM Model

\[ f_\mu = \left[ 1 - \exp\left(-\frac{y^*}{14}\right) \right]^2 \left\{ 1 + \frac{5}{R_{t/4}} \exp\left(-\frac{R_t}{200}\right)^2 \right\} \]

\[ R_t = \frac{k^2}{\nu \varepsilon} \]

\[ y^* = \frac{U_\varepsilon y}{\nu}, \quad U_\varepsilon = (\varepsilon \nu)^{1/4} \]

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 \[8\], i.e.

\[ C_\varepsilon 1 = 1.5, C_\varepsilon 2 = 1.9, \sigma_k = 1.4, \sigma_\varepsilon = 1.4, C_\mu = 0.09 \] (8.2)

When the turbulent Prandtl numbers, \( \sigma_k \) and \( \sigma_\varepsilon \) (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 \( \sigma_k \) and \( \sigma_\varepsilon \). When \( \sigma_k \) and \( \sigma_\varepsilon \) are positive, the PITM model is used (see Section 9).

The function \( f_\varepsilon \), 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 \[9\]

\[ f_k = \max\left[f_{k,\text{min}}, \min\left(1, 1 - \frac{\psi - 1}{C_\varepsilon 2 - C_\varepsilon 1}\right)\right] \] (8.3)

\[ \psi = \max\left(1, \frac{k^{3/2}/\varepsilon}{C_{\text{DES}} \Delta_{\text{max}}}\right), \quad \Delta_{\text{max}} = \max(\Delta x_1, \Delta x_2, \Delta x_3) \]

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

At the wall-adjacent cells, \( \varepsilon \) is not solved but it is fixed as

\[ \varepsilon_P = \frac{2\nu k}{y^2} \] (8.4)

where subscript \( P \) denotes wall-adjacent cells and \( y \) is the distance between the cell center and the wall.

9 The PITM Model

PITM is an acronym for Partially Integrated Transport Model \[10, 11\]. It is identical to the PANS model, except that \( f_\varepsilon = 1 \) everywhere and \( f_k = 1 \) in the diffusion terms of the \( k \) and \( \varepsilon \) equations. Setting `pans=True` and positive values \( \sigma_k \) and \( \sigma_\varepsilon \) activates PITM.

10 The \( k - \omega \) DES model

modules: calck_kom, calcom, vist_kom

The Wilcox \( k - \omega \) RANS turbulence model \[12\] modified for DES reads

\[ \frac{\partial k}{\partial t} + \frac{\partial \bar{v}_i k}{\partial x_i} = p^k - F_{\text{DES} \mu \omega} + \frac{\partial}{\partial x_j} \left[ (\nu + \frac{\nu_\varepsilon}{\sigma_k}) \frac{\partial k}{\partial x_j} \right] \]

\[ \frac{\partial \omega}{\partial t} + \frac{\partial \bar{v}_i \omega}{\partial x_i} = C_{\omega 1} \frac{\omega^k}{P^k} - C_{\omega 2} \omega^2 + \frac{\partial}{\partial x_j} \left[ (\nu + \frac{\nu_\varepsilon}{\sigma_\omega}) \frac{\partial \omega}{\partial x_j} \right] \] (10.1)
11. Inlet boundary conditions

\[ \nu_t = \frac{k}{\omega} \]

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

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

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

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 \( \varepsilon \)). 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 [13, 14].

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

Another method based partly on synthesized fluctuations is the vortex method [15]. 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 [16, 17] for creating turbulence for generating noise. It was later further developed for inlet boundary conditions [18, 19, 20, 21].

A turbulent fluctuating velocity fluctuating field (whose average is zero) can be expressed using a Fourier series, see [22]. Let us re-write this formula as

\[ a_n \cos(nx) + b_n \sin(nx) = c_n \cos(\alpha_n) \cos(nx) + c_n \sin(\alpha_n) \sin(nx) = c_n \cos(nx - \alpha_n) \tag{11.1} \]

where \( a_n = c_n \cos(\alpha_n), b_n = c_n \sin(\alpha_n) \). The new coefficient, \( c_n \), and the phase angle, \( \alpha_n \), 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

$$v'(x) = 2 \sum_{n=1}^{N} \hat{u}^n \cos(\kappa_n \cdot x + \psi^n) \sigma^n$$  \hspace{1cm} (11.3)

where $\hat{u}^n$, $\psi^n$ and $\sigma^n$ 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 $\phi^n$ and $\theta^n$ determine the direction of the wavenumber vector $\kappa$, see Eq. 11.3 and Eq. 11.1; $\alpha^n$ denotes the direction of the velocity vector, $v'$. For more details, see [22].

**11.3 Highest wave number**

Define the highest wave number based on mesh resolution $\kappa_{\text{max}} = \frac{2\pi}{(2\Delta)}$ (see [22]), where $\Delta$ 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 $\text{dmin}_{\text{syn}}$ (which can be set to a fraction of $dz$).

**11.4 Smallest wave number**

Define the smallest wave number from $\kappa_1 = \frac{\kappa_e}{p}$, 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 $\delta$ denotes the inlet boundary layer thickness. In [19, 20, 21] 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 $\kappa_e$. A value $p = 2$ is suitable.

**11.5 Divide the wave number range**

Divide the wavenumber space, $\kappa_{\text{max}} - \kappa_1$, 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

$$\hat{u}^n = \left( E(\kappa) \Delta \kappa \right)^{1/2}$$ \hspace{1cm} (11.4)

$$E(\kappa) = c_E \frac{\frac{u_{\text{rms}}^2}{\kappa_e}}{1 + \left( \frac{\kappa}{\kappa_e} \right)^4} \exp \left( -2 \left( \frac{\kappa}{\kappa_e} \right)^2 \right)$$

$$\kappa_1 = \left( \kappa_i \kappa_e \right)^{1/2}, \quad \kappa_\eta = \varepsilon^{1/4} \nu^{-3/4}$$

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

$$k = \int_0^{\infty} E(\kappa) d\kappa$$  \hspace{1cm} (11.5)
11.7 Computing the fluctuations

Figure 11.1: The wave-number vector, $\kappa_n^i$, and the velocity unit vector, $\sigma_n^i$, are orthogonal (in physical space) for each wave number $n$.

which gives $[23]$  

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

where  

$$\Gamma(z) = \int_0^\infty e^{-z'} z'^{-1} \, dz'$$  \hspace{1cm} (11.7)  

11.7 Computing the fluctuations

Having $\hat{u}^n$, $\kappa_n^i$, $\sigma_n^i$ and $\psi^n$, allows the expression in Eq. 11.3 to be computed, i.e.

$$v'_1 = 2 \sum_{n=1}^N \hat{u}^n \cos(\beta^n) \sigma_1$$

$$v'_2 = 2 \sum_{n=1}^N \hat{u}^n \cos(\beta^n) \sigma_2$$

$$v'_3 = 2 \sum_{n=1}^N \hat{u}^n \cos(\beta^n) \sigma_3$$

$$\beta^n = k_1^n x_1 + k_2^n x_2 + k_3^n x_3 + \psi^n$$  \hspace{1cm} (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, $V'_1$, $V'_2$, $V'_3$, are
11.8. Introducing time correlation

\[ \log(\kappa) \]

\[ \kappa e \]

\[ \kappa n \]

\[ \kappa n \]

\[ \Delta \kappa n \]

\[ E(\kappa) \propto \kappa^{-5/3} \]

Figure 11.2: Modified von Kármán spectrum

computed based on an asymmetric time filter

\[
\begin{align*}
(V'_1)_m &= a(V'_1)_{m-1} + b(v'_1)_m \\
(V'_2)_m &= a(V'_2)_{m-1} + b(v'_2)_m \\
(V'_3)_m &= a(V'_3)_{m-1} + b(v'_3)_m
\end{align*}
\] (11.9)

where \( m \) denotes the time step number and

\[ a = \exp(-\Delta t/T_{int}) \] (11.10)

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

\[ b = (1-a^2)^{0.5} \] (11.11)

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

\[ \exp(-\hat{t}/T_{int}) \] (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, \( x_3 \))

\[
\begin{align*}
\bar{v}_1(0, x_2, x_3, t) &= V_{1, in}(x_2) + v'_{1, in}(x_2, x_3, t) \\
\bar{v}_2(0, x_2, x_3, t) &= V_{2, in}(x_2) + v'_{2, in}(x_2, x_3, t) \\
\bar{v}_3(0, x_2, x_3, t) &= V_{3, in}(x_2) + v'_{3, in}(x_2, x_3, t)
\end{align*}
\] (11.13)

where \( v'_{1, in} = (V'_1)_m, v'_{2, in} = (V'_2)_m \) and \( v'_{3, in} = (V'_3)_m \) (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
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 [26].

