ENGG*6790: Theory and Applications of Turbulence

Introduction to Python Programming

Amir A. Aliabadi

July 3, 2017

1 Introduction

`Python` is a widely used high-level programming language for general-purpose programming, created by Guido van Rossum and first released in 1991. An interpreted language, `Python` has a design philosophy which emphasizes code readability (notably using whitespace indentation to delimit code blocks rather than curly brackets or keywords), and a syntax which allows programmers to express concepts in fewer lines of code than possible in languages such as C++ or Java. The language provides constructs intended to enable writing clear programs on both a small and large scale.

`Python` features a dynamic type system and automatic memory management and supports multiple programming paradigms, including object-oriented, imperative, functional programming, and procedural styles. It has a large and comprehensive standard library.

`Python` interpreters are available for many operating systems, allowing `Python` code to run on a wide variety of systems. `CPython`, the reference implementation of `Python`, is open source software and has a community-based development model, as do nearly all of its variant implementations. `CPython` is managed by the non-profit `Python` Software Foundation.

Figure 1: The logo of `Python` programming language.

2 Installation

To install Python, go to the following link and download the latest version for the appropriate operating system. Complete the installation steps.

https://www.python.org/downloads/

The standard application launcher for Python is IDLE, which opens a Shell. A shell is a user interface for access to an operating system's services. In general, operating system shells use either a command-line interface (CLI) or graphical user interface (GUI), depending on a computer's role and particular operation. It is named a shell because it is a layer around the operating system kernel. IDLE is a CLI shell that is shown in Figure 2.

Figure 2: An example of the IDLE shell that is accessible by standard installation of Python on Mac operating system.

Another application launcher for Python is IDE from PyCharm that can be downloaded from the following link. Download the Community version of IDE from PyCharm, which is lightweight and

suitable for scientific development, for the appropriate operating system.

`http://www.jetbrains.com/pycharm/download/`

By launching `IDE` from `PyCharm` the following window opens that asks to open a new project or an existing project, as shown in Figure 3. Chose to open a new project and specify the directory path for the project files to be developed.

Figure 3: Openning screen of the `IDE` launcher.

3 Creating and Running a Simple Program

Click menu item `File` and then `New` to create a `Python` text file. Name the file the same as this lab's title, i.e. `PythonProgramming`. To run a simple python program to print `Hello World!` in the output console, enter the following lines of code within the editor just created for `PythonProgramming`.

```
import random
import sys
import os

#This line prints a message
print("Hello World!")
```

The `import` command allows us to use modules from various libraries in order to perform specific programming tasks. The `random` module allows us to generate a random number. The `sys` and `os` modules launch the operating system. The line `#This line prints a message` is a comment

that is not going to be executed. The `print("...")` command prints a message on the output console. In `Python` programming both double quotations " and single quotations ' perform the same task. For instance we could have used `print('...')` in the above program.

Then click menu item `Run` and then click `Run`. A run console sub window opens within the `IDE` where the message `Hello World!` will be printed, as shown in Figure 4.

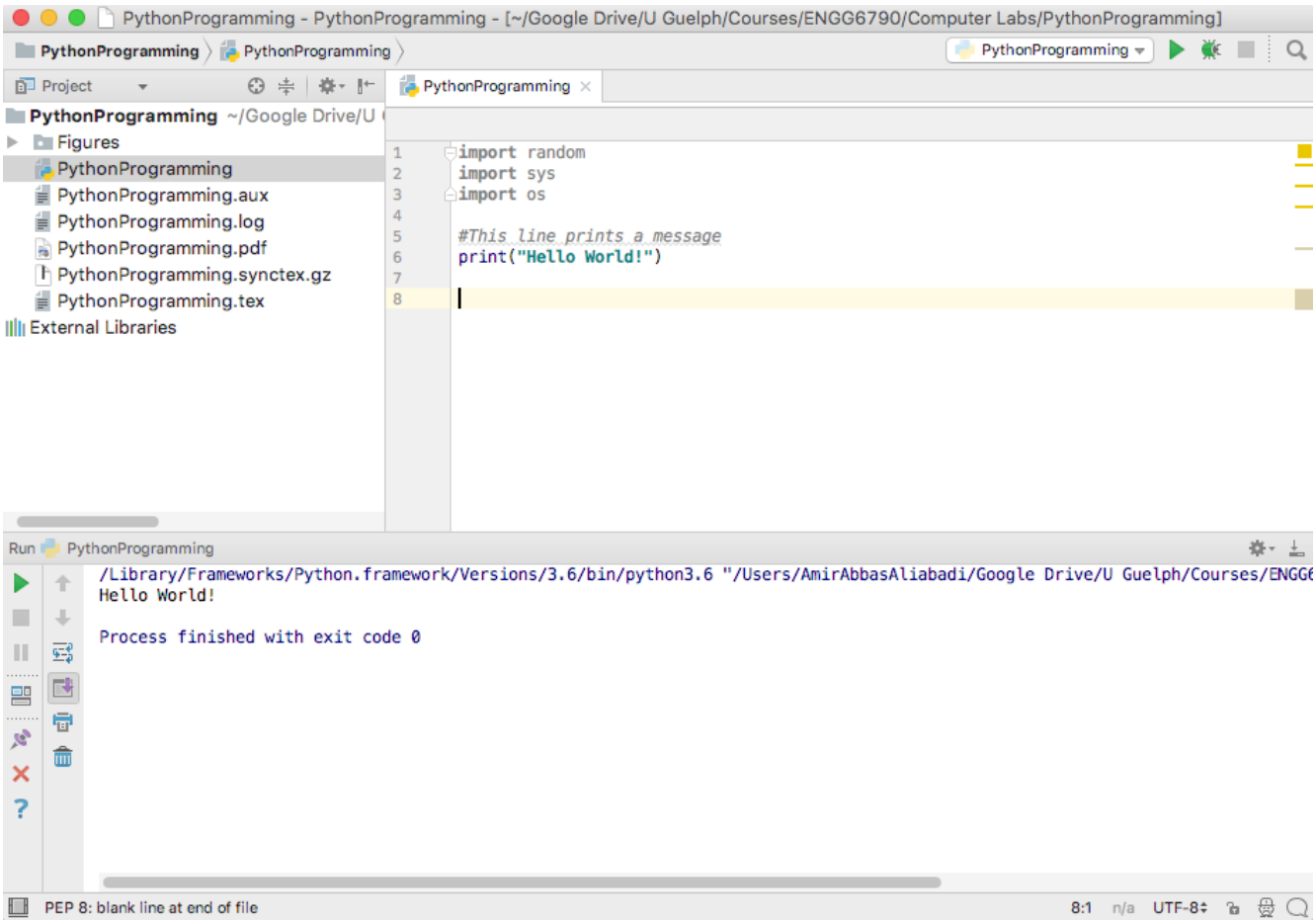


Figure 4: A simple program to print `Hello World!` in the output console.

4 Installing Python Interpreter Packages

`Python` programming is based on numerous interpreter packages that have been developed by the community. The appropriate packages for a `Python` program must be installed before a package can be used using the `import` command. To install an interpreter package the menu item `PyCharm Community Edition` should be used and then `Preferences` must be clicked. When the `Preferences` window opens, the `Project Interpreter` under the current project can be opened and viewed, as shown in Figure 5.

The list of all interpreter packages can be viewed by double-clicking on a package. A list of all

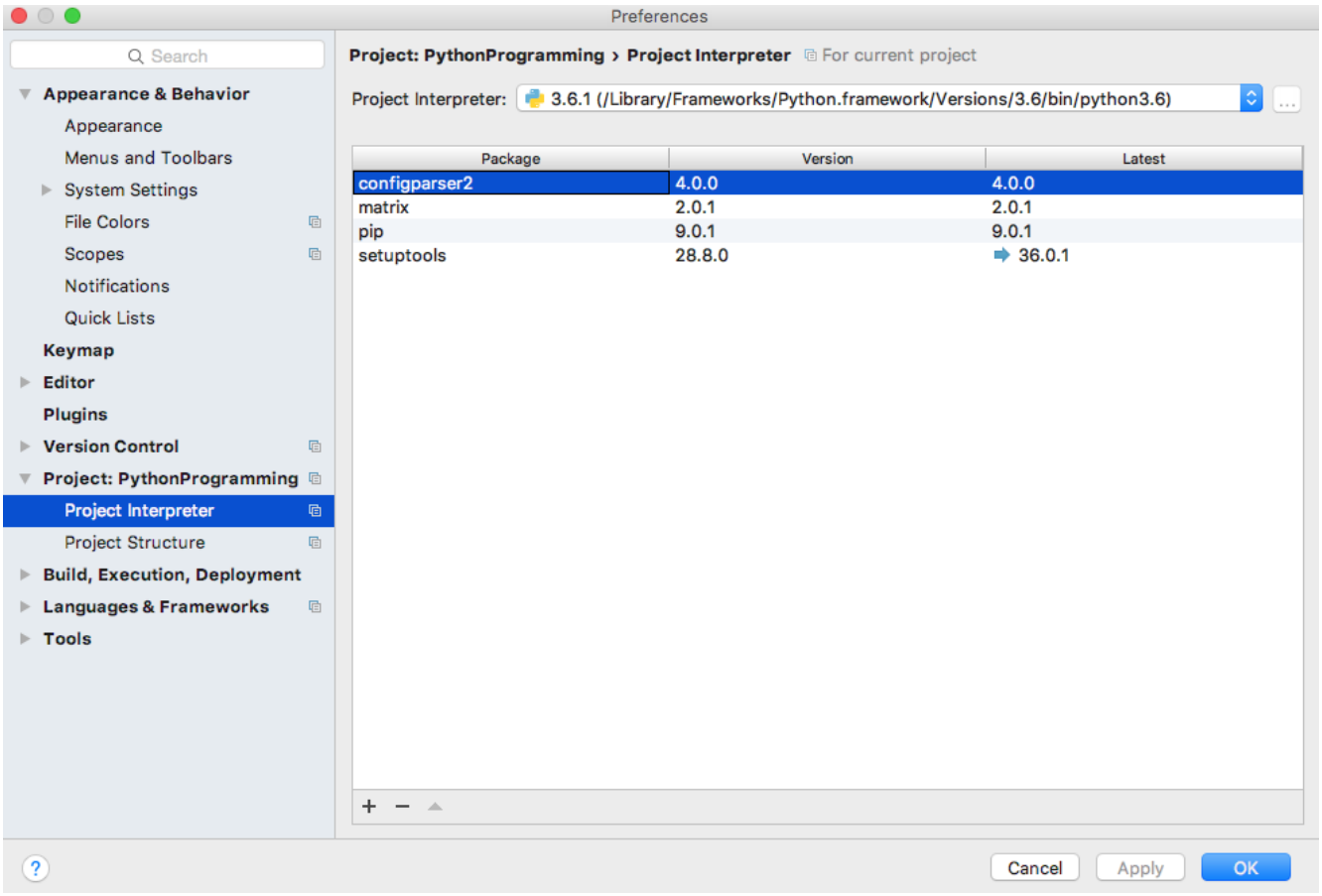


Figure 5: `Preferences` window and the `Project Interpreter`.

interpreters sorter alphabetically can be obtained. It is possible to search for an interpreter by starting to type the interpreter name as shown in Figure 6. For instance, information about the `numpy` interpreter is shown in the `Available Packages` window. A brief description is provided on the right. This interpreter is used for array processing for numbers, strings, records, and objects. The `Version` and interpreter `Author` is also provided. It is possible to install the interpreter package by selecting the desired version and then check marking the `Install` option. Finally the botton `Install Package` can be clicked. After installation, this package or module can be used by the `import` command in the `Python` program.

5 Calculation of Reynolds Stress Tensor and Turbulent Kinetic Energy

It was discussed in lectures that the instantaneous velocity vector in a turbulent flow can be decomposed into the mean velocity and the turbulent fluctuating velocity in the so called Reynolds decomposition, i.e.

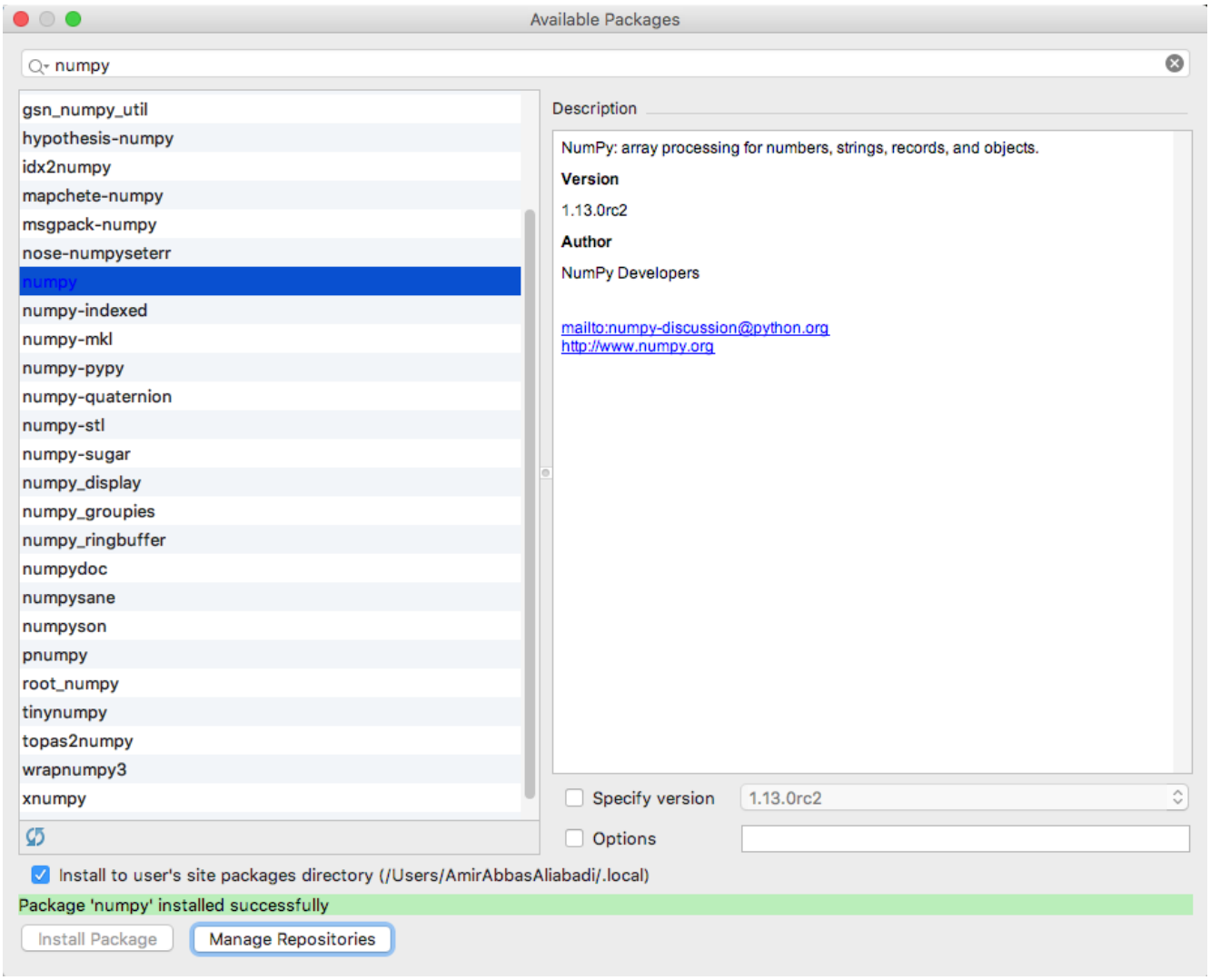


Figure 6: The `numpy` interpreter.

$$\underbrace{\mathbf{U}(\mathbf{x},t)}_{\text{Instantaneous Velocity}} = \underbrace{\langle \mathbf{U}(\mathbf{x},t) \rangle}_{\text{Mean Velocity}} + \underbrace{\mathbf{u}(\mathbf{x},t)}_{\text{Fluctuating Velocity}} , \tag{1}$$

where $\mathbf{x}$ is position and t is time. In the cartesian coordinates with x, y, and z coordinate axes, the velocity components for this equation can be written as

$$U = \langle U \rangle + u \tag{2}$$
$$V = \langle V \rangle + v \tag{3}$$
$$W = \langle W \rangle + w. \tag{4}$$

It is worth remembering that Reynolds stresses $\langle u_i u_j \rangle$ are components of a second-order tensor, with the property that it is symmetric, i.e. $\langle u_i u_j \rangle = \langle u_j u_i \rangle$. The diagonal components of this

tensor, i.e. $\langle u_1^2 \rangle = \langle u_1 u_1 \rangle$, $\langle u_2^2 \rangle$, and $\langle u_3^2 \rangle$ are called *normal stresses*, while the off-diagonal components, e.g. $\langle u_1 u_2 \rangle$, are called *shear stresses*. The Reynolds stress tensor for a flow using a Cartesian coordinate system can be shown in the matrix as follows

$$\begin{bmatrix} \langle u^2 \rangle & \langle uv \rangle & \langle uw \rangle \\ \langle vu \rangle & \langle v^2 \rangle & \langle vw \rangle \\ \langle wu \rangle & \langle wv \rangle & \langle w^2 \rangle \end{bmatrix}$$

The turbulent kinetic energy is defined as the one half of the sum of the normal stresses of the Reynolds stress tensor. In the Cartesian coordinate system, the turbulent kinetic energy can be defined as

$$k = \frac{1}{2} \left(\langle u^2 \rangle + \langle v^2 \rangle + \langle w^2 \rangle \right). \tag{5}$$

In this lab we wish to calculate the components of the Reynolds stress tensor and the turbulent kinetic energy for a fluid flow. A probe is used to make eight measurements of velocity components in the x, y, and z directions. The measurements are shown in the table below.

Table 1: Turbulence probe measurements in a fluid flow

Measurement	1	2	3	4	5	6	7	8
U [m s^{-1}]	1	2	4	3	5	1	2	6
V [m s^{-1}]	2	3	2	4	6	2	3	5
W [m s^{-1}]	4	3	1	2	2	3	2	5

We will use these measurements to calculate components of the Reynolds stress and the turbulent kinetic energy. We can assume that the flow experiment is repeated eight times under identical conditions and that the flow measurements are made at a specific and consistent location and time for each experiment. As a result, all statistical means that are calculated are *ensemble averages*.

6 Python Script

Copy and paste the following code in the `IDE` environment. Note that you must have installed the `numpy` package for this code to work.

```
import random
import sys
import os
import numpy

#Using arrays define instantaneous velocity in the x, y, and z directions [m s^-1]
U=[1, 2, 4, 3, 5, 1, 2, 6]
```

```
V=[2, 3, 2, 4, 6, 2, 3, 5]
W=[4, 3, 1, 2, 2, 3, 2, 5]

#Print newline and then the results
print("\n")
print("U=",U)
print("V=",V)
print("W=",W)

#Calculate the ensemble mean for each instantaneous velocity measurement [m s^-1]
Umean=numpy.mean(U)
Vmean=numpy.mean(V)
Wmean=numpy.mean(W)

print("\n")
print("Umean=",Umean)
print("Vmean=",Vmean)
print("Wmean=",Wmean)

#Calculate turbulent velocity fluctuations [m s^-1]
u=U-Umean
v=V-Vmean
w=W-Wmean

print("\n")
print("u=",u)
print("v=",v)
print("w=",w)

#Calculate the square of the turbulent velocity fluctuation [m^2 s^-2]
u2=numpy.multiply(u,u)
v2=numpy.multiply(v,v)
w2=numpy.multiply(w,w)

print("\n")
print("u2=",u2)
print("v2=",v2)
print("w2=",w2)

#Calculate variances of turbulent velocity fluctuations [m^2 s^-2]
u2mean=numpy.mean(u2)
v2mean=numpy.mean(v2)
w2mean=numpy.mean(w2)

print("\n")
print("u2mean=",u2mean)
```

```
print("v2mean=",v2mean)
print("w2mean=",w2mean)

#Calculate the turbulent kinetic energy [m^2 s^-2]
k=0.5*(u2mean+v2mean+w2mean)

print("\n")
print("k=",k)

#Calculate the products of velocity fluctuations [m^2 s^-2]
uv=numpy.multiply(u,v)
uw=numpy.multiply(u,w)
vw=numpy.multiply(v,w)

print("\n")
print("uv=",uv)
print("uw=",uw)
print("vw=",vw)

#Calculate the mean for products of velocity fluctuations [m^2 s^-2]
uvmean=numpy.mean(uv)
uwmean=numpy.mean(uw)
vwmean=numpy.mean(vw)

print("\n")
print("uvmean=",uvmean)
print("uwmean=",uwmean)
print("vwmean=",vwmean)

#Create the Reynolds Stress matrix
ReynoldsStress=[[u2mean, uvmean, uwmean],\
                [uvmean, v2mean, vwmean],[uwmean, vwmean, w2mean]]

print("\n")
print("Reynolds Stress Tensor=", ReynoldsStress)
print("ReynoldsStress[0][2]=", ReynoldsStress[0][2])
```

This script utilizes the functionality of the `numpy` package to perform the calculation. Note that each command like `U=[]` defines an array or vector. The command `numpy.mean()` calculates the mean of the elements of an array. The command `numpy.multiply()` calculates multiplication of two arrays, element by element. After the calculation of the components of the Reynolds stress, a three by three matrix `ReynoldsStress` is defined to store the components of the Reynolds stress. Note that each element of this matrix can be accessed by specifying indices in `[][]` format. For instance `ReynoldsStress[0][2]` returns the element located on the zeroth row and the second column. In `Python` programming, the indices for arrays and matrices start from 0. The results of running this script should look like the following:

```
/Library/Frameworks/Python.framework/Versions/3.6/bin/python3.6 "/
Users/AmirAbbasAliabadi/Google Drive/U Guelph/Courses/ENGG6790/
Computer Labs/PythonProgramming/PythonProgramming"

U= [1, 2, 4, 3, 5, 1, 2, 6]
V= [2, 3, 2, 4, 6, 2, 3, 5]
W= [4, 3, 1, 2, 2, 3, 2, 5]

Umean= 3.0
Vmean= 3.375
Wmean= 2.75

u= [-2. -1.  1.  0.  2. -2. -1.  3.]
v= [-1.375 -0.375 -1.375  0.625  2.625 -1.375 -0.375  1.625]
w= [ 1.25  0.25 -1.75 -0.75 -0.75  0.25 -0.75  2.25]

u2= [ 4.  1.  1.  0.  4.  4.  1.  9.]
v2= [ 1.890625  0.140625  1.890625  0.390625  6.890625  1.890625  0.140625
  2.640625]
w2= [ 1.5625  0.0625  3.0625  0.5625  0.5625  0.0625  0.5625  5.0625]

u2mean= 3.0
v2mean= 1.984375
w2mean= 1.4375

k= 3.2109375

uv= [ 2.75   0.375 -1.375  0.     5.25   2.75   0.375  4.875]
uw= [-2.5  -0.25 -1.75 -0.   -1.5  -0.5   0.75  6.75]
vw= [-1.71875 -0.09375  2.40625 -0.46875 -1.96875 -0.34375  0.28125  3.65625]

uvmean= 1.875
uwmean= 0.125
vwmean= 0.21875

Reynolds Stress Tensor= [[3.0, 1.875, 0.125], [1.875, 1.984375, 0.21875], ...
[0.125, 0.21875, 1.4375]]
ReynoldsStress[0][2]= 0.125

Process finished with exit code 0
```

ENGG*6790: Theory and Applications of Turbulence

One-point Turbulent Statistics

Amir A. Aliabadi

November 11, 2017

1 Introduction

In lectures we discuss the significance of the Reynolds stress tensor, which describes various statistics such as normal stresses of all components of momentum as well as pair-wise components of the shear stress:

$$\begin{bmatrix} \langle u_1^2 \rangle & \langle u_1 u_2 \rangle & \langle u_1 u_3 \rangle \\ \langle u_2 u_1 \rangle & \langle u_2^2 \rangle & \langle u_2 u_3 \rangle \\ \langle u_3 u_1 \rangle & \langle u_3 u_2 \rangle & \langle u_3^2 \rangle \end{bmatrix}$$

The entries of the Reynolds stress tensor can also be described as one-point turbulent statistics of the flow because they each report a turbulent quantity at one point in the domain. One-point turbulent statistics in a flow can be calculated if time series of a measurement (or a number of measurements such as momentum, temperature, concentration, etc.) is available at high frequency.

One-point statistics can be also described by alternative terminology. Consider a property in the flow is measured at high frequency, such as velocity in the $x-$direction,

$$U = \langle U \rangle + u \tag{1}$$

The normal stress $\langle u^2 \rangle$ can also be called the variance of U. After all, if U is a random variable, its variance in statistics gives the same mathematical quantity as the normal stress. The variance of random variable U in statistics is shown with

$$\text{var}(U) = \sigma_U^2 = \langle u^2 \rangle \tag{2}$$

Some times in turbulence studies the variance is normalized by the square of the mean quantity of the variable, for which the variance is being calculated. For instance, in the same example, the variance can be normalized by $\langle U \rangle^2$ so that the following quantity is reported,

$$\frac{\mathrm{var}(U)}{\langle U\rangle^2} = \frac{\sigma_U^2}{\langle U\rangle^2} = \frac{\langle u^2\rangle}{\langle U\rangle^2} \tag{3}$$

The shear stress $\langle uv\rangle$ can also be called the covariance of U and V. After all, if U and V are random variables, their covariance in statistics gives the same mathematical quantity as the shear stress. The covariance of two random variables U and V in statistics is shown with

$$\mathrm{cov}(U, V) = \langle uv\rangle \tag{4}$$

In turbulence studies the covariance is also normalized by a mean quantity relevant to the study. For instance, in the same example, the covariance can be normalized by one of the $\langle U\rangle^2$, $\langle V\rangle^2$, $|\langle U\rangle\langle V\rangle|$, or even $\langle U\rangle^2 + \langle V\rangle^2$, so that one of the following quantities may be reported,

$$\frac{\mathrm{cov}(U, V)}{\langle U\rangle^2} = \frac{\langle uv\rangle}{\langle U\rangle^2} \tag{5}$$

$$\frac{\mathrm{cov}(U, V)}{\langle V\rangle^2} = \frac{\langle uv\rangle}{\langle V\rangle^2} \tag{6}$$

$$\frac{\mathrm{cov}(U, V)}{|\langle U\rangle\langle V\rangle|} = \frac{\langle uv\rangle}{|\langle U\rangle\langle V\rangle|} \tag{7}$$

$$\frac{\mathrm{cov}(U, V)}{\langle U\rangle^2 + \langle V\rangle^2} = \frac{\langle uv\rangle}{\langle U\rangle^2 + \langle V\rangle^2} \tag{8}$$

The choice of the normalization statistic is somewhat arbitrary given the context of the study. For instance, in meteorology, the statistic $\langle U\rangle^2 + \langle V\rangle^2$ is used, which gives the average wind speed in the horizontal direction, with $x-$direction and $y-$direction being horizontal and the $z-$direction pointing normal to the earth surface with the positive sign upward.

It is also possible to calculate variance and covariance for any number or combination of variables. For instance, if $T = \langle T\rangle + t$ is a random variable representing temperature, it is possible to calculate $\mathrm{var}(T)$, $\mathrm{cov}(W, T)$, $\mathrm{cov}(U, T)$, $\mathrm{cov}(U, W)$, etc, with the appropriate normalization statistics.

In this lab, we wish to calculate turbulent statistics for airflow and temperature using a dataset from a micro-climate study on the campus of the University of Guelph. The campaign was conducted from August 13, 2017 to August 25, 2017. Part of the study involved installing a sonic anemometer on the roof of the Rozhanski Hall. The anemometer measured air velocity in horizontal components U, V and vertical component W in units of $\mathrm{m\,s^{-1}}$. Note that V was air velocity along canyon axis, while U was air velocity cross canyon axis. It also measured air sonic temperature T in units of K. The measurement was conducted at a sampling frequency of 4 Hz. Figure below shows the campaign site and the roof anemometer.

Figure 1: University of Guelph micro-climate campaign in August 2017: campaign site (top) and sonic anemometer installation on the roof of Rozhanski Hall (bottom)

We wish to compute the following statistics at time intervals of 30 min. The statistics involve mean quantities of variables, variances, and covariances. Table below shows the statistics to be calculated and the normalization statistics to be used. For temperature normalization, the maximum half-hourly variation in temperature can be assumed, i.e. $\Delta T = T_{max} - T_{min}$

Table 1: Turbulent statistics to be calculated and normalization statistics.

Statistic	Description	Normalization	Units
$\text{avg}(U) = \langle U \rangle$	Mean velocity (x)	-	[m s^{-1}]
$\text{avg}(V) = \langle V \rangle$	Mean velocity (y)	-	[m s^{-1}]
$\sqrt{\langle U \rangle^2 + \langle V \rangle^2}$	Mean horizontal speed	-	[m s^{-1}]
$\text{avg}(W) = \langle W \rangle$	Mean velocity (z)	-	[m s^{-1}]
$\text{avg}(T) = \langle T \rangle$	Mean temperature	-	[K]
$\text{var}(U) = \sigma_U^2 = \langle u^2 \rangle$	Velocity variance (x)	$\langle U \rangle^2 + \langle V \rangle^2$	[m^2 s^{-2}]
$\text{var}(V) = \sigma_V^2 = \langle v^2 \rangle$	Velocity variance (y)	$\langle U \rangle^2 + \langle V \rangle^2$	[m^2 s^{-2}]
$\text{var}(W) = \sigma_W^2 = \langle w^2 \rangle$	Velocity variance (z)	$\langle U \rangle^2 + \langle V \rangle^2$	[m^2 s^{-2}]
$\text{var}(T) = \sigma_T^2 = \langle t^2 \rangle$	Temperature variance	ΔT^2	[K^2]
$k = \frac{1}{2}\left(\langle u^2 \rangle + \langle v^2 \rangle + \langle w^2 \rangle\right)$	Turbulent kinetic energy	$\langle U \rangle^2 + \langle V \rangle^2$	[m^2 s^{-2}]
$\text{cov}(U, V) = \langle uv \rangle$	Turbulent kinematic mass flux (x, y)	$\langle U \rangle^2 + \langle V \rangle^2$	[m^2 s^{-2}]
$\text{cov}(U, W) = \langle uw \rangle$	Turbulent kinematic mass flux (x, z)	$\langle U \rangle^2 + \langle V \rangle^2$	[m^2 s^{-2}]
$\text{cov}(V, W) = \langle vw \rangle$	Turbulent kinematic mass flux (y, z)	$\langle U \rangle^2 + \langle V \rangle^2$	[m^2 s^{-2}]
$\text{cov}(U, T) = \langle ut \rangle$	Turbulent kinematic heat flux (x)	$\sqrt{\langle U \rangle^2 + \langle V \rangle^2}\Delta T$	[K m s^{-1}]
$\text{cov}(V, T) = \langle vt \rangle$	Turbulent kinematic heat flux (y)	$\sqrt{\langle U \rangle^2 + \langle V \rangle^2}\Delta T$	[K m s^{-1}]
$\text{cov}(W, T) = \langle wt \rangle$	Turbulent kinematic heat flux (z)	$\sqrt{\langle U \rangle^2 + \langle V \rangle^2}\Delta T$	[K m s^{-1}]

2 Python Script

We perform the calculation of turbulent statistics in one script and the plotting of the results in another script. Complete the following script for calculations. In this script, we define a file name as `"Roof4Hz.txt"` to be read by the program. We subsequently define another file name as `"Roof4HzOnePointStatistics.txt"` to write the result of our calculations. Note that the input file name has many columns of data, not all of which need to be read by the program. Use of the `usecols=[...]` argument in the `numpy.loadtxt()` function allows us to only read the columns that we need.

An important requirement for calculating turbulent statistics is that the time series data must be detrended for each time interval, in which we desire to calculate the turbulent statistics. The idea behind detrending is that many environmental data show linear trends that must be eliminated (or subracted) from the data before calculating turbulent statistics. These linear trends really do not contribute to turbulence and are slow background variations (in this case diurnal variations). If the linear trend is not removed from the data, one may report spuriously high turbulent statistics. Figure below shows a trended and a detrended time series.

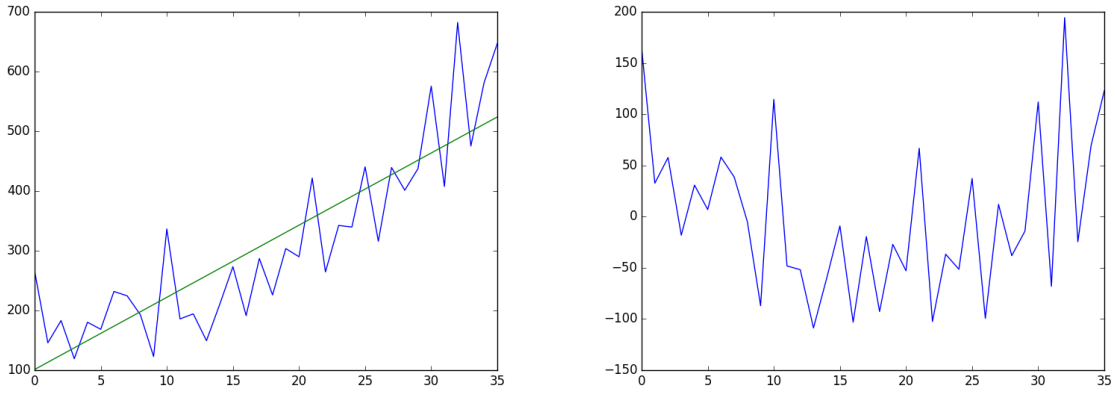


Figure 2: A trended time series (left) and a detrended time series (right); turbulent statistics must always be calculated after removing a trend from a time series, or else spurious statistics may be reported.

Detrending of the time series is achieved by first fitting a first order polynomial to the time series using the `numpy.polyfit()` function. Subsequently, a model is built based on this fit using the `numpy.polyval()` function. Finally, the model, which is really only a line, is subtracted from the original time series to give the detrended time series. All the subsequent turbulent statistics are calculated from the detrended time series. Of course, the detrended time series contains all the turbulent fluctuations.

To calculate the variances and covariances, we have used the `numpy.cov()` function. This function is extremely useful. It takes two vectors as arguments and returns a 2 by 2 matrix. The main diagonal elements of the matrix are the variances of the two vectors provided, and the off-diagonal elements are covariances. The elements of the matrix can be accessed and assigned to appropriate variable. Element `[0,0]` is the variance of the first vector argument, element `[1,1]` is the variance of the second vector argument, and element `[0,1]` (the same as element `[1,0]`) is the covariance of the two vectors.

Note that the iterations move forward for each 30 min window of data. In each iteration, the program calculates the statistics, stores the statistics in vectors, and moves on to the next 30 min window.

Finally, the data are written to a file with an appropriate header. Note that using `#`, or comment line, in a text file results in useful header information that will not really be read by the `numpy.loadtxt()` function. It is recommended to describe in the text file exactly what information each column holds and the units associated with it. It is also useful to number the columns so subsequently files can be read conveniently.

```
#Calculate one-point turbulent statistics
import random
import sys
import os
```

```
import numpy
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import datetime

#Define averaging period in number of data points: minutes * seconds * samples
AverageSample=30*60*4

#Define file names
fileName = "Roof4Hz.txt"

outputFileNameOnePointStatistics="Roof4HzOnePointStatistics.txt"

#Load all data in a matrix
data4Hz = numpy.loadtxt(fileName, usecols=[0,1,2,3,4,5,8,9,10,11])

year4Hz=data4Hz[:,0]
month4Hz=data4Hz[:,1]
day4Hz=data4Hz[:,2]
timeHr4Hz=data4Hz[:,3]
timeMin4Hz=data4Hz[:,4]
timeSec4Hz=data4Hz[:,5]
U4Hz=data4Hz[:,6]
V4Hz=data4Hz[:,7]
W4Hz=data4Hz[:,8]
TSonic4Hz=data4Hz[:,9]

N4Hz=numpy.size(year4Hz)

#Calculate the number of samples and then detrend data
NSample=int(N4Hz/AverageSample)

#Define statistics, S is the wind speed in the horizontal direction
yearavg=numpy.zeros((NSample,1))
monthavg=...
dayavg=...
timeHravg=...
timeMinavg=...
Uavg=...
Vavg=...
Savg=...
Wavg=...
TSonicavg=...
Uvar=...
Vvar=...
Wvar=...
```

```
TSonicvar=...
k=...
UVcov=...
UWcov=...
VWcov=...
UTSoniccov=...
VTSoniccov=...
WTSoniccov=...

for i in range(0,NSample):
    #Calculate year, month, day, hour, and minute for each sample
    yearavg[i] = numpy.mean(year4Hz[i*AverageSample:(i+1)*AverageSample])
    monthavg[i] = numpy.mean(month4Hz[i*AverageSample:(i+1)*AverageSample])
    dayavg[i] = ...
    timeHravg[i] = ...
    timeMinavg[i] = numpy.mean(timeMin4Hz[i*AverageSample:(i+1)*AverageSample])+1

    #Calculate averages
    Uavg[i] = numpy.mean(U4Hz[i*AverageSample:(i+1)*AverageSample])
    Vavg[i] = ...
    Wavg[i] = ...
    TSonicavg[i] = ...
    Savg[i] = numpy.mean(numpy.sqrt(U4Hz[i*AverageSample:(i+1)*AverageSample] ** 2 + \
                            V4Hz[i*AverageSample:(i+1)*AverageSample] ** 2))

    #Define a vector for the number of data points in each sample
    x = [j for j in range(0, AverageSample)]
    #Detrend each sample, i.e. remove a straight line fit from the sample
    U = U4Hz[i*AverageSample:(i+1)*AverageSample]
    Umodel = numpy.polyfit(x,U,1)
    Utrend = numpy.polyval(Umodel,x)
    Udetrended = U - Utrend
    V = ...
    Vmodel = ...
    Vtrend = ...
    Vdetrended = ...
    W = ...
    Wmodel = ...
    Wtrend = ...
    Wdetrended = ...
    TSonic = ...
    TSonicmodel = ...
    TSonictrend = ...
    TSonicdetrended = ...

    #Calculate variances, and covariances
```

```
    UVCovMatrix = numpy.cov(Udetrended, Vdetrended)
    UWCovMatrix = numpy.cov(Udetrended, Wdetrended)
    VWCovMatrix = ...
    UTSonicCovMatrix = ...
    VTSonicCovMatrix = ...
    WTSonicCovMatrix = ...

    Uvar[i] = UVCovMatrix[0,0]
    Vvar[i] = UVCovMatrix[1,1]
    Wvar[i] = ...
    TSonicvar[i] = ...
    k[i] = ...

    UVcov[i] = UVCovMatrix[0,1]
    UWcov[i] = ...
    VWcov[i] = ...
    UTSoniccov[i] = ...
    VTSoniccov[i] = ...
    WTSoniccov[i] = ...

#Write data to file
outputFile = open(outputFileNameOnePointStatistics, "w")
outputFile.write("#Times in Local Daylight Time \n")
outputFile.write("#0:Year \t 1:Month \t 2:Day \t 3:Hour \t 4:Minute \t \
    5:Uavg (m s-1) \t 6:Vavg (m s-1) \t 7:Savg (m s-1) \t 8:Wavg (m s-1) \t \
    9:TSonicavg (K) \t 10:Uvar (m2 s-2) \t 11:Vvar (m2 s-2) \t \
    12:Wvar (m2 s-2) \t 13:TSonicvar (K2) \t 14:k (m2 s-2) \t \
    15:UVcov (m2 s-2) \t 16:UWcov (m2 s-2) \t 17:VWcov (m2 s-2) \t \
    18:UTSoniccov (Km s-1) \t 19:VTSoniccov (Km s-1) \t 20:WTSoniccov (Km s-1) \n")

for i in range(0,NSample):
    outputFile.write("%i \t %i \t %i \t %i \t %i \t \
    %f \t %f \t %f \t %f \t %f \t \
    %f \t %f \t %f \t %f \t %f \t \
    %f \t %f \t %f \
    %f \t %f \t %f \n" \
    % (yearavg[i], ...))
outputFile.close()
```

After running this script, we can generate a text file with all the turbulent statistics. The next step is to run a new script for reading the results and plotting the turbulent statistics. This script is given to you as `PlotResults`. In this script we use some new libraries that enable plotting information versus date and time. These libraries are `matplotlib.dates` and `datetime`.

