Fluid Mechanics MTF053

Numerical Analysis of Fully Developed Channel Flow

Computer Assignment 1

Division of Fluid Dynamics
Department of Mechanics and Maritime Sciences
Chalmers University of Technology

Contents

1 Instructions

The design task should be done by a group of max four students. All sub-tasks must be solved and presented orally to the responsible assistant at an assessment meeting (you will be contacted by the responsible assistant to set a time and date for the meeting). Plots and figures that show that the group has completed the tasks should be brought to the assessment meeting (one printout per group is enough)

1.1 Related course material

You can find information in the course book *F. M. White Fluid Mechanics* 8^{th} *ed.* (White, 2016) and in the following documents:

1. MTF053_C04.pdf (lecture notes chapter 4)
2. MTF053_C06.pdf (lecture notes chapter 6)
3. MTF053_Equation-for-Boundary-Layer-Flows.pdf
4. MTF053_Turbulence.pdf
5. MTF053_Formulas-Tables-and-Graphs.pdf

1.2 Python scripts and data files

Download the following files:

1. LaminarBL.py
2. TurbulentBL.py
3. MTF053_CA2_TurbulentBL.txt

2 Introduction

In this exercise, you will study a fully developed channel flow (flow between two parallel plates) numerically. You will start with a laminar flow case as that problem can be solved analytically and thus it is possible to make a comparison and get a feeling for the accuracy of the numerical method.

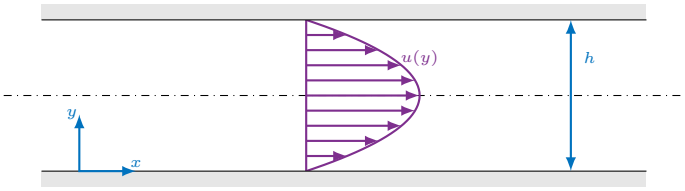

Figure 2.1: Channel flow definition

Fig. 2.1 above defines the geometry. A fluid flows through the channel from left to right and we want to find the velocity field. Gravity is aligned with the negative y-direction.
Since the flow will be fully symmetric, it is sufficient to simulate half the channel. The boundary conditions that should be applied are therefore a solid wall and a symmetry boundary condition.

Geometry	
channel height h	6.0 cm
Boundary Conditions	
solid wall (no slip)	$u = 0$ at $y = 0$
symmetry	$\frac{\partial u}{\partial y} = 0$ at $y = h/2$
Flow Properties	
fluid	air @ atmospheric pressure and $20°C$
Case I – laminar flow	
pressure gradient	$\partial p/\partial x = -0.01$ Pa/m
Case II – turbulent flow	
pressure gradient	$\partial p/\partial x = -8.0$ Pa/m

3 Case I – Laminar Flow

Starting from the Navier-Stokes equations in two dimensions and assuming laminar flow, one can show that the differential equations describing the flow reduces to

$$\frac{\partial}{\partial y}\left(\mu\frac{\partial u}{\partial y}\right) = \frac{\partial p}{\partial x} \tag{3.1}$$

The Navier-Stokes equations in two dimensions are given by the continuity equation and the x and y components of the momentum equation

$$\frac{\partial u}{\partial x} + \frac{\partial v}{\partial y} = 0 \tag{3.2}$$

$$\rho\left(u\frac{\partial u}{\partial x} + v\frac{\partial u}{\partial y}\right) = -\frac{\partial p}{\partial x} + \rho g_x + \frac{\partial}{\partial x}\left(\mu\frac{\partial u}{\partial x}\right) + \frac{\partial}{\partial y}\left(\mu\frac{\partial u}{\partial y}\right) \tag{3.3}$$

$$\rho\left(u\frac{\partial v}{\partial x} + v\frac{\partial v}{\partial y}\right) = -\frac{\partial p}{\partial y} + \rho g_y + \frac{\partial}{\partial x}\left(\mu\frac{\partial v}{\partial x}\right) + \frac{\partial}{\partial y}\left(\mu\frac{\partial v}{\partial y}\right) \tag{3.4}$$

Task 1.1:

Show that the Navier-Stokes equations Eqns. 3.2 – 3.4 reduces to Eqn. 3.1 for fully developed laminar channel flow in two dimensions. Examining Eqn. 3.1, it is obvious that for our specified application, it is possible to reduce the number of terms in Eqns. 3.2 – 3.4 significantly, which means that you will remove terms in the equations as part of the derivations. Please note, however, that all such modifications of the equations must be justified.