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 \( \delta \) 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.

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, \( \langle v'_1 \rangle^m \), see Eq. 11.9.

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 [24]

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

(11.14)

where \( \kappa = 0.4 \) and \( B = 5.2 \).

The method to prescribed fluctuating inlet boundary conditions have been used for channel flow [21], for diffusor flow [14] as well as for the flow over a bump and an axisymmetric hill [25].

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 [26].

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 \( \delta \) 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 \( \left( \frac{|u'|}{|v'|} \right)_{\text{max}}^{1/2} \), 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

14 Modules

14.1 bc_outlet_bc
Neumann outlet boundary conditions are set.

14.2 calceps
Source terms in the \( \varepsilon \) equation (AKN) are computed, see Section 8. When PANS is used, \( \text{pans} = \text{True}, f_k \) 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, \( C_{\text{DES}} \) is computed (it is stored in \( f_{k3d} \)). 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 \( \omega \) 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 \( a_w, a_E \ldots \) are computed in the same way as in Section 2.4 (with the diffusion coefficient \( \Gamma \) set to one).

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 $\text{modify}_v$.

14.9 calcw
Source terms in the $\bar{w}$ equation are computed. The user can define additional source terms in $\text{modify}_w$.

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

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 $f3kd$ in $\text{modify}_f$.

14.13 compute_inlet_fluct
Compute synthetic fluctuations (see Section 11.1)

14.14 conv
Compute the convection as a vector product $v \cdot A$ at the west, south and low faces (stored in arrays $\text{convw}$, $\text{convs}$ and $\text{convl}$). 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 $\text{convw}$, $\text{convs}$ and $\text{convl}$ (which are defined at the control volume faces) are corrected so as to satisfy continuity, see Eq. 4.5.

14.16 fix_omega
This routine may be used for fix $\omega$ in the wall-adjacent cell center 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 $\omega$ near a south boundary we use

```plaintext
aw3d[:,0,:]=0
ae3d[:,0,:]=0
as3d[:,0,:]=0
an3d[:,0,:]=0
al3d[:,0,:]=0
```
14.17. **crank_nicol**

```python
ah3d[:,0,:]=0
ap3d[:,0,:]=1
su3d[:,0,:]=om_bc_south
```

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 **dphi1x, dphi1y, dphi1z**

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 $\varepsilon, k, \bar{u}, \bar{v}$ 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_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

- $\bar{u}, \bar{v}, \bar{w}, \bar{p}, \bar{k}, \varepsilon$ and $\omega$ fields

are stored in the files
14.26 save.file

- 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 ≥ itstep_start. Time-averaged data of the

- \bar{u}, \bar{v}, \bar{w}, \bar{p}, k, f_k, \varepsilon, \nu_\ell + \nu, \bar{\nu}_u^2, \bar{\nu}_v^2, \bar{\nu} and \bar{\nu}\bar{v}

are stored in the files

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

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_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.lgmres or linalg.gmres.

14.31 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").
14.32 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 these 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`).

14.33 synt_fluct

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

14.34 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 $itstep \geq itstep_start$.

14.35 update

At the end of each time step, all variables are updated, i.e. $u3d_{old} = u3d$, $v3d_{old} = v3d$, etc.

14.36 vist_kom

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

14.37 vist_pans

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

14.38 vist_smag

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

14.39 vist_wale

The turbulent viscosity is computed using the WALE model, see Section 7.
15. DNS of fully-developed channel flow at $Re_T = 500$

To follow the execution of \texttt{pyCALC-LES}, it is recommended to start reading at the line \textit{the execution of the code starts here}. To find where the time stepping starts, look for the line \textit{start of time stepping}. You can also look at the \texttt{pyCALC-LES} flowchart.

The grid is created using the script \texttt{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.

```python
import numpy as np
import sys

ni=96
nj=96
yfac=1.09  # stretching
ymax=2
xmax=3.2
viscos=1/500
dy=0.1
yc=np.zeros(nj+1)
yc[0]=0.
for j in range(1,int(nj/2)+1):
    yc[j]=yc[j-1]+dy
dy=yfac*dy

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

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

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

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

zmax, nk=np.loadtxt('z.dat')
nk=np.int(nk)
dz=zmax/nk
```
and the *z.dat* file reads

\[1.6 \ 96\]

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

### 15.1 *setup_case.py*

This module consists of 10 sections.

#### 15.1.1 Section 1

We choose the central-differencing scheme for convection

```python
scheme='c'
```

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

```python
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.

```python
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.

```python
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 \(\vec{u}, \vec{v}\) and \(\vec{w}\).

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

```python
maxit=5
min_iter=1
sormax=1e-3
amg_relax='default'
solver_vel='lgmres'
sweep_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

\[ |Ax - b|/|b| < \gamma \]  \hspace{1cm} (15.1)

where \( \gamma \) 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

\texttt{solver\_p='pyamgx'}

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

\texttt{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 \((i,j,k) = (10,10,10)\).

\texttt{imon=10}
\texttt{jmon=10}
\texttt{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\textsuperscript{th} time step.


\begin{verbatim}
ntstep=15000
uin=20
dt=0.5*(x2d[1,0]-x2d[0,0])*np.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
\end{verbatim}

15.1.7 Section 9

The residual of the momentum equation and the continuity equation are normalized by
\texttt{resnorm\_vel} and \texttt{resnorm\_p} which are set to
\begin{verbatim}
resnorm_p=uin*zmax*y2d[1,-1]
resnorm_vel=uin**2*zmax*y2d[1,-1]
\end{verbatim}

15.1.8 Section 10

The boundary conditions are set here. We have cyclic boundary conditions in $x$ and $z$
directions and hence
\begin{verbatim}
cyclic_x = True
cyclic_z = True
\end{verbatim}

The south and north boundaries we define as walls (Dirichlet)
\begin{verbatim}
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'
\end{verbatim}

and the value for all variables is set to zero
\begin{verbatim}
u_bc_south=np.zeros((ni,nk))
u_bc_north=np.zeros((ni,nk))
v_bc_south=np.zeros((ni,nk))
v_bc_north=np.zeros((ni,nk))
w_bc_south=np.zeros((ni,nk))
w_bc_north=np.zeros((ni,nk))
\end{verbatim}

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. \texttt{modify\_u, modify\_v, modify\_w, modify\_p, modify\_k, modify\_eps and modify\_om}. It includes
also modules for modifying initial boundary conditions (\texttt{modify\_init}), convections
(\texttt{modify\_conv}), inlet (\texttt{modify\_inlet}) and outlet boundary conditions (\texttt{modify\_outlet})
15.3. Run the code

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

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

This script simply collects all Python 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 $Re_\tau = 5 \times 200$ using $k - \omega$ DES

You find `setup_case.py` and `modify_case.py` in a directory with the name `channel5200-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 $\epsilon$ equations.

```
scheme_turb='u'
```

We use also first-order time discretization for $k$ and $\omega$

```
acrank_conv_kom=1
```

16.1.2 Section 2

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

```
kom_des = True
j10=0
```

The variable $j10$ 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 $\omega$ 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 \( \omega \) are set as \( k = 0 \) and \( \omega \) as below (see Eq. 10.2).