The script first reads the results text file and assigns the results to specific vectors. Next, it finds the maximum variation in half-hourly temperature. This quantity is needed for normalizing

variances and covariances that involve temperature. The next step is to create a vector to contain the time for each measurement in seconds. This is performed by giving the half-hourly year, month, day, hour, and minute to the function `datetime.datetime().timestap()`. And finally there is another command that allows creating a vector to contain date and time in the `YYYY-MM-HH-mm-ss` format.

For each turbulent statistic, two plots are generated. The first plot shows the time series for the quantity of interest. The second plot shows the diurnal variation of the quantity of interest. The diurnal plot overlays the quantity of interest over many days as a function of hour in the day from 0 to 23 of the Local Daylight Time zone. This helps identify which quantities exhibit a strong diurnal variation. To plot all figures simultaneously, the function `fig.show()` is used for each figure and finally the function `plt.show()` is used at the end of the script.

After successfully running the second script, the following figures should be obtained. Try to answer the following questions.

- Which one of the mean velocity components or the horizontal wind speed show a significant diurnal cycle?
- Does the mean temperature show a significant diurnal cycle? How do you interpret this physically?
- On average, the variances of velocity components represent what fraction of the square mean horizontal wind speed? 0.1%, 1%, 10% or 100%?
- Which one of the variances exhibit a significant diurnal cycle? How do you interpret this physically?
- On average, the kinetic energy represents what fraction of the square mean horizontal wind speed? 0.1%, 1%, 10% or 100%?
- Considering the turbulent kinematic heat fluxes, which fluxes exhibit both significantly positive and significantly negative values as a function of diurnal cycle? How do you interpret this physically?
- Considering the turbulent kinematic heat fluxes, which fluxes exhibit only a significantly positive or only a significantly negative value as a function of diurnal cycle? How do you interpret this physically?

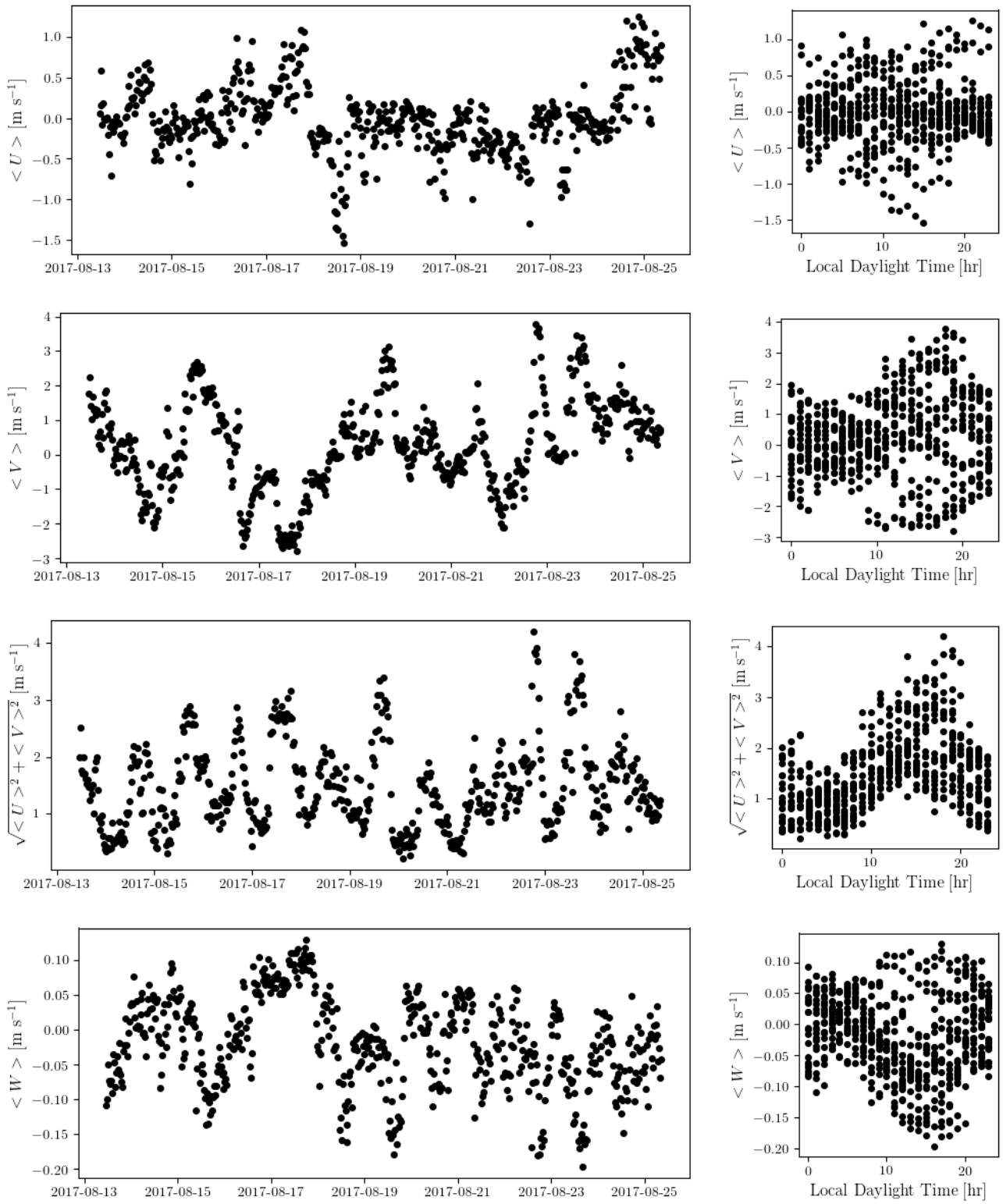


Figure 3: Mean velocities and mean wind speed in the horizontal direction.

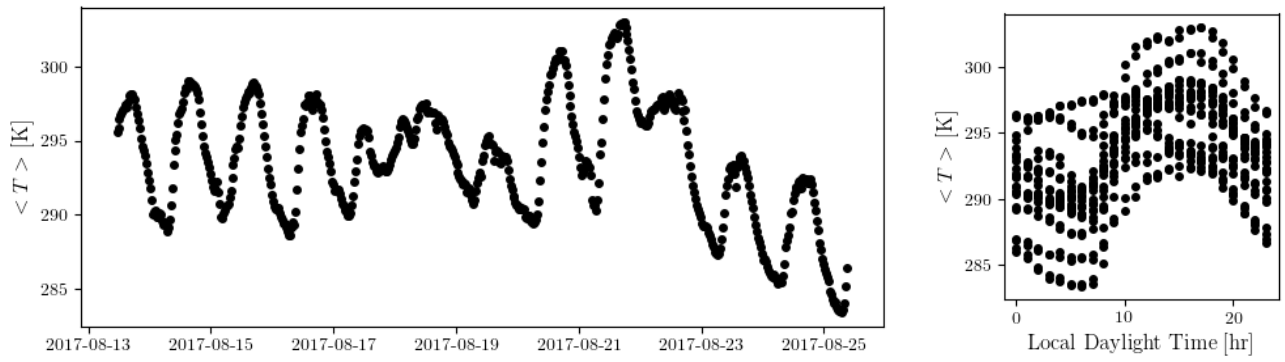


Figure 4: Mean temperature.

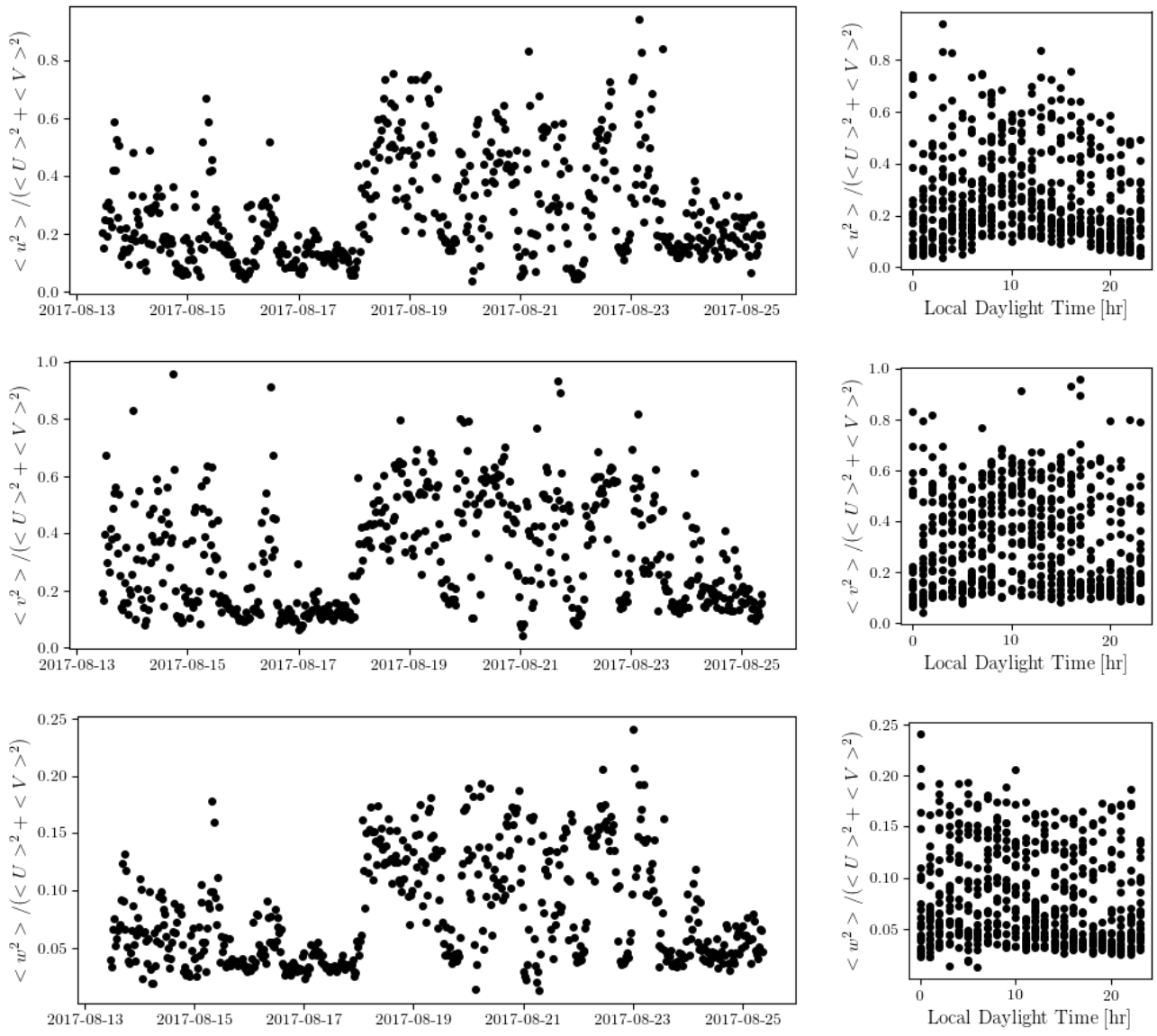


Figure 5: Normalized variances of velocity components.

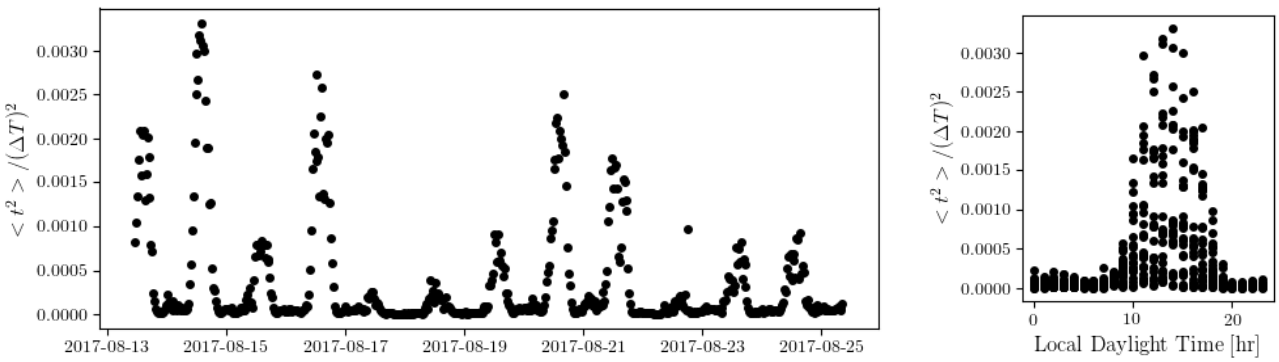


Figure 6: Normalized variance of temperature.

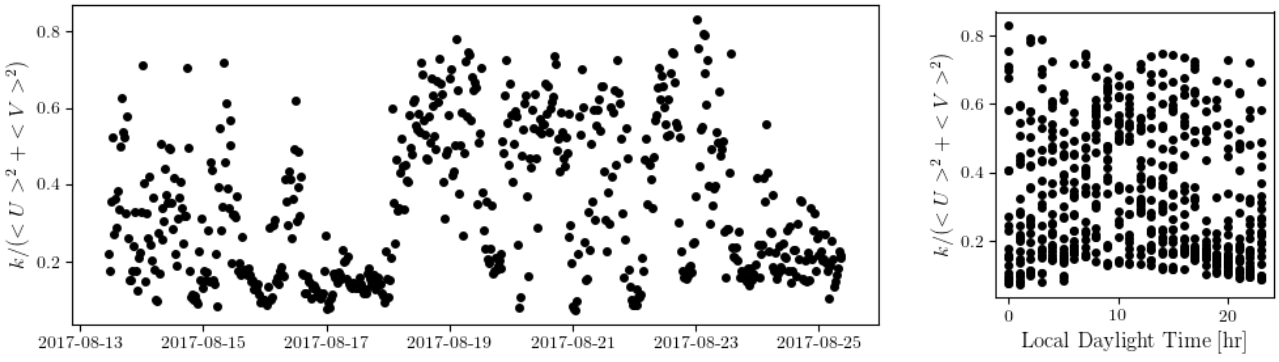


Figure 7: Normalized turbulent kinetic energy.

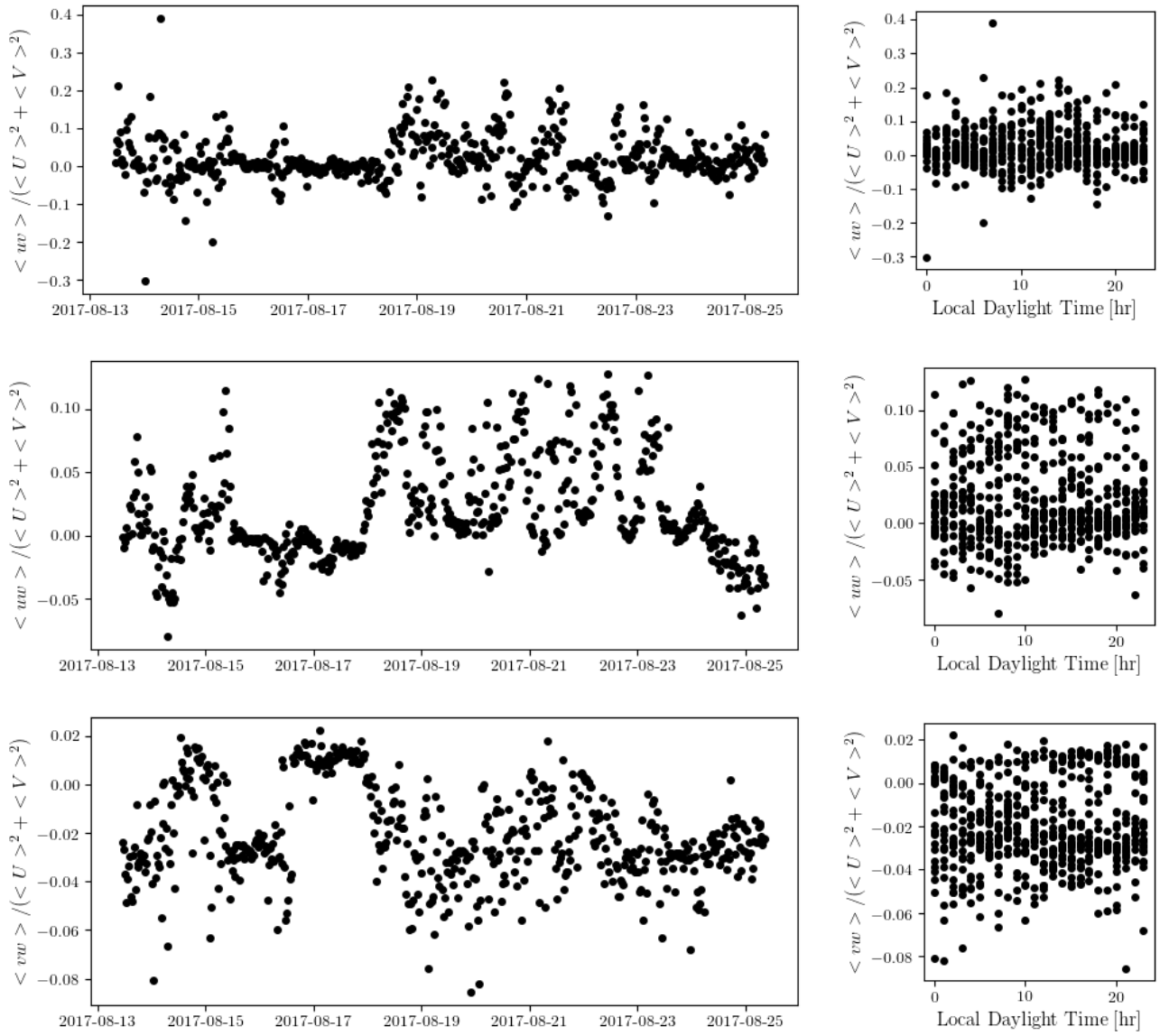


Figure 8: Normalized turbulent kinematic mass fluxes.

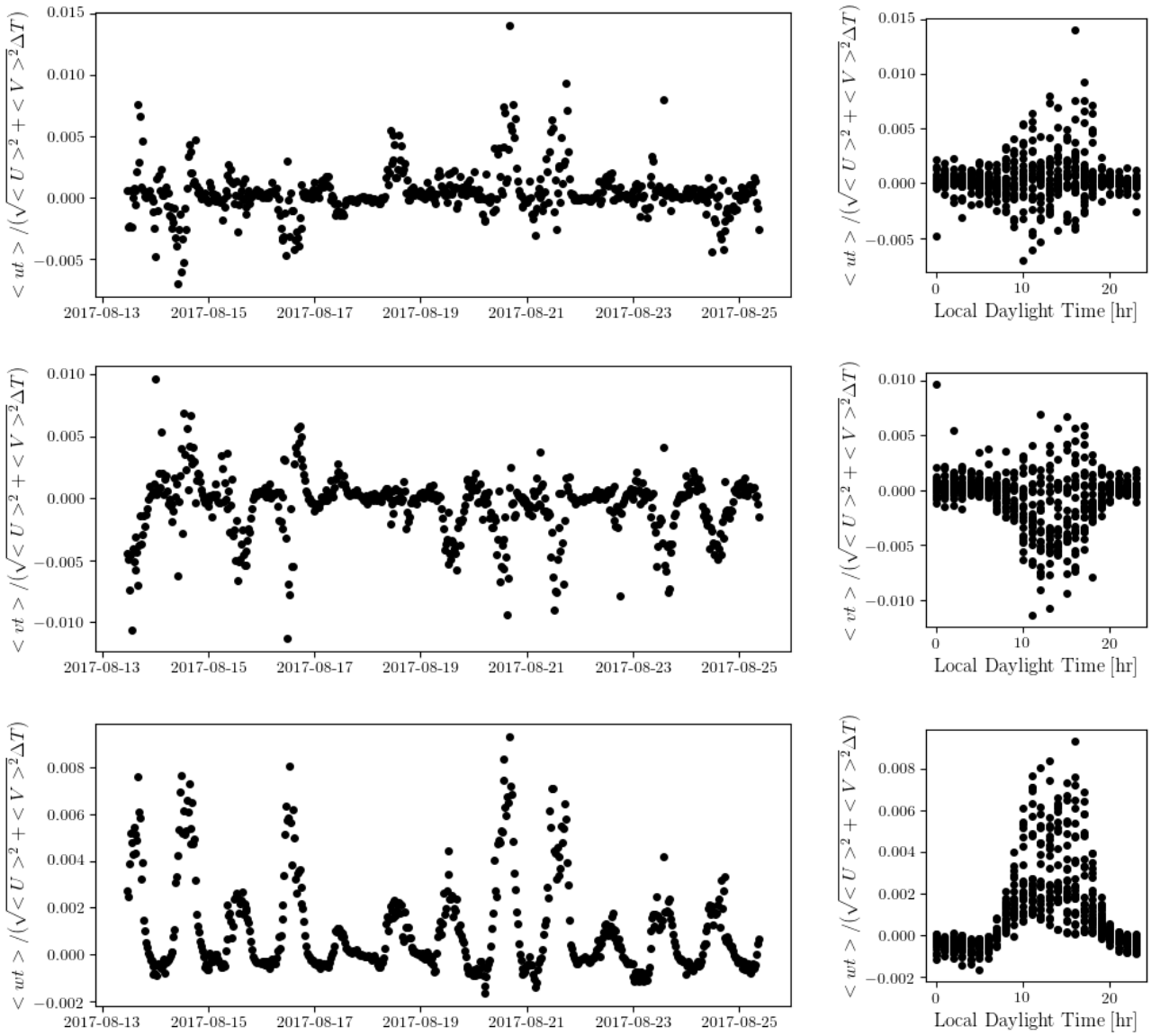


Figure 9: Normalized turbulent kinematic heat fluxes.

ENGG*6790: Theory and Applications of Turbulence

Round Jet Similarity

Amir A. Aliabadi

January 31, 2018

1 Introduction

Flow dynamics of a self-similar round jet is discussed in lectures. The jet is produced by ejecting flow out of a nozzle with the following parameters, with the parameters explained in the figure below.

$$U_J = 1\,\mathrm{m\ s^{-1}} \tag{1}$$
$$d = 0.01\,\mathrm{m} \tag{2}$$
$$x_0 = 4d = 0.04\,\mathrm{m} \tag{3}$$
$$B = 5.9 \tag{4}$$

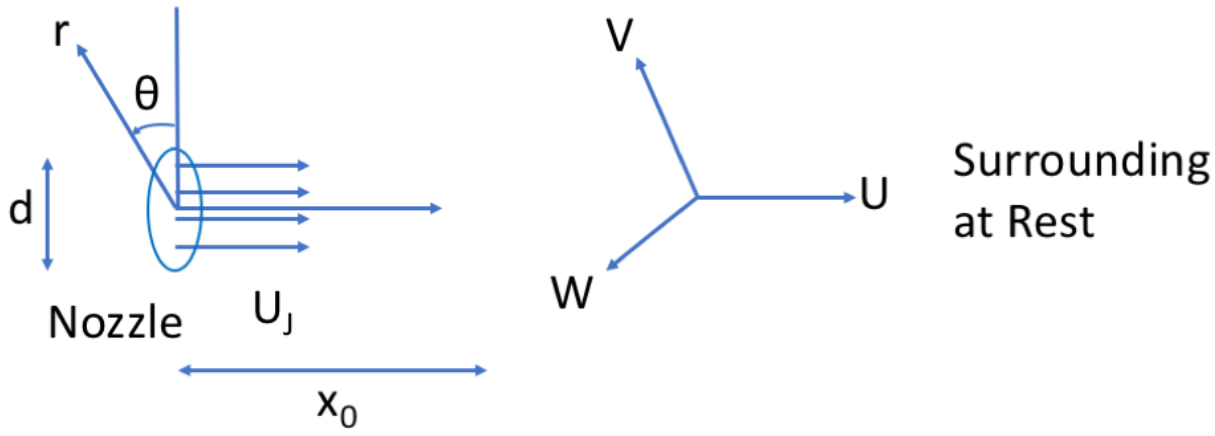


Figure 1: Schematic of a round jet nozzle with polar cylindrical and Cartesian coordinate systems.

It has been shown in the lectures that the mean axial velocity on the centre-line can be calculated using the following formulat. The constants B and x_0 have been fitted experimentally.

$$\frac{U_0(x)}{U_J} = \frac{B}{(x - x_0)/d} \tag{5}$$

Other turbulent statistics in the flow have been fitted experimentally using the following formulat and table. These provide even functions with respect to $\eta \equiv r/(x - x_0)$. The multiplication of the polynomial and exponential function provides an excellent fit over the range in which data were taken.

$$p(\eta) = \left[C_0 + C_2\eta^2 + C_4\eta^4 + ...\right] \exp\left(-A\eta^2\right). \tag{6}$$

Table 1: Constants to determine turbulent properties of a self-similar round jet.

$p(\eta)$	C_0	C_2	C_4	C_6	A
$\langle U\rangle/U_0(x)$	1.0	–1.925	0.0	0.0	63
$\langle u^2\rangle/U_0^2(x)$	7.778e–2	2.79e1	–2.02e3	4.3e5	257
$\langle v^2\rangle/U_0^2(x)$	5.457e–2	0.355	–4.298e1	0.0	89
$\langle w^2\rangle/U_0^2(x)$	5.78e–2	–1.71	2.73e–1	0.0	42
$\langle uv\rangle/U_0^2(x)$	4.375e–1	–3.931e1	1.55e2	1.342e4	90
$\epsilon/\left[U_0^3(x)/(x - x_0)\right]$	0.3549	11.99	–1635	43470	201

In this lab we desire to calculate the mean axial centre-line velocity $U_0(x)$ as a function of axial distance x. We also wish to calculate various other quantities as a function of radial distance r at selected axial distances. These are the mean axial velocity $\langle U\rangle$, turbulent fluctuating velocity variances $\langle u^2\rangle$, $\langle v^2\rangle$, $\langle w^2\rangle$, shear stress $\langle uv\rangle$, turbulent kinetic energy k, and dissipation rate ϵ.

2 Python Script

Copy and paste the following code in the `IDE` environment. Some lines of the code are shown with `...`, which require you to complete them as coding exercise. Note that you must install the `matplotlib` package for this lab to enable plotting using `Python`.

```
import random
import sys
import os
import numpy
import matplotlib.pyplot as plt
```

`Python` provides the flexibility to define short form commands for faster programming. The command `import ... as ...` enables you to do this. For instance the command `plt` is defined here for plotting. First the jet parameters are defined:

```
#Define nozzle constants
d=0.01          #[m]
```

```
x0=4*d          #[m]
B=5.9

#Define nozzle exit velocity [m s^-1]
UJ=1
```

The next step is to discretize the axial domain x by increments `dx`. The command `numpy.linspace()` creates an array given a minimum value, maximum value, and the number of elements. Here it is used to create the `x` array. The command `len()` fines the number of elements in an array. It is used here to store the number of elements in the `x` array.

```
#Define x axis from x0+10*dx to xmax with dx increments
dx=0.01         #[m]
xmin=x0+10*dx   #[m]
xmax=1          #[m]
x=numpy.linspace(xmin, xmax, (xmax-xmin+dx)/dx)
nx=len(x)
```

We discretize the radial domain r in slightly different way, not by specifying increments `dr` but by specifying the number of elements.

```
#Define r axis from 0 to 0.3 with 0.01 [m] increments
r=numpy.linspace(0, 0.3, 31)
nr=len(r)
```

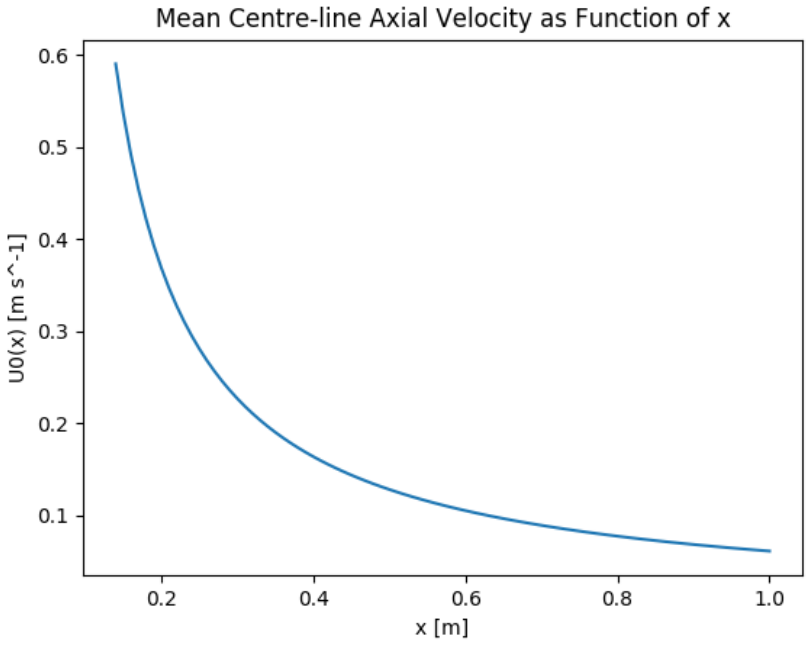
`Python` can perform vector calculations in a very concise syntax. For instance we can calculate the centre-line mean axial velocity using the following line.

```
#Calculate centre-line mean axial velocity as a function of x [m s^-1]
U0=UJ*B/((x-x0)/d)
```

We can plot the centre-line mean axial velocity as a function of axial distance using the `plt.plot()` command. The command takes two vectors of equal size and `x` and `y` vectors and creates a plot. It is possible to label the plot using the `plt.xlabel()` and `plt.ylabel()` commands. It is also possible to create a title for the plot using the `plt.title()` command. After creating the plot, it will remain in the background. The plot can be shown in the foreground using the `plt.show()` command. After running the code it is possible to obtain the following plot.

```
#Plot the mean centre-line axial velocity versus x
plt.plot(x,U0)
plt.xlabel('x [m]')
plt.ylabel('U0(x) [m s^-1]')
plt.title('Mean Centre-line Axial Velocity as Function of x')
plt.show()
```

Other solution variables must be obtained over the entire $x - r$ domain so a two dimensional variable or a matrix is necessary to define and initialize. The mean axial velocity as a function x



and r can be defined and initialized using the `numpy.zeros((nx,nr))` command. This command creates a two dimensional matrix with `nx` rows, each representing an axial location and `nr` columns, each representing a radial location.

```
#Define and initialize a mean axial velocity [m s^-1]
Umean=numpy.zeros((nx,nr))
```

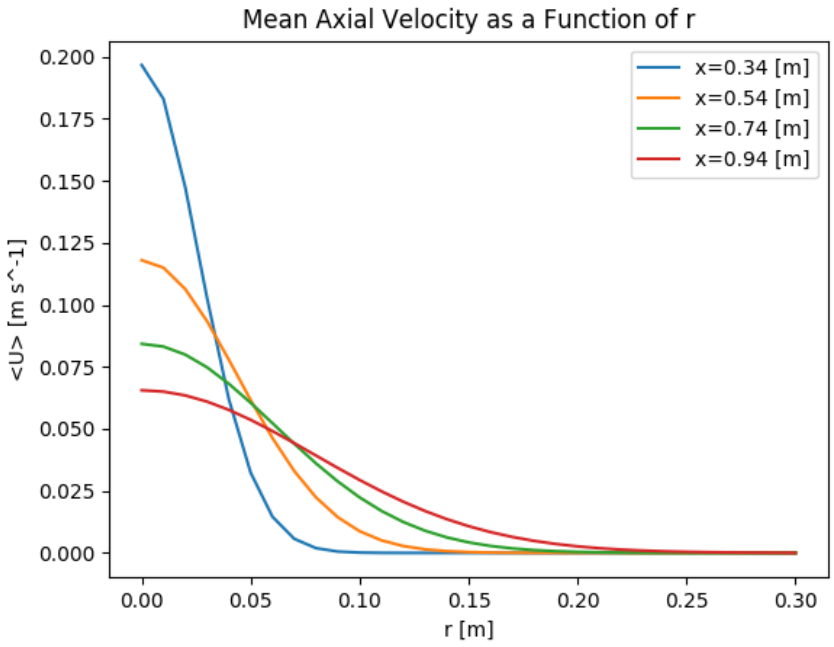
The mean axial velocity over the $x - r$ domain can be calculated by looking up constants `C0`, `C2`, `C4`, `C6`, and `A`, and iterating two nested `for` loops over `i` and `j` indices representing `x` and `r` positions. Note that the exponential operation is specified using `**` in `Python` as opposed to `^` in other programming languages. The elements of an array can be accessed by specifying indices in the `[][]` format. The exponential function can be deployed using the `numpy.exp()` syntax.

```
#Calculate mean axial velocity over the x-r domain
C0=1.0
C2=-1.925
C4=0
C6=0
A=63

for i in range(0, nx-1):
    for j in range (0, nr-1):
        eta=r[j]/(x[i]-x0)
        Umean[i][j]=U0[i]*(C0+C2*eta**2+C4*eta**4+C6*eta**6)*numpy.exp(-A*eta**2)
```

Multiple figures can be plotted on the same graph using the following code. A label should be defined by adding a few strings together using the `+` operator. The `str()` command takes a

numerical variable and returns it as a string. The command `plt.legend()` adds the legends to a plot. The plot is automatically colour coded for each legend. After running the code the following plot can be obtained.

```
#Plot the mean axial velocity versus r
plt.plot(r,Umean[20][:],label='x='+str(x[20])+' [m]')
plt.plot(r,Umean[40][:],label='x='+str(x[40])+' [m]')
plt.plot(r,Umean[60][:],label='x='+str(x[60])+' [m]')
plt.plot(r,Umean[80][:],label='x='+str(x[80])+' [m]')
plt.xlabel('r [m]')
plt.ylabel('<U> [m s^-1]')
plt.title('Mean Axial Velocity as a Function of r')
plt.legend()
plt.show()
```



The following code should be completed to obtained the rest of the variables for turbulent statistics. Note that you should look up the proper constants for each variable. In addition, the expression for the calculation of each variable should be appropriately adjusted.

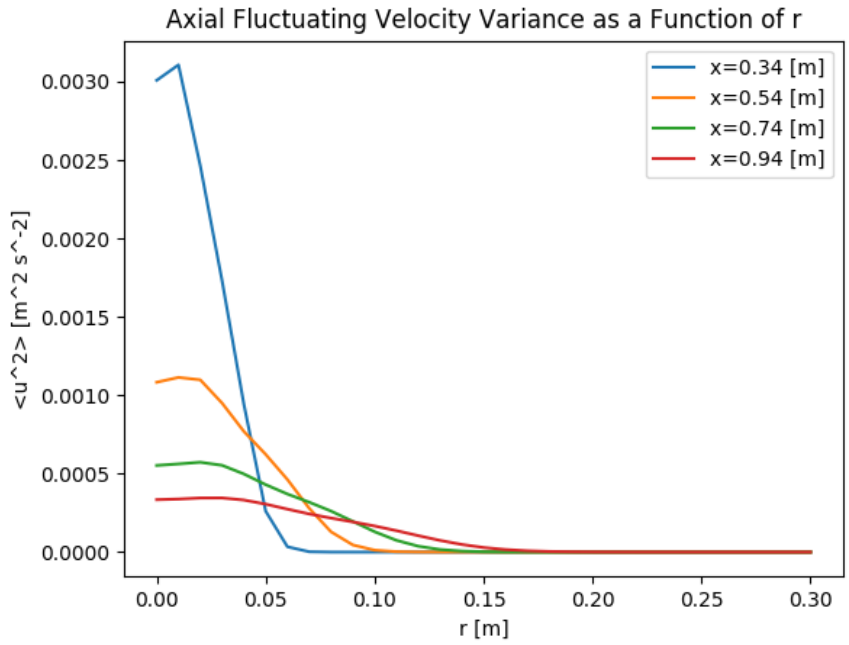
```
#Define and initialize axial fluctuating velocity variance [m^2 s^-2]
u2mean=...

#Calculate axial fluctuating velocity variance over the x-r domain
C0=...
C2=...
C4=...
C6=...
```

```
A=...

for i in range(0, nx-1):
    for j in range (0, nr-1):
        eta=r[j]/(x[i]-x0)
        u2mean[i][j]=U0[i]**2*\
                    (C0+C2*eta**2+C4*eta**4+C6*eta**6)*numpy.exp(-A*eta**2)

#Plot the axial fluctuating velocity variance versus r
plt.plot(...
plt.plot(...
plt.plot(...
plt.plot(...
plt.xlabel('r [m]')
plt.ylabel('<u^2> [m^2 s^-2]')
plt.title('Axial Fluctuating Velocity Variance as a Function of r')
plt.legend()
plt.show()

#Define and initialize radial fluctuating velocity variance [m^2 s^-2]
v2mean=...

#Calculate radial fluctuating velocity variance over the x-r domain
C0=...
C2=...
C4=...
C6=...
A=...

for i in range(0, nx-1):
    for j in range (0, nr-1):
        eta=r[j]/(x[i]-x0)
        v2mean[i][j]=...

#Plot the radial fluctuating velocity variance versus r
plt.plot(...
plt.plot(...
plt.plot(...
plt.plot(...
plt.xlabel('r [m]')
plt.ylabel('<v^2> [m^2 s^-2]')
plt.title('Radial Fluctuating Velocity Variance as a Function of r')
plt.legend()
plt.show()

#Define and initialize circumferential fluctuating velocity variance [m^2 s^-2]
```

```
w2mean=...

#Calculate circumferential fluctuating velocity variance over the x-r domain
C0=...
C2=...
C4=...
C6=...
A=...

for i in range(0, nx-1):
    for j in range (0, nr-1):
        eta=r[j]/(x[i]-x0)
        w2mean[i][j]=...

#Plot the circumferential fluctuating velocity variance versus r
plt.plot(...
plt.plot(...
plt.plot(...
plt.plot(...
plt.xlabel('r [m]')
plt.ylabel('<w^2> [m^2 s^-2]')
plt.title('Circumferential Fluctuating Velocity Variance as a Function of r')
plt.legend()
plt.show()

#Define and initialize shear stress [m^2 s^-2]
uvmean=...

#Calculate shear stress over the x-r domain
C0=...
C2=...
C4=...
C6=...
A=...

for i in range(0, nx-1):
    for j in range (0, nr-1):
        eta=r[j]/(x[i]-x0)
        uvmean[i][j]=...

#Plot the shear stress versus r
plt.plot(...
plt.plot(...
plt.plot(...
plt.plot(...
plt.xlabel('r [m]')
```

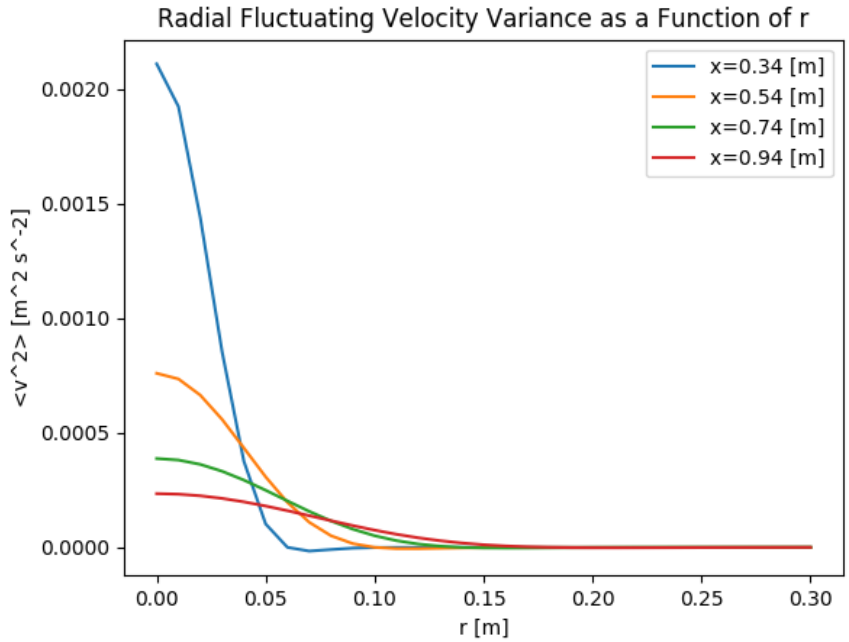
```
plt.ylabel('<uv> [m^2 s^-2]')
plt.title('Shear Stress as a Function of r')
plt.legend()
plt.show()

#Calculate the turbulent kinetic energy [m^2 s^-2]
k=0.5*(u2mean+v2mean+w2mean)

#Plot the turbulent kinetic energy versus r
plt.plot(...
plt.plot(...
plt.plot(...
plt.plot(...
plt.xlabel('r [m]')
plt.ylabel('k [m^2 s^-2]')
plt.title('Turbulent Kinetic Energy as a Function of r')
plt.legend()
plt.show()

#Define and initialize dissipation rate [m^2 s^-3]
epsilon=...

#Calculate dissipation rate over the x-r domain
C0=...
C2=...
C4=...
C6=...
A=...

for i in range(0, nx-1):
    for j in range (0, nr-1):
        eta=r[j]/(x[i]-x0)
        epsilon[i][j]=U0[i]**3/(x[i]-x0)*\
                      (C0+C2*eta**2+C4*eta**4+C6*eta**6)*numpy.exp(-A*eta**2)

#Plot the dissipation rate versus r
plt.plot(...
plt.plot(...
plt.plot(...
plt.plot(...
plt.xlabel('r [m]')
plt.ylabel('epsilon [m^2 s^-3]')
plt.title('Dissipation Rate as a Function of r')
plt.legend()
plt.show()
```

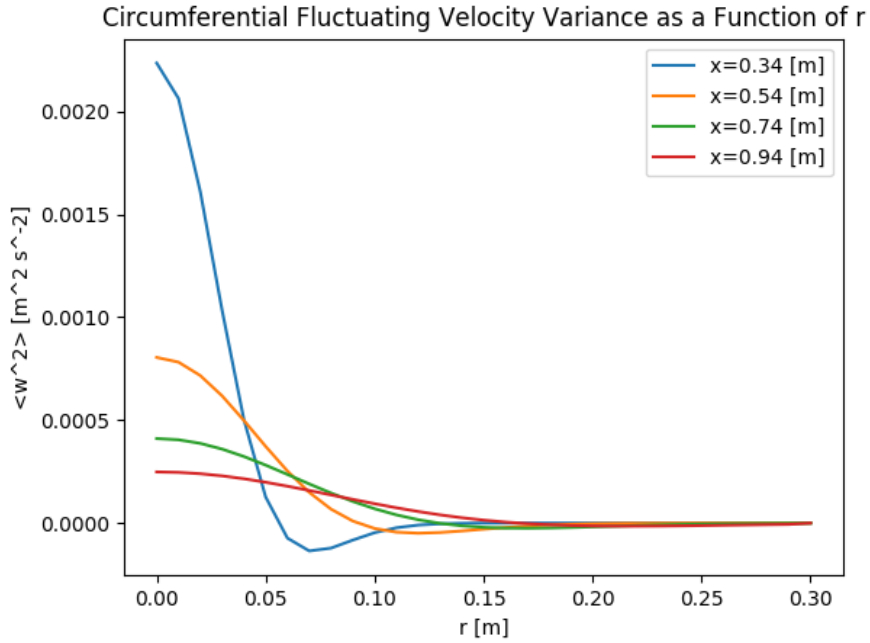
After executing the code the following plots must be obtained.