Hint: the lecture notes for chapter 4 (`MTF053_C04.pdf`) may be of use here.

Task 1.2:

a) Solve Eqn. 3.1 analytically for the above-listed boundary conditions and flow properties

b) Calculate the Reynolds number

Remember that we are investigating the flow in a two-dimensional channel, i.e., it is **not** a pipe flow. The correct hydraulic diameter for such a *non-circular pipe flow* can be found in `MTF053_Formulas-Tables-and-Graphs.pdf`. Also, the Reynolds number should be calculated using the average velocity, which can be obtained by integrating over the channel

$$V = \frac{1}{h}\int_0^h u(y)dy$$

With the analytical solution in place, it's now time to attack the problem numerically. The numerical solution will then be compared to the analytical solution. In the numerical approach, the flow velocity will be approximated in a discrete number of points. This is accomplished by dividing the domain (in our case a line from $y = 0$ to $y = 3.0$ cm) in a number, let's say N, cells. In each of the N cells there is a node that may or may not be located in the cell center. In our rather simple case, we will divide the domain into N cells of equal size – an equidistant mesh – and the nodes will be placed in the cell centers. The result looks like in Fig. 3.1 below. The next step is to set up a system of algebraic equations with one equation per node, i.e., N equations and N unknowns.

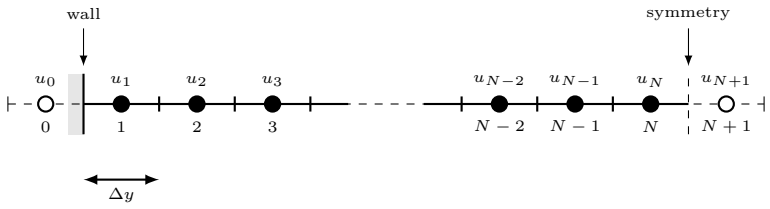

Figure 3.1: Computational domain

There are in principle three available methods to discretize an equation like Eqn. 3.1, namely the Finite-Element Method (FEM), Finite Difference, and the Finite Volume Method (FVM). In the Finite-Element Method (FEM), the velocity u is approximated by linear combination of base functions each of which is zero everywhere except for in one cell. Using FEM, you will have the possibility to have control over the numerical accuracy. The drawback with the method is that the implementation becomes rather complex. FEM is often used in solid mechanics where the governing equations are less difficult to handle than in fluid mechanics. There are however several examples of solvers for fluid flows based on FEM.

A simple, and often useful way to solve simple numerical problems is to expand all derivatives completely and approximate them using Taylor expansions. Expanding the derivatives in Eqn.3.1, we get

$$\frac{\partial}{\partial y}\left(\mu\frac{\partial u}{\partial y}\right) = \frac{\partial p}{\partial x} \Rightarrow \mu\frac{\partial^2 u}{\partial y^2} + \frac{\partial \mu}{\partial y}\frac{\partial u}{\partial y} = \frac{\partial p}{\partial x}$$

Now, replacing the derivatives with Taylor expansions we get the following equation for cell number i in our mesh

$$\mu\frac{u_{i+1} - 2u_i + u_{i-1}}{\Delta y^2} + \frac{\mu_{i+1} - \mu_{i-1}}{2\Delta y}\frac{u_{i+1} - u_{i-1}}{2\Delta y} = \frac{\partial p}{\partial x}$$

The viscosity is constant which means that the second term on the left-hand side of the equation would be zero, but it's kept in the equation to show how to approximate a first-order derivative. This discretization method is often referred to as finite difference. Finite difference is useful for simple applications. It is often used for the discretization of temporal derivatives. It is however difficult to use for curvilinear problems, i.e., anything that is not rectangular. Moreover, numerical errors introduced in the discretization may result in mass or momentum sources, which leads to a solution that does not fulfill the continuity equation and that is of course problematic. In the Finite-Volume Method (FVM), the equations are integrated over each of the cells. By integrating a governing equation over each of the cells, we get an equation for each cell. Integrating Eqn. 3.1 over cell number i, we get

$$\left(\mu\frac{\partial u}{\partial y}\right)_{i+\frac{1}{2}} - \left(\mu\frac{\partial u}{\partial y}\right)_{i-\frac{1}{2}} = \frac{\partial p}{\partial x}\Delta y_i \tag{3.5}$$