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

k_bc_south_type = 'd'
k_bc_north_type = 'd'

# boundary conditions for omega
xwall_s = 0.5*(x2d[0:-1,0]+x2d[1:,0])
ywall_s = 0.5*(y2d[0:-1,0]+y2d[1:,0])
dist2_s = (yp2d[:,0]-ywall_s)**2+(xp2d[:,0]-xwall_s)**2
om_bc_south = 6*viscos/c_omega_2/dist2_s

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

xwall_n = 0.5*(x2d[0:-1,-1]+x2d[1:,1])
ywall_n = 0.5*(y2d[0:-1,-1]+y2d[1:,1])
dist2_n = (yp2d[:,-1]-ywall_n)**2+(xp2d[:,-1]-xwall_n)**2
om_bc_north = 10*6*viscos/c_omega_2/dist2_n

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

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

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.

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

17 RANS of channel flow at \( Re_\tau = 5200 \) 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 simulate 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.

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

**17.1.2 Section 2**

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

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

**17.1.3 Section 3**

We don’t start from a previous solution.

```python
restart = False
```

**17.1.4 Section 8**

we increase the time step.

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

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

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

**17.1.5 Section 10**

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

```python
cyclic_z=False
```

In the $z$ direction we set Neumann boundary condition for all variables except $\bar{w}$ (which is set to zero).
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 Erfotac. The mesh has $160 \times 80$ cells in the $x - y$ plane and 32 cells in the $z$ direction with $x_{\text{max}} = 4.5$.

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 $\varepsilon$

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

18.1.2 Section 2

The PANS model is selected

```python
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).

```python
viscos=1/10500
```
18.1.4 Section 6

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

\[
\text{min\_iter}=2
\]

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

\[
\text{solv\_turb}='\text{tdma}' \\
\text{nsweep\_keps}=1
\]

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.

\[
\text{ntstep}=15000 \\
\text{uin}=1 \\
\text{dt}=0.2\times(x2d[1,0]-x2d[0,0])\times\text{np.ones(ntstep)}/\text{uin} \\
\text{itstep\_start}=7500
\]

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 Irannejad [27].

First, compute the viscous forces at the walls,

\[
\text{taus}=\text{np.sum(\text{viscos*as\_bound}*u3d[:,0,:])} \\
\text{taun}=\text{np.sum(\text{viscos*an\_bound}*u3d[:,1,:])}
\]

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

\[
\text{sumps}=\text{np.sum(p3d[:,0,:]*areasx[:,0,:])} \\
\text{total\_forces}=\text{taus+taun+sumps}
\]

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

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

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

\[
\text{beta}=\text{total\_forces/sumvol} \\
\text{su3d}=\text{su3d+beta*vol}
\]
18.2.2 fix_eps

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

```python
# south wall
aw3d[:,0,:]=0
ae3d[:,0,:]=0
as3d[:,0,:]=0
an3d[:,0,:]=0
al3d[:,0,:]=0
ah3d[:,0,:]=0
ap_max=np.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=np.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. 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 $|\mathbf{\nu}'\mathbf{\nu}'_2|$ 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

```python
wale = True
```
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 $y$, $u$, $k$, $\omega$ and $v'_1 v'_2$ profiles are stored on disk in pl_uvw.py). We read $\bar{u}$

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

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 $v'_1 v'_2$

```python
if itstep == 0:
    y_u_k_om=np.loadtxt('y_u_k_om_uv_395.dat')
    y_rans=y_u_k_om[:,0]
    u_rans=y_u_k_om[:,1]
# make it 2D
    u_rans=np.repeat(u_rans[:,None], repeats=nk, axis=1)
    uv_rans=np.abs(y_u_k_om[:,4])
```

A grid in the $z$ direction is created and we call `synt_fluct` to generate the synthetic fluctuations, see Eq. 11.8.

```python
zp = np.linspace(0, zmax, nk)
u synt, vsy nt, wsy nt=synt_fluct(nmodes_synt, itstep, L_t_synt, y_rans, zp, \u v_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

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

Next, we set the initial fields of $v'_2$, $v'_3$ and $v'_3$ (see Eq. 11.13) and compute $a$ and $b$ (see Eqs. 11.10 and 11.11).

```python
usynt_inlet=usynt
vsynt_inlet=vsynt
wsynt_inlet=wsynt
# tturb from ustar=1
    t turb=L_t_synt/1
    a_sy nt=np.exp(-dt[itstep]/tturb)
    b_sy nt=(1.-a_sy nt**2)**0.5
```

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

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

Finally, we superimpose the synthetic fluctuations to the mean flow and store the inlet fields in $u_{bc\_west}$, $v_{bc\_west}$ and $w_{bc\_west}$ which are returned as a results from the `modify_inlet`
19.2. modify case.py

```python
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

```python
su3d[0,:,::]= su3d[0,:,::]+np.maximum(convw[0,:,::],0)*u_bc_west
sp3d[0,:,::]= sp3d[0,:,::]-np.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

```python
su3d[0,:,::]= su3d[0,:,::]+np.maximum(convw[0,:,::],0)*v_bc_west
sp3d[0,:,::]= sp3d[0,:,::]-np.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

```python
su3d[0,:,::]= su3d[0,:,::]+np.maximum(convw[0,:,::],0)*w_bc_west
sp3d[0,:,::]= sp3d[0,:,::]-np.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.

```python
flow_in=np.sum(convw[0,:,::])
flow_out=np.sum(convw[-1,:,::])
ares=areaw[-1,:,::]

area_out=np.sum(areaw[-1,:,::])

ares=areaw[-1,:,::]

Next, compare global inflow and outflow, compute a corrective velocity, \( u_{inc} \) and correct the convective fluxes so that global balance is satisfied.

```python
uinc=(flow_in-flow_out)/area_out
convw[-1,:,::]=convw[-1,:,::]+uinc*ares
```
Note that Neumann boundary conditions are set for $\bar{u}$, $\bar{v}$, ... since

\[
\text{phi_bc_east_type='n'}
\]

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.

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.

\[
\text{kom_des = True}
\]

The interface is automatically computed

\[
j10 = 0
\]

20.1.2 Section 4

The Reynolds number is set to 5, 200.

\[
\text{viscos=1/5200}
\]

20.1.3 Section 6

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

\[
\text{solver_turb='tdma'}
\]
\[
\text{nsweep_kom=1}
\]

20.1.4 Section 10

The boundary conditions for $k$ and $\omega$ at the walls are set.

\[
k_{bc\_south}=np.zeros((ni,nk))
\]
\[
k_{bc\_north}=np.zeros((ni,nk))
\]
\[
k_{bc\_south\_type='d'}
\]
\[
k_{bc\_north\_type='d'}
\]
# boundary conditions for omega
om_bc_south = np.zeros((ni,nk))
om_bc_north = np.zeros((ni,nk))
xwall_s = 0.5 * (x2d[0:-1,0]+x2d[1:,0])
ywall_s = 0.5 * (y2d[0:-1,0]+y2d[1:,0])
dist2_s = (yp2d[:,0]-ywall_s)**2+(xp2d[:,0]-xwall_s)**2
om_bc_south = 10 * 6 * viscos / 0.075 / dist2_s