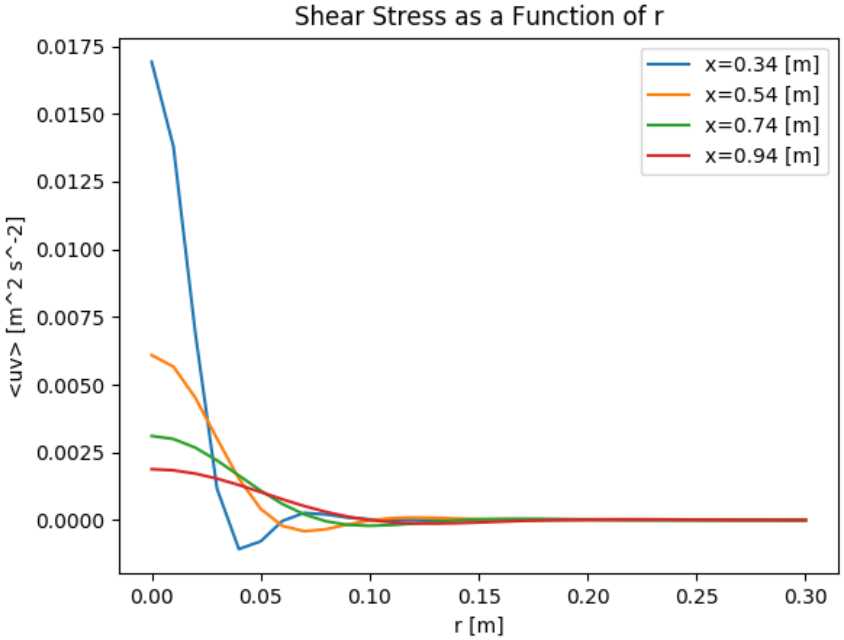


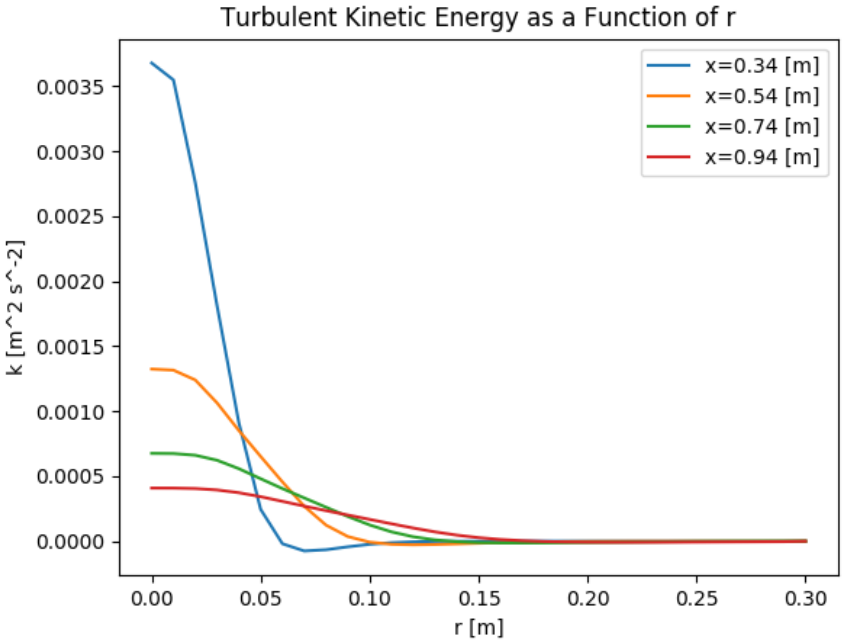
Try to answer the following questions.

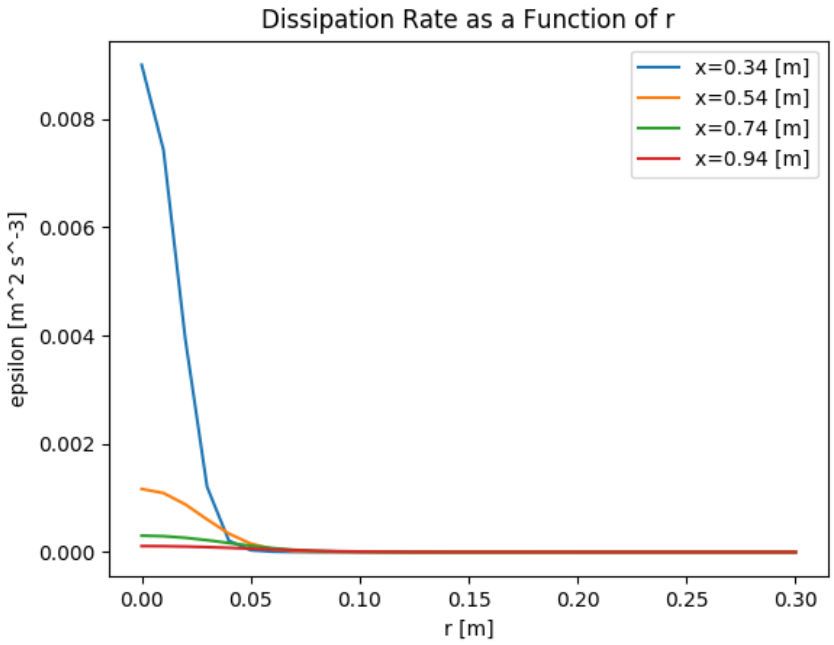
- For which turbulent statistic(s) the peak for a given curve at an axial distance does not occur at $r = 0\,\text{m}$. Can this behaviour be normal?
- For some turbulent fluctuating velocity variances, i.e. $\langle u_i^2 \rangle$, the curves slightly appear in the negative region. Can this be physically possible? If not, what is the cause for this?
- What would happen if for each plot below, the turbulent statistic were normalized?


Radial Fluctuating Velocity Variance as a Function of r
x=0.34 [m]
x=0.54 [m]
x=0.74 [m]
x=0.94 [m]
<v^2> [m^2 s^-2]
0.0020
0.0015
0.0010
0.0005
0.0000
0.00
0.05
0.10
0.15
0.20
0.25
0.30
r [m]


Circumferential Fluctuating Velocity Variance as a Function of r
x=0.34 [m]
x=0.54 [m]
x=0.74 [m]
x=0.94 [m]
<w^2> [m^2 s^-2]
0.0020
0.0015
0.0010
0.0005
0.0000
0.00
0.05
0.10
0.15
0.20
0.25
0.30
r [m]


Shear Stress as a Function of r
x=0.34 [m]
x=0.54 [m]
x=0.74 [m]
x=0.94 [m]
<uv> [m^2 s^-2]
0.0175
0.0150
0.0125
0.0100
0.0075
0.0050
0.0025
0.0000
0.00
0.05
0.10
0.15
0.20
0.25
0.30
r [m]


Turbulent Kinetic Energy as a Function of r
x=0.34 [m]
x=0.54 [m]
x=0.74 [m]
x=0.94 [m]
k [m^2 s^-2]
0.0035
0.0030
0.0025
0.0020
0.0015
0.0010
0.0005
0.0000
0.00
0.05
0.10
0.15
0.20
0.25
0.30
r [m]


Dissipation Rate as a Function of r
x=0.34 [m]
x=0.54 [m]
x=0.74 [m]
x=0.94 [m]
epsilon [m^2 s^-3]
0.008
0.006
0.004
0.002
0.000
0.00
0.05
0.10
0.15
0.20
0.25
0.30
r [m]

ENGG*6790: Theory and Applications of Turbulence

Two-point Turbulent Statistics

Amir A. Aliabadi

February 9, 2018

1 Introduction

In lectures various two-point turbulent statistics were introduced. The autocorrelation function as a two-point and one-time statistic is given by finding the covariance of turbulent fluctuations at two points $\mathbf{x}$ and $\mathbf{x}+\mathbf{r}$:

$$R_{ij}(\mathbf{r},\mathbf{x},t) \equiv \langle u_i(\mathbf{x}+\mathbf{r},t)u_j(\mathbf{x},t)\rangle. \tag{1}$$

If such a correlation sharply decreases with the increasing distance between the two points, then turbulent flow ought to exhibit small turbulent eddies. On the other hand if this correlation is significant even at larger distances between the two points, then the flow exhibits large turbulent eddies. In the limit of no separation distance between the two points, the correlation represent the Reynolds stresses. In the limit of a very large distance between the two points, the correlation ought to drop to zero since a flow cannot exhibit infinitely large turbulent eddies. The autocorrelation function can give the integral lengthscales of the flow, depending on which components of velocity the autocorrelation is calculated for. For instance the longitudinal and transverse integral lengthscales have been introduced.

Also, the second-order velocity structure functions were introduced as useful statistics to describe the nature of turbulent flows. The second-order velocity structure function is the covariance of the difference in velocity between two points $\mathbf{x}$ and $\mathbf{x}+\mathbf{r}$:

$$D_{ij}(\mathbf{r},\mathbf{x},t) \equiv \langle [U_i(\mathbf{x}+\mathbf{r},t)-U_i(\mathbf{x},t)][U_j(\mathbf{x}+\mathbf{r},t)-U_j(\mathbf{x},t)]\rangle. \tag{2}$$

It is understood that the structure function is computed as an ensemble average for a given location $\mathbf{x}$ and the separation distance $\mathbf{r}$, and at a particular time t for a turbulent flow. It seems that only eddies of size $|\mathbf{r}|$ or smaller can make a significant contribution to the structure function. In other words, if an eddy is much larger than $|\mathbf{r}|$, then it affects velocities at two locations in a similar way, which does not alter the correlation.

The structure function can be related to dissipation rate of turbulent kinetic energy in the flow, and various correlations can be established between structure function and the dissipation rate. Note that while the autocorrelation function is a covariance of the turbulent fluctuations, the structure function is the covariance for the difference of velocities of interest at two points.

In this lab, we wish to calculate turbulent statistics for airflow and temperature using a dataset from a micro-climate study on the campus of the University of Guelph. The campaign was conducted from August 13, 2017 to August 25, 2017. Part of the study involved installing two sonic anemometers: one on the roof of the Rozhanski Hall and the other on the street of Reek Walk. The anemometers measured air velocity in horizontal components U, V and vertical component W in units of $\mathrm{m\,s^{-1}}$. Note that V was air velocity along canyon axis, while U was air velocity cross canyon axis. The anemometers also measured air sonic temperature T in units of K. The measurement was conducted at a sampling frequency of 4 Hz. Figure below shows the campaign site and the anemometers.

Figure 1: University of Guelph micro-climate campaign in August 2017: campaign site (top); station 1: sonic anemometer installation on the roof of Rozhanski Hall (bottom left); station 2: sonic anemometer installation on the street of Reek Walk (bottom right).

We wish to compute the following statistics at time intervals of 30 min. The statistics involve autocorrelations and structure functions. Table below shows the statistics to be calculated. In this lab we do not normalize the statistics.

Table 1: Turbulent statistics to be calculated; subscript 1 refers to the roof data and subscript 2 refers to the street data.

Statistic	Units
$R_{UU} = \langle u_1 u_2 \rangle$	[m^2 s^{-2}]
$R_{VV} = \langle v_1 v_2 \rangle$	[m^2 s^{-2}]
$R_{WW} = \langle w_1 w_2 \rangle$	[m^2 s^{-2}]
$R_{UV} = \langle u_1 v_2 \rangle$	[m^2 s^{-2}]
$R_{VU} = \langle v_1 u_2 \rangle$	[m^2 s^{-2}]
$R_{UW} = \langle u_1 w_2 \rangle$	[m^2 s^{-2}]
$R_{WU} = \langle w_1 u_2 \rangle$	[m^2 s^{-2}]
$R_{VW} = \langle v_1 w_2 \rangle$	[m^2 s^{-2}]
$R_{WV} = \langle w_1 v_2 \rangle$	[m^2 s^{-2}]
$R_{TT} = \langle t_1 t_2 \rangle$	[K^2]
$R_{UT} = \langle u_1 t_2 \rangle$	[K m s^{-1}]
$R_{TU} = \langle t_1 u_2 \rangle$	[K m s^{-1}]
$R_{VT} = \langle v_1 t_2 \rangle$	[K m s^{-1}]
$R_{TV} = \langle t_1 v_2 \rangle$	[K m s^{-1}]
$R_{WT} = \langle w_1 t_2 \rangle$	[K m s^{-1}]
$R_{TW} = \langle t_1 w_2 \rangle$	[K m s^{-1}]
$D_{UU} = \langle [U_1 - U_2][U_1 - U_2] \rangle$	[m^2 s^{-2}]
$D_{VV} = \langle [V_1 - V_2][V_1 - V_2] \rangle$	[m^2 s^{-2}]
$D_{WW} = \langle [W_1 - W_2][W_1 - W_2] \rangle$	[m^2 s^{-2}]
$D_{UV} = \langle [U_1 - U_2][V_1 - V_2] \rangle$	[m^2 s^{-2}]
$D_{UW} = \langle [U_1 - U_2][W_1 - W_2] \rangle$	[m^2 s^{-2}]
$D_{VW} = \langle [V_1 - V_2][W_1 - W_2] \rangle$	[m^2 s^{-2}]
$D_{TT} = \langle [T_1 - T_2][T_1 - T_2] \rangle$	[K^2]
$D_{UT} = \langle [U_1 - U_2][T_1 - T_2] \rangle$	[K m s^{-1}]
$D_{VT} = \langle [V_1 - V_2][T_1 - T_2] \rangle$	[K m s^{-1}]
$D_{WT} = \langle [W_1 - W_2][T_1 - T_2] \rangle$	[K m s^{-1}]

2 Python Script

We perform the calculation of turbulent statistics in one script and the plotting of the results in another script. Complete the following script for calculations. In this script, we define two file names as `"Roof4Hz.txt"` and `"Street4Hz.txt"` to be read by the program. We subsequently define another file name as `"4HzTwoPointStatistics.txt"` to write the result of our calculations. Note that the input file name has many columns of data, not all of which need to be read by the program. Use of the `usecols=[...]` argument in the `numpy.loadtxt()` function allows us to only read the columns that we need.

Note that turbulent fluctuations should only be detrended for the autocorrelation function and not the structure function. The structure function should take the original observations. Complete the first script below first.

```
#Calculate two-point turbulent statistics
import random
import sys
import os
import numpy
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import datetime

#Define averaging period in number of data points: minutes * seconds * samples
AverageSample=30*60*4

#Define file names 1: Roof, 2: Street
fileName1 = "Roof4Hz.txt"
fileName2 = "Street4Hz.txt"

outputFileNameTwoPointStatistics="4HzTwoPointStatistics.txt"

#Load all data in matrices
data4Hz1 = numpy.loadtxt(fileName1, usecols=[0,1,2,3,4,5,8,9,10,11])

year4Hz1=data4Hz1[:,0]
month4Hz1=data4Hz1[:,1]
day4Hz1=data4Hz1[:,2]
timeHr4Hz1=data4Hz1[:,3]
timeMin4Hz1=data4Hz1[:,4]
timeSec4Hz1=data4Hz1[:,5]
U4Hz1=data4Hz1[:,6]
V4Hz1=data4Hz1[:,7]
W4Hz1=data4Hz1[:,8]
TSonic4Hz1=data4Hz1[:,9]

N4Hz1=numpy.size(year4Hz1)

data4Hz2 = numpy.loadtxt(fileName2, usecols=[0,1,2,3,4,5,8,9,10,11])

year4Hz2=data4Hz2[:,0]
month4Hz2=data4Hz2[:,1]
day4Hz2=data4Hz2[:,2]
timeHr4Hz2=data4Hz2[:,3]
timeMin4Hz2=data4Hz2[:,4]
timeSec4Hz2=data4Hz2[:,5]
```

```
U4Hz2=data4Hz2[:,6]
V4Hz2=data4Hz2[:,7]
W4Hz2=data4Hz2[:,8]
TSonic4Hz2=data4Hz2[:,9]

#Calculate the number of samples
NSample=int(N4Hz1/AverageSample)

#Define statistics: the first subscript is for roof the second is for street
#R for two-point correlation
#D for structure function
yearavg=numpy.zeros((NSample,1))
monthavg=numpy.zeros((NSample,1))
dayavg=numpy.zeros((NSample,1))
timeHravg=numpy.zeros((NSample,1))
timeMinavg=numpy.zeros((NSample,1))
RUU=numpy.zeros((NSample,1))
RVV=...
RWW=...
RUV=...
RVU=...
RUW=...
RWU=...
RVW=...
RWV=...
RTT=...
RUT=...
RTU=...
RVT=...
RTV=...
RWT=...
RTW=...
DUU=...
DVV=...
DWW=...
DUV=...
DUW=...
DVW=...
DTT=...
DUT=...
DVT=...
DWT=...

for i in range(0,NSample):
    #Calculate year, month, day, hour, and minute for each sample
    yearavg[i] = numpy.mean(year4Hz1[i*AverageSample:(i+1)*AverageSample])
```

```
monthavg[i] = numpy.mean(month4Hz1[i*AverageSample:(i+1)*AverageSample])
dayavg[i] = numpy.mean(day4Hz1[i*AverageSample:(i+1)*AverageSample])
timeHravg[i] = numpy.mean(timeHr4Hz1[i*AverageSample:(i+1)*AverageSample])
timeMinavg[i] = numpy.mean(timeMin4Hz1[i*AverageSample:(i+1)*AverageSample])+1

#Define a vector for the number of data points in each sample
x = [j for j in range(0, AverageSample)]
#Detrend each sample, i.e. remove a straight line fit from the sample
U1 = U4Hz1[i*AverageSample:(i+1)*AverageSample]
Umodel1 = numpy.polyfit(x,U1,1)
Utrend1 = numpy.polyval(Umodel1,x)
Udetrended1 = U1 - Utrend1
U2 = U4Hz2[i*AverageSample:(i+1)*AverageSample]
Umodel2 = numpy.polyfit(x,U2,1)
Utrend2 = numpy.polyval(Umodel2,x)
Udetrended2 = U2 - Utrend2
V1 = ...
Vmodel1 = ...
Vtrend1 = ...
Vdetrended1 = ...
V2 = ...
Vmodel2 = ...
Vtrend2 = ...
Vdetrended2 = ...
W1 = ...
Wmodel1 = ...
Wtrend1 = ...
Wdetrended1 = ...
W2 = ...
Wmodel2 = ...
Wtrend2 = ...
Wdetrended2 = ...
TSonic1 = ...
TSonicmodel1 = ...
TSonictrend1 = ...
TSonicdetrended1 = ...
TSonic2 = ...
TSonicmodel2 = ...
TSonictrend2 = ...
TSonicdetrended2 = ...

#Calculate variances, and covariances
RUUCovMatrix = numpy.cov(Udetrended1, Udetrended2)
RVVCovMatrix = ...
RWWCovMatrix = ...
RUVCovMatrix = numpy.cov(Udetrended1, Vdetrended2)
```

```
RVUCovMatrix = ...
RUWCovMatrix = ...
RWUCovMatrix = ...
RVWCovMatrix = ...
RWVCovMatrix = ...
RTTCovMatrix = numpy.cov(TSonicdetrended1, TSonicdetrended2)
RUTCovMatrix = numpy.cov(Udetrended1, TSonicdetrended2)
RTUCovMatrix = ...
RVTCovMatrix = ...
RTVCovMatrix = ...
RWTCovMatrix = ...
RTWCovMatrix = ...

RUU[i] = RUUCovMatrix[1,0]
RVV[i] = RVVCovMatrix[1,0]
RWW[i] = RWWCovMatrix[1,0]
RUV[i] = RUVCovMatrix[1,0]
RVU[i] = ...
RUW[i] = ...
RWU[i] = ...
RVW[i] = ...
RWV[i] = ...
RTT[i] = ...
RUT[i] = ...
RTU[i] = ...
RVT[i] = ...
RTV[i] = ...
RWT[i] = ...
RTW[i] = ...

DUUCovMatrix = numpy.cov(U1-U2, U1-U2)
DVVCovMatrix = numpy.cov(V1-V2, V1-V2)
DWWCovMatrix = ...
DUVCovMatrix = ...
DUWCovMatrix = ...
DVWCovMatrix = ...
DTTCovMatrix = numpy.cov(TSonic1-TSonic2, TSonic1-TSonic2)
DUTCovMatrix = numpy.cov(U1-U2, TSonic1-TSonic2)
DVTCovMatrix = ...
DWTCovMatrix = ...

DUU[i] = DUUCovMatrix[1,0]
DVV[i] = DVVCovMatrix[1,0]
DWW[i] = DWWCovMatrix[1,0]
DUV[i] = DUVCovMatrix[1,0]
DUW[i] = ...
```

```
    DVW[i] = ...
    DTT[i] = ...
    DUT[i] = ...
    DVT[i] = ...
    DWT[i] = ...

#Write data to file
outputFile = open(outputFileNameTwoPointStatistics, "w")
outputFile.write("#Times in Local Daylight Time \n")
outputFile.write("#0:Year \t 1:Month \t 2:Day \t 3:Hour \t 4:Minute \t \
    5:RUU (m2 s-2) \t 6:RVV (m2 s-2) \t 7:RWW (m2 s-2) \t \
    8:RUV (m2 s-2) \t 9:RVU (m2 s-2) \t 10:RUW (m2 s-2) \t \
    11:RWU (m2 s-2) \t 12:RVW (m2 s-2) \t 13:RWV (m2 s-2) \t \
    14:RTT (K2) \t 15:RUT (Km s-1) \t 16:RTU (Km s-1) \t \
    17:RVT (Km s-1) \t 18:RTV (Km s-1) \t 19:RWT (Km s-1) \t 20:RTW (Km s-1) \t \
    21:DUU (m2 s-2) \t 22:DVV (m2 s-2) \t 23:DWW (m2 s-2) \t 24:DUV (m2 s-2) \t \
    25:DUW (m2 s-2) \t 26:DVW (m2 s-2) \t \
    27:DTT (K2) \t 28:DUT (Km s-1) \t 29:DVT (Km s-1) \t 30:DWT (Km s-1) \n")

for i in range(0,NSample):
    outputFile.write("%i \t %i \t %i \t %i \t %i \t \
    %f \t %f \t %f \t \
    %f \t %f \t %f \t %f \t %f \t %f \t \
    %f \t %f \t %f \t \
    %f \t %f \t %f \t %f \t \
    %f \t %f \t %f \t %f \t %f \t %f \t \
    %f \t %f \t %f \t %f \n" \
    % (yearavg[i], ...))
outputFile.close()
```

After running this script, we can generate a text file with all the turbulent statistics. The next step is to run a new script for reading the results and plotting the turbulent statistics. This script is given to you as `PlotResults`. The script first reads the results text file and assigns the results to specific vectors. The next step is to create a vector to contain the time for each measurement in seconds.

For each turbulent statistic, two plots are generated. The first plot shows the time series for the quantity of interest. The second plot shows the diurnal variation of the quantity of interest. The diurnal plot overlays the quantity of interest over many days as a function of hour in the day from 0 to 23 of the Local Daylight Time zone. This helps identify which quantities exhibit a strong diurnal variation. Due to the presence of spurious data in this lab, we will limit the vertical axis in the range $[-0.5, 0.5]$ for autocorrelation functions and $[-1, 3]$ for structure functions. This can be done using the command `plt.ylim((-0.5,0.5))` or `plt.ylim((-1,3))` for each plot, respectively.

After successfully running the second script, the following figures should be obtained. Try to

answer the following questions.

- We calculated both R_{UV} and R_{VU}, but we only calculated D_{UV}. Why did we not calculate D_{VU}?
- Explain why the magnitudes of R_{UU} and R_{VV} during the early afternoon is greater than the magnitude of R_{WW}? Try to explain why is R_{UU} negative, while R_{VV} is positive during the early afternoon?
- Explain why the magnitudes of R_{UV} and R_{VU} are greater than the magnitudes of R_{UW}, R_{WU}, R_{VW}, and R_{WV}?
- Considering all the autocorrelation functions for velocities, and knowing the fact that they describe integral lengthscales, explain turbulent eddies in which direction and during what time periods are the largest?
- We can observe the fact that R_{TT} is positive during the early afternoon hours. Does this suggest that the eddies causing rising hot thermals from the surfaces may have integral lengthscales that can be as large as the building length scale?
- Mathematically, explain why D_{UU}, D_{VV}, D_{WW}, and D_{TT} are strictly positive? Why do they peak in the early afternoon hours?
- Explain why D_{UU} and D_{VV} are greater than D_{WW}?
- Explain why D_{UV} tends to be positive during the early afternoon hours?
- Explain why D_{VW} tends to be negative during the early afternoon hours?
- Explain why D_{WT} tends to be positive during the early afternoon hours?

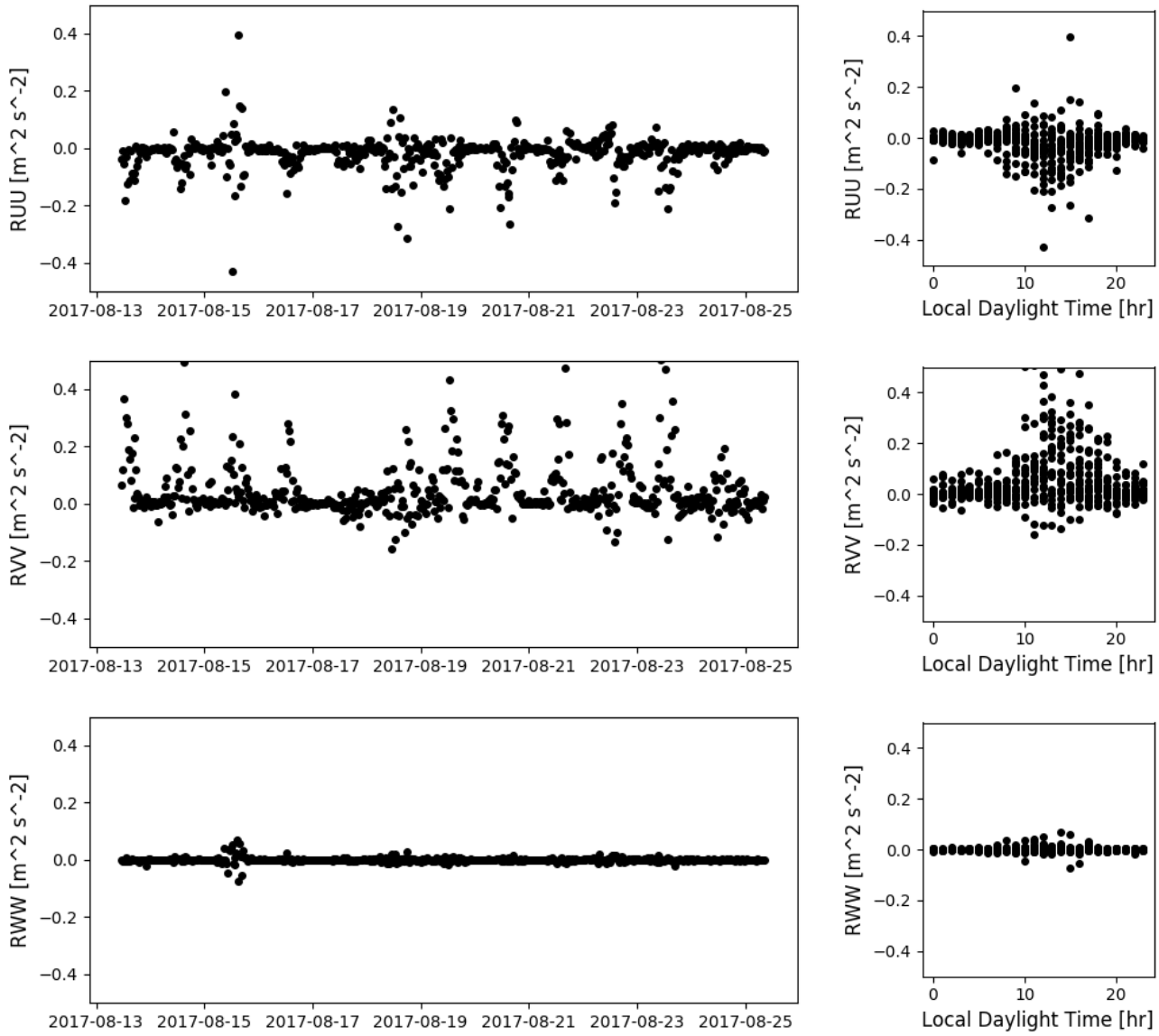


Figure 2: Autocorrelation functions for R_{UU}, R_{VV}, and R_{WW}.

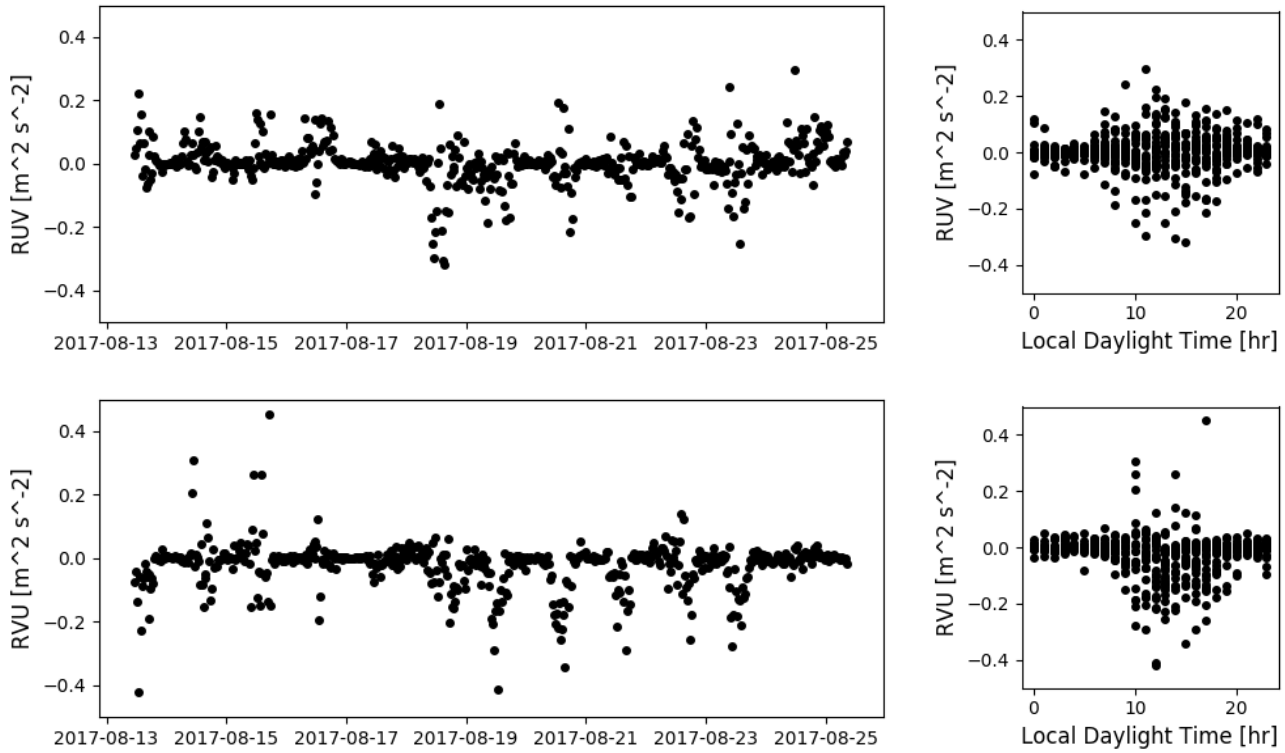


Figure 3: Autocorrelation functions for R_{UV} and R_{VU}.

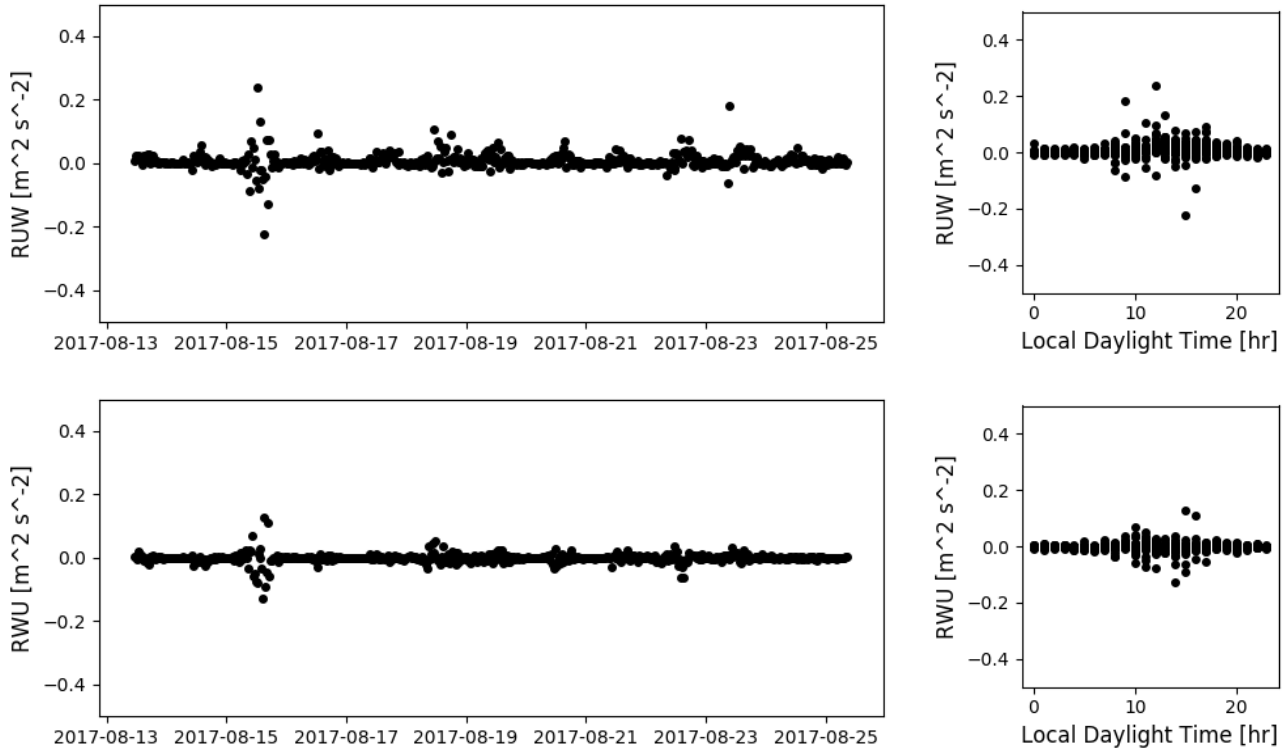


Figure 4: Autocorrelation functions for R_{UW} and R_{WU}.