In order to be able to perform the calculations we need estimates of the velocity derivatives at the cell faces. An approximation of the derivative can be obtained using the velocities in the cells closest to the cell face

$$\left(\mu\frac{\partial u}{\partial y}\right)_{i+\frac{1}{2}} = \mu\frac{u_{i+1} - u_i}{y_{i+1} - y_i} = \{y_{i+1} - y_i = \Delta y\} = \mu\frac{u_{i+1} - u_i}{\Delta y}$$

$$\left(\mu\frac{\partial u}{\partial y}\right)_{i-\frac{1}{2}} = \mu\frac{u_i - u_{i-1}}{y_i - y_{i-1}} = \{y_i - y_{i-1} = \Delta y\} = \mu\frac{u_i - u_{i-1}}{\Delta y}$$

Inserted in Eqn. 3.5 and assuming equidistant mesh, we get for cell i

$$\mu\frac{u_{i+1}-u_i}{\Delta y}-\mu\frac{u_i-u_{i-1}}{\Delta y}=\frac{\partial p}{\partial x}\Delta y$$

which gives

$$u_{i+1}-2u_i+u_{i-1}=\frac{\partial p}{\partial x}\frac{\Delta y^2}{\mu} \tag{3.6}$$

As can be seen, we get the same results as we would get using the finite difference technique for this specific problem.

The expression given by Eqn. 3.6 is for a generic cell i and if applied to all cells we will get a system of N equations. The remaining problem now is to implement the boundary conditions. If we examine the equation for cell 1 (the cell closest to the wall) we get

$$u_2-2u_1+u_0=\frac{\partial p}{\partial x}\frac{\Delta y^2}{\mu}$$

We don't have a value for u_0 (see Fig. 3.1) appearing in this equation but using a ghost value technique, we can calculate a value for u_0 using the boundary condition. The wall boundary condition is a no-slip condition, i.e., the velocity should be zero at the cell face that coincides with the wall. Setting the velocity u_0 equal to $u_0=-u_1$, we will assure that the wall velocity will be zero.

$$u(y=0)=\frac{1}{2}(u_1+u_0)=0.\Rightarrow u_0=-u_1$$

Now, let's have a look at the equation for cell N

$$u_{N+1}-2u_N+u_{N-1}=\frac{\partial p}{\partial x}\frac{\Delta y^2}{\mu}$$

In order to evaluate the equation for cell N we need the velocity in cell $N+1$, which again is outside of our computational domain (see Fig. 3.1). We will use a ghost cell technique to implement the boundary condition here as well. The appropriate boundary condition is that the velocity derivative should be zero at the cell face that coincides with the centerline of the channel. Setting u_{N+1} equal to u_N, we will fulfill the boundary condition.

$$\left.\frac{\partial u}{\partial y}\right|_{y=h/2}=\frac{u_{N+1}-u_N}{\Delta y}=0.\Rightarrow u_{N+1}=u_N$$

Task 1.3:

Write the equation system that we get by evaluating Eqn. 3.6 in all of the N cells on matrix form, i.e., write Eqn. 3.6 as $A\mathbf{u} = \mathbf{b}$, where $\mathbf{u}$ is a vector with N elements. What does the matrix A and the vector $\mathbf{b}$ look like?

To solve this rather simple problem, all we have to do now is to invert the matrix A and then the velocity values are calculated as $\mathbf{u} = A^{-1}\mathbf{b}$

Task 1.4:

Write a `Python` code that solves the equation system for the specified boundary conditions and a given number of cells N. The tricky part is to generate the matrix A. Use the `Python` functions `scipy.sparse.diags`, `scipy.sparse.linalg.spsolve`, and `numpy.ones` (see the example in `LaminarBL.py` and/or Appendix A).

Note! the `sparse` library functions are used to reduce the computational time.

With both the numerical and analytical solution in place, we can estimate the discretization error. Let's define the error as the absolute value of the relative difference between the numerical velocity value and the analytical velocity value in the node closest to the centerline of the channel.

$$e = \left| \frac{u_{numeric} - u_{analytic}}{u_{analytic}} \right| \tag{3.7}$$

Task 1.5:

a) Calculate the error (e) according to Eqn. 3.7 for 10 different values of N spaning from $N = 10$ to $N = 1000$. The `Python` function `numpy.logspace` might be useful here (see example below).

b) Make a plot that shows the relative error as a function of cell count (N). Use the command `loglog` instead of `plot`. How does the error decay with increased cell count?