# make it 2D
om_bc_south = np.repeat(om_bc_south[:,None], repeats=nk, axis=1)
xwall_n = 0.5 * (x2d[0:-1,-1]+x2d[1:,-1])
ywall_n = 0.5 * (y2d[0:-1,-1]+y2d[1:,-1])
dist2_n = (yp2d[:,1]-ywall_n)**2+(xp2d[:,1]-xwall_n)**2
om_bc_north = 10 * 6 * viscos / 0.075 / dist2_n

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

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 \( \omega \). \( k_{\text{init}} \) is set to 20% of the RANS value and \( \omega_{\text{iniy}} \) is set to \( k_{\text{init}}^{1/2} / (0.01 \Delta_{\text{max}}) \).

data = np.loadtxt('y_u_k_om_uv_5200_nj96.txt')
u_rans = data[:,1]

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

k_rans = data[:,2]

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

om_rans = data[:,3]

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

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

vis3d = k3d / om3d + viscos
20.2. modify_case.py

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

```python
if itsstep == 0:
    y_u_k_om=np.loadtxt('y_u_k_om_uv_5200_nj96.txt')
    y_rans=y_u_k_om[:,0]
    u_rans=y_u_k_om[:,1]
# make it 2D
    u_rans=np.repeat(u_rans[:,None], repeats=nk, axis=1)
    k_rans=y_u_k_om[:,2]
# make it 2D
    k_rans=np.repeat(k_rans[:,None], repeats=nk, axis=1)
    eps_rans=y_u_k_om[:,3]
# make it 2D
    eps_rans=np.repeat(eps_rans[:,None], repeats=nk, axis=1)
    uv_rans=np.abs(y_u_k_om[:,4])
# store k and omega
    k_bc_west=k_rans
    om_bc_west=om_rans

Compared to Section we store also \( k \) and \( \omega \) in \( k_{bc\_west} \) and \( \omega_{bc\_west} \).

20.2.3 modify_k

We need to add inlet boundary conditions.

```python
su3d[0,:,\ldots]=su3d[0,:,\ldots]+np.maximum(convw[0,:,\ldots],0)*k_{bc\_west}
sp3d[0,:,\ldots]=sp3d[0,:,\ldots]-np.maximum(convw[0,:,\ldots],0)
vist=vis3d[0,:,\ldots]-viscos
su3d[0,:,\ldots]=su3d[0,:,\ldots]+vist*aw_bound*k_{bc\_west}
sp3d[0,:,\ldots]=sp3d[0,:,\ldots]-vist*aw_bound

We prescribe RANS inlet conditions on \( k \) and \( \omega \). Hence, we must make sure that they are turned into values relevant to LES. This is done by adding commutation terms [28, 29]. It is implemented as:

```python
delt_i1=0.09*(-0.25)*k_{bc\_west}**0.5/om_{bc\_west}
delt_i2=vol[0,:,\ldots]**0.333333
flux_k_RANS=np.maximum(u_{bc\_west},0)*k3d[0,:,\ldots]
vis_smag = (0.1 *delt_i2)**2*gen[0,:,\ldots]**0.5
rk_smag=(vis_smag/delt_i2)**2
flux_k_LES=u3d[0,:,\ldots]*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,:,\ldots]=sp3d[0,:,\ldots]+np.minimum(comm_term,0.)*vol[0,:,\ldots]/k3d[0,:,\ldots]
```
20.2.4 **modify\_om**

Inlet boundary conditions

\[
\begin{align*}
su3d[0,:,:] &= su3d[0,:,:] + \text{np.maximum}(\text{convw}[0,:,:],0) \times \text{om\_bc\_west} \\
sp3d[0,:,:] &= \text{sp3d}[0,:,:] - \text{np.maximum}(\text{convw}[0,:,:],0) \\
vist &= \text{vis3d}[0,:,:] - \text{viscos} \\
su3d[0,:,:] &= su3d[0,:,:] + \text{vist} \times \text{aw\_bound} \times \text{om\_bc\_west} \\
sp3d[0,:,:] &= \text{sp3d}[0,:,:] - \text{vist} \times \text{aw\_bound}
\end{align*}
\]

and the commutation term

\[
\begin{align*}
\text{prod\_extra} &= -\text{om3d}[0,:,:]/\text{k3d}[0,:,:] \times \text{comm\_term} \\
su3d[0,:,:] &= su3d[0,:,:] + \text{np.maximum}(\text{prod\_extra},0) \times \text{vol}[0,:,:]
\end{align*}
\]

## 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 \( \Delta t = 7.83 \times 10^{-4} \). We stretch the grid in the \( y \) direction by 10\% but limit the cell size to \( \Delta y_{\text{max}} = 0.05 \). 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 \( z\_\text{dat} \) 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
```

#### 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 $\omega$.

```python
solver_turb='tdma'
sweep_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).

```python
ntstep=400
uin=1
dt=4*(x2d[1,0]-x2d[0,0])*np.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.

```python
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).

```python
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).

```python
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 $k = 10^{-5}$. 
The wall boundary condition of $\omega$ is multiplied by a factor of 10

$$om_{bc\_south}=10\times6\times\text{viscos}/0.075/dist_{2\_s}$$

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.

```python
# set inlet field in entre domain
u3d=np.repeat(u_bc_west[None,:,:], repeats=ni, axis=0)
k3d=np.repeat(k_bc_west[None,:,:], repeats=ni, axis=0)
om3d=np.repeat(om_bc_west[None,:,:], repeats=ni, axis=0)
vis3d=k3d/om3d+viscos
```

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 22. Look at the script create-inlet-rans-profiles.py. Here we extract $\bar{u}$, $\bar{v}$, $k$, $\omega$ and $v_1^\prime$, $v_2^\prime$ at cells $ni-10$. The data are stored in file $y\_u\_v\_k\_om\_uv\_re-theta\_2500.txt$.

22 DES of boundary layer flow with $k-\omega$ model and commutation terms

Here we will do DES of a developing boundary layer. First, we create the mesh. The script generate-bound-layer-grid.py is used. The mesh in the $y$ direction is the same as in Section 21. The first grid size in the $x$ direction is set as $\Delta x = 0.1\delta_n$ with $\delta_n = 0.86$. The number of cells is set to $ni = 500$. This boundary layer thickness is found from the plot file in Section 21. The stretching of the grid in the $x$ direction is set to 0.3%. The grid in the $z$ direction is defined as $nk=64$ and $z_{max}=1.6$; thus the $z$.dat file reads

```
1.6 64
```

Only the changes compared to Section 21 will commented below
22.1 setup_case.py

22.1.1 Section 1
We choose the central-differencing scheme for convection and Crank-Nicolson for time discretization.

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

22.1.2 Section 2
Choose $k - \omega$ DES model and fix the interface to gridline 33.

```python
kom_des = True
kom = False
j10 = -33
```

22.1.3 Section 6
The `lgmres` solver is chosen for the velocities and the `tdma` for the turbulent quantities. Hence

```python
solver_vel = 'lgmres'
solver_turb = 'tdma'
nsweep_kom = 1
```

The convergence limits are set as

```python
convergence_limit_u = 1e-5
convergence_limit_v = 1e-5
convergence_limit_w = 1e-5
convergence_limit_p = 1e-2
```

and the minimum number of global iterations at each time step is set to one, i.e.

```python
min_iter = 1
```

22.1.4 Section 8
Choose $k - \omega$ DES model. Set the number of time steps to 15,000 and average over the last 7,500. Reduce the time step (compared to Section 21).

```python
ntstep = 15000
uin = 1
dt = 0.25 * (x2d[1,0] - x2d[0,0]) * np.ones(ntstep) / uin
itstep_start = 7500
```
22.1.5 Section 10

Use cyclic boundary conditions in the $z$ direction

```python
    cyclic_z = True
```

No inlet boundary conditions are set here since they depend on the flow field. They are set in `modify_case.py`.