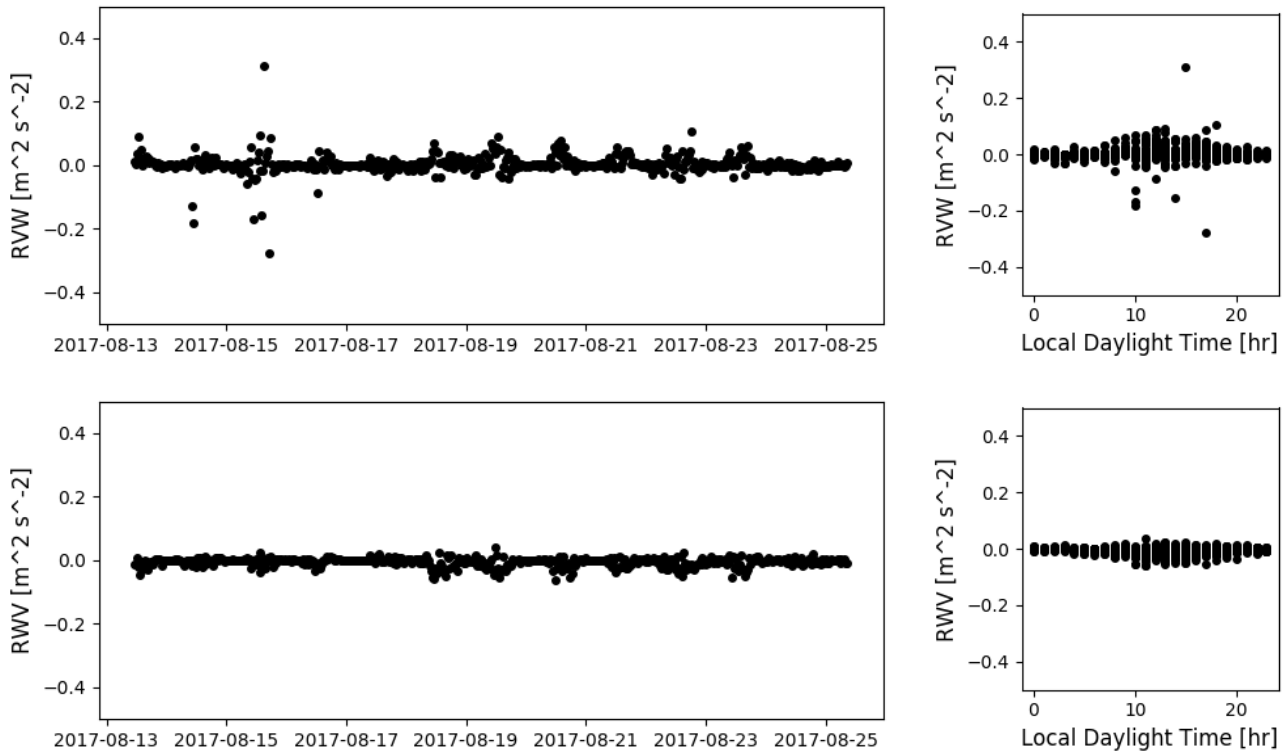


Figure 5: Autocorrelation functions for R_{VW} and R_{WV}.

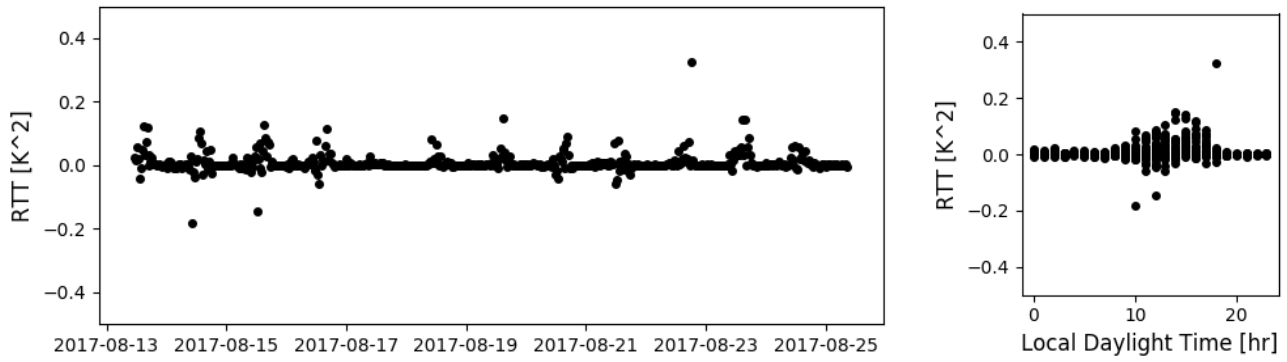


Figure 6: Autocorrelation functions for R_{TT}.

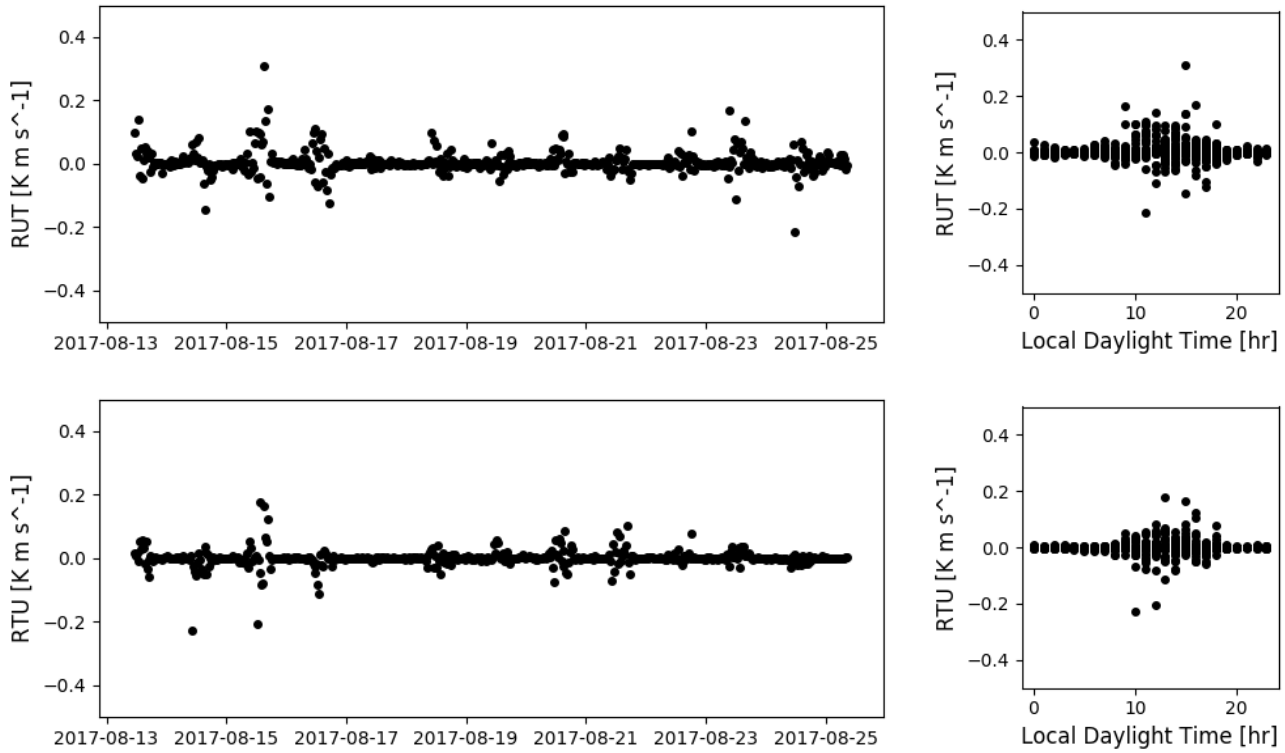


Figure 7: Autocorrelation functions for R_{UT} and R_{TU}.

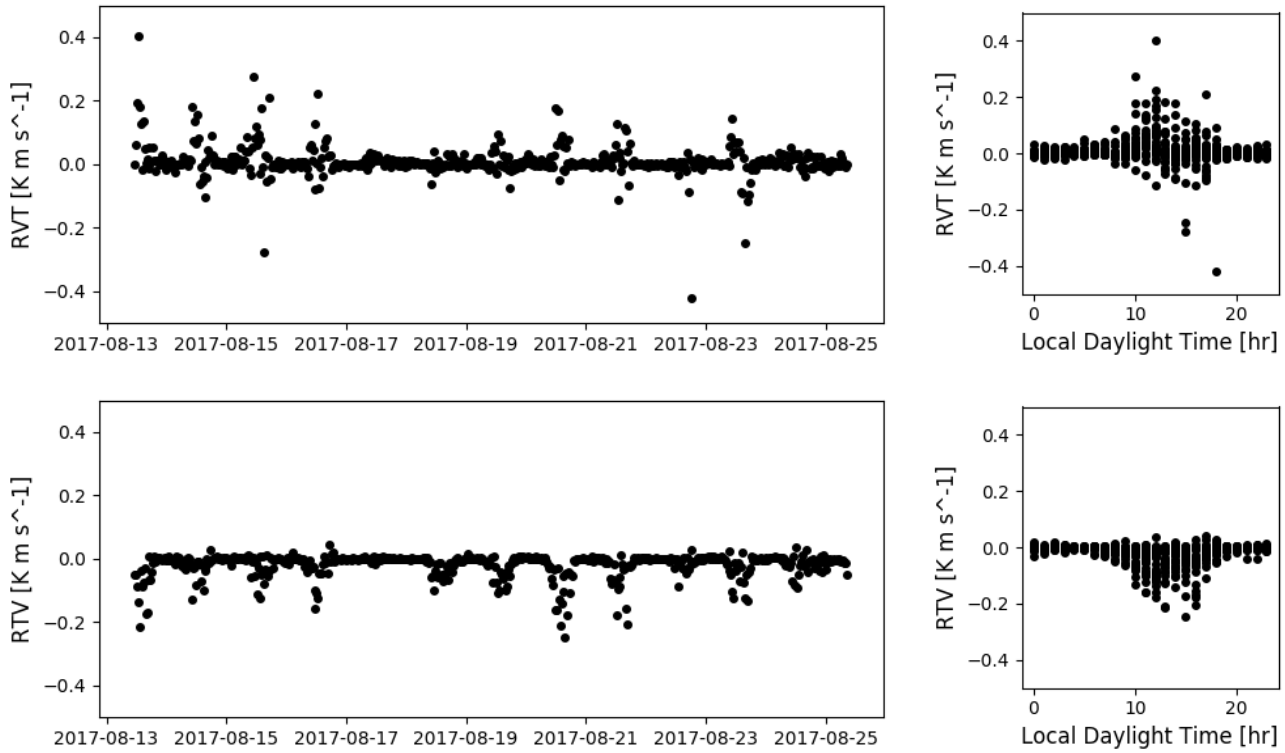


Figure 8: Autocorrelation functions for R_{VT} and R_{TV}.

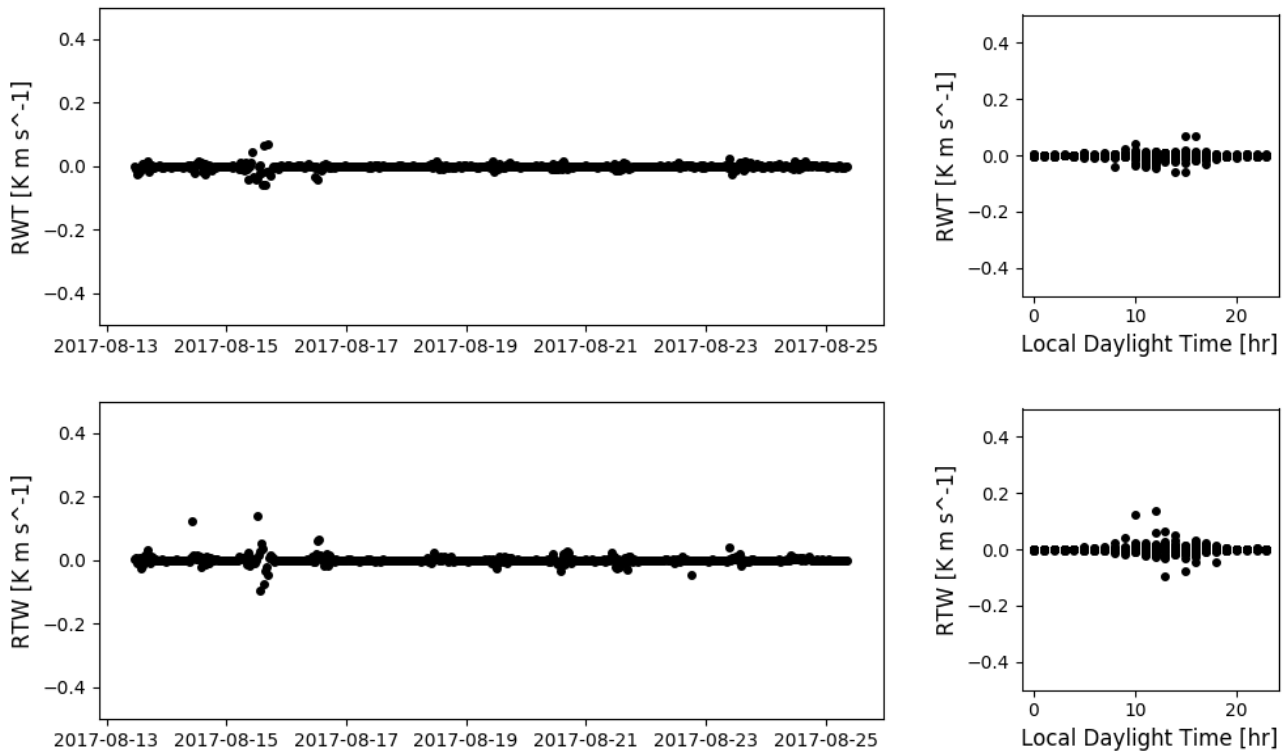


Figure 9: Autocorrelation functions for R_{WT} and R_{TW}.

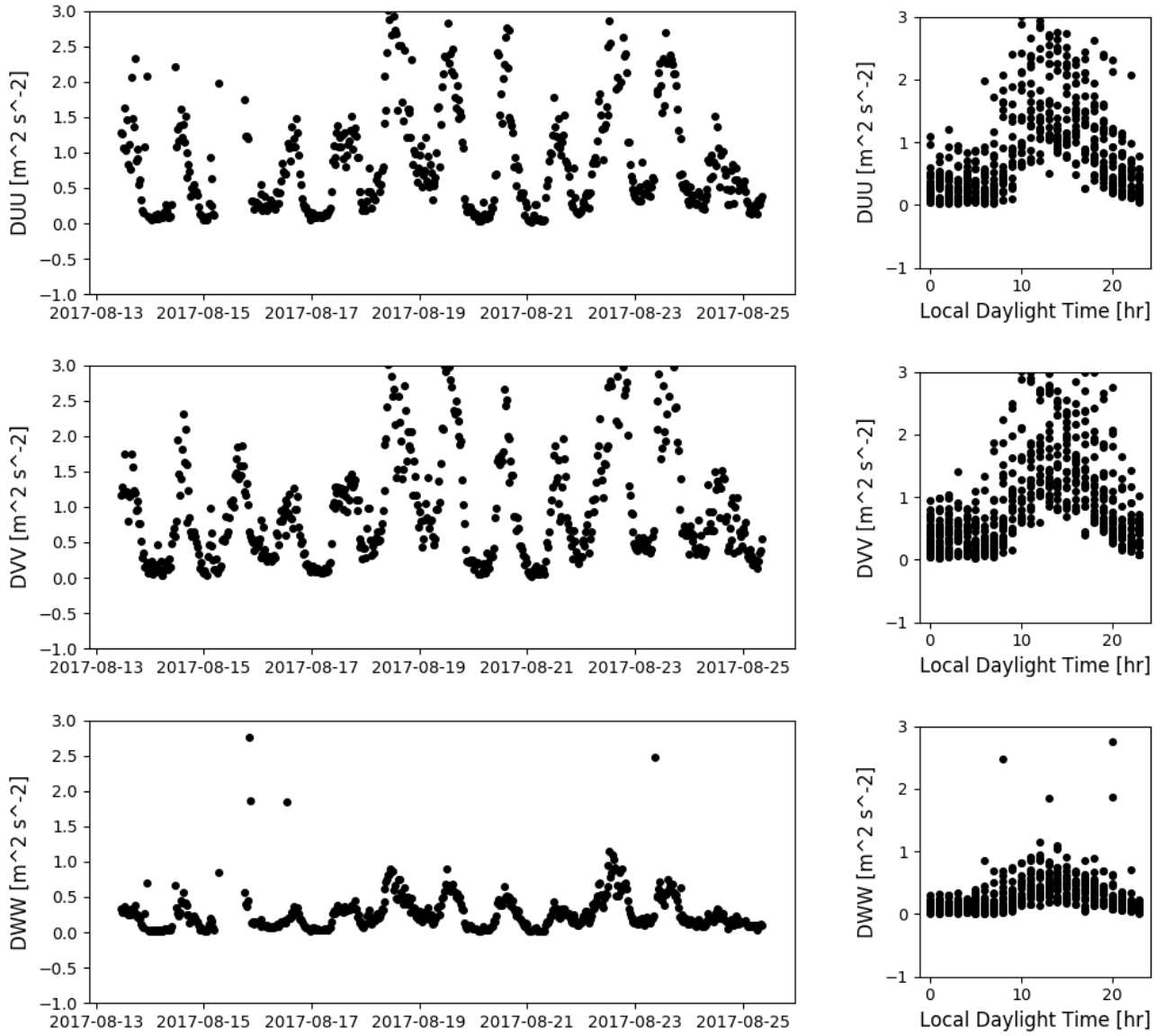


Figure 10: Structure functions for D_{UU}, D_{VV}, and D_{WW}.

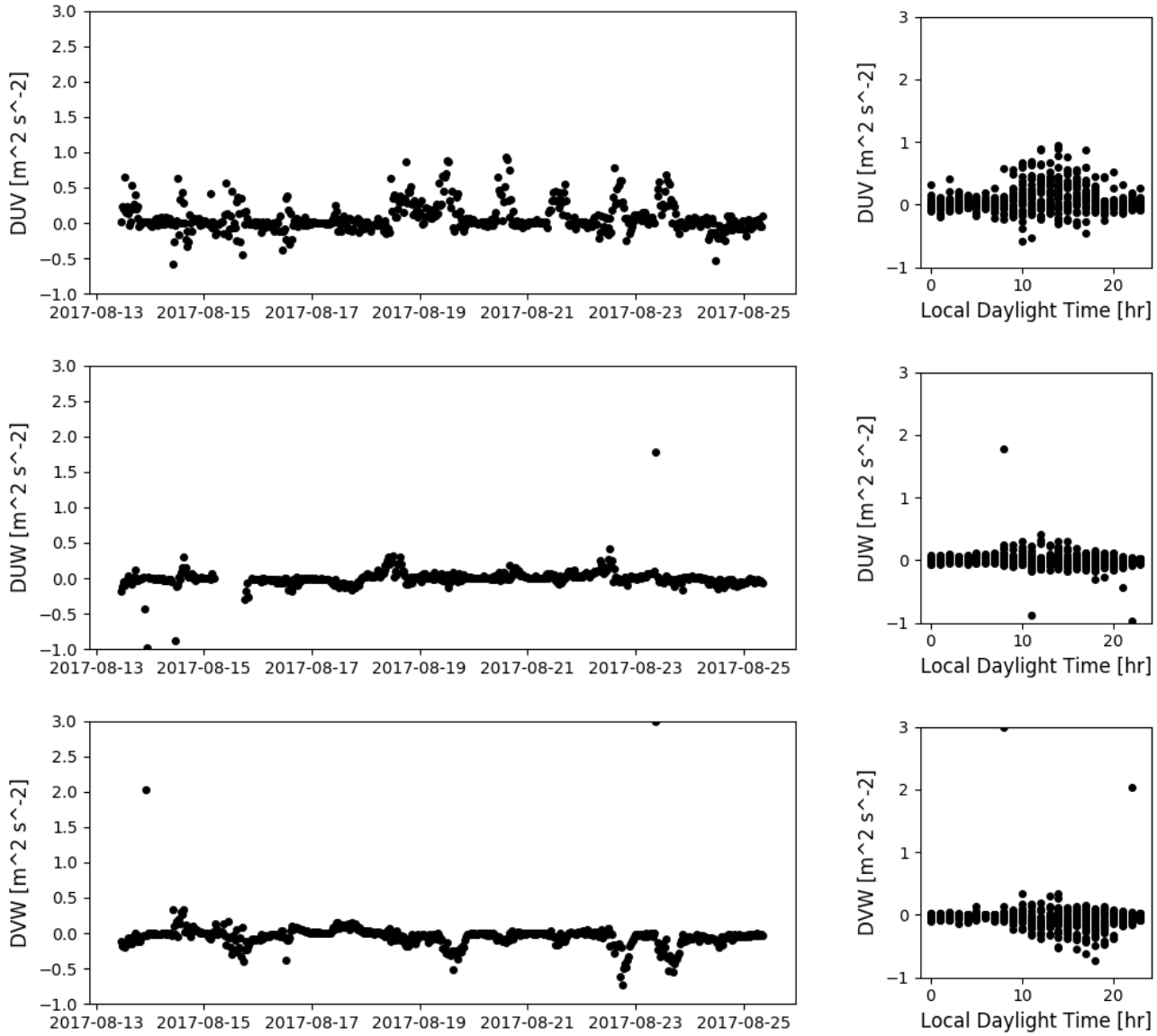


Figure 11: Structure functions for D_{UV}, D_{UW}, and D_{VW}.

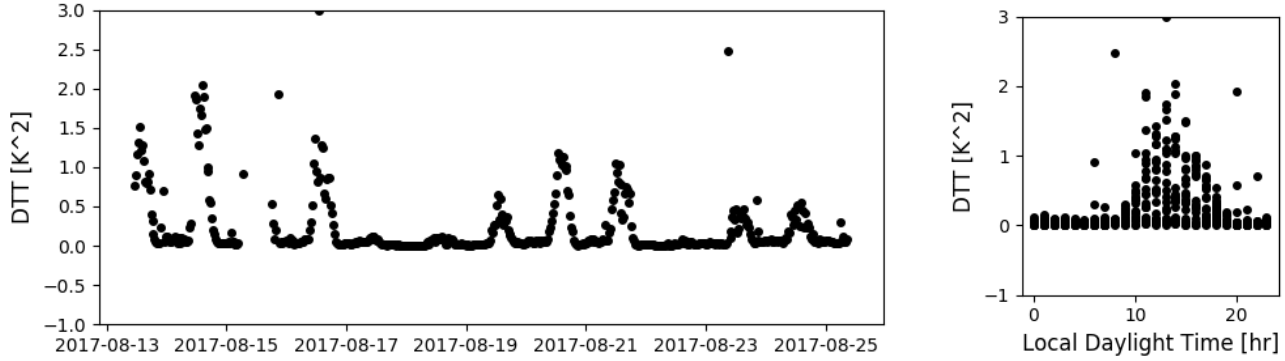


Figure 12: Structure functions for D_{TT}.

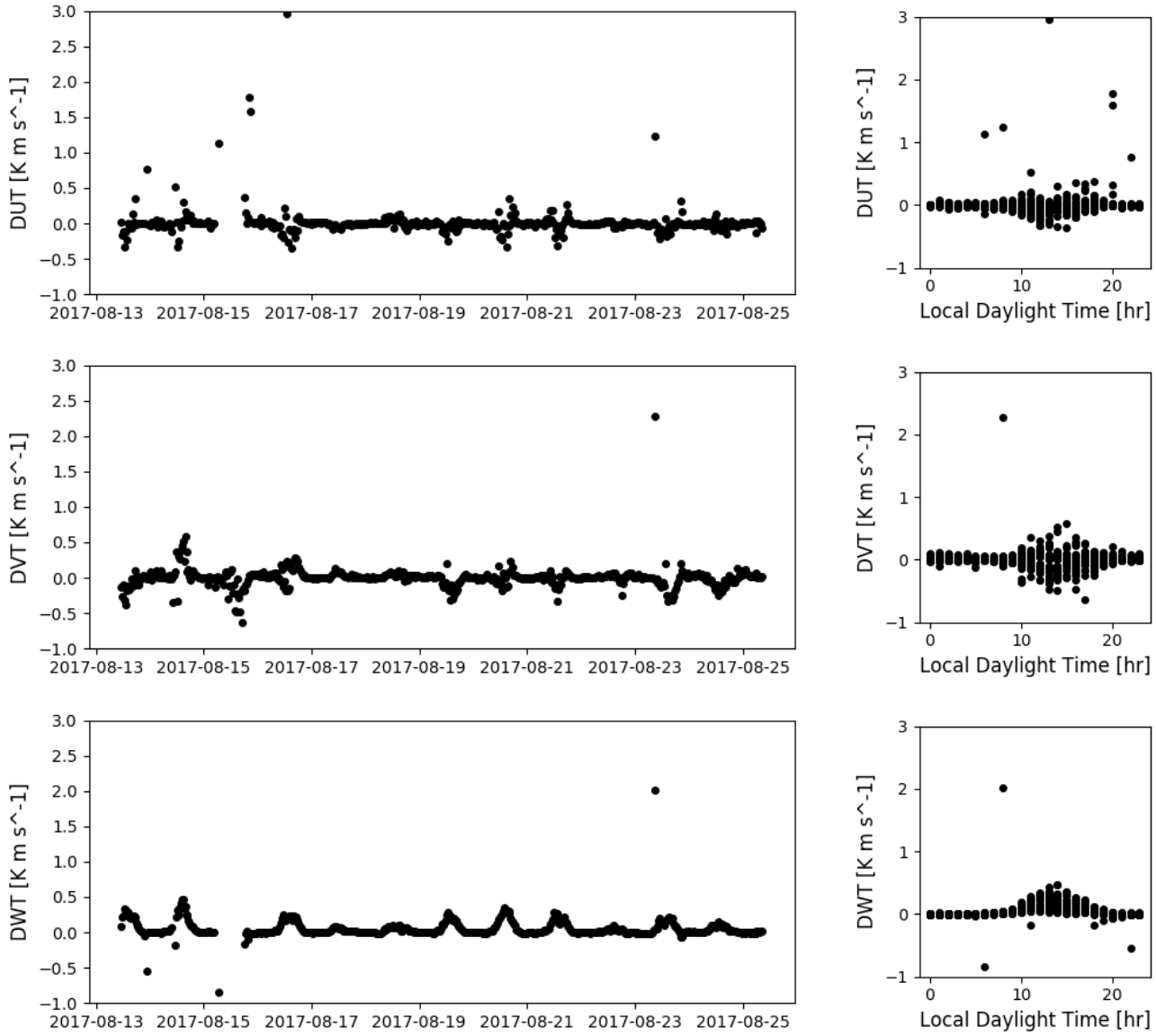


Figure 13: Structure functions for D_{UT}, D_{VT}, and D_{WT}.

ENGG*6790: Theory and Applications of Turbulence

Integral Time and Length Scales

Amir A. Aliabadi

November 11, 2017

1 Introduction

In lectures, the concepts of integral time and length scales are discussed. For a statistically stationary process, the *autocovariance* is given by

$$R_{ii}(s) \equiv \langle u_i(t) u_i(t+s) \rangle \tag{1}$$

where a flow quantity is measured at the same location with a time shift s over an indefinite time period. Note that autocovariance is not a function of time t but instead a function of time shift s. Here, $u_i(t) \equiv U_i(t) - \langle U_i(t) \rangle$ is the velocity fluctuation. Related to the concept of autocorrelation is the concept of *integral timescale*. Usually the autocorrelation function diminishes quickly as a function of time shift. This means that usually the integral of the autocorrelation function over time shift converges to the integral timescale define as

$$\overline{\tau}_{ii} = \frac{1}{\langle u_i^2 \rangle} \int_0^\infty R_{ii}(s) ds. \tag{2}$$

In essence, the integral timescale looks at the overall memory of the process and how strongly it is influenced or correlated by state of the flow in a previous time. This integral timescale can be understood as the characteristic time for eddies.

In a related concept, the two-point and one-time autocovariance for the same component of the flow is defined by

$$R_{ii}(\mathbf{r}, \mathbf{x}, t) \equiv \langle u_i(\mathbf{x}, t) u_i(\mathbf{x} + \mathbf{r}, t) \rangle, \tag{3}$$

which is also known as a *two-point correlation*. With this statistic it is possible to define an *integral lengthscale*, for instance by

$$L_{11}(\mathbf{x},t) \equiv \frac{1}{R_{11}(0,\mathbf{x},t)} \int_0^\infty R_{11}(\mathbf{e}_1 r,\mathbf{x},t)dr, \tag{4}$$

where $\mathbf{e}_1$ is the unit vector in the x_1-coordinate direction. The integral length scale measures the correlation distance of a process in terms of space or time. In essence, it looks at the overall memory of the process and how it is influenced by previous positions and parameters. This integral length scale can be understood as the characteristic size for eddies.

Measurement of the one-time and two-point velocity correlation function $R_{ii}(\mathbf{r})$ experimentally is very difficult since this requires two stationary probes that need to be used in a large number of experiments with variable separation distance $\mathbf{r}$. Alternatively it is possible to use a moving probe in turbulent flow that travels at very high speed V along $\mathbf{r}$. Let us assume that $\mathbf{r}$ is in the direction of the first component of the Cartesian coordinate system, i.e. $\mathbf{r} = r\mathbf{e}_1$, where $\mathbf{e}_1$ is the unit vector along the first component. Now if the autocorrelation for the moving probe with a time shift s is calculated, we obtain $R_{ii}^{(m)}(s)$, where the superscript (m) signifies the moving probe. On the other hand for a time shift of s, the probe has moved equal to

$$r = Vs. \tag{5}$$

It is possible to show that if the probe is moving infinitely fast in the flow, then this autocorrelation is equal to the one-time and two-point velocity correlation, i.e.

$$R_{ii}^{(m)}(s) = R_{ii}(r). \tag{6}$$

This concept is the basis of turbulence measurements using flying hot-wire anemometers, where a hot-wire anemometer that is very sensitive to flow measurements with a large sampling rate is flown in a turbulent flow to reveal one-time and two-point velocity correlation functions. In many studies, particularly in atmospheric flows, there is a simpler approach to use. In this technique only a single stationary probe is required for the approximation to be valid so that

$$R_{ii}(s) = R_{ii}(r) \tag{7}$$

where now $V = r/s$ represents the average wind speed. The approximation of spatial correlations by temporal correlations is known as the *Taylor hypothesis* and is only valid for the *frozen turbulence approximation*. This approximation states that eddies can be conceived as frozen and moving with the flow as they travel past a stationary probe. This condition occurs in the atmosphere when turbulent fluctuations are much smaller in magnitude than the average flow velocity, for instance

$$\frac{u_i}{\langle U_i \rangle} \ll 1. \tag{8}$$

Using the Taylor hypothesis, it is possible to approximate the integral length scale such that

$$L_{ii} = \frac{1}{\langle u_i^2 \rangle} \int_0^\infty R_{ii}(r,t)dr \simeq \frac{\langle U_i \rangle}{\langle u_i^2 \rangle} \int_0^\infty R_{ii}(s,t)ds = \langle U_i \rangle \overline{\tau}_{ii} \tag{9}$$

In this lab, we wish to calculate turbulent statistics for airflow and temperature using a dataset from a micro-climate study on the campus of the University of Guelph. The campaign was conducted from August 13, 2017 to August 25, 2017. Part of the study involved installing a sonic anemometer on the roof of the Rozhanski Hall. The anemometer measured air velocity in horizontal components U, V and vertical component W in units of $\mathrm{m\,s^{-1}}$. Note that V was air velocity along canyon axis, while U was air velocity cross canyon axis. It also measured air sonic temperature T in units of K. The measurement was conducted at a sampling frequency of 4 Hz. Figure below shows the campaign site and the roof anemometer.

We wish to compute the following integral scales at time intervals of 30 min. A particular challenge is that due to the transient nature of the flow, we cannot integrate the necessary integrals to infinity. Since our time intervals are limited we will perform the necessary integrations up to 1 minute and 2 minutes and compare the effects on the integral scales.

Table 1: Integral scales to be calculated

Statistic	Description	Units
τ_{UU}	Integral timescale in $x-$direction	[s]
τ_{VV}	Integral timescale in $y-$direction	[s]
τ_{WW}	Integral timescale in $z-$direction	[s]
τ_{TT}	Integral timescale for thermal fluctuations	[s]
L_{UU}	Integral lengthscale in $x-$direction	[m]
L_{VV}	Integral lengthscale in $y-$direction	[m]
L_{WW}	Integral lengthscale in $z-$direction	[m]

2 Python Script

We perform the calculation of integral scales in one script and the plotting of the results in another script. Complete the following script for calculations. In this script, we define a file name as `"Roof4Hz.txt"` to be read by the program. We subsequently define another file name as `"Roof4HzIntegralScales.txt"` to write the result of our calculations. Note that the input file name has many columns of data, not all of which need to be read by the program. Use of the `usecols=[...]` argument in the `numpy.loadtxt()` function allows us to only read the columns that we need.

An important requirement for calculating the integral scales is that the time series data must be detrended for each time interval, in which we desire to calculate the integral scales. The idea behind detrending is that many environmental data show linear trends that must be eliminated (or subracted) from the data before calculating turbulent statistics. These linear trends really do not contribute to turbulence and are slow background variations (in this case diurnal variations).

Figure 1: University of Guelph micro-climate campaign in August 2017: campaign site (top) and sonic anemometer installation on the roof of Rozhanski Hall (bottom)

If the linear trend is not removed from the data, one may report spuriously high integral scales. Figure below shows a trended and a detrended time series.

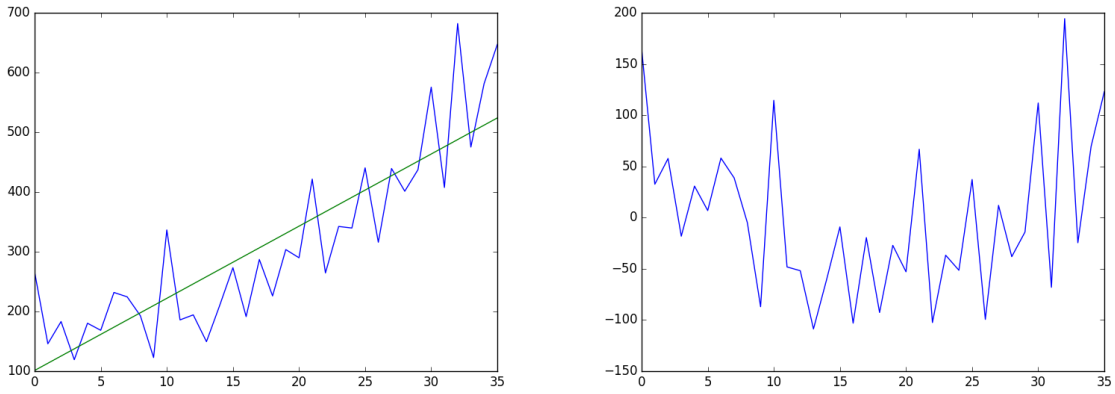


Figure 2: A trended time series (left) and a detrended time series (right); integral scales must always be calculated after removing a trend from a time series, or else spurious results may be reported.

Detrending of the time series is achieved by first fitting a first order polynomial to the time series using the `numpy.polyfit()` function. Subsequently, a model is built based on this fit using the `numpy.polyval()` function. Finally, the model, which is really only a line, is subtracted from the original time series to give the detrended time series. All the subsequent integral scales are calculated from the detrended time series. Of course, the detrended time series contains all the turbulent fluctuations.

To calculate autocorrelations with a time shift, we should use a sliding window, with variable `WindowLength`, that is only half the width of each time interval, and then calculate the autocorrelation for each time shift. It is trivial that the time shift cannot be greater than half of the sampling interval, i.e. 15 minutes. We should also define a variable `NIntegral` that will cover either 1 minutes or two minutes.

To calculate the variances, we have used the `numpy.cov()` function. This function is extremely useful. It takes two vectors as arguments and returns a 2 by 2 matrix. The main diagonal elements of the matrix are the variances of the two vectors provided, and the off-diagonal elements are covariances. The elements of the matrix can be accessed and assigned to appropriate variables. Element `[0,0]` is the variance of the first vector argument, element `[1,1]` is the variance of the second vector argument, and element `[0,1]` (the same as element `[1,0]`) is the covariance of the two vectors.

Note that the iterations move forward for each 30 min window of data. In each iteration, the program calculates the integral scales, stores them in vectors, and moves on to the next 30 min window. Since the volume of calculations is large for this lab, we print to screen the iteration number after each iteration completed.

Finally, the data are written to a file with an appropriate header. Note that using `#`, or com-

ment line, in a text file results in useful header information that will not really be read by the `numpy.loadtxt()` function. It is recommended to describe in the text file exactly what information each column holds and the units associated with it. It is also useful to number the columns so subsequently files can be read conveniently.

```
#Calculate integral time and length scales assuming Taylor hypothesis
#using roof/street weather stations with 4Hz sampling
import random
import sys
import os
import numpy
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import datetime

#Define averaging period in number of data points: minutes * seconds * samples
AverageSample=30*60*4

#Define averaging period for taking the integrals:
#this will be half of AverageSample since we have sliding window
WindowLength=int(AverageSample/2)

#Number of integration points: this is adjustable.
#This should be as large as possible
#Without making the integral time scale negative. Why?
NIntegral=1*60*4

#Sampling period [s]
dt=0.25

#Define file names
fileName = "Roof4Hz.txt"

outputFileName="Roof4HzIntegralScales.txt"

#Load all data in a matrix
data4Hz = numpy.loadtxt(fileName, usecols=[0,1,2,3,4,5,8,9,10,11])

year4Hz=data4Hz[:,0]
month4Hz=data4Hz[:,1]
day4Hz=data4Hz[:,2]
timeHr4Hz=data4Hz[:,3]
timeMin4Hz=data4Hz[:,4]
timeSec4Hz=data4Hz[:,5]
U4Hz=data4Hz[:,6]
V4Hz=data4Hz[:,7]
```

```
W4Hz=data4Hz[:,8]
TSonic4Hz=data4Hz[:,9]

N4Hz=numpy.size(year4Hz)

#Calculate the number of samples
NSample=int(N4Hz/AverageSample)

#Define statistics and integral scales
yearavg=numpy.zeros((NSample,1))
monthavg=numpy.zeros((NSample,1))
dayavg=numpy.zeros((NSample,1))
timeHravg=numpy.zeros((NSample,1))
timeMinavg=numpy.zeros((NSample,1))
Uavg=numpy.zeros((NSample,1))
Vavg=numpy.zeros((NSample,1))
Wavg=numpy.zeros((NSample,1))
TSonicavg=numpy.zeros((NSample,1))
Uvar=numpy.zeros((NSample,1))
Vvar=numpy.zeros((NSample,1))
Wvar=numpy.zeros((NSample,1))
TSonicvar=numpy.zeros((NSample,1))
TauUU=numpy.zeros((NSample,1))
TauVV=...
TauWW=...
TauTT=...
LUU=numpy.zeros((NSample,1))
LVV=...
LWW=...

#Define autocorrelation functions needed for each sample,
#i.e. R(s), where s is time shift
RUU=numpy.zeros((NIntegral,1))
RVV=...
RWW=...
RTT=...

RUUAll=numpy.zeros((WindowLength,1))
RVVAll=...
RWWAll=...
RTTAll=...

for i in range(0,NSample):
    #Calculate year, month, day, hour, and minute for each sample
    yearavg[i] = numpy.mean(year4Hz[i*AverageSample:(i+1)*AverageSample])
    monthavg[i] = numpy.mean(month4Hz[i*AverageSample:(i+1)*AverageSample])
```

```
dayavg[i] = numpy.mean(day4Hz[i*AverageSample:(i+1)*AverageSample])
timeHravg[i] = numpy.mean(timeHr4Hz[i*AverageSample:(i+1)*AverageSample])
timeMinavg[i] = numpy.mean(timeMin4Hz[i*AverageSample:(i+1)*AverageSample])+1

#Calculate averages
Uavg[i] = numpy.mean(U4Hz[i*AverageSample:(i+1)*AverageSample])
Vavg[i] = numpy.mean(V4Hz[i*AverageSample:(i+1)*AverageSample])
Wavg[i] = ...
TSonicavg[i] = numpy.mean(TSonic4Hz[i*AverageSample:(i+1)*AverageSample])

#Define a vector for the number of data points in each sample
x = [j for j in range(0, AverageSample)]
#Detrend each sample, i.e. remove a straight line fit from the sample
U = U4Hz[i*AverageSample:(i+1)*AverageSample]
Umodel = numpy.polyfit(x,U,1)
Utrend = numpy.polyval(Umodel,x)
Udetrended = U - Utrend
V = V4Hz[i*AverageSample:(i+1)*AverageSample]
Vmodel = numpy.polyfit(x,V,1)
Vtrend = numpy.polyval(Vmodel,x)
Vdetrended = V - Vtrend
W = ...
Wmodel = ...
Wtrend = ...
Wdetrended = ...
TSonic = TSonic4Hz[i*AverageSample:(i+1)*AverageSample]
TSonicmodel = numpy.polyfit(x,TSonic,1)
TSonictrend = numpy.polyval(TSonicmodel,x)
TSonicdetrended = TSonic - TSonictrend

#Calculate variances
UVCovMatrix = numpy.cov(Udetrended, Vdetrended)
UWCovMatrix = numpy.cov(Udetrended, Wdetrended)
VWCovMatrix = ...
UTSonicCovMatrix = numpy.cov(Udetrended, TSonicdetrended)
VTSonicCovMatrix = numpy.cov(Vdetrended, TSonicdetrended)
WTSonicCovMatrix = numpy.cov(Wdetrended, TSonicdetrended)

Uvar[i] = UVCovMatrix[0,0]
Vvar[i] = UVCovMatrix[1,1]
Wvar[i] = VWCovMatrix[1,1]
TSonicvar[i] = UTSonicCovMatrix[1,1]

#Calculate autocorrelation functions for a sliding window,
#Half the size of each sample interval
#Iterate over s, i.e. the time shift
```

```
    for s in range(0,NIntegral):
        #For each time shift s, iterate over the sliding window
        for k in range(0,WindowLength):
            RUUAll[k] = Udetrended[k]*Udetrended[k+s]
            RVVAll[k] = ...
            RWWAll[k] = ...
            RTTAll[k] = TSonicdetrended[k]*TSonicdetrended[k+s]
        #Calculate autocorrelation function for each s
        RUU[s] = numpy.mean(RUUAll)
        RVV[s] = numpy.mean(RVVAll)
        RWW[s] = ...
        RTT[s] = ...

    #Calculate integral scales
    TauUU[i] = numpy.sum(RUU*dt)/Uvar[i]
    TauVV[i] = ...
    TauWW[i] = ...
    TauTT[i] = numpy.sum(RTT*dt)/TSonicvar[i]

    LUU[i] = numpy.absolute(TauUU[i]*Uavg[i])
    LVV[i] = ...
    LWW[i] = ...

    #Print iteration number to see progress
    print('Iteration = ',i)

#Write data to file
outputFile = open(outputFileName, "w")
outputFile.write("#Times in Local Daylight Time - Subtract 1 hour for LST \n")
outputFile.write("#0:Year \t 1:Month \t 2:Day \t 3:Hour \t 4:Minute \t \
    5:Uavg (m s-1) \t 6:Vavg (m s-1) \t 7:Wavg (m s-1) \t 8:TSonicavg (K) \t \
    9:TauUU (s) \t 10:TauVV (s) \t 11:TauWW (s) \t 12:TauTT (s) \
    13:LUU (m) \t 14:LVV (m) \t 15:LWW (m) \n")

for i in range(0,NSample):
    outputFile.write("%i \t %i \t %i \t %i \t %i \t \
    %f \t %f \t %f \t %f \t \
    %f \t %f \t %f \t %f \t \
    %f \t %f \t %f \n" \
    % (yearavg[i],monthavg[i],dayavg[i],timeHravg[i],timeMinavg[i], \
       Uavg[i], Vavg[i], Wavg[i], TSonicavg[i], \
       TauUU[i], TauVV[i], TauWW[i], TauTT[i], \
       LUU[i], LVV[i], LWW[i]))
outputFile.close()
```

After running this script, we can generate a text file with all the integral scales. The next step

is to run a new script for reading the results and plotting the integral scales. This script is given to you as `PlotResults`. In this script we use some new libraries that enable plotting information versus date and time. These libraries are `matplotlib.dates` and `datetime`.