Note! $u_{analytic}$ should be calculated for the y-coordinate corresponding to the cell center of the last cell, not at the center of the duct.

```
import numpy as np
import matplotlib.pyplot as plt

NN = np.int_(np.logspace(1,3,10))

err = []

for N in NN:

  # solve problem for AU=b for N cells
  # calculate relative error

  ...

  err.append(relative_error)

# plot relative error versus cell count

fig = plt.figure()
ax  = fig.add_subplot(111)
ax.loglog(NN,err,'-o',linewidth=2)
ax.set_xlabel('number of cells',fontsize=14)
ax.set_ylabel('relative error [%]',fontsize=14)
ax.set_title('Relative error as a function of cell count',fontsize=14)
ax.set_xlim(NN[0],NN[-1])
ax.set_ylim(err[-1],err[0])
for label in (ax.get_xticklabels() + ax.get_yticklabels()):
  label.set_fontsize(14)
ax.grid(which='both')
```

4 Case II – Turbulent Flow

Now we are ready for the real challenge – we will simulate the same flow but under turbulent conditions. Note that the pressure gradient is increased from 0.01 Pa/m to 8.0 Pa/m, i.e., the pressure gradient driving the flow is increased but geometry, fluid properties, and boundary conditions are the same as in the laminar case.

Since the flow is now assumed to be turbulent, we will apply the Reynolds averaging technique which is based on a that the flow variables are written as the sum of a temporal average and a fluctuating part

$$u = \overline{u} + u', \; v = \overline{v} + v', \; p = \overline{p} + p' \tag{4.1}$$

where the overlined quantities are the time averages and the primed quantities are unsteady fluctuations. The time averages of the fluctuating quantities are zero.

Task 2.1:

Replace u, v, and p in the Navier-Stokes equations (Eqns. 3.2-3.4) with the expressions given in Eqn. 4.1 and time average the equations to get the Reynolds-Averaged Navier-Stokes (RANS) equations. Use the fact that the time average of a fluctuation is zero. Then, use the same arguments as in Task 1.1 to show that the equations can be simplified to the following relation for our specific problem

$$0 = -\frac{\partial \overline{p}}{\partial x} + \frac{\partial}{\partial y}\left(\mu \frac{\partial \overline{u}}{\partial y} - \rho \overline{u'v'}\right) \tag{4.2}$$

Hint: make use of the derivation available in the lecture notes for Chapter 6 (`MTF053_C06.pdf`).

The terms $-\rho\overline{u'v'}$, $-\rho\overline{u'^2}$, and $-\rho\overline{v'^2}$ appearing in the RANS equations are called Reynolds stresses since they appear as stresses in the averaged equations. The Reynolds stresses are unknowns in the RANS equations. The introduction of new unknowns is a direct consequence of the averaging of the equations and is often referred to as the closure problem. New unknowns are introduced but the number of equations remains. Turbulence modeling as a means to close the system of equations is a research field of its own and there are several approaches, more or less complicated. A rather simple approach to model $-\rho\overline{u'v'}$ that works well for fully developed channel flow is Prandtl's mixing length model. The mixing length model is based on the assumption that the turbulent shear stress behaves as the viscous shear stress and therefore can be expressed as

$$-\rho\overline{u'v'} = \mu_{turb}\frac{\partial \overline{u}}{\partial y} \tag{4.3}$$

where μ_{turb} is the eddy viscosity that, in contrast to the fluid viscosity, depends on the flow itself, i.e., it is not a fluid property. According to the mixing length model μ_{turb} can be obtained as

$$\mu_{turb} = \rho l_m^2 \frac{\partial \overline{u}}{\partial y} \tag{4.4}$$

where $l_m = \kappa y$

l_m is the mixing length (the property that has given the model its name) and κ is the von Kármán constant ($\kappa = 0.41$). For more details see MTF053_Turbulence.pdf.