We set the inlet length scale for the synthetic fluctuations, the number of synthetic modes and we don’t mirror the synthetic fluctuations (the shear stress is negative in the entire boundary layer

```python
    L_t_synt=0.2
    nmodes_synt=600
    jmirror_synt=0
```

22.2 `modify_case.py`

22.2.1 `modify_init`

First, we re-define the `dist3d` variable. In module `init` it is computed as the smallest distance to the south and north walls. Here we have only one wall (the south). Note that `dist3d` must be defined as a `global` variable, otherwise the change made to it will not be transmitted to the main module.

```python
global dist3d
# re-define dist3d = distance from south wall
    ywall_s=0.5*(y2d[0:-1,0]+y2d[1:,0])
    dist_s=yp2d-ywall_s[:,None]
    dist=dist_s
    dist3d=np.repeat(dist[:,None,None], repeats=nk, axis=2)
```

Initial condition are set in the same way as in Section 20. The RANS profiles (stored in file `y_u_v_k_om_uv_re-theta-2500.txt`), see Section 21.2.1.

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

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

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

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

# set inlet field in the entire domain
```
23. RANS of hump flow using the \( k - \omega \) model

![Figure 23.1: Hump flow. The grid. Every 8th grid line is shown.](image)

```python
u3d = np.repeat(u_rans[None,:,:], repeats=ni, axis=0)

# set LES values of k and omega in the entire domain
k3d = 0.2*np.repeat(k_rans[None,:,:], repeats=ni, axis=0)
om3d = k3d**0.5/(0.01*delta_max)
vis3d = k3d/om3d+viscos
```

22.2.2 modify inlet

The inlet boundary conditions are now set in this section since they depend on the flow field. We read data from file

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

The synthetic inlet fluctuations are set in exactly the same as in Section 19. Note that inlet the friction velocity must now be computed (it is not equal to one). Also the commutations terms are prescribed in the same was as in Section 20.

23 RANS of hump flow using the \( k - \omega \) model

The grid is shown in Fig. 23.1. It happens to give good results, but that’s probably fortuitous; 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.

The setup of this flow is very similar to that in Section 21. The main difference is 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 \( Re_{\theta} = 6300 \)).

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

```text
0.2 2
```

23.1 setup_case.py

The TDMA solver is found suitable for this flow.

23.1.1 Section 6

```python
solver_vel = 'tdma'
solver_turb = 'tdma'
```
The number of sweeps is set to two for velocities and one for \( k \) ans \( \omega \). Furthermore, we require that at least two global iterations should be made (it improves convergence).

```plaintext	nsweep_vel=2
nsweep_kom=1
min_iter=2
```

Recall that when the solver ‘tdma’ is employed, the convergence limits are not used.

## 24 DES of hump flow using the \( k - \omega \) model

The same grid is used as in Section 23 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. 23.1. 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.

The settings are very similar to those in Section 23 except that we now use central differencing and synthetic inlet fluctuations (cf. the difference between Section 21 and 22).

### 24.1 setup_case.py

The main difference compared to Section 22 is that the TDMA solver is found suitable for this flow.

#### 24.1.1 Section 6

```plaintext
solver_vel='tdma'
solver_turb='tdma'
```

The number of sweeps is set to one and we require that at least two global iterations should be made, i.e.

```plaintext
nsweep_vel=1
nsweep_kom=1
min_iter=2
```

Recall that when the solver ‘tdma’ is employed, the convergence limits are not used.

## 25 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.
25.1 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.

### 25.1.1 New grid

The grid is generated using the script `generate-channel-grid.py`. 90 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-python &
```

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.

### 25.1.2 Boundary wall conditions on $\omega$

The wall boundary conditions on $\omega$ are set in Section 10 in `setup_case.py` according to Eq. 10.2. This is not entirely correct, because it prescribes $\omega$ at the wall, whereas it should be prescribed at the cell center. With the present boundary condition, the value of $\omega$ at the cell center will be too small. Try to compensate this by increasing the value of $\omega$ 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
25.1. Channel flow, RANS

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 $\omega$ at the wall-adjacent cell center? Or is it still too small? If so, increase fact.

Another way is to prescribe $\omega$ 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 $\omega_{wall}$ is the wall boundary condition. However, this option does not work in pyCALC-LES (the simulations diverge 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`

```python
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 be done in `modify_case`, since the $a_P$ and $S_U$ are modified in module `crank_nicol` after leaving `modify_case`. 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 $\omega_{wall}$ as the wall-adjacent cell value) and find out how large the effect is on the results.

25.1.3 $k-\varepsilon$ model

Now simulate the same flow with the AKN $k-\varepsilon$ model. You set `keps=True`. You need to set the wall boundary for $\varepsilon$ according to Eq. 8.4. Do that in module `fix_eps`.

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

### 25.2 Boundary layer flow, RANS

Read Section 21 carefully. This is a developing boundary layer flow. At the inlet, \( \bar{u} = 1 \), \( \bar{\omega} = 1 \) and \( k = 10^{-4} \) 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 22 (but you need to increase \( n_i \)). Neumann boundary conditions are used at the free (north) boundary for \( k \) and \( \omega \). Do some sensitivity checks.

- Is the flow sensitive to the inlet values of \( k \) and \( \omega \)?
- The TDMA solved is used for \( k \) and \( \omega \).
  - 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 \( \omega \))
- The \( \omega_{wall} \) value is set to \( 10 \omega_{wall} \). See Section 10 in setup_case.py What happens if you fix is to \( \omega_{wall} \) in the center of the cell (as you did in Section 25.1.2).

### 25.3 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 \( nj = 40 \). 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).

In the full channel (i.e. \(y_{\text{max}} = 2\) 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 \(j_{\text{mirror\_synt}}=0\).

In \texttt{modify\_init} and \texttt{modify\_inlet}, the variables \(y_{\text{rans}}, u_{\text{rans}}\) and \(uv_{\text{rans}}\) 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.

\[
y_{\text{rans}} = y_{\text{u\_k\_om}}[0:nj,0]
\]

Finally, reduce the integration to \texttt{ntstep} and \texttt{itstep\_start}.

Now run the code. The simulation should take approximately half an hour. By loosening the convergence limits in the Python 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 \texttt{ntstep} and \texttt{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 \(n_{\text{modes\_synt}}=600\). What happens if you increase or decrease it? What about the CPU time?

- The SGS viscosity is plotted. You find that \(\nu_{\text{sgs}}/\nu \simeq 1\). 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 [21]?

- 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 \texttt{grep time} at the prompter)? Can you loosen the convergence criteria?

- The integral turbulent timescale of the synthetic fluctuations is set to \(L_t/\tau_u\) (see \texttt{tturb=L_t\_synt in modify\_inlet}). Note that this value gives \(a_{\text{synt}}=0.994\) and \(b_{\text{synt}}=0.108\) (see file \texttt{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 \texttt{synt\_fluct}. It reads the files \texttt{a\_synt\_inlet.dat} and \texttt{R\_synt\_inlet.dat}. The eigenvalues and the eigenvectors have been computed using a Reynolds stress tensor created with EARSFM and a 1D RANS simulation. They were computed using the script \texttt{compute\_a\_and\_R\_from\_earsm.py}. Try another Reynolds stress tensor (e.g. from DNS). This task is \texttt{optional}.

- Change any other parameters. For example, you can make more changes in the synthetic fluctuation generator (file \texttt{../synt\_fluct.py}).
25.4 Channel flow, inlet-outlet with heat transfer, $Re = 395$

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.

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 first 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;
/usr/bin/time -a -o out python -u exec-pyCALC-LES.py > out
```

Now you need to define many new variables in file globals such as $t_{bc_{east}}$, $t_{bc_{west}}$, $t_{bc_{south}}$, $t_{bc_{north}}$.