The script first reads the results text file and assigns the results to specific vectors. The next step is to create a vector to contain the time for each measurement in seconds. This is performed by giving the half-hourly year, month, day, hour, and minute to the function `datetime.datetime().timestap()`. And finally there is another command that allows creating a vector to contain date and time in the `YYYY-MM-HH-mm-ss` format.

For each integral scale, two plots are generated. The first plot shows the time series for the quantity of interest. The second plot shows the diurnal variation of the quantity of interest. The diurnal plot overlays the quantity of interest over many days as a function of hour in the day from 0 to 23 of the Local Daylight Time zone. This helps identify which quantities exhibit a strong diurnal variation. To plot all figures simultaneously, the function `fig.show()` is used for each figure and finally the function `plt.show()` is used at the end of the script.

You need to run the analysis at least twice, once for 1 minute integration time and a second time for 2 minute integration time. After successfully running the scripts, the following figures should be obtained. Try to answer the following questions.

- On a daily average, rank the magnitude of the integral timescales from largest to smallest. Explain your observation.
- On a daily average, rank the magnitude of the integral lengthscales from largest to smallest. Explain your observation.
- Is there a relationship between the magnitudes of timescales and lengthscales?
- Which one of the scales exhibits a strong diurnal variation? Reason why it is so.
- Without reproducing the figures, what is the effect of increasing the integration time from 1 minute to 2 minutes on the integral scales calculated?
- Without reproducing the figures, increase the integration time to even a large value, such as 5 minutes. What are the effects on the integral scales calculated? If your results are not sensible, or appear to be spurious, suggest the possible reasons.

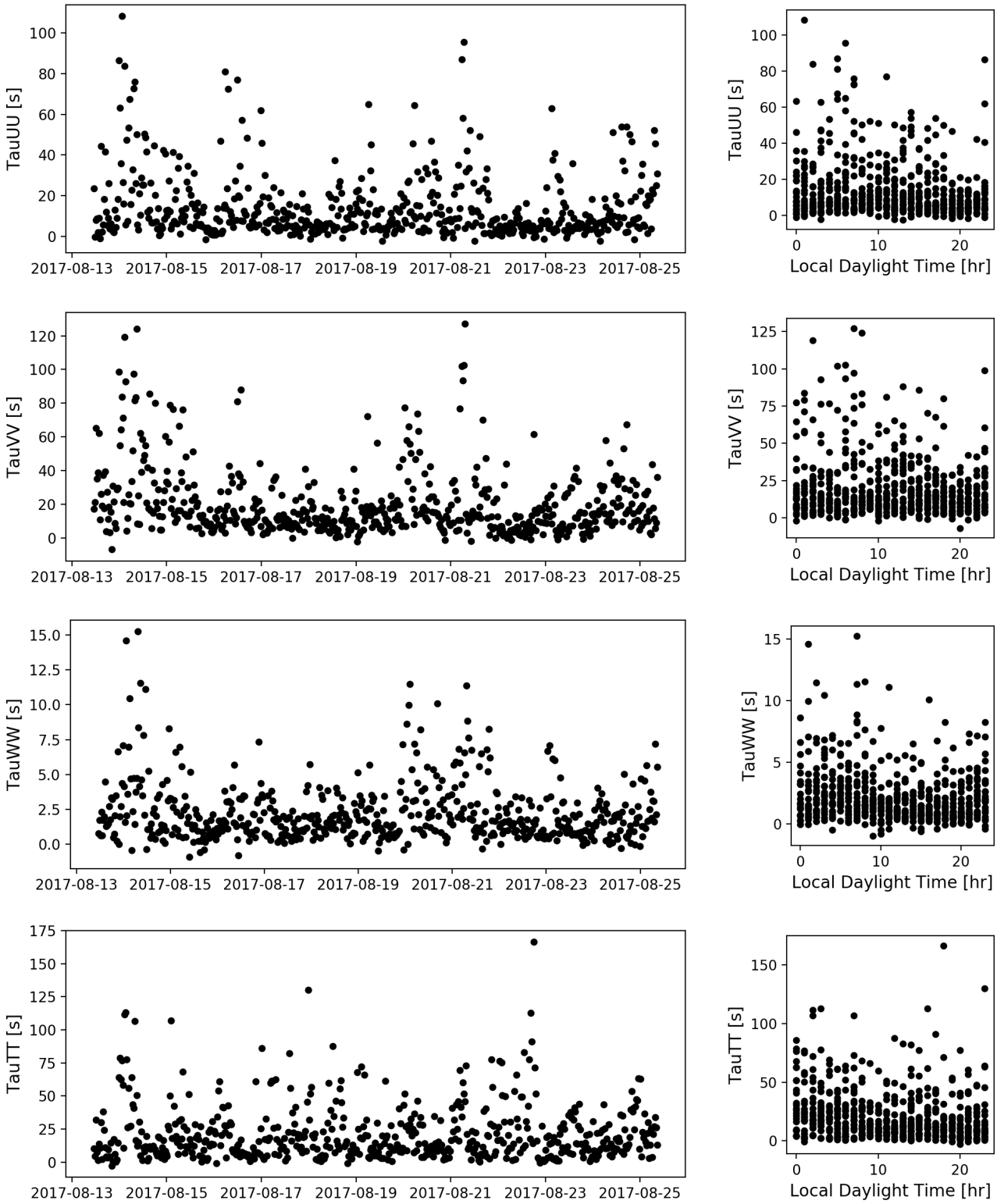


Figure 3: Integral timescales for 2 minute integration time.

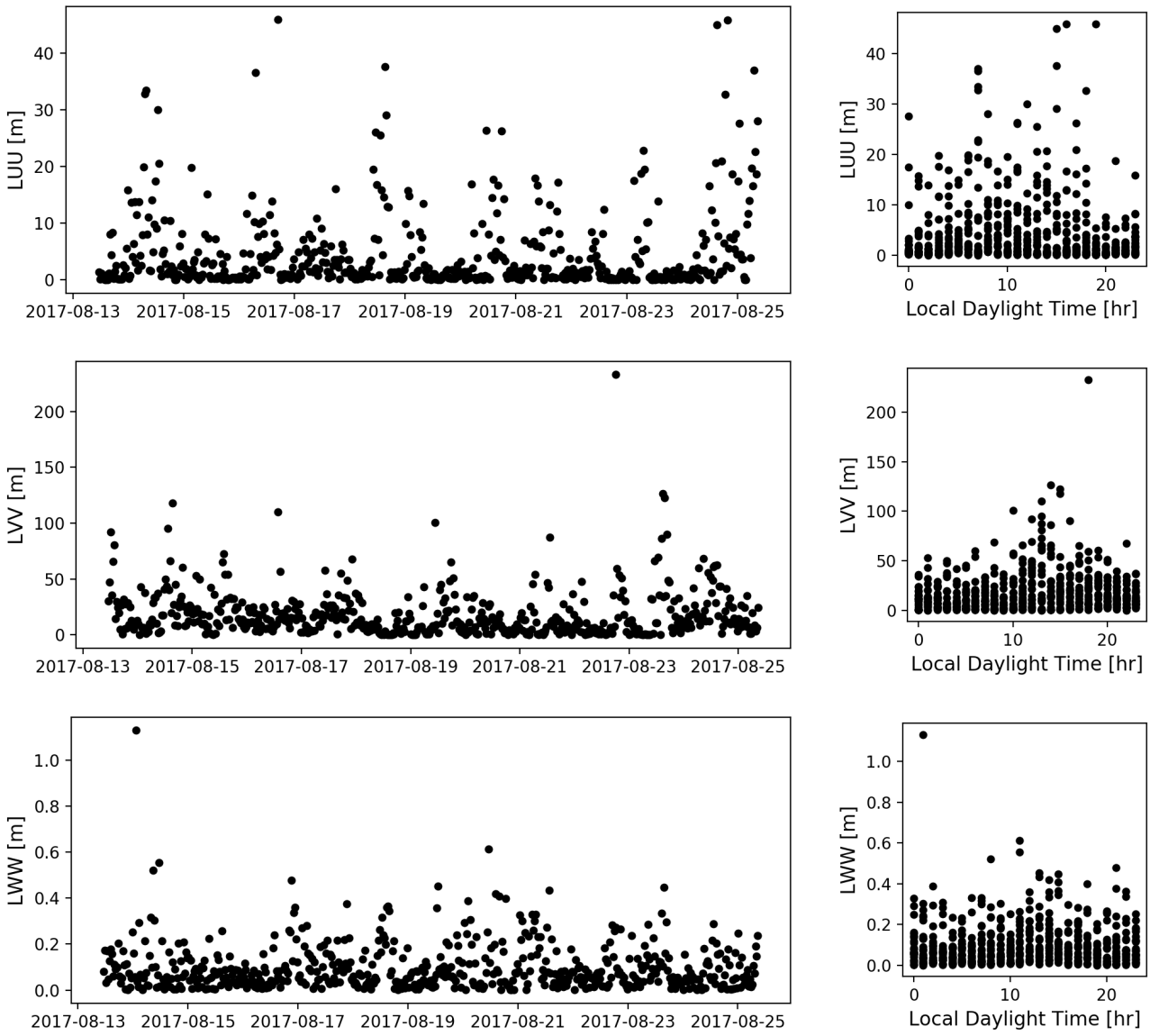


Figure 4: Integral lengthscales for 2 minute integration time.

ENGG*6790: Theory and Applications of Turbulence

Analysis Using Statistical Percentiles

Amir A. Aliabadi

November 12, 2017

1 Introduction

Environmental data are often messy and contain many gaps. As a result, instead of mean and standard deviation, sometimes other statistics such as percentiles are used. For instance the median or 50th percentile gives an estimate of the centre of data. The 25th and 75th percentiles also give an indication of data spread or variation.

In this lab, we wish to group and plot the 50th percentile for various turbulent statistics based on wind angle for airflow and diurnal time using a dataset from a micro-climate study on the campus of the University of Guelph. The campaign was conducted from August 13, 2017 to August 25, 2017. Part of the study involved installing a sonic anemometer on the roof of the Rozhanski Hall. The anemometer measured air velocity in horizontal components U, V and vertical component W in units of $\mathrm{m\,s^{-1}}$. Note that V was air velocity along canyon axis, while U was air velocity cross canyon axis. It also measured air sonic temperature T in units of K. The measurement was conducted at a sampling frequency of 4 Hz. Figure below shows the campaign site and the roof anemometer.

We wish to group data based on wind angle on the roof level and diurnal time. The mean statistics are collected at time intervals of 30 min from previous labs and will be used here. The wind angle is considered as 0 (or 360) when blowing from Northwest along the street canyon axis. The wind angle is considered positivie clockwise. Eight wind sectors are considred: N, NE, E, SE, S, SW, W, and NW. Each sector is 45 degrees wide. A particular challenge in this lab is that there is not always data available for a particular wind angle and diurnal time.

2 Python Script

Complete the following script for grouping of data. In this script, we define a file names as `"Roof30min.txt"`, `Roof4HzOnePointStatistics.txt`, and `Roof4HzIntegralScales.txt` to be read by the program. The wind angle is read from the first file name, while various turbulent

Figure 1: University of Guelph micro-climate campaign in August 2017: campaign site (top) and sonic anemometer installation on the roof of Rozhanski Hall (bottom)

statistics are read from the other two file names.

All entries for which there is no data are written as `nan` or `not a number`. Initially, when vectors are defined, they are initialized with `nan` values using the expression `numpy.nan`. The script carefully uses the modulus operator `%` to identify the proper index for various vectors or matrices to write the data into the proper element. The particular function that help us calculate percentiles of data containing `nan` values is `numpy.nanpercentile`.

```
#Data classification based on hour (equivalently Re and Ri) and wind angle
import random
import sys
import os
import numpy
import matplotlib.pyplot as plt
import matplotlib.dates as mdates
import datetime

#Define file names
#Always need Roof half-hourly data to retreive wind angle
Roof30minFileName = "Roof30min.txt"
StationOnePointStatisticsFileName = "Roof4HzOnePointStatistics.txt"
StationIntegralScalesFileName = "Roof4HzIntegralScales.txt"

#Must always load roof data

dataRoof30min = numpy.loadtxt(Roof30minFileName)

yearRoof=dataRoof30min[:,0]
monthRoof=dataRoof30min[:,1]
dayRoof=dataRoof30min[:,2]
timeHrRoof=dataRoof30min[:,3]
timeMinRoof=dataRoof30min[:,4]
RadAvgRoof=dataRoof30min[:,14]
RadStdRoof=dataRoof30min[:,15]
WDAvgRoof=dataRoof30min[:,16]
WDStdRoof=dataRoof30min[:,17]

NRoof=numpy.size(yearRoof)

#Correct for daylight saving to use time in local standard time
for i in range(0,NRoof):
    if (timeHrRoof[i] > 0):
        timeHrRoof[i] = timeHrRoof[i]-1
    else:
        timeHrRoof[i] = 23
        dayRoof[i] = dayRoof[i] - 1
```

```
#Load data for either roof

dataStationOnePointStatistics = numpy.loadtxt(StationOnePointStatisticsFileName)

UAvgStation = dataStationOnePointStatistics[:, 5]
VAvgStation = dataStationOnePointStatistics[:, 6]
SAvgStation = dataStationOnePointStatistics[:, 7]
WAvgStation = dataStationOnePointStatistics[:, 8]
TSonicAvgStation = dataStationOnePointStatistics[:, 9]
UVarStation = dataStationOnePointStatistics[:, 10]
VVarStation = dataStationOnePointStatistics[:, 11]
WVarStation = dataStationOnePointStatistics[:, 12]
TSonicVarStation = dataStationOnePointStatistics[:, 13]
kStation = dataStationOnePointStatistics[:, 14]
UVCovStation = dataStationOnePointStatistics[:, 15]
UWCovStation = dataStationOnePointStatistics[:, 16]
VWCovStation = dataStationOnePointStatistics[:, 17]
UTSonicCovStation = dataStationOnePointStatistics[:, 18]
VTSonicCovStation = dataStationOnePointStatistics[:, 19]
WTSonicCovStation = dataStationOnePointStatistics[:, 20]

dataStationIntegralScales = numpy.loadtxt(StationIntegralScalesFileName)

TauUUStation = dataStationIntegralScales[:, 9]
TauVVStation = dataStationIntegralScales[:, 10]
TauWWStation = dataStationIntegralScales[:, 11]
TauTTStation = dataStationIntegralScales[:, 12]
LUUStation = dataStationIntegralScales[:, 13]
LVVStation = dataStationIntegralScales[:, 14]
LWWStation = dataStationIntegralScales[:, 15]

#Define classified matrices
timeHr=[10,11,12,13,14,15,16,17,18,19,20,21,22,23,0,1,2,3,4,5,6,7,8,9]

NSonicHourly=int(NRoof/24)

#Create matrices for all data: 24 times and 8 wind angles
UAvgStationAll=numpy.zeros((24,8,NSonicHourly))
UAvgStationAll[:]=numpy.nan
VAvgStationAll=numpy.zeros((24,8,NSonicHourly))
VAvgStationAll[:]=numpy.nan
SAvgStationAll=numpy.zeros((24,8,NSonicHourly))
SAvgStationAll[:]=numpy.nan
WAvgStationAll=numpy.zeros((24,8,NSonicHourly))
WAvgStationAll[:]=numpy.nan
```

```
TSonicAvgStationAll=numpy.zeros((24,8,NSonicHourly))
TSonicAvgStationAll[:]=numpy.nan
UVarStationAll=numpy.zeros((24,8,NSonicHourly))
UVarStationAll[:]=numpy.nan
VVarStationAll=numpy.zeros((24,8,NSonicHourly))
VVarStationAll[:]=numpy.nan
WVarStationAll=numpy.zeros((24,8,NSonicHourly))
WVarStationAll[:]=numpy.nan
TSonicVarStationAll=numpy.zeros((24,8,NSonicHourly))
TSonicVarStationAll[:]=numpy.nan
kStationAll=numpy.zeros((24,8,NSonicHourly))
kStationAll[:]=numpy.nan
UVCovStationAll=numpy.zeros((24,8,NSonicHourly))
UVCovStationAll[:]=numpy.nan
UWCovStationAll=numpy.zeros((24,8,NSonicHourly))
UWCovStationAll[:]=numpy.nan
VWCovStationAll=numpy.zeros((24,8,NSonicHourly))
VWCovStationAll[:]=numpy.nan
UTSonicCovStationAll=numpy.zeros((24,8,NSonicHourly))
UTSonicCovStationAll[:]=numpy.nan
VTSonicCovStationAll=numpy.zeros((24,8,NSonicHourly))
VTSonicCovStationAll[:]=numpy.nan
WTSonicCovStationAll=numpy.zeros((24,8,NSonicHourly))
WTSonicCovStationAll[:]=numpy.nan
TauUUStationAll=...
TauUUStationAll[:]=...
TauVVStationAll=...
TauVVStationAll[:]=...
TauWWStationAll=...
TauWWStationAll[:]=...
TauTTStationAll=...
TauTTStationAll[:]=...
LUUStationAll=...
LUUStationAll[:]=...
LVVStationAll=...
LVVStationAll[:]=...
LWWStationAll=...
LWWStationAll[:]=...

UAvgStation50=numpy.zeros((24,8))
UAvgStation50[:]=numpy.nan
VAvgStation50=numpy.zeros((24,8))
VAvgStation50[:]=numpy.nan
SAvgStation50=numpy.zeros((24,8))
SAvgStation50[:]=numpy.nan
WAvgStation50=numpy.zeros((24,8))
```

```
WAvgStation50[:]=numpy.nan
TSonicAvgStation50=numpy.zeros((24,8))
TSonicAvgStation50[:]=numpy.nan
UVarStation50=numpy.zeros((24,8))
UVarStation50[:]=numpy.nan
VVarStation50=numpy.zeros((24,8))
VVarStation50[:]=numpy.nan
WVarStation50=numpy.zeros((24,8))
WVarStation50[:]=numpy.nan
TSonicVarStation50=numpy.zeros((24,8))
TSonicVarStation50[:]=numpy.nan
kStation50=numpy.zeros((24,8))
kStation50[:]=numpy.nan
UVCovStation50=numpy.zeros((24,8))
UVCovStation50[:]=numpy.nan
UWCovStation50=numpy.zeros((24,8))
UWCovStation50[:]=numpy.nan
VWCovStation50=numpy.zeros((24,8))
VWCovStation50[:]=numpy.nan
UTSonicCovStation50=numpy.zeros((24,8))
UTSonicCovStation50[:]=numpy.nan
VTSonicCovStation50=numpy.zeros((24,8))
VTSonicCovStation50[:]=numpy.nan
WTSonicCovStation50=numpy.zeros((24,8))
WTSonicCovStation50[:]=numpy.nan
TauUUStation50=...
TauUUStation50[:]=...
TauVVStation50=...
TauVVStation50[:]=...
TauWWStation50=...
TauWWStation50[:]=...
TauTTStation50=...
TauTTStation50[:]=...
LUUStation50=...
LUUStation50[:]=...
LVVStation50=...
LVVStation50[:]=...
LWWStation50=...
LWWStation50[:]=...

#Calculate hourly and wind angle classified data

#Classify data based on time of day and wind angle on roof
for i in range(0,NRoof):
    #Compute indices, for wind angle rotate -22.5 degrees then divide by 45. Why?
    #Always classify by roof level wind angle
```

```
    TimeIndex=int((i%48)/2)
    AngleIndex=int(((WDAvgRoof[i]+360-22.5)%360)/45)
    SampleIndex=int(i/48)*2+(i%2)
    UAvgStationAll[TimeIndex][AngleIndex][SampleIndex]=UAvgStation[i]
    VAvgStationAll[TimeIndex][AngleIndex][SampleIndex]=VAvgStation[i]
    SAvgStationAll[TimeIndex][AngleIndex][SampleIndex]=SAvgStation[i]
    WAvgStationAll[TimeIndex][AngleIndex][SampleIndex]=WAvgStation[i]
    TSonicAvgStationAll[TimeIndex][AngleIndex][SampleIndex]=TSonicAvgStation[i]
    UVarStationAll[TimeIndex][AngleIndex][SampleIndex]=UVarStation[i]
    VVarStationAll[TimeIndex][AngleIndex][SampleIndex]=VVarStation[i]
    WVarStationAll[TimeIndex][AngleIndex][SampleIndex]=WVarStation[i]
    TSonicVarStationAll[TimeIndex][AngleIndex][SampleIndex]=TSonicVarStation[i]
    kStationAll[TimeIndex][AngleIndex][SampleIndex]=kStation[i]
    UVCovStationAll[TimeIndex][AngleIndex][SampleIndex]=UVCovStation[i]
    UWCovStationAll[TimeIndex][AngleIndex][SampleIndex]=UWCovStation[i]
    VWCovStationAll[TimeIndex][AngleIndex][SampleIndex]=VWCovStation[i]
    UTSonicCovStationAll[TimeIndex][AngleIndex][SampleIndex]=UTSonicCovStation[i]
    VTSonicCovStationAll[TimeIndex][AngleIndex][SampleIndex]=VTSonicCovStation[i]
    WTSonicCovStationAll[TimeIndex][AngleIndex][SampleIndex]=WTSonicCovStation[i]
    TauUUStationAll[TimeIndex][AngleIndex][SampleIndex]=...
    TauVVStationAll[TimeIndex][AngleIndex][SampleIndex]=...
    TauWWStationAll[TimeIndex][AngleIndex][SampleIndex]=...
    TauTTStationAll[TimeIndex][AngleIndex][SampleIndex]=...
    LUUStationAll[TimeIndex][AngleIndex][SampleIndex]=...
    LVVStationAll[TimeIndex][AngleIndex][SampleIndex]=...
    LWWStationAll[TimeIndex][AngleIndex][SampleIndex]=...

#Compute percentiles for Re and Ri
for i in range(0,24):
    for j in range(0,8):
        UAvgStation50[i][j]=numpy.nanpercentile(UAvgStationAll[i][j][:],50)
        VAvgStation50[i][j]=numpy.nanpercentile(VAvgStationAll[i][j][:],50)
        SAvgStation50[i][j]=numpy.nanpercentile(SAvgStationAll[i][j][:],50)
        WAvgStation50[i][j]=numpy.nanpercentile(WAvgStationAll[i][j][:],50)
        TSonicAvgStation50[i][j]=numpy.nanpercentile(TSonicAvgStationAll[i][j][:],50)
        UVarStation50[i][j]=numpy.nanpercentile(UVarStationAll[i][j][:],50)
        VVarStation50[i][j]=numpy.nanpercentile(VVarStationAll[i][j][:],50)
        WVarStation50[i][j]=numpy.nanpercentile(WVarStationAll[i][j][:],50)
        TSonicVarStation50[i][j]=numpy.nanpercentile(TSonicVarStationAll[i][j][:],50)
        kStation50[i][j]=numpy.nanpercentile(kStationAll[i][j][:],50)
        UVCovStation50[i][j]=numpy.nanpercentile(UVCovStationAll[i][j][:],50)
        UWCovStation50[i][j]=numpy.nanpercentile(UWCovStationAll[i][j][:],50)
        VWCovStation50[i][j]=numpy.nanpercentile(VWCovStationAll[i][j][:],50)
        UTSonicCovStation50[i][j]=numpy.nanpercentile(UTSonicCovStationAll[i][j][:],50)
        VTSonicCovStation50[i][j]=numpy.nanpercentile(VTSonicCovStationAll[i][j][:],50)
        WTSonicCovStation50[i][j]=numpy.nanpercentile(WTSonicCovStationAll[i][j][:],50)
```

```
        TauUUStation50[i][j]=...
        TauVVStation50[i][j]=...
        TauWWStation50[i][j]=...
        TauTTStation50[i][j]=...
        LUUStation50[i][j]=...
        LVVStation50[i][j]=...
        LWWStation50[i][j]=...

#Plot results
#The code is given.
```

Note that the plotting script is also given to you in the same script. Upon successfully running the script, you should get the following figures.

- Explain if the effect of wind angle is evident on the median values of mean wind velocity components.
- Is there a diurnal variation on turbulent variances? When do the median values for turbulent variances peak?
- Explain if kinematic momentum fluxes are directional, i.e. depend on wind angle. Why?
- Explain if horizontal or vertical components of the kinematic heat flux are positive or negative. Why?
- How do you compare the integral timescale magnitudes as a function of directions?
- How do you compare the integral lengthscale magnitudes as a function of directions?
- Which one of the integral scales exhibit a significant diurnal variation? Why?

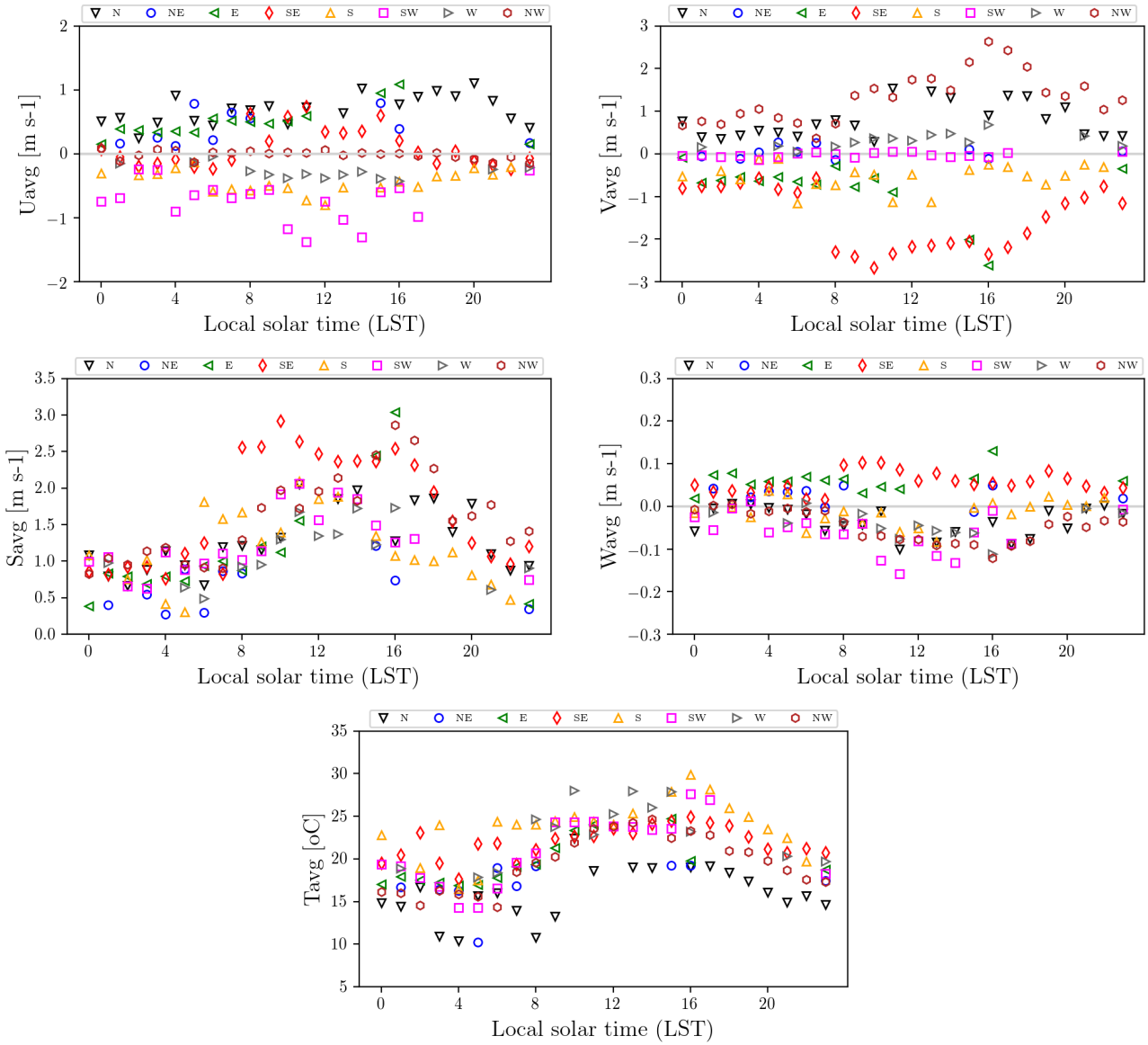


Figure 2: Medians for mean statistics grouped based on wind angle and time of day.

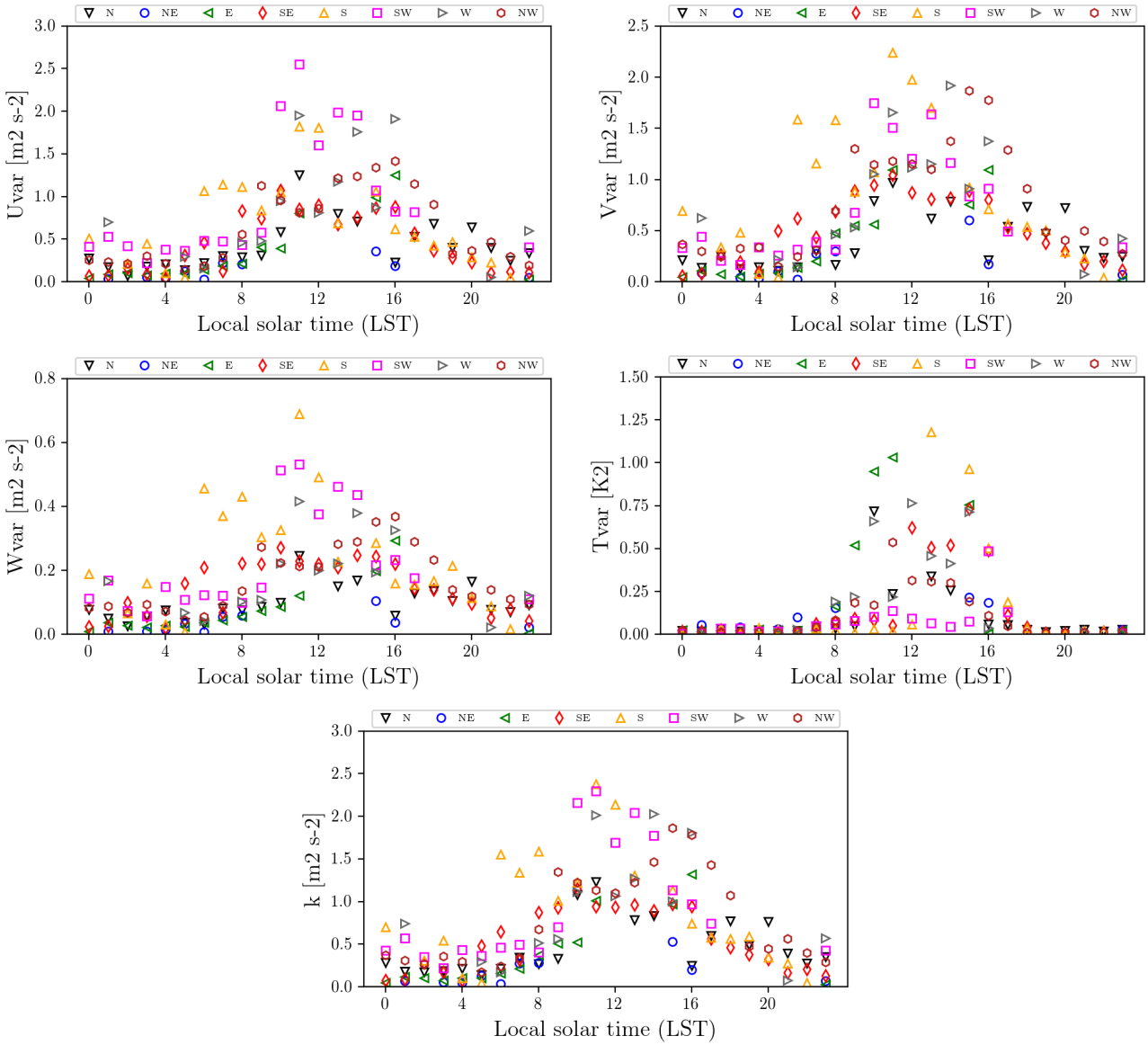


Figure 3: Medians for variance statistics grouped based on wind angle and time of day.

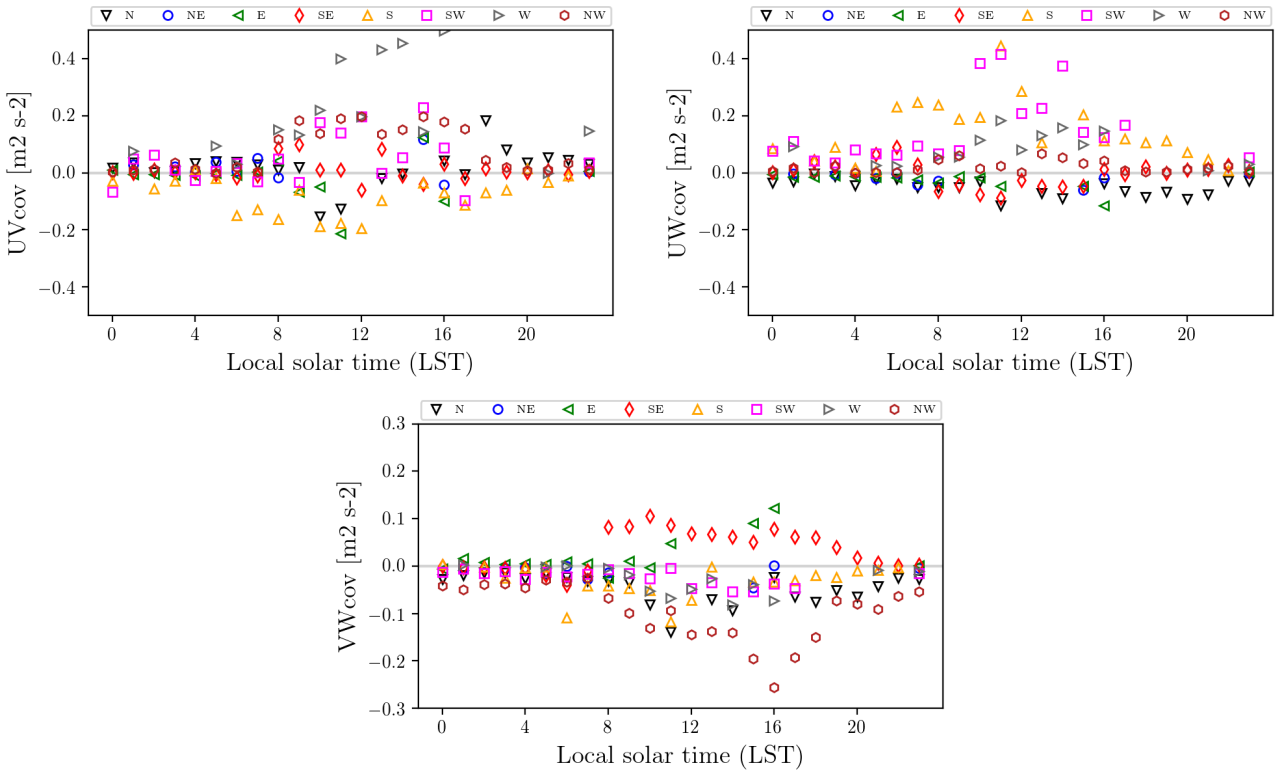


Figure 4: Medians for kinematic momentum fluxes grouped based on wind angle and time of day.