The total viscosity is the sum of the fluid viscosity and the eddy viscosity

$$\mu_{tot} = \mu + \mu_{turb} \tag{4.5}$$

Now, replacing μ with μ_{tot}, u with $\overline{u}$, and p with $\overline{p}$ in Eqn. 3.1, the equation can be used for turbulent flow.

$$\frac{\partial}{\partial y}\left(\mu_{tot}\frac{\partial \overline{u}}{\partial y}\right) = \frac{\partial \overline{p}}{\partial x} \tag{4.6}$$

or

$$\frac{\partial}{\partial y}\left(\left(\mu + \rho\kappa^2 y^2 \frac{\partial \overline{u}}{\partial y}\right)\frac{\partial \overline{u}}{\partial y}\right) = \frac{\partial \overline{p}}{\partial x} \tag{4.7}$$

So far so good. One problem remains to be solved though. The wall boundary condition needs to be modified. The viscous sublayer is extremely thin and resolving that using an equidistant mesh leads to a need for very small cells and thus the computational cost increases significantly. Moreover, the mixing length model doesn't work very well in the viscous sublayer. Instead, we will assume that the center of the cell closest to the wall is located within the log-layer, and thus the log law can be used. The ghost cell value to the left of the first cell (see Fig. 3.1) is calculated using the log law

$$\overline{u}_0 = \frac{u^*}{\kappa}\ln\left(\frac{y_0 u^*}{\nu}\right) + Bu^* \tag{4.8}$$

where $B = 5.0$ and y_0 is selected such that $y_0^+ = (y_0 u^*)/\nu = 30$ and thus we will ensure that the first cell is within the log region.

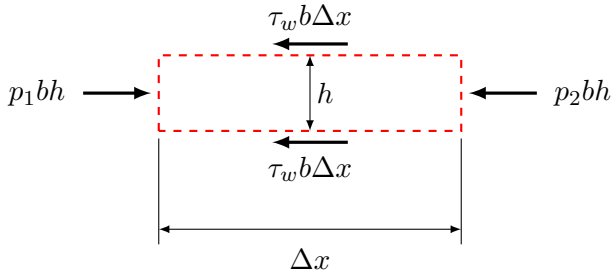

Figure 4.1: Forces on a control volume with length Δx (the channel height is h and the width into the paper is b).

Note that u^* by definition is related to the pressure gradient. Figure 4.1 shows a schematic representation of the forces on a small control volume with length Δx. A force balance in the flow direction gives

$$(p_1 - p_2)bh - 2\tau_w b\Delta x = 0 \Rightarrow \tau_w = \frac{(p_1 - p_2)h}{2\Delta x} = \{p_1 - p_2 = -\Delta p\} = -\frac{\Delta p}{\Delta x}\frac{h}{2}$$

u^* is defined as $\tau_w = \rho u^{*2}$ and thus

$$\rho u^{*2} = -\frac{\Delta p}{\Delta x}\frac{h}{2}$$

Now, let the length of the control volume go to zero

$$u^* = \sqrt{\left(-\frac{\partial p}{\partial x}\right)\frac{h}{2\rho}} \tag{4.9}$$

As for the laminar flow case we will use finite volume discretization (FVM) and the equations for each of the cells are obtained by integrating Eqn. 4.7 over the cells and inserting approximations of the velocity derivatives at the cell faces.

Integration of Eqn. 4.7 over cell i gives

$$\left(\left(\mu + \rho\kappa^2 y^2 \frac{\partial \overline{u}}{\partial y}\right)\frac{\partial \overline{u}}{\partial y}\right)_{i+\frac{1}{2}} - \left(\left(\mu + \rho\kappa^2 y^2 \frac{\partial \overline{u}}{\partial y}\right)\frac{\partial \overline{u}}{\partial y}\right)_{i-\frac{1}{2}} = \frac{\partial \overline{p}}{\partial x}\Delta y_i$$

Now, we need to insert cell face coordinates and approximations of the velocity gradients at the cell faces. In the following it is assumed that the mesh is equidistant and thus $\Delta y_i = \Delta y$

$$\left(\left(\mu + \rho\kappa^2 \left(\frac{y_{i+1} + y_i}{2}\right)^2 \frac{\overline{u}_{i+1} - \overline{u}_i}{\Delta y}\right) \frac{\overline{u}_{i+1} - \overline{u}_i}{\Delta y}\right) - \left(\left(\mu + \rho\kappa^2 \left(\frac{y_i + y_{i-1}}{2}\right)^2 \frac{\overline{u}_i - \overline{u}_{i-1}}{\Delta y}\right) \frac{\overline{u}_i - \overline{u}_{i-1}}{\Delta y}\right) = \frac{\partial \overline{p}}{\partial x} \Delta y$$