You need to initialize temperature (see line 1560 in pyCALC-LES-heat.py) with the command

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

Then you need to create a new routine for temperature, calc_t. You need to call coeff, bc ... You can, for example, copy the lines used for u3d (see line 1704 in pyCALC-LES-heat.py). You need to define a viscous Prandtl number ($\text{prand\_visc}$) in the call to coeff; don’t forget to add $\text{prand\_visc}$ 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.

25.4.1 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

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

(25.1)

see, e.g., Section 11.1 in [22]. $\beta$ 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. 25.1 to su3d in module modify_v (don’t forget to multiply by volume).
25.5 RANS of channel flow at \( Re_{\tau} = 5200 \): \( k - \omega \) 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 \( \Delta y \) 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 \( \varepsilon \) are given in Section 11.14.1 in [22]. They can be summarized as

\[ \bar{u}: \text{set the wall shear stress as } \tau_w = \rho u_\tau^2 \text{ (recall that } \rho = 1 \). \]  
\[ \text{The log-law reads} \]
\[ \frac{\bar{u}}{u_\tau} = \frac{1}{\kappa} \ln \left( \frac{E u_\tau y}{\nu} \right) \]  
(25.2)

where \( E = 0 \) and \( \kappa = 0.41 \).

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

\[ \omega: \text{set } \omega \text{ at the wall-adjacent cells as } \omega_P = C_\mu^{-1/2} u_\tau / (\kappa y_{wall}), \text{ see Eq. 3.27 in [30]} \]

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 \( \omega \) 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_\varepsilon` and `fix_omega`.

- The expression for \( u_\tau \) in the log-law (Eq. 25.2) is implicit. Hence, compute \( u_\tau \) from the log-law in an iterative way (you could make 2–3 iterations).

- Print \( u_\tau \) 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 `pluvw.py`. The friction velocity is here computed as

\[ u_\tau = \left( \frac{\nu}{\nu} \frac{\partial \bar{u}}{\partial y} \right)^{1/2} \]

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

25.6 Channel flow, inlet-outlet, \( Re_{\tau} = 5200 \)

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 25.3. Run the code, plot and compare with the results in Section 20.
25.7 \textit{Channel flow, inlet-outlet, }$Re_\tau = 5200$, \textit{using wall functions}

25.6.1 Neumann boundary condition on $k$

The discretized commutation term in the $k$ equation is in effect a negative convection term [28]. 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 [28]). Make the changes, run the code and compare the results with those in Section 25.6.

25.6.2 No commutation terms

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

25.6.3 No commutation terms in URANS region

As discussed in [28], 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 25.6?

25.7 \textit{Channel flow, inlet-outlet, }$Re_\tau = 5200$, \textit{using wall functions}

Implement wall functions in the same way as Section 25.5. Copy all files from Section 25.6. Modify the grid, setup\_case and modify\_case in the same way as in Section 25.5.

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 [31].

25.8 \textit{Channel flow, fully developed, }$Re_\tau = 5200$

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 25.6. In Section 10 in setup\_case.py, set

```
cyclic_x = True
```

You don’t need to change $u_{bc\,\text{west\_type}}$, $u_{bc\,\text{east\_type}}$... $om_{bc\,\text{east\_type}}$.

We must have reasonably good initial condition. A good way it to use the results in Section 25.6 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 $u_\tau$ is (or $\tau_w$). It should be equal to one (because $\tau_w$ must balance the pressure gradient, see Section “Force balance, channel flow” in [22]). 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?

### 25.8.1 Wall boundary condition of $\omega$

In Section 25.1.2 you investigated the sensitivity of the flow to the wall boundary condition of $\omega$. You compared three different boundary conditions.

1. Equation 10.2 (this is what you used in Section 25.8)
2. Multiply Eq. 10.2 by a factor of 10.
3. Set Eq. 10.2 in the cell center by using the module fixomega.

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 $\omega$ values differ from the correct value in Eq. 10.2?

### 25.8.2 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 $j_{l0}$ to a negative value, i.e.
   \[
   \text{np.abs}(j_{l0}) = j_{l0}.
   \]
2. The LES length scale is $\Delta$, see Eq. 10.1. Replace $\Delta$ by the IDDES length scale, $\Delta_{dw}$, see Eq. 8 in [29]. Note that you must not use any for loops. Run the code and compare with the results obtained in Section 25.8.

### 25.8.3 Change turbulence model

Up to now, you have used the standard Wilcox $k-\varepsilon$ model. Now switch to the $k-\omega$ model used in [29]

### 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 $u$, $v$ and $w$ equations (1: fully implicit)
- acrank_conv_keps: time integration scheme for convection and diffusion in $k$ and $\varepsilon$ equations (1: fully implicit)
A. Variables in pyCALC-LES

acr ank_con v_kom: time integration scheme for convection and diffusion in $k$ and $\omega$ equations (1: fully implicit)

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

am g_cycle: type of cycle in the pyAMG solver for the pressure equation (‘V’, ‘W’, ‘F’, ‘AMLI’)

am g_cycle_phi: type of cycle in the pyAMG solver for all equations except the pressure equation (‘V’, ‘W’, ‘F’, ‘AMLI’)


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

apo3d: $a_P^0$, 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

area$: 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)

c_eps1: $C_{\varepsilon_1}$ coefficient in the $k-\varepsilon$ model

c_eps2: $C_{\varepsilon_2}$ coefficient in the $k-\varepsilon$ model

c_omega1: $C_{\omega_1}$ coefficient in the $k-\omega$ model

c_omega2: $C_{\omega_2}$ coefficient in the $k-\omega$ model

cmu: $C_{\mu}$ 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 $\varepsilon, k, \omega$ (max(limit,limit norm(su3d))); if negative: abs(limit))
convergence_limit_om, convergence_limit_k, convergence_limit_om: convergence limit in Python solver for $\varepsilon, k, \omega$ (max(limit,limit · norm(su3d)); if negative: abs(limit))

convergence_limit_p: convergence limit in Python solver for $\bar{p}$ (relative limit)

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

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

convergence_limit_w: convergence limit in Python solver for $\bar{w}$ (max(limit,limit · 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 ($\Delta x, \Delta y, \Delta z$)

dist3d: smallest distance to south or north wall

dmin_snt: 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, $\varepsilon$

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

epsbc_east, epsbc_north, epsbc_south, epsbc_west, epsbc_high, epsbc_low: boundary values of $\varepsilon$ at east, north, south, west and high/low boundary. Default: 0

epsbc_east_type, epsbc_north_type, epsbc_south_type, epsbc_west_type: see below

epsbc_high_type, epsbc_low_type: type of b.c. for $\varepsilon$ ('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: $f_x, f_y, f_z$, the interpolation function in $i, j$ and $k$ direction
A. Variables in \texttt{pyCALC-LES}