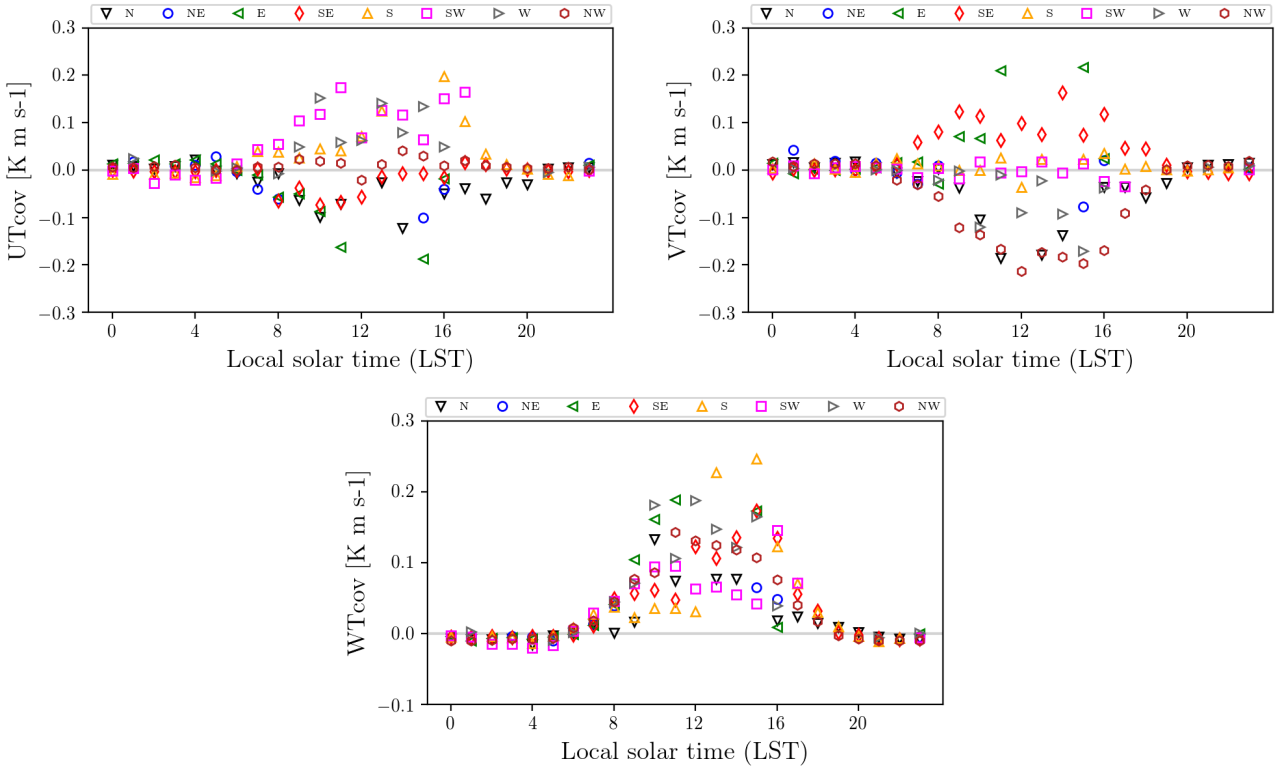


Figure 5: Medians for kinematic heat fluxes grouped based on wind angle and time of day.

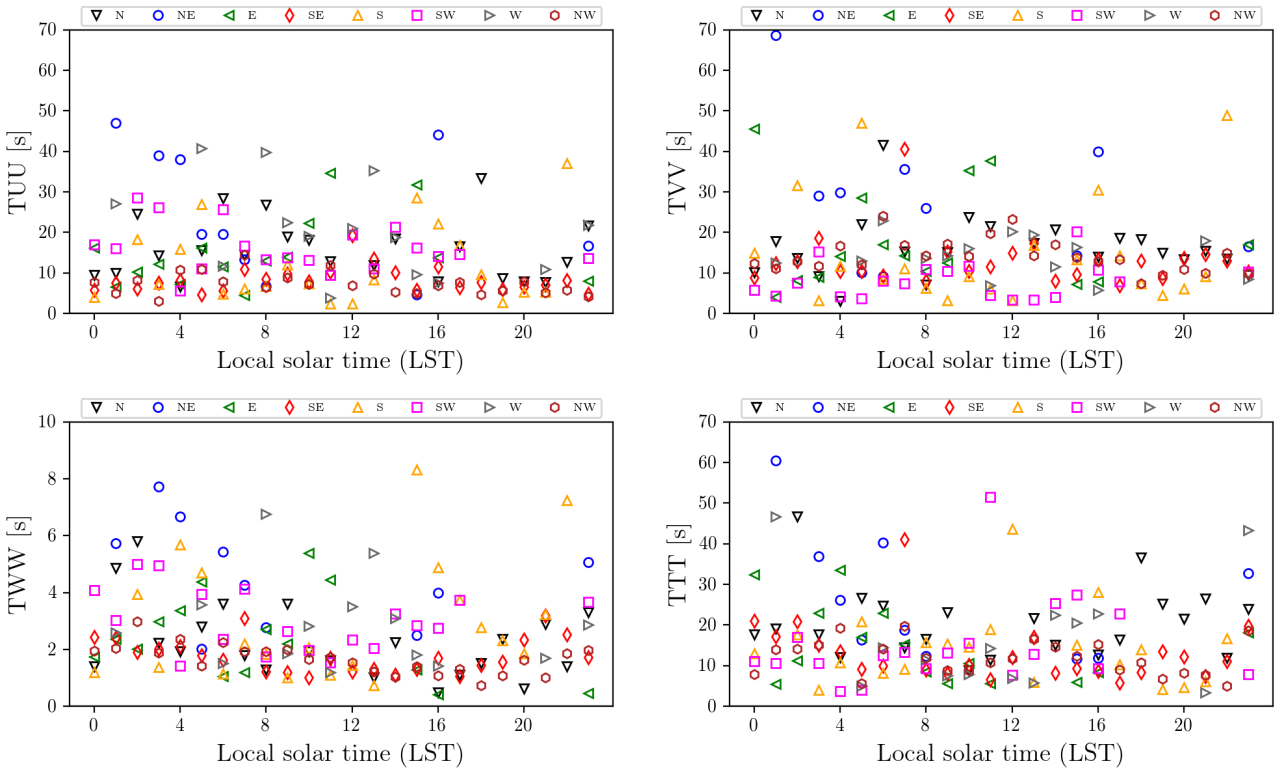


Figure 6: Medians for integral timescales grouped based on wind angle and time of day.

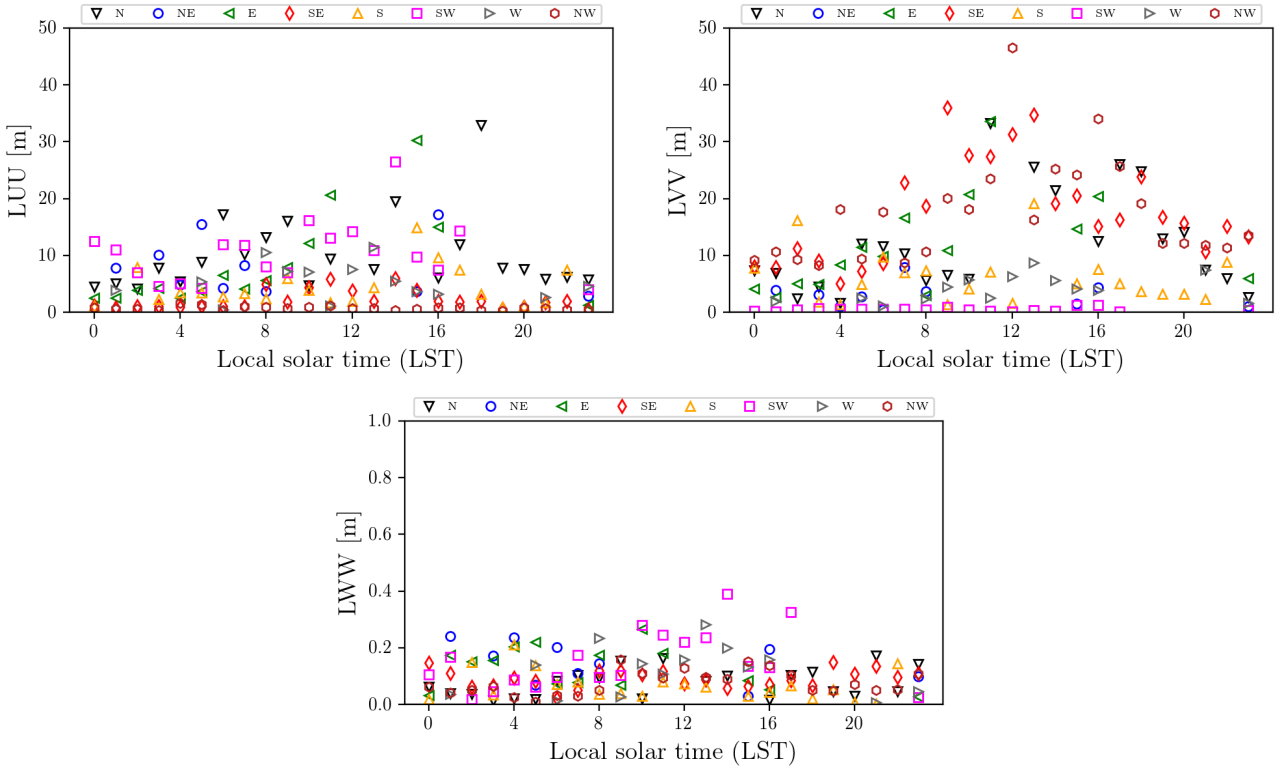


Figure 7: Medians for integral lengthscales grouped based on wind angle and time of day.

ENGG*6790: Theory and Applications of Turbulence

Discrete Fourier Transform Analysis in Time and Frequency Domains

Amir A. Aliabadi

August 20, 2018

1 Introduction

In the lectures the discrete Fourier transform analysis is introduced as a tool to represent a time series signal in the frequency domain. There are a number of ways to describe frequency:

$$n = \text{number of cycles per entire time period of signal } \mathcal{P}, \tag{1}$$

$$f = \text{number of cycles per second } = \frac{n}{\mathcal{P}} = \frac{n}{N\Delta t}, \tag{2}$$

$$\omega = \text{radians per second } = 2\pi f = \frac{2\pi n}{N\Delta t}. \tag{3}$$

A frequency of zero ($n = 0$) denotes a mean value. The *fundamental frequency*, where $n = 1$, means that exactly one wave fills the entire time period $\mathcal{P}$. Higher frequencies correspond to *harmonics* of the fundamental frequency.

Using Euler's (1707-1783) formula, $e^{ix} = \cos(x) + i\sin(x)$, as a short notation for sines and cosines, we can write the forward *Fourier-transform* to express the relationship between a time series $A(k)$ and frequency domain $F_A(n)$ using

$$F_A(n) = \sum_{k=0}^{N-1} \left[\frac{A(k)}{N}\right] e^{-i2\pi nk/N}. \tag{4}$$

In turbulence studies, it is often desired to know how much of the variance of a fluctuating time series signal is associated with a particular frequency or range of frequencies. The answer to this question is possible using the discrete Fourier transform. The square of the norm of the complex Fourier transform for any frequency n is given by

$$|F_A(n)|^2 = [F_{real}(n)]^2 + [F_{imag}(n)]^2. \tag{5}$$

When $|F_A(n)|^2$ is summed over frequencies from $n = 1$ to $N - 1$, the result equals the total biased variance of the original time series, i.e.

$$\sigma_A^2 = \frac{1}{N}\sum_{k=0}^{N-1}(A(k) - \langle A(k)\rangle_T)^2 = \sum_{n=1}^{N-1}|F_A(n)|^2 \tag{6}$$

where the time average $\langle\rangle_T$ is the only average available to us for calculating the variance. Note that the square of the norm of the complex Fourier transform is summed starting at $n = 1$ instead of $n = 0$. This is trivial since there are no turbulent fluctuations associated with $n = 0$.

We can interpret $|F_A(n)|^2$ as the portion of variance explained by waves of frequency n. For frequencies greater than the Nyquist frequency, the $|F_A(n)|^2$ values are identically equal to those at the corresponding folded lower frequencies, since the Fourier transform of high frequencies are the same as those for the low frequencies, except for a sign change in the imaginary part. Frequencies higher than the Nyquist frequency cannot be resolved by Fourier transform, therefore, $|F_A(n)|^2$ values at high frequencies should be folded back and added to those at the lower frequencies. Therefore, the *discrete spectral intensity* or *discrete spectral energy*, $E_A(n)$, is defined as $E_A(n) = 2|F_A(n)|^2$, for $n = 1$ to $n = n_f$, with N being odd, while for N being even, $E_A(n) = 2|F_A(n)|^2$ for frequencies from $n = 1$ to $n = n_f - 1$, but $E_A(n) = |F_A(n)|^2$ for $n = n_f$.

In this lab we will analyze turbulence measurements of wind speed at a high frequency of 40 Hz using an aircraft probe. The aircraft data is analyzed for a short time period associated with a slanted profile of about 200 m vertical displacement and a horizontal displacement of several hundred meters. During this time, the aircraft climbed and approximately moved in a straight line. The key concept in turbulence measurements using aircrafts is the Taylor hypothesis. When flying with an aircraft that moves several times faster than the wind speed, it is possible to assume that atmospheric turbulence is frozen–otherwise flying would be dangerous–and hence use of the Taylor hypothesis. This gives an equivalency between stationary and moving probe measurements of turbulence. For moving probes, the wavenumber can be given as

$$\kappa = \frac{2\pi}{\lambda} = \frac{2\pi f}{\overline{V_a}} = \frac{2\pi n}{\mathcal{P}\overline{V_a}} \tag{7}$$

where $\overline{V_a}$ is the average horizontal velocity of the aircraft. In this lab the aircraft data is read into `Python` from a text file. This text file lists probe measurements in various columns as a function of time. Not all columns are read in this analysis, so that only selective columns are used. These are time, wind speed in the horizontal and vertical directions, altitude, latitude, longitude, and aircraft velocity in the horizontal directions. The horizontal directions are assumed in the North and East directions, corresponding to the x and y directions respectively, while the upward vertical direction is assumed as z. Note that this system of coordinates is not right handed, but for our purposes this does not cause any problems. The number of data in the file is intentionally set as odd, so that the analysis of discrete spectral energy will be simplified.

In this lab we also try to reconstruct the components of the Reynolds stress tensor using data from both time and frequency domains. The components of the Reynolds stress tensor are given as

$$\begin{bmatrix} \langle u_1^2 \rangle & \langle u_1 u_2 \rangle & \langle u_1 u_3 \rangle \\ \langle u_2 u_1 \rangle & \langle u_2^2 \rangle & \langle u_2 u_3 \rangle \\ \langle u_3 u_1 \rangle & \langle u_3 u_2 \rangle & \langle u_3^2 \rangle \end{bmatrix}$$

2 Python Script

Complete the following script. Note that time, wind velocity toward the North, wind velocity toward the East, latitude, longitude, altitude, probe velocity toward the North, probe velocity toward the East, and the wind velocity are read from column 0, 4, 5, 6, 7, 8, 9, 10, and 16 of the text file respectively. Since the number of data points is odd, we know exactly how to calculate the Nyquist frequency.

```
import random
import sys
import os
import numpy
import matplotlib.pyplot as plt

#Load all data in a matrix
data = numpy.loadtxt("DiscreteFourierTransformData.txt")

#Extract time, t, in [hr] and then convert to [s]
t=data[:,0]
t=t*3600

#Extract wind speed blowing to North, U, and East V, in [m s^-1]
U=data[:,4]
V=data[:,5]

#Extract latitude and longitude in [deg]
Lat=data[:,6]
Lon=data[:,7]

#Extract altitude [m]
Alt=data[:,8]

#Extract probe velocity in the North and East directions [m s^-1]
ProbeVelN=data[:,9]
ProbeVelE=data[:,10]
```

```
#Extract wind speed in the vertical direction, downward positive, [m s^-1]
#Multiply by -1 so that upward is positive
W=data[:,16]
W=-W

#Extract the length of the sample, in this case it is odd
N=len(t)

#Calculate the Nyquist frequency for the odd N
nf=int((N+1)/2)

#Create frequency vector starting from 0 ending at nf
cycles=numpy.linspace(0,nf,nf+1)

#Calculate entire time period T [s]
P=t[N-1]-t[0]

#Calculate aircraft average velocity
ProbeVelNMean=numpy.mean(ProbeVelN)
ProbeVelEMean=numpy.mean(ProbeVelE)
ProbeVelMean=(ProbeVelNMean**2+ProbeVelEMean**2)**0.5

#Calculate wavenumber for a moving probe
kappa=numpy.zeros((nf+1,1))
for n in range(0, nf+1):
    kappa[n] =2*numpy.pi*cycles[n]/(P*ProbeVelMean)

#Define vectors in the frequency domain knowing that N is odd
FUreal=numpy.zeros((nf+1,1))
FUimag=numpy.zeros((nf+1,1))
FU2=numpy.zeros((nf+1,1))
EU=numpy.zeros((nf+1,1))

FVreal=numpy.zeros((nf+1,1))
FVimag=numpy.zeros((nf+1,1))
FV2=numpy.zeros((nf+1,1))
EV=numpy.zeros((nf+1,1))

FWreal=numpy.zeros((nf+1,1))
FWimag=numpy.zeros((nf+1,1))
FW2=numpy.zeros((nf+1,1))
EW=numpy.zeros((nf+1,1))

Ek=numpy.zeros((nf+1,1))
```

```
CoUV=numpy.zeros((nf+1,1))
EUV=numpy.zeros((nf+1,1))

CoUW=numpy.zeros((nf+1,1))
EUW=numpy.zeros((nf+1,1))

CoVW=numpy.zeros((nf+1,1))
EVW=numpy.zeros((nf+1,1))


#Perform a forward fourier transform for wind velocities knowing N is odd
for n in range(0, nf+1):
    sumreal=0
    sumimag=0
    for k in range(0,N):
        sumreal=sumreal+U[k]*numpy.cos(2*numpy.pi*n*k/N)
        sumimag=sumimag-U[k]*numpy.sin(2*numpy.pi*n*k/N)
    FUreal[n]=sumreal/N
    FUimag[n]=sumimag/N
    FU2[n]=(FUreal[n])**2+(FUimag[n])**2

for n in range(0, nf+1):
    sumreal=0
    sumimag=0
    for k in range(0,N):
        sumreal=...
        sumimag=...
    FVreal[n]=sumreal/N
    FVimag[n]=sumimag/N
    FV2[n]=(FVreal[n])**2+(FVimag[n])**2

for n in range(0, nf+1):
    sumreal=0
    sumimag=0
    for k in range(0,N):
        sumreal=...
        sumimag=...
    FWreal[n]=sumreal/N
    FWimag[n]=sumimag/N
    FW2[n]=(FWreal[n])**2+(FWimag[n])**2

#Calculate discrete energy spectra for individual velocities
#turbulent kinetic energy, and the co-spectra for pair of velocities
for n in range(0, nf+1):
    #Calculate spectral energy knowing that N is odd
    EU[n]=2*FU2[n]
```

```
    EV[n]=...
    EW[n]=...
    Ek[n]=0.5*(EU[n]+EV[n]+EW[n])
    CoUV[n]=FUreal[n]*FVreal[n]+FUimag[n]*FVimag[n]
    CoUW[n]=...
    CoVW[n]=...
    EUV[n]=2*CoUV[n]
    EUW[n]=2*CoUW[n]
    EVW[n]=2*CoVW[n]

#Print all components of the Reynolds stress using
#the frequency domain, remember to subtract the first term corresponding to n=0
u2meanFreq=numpy.sum(EU[1:nf+1])
uvmeanFreq=numpy.sum(EUV[1:nf+1])
uwmeanFreq=numpy.sum(EUW[1:nf+1])
uvmeanFreq=numpy.sum(EUV[1:nf+1])
v2meanFreq=numpy.sum(EV[1:nf+1])
vwmeanFreq=numpy.sum(EVW[1:nf+1])
uwmeanFreq=numpy.sum(EUW[1:nf+1])
vwmeanFreq=numpy.sum(EVW[1:nf+1])
w2meanFreq=numpy.sum(EW[1:nf+1])

ReynoldsStressFrequencyDomain=[[u2meanFreq, uvmeanFreq, uwmeanFreq],\
                [uvmeanFreq, v2meanFreq, vwmeanFreq],\
                [uwmeanFreq, vwmeanFreq, w2meanFreq]]

print("ReynoldsStressFrequencyDomain=",ReynoldsStressFrequencyDomain)

#For comparison, we now calculate and print
#all components of the Reynolds stress using the time series
UmeanTime=numpy.mean(U)
VmeanTime=numpy.mean(V)
WmeanTime=numpy.mean(W)

u=U-UmeanTime
v=V-VmeanTime
w=W-WmeanTime

u2=numpy.multiply(u,u)
v2=numpy.multiply(v,v)
w2=numpy.multiply(w,w)

u2meanTime=numpy.mean(u2)
v2meanTime=numpy.mean(v2)
w2meanTime=numpy.mean(w2)
```

```
kTime=0.5*(u2meanTime+v2meanTime+w2meanTime)

uv=numpy.multiply(u,v)
uw=numpy.multiply(u,w)
vw=numpy.multiply(v,w)

uvmeanTime=numpy.mean(uv)
uwmeanTime=numpy.mean(uw)
vwmeanTime=numpy.mean(vw)

ReynoldsStressTimeDomain=[[u2meanTime, uvmeanTime, uwmeanTime],\
                [uvmeanTime, v2meanTime, vwmeanTime],\
                [uwmeanTime, vwmeanTime, w2meanTime]]

print("ReynoldsStressTimeDomain=",ReynoldsStressTimeDomain)

#Plot the aircraft latitude versus longitude
plt.plot(Lon,Lat)
plt.xlabel('Lon [deg]')
plt.ylabel('Lat [deg]')
plt.title('Aircraft Latitude versus Longitude')
plt.show()

#Plot the aircraft altitude versus time
plt.plot(t,Alt)
plt.xlabel('t [s]')
plt.ylabel('Alt [m]')
plt.title('Aircraft Altitude versus Time')
plt.show()

#Plot the frequency and energy spectra
plt.plot(kappa,FU2,'ko')
plt.xlabel('kappa [m^-1]')
plt.ylabel('FU2 [m^2 s^-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('FU2 Frequency Spectrum')
plt.show()

plt.plot(kappa,EU,'ko')
plt.xlabel('kappa [m^-1]')
plt.ylabel('EU [m^2 s^-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('EU Discrete Energy Spectrum')
plt.show()
```

```
plt.plot(kappa,FV2,'ro')
plt.xlabel('kappa [m^-1]')
plt.ylabel('FV2 [m^2 s^-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('FV2 Frequency Spectrum')
plt.show()

plt.plot(kappa,EV,'ro')
plt.xlabel('kappa [m^-1]')
plt.ylabel('EV [m^2 s^-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('EV Discrete Energy Spectrum')
plt.show()

plt.plot(kappa,FW2,'bo')
plt.xlabel('kappa [m^-1]')
plt.ylabel('FW2 [m^2 s^-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('FW2 Frequency Spectrum')
plt.show()

plt.plot(kappa,EW,'bo')
plt.xlabel('kappa [m^-1]')
plt.ylabel('EW [m^2 s^-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('EW Discrete Energy Spectrum')
plt.show()

#Plot the discrete energy spectrum for turbulent kinetic energy
plt.plot(kappa,Ek,'go')
plt.xlabel('kappa [m^-1]')
plt.ylabel('Ek [m^2 s^-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('Ek Discrete Energy Spectrum')
plt.show()

#Plot absolute value of co-spectra and discrete energy co-spectra
plt.plot(kappa,numpy.absolute(CoUV),'ko')
plt.xlabel('kappa [m^-1]')
plt.ylabel('|CoUV| [m^2 s^-2]')
```

```
plt.xscale('log')
plt.yscale('log')
plt.title('CoUV Frequency Co-spectrum')
plt.show()

plt.plot(kappa,numpy.absolute(EUV),'ko')
plt.xlabel('kappa [m^-1]')
plt.ylabel('|EUV| [m^2 s^-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('EUV Discrete Energy Co-spectrum')
plt.show()

plt.plot(kappa,numpy.absolute(CoUW),'ro')
plt.xlabel('kappa [m^-1]')
plt.ylabel('|CoUW| [m^2 s^-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('CoUW Frequency Co-spectrum')
plt.show()

plt.plot(kappa,numpy.absolute(EUW),'ro')
plt.xlabel('kappa [m^-1]')
plt.ylabel('|EUW| [m^2 s^-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('EUW Discrete Energy Co-spectrum')
plt.show()

plt.plot(kappa,numpy.absolute(CoVW),'bo')
plt.xlabel('kappa [m^-1]')
plt.ylabel('|CoVW| [m^2 s^-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('CoVW Frequency Co-spectrum')
plt.show()

plt.plot(kappa,numpy.absolute(EVW),'bo')
plt.xlabel('kappa [m^-1]')
plt.ylabel('|EVW| [m^2 s^-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('EVW Discrete Energy Co-spectrum')
plt.show()
```

Upon running the code the latitude, longitude, and altitude positions of the aircraft are plotted

to show the coordinates of the slanted profile. This is shown in the figure below.

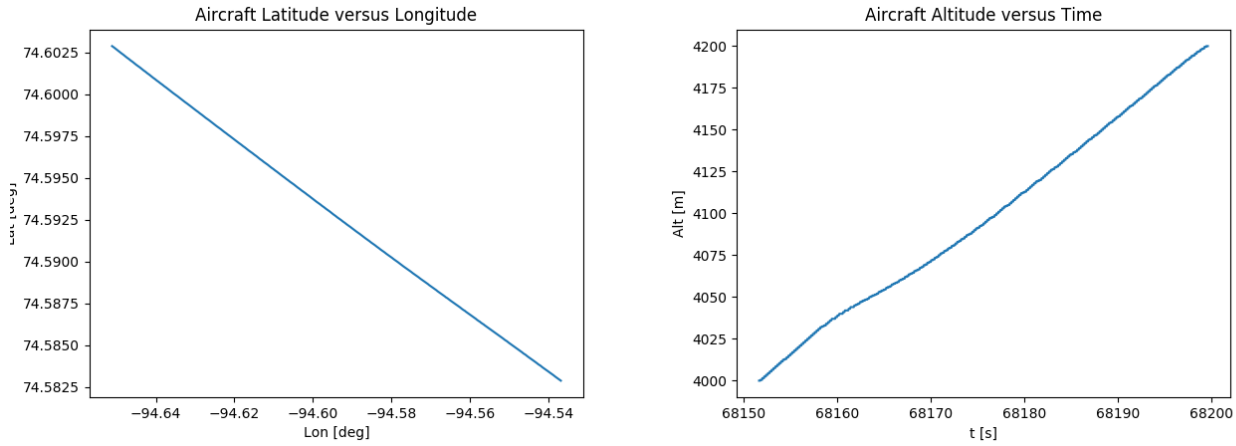


Figure 1: Plot of the aircraft movement; latitude vs. longitude (left); altitude vs. time (right).

The code also attempts to calculate all components of the Reynolds stress from both time domain and frequency domain representations of data. Note that when calculating the components of Reynolds stress from frequency domain data, one should not sum the discrete energy spectrum or co-spectrum series from $n = 0$, but one should sum these series from $n = 1$. The code should give the following results for the components of the Reynolds stress. Note that these components are remarkably close regardless of being calculated using frequency domain or time domain data. Also note that the normal stresses are always positive, while the shear stresses may be positive or negative, as indicated below.

```
ReynoldsStressFrequencyDomain=
[[0.32312158616159103, 0.044177412862565171, -0.11039639236231739],
[0.044177412862565171, 0.16474804187050252, -0.008244878812666015],
[-0.11039639236231739, -0.008244878812666015, 0.046163483303131858]]

ReynoldsStressTimeDomain=
[[0.32309356521181865, 0.044162249233617584, -0.11038926755049001],
[0.044162249233617584, 0.16472656197684166, -0.008243732828261500652],
[-0.11038926755049001, -0.008243732828261500652, 0.046161118580248905]]
```

Upon running the code one may obtain the following discrete frequency and discrete spectral energy plots from the data. Note that the plots have been drawn using wave number representation for frequency. Alternatively one could plot these spectra versus frequency f or number of cycles n per time period $\mathcal{P}$. The discrete spectral plots are more densely populated at higher wave numbers (or frequencies) than they are at lower wave numbers.

The discrete energy spectrum for the turbulent kinetic energy can be obtained by combining the discrete energy spectra for U, V, and W. As discussed in the lecture, this spectrum, once converted to energy spectrum density, vs. wave number exhibits a slop of $-5/3$ in the inertial subrange, when represented in the log-log scale. Of course, the inertial subrange only occupies a limited

region of measured wave numbers. The figure below is the resulting discrete energy spectrum for the turbulent kinetic energy.

The absolute value of discrete frequency and discrete spectral energy for co-spectra is shown in the figure below. Note that the absolute value of the co-spectra must be used so that all data can appear in a log-log plot, otherwise negative values of co-spectra cannot be shown.

Try to answer the following questions.

- Explain why when calculating the components of Reynolds stress from frequency domain data, one should not sum the discrete energy spectrum or co-spectrum series from $n = 0$, but one should sum these series from $n = 1$?
- Why are the discrete spectral plots more densely populated at higher wave numbers (or frequencies)?
- What is the difference between discrete frequency and discrete energy spectra?
- Would the shape of the spectral plots be different if they were plotted versus frequency f or number of cycles n per time period $\mathcal{P}$?
- Can you give an approximate range of wave numbers for the data set that corresponds to the inertial subrange of turbulence?

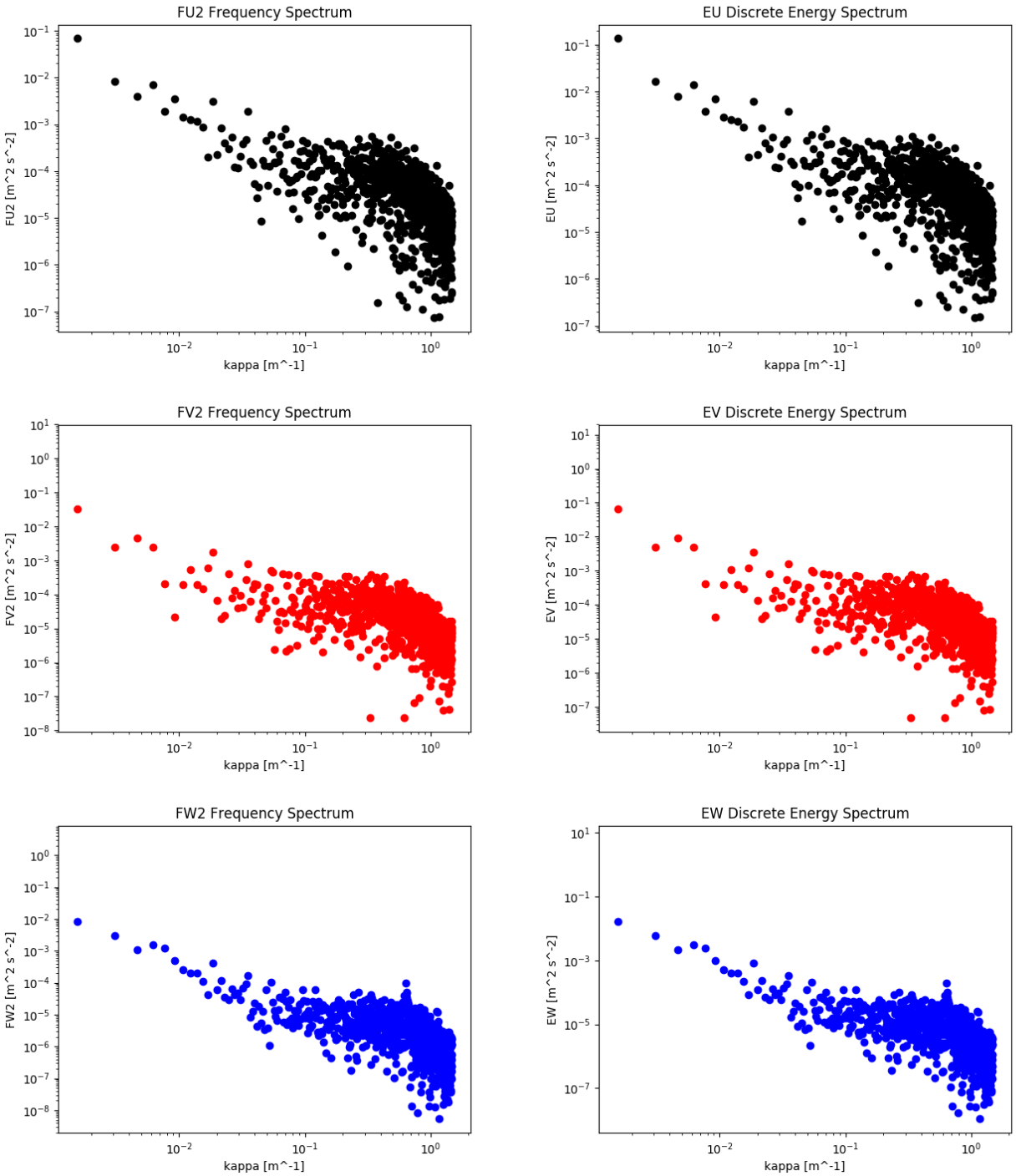


Figure 2: Plot of the discrete frequency and discrete spectral energy for U (top), V (middle), and W (bottom).

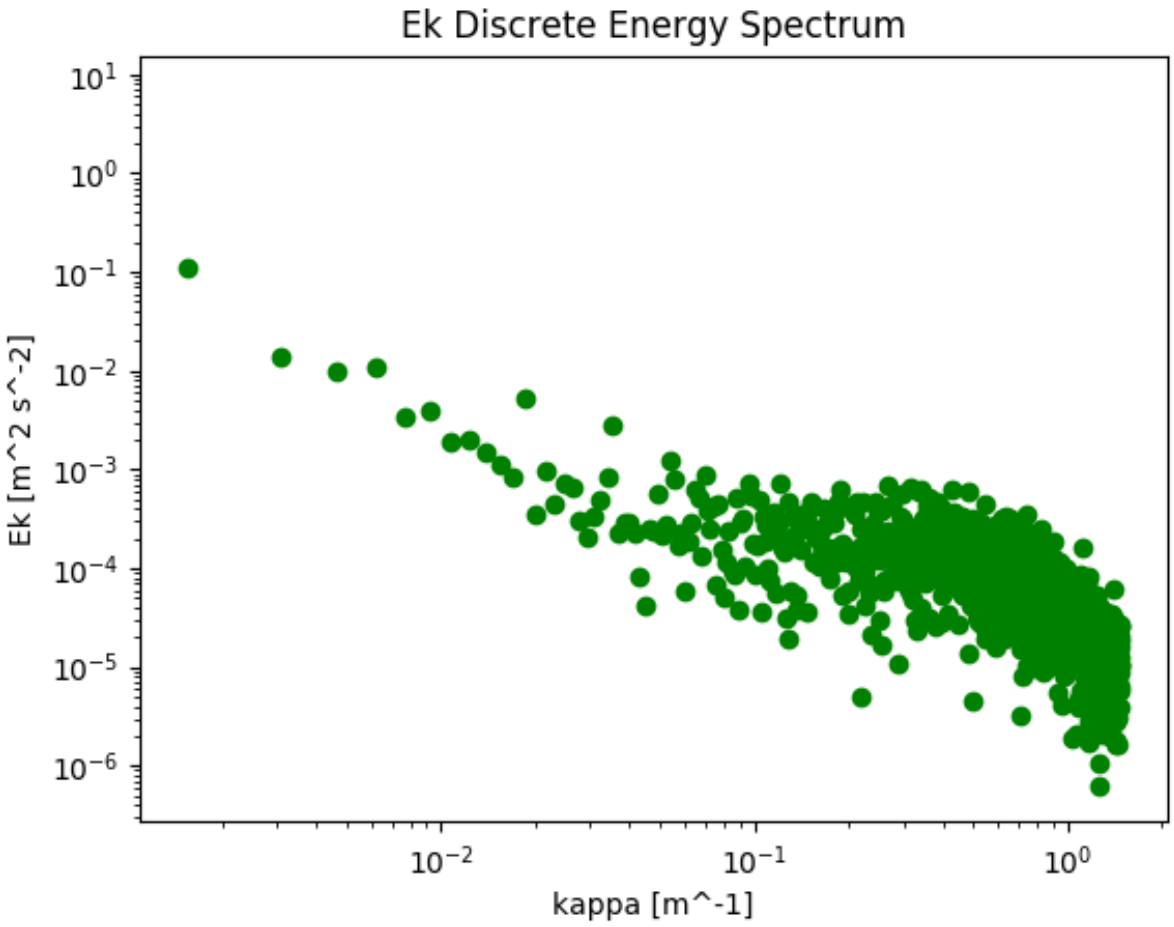


Figure 3: Plot of the discrete spectral energy for the turbulent kinetic energy.

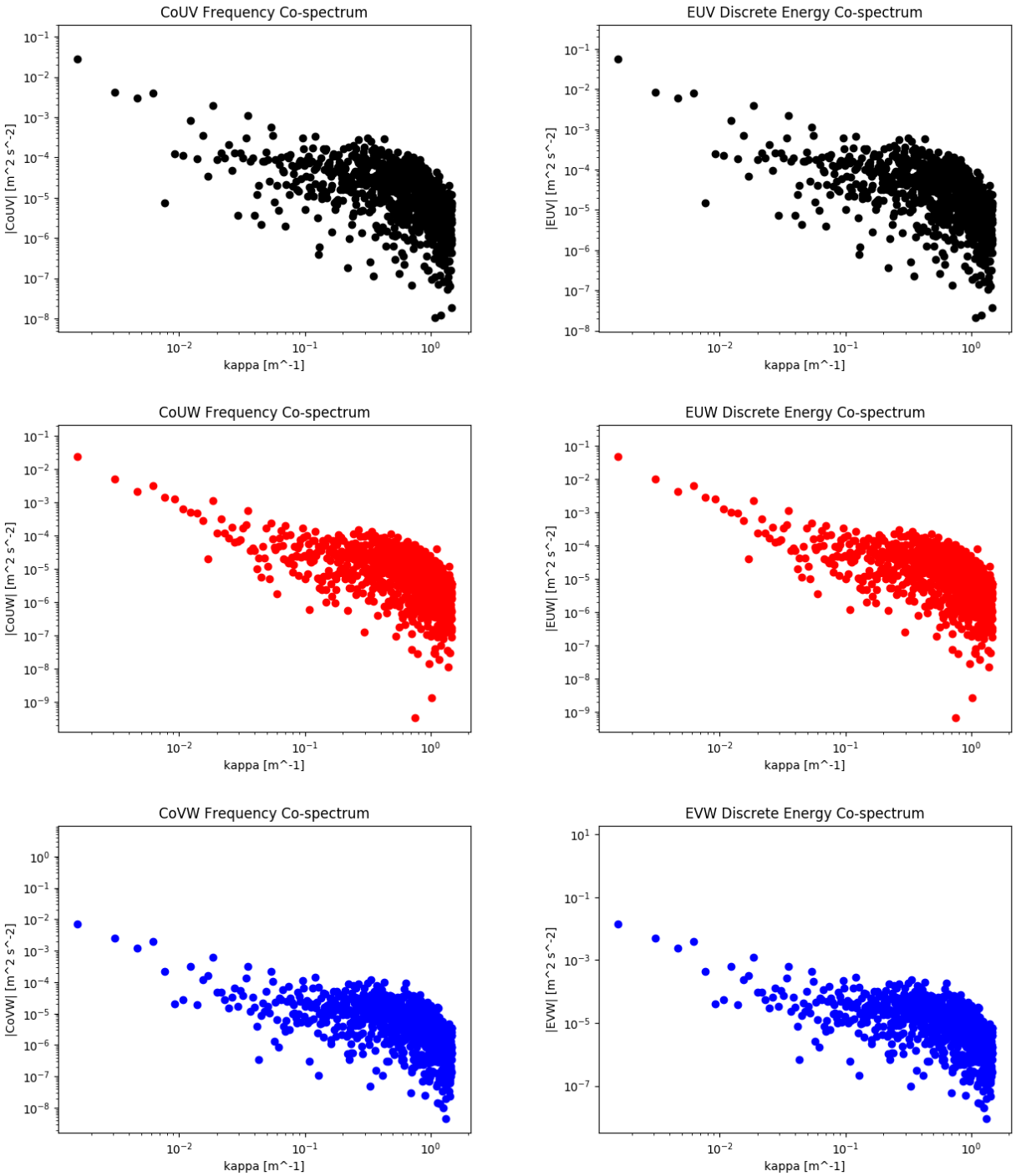


Figure 4: Plot of the absolute value of discrete frequency and discrete spectral energy for co-spectra UV (top), UW (middle), and VW (bottom).