Collecting terms gives

$$\begin{aligned} &\left(\mu + \rho\kappa^2 \left(\frac{y_i + y_{i-1}}{2}\right)^2 \frac{\overline{u}_i - \overline{u}_{i-1}}{\Delta y}\right) \overline{u}_{i-1} - \\ &\left(2\mu + \rho\kappa^2 \left(\left(\frac{y_{i+1} + y_i}{2}\right)^2 \frac{\overline{u}_{i+1} - \overline{u}_i}{\Delta y} + \left(\frac{y_i + y_{i-1}}{2}\right)^2 \frac{\overline{u}_i - \overline{u}_{i-1}}{\Delta y}\right)\right) \overline{u}_i + \\ &\left(\mu + \rho\kappa^2 \left(\frac{y_{i+1} + y_i}{2}\right)^2 \frac{\overline{u}_{i+1} - \overline{u}_i}{\Delta y}\right) \overline{u}_{i+1} = \frac{\partial \overline{p}}{\partial x} \Delta y^2 \end{aligned} \tag{4.10}$$

In Eqn. 4.10 above, i takes all values from 1 to N. As in the laminar case, the system of equations obtained if applying Eqn. 4.10 to all cells in the computational domain can be expressed in matrix form as $A\mathbf{u} = \mathbf{b}$, but now matrix A will depend on $\mathbf{u}$ and $\mathbf{y}$.

A possible approach to solve the numerical problem would be to iterate to find the solution.

1. Initialize vector $\mathbf{u}$ with a velocity value for each cell (start guess)
2. Calculate the coefficients of A based on $\mathbf{u}$ and $\mathbf{y}$
3. Invert Matrix A
4. Calculate $\mathbf{u}$ as $\mathbf{u} = A^{-1}\mathbf{b}$

   Iterate 2-4 until converged

The provided `Python` script `turbulentBL.py` (also available in Appendix B) does just that.

Task 2.2:

In the provided `Python` script `turbulentBL.py` (also available in Appendix B), set the number of cells to $N = 1000$. Run the script for 50, 100, 200, and 300 iterations and monitor the development of the solution – how does the number of iterations affect the velocity profile? Suggest a measure of convergence for the numerical simulation and estimate the number of iterations needed to reach convergence.

Task 2.3:

u^+ is a non-dimensional velocity and y^+ is a non-dimensional wall-normal coordinate, respectively. These properties are defined as

$$u^+ = \overline{u}/u^* \text{ and } y^+ = yu^*/\nu$$

Plot u^+ as a function of y^+. Use `semilogx` instead of `plot`

Task 2.4:

When converged, the velocity profile has significantly lower velocities than the parabolic profile that was used as a starting guess. What is the reason for that? Suggest an alternative starting guess.

Task 2.5:

Calculate the Reynolds number for the turbulent flow.

Task 2.6:

In the provided `Python` script, there is a variable called `relax` used as follows:

```
U = relax*spsolve(A,b)+(1.-relax)*Uold
```

where `U` is a vector containing the velocity values in all cells, `A` is a coefficient matrix, `b` is the right-hand side vector, `spsolve` is a `Python` function solving linear equation systems (in our case `AU=b`), and `Uold` is a vector containing the velocity values from the previous iteration. What does the `relax` variable do?

Task 2.7:

The provided data file MTF053_CA2_TurbulentBL.txt contains measured velocities obtained for a turbulent flow over a **flat plate** in a wind tunnel. Plot u^+ as a function of y^+ for the simulated turbulent channel flow and for the measured data. Compare the two profiles and comment on any differences and try to draw some conclusions.