\begin{itemize}
  \item \texttt{gen: $P_k^\text{ex}$ excluding the turbulent viscosity (used in the $k$, $\varepsilon$ and $\omega$ equations)}
  \item \texttt{imon, jmon, kmon: print time history of variables for this node}
  \item \texttt{iter: current global iteration}
  \item \texttt{itstep: current time step}
  \item \texttt{itstep\_save: instantaneous and time-averaged field are saved on disk every \texttt{itstep\_save} time step}
  \item \texttt{itstep\_start: time averaging starts}
  \item \texttt{itstep\_stats: time averaging is done every \texttt{itstep\_stats} time step}
  \item \texttt{itstep\_stats\_counter: counter for how many samples are used for time averaging}
  \item \texttt{j10: when $j10 < 0$, the \texttt{LES-RANS} interface in the $k - \omega$ DES model is fixed at cell \texttt{np.abs(j10)}
  \item \texttt{jmirror\_synt: the sign of the $v$ synthetic are changed for nodes $j \geq jm\text{mirror}$ (in module \texttt{synt\_fluct})}
  \item \texttt{k3d: modeled turbulent kinetic energy, $k$}
  \item \texttt{k3d\_mean: time-averaged modeled turbulent kinetic energy, $\langle k \rangle$}
  \item \texttt{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}
  \item \texttt{k\_bc\_east\_type, k\_bc\_north\_type, k\_bc\_south\_type, k\_bc\_west\_type: see below}
  \item \texttt{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}
  \item \texttt{keps: the \texttt{AKN} $k - \varepsilon$ model is used (RANS)}
  \item \texttt{kom: the Wilcox $k - \omega$ model is used (RANS)}
  \item \texttt{kom\_des: the DES Wilcox $k - \omega$ model is used}
  \item \texttt{Lt\_synt: length scale of the synthetic fluctuations, see Eq. 11.4}
  \item \texttt{maxit: maximum number of global iterations (solving $\bar{u}, \bar{v}, \bar{w}, \bar{p}$, ...)}
  \item \texttt{ni, nj, nk: number of cell centers in i, j and k direction}
  \item \texttt{nmodes\_synt: number of modes when generating synthetic fluctuations}
  \item \texttt{norm\_order: order of norm when computing residual for $\bar{u}, \bar{v}, \bar{w}, k, \varepsilon$ and $\omega$. Default: 2}
  \item \texttt{nsweep\_keps: maximum number of iterations in the Python solver when solving the $k$ and $\varepsilon$ equations in solver called in \texttt{solve\_3d} }
\end{itemize}
A. Variables in pyCALC-LES

\texttt{nsweep.kom}: maximum number of iterations in the Python solver when solving the \( k \) and \( \omega \) equations in solver called in \texttt{solve\_3d}

\texttt{nsweep.vel}: maximum number of iterations in the Python solver when solving the \( \bar{u}, \bar{v} \) and \( w \) equations in solver called in \texttt{solve\_3d}

\texttt{ntstep}: number of time steps

\texttt{om3d}: specific dissipation of turbulent kinetic energy, \( \omega \)

\texttt{om3d.mean}: time-averaged modeled specific dissipation of turbulent kinetic energy, \( \langle \omega \rangle \)

\texttt{om\_bc\_east, om\_bc\_north, om\_bc\_south, om\_bc\_west, om\_bc\_high, om\_bc\_low}: boundary values of \( \omega \) at east, north, south, west and high/low boundary. Default: 0

\texttt{om\_bc\_east\_type, om\_bc\_north\_type, om\_bc\_south\_type, om\_bc\_west\_type}: see below

\texttt{om\_bc\_high\_type, om\_bc\_low\_type}: type of b.c. for \( \omega \) (‘d’=Dirichlet, ’n’=Neumann’ or ’2’=\( \partial^2 \omega / \partial n^2 = 0 \)). Default: Dirichlet

\texttt{p3d}: pressure, \( \bar{p} \)

\texttt{p3d.mean}: time-averaged pressure, \( \langle \bar{p} \rangle \)

\texttt{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

\texttt{p\_bc\_east\_type, p\_bc\_north\_type, p\_bc\_south\_type, p\_bc\_west\_type}: see below

\texttt{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

\texttt{pans}: PANS (based on \( k - \varepsilon \)) or PITM is used. PANS is used when \texttt{prand\_k} and \texttt{prand\_eps} are positive, otherwise PITM

\texttt{prand\_eps}: \( \sigma_\varepsilon \), turbulent Prandtl number in the \( \varepsilon \) equation

\texttt{prand\_k}: \( \sigma_k \), turbulent Prandtl number in the \( k \) equation

\texttt{prand\_omega}: \( \sigma_\omega \), turbulent Prandtl number in the \( \omega \) equation

\texttt{residual\_p}: residual for the continuity equation

\texttt{residual\_u}: residual for the \( \bar{u} \) equation

\texttt{residual\_v}: residual for the \( \bar{v} \) equation

\texttt{residual\_w}: residual for the \( \bar{w} \) equation

\texttt{resnorm\_p}: the residual of the continuity equation is normalised by this quantity

\texttt{restart}: a restart from a previous simulation is made, see Section 14.24
A. Variables in pyCALC-LES

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 'q'=QUICK, see Section 14.10

scheme turb: discretization scheme for $k$, $\varepsilon$ and $\omega$. '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 the $\bar{p}$ 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 $k$, $\varepsilon$ and $\omega$. 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 bc. 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, $\langle V'_1 \rangle_m$, see 11.9

uu3d stress: time-averaged resolved stress, $\langle \bar{v}_1' \bar{v}_1' \rangle$

uv3d stress: time-averaged resolved stress, $\langle \bar{v}_1' \bar{v}_2' \rangle$

v3d: $\bar{v}$ velocity

v3d mean: time-averaged $\bar{v}$ velocity, $\langle \bar{v} \rangle$
A. Variables in pyCALC-LES

\( \text{v}_{\text{bc\_east}}, \text{v}_{\text{bc\_north}}, \text{v}_{\text{bc\_south}}, \text{v}_{\text{bc\_west}}, \text{v}_{\text{bc\_high}}, \text{v}_{\text{bc\_low}}: \)
boundary values of \( \bar{v} \) at east, north, south, west and high/low boundary. Default: 0

\( \text{v}_{\text{bc\_east\_type}}, \text{v}_{\text{bc\_north\_type}}, \text{v}_{\text{bc\_south\_type}}, \text{v}_{\text{bc\_west\_type}}: \)
see below

\( \text{v}_{\text{bc\_high\_type}}, \text{v}_{\text{bc\_low\_type}}: \) type of b.c. for \( \bar{v} \) (‘d’=Dirichlet, ‘n’=Neumann’ or ‘2’=\( \partial^2 v/\partial n^2 = 0 \)). Default: Dirichlet

\( \text{vis3d}: \) total viscosity, \( \nu + \nu_t \)

\( \text{vis3d\_mean}: \) time-averaged total viscosity, \( \langle \nu_t + \nu \rangle \)

\( \text{viscos}: \) viscosity, \( \nu \). Note that \( \nu = \mu \) since \( \rho = 1 \).

\( \text{vol}: \) volume of a control volume

\( \text{vsynt\_inlet}: \) synthetic inlet fluctuation in the \( y \) direction, \( (V'_2)_{m} \), see 11.9

\( \text{vtk}: \) if TRUE, save results in VTK format

\( \text{vtk\_file\_name}: \) file name of VTK output files

\( \text{vtk\_movie}: \) if TRUE, save results every \( \text{itstep\_save\_time\_step} \) in VTK format

\( \text{vv3d\_stress}: \) time-averaged resolved stress, \( \langle v'_2^2 \rangle \)

\( \text{w3d}: \) \( \ddot{w} \) velocity

\( \text{w3d\_mean}: \) time-averaged \( \ddot{w} \) velocity, \( \langle \ddot{w} \rangle \)

\( \text{w}_{\text{bc\_east}}, \text{w}_{\text{bc\_north}}, \text{w}_{\text{bc\_south}}, \text{w}_{\text{bc\_west}}, \text{w}_{\text{bc\_low}}, \text{w}_{\text{bc\_high}}: \)
boundary values of \( \bar{w} \) at east, north, south, west, and high/low boundary. Default: 0

\( \text{w}_{\text{bc\_east\_type}}, \text{w}_{\text{bc\_north\_type}}, \text{w}_{\text{bc\_south\_type}}, \text{w}_{\text{bc\_west\_type}}: \)
see below

\( \text{w}_{\text{bc\_high\_type}}, \text{w}_{\text{bc\_low\_type}}: \) type of b.c. for \( \bar{w} \) (‘d’=Dirichlet, ‘n’=Neumann’ or ‘2’=\( \partial^2 w/\partial n^2 = 0 \)). Default: Dirichlet