ENGG*6790: Theory and Applications of Turbulence

1D Momentum Equation over Flat Surface with Constant Effective Viscosity

Amir A. Aliabadi

March 6, 2018

1 Introduction

In this lab we will solve the steady 1D momentum equation over flat surface with constant effective viscosity using a finite difference scheme. We assume that the flow represents atmospheric flow over flat terrain. In the lectures, using gradient-diffusion hypothesis, the momentum equation is provided as

$$\underbrace{\frac{\overline{D}}{\overline{D}t}\langle U_j\rangle}_{\text{Material Derivative of Mean}} = \underbrace{\frac{\partial}{\partial x_i}\left[\nu_{eff}\left(\frac{\partial\langle U_i\rangle}{\partial x_j}+\frac{\partial\langle U_j\rangle}{\partial x_i}\right)\right]}_{\text{Surface Forces and Reynolds Stress}} - \underbrace{\frac{1}{\rho}\frac{\partial}{\partial x_j}(\langle p\rangle+\frac{2}{3}\rho k)}_{\text{Modified Pressure}}, \tag{1}$$

The effective viscosity is the sum of the molecular viscosity and the turbulent viscosity, such that

$$\underbrace{\nu_{eff}(\mathbf{x},t)}_{\text{Effective Viscosity}} = \underbrace{\nu}_{\text{Molecular Viscosity}} + \underbrace{\nu_T(\mathbf{x},t)}_{\text{Turbulent Viscosity}} . \tag{2}$$

Suppose we use the Cartesian coordinate system with coordinate axes of x, y, and z, and velocities corresponding to these axes being $U = \langle U\rangle + u$, $V = \langle V\rangle + v$, $W = \langle W\rangle + w$, respectively, as shown in the schematic. We can assume that mean flow is only in the x direction parallel to the surface and that the direction z is normal to the surface. Under steady state conditions $\langle V\rangle = \langle W\rangle = 0$. In addition, the mean velocity $\langle U\rangle$ in the x and y directions does not change. Also assume that the modified pressure has a constant gradient in the x direction. The 1D momentum equation then simplifies to

$$0 = \nu_{eff}\frac{d^2\langle U\rangle}{dz^2} - \tau. \tag{3}$$

where τ is the constant modified pressure gradient in the x direction divided by density. Using a finite difference scheme we can assume a uniform vertical discretization of Δz and approximate

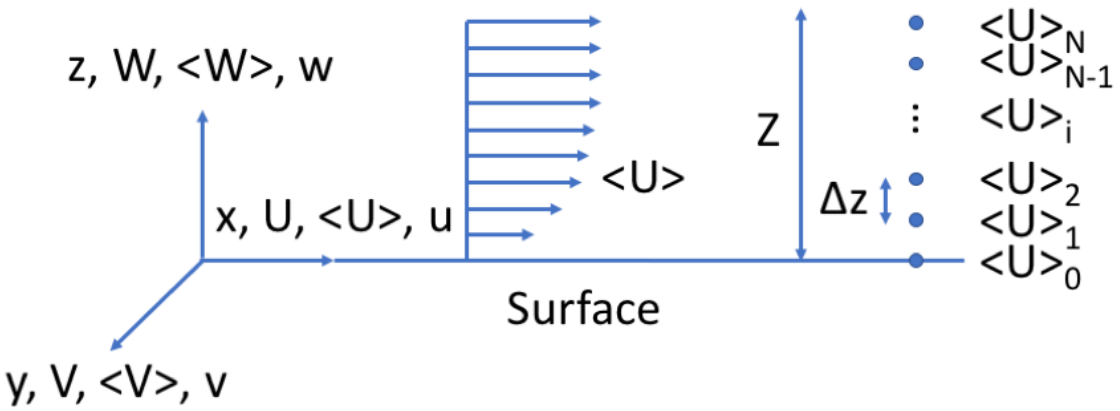


Figure 1: Schematic of 1D flow over flat surface using Cartesian coordinate system; finite difference representation of the 1D flow.

the second derivative of $\langle U \rangle$ with

$$\left(\frac{d^2\langle U\rangle}{dz^2}\right)_i \approx \frac{\langle U\rangle_{i+1} - 2\langle U\rangle_i + \langle U\rangle_{i-1}}{(\Delta z)^2}. \tag{4}$$

Therefore, the finite difference representation of the 1D momentum equation can be provided using the following equation, which can be rearranged to the following form for simplicity

$$0 = \nu_{eff}\frac{\langle U\rangle_{i+1} - 2\langle U\rangle_i + \langle U\rangle_{i-1}}{(\Delta z)^2} - \tau. \tag{5}$$

$$\langle U\rangle_{i-1} - 2\langle U\rangle_i + \langle U\rangle_{i+1} = \frac{\tau(\Delta z)^2}{\nu_{eff}} = b. \tag{6}$$

Note that $\langle U\rangle_i$ represents the velocity at node i. i varies from 0 to N and the total number of nodes are $N+1$. $\langle U\rangle_0$ represents velocity at the surface while $\langle U\rangle_N$ represents velocity at the final node. It is assumed that the vertical domain is Z. The vertical discretization can be given by $\Delta Z = Z/N$.

For the interior of the domain we can obtain $N-1$ linear equations, while there are $N+1$ unknown velocities $\langle U\rangle_i$. The extra two equations necessary to solve the linear system of equations can be obtained assuming boundary conditions. At the wall we can assume the *no-slip* boundary condition which requires

$$\langle U\rangle_0 = 0, \tag{7}$$

while at the top of the domain we can assume the *zero-gradient* boundary condition, justified by the fact that velocity far away from the surface should not change as a function of height

$$\langle U \rangle_N = \langle U \rangle_{N-1}. \tag{8}$$

The overall equations to be solved can be listed as follows, which can also be shown in matrix form for notational conciseness. The resulting $\mathbf{A}$ matrix is usually very sparse and a handful of efficient numerical techniques can be used to solve the system of equations. For instance, these equations can be solved using the Gaussian elimination technique or a simple matrix inversion if the system is small.

$$\begin{aligned}
\langle U \rangle_0 &= 0 \\
\langle U \rangle_0 - 2\langle U \rangle_1 + \langle U \rangle_2 &= b \\
\langle U \rangle_1 - 2\langle U \rangle_2 + \langle U \rangle_3 &= b \\
&\vdots \\
\langle U \rangle_{N-2} - 2\langle U \rangle_{N-1} + \langle U \rangle_N &= b \\
\langle U \rangle_N - \langle U \rangle_{N-1} &= 0.
\end{aligned} \tag{9}$$

$$\mathbf{A}\langle \mathbf{U} \rangle = \mathbf{B} \tag{10}$$

$$\begin{bmatrix}
1 & 0 & 0 & 0 & \dots & 0 & 0 & 0 \\
1 & -2 & 1 & 0 & \dots & 0 & 0 & 0 \\
0 & 1 & -2 & 1 & \dots & 0 & 0 & 0 \\
0 & 0 & 1 & -2 & \dots & 0 & 0 & 0 \\
\vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots & \vdots \\
0 & 0 & 0 & 0 & \dots & -2 & 1 & 0 \\
0 & 0 & 0 & 0 & \dots & 1 & -2 & 1 \\
0 & 0 & 0 & 0 & \dots & 0 & -1 & 1
\end{bmatrix}
\begin{bmatrix}
\langle U \rangle_0 \\ \langle U \rangle_1 \\ \langle U \rangle_2 \\ \langle U \rangle_3 \\ \vdots \\ \langle U \rangle_{N-2} \\ \langle U \rangle_{N-1} \\ \langle U \rangle_N
\end{bmatrix}
=
\begin{bmatrix}
0 \\ b \\ b \\ b \\ \vdots \\ b \\ b \\ 0
\end{bmatrix}$$

The simulation is desired for 4 combinations of τ and ν_{eff} shown in table below. Case 1 represents a low pressure gradient and low effective viscosity. Case 2 represents a high pressure gradient and low effective viscosity. Case 3 represents a low pressure gradient and high effective viscosity. Case 4 represents a high pressure gradient and high effective viscosity.

2 Python Script

Complete the following code. Note that to solve a linear system of equation, the command `numpy.linalg.solve()` can be used, which is a functionality of the `numpy` interpreter package.

Table 1: Simulation cases with varying amount of horizontal pressure gradient and and effective viscosities.

Case	τ [m s^{-2}]	ν_{eff} [m^2 s^{-1}]
1	-0.01	5
2	-0.02	5
3	-0.01	10
4	-0.02	10

```
import random
import sys
import os
import numpy
import matplotlib.pyplot as plt

#Define horizontal pressure gradient divided by density [m s^-2]
tau=-0.01

#Define effective viscosity [m^2 s^-1]
nuEff=5

#Define z axis from 0 to Z with dz increments
Z=100       #[m]
N=20
dz=Z/N      #[m]
z=numpy.linspace(0, Z, N+1)

#Define and initialize a mean velocity vector [m s^-1]
Umean=numpy.zeros((N+1,1))

#Define and initialize the A matrix
A=numpy.zeros((N+1,N+1))
A[0][0]=1
for i in range(1, N):
        A[i][i-1]=1
        A[i][i]=-2
        A[i][i+1]=1
A[N][N-1]=-1
A[N][N]=1

#Define and initialize the B vector
b=...
B=...
B[0]=...
for i in range (1, N):
    B[i]=...
```

```
B[N]=...

#Solve the linear system of equations A Umean = B
Umean = numpy.linalg.solve(A, B)

#Plot the mean velocity versus z
plt.plot(Umean,z)
plt.xlabel('<U> [m s^-1]')
plt.ylabel('z [m]')
plt.title('Mean Velocity as Function of z')
plt.show()
```

Upon completing the code. You should get the following figures. Try to answer the following questions.

- For a constant effective viscosity, what is the role of pressure gradient on wind speed?
- For a constant pressure gradient, what is the role of effective viscosity on wind speed?
- Why is the solution for cases 1 and 4 identical? try to reason by interpreting the simplified 1D momentum equation.

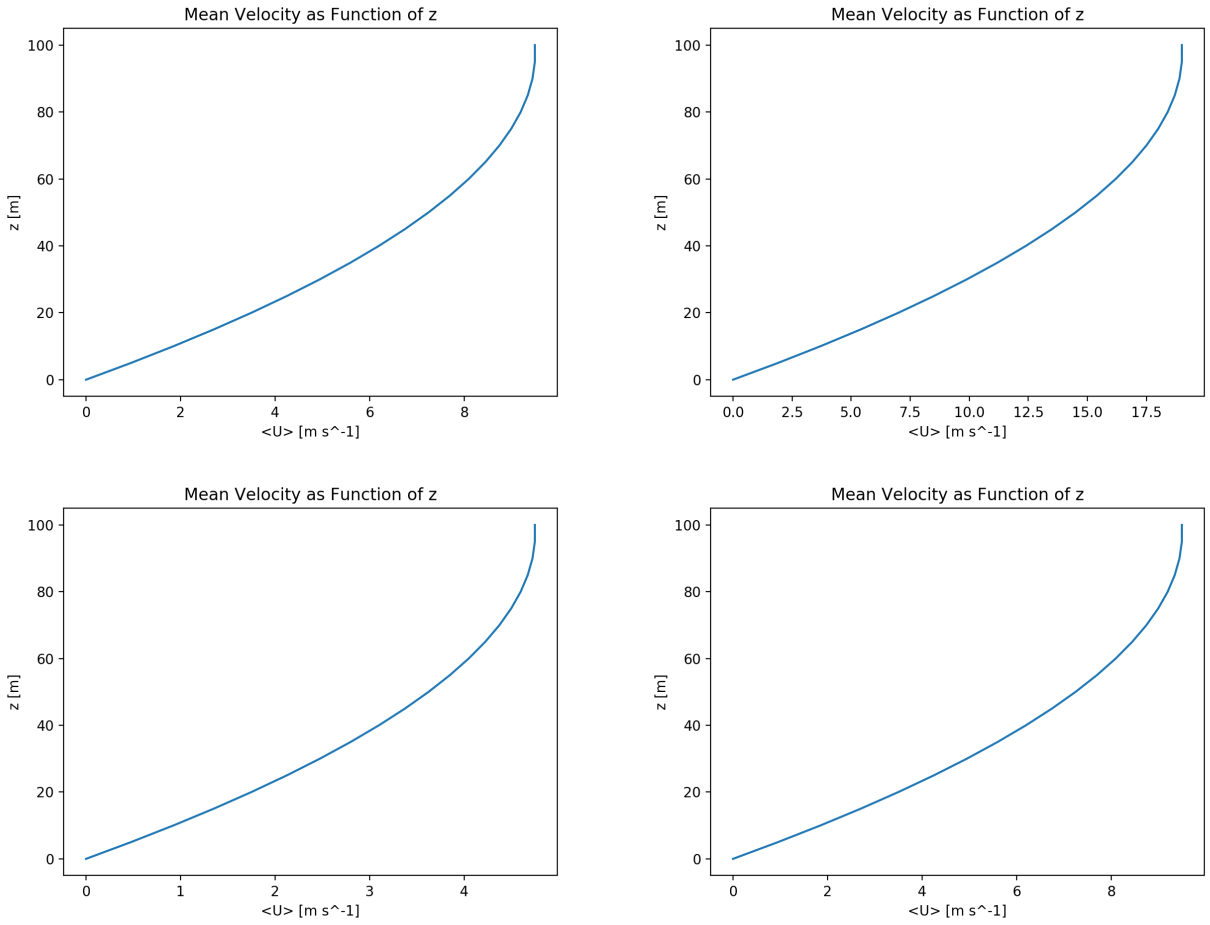


Figure 2: Case1 (top left), Case 2 (top right), Case 3 (bottom left), Case 4 (bottom right)

ENGG*6790: Theory and Applications of Turbulence

1D Momentum Equation over Flat Surface
with Effective Viscosity Formulated by a Mixing Length Model

Amir A. Aliabadi

March 8, 2018

1 Introduction

In this lab we will solve the steady 1D momentum equation over flat surface with effective viscosity formulated by mixing length using a finite difference scheme. We assume that the flow represents atmospheric flow over flat terrain. In the lectures, using gradient-diffusion hypothesis, the momentum equation is provided as

$$\underbrace{\frac{\overline{D}}{\overline{D}t}\langle U_j\rangle}_{\text{Material Derivative of Mean}} = \underbrace{\frac{\partial}{\partial x_i}\left[\nu_{eff}\left(\frac{\partial\langle U_i\rangle}{\partial x_j}+\frac{\partial\langle U_j\rangle}{\partial x_i}\right)\right]}_{\text{Surface Forces and Reynolds Stress}} - \underbrace{\frac{1}{\rho}\frac{\partial}{\partial x_j}(\langle p\rangle+\frac{2}{3}\rho k)}_{\text{Modified Pressure}}, \tag{1}$$

The effective viscosity is the sum of the molecular viscosity and the turbulent viscosity, such that

$$\underbrace{\nu_{eff}(\mathbf{x},t)}_{\text{Effective Viscosity}} = \underbrace{\nu}_{\text{Molecular Viscosity}} + \underbrace{\nu_T(\mathbf{x},t)}_{\text{Turbulent Viscosity}} . \tag{2}$$

Suppose we use the Cartesian coordinate system with coordinate axes of x, y, and z, and velocities corresponding to these axes being $U = \langle U\rangle + u$, $V = \langle V\rangle + v$, $W = \langle W\rangle + w$, respectively, as shown in the schematic. We can assume that mean flow is only in the x direction parallel to the surface and that the direction z is normal to the surface. Under steady state conditions $\langle V\rangle = \langle W\rangle = 0$. In addition, the mean velocity $\langle U\rangle$ in the x and y directions does not change. Assume that the modified pressure has a constant gradient in the x direction, and that effective viscosity can be approximate by the turbulent viscosity $\nu_{eff} \approx \nu_T$, the 1D momentum equation then simplifies to

$$0 = \frac{d}{dz}\left(\nu_T\frac{d\langle U\rangle}{dz}\right) - \tau. \tag{3}$$

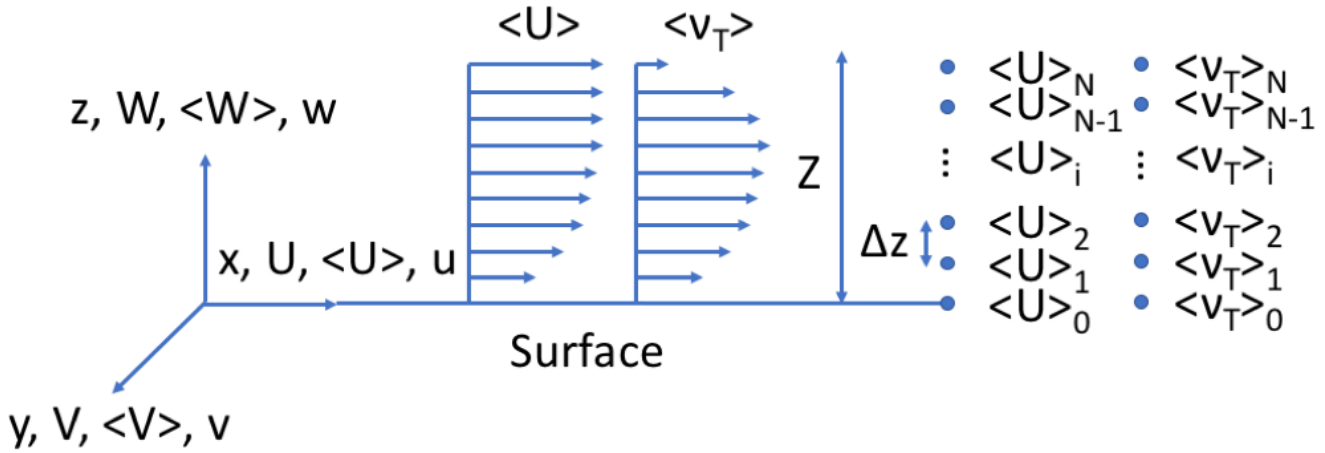


Figure 1: Schematic of 1D flow over flat surface using Cartesian coordinate system; finite difference representation of the 1D flow.

where τ is the constant modified pressure gradient in the x direction divided by density. Note that turbulent viscosity itself is a function of z so it cannot be taken out of the outer derivative. Instead the outer derivative operation should be applied such that

$$0 = \frac{d\nu_T}{dz}\frac{d\langle U\rangle}{dz} + \nu_T\frac{d^2\langle U\rangle}{dz^2} - \tau. \tag{4}$$

This momentum equation is highly nonlinear because it has a product of zeroth, first, and second order derivatives. The turbulent viscosity itself can be formulated using the mixing length model such that

$$\nu_T = \ell_m^2\left|\frac{d\langle U\rangle}{dz}\right| = \ell_m^2\frac{d\langle U\rangle}{dz} \tag{5}$$

where the absolute value around the derivative of mean velocity is removed assuming that this derivative is positive. The mixing length ℓ_m is formulated using

$$\ell_m = \frac{\kappa z}{1 + \frac{\kappa z}{\ell_0}} \tag{6}$$

where $\kappa = 0.41$ is the von Kármán constant, and ℓ_0 is the maximum mixing length. This formulation for mixing length has the nice property that it is bounded between zero and ℓ_0, which is physically sound since mixing length increases linearly in the log-law sublayer near a wall but cannot increase indefinitely in the interior of the domain. This formulation results in

$$\begin{cases} z \to 0 & \ell_m \to \kappa z \\ z \to \infty & \ell_m \to \ell_0 \end{cases}$$

So we have two equations: momentum and viscosity, which can be linearized and solved using a finite difference scheme. For notational convenience we can represent derivatives by superscripts such that

$$\begin{cases} \nu_T^{(0)} = \nu_T, \nu_T^{(1)} = \frac{d\nu_T}{dz} \\ \langle U \rangle^{(0)} = \langle U \rangle, \langle U \rangle^{(1)} = \frac{d\langle U \rangle}{dz}, \langle U \rangle^{(2)} = \frac{d^2\langle U \rangle}{dz^2} \end{cases}$$

The two equations to be solved can be formulated as follows, where each term in each equation can be renamed by a function f

$$0 = \underbrace{\nu_T^{(1)} \langle U \rangle^{(1)}}_{f_{1,mom.}} + \underbrace{\nu_T^{(0)} \langle U \rangle^{(2)}}_{f_{2,mom.}} - \tau \tag{7}$$

$$0 = \underbrace{-\nu_T^{(0)}}_{f_{1,vis.}} + \underbrace{\ell_m^2 \langle U \rangle^{(1)}}_{f_{2,vis.}} \tag{8}$$

Each f function can be replaced by its approximate using the Newton method expressing the function around an arbitrary point z_i. Beginning with the momentum equation the f function approximations are

$$\begin{aligned} f_{1,mom.} &\approx f_{1,mom.}(z_i) + \frac{\partial f_{1,mom.}}{\partial \nu_T^{(1)}}|_{z_i} \left[\nu_T^{(1)}(z) - \nu_T^{(1)}(z_i)\right] + \frac{\partial f_{1,mom.}}{\partial \langle U \rangle^{(1)}}|_{z_i} \left[\langle U \rangle^{(1)}(z) - \langle U \rangle^{(1)}(z_i)\right] \\ &= \nu_T^{(1)}(z_i)\langle U \rangle^{(1)}(z_i) + \langle U \rangle^{(1)}(z_i) \left[\nu_T^{(1)}(z) - \nu_T^{(1)}(z_i)\right] + \nu_T^{(1)}(z_i) \left[\langle U \rangle^{(1)}(z) - \langle U \rangle^{(1)}(z_i)\right] \\ &= -\nu_T^{(1)}(z_i)\langle U \rangle^{(1)}(z_i) + \nu_T^{(1)}(z_i)\langle U \rangle^{(1)}(z) + \langle U \rangle^{(1)}(z_i)\nu_T^{(1)}(z) \end{aligned} \tag{9}$$

Note that $f_{1,mom.}$ has been replaced by a linear combinations of $\nu_T^{(1)}(z)$ and $\langle U \rangle^{(1)}(z)$. In a similar fashion, $f_{2,mom.}$ can be replaced by its linearized approximation such that

$$\begin{aligned} f_{2,mom.} &\approx f_{2,mom.}(z_i) + \frac{\partial f_{2,mom.}}{\partial \nu_T^{(0)}}|_{z_i} \left[\nu_T^{(0)}(z) - \nu_T^{(0)}(z_i)\right] + \frac{\partial f_{2,mom.}}{\partial \langle U \rangle^{(2)}}|_{z_i} \left[\langle U \rangle^{(2)}(z) - \langle U \rangle^{(2)}(z_i)\right] \\ &= \nu_T^{(0)}(z_i)\langle U \rangle^{(2)}(z_i) + \langle U \rangle^{(2)}(z_i) \left[\nu_T^{(0)}(z) - \nu_T^{(0)}(z_i)\right] + \nu_T^{(0)}(z_i) \left[\langle U \rangle^{(2)}(z) - \langle U \rangle^{(2)}(z_i)\right] \\ &= -\nu_T^{(0)}(z_i)\langle U \rangle^{(2)}(z_i) + \nu_T^{(0)}(z_i)\langle U \rangle^{(2)}(z) + \langle U \rangle^{(2)}(z_i)\nu_T^{(0)}(z) \end{aligned} \tag{10}$$

Note that $f_{2,mom.}$ has been replaced by a linear combinations of $\nu_T^{(0)}(z)$ and $\langle U \rangle^{(2)}(z)$. Therefore, the momentum equation can be replaced by the following equation

$$c_1\langle U\rangle^{(1)} + c_2\langle U\rangle^{(2)} + c_3\nu_T^{(0)} + c_4\nu_T^{(1)} = c_b \tag{11}$$

where c_1, c_2, c_3, c_4, and c_b, are constants that can be calculated using the linearized equations above such that

$$\begin{cases} c_1 = \nu_T^{(1)}(z_i) \\ c_2 = \nu_T^{(0)}(z_i) \\ c_3 = \langle U\rangle^{(2)}(z_i) \\ c_4 = \langle U\rangle^{(1)}(z_i) \\ c_b = -\left[-\nu_T^{(1)}(z_i)\langle U\rangle^{(1)}(z_i) - \nu_T^{(0)}(z_i)\langle U\rangle^{(2)}(z_i) - \tau\right] \end{cases}$$

We can now focus our attention on linearizing the turbulent viscosity equation. The first term is already linearized so we do not need to apply the Newton's method, i.e.

$$f_{1,vis.} = -\nu_T^{(0)}(z) \tag{12}$$

The second term $f_{2,vis.}$ can be linearized conveniently. Here we can treat ℓ_m as a constant, after all, it can be simply calculated for any z_i

$$\begin{aligned} f_{2,vis.} &\approx f_{2,vis.}(z_i) + \frac{df_{2,vis.}}{d\langle U\rangle^{(1)}}|_{z_i}\left[\langle U\rangle^{(1)}(z) - \langle U\rangle^{(1)}(z_i)\right] \\ &= \ell_m^2(z_i)\langle U\rangle^{(1)}(z_i) + \ell_m^2(z_i)\left[\langle U\rangle^{(1)}(z) - \langle U\rangle^{(1)}(z_i)\right] \\ &= \ell_m^2(z_i)\langle U\rangle^{(1)}(z) \end{aligned} \tag{13}$$

Therefore, the turbulent viscosity equation can be replaced by the following equation

$$d_1\nu_T^{(0)} + d_2\langle U\rangle^{(1)} = d_b \tag{14}$$

where d_1, d_2, and d_b, are constants that can be calculated using the linearized equations above such that

$$\begin{cases} d_1 = -1 \\ d_2 = \ell_m^2(z_i) \\ d_b = 0 \end{cases}$$

Now consider that we want to represent the linearized momentum and turbulent viscosity equations using finite differences. Consider a vertical discretization Δz as shown in the figure. Using central differencing the derivatives can be replaced by values at indices $i-1$, i, and $i+1$ such that

$$\frac{c_1}{2\Delta z}(\langle U\rangle_{i+1} - \langle U\rangle_{i-1}) + \frac{c_2}{(\Delta z)^2}(\langle U\rangle_{i+1} - 2\langle U\rangle_i + \langle U\rangle_{i-1}) + c_3\nu_{T,i} + \frac{c_4}{2\Delta z}(\nu_{T,i+1} - \nu_{T,i-1}) = c_b \quad (15)$$

$$d_1\nu_{T,i} + \frac{d_2}{2\Delta z}(\langle U\rangle_{i+1} - \langle U\rangle_{i-1}) = d_b \quad (16)$$

As can bee seen the unknowns $\langle U\rangle_i$ and $\nu_{T,i}$ appear in both discretized momentum and turbulent viscosity equations. Therefore, these equations should be combined to arrive at a linear system of equations and subsequently solved using a linear algebra solver. Let us define the unknowns vector $\mathbf{X}$ such that

$$\mathbf{X} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \\ x_N \\ x_{N+1} \\ x_{N+2} \\ \vdots \\ x_{2N} \\ x_{2N+1} \end{bmatrix} = \begin{bmatrix} \langle U\rangle_0 \\ \langle U\rangle_1 \\ \vdots \\ \langle U\rangle_{N-1} \\ \langle U\rangle_N \\ \nu_{T,0} \\ \nu_{T,1} \\ \vdots \\ \nu_{T,N-1} \\ \nu_{T,N} \end{bmatrix}$$

As can be seen the first half of vector $\mathbf{X}$ contains the $\langle U\rangle_i$ solutions and the second half of vector $\mathbf{X}$ contains the $\nu_{T,i}$ solutions. Note that $\nu_{T,i}$ maps to x_{N+1+i}. Now we need $2N+2$ linear equations to solve for $\mathbf{X}$.

$$\begin{cases} \text{Equation 0: } a_{0,0}x_0 + a_{0,1}x_1 + ... + a_{0,2N}x_{2N} + a_{0,2N+1}x_{2N+1} = b_0 \\ \text{Equation 1: } a_{1,0}x_0 + a_{1,1}x_1 + ... + a_{1,2N}x_{2N} + a_{1,2N+1}x_{2N+1} = b_1 \\ ... \\ \text{Equation } i\text{: } a_{i,0}x_0 + a_{i,1}x_1 + ... + a_{i,2N}x_{2N} + a_{i,2N+1}x_{2N+1} = b_i \\ ... \\ \text{Equation } 2N\text{: } a_{2N,0}x_0 + a_{2N,1}x_1 + ... + a_{2N,2N}x_{2N} + a_{2N,2N+1}x_{2N+1} = b_{2N} \\ \text{Equation } 2N+1\text{: } a_{2N+1,0}x_0 + a_{2N+1,1}x_1 + ... + a_{2N+1,2N}x_{2N} + a_{2N+1,2N+1}x_{2N+1} = b_{2N+1} \end{cases}$$

Our next task is to identify $a_{i,j}$ and b_i. These can be inferred from the discretized momentum and turbulent viscosity equations. $a_{i,j}$ are mostly zero except for where there is a non-zero coefficient in the corresponding equations. The first N equations (equation 0, equation 1, ... equation N) are the momentum equations. The boundary condition for $\langle U\rangle$ at the surface provides the zeroth equation, i.e.

$$\begin{cases} x_0 = 0 \\ a_{0,0} = 1 \\ b_0 = 0 \end{cases}$$

The next $i : 1 \rightarrow N-1$ equations correspond to the momentum equation in the interior of the domain, so the coefficients can be obtained as follows

$$\begin{cases} \left(\frac{-c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2}\right) x_{i-1} + \left(\frac{-2c_2}{(\Delta z)^2}\right) x_i + \left(\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2}\right) x_{i+1} + \\ \left(\frac{-c_4}{2\Delta z}\right) x_{i-1+N+1} + c_3 x_{i+N+1} + \left(\frac{c_4}{2\Delta z}\right) x_{i+1+N+1} = c_b \\ a_{i,i-1} = \frac{-c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2} \\ a_{i,i} = \frac{-2c_2}{(\Delta z)^2} \\ a_{i,i+1} = \frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2} \\ a_{i,i-1+N+1} = \frac{-c_4}{2\Delta z} \\ a_{i,i+N+1} = c_3 \\ a_{i,i+1+N+1} = \frac{c_4}{2\Delta z} \\ b_i = c_b \end{cases}$$

Note that where the turbulent viscosity term appears the index is shifted by $N+1$. The boundary condition for $\langle U \rangle$ at the top of the domain provides the N^{th} equation, i.e.

$$\begin{cases} x_{N-1} - x_N = 0 \\ a_{N,N-1} = 1 \\ a_{N,N} = -1 \\ b_N = 0 \end{cases}$$

The boundary condition for ν_T at the surface provides the $N + 1^{th}$ equation, i.e.

$$\begin{cases} x_{N+1} = 0 \\ a_{N+1,N+1} = 1 \\ b_{N+1} = 0 \end{cases}$$

The next $i : N+2 \rightarrow 2N$ equations correspond to the turbulent viscosity equation in the interior of the domain, so the coefficients can be obtained as follows

$$\begin{cases} \left(\frac{-d_2}{2\Delta z}\right) x_{i-1-(N+1)} + \left(\frac{d_2}{2\Delta z}\right) x_{i+1-(N+1)} + d_1 x_i = d_b \\ a_{i,i-1-(N+1)} = \frac{-d_2}{2\Delta z} \\ a_{i,i+1-(N+1)} = \frac{d_2}{2\Delta z} \\ a_{i,i} = d_1 \\ b_i = d_b \end{cases}$$

Note that where the momentum term appears the index is shifted by $-(N+1)$. The boundary condition for ν_T at the top of the domain provides the $2N+1^{th}$ equation. Knowing that the velocity gradient at this boundary is zero, the turbulent viscosity must also be zero, i.e.

$$\begin{cases} x_{2N+1} = 0 \\ a_{2N+1,2N+1} = 1 \\ b_{2N+1} = 0 \end{cases}$$

Finally, we have arrived at linear system of equations that can be solved to provide the unknowns. This system is given as follows

$$\mathbf{AX} = \mathbf{B} \tag{17}$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,2N} & a_{0,2N+1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,2N} & a_{1,2N+1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{2N,0} & a_{2N,1} & \dots & a_{2N,2N} & a_{2N,2N+1} \\ a_{2N+1,0} & a_{2N+1,1} & \dots & a_{2N+1,2N} & a_{2N+1,2N+1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{2N} \\ x_{2N+1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{2N} \\ b_{2N+1} \end{bmatrix}$$

Note that the coefficients identified, themselves, depend on the solution. As a result the solution to the system of equations above must be found iteratively. That is, an initial guess for the solution vector $\mathbf{X}$ must be assumed. Then the system of equations must be solved iteratively. After each iteration the solutions must be updated. Subsequently, the system of equations must be solved again, until the difference between successive solutions is less than a specified error. For this purpose, the maximum norm can be considered. Suppose that a relative error of $Err = 0.01$ is specified for either the momentum or turbulent viscosity solution. Also suppose the x_i and $x_i^{(new)}$ represent two successive solutions for a specific point. The iteration can be stopped if the following conditions are met

$$\begin{cases} L_{\infty,mom.} = \max\left(\left|\frac{x_0^{(new)} - x_0}{x_0}\right|, \left|\frac{x_1^{(new)} - x_1}{x_1}\right|, ..., \left|\frac{x_N^{(new)} - x_N}{x_N}\right| \right) < Err \\ L_{\infty,vis.} = \max\left(\left|\frac{x_{N+1}^{(new)} - x_{N+1}}{x_{N+1}}\right|, \left|\frac{x_{N+2}^{(new)} - x_{N+2}}{x_{N+2}}\right|, ..., \left|\frac{x_{2N+1}^{(new)} - x_{2N+1}}{x_{2N+1}}\right| \right) < Err \end{cases}$$

The simulation is desired for 4 combinations of τ and ℓ_0 shown in table below. Case 1 represents a low pressure gradient and small mixing length. Case 2 represents a high pressure gradient and small mixing length. Case 3 represents a low pressure gradient and large mixing length. Case 4 represents a high pressure gradient and large mixing length.

Table 1: Simulation cases with varying amount of horizontal pressure gradient and mixing length.

Case	τ [m s^{-2}]	ℓ_0 [m]
1	-0.005	10
2	-0.01	10
3	-0.005	20
4	-0.01	20

2 Python Script

Complete the following code. Note that since we are solving the system of linear equations iteratively, we must initialize our solution. Conditional statements in `Python` are similar to other languages, i.e. `if` and `else` syntax is used. The conditional statement used in the loop checks if both relative errors are less than the maximum error defined in the code. If the condition is met the loop is broken using the `break` command. After the execution of the loop, the relative errors are checked again and solutions are plotted only if these errors are less than the maximum error defined.

```
import random
import sys
import os
import numpy
import matplotlib.pyplot as plt

#Define horizontal pressure gradient divided by density [m s^-2]
tau=-0.005

#Define von Karman constant
kappa=0.41

#Define maximum mixing length [m]
l0=10

#Define maximum iteration number
MaxIter=100

#Define relative error
Err=0.01
```

```
#Define z axis from 0 to Z with dz increments
Z=100       #[m]
N=50
dz=Z/N      #[m]
z=numpy.linspace(0, Z, N+1)

#Define and initialize a mean velocity vector [m s^-1]
Uinitial=1
Umean=numpy.zeros((N+1,1))
Umean[:]=Uinitial

#Define and initialize turbulent viscosity [m^2 s^-1]
nuTinitial=1
nuT=numpy.zeros((N+1,1))
nuT[:]=nuTinitial

#Define unknown vector X, coefficient matrix A, and vector B, in AX=B
x=numpy.zeros((2*N+2,1))
xnew=numpy.zeros((2*N+2,1))
b=numpy.zeros((2*N+2,1))
a=numpy.zeros((2*N+2,2*N+2))

#Initialize solution vector X
#This is a short syntax for for loop
x[0:N+1]=Umean[0:N+1]
x[N+1:2*N+2]=nuT[0:N+1]

for iter in range(1, MaxIter):

    #Momentum equations
    #i = 0
    a[0][0]=1
    b[0]=0
    #i = 1 to N-1
    for i in range(1, N):
        #Calculate derivatives by finite differences for the current i index
        #Remember to shift indices by N+1 if needed
        nuT0=x[i+N+1]
        nuT1=(x[i+1+N+1]-x[i-1+N+1])/(2*dz)
        Umean1=(x[i+1]-x[i-1])/(2*dz)
        Umean2=(x[i+1]-2*x[i]+x[i-1])/(dz**2)
        #Set constants necessary to build the coefficient matrix
        c1=nuT1
        c2=nuT0
        c3=Umean2
        c4=Umean1
```

```
    cb=-(-nuT1*Umean1-nuT0*Umean2-tau)
    #Set the coefficient matrix and the B vector
    a[i][i-1]=-c1/(2*dz)+c2/(dz**2)
    a[i][i]=-2*c2/(dz**2)
    a[i][i+1]=c1/(2*dz)+c2/(dz**2)
    a[i][i-1+N+1]=-c4/(2*dz)
    a[i][i+N+1]=c3
    a[i][i+1+N+1]=c4/(2*dz)
    b[i]=cb
#i = N
a[N][N-1]=1
a[N][N]=-1
b[N]=0

#Turbulent viscosity equations
#i = N+1
a[N+1][N+1]=...
b[N+1]=...
#i = N+2 to 2N
for i in range(N+2, 2*N+1):
    #Here there are no derivatives to calculate for the current i index
    #Set constants necessary to build the coefficient matrix
    #Shift indices by -(N+1) if needed
    d1=-1
    d2=(kappa*z[i-(N+1)]/(1+(kappa*z[i-(N+1)])/l0))**2
    db=0
    #Set the coefficient matrix and the B vector
    a[i][i-1-(N+1)]=...
    a[i][i+1-(N+1)]=...
    a[i][i]=...
    b[i]=...
#i = 2N+1
a[2*N+1][2*N+1]=...
b[2*N+1]=...

xnew=numpy.linalg.solve(a,b)

#Calculate maximum norm errors for both momentum and turbulent viscosity
ErrUmean=numpy.max(numpy.abs(numpy.divide(xnew[1:N+1]-x[1:N+1],x[1:N+1])))
ErrnuT=...

if ErrUmean < Err and ErrnuT < Err:
    print('Solutions converged at iteration: ',iter)
    #Exit the loop
    break
```

```
    #Update solution
    x[:]=xnew[:]

#Assign the X vector to the original Umean and nuT vectors
Umean[0:N+1]=x[0:N+1]
nuT[0:N+1]=x[N+1:2*N+2]

if ErrUmean < Err and ErrnuT < Err:

    #Plot the mean velocity versus z
    plt.plot(Umean,z)
    plt.xlabel('<U> [m s^-1]')
    plt.ylabel('z [m]')
    plt.title('Mean Velocity as Function of z')
    plt.show()

    #Plot the turbulent viscosity versus z
    plt.plot(nuT,z)
    plt.xlabel('nuT [m^2 s^-1]')
    plt.ylabel('z [m]')
    plt.title('Turbulent Viscosity as Function of z')
    plt.show()

else:
    print('Solutions did not converge!')
```

Upon completing the code. You should get the following figures. Try to answer the following questions.