References

F. M. White. *Fluid Mechanics*. McGraw-Hill, New York, 8 edition, 2016.

A Laminar Boundary-Layer Solver

The provided `Python` script `LaminarBL.py` is a template for a sparse-matrix based solver for numerical estimation of the boundary layer velocity profile in a fully developed laminar channel flow. In order to be able to run the script you will need to add some additional lines of code (indicated in the template file).

```
import numpy as np
from scipy.sparse import diags
from scipy.sparse.linalg import spsolve
import matplotlib.pyplot as plt

#=========================================================
# numerical calculation of a laminar boundary layer
# velocity profile using a finite-volume approach
#=========================================================
#
#      0     1     2            N-1    N    N+1
#   ------------------ - - - -------------------
#   |     |     |     |       |     |     |     |
#   |  o  |  o  |  o  |       |  o  |  o  |  o  |
#   | G1  |     |     |       |     |     | G2  |
#   ------------------ - - - -------------------
#         ^                                 ^
#        y=0.                              y=h/2
#        u=0.                             du/dy=0
#
# The computational domain contains N cells ranging
# from cell 1 to cell N. G1 and G2 are ghost cells
# (cells outside of the computational domain used for
# boundary condition implementation).
#=========================================================

if __name__ == "__main__":

  savefig = 1

  h     =  0.06   # channel height
  mu    =  1.8e-5 # viscosity
  rho   =  1.2    # density
  dpdx  = -0.01   # pressure gradient

  # number of cells

  N = 10 # <= CHANGE CELL COUNT HERE

  # allocate vectors

  y = np.zeros(N+2)
  U = np.zeros(N+2)

  # calculate cell size (dy) assuming equidistant grid

  dy=(h/2.0)/N

  # add wall-normal coordinates for all cells (cell 1 to N)

  y[slice(1,N+1)] = np.linspace(dy/2.0,0.5*h-(dy/2.0),N)

```

```
# add centerline coordinate

y[-1]=h/2.

# calculate the velocity for all y-coordinates (analytic solution)

Ua = ... # <= ADD ANALYTIC SOLUTION HERE Ua=f(y)

# generate matrix diagonal vector and off-diagonal vectors

diagonal  = -2.0*np.ones(N)
off_diagonal = np.ones(N-1)

# set boundary conditions (update diagonal elements)

diagonal[ 0] = ... # <= ADD LEFT-END BOUNDARY CONDITION HERE
diagonal[-1] = ... # <= ADD RIGHT-END BOUNDARY CONDITION HERE

# generate sparse matrix (A)

A = diags([diagonal,off_diagonal,off_diagonal],[0, -1, 1],format="csr")

# generate right-hand-side vector (b)

b = (dpdx*dy**2/mu)*np.ones(N)

# solve linear equation system AU=b => U=inv(A)b

U[slice(1,N+1)] = spsolve(A,b)

# update boundary velocity

U[-1] = U[-2]

# plot velocity profile

fig = plt.figure()
ax  = fig.add_subplot(111)
ax.plot(U,y,linewidth=2,label='numeric')
ax.plot(Ua,y,linewidth=2,label='analytic')
ax.set_xlabel('axial velocity (u [m/s])',fontsize=14)
ax.set_ylabel('wall-normal coordinate (y [m])',fontsize=14)
ax.set_title('Velocity profile - laminar channel flow',fontsize=14)
ax.legend(fontsize=14)
ax.set_xlim(0,0.30)
ax.set_ylim(0,0.5*h)
for label in (ax.get_xticklabels() + ax.get_yticklabels()):
  label.set_fontsize(14)
ax.grid()
if savefig:
  plt.savefig('laminar-profile.png',bbox_inches='tight')
else:
  plt.show()
```

B Turbulent Boundary-Layer Solver

The `Python` script `TurbulentBL.py` provides a sparse-matrix based solver for numerical estimation of the boundary layer velocity profile in a fully developed turbulent channel flow. Make changes according to the specifications given in Task 2.2.

```
import numpy as np
from scipy.sparse import diags
from scipy.sparse.linalg import spsolve
import matplotlib.pyplot as plt

#=========================================================
# numerical calculation of a turbulent boundary layer
# velocity profile using a finite-volume approach
#=========================================================
#
#      0     1     2            N-1    N    N+1
#   ------------------- - - - -------------------
#   |     |     |     |       |     |     |     |
#   |  o  |  o  |  o  |       |  o  |  o  |  o  |
#   | G1  |     |     |       |     |     | G2  |
#   ------------------- - - - -------------------
#      ^                                   ^
#     y=y0                                 y=h/2
#     u=u0                                du/dy=0
#
# The computational domain contains N cells ranging
# from cell 1 to cell N. G1 and G2 are ghost cells
# (cells outside of the computational domain used for
# boundary condition implementation).
#=========================================================

if __name__ == "__main__":

  savefig = 1

  N     =  1000   # number of cells
  niter =  100    # number of iterations

  h     =  0.06   # channel height
  mu    =  1.8e-5 # fluid viscosity
  rho   =  1.2    # fluid density
  dpdx  = -8.0    # pressure gradient
  kappa =  0.41   # von Karman constant
  B     =  5.0    # integration constant in the log-law
  relax =  0.5    # under relaxation

  # we will assume that we are in the log-region and thus the
  # lower boundary will coincide with the lower end of the
  # log region (here assumed to be at y+=30). The velocity
  # at y+=30 is calculated using the log law.

  yplus0 = 30
  ustar  = np.sqrt(-dpdx*h/(2.0*rho))
  y0     = yplus0*mu/(rho*ustar)
  u0     = ustar*np.log(yplus0)/kappa+B*ustar

  # calculate cell size (dy) assuming equidistant grid