\( \text{wale}: \) the WALE model is used

\( \text{wsynt\_inlet}: \) synthetic inlet fluctuation in the \( z \) direction, \( (V'_3)_{m} \), see 11.9

\( \text{ww3d\_stress}: \) time-averaged resolved stress, \( \langle v'_3^2 \rangle \)

\( \text{x2d}: \) the \( x \) coordinate of a corner of a control volume, see Fig. 1.3

\( \text{xp2d}: \) the \( x \) coordinate of the center of a control volume, see Fig. 1.3

\( \text{y2d}: \) the \( y \) coordinate of a corner of a control volume, see Fig. 1.3

\( \text{yp2d}: \) the \( y \) coordinate of the center a control volume, see Fig. 1.3

\( \text{z}: \) the \( z \) coordinate of the face of a control volume, see Fig. 1.4

\( \text{zmax}: \) extent of the computational domain in the \( z \) direction

\( \text{zp}: \) 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 $a_W, a_E, a_S, a_N, a_L, a_P, S_n$ 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, $n_i \times n_j = (3, 4)$

\[
\begin{bmatrix}
C0 & C1 & C2 & C3 & C4 & C5 & C6 & C7 & C8 & C9 & C10 & C11 \\
L0: & a_P,0 & -a_N,0 & 0 & 0 & -a_E,0 & 0 & 0 & 0 & -a_W,0 & \\
L1: & -a_S,1 & a_P,1 & -a_N,1 & 0 & 0 & -a_E,1 & 0 & 0 & 0 & -a_W,1 & \\
L2: & 0 & -a_S,2 & a_P,2 & -a_N,2 & 0 & 0 & -a_E,2 & 0 & 0 & -a_W,2 & \\
L3: & 0 & 0 & -a_S,3 & a_P,3 & 0 & 0 & -a_E,3 & 0 & 0 & -a_W,3 & \\
L4: & -a_W,4 & 0 & 0 & 0 & a_P,4 & -a_N,4 & 0 & 0 & -a_E,4 & 0 & \\
L5: & 0 & -a_W,5 & 0 & 0 & -a_S,5 & a_P,5 & -a_N,5 & 0 & 0 & -a_E,5 & 0 & 0 & \\
L6: & 0 & 0 & -a_W,6 & 0 & -a_S,6 & -a_P,6 & -a_N,6 & 0 & 0 & -a_E,6 & 0 & \\
L7: & 0 & 0 & 0 & -a_W,7 & 0 & 0 & -a_S,7 & -a_P,7 & 0 & 0 & -a_E,7 & \\
L8: & -a_E,8 & 0 & 0 & 0 & -a_W,8 & 0 & 0 & 0 & a_P,8 & -a_N,8 & 0 & \\
L9: & 0 & -a_E,9 & 0 & 0 & 0 & -a_W,9 & 0 & 0 & -a_S,9 & a_P,9 & -a_N,9 & 0 & \\
L10: & 0 & 0 & -a_E,10 & 0 & 0 & 0 & -a_W,10 & 0 & 0 & -a_S,10 & a_P,10 & -a_N,10 & \\
L11: & 0 & 0 & 0 & -a_E,11 & 0 & 0 & 0 & -a_W,11 & 0 & 0 & -a_S,11 & a_P,11 & \\
\end{bmatrix}
\]

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

Figure B.2: Matrix, \( A \), for 2D flow, \( n_i \times n_j = (3, 2) \). 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 $x,i$

#### 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.

```
\[
\begin{bmatrix}
  C0 & C1 & C2 & C3 & C4 & C5 & C6 & C7 & C8 & C9 & C10 & C11 \\
  L0 : & ap_0 & a_H & a_N & a_E & 0 & 0 & 0 & -aw_0 & 0 & 0 & 0 \\
  L1 : & -a_L & a_P & 0 & a_N & 0 & -a_E & 0 & 0 & 0 & -aw_1 & 0 & 0 \\
  L2 : & -a_S & 0 & a_H & 0 & 0 & -aw_2 & 0 & 0 & 0 & 0 & 0 & 0 \\
  L3 : & 0 & -a_S & a_L & a_P & 0 & 0 & 0 & -aw_3 & 0 & 0 & 0 & 0 \\
  L4 : & -aw & 0 & 0 & 0 & ap & -a_H & -a_N & 0 & -a_E & 0 & 0 & 0 \\
  L5 : & 0 & -aw & 0 & 0 & -a_L & ap & 0 & 0 & 0 & -aw & 0 & 0 \\
  L6 : & 0 & 0 & -aw & 0 & -a_S & 0 & ap & -a_H & 0 & 0 & -aw & 0 \\
  L7 : & 0 & 0 & 0 & -aw & 0 & -aw & 0 & a_P & 0 & 0 & 0 & -aw \\
  L8 : & -a_E & 0 & 0 & 0 & -aw & 0 & 0 & a_P & -a_H & -aw & 0 & 0 \\
  L9 : & 0 & -a_E & 0 & 0 & 0 & -aw & 0 & 0 & a_P & 0 & -aw & 0 \\
  L10 : & 0 & 0 & -aw & 0 & 0 & 0 & -aw & 0 & 0 & a_P & -aw & 0 \\
  L11 : & 0 & 0 & 0 & -aw & 0 & 0 & 0 & -aw & 0 & 0 & a_P & -aw \\
\end{bmatrix}
\]  

(B.3)
```
B.4 3D grid, \( n_i \times n_j \times n_k = (2, 2, 3) \), cyclic in \( z, k \)

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

Figure B.4: Matrix, \( A \), for 3D flow, \( n_i \times n_j \times n_k = (2, 2, 3) \). Cyclic in \( z, k \). The coefficients due to cyclic boundary conditions are colored in blue.

C Using pyAMGx on GPU

pyAMGx is a Python interface to the NVIDIA AMGX library. pyAMGx can be used to construct complex solvers and preconditioners to solve 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) nVidia graphics card. You can check which graphics card you have with the Linux command

```
lspci
```

Look for the line starting with \texttt{lUSB controller}:

Start by getting the nVidia CUDA toolkit. In Ubuntu, type

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

After installation you can check the installation

```
nvcc -version
```

You need to install the AMGX library. Instructions are found at here.

The AMGX library is compiled with the \texttt{gcc} compiler. There seems to be some problems with version \texttt{gcc 10.3}
C. Using pyAMGx on GPU

- gcc.gnu.org
- github.com

Check which gcc compiler that is installed on your computer

dpkg -l | grep gcc | awk '{print $2}"

If you have version gcc 10.3, you may need to install version 9 (Ubuntu command)

sudo apt-get install gcc-9 g++-9

Then build code with flags to gcc-9. Start with no mpi functionality to reduce possible errors, i.e.

mkdir build
cd build

cmake -D CMAKE_NO_MPI="TRUE" -D CMAKE_C_COMPILER=gcc-9 -D CMAKE_CXX_COMPILER=g++-9 .
make all

Before installing pyamgx, you should export the following environment variables

1. AMGX_DIR: Path to the AMGX project root directory
2. AMGX_BUILD_DIR: If AMGX was built in a directory other than $AMGX_DIR/build

Now set AMGX_BUILD_DIR to that directory. For example

export AMGX_DIR=/home/lada/AMGX-main

Download pyAMGx and install it

git clone https://github.com/shwina/pyamgx

cd pyamgx

pip install .

Now just use

import pyamgx

in your Python script!

To select the pyamgx solver in pyCALC-LES, set

solver_p='pyamgx' pyamgx

in setup_case.
C. Using pyAMGx on GPU

References


C. Using pyAMGx on GPU


C. Using pyAMGx on GPU