```

```
54  dy = (0.5*h-y0)/(N+0.5)
55
56  # generate cell center coordinate vectors
57
58  # cell centers (including ghost cells): y_(0) to y_(N+1)
59  y = np.linspace(y0,0.5*h+0.5*dy,N+2)
60
61  # cell centers (excluding ghost cells): u_(1) to y_(N)
62  y0=y[slice(1,N+1)]
63
64  # right-shifted cell centers: y_(2) to y_(N+1)
65  # (right ghost cell (G2) included)
66  yp = y[slice(2,N+2)]
67
68  # left-shifted cell centers: y_(0) to y_(N-1)
69  # (left ghost cell (G1) included)
70  ym = y[slice(0,N)]
71
72  # generate right-hand-side vector (b)
73
74  b0 = dpdx*dy**2
75  b  = b0*np.ones(N)
76
77  # initialize velocity vector (U)
78  # here a laminar solution is used as the initial condition
79
80  U = dpdx*y*(y-h)/(2.0*mu)
81
82  # implement boundary conditions
83  U[ 0]=u0
84  U[-1]=U[-2]
85
86  # cell center velocity vector: U_(1) to U_(N)
87  U0=U[slice(1,N+1)]
88
89  # right-shifted velocity vector: U_(2) to U_(N+1)
90  # (right ghost cell (G2) included)
91  Up=U[slice(2,N+2)]
92
93  # left-shifted velocity vector: U_(0) to U_(N-1)
94  # (left ghost cell (G1) included)
95  Um=U[slice(0,N)]
96
97  # update velocity vector iteratively until converged
98  # (increase the number of iterations if not converged)
99
100 for i in range(niter):
101
102   # store the velocity vector from the previous iteration
103   Uold = U.copy()
104
105   # generate matrix diagonal and off-diagonal elements
106
107   off_diagonal_p =  (rho*kappa**2* 0.25*(yp+y0)**2*(Up-U0)/dy+mu)
108   diagonal       = -(rho*kappa**2*(0.25*(yp+y0)**2*(Up-U0)/dy+\
109                      0.25*(y0+ym)**2*(U0-Um)/dy)+2*mu)
110   off_diagonal_m =  (rho*kappa**2* 0.25*(y0+ym)**2*(U0-Um)/dy+mu)
111
112   # assemble a sparse matrix (A)
113
```

```
    A = diags([diagonal,off_diagonal_m[slice(1,N)],off_diagonal_p[slice(0,N-1)]],[0, -1, 1],format="csr")

    # update right-hand-side vector (boundary cell values)

    b[ 0] = b0 - off_diagonal_m[ 0]*u0
    b[-1] = b0 - off_diagonal_p[-1]*U0[-1]

    # calculate a new velocity vector solving the linear system AU=b
    U[slice(1,N+1)] = relax*spsolve(A,b)+(1.-relax)*Uold[slice(1,N+1)]

    # set the velocity in the right ghost cell (G2) to
    # U_(N), which ensures that du/dy=0 at the center
    U[-1] = U[-2]

    print('iteration %4d of %4d' %(i+1,niter))


# set last y-coordinate to channel center
y[-1] = h/2.

# plot velocity profile

fig = plt.figure()
ax  = fig.add_subplot(111)
ax.plot(U,y,linewidth=2)
ax.set_xlabel('axial velocity (u [m/s])',fontsize=14)
ax.set_ylabel('wall-normal coordinate (y [m])',fontsize=14)
ax.set_title('Velocity profile - turbulent channel flow',fontsize=14)
for label in (ax.get_xticklabels() + ax.get_yticklabels()):
  label.set_fontsize(14)
ax.grid()
if savefig:
  plt.savefig('turbulent-profile.png',bbox_inches='tight')
else:
  plt.show()
```