- What is the effect of increasing pressure gradient on mean velocity and turbulent viscosity?
- What is the effect of increasing mixing length on mean velocity and turbulent viscosity?
- Instead of a non-zero initial turbulent viscosity, initialize `nuTinitial` to 0. Can you obtain a converged solution? Reason why.

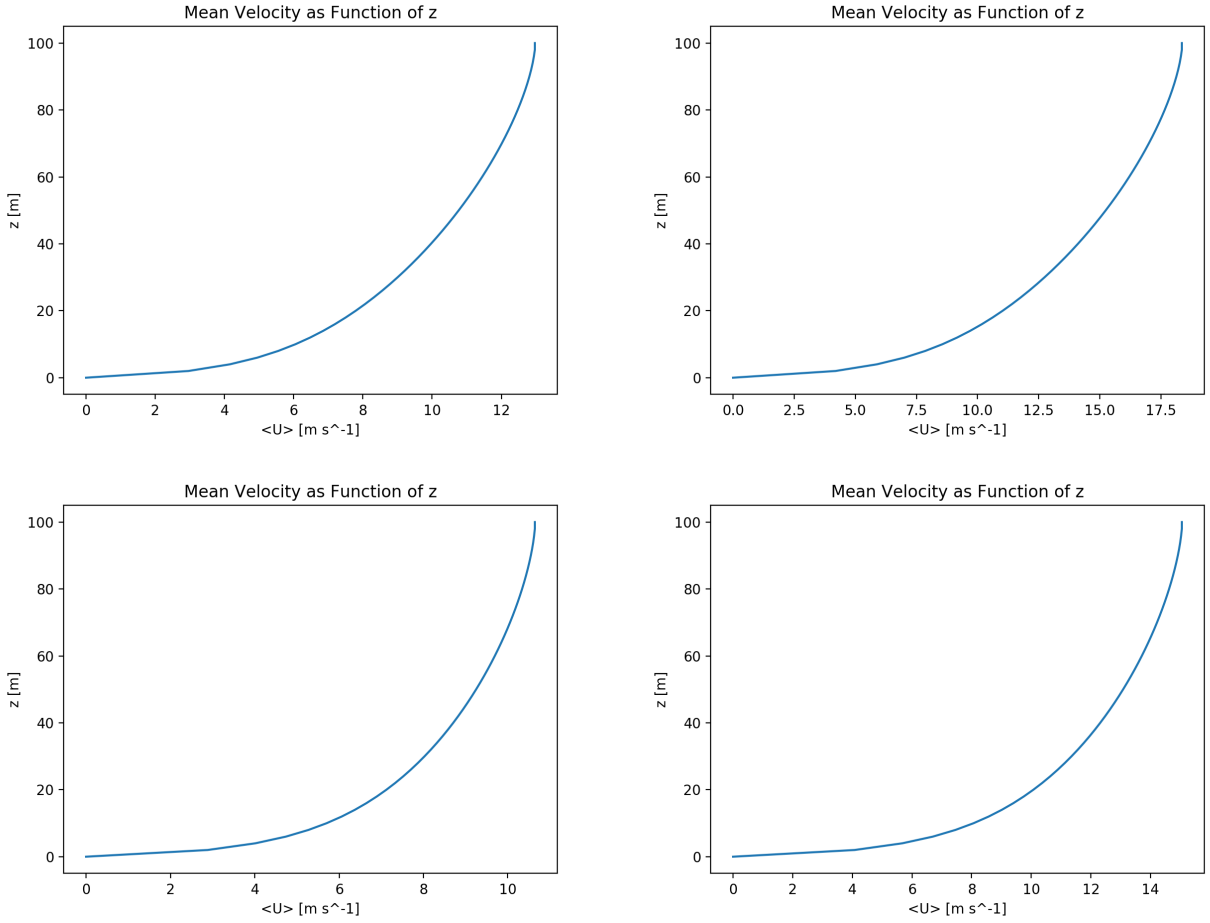


Figure 2: Momentum for Case1 (top left), Case 2 (top right), Case 3 (bottom left), Case 4 (bottom right)

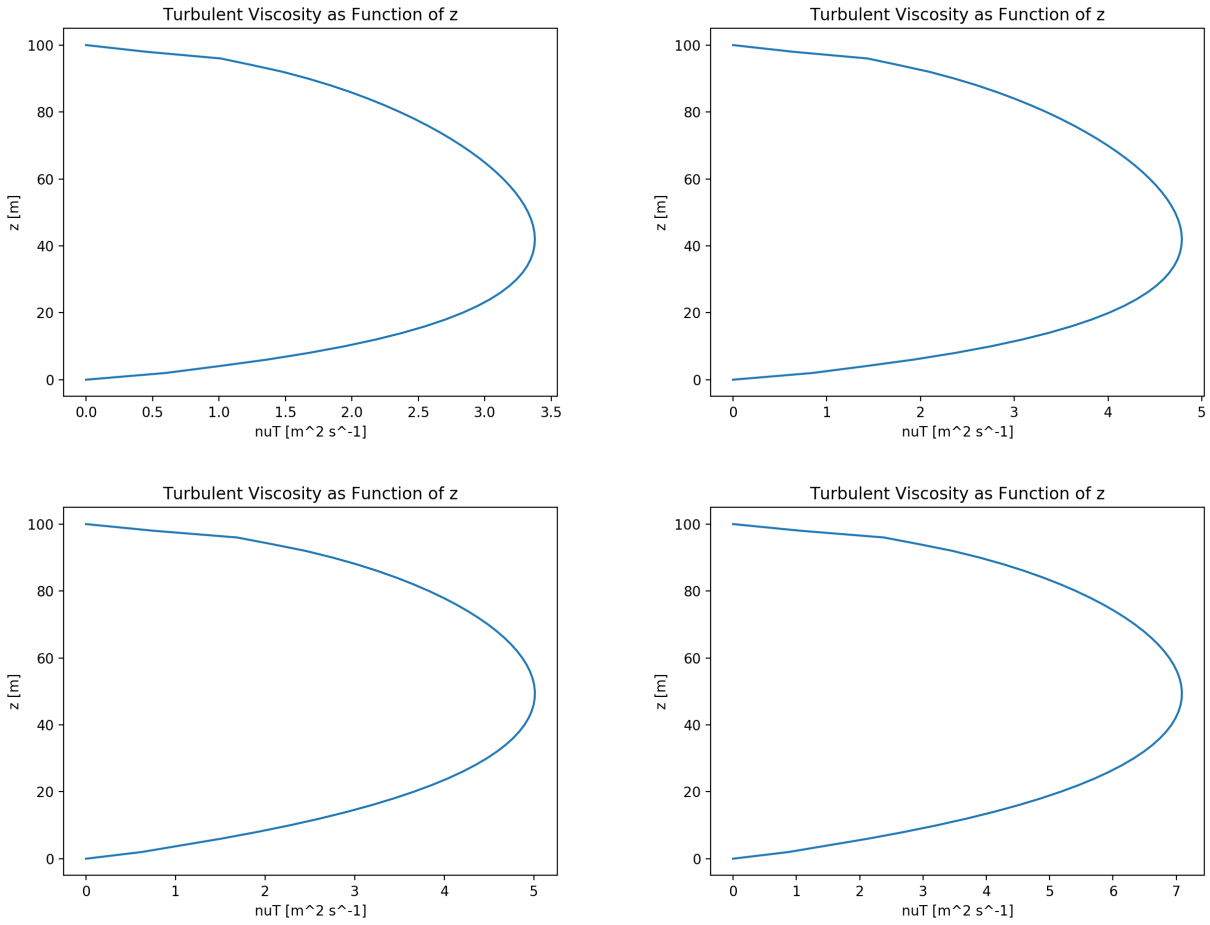


Figure 3: Turbulent viscosity for Case1 (top left), Case 2 (top right), Case 3 (bottom left), Case 4 (bottom right)

ENGG*6790: Theory and Applications of Turbulence

1D Momentum and Turbulent Kinetic Energy Equations over Flat Surface Formulated as a Steady Model and Analyzed for Grid Convergence

Amir A. Aliabadi

March 14, 2019

1 Introduction

In this lab we wish to quantify order of convergence and the Grid Convergence Index (GCI) for a steady 1D momentum and turbulent kinetic energy model of turbulence. In the lectures, the *order of grid convergence* was introduced as a quantity and quantifies the behaviour of the solution error defined as the difference between the discrete solution and the exact solution,

$$E = f(h) - f_{exact} = Ch^p + H.O.T. \tag{1}$$

where C is a constant, h is some measure of mesh spacing, and p is the order of convergence. The Higher Order Terms ($H.O.T.$) are negligible compared to Ch^p.

A numerical code uses a numerical algorithm that will provide a theoretical order of convergence; however, the boundary conditions, numerical models, and mesh will reduce this order so that the observed order of convergence will likely be lower. Neglecting $H.O.T$ and taking the logarithm of both sides of the above equation result in

$$\ln E = \ln C + p \ln h. \tag{2}$$

The order of convergence p can be obtained from the slope of the curve of $\ln E$ versus $\ln h$. If such data points are available, the slope can be read from the graph or the slope can be computed from a least-squares fit to the data.

Once the order of convergence is determined, it is possible to calculate the Grid Convergence Index (GCI). The GCI is a measure of the percentage the computed solution is away from the asymptotic computed solution. It indicates an error band on how far the solution is from the asymptotic value and how much the solution would change with a further refinement of the mesh. A small value of GCI indicates that the computation is within the asymptotic range. The GCI is defined as

$$GCI_{mn} = \frac{F_s|\epsilon_{mn}|}{r^p - 1} \tag{3}$$

where F_s is a factor of safety. The refinement may be in either space or time. The factor of safety is recommended to be 3.0 for comparisons of two meshes and 1.25 for comparison over three meshes or more. The relative error ϵ_{mn} is defined by

$$\epsilon_{mn} = \frac{\phi_m - \phi_n}{\phi_n}. \tag{4}$$

where ϕ_m and ϕ_n are any solution of interest at two consecutive levels, or resolutions, of the mesh. The choice of F_s is affected by whether two or three levels of mesh have been used to estimate p.

In the lectures the turbulent kinetic energy model was introduced as one transport equation to predict the turbulent kinetic energy. This turbulent kinetic energy was then used to formulate turbulent viscosity so that the momentum equation can be solved. The turbulent kinetic energy equation is given as

$$\underbrace{\frac{\overline{D}k}{\overline{D}t}}_{\text{Material Derivative}} \equiv \underbrace{\frac{\partial k}{\partial t}}_{\text{Storage}} + \underbrace{\langle U\rangle.\nabla k}_{\text{Advection}} = \underbrace{\nabla.\left(\frac{\nu_T}{\sigma_k}\nabla k\right)}_{\text{Energy Flux Divergence}} + \underbrace{\mathcal{P}}_{\text{Production}} - \underbrace{\epsilon}_{\text{Dissipation}}, \tag{5}$$

$$\begin{cases} \nu_T = ck^{1/2}\ell_m, \\ \epsilon = C_D\frac{k^{3/2}}{\ell_m}, \\ \ell_m(\mathbf{x}, t) \text{ known.} \end{cases}$$

This model can be employed to develop a one-dimensional transport model for momentum and turbulent kinetic energy under steady-state conditions. Suppose we use the Cartesian coordinate system with coordinate axes of x, y, and z, and velocities corresponding to these axes being $U = \langle U\rangle + u$, $V = \langle V\rangle + v$, $W = \langle W\rangle + w$, respectively, as shown in the schematic. We can assume that mean flow is only in the x direction parallel to the surface and that the direction z is normal to the surface so that $\langle V\rangle = \langle W\rangle = 0$. Assume that the modified pressure has a constant gradient in the x direction, the 1D momentum equation then simplifies to

$$0 = \underbrace{\frac{\partial}{\partial z}\left(\nu_T\frac{\partial\langle U\rangle}{\partial z}\right)}_{\text{Surface Forces and Reynolds Stress}} - \underbrace{\tau}_{\text{Modified Pressure Forces}} \tag{6}$$

The one-dimensional turbulent kinetic energy equation can be developed as follows

$$0 = \underbrace{\frac{\partial}{\partial z}\left(\frac{\nu_T}{\sigma_k}\frac{\partial k}{\partial z}\right)}_{\text{Energy Flux Divergence}} + \underbrace{\nu_T\left(\frac{\partial\langle U\rangle}{\partial z}\right)^2}_{\text{Shear Production}} - \underbrace{\epsilon}_{\text{Dissipation}} \tag{7}$$

where the energy flux divergence was discussed in the lectures. This term ensures that the resulting model transport equation for k yields smooth solutions, and that a boundary condition can be imposed on k everywhere in the boundary of the domain. Otherwise the model may diverge if other transport mechanisms for k are much smaller than this term. The shear production term, is an example of a production term $\mathcal{P}$, that contributes to the generation of the turbulent kinetic energy. Here, when there is non-zero mean velocity gradient, turbulent kinetic energy is generated. The dissipation term is responsible for consuming turbulent kinetic energy down the energy cascade.

To close the turbulence model we can assume that the turbulent Prandtl number is unity, i.e. $\sigma_k = 1$. We can model turbulent viscosity, dissipation rate, and the appropriate mixing length as follows.

$$\begin{cases} \nu_T = C_k \ell_m k^{1/2}, \\ \epsilon = C_\epsilon \ell_m^{-1} k^{3/2}, \\ \ell_m = \kappa z / (1 + \frac{\kappa z}{\ell_0}). \end{cases}$$

where $\kappa = 0.41$ is the von Kármán constant, and ℓ_0 is the maximum mixing length. This formulation for mixing length has the nice property that it is bounded between zero and ℓ_0, which is physically sound since mixing length increases linearly in the log-law sublayer near a wall but cannot increase indefinitely in the interior of the domain. This formulation results in

$$\begin{cases} z \to 0 & \ell_m \to \kappa z \\ z \to \infty & \ell_m \to \ell_0 \end{cases}$$

So we have two equations: momentum and turbulent kinetic energy. We can eliminate ν_T and ϵ from the momentum and turbulent kinetic energy equations by direct substitutions and simplifications using the chain rule. So the two equations can be re-expressed as

$$\begin{aligned} 0 &= \frac{\partial}{\partial z}\left(C_k \ell_m k^{1/2} \frac{\partial \langle U \rangle}{\partial z} \right) - \tau \\ &= 0.5 C_k \ell_m k^{-1/2} \frac{\partial k}{\partial z} \frac{\partial \langle U \rangle}{\partial z} + C_k \ell_m k^{1/2} \frac{\partial^2 \langle U \rangle}{\partial z^2} - \tau \end{aligned} \tag{8}$$

$$\begin{aligned} 0 &= \frac{\partial}{\partial z}\left(C_k \ell_m k^{1/2} \frac{\partial k}{\partial z} \right) + C_k \ell_m k^{1/2} \left(\frac{\partial \langle U \rangle}{\partial z} \right)^2 - C_\epsilon \ell_m^{-1} k^{3/2} \\ &= 0.5 C_k \ell_m k^{-1/2} \left(\frac{\partial k}{\partial z} \right)^2 + C_k \ell_m k^{1/2} \frac{\partial^2 k}{\partial z^2} + C_k \ell_m k^{1/2} \left(\frac{\partial \langle U \rangle}{\partial z} \right)^2 - C_\epsilon \ell_m^{-1} k^{3/2}. \end{aligned} \tag{9}$$

As can be seen the two equations are extremely non-linear. They involve non-integer powers of the unknowns and their derivatives. They also involve the multiplication of the unknowns and

derivatives. These equations can be linearized and solved using a finite difference scheme. Figure below shows the finite difference representation of the solution spaces for momentum and turbulent kinetic energy.

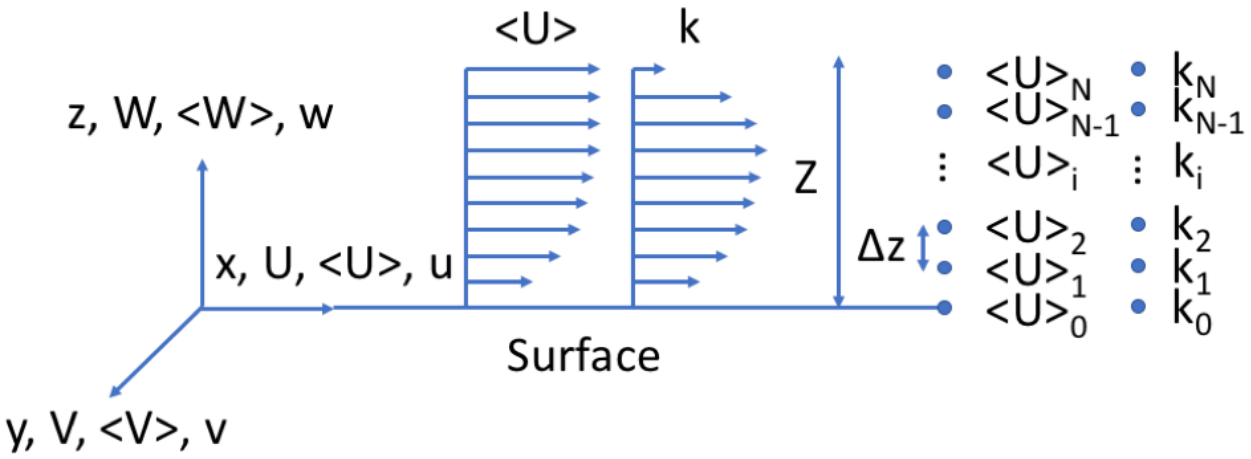


Figure 1: Schematic of 1D flow over flat surface using Cartesian coordinate system; finite difference representation of the 1D flow.

For notational convenience we can represent derivatives by superscripts and subsequently re-express the momentum and turbulent kinetic energy equations using this notation.

$$\begin{cases} \langle U\rangle^{(0)} = \langle U\rangle, \langle U\rangle^{(1)} = \frac{\partial\langle U\rangle}{\partial z}, \langle U\rangle^{(2)} = \frac{\partial^2\langle U\rangle}{\partial z^2} \\ k^{(0)} = k, k^{(1)} = \frac{\partial k}{\partial z}, k^{(2)} = \frac{\partial^2 k}{\partial z^2} \end{cases}$$

$$0 = \underbrace{0.5C_k\ell_m(k^{(0)})^{-1/2}k^{(1)}\langle U\rangle^{(1)}}_{f_{1,mom}} + \underbrace{C_k\ell_m(k^{(0)})^{1/2}\langle U\rangle^{(2)}}_{f_{2,mom}} - \tau \tag{10}$$

$$0 = \underbrace{0.5C_k\ell_m(k^{(0)})^{-1/2}(k^{(1)})^2}_{f_{1,tke}} + \underbrace{C_k\ell_m(k^{(0)})^{1/2}k^{(2)}}_{f_{2,tke}} + \underbrace{C_k\ell_m(k^{(0)})^{1/2}(\langle U\rangle^{(1)})^2}_{f_{3,tke}} \underbrace{-C_\epsilon\ell_m^{-1}(k^{(0)})^{3/2}}_{f_{4,tke}}. \tag{11}$$

where each non-linear term in the equations have been renamed by a function f, Next, each f function can be replaced by its approximate using the Newton method expressing the function around an arbitrary point z_i. Beginning with the momentum equation the f function approximations are

$$
\begin{aligned}
f_{1,mom.} &\approx f_{1,mom.}(z_i) \\
&+ \frac{\partial f_{1,mom.}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{1,mom.}}{\partial k^{(1)}}|_{z_i} \left[k^{(1)}(z) - k^{(1)}(z_i)\right] \\
&+ \frac{\partial f_{1,mom.}}{\partial \langle U\rangle^{(1)}}|_{z_i} \left[\langle U\rangle^{(1)}(z) - \langle U\rangle^{(1)}(z_i)\right] \\
&= 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) \\
&- 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) [k^{(0)}(z) - k^{(0)}(z_i)] \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(1)}(z_i) [k^{(1)}(z) - k^{(1)}(z_i)] \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \left[\langle U\rangle^{(1)}(z) - \langle U\rangle^{(1)}(z_i)\right] \\
&= -0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) \\
&- 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) k^{(0)}(z) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(1)}(z_i) k^{(1)}(z) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z)
\end{aligned} \tag{12}
$$

$$
\begin{aligned}
f_{2,mom.} &\approx f_{2,mom.}(z_i) \\
&+ \frac{\partial f_{2,mom.}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{2,mom.}}{\partial \langle U\rangle^{(2)}}|_{z_i} \left[\langle U\rangle^{(2)}(z) - \langle U\rangle^{(2)}(z_i)\right] \\
&= C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U\rangle^{(2)}(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(2)}(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ C_k \ell_m (k^{(0)})^{1/2}(z_i) \left[\langle U\rangle^{(2)}(z) - \langle U\rangle^{(2)}(z_i)\right] \\
&= -0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(2)}(z_i) k^{(0)}(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(2)}(z_i) k^{(0)}(z) \\
&+ C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U\rangle^{(2)}(z)
\end{aligned} \tag{13}
$$

$$
\begin{aligned}
f_{1,tke} &\approx f_{1,tke}(z_i) \\
&+ \frac{\partial f_{1,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{1,tke}}{\partial k^{(1)}}|_{z_i} \left[k^{(1)}(z) - k^{(1)}(z_i)\right] \\
&= 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(k^{(1)})^2(z_i) \\
&- 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i)(k^{(1)})^2(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \left[k^{(1)}(z) - k^{(1)}(z_i)\right] \\
&= -0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(k^{(1)})^2(z_i) \\
&- 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i)(k^{(1)})^2(z_i) k^{(0)}(z) \\
&+ C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) k^{(1)}(z)
\end{aligned} \tag{14}
$$

$$
\begin{aligned}
f_{2,tke} &\approx f_{2,tke}(z_i) \\
&+ \frac{\partial f_{2,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{2,tke}}{\partial k^{(2)}}|_{z_i} \left[k^{(2)}(z) - k^{(2)}(z_i)\right] \\
&= C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(2)}(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ C_k \ell_m (k^{(0)})^{1/2}(z_i) \left[k^{(2)}(z) - k^{(2)}(z_i)\right] \\
&= -0.5 C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(2)}(z_i) k^{(0)}(z) \\
&+ C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z)
\end{aligned} \tag{15}
$$

$$
\begin{aligned}
f_{3,tke} &\approx f_{3,tke}(z_i) \\
&+ \frac{\partial f_{3,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{3,tke}}{\partial \langle U \rangle^{(1)}}|_{z_i} \left[\langle U \rangle^{(1)}(z) - \langle U \rangle^{(1)}(z_i)\right] \\
&= C_k \ell_m (k^{(0)})^{1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ 2 C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U \rangle^{(1)}(z_i) \left[\langle U \rangle^{(1)}(z) - \langle U \rangle^{(1)}(z_i)\right] \\
&= -1.5 C_k \ell_m (k^{(0)})^{1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) k^{(0)}(z) \\
&+ 2 C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U \rangle^{(1)}(z_i) \langle U \rangle^{(1)}(z)
\end{aligned} \tag{16}
$$

$$
\begin{aligned}
f_{4,tke} &\approx f_{4,tke}(z_i) \\
&+ \frac{\partial f_{4,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&= -C_\epsilon \ell_m^{-1} (k^{(0)})^{3/2}(z_i) \\
&- 1.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{1/2}(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&= 0.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{3/2}(z_i) \\
&- 1.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{1/2}(z_i) k^{(0)}(z)
\end{aligned} \tag{17}
$$

Now we can write the linearized forms of the momentum and turbulent kinetic energy equations.

$$
c_1 \langle U \rangle^{(1)} + c_2 \langle U \rangle^{(2)} + c_3 k^{(0)} + c_4 k^{(1)} = c_b \tag{18}
$$

$$
\begin{cases}
c_1 = 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \\
c_2 = C_k \ell_m (k^{(0)})^{1/2}(z_i) \\
c_3 = -0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) k^{(1)}(z_i) \langle U \rangle^{(1)}(z_i) + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U \rangle^{(2)}(z_i) \\
c_4 = 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U \rangle^{(1)}(z_i) \\
c_b = -\left[-0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U \rangle^{(1)}(z_i) - 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U \rangle^{(2)}(z_i) k^{(0)}(z_i) - \tau\right]
\end{cases}
$$

$$
d_1 k^{(0)} + d_2 k^{(1)} + d_3 k^{(2)} + d_4 \langle U \rangle^{(1)} = d_b \tag{19}
$$

$$
\begin{cases}
d_1 = -0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) (k^{(1)})^2(z_i) + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(2)}(z_i) \\
+0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) (\langle U \rangle^{(1)})^2(z_i) - 1.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{1/2}(z_i) \\
d_2 = C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \\
d_3 = C_k \ell_m (k^{(0)})^{1/2}(z_i) \\
d_4 = 2 C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U \rangle^{(1)}(z_i) \\
d_b = -\left[-0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i) (k^{(1)})^2(z_i) - 0.5 C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z_i)\right] \\
-\left[-1.5 C_k \ell_m (k^{(0)})^{1/2}(z_i) (\langle U \rangle^{(1)})^2(z_i) + 0.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{3/2}(z_i)\right]
\end{cases}
$$

Now consider that we want to represent the linearized momentum and turbulent kinetic energy equations using finite differences. Consider a vertical discretization Δz as shown in the figure. Using central differencing the spatial derivatives can be replaced by values at indices $i-1$, i, and $i+1$. We can replace the momentum and turbulent kinetic energy equations by their discretized versions as follows

$$
c_1 \frac{\langle U \rangle_{i+1} - \langle U \rangle_{i-1}}{2\Delta z} + c_2 \frac{\langle U \rangle_{i+1} - 2\langle U \rangle_i + \langle U \rangle_{i-1}}{(\Delta z)^2} + c_3 k_i + c_4 \frac{k_{i+1} - k_{i-1}}{2\Delta z} = c_b \tag{20}
$$

$$d_1 k_i + d_2 \frac{k_{i+1} - k_{i-1}}{2\Delta z} + d_3 \frac{k_{i+1} - 2k_i + k_{i-1}}{(\Delta z)^2} + d_4 \frac{\langle U \rangle_{i+1} - \langle U \rangle_{i-1}}{2\Delta z} = d_b \tag{21}$$

The above equations must be rearranged as follows.

$$\begin{aligned}\left(-\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2}\right) \langle U \rangle_{i-1} + \left(-\frac{2c_2}{(\Delta z)^2}\right) \langle U \rangle_i + \left(\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2}\right) \langle U \rangle_{i+1} \\ + \left(-\frac{c_4}{2\Delta z}\right) k_{i-1} + c_3 k_i + \left(\frac{c_4}{2\Delta z}\right) k_{i+1} = c_b\end{aligned} \tag{22}$$

$$\begin{aligned}\left(-\frac{d_2}{2\Delta z} + \frac{d_3}{(\Delta z)^2}\right) k_{i-1} + \left(d_1 - \frac{2d_3}{(\Delta z)^2}\right) k_i + \left(\frac{d_2}{2\Delta z} + \frac{d_3}{(\Delta z)^2}\right) k_{i+1} \\ + \left(-\frac{d_4}{2\Delta z}\right) \langle U \rangle_{i-1} + \left(\frac{d_4}{2\Delta z}\right) \langle U \rangle_{i+1} = d_b\end{aligned} \tag{23}$$

As can bee seen the unknowns $\langle U \rangle_i$ and k_i appear in both discretized momentum and turbulent kinetic energy equations. Therefore, these equations should be combined to arrive at a linear system of equations and subsequently solved using a linear algebra solver. Let us define the unknowns vector $\mathbf{X}$ such that

$$\mathbf{X} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \\ x_N \\ x_{N+1} \\ x_{N+2} \\ \vdots \\ x_{2N} \\ x_{2N+1} \end{bmatrix} = \begin{bmatrix} \langle U \rangle_0 \\ \langle U \rangle_1 \\ \vdots \\ \langle U \rangle_{N-1} \\ \langle U \rangle_N \\ k_0 \\ k_1 \\ \vdots \\ k_{N-1} \\ k_N \end{bmatrix}$$

As can be seen the first half of vector $\mathbf{X}$ contains the $\langle U \rangle_i$ solutions and the second half of vector $\mathbf{X}$ contains the k_i solutions. Note that k_i maps to x_{i+N+1}. Now we need $2N + 2$ linear equations to solve for $\mathbf{X}$, i.e.

$$\begin{cases} \text{Equation 0: } a_{0,0}x_0 + a_{0,1}x_1 + ... + a_{0,2N}x_{2N} + a_{0,2N+1}x_{2N+1} = b_0 \\ \text{Equation 1: } a_{1,0}x_0 + a_{1,1}x_1 + ... + a_{1,2N}x_{2N} + a_{1,2N+1}x_{2N+1} = b_1 \\ ... \\ \text{Equation } i\text{: } a_{i,0}x_0 + a_{i,1}x_1 + ... + a_{i,2N}x_{2N} + a_{i,2N+1}x_{2N+1} = b_i \\ ... \\ \text{Equation } 2N\text{: } a_{2N,0}x_0 + a_{2N,1}x_1 + ... + a_{2N,2N}x_{2N} + a_{2N,2N+1}x_{2N+1} = b_{2N} \\ \text{Equation } 2N+1\text{: } a_{2N+1,0}x_0 + a_{2N+1,1}x_1 + ... + a_{2N+1,2N}x_{2N} + a_{2N+1,2N+1}x_{2N+1} = b_{2N+1} \end{cases}$$

Our next task is to identify $a_{i,j}$ and b_i. These can be inferred from the discretized momentum and turbulent kinetic energy equations. $a_{i,j}$ are mostly zero except for where there is a non-zero coefficient in the corresponding equations. The first N equations (equation 0, equation 1, ... equation N) are the momentum equations. The boundary condition for $\langle U \rangle$ at the surface provides the zeroth equation, i.e.

$$\begin{cases} x_0 = 0 \\ a_{0,0} = 1 \\ b_0 = 0 \end{cases}$$

The next $i : 1 \rightarrow N-1$ equations correspond to the momentum equation in the interior of the domain, so the coefficients can be obtained as follows

$$\begin{cases} \left(-\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2}\right) x_{i-1} + \left(-\frac{2c_2}{(\Delta z)^2}\right) x_i + \left(\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2}\right) x_{i+1} \\ + \left(-\frac{c_4}{2\Delta z}\right) x_{i-1+N+1} + c_3 x_{i+N+1} + \left(\frac{c_4}{2\Delta z}\right) x_{i+1+N+1} = c_b \\ a_{i,i-1} = -\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2} \\ a_{i,i} = -\frac{2c_2}{(\Delta z)^2} \\ a_{i,i+1} = \frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2} \\ a_{i,i-1+N+1} = -\frac{c_4}{2\Delta z} \\ a_{i,i+N+1} = c_3 \\ a_{i,i+1+N+1} = \frac{c_4}{2\Delta z} \\ b_i = c_b \end{cases}$$

Note that where the turbulent kinetic energy term appears the index is shifted by $N+1$. The boundary condition for $\langle U \rangle$ at the top of the domain provides the N^{th} equation, i.e.

$$\begin{cases} x_{N-1} - x_N = 0 \\ a_{N,N-1} = 1 \\ a_{N,N} = -1 \\ b_N = 0 \end{cases}$$

The boundary condition for k at the surface provides the $N+1^{th}$ equation, i.e.

$$\begin{cases} x_{N+1} = 0 \\ a_{N+1,N+1} = 1 \\ b_{N+1} = 0 \end{cases}$$

The next $i : N+2 \rightarrow 2N$ equations correspond to the turbulent kinetic energy equation in the interior of the domain, so the coefficients can be obtained as follows

$$\begin{cases} \left(-\frac{d_2}{2\Delta z} + \frac{d_3}{(\Delta z)^2}\right) x_{i-1} + \left(d_1 - \frac{2d_3}{(\Delta z)^2}\right) x_i + \left(\frac{d_2}{2\Delta z} + \frac{d_3}{(\Delta z)^2}\right) x_{i+1} \\ + \left(-\frac{d_4}{2\Delta z}\right) x_{i-1-(N+1)} + \left(\frac{d_4}{2\Delta z}\right) x_{i+1-(N+1)} = d_b \\ a_{i,i-1-(N+1)} = -\frac{d_4}{2\Delta z} \\ a_{i,i+1-(N+1)} = \frac{d_4}{2\Delta z} \\ a_{i,i-1} = -\frac{d_2}{2\Delta z} + \frac{d_3}{(\Delta z)^2} \\ a_{i,i} = d_1 - \frac{2d_3}{(\Delta z)^2} \\ a_{i,i+1} = \frac{d_2}{2\Delta z} + \frac{d_3}{(\Delta z)^2} \\ b_i = d_b \end{cases}$$

Note that where the momentum term appears the index is shifted by $-(N+1)$. The boundary condition for k at the top of the domain provides the $2N+1^{th}$ equation. The vertical gradient of the turbulent kinetic energy must be zero, i.e.

$$\begin{cases} x_{2N} - x_{2N+1} = 0 \\ a_{2N+1,2N} = 1 \\ a_{2N+1,2N+1} = -1 \\ b_{2N+1} = 0 \end{cases}$$

Finally, we have arrived at linear system of equations that can be solved to provide the unknowns. This system is given as follows

$$\mathbf{AX} = \mathbf{B} \tag{24}$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,2N} & a_{0,2N+1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,2N} & a_{1,2N+1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{2N,0} & a_{2N,1} & \dots & a_{2N,2N} & a_{2N,2N+1} \\ a_{2N+1,0} & a_{2N+1,1} & \dots & a_{2N+1,2N} & a_{2N+1,2N+1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{2N} \\ x_{2N+1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{2N} \\ b_{2N+1} \end{bmatrix}$$

Note that the coefficients identified, themselves, depend on the solution. As a result the solution to the system of equations above must be found iteratively. That is, an initial guess for the solution vector $\mathbf{X}$ must be assumed. Then the system of equations must be solved iteratively. After each iteration the solutions must be updated. The system of equations must be solved iteratively until the difference between successive solutions is less than a specified error. For this purpose, the maximum norm can be considered. Suppose that a relative error of $Err = 0.01$ is specified for either the momentum or turbulent kinetic energy solution. Also suppose the x_i and $x_i^{(new)}$ represent two successive solutions for a specific point. The iteration can be stopped if the following conditions are met

$$\begin{cases} L_{\infty,mom.} = \max\left(|\frac{x_0^{(new)} - x_0}{x_0}|, |\frac{x_1^{(new)} - x_1}{x_1}|, ..., |\frac{x_N^{(new)} - x_N}{x_N}| \right) < Err \\ L_{\infty,tke} = \max\left(|\frac{x_{N+1}^{(new)} - x_{N+1}}{x_{N+1}}|, |\frac{x_{N+2}^{(new)} - x_{N+2}}{x_{N+2}}|, ..., |\frac{x_{2N+1}^{(new)} - x_{2N+1}}{x_{2N+1}}| \right) < Err \end{cases}$$

When solving a system of equations iteratively, it is sometimes more stable to only partially update a solution after each iteration. This is known as under relaxation. Consider that ϕ^{n-1} is the solution space found in the previous iteration and ϕ^{new} is the newly found solution. With the under relaxation factor $0 < \alpha < 1$, the solution can be updated for the next iteration such that

$$\phi^n = \phi^{n-1} + \alpha(\phi^{new} - \phi^{n-1}). \tag{25}$$

Particularly, whenever solving non-linear system of equations, this method improves stability of obtaining a numerical solution.

The simulation is desired for 6 levels of mesh shown in table below. These levels are ordered from finest to coarsest. The finest mesh is simulated to approximate an exact solution, while the GCI is desired for the maximum velocity on top of the domain. Five GCI levels, 1, 2, 3, 4, and 5, will be calculated for mesh levels 21, 32, 43, 54, 65.

2 Python Script

Complete the following code.

```
import random
```

Table 1: Simulation cases with various mesh resolutions

Mesh Level	N	h[m]
0	1000	0.1
1	64	1.5625
2	32	3.125
3	16	6.25
4	8	12.5
5	4	25
6	2	50

```
import sys
import os
import numpy
import matplotlib.pyplot as plt

#Define under-relaxation factor
alpha=0.1

#Define horizontal pressure gradient divided by density [m s^-2]
tau=-0.005

#Define von Karman constant
kappa=0.41

#Define maximum mixing length [m]
l0=10

#Define turbulence model constants, Martilli et al. (2002)
Ck=0.4
Ce=0.71

#Define maximum iteration number
MaxIter=100

#Define relative error
Err=0.01

#Define z axis from 0 to Z with dz increments
Z=100       #[m]
N=2
dz=Z/N      #[m]
z=numpy.linspace(0,Z,N+1)

#Define and initialize a mean velocity vector [m s^-1]
Uinitial=1
```

```
Umean=numpy.zeros((N+1,1))
Umean[:]=Uinitial

#Define and initialize turbulent kinetic energy [m^2 s^-2]
kinitial=0.1
k=numpy.zeros((N+1,1))
k[:]=kinitial

#Define unknown vector X, coefficient matrix A, and vector B, in AX=B
x=numpy.zeros((2*N+2,1))
xnew=numpy.zeros((2*N+2,1))
b=numpy.zeros((2*N+2,1))
a=numpy.zeros((2*N+2,2*N+2))

#Initialize solution vector X
#This is a short syntax for for loop
x[0:N+1]=Umean[0:N+1]
x[N+1:2*N+2]=k[0:N+1]

for iter in range(1, MaxIter):
    #Momentum equations
    #i=0
    a[0][0]=1
    b[0]=0
    #i=1 to N-1
    for i in range(1, N):
        #Calculate derivatives by finite differences for the current i index
        #Remember to shift indices by N+1 if needed
        lm=kappa*z[i]/(1+(kappa*z[i])/l0)
        k0=x[i+N+1]
        k1=(x[i+1+N+1]-x[i-1+N+1])/(2*dz)
        k2=(x[i+1+N+1]-2*x[i+N+1]+x[i-1+N+1])/(dz**2)
        Umean0=x[i]
        Umean1=(x[i+1]-x[i-1])/(2*dz)
        Umean2=(x[i+1]-2*x[i]+x[i-1])/(dz**2)
        # Set constants necessary to build the coefficient matrix
        c1=0.5*Ck*lm*k0**(-1/2)*k1
        c2=Ck*lm*k0**(1/2)
        c3=-0.25*Ck*lm*k0**(-3/2)*k1*Umean1+0.5*Ck*lm*k0**(-1/2)*Umean2
        c4=0.5*Ck*lm*k0**(-1/2)*Umean1
        cb=-(-0.25*Ck*lm*k0**(-1/2)*k1*Umean1-0.5*Ck*lm*k0**(-1/2)*Umean2*k0-tau)
        # Set the coefficient matrix and the B vector
        a[i][i-1]=...
        a[i][i]=...
        a[i][i+1]=...
        a[i][i-1+N+1]=...
```

```
    a[i][i+N+1]=...
    a[i][i+1+N+1]=...
    b[i]=...
#i=N
a[N][N-1]=...
a[N][N]=...
b[N]=...

#Kinetic energy equations
#i=N+1
a[N+1][N+1]=1
b[N+1]=0
#i=N+2 to 2N
for i in range(N+2, 2*N+1):
    #Calculate derivatives by finite differences for the current i index
    #Shift indices by -(N+1) if needed
    lm=kappa*z[i-(N+1)]/(1+(kappa*z[i-(N+1)])/l0)
    k0=x[i]
    k1=(x[i+1]-x[i-1])/(2*dz)
    k2=(x[i+1]-2*x[i]+x[i-1])/(dz ** 2)
    Umean0=x[i-(N+1)]
    Umean1=(x[i+1-(N+1)]-x[i-1-(N+1)])/(2*dz)
    Umean2=(x[i+1-(N+1)]-2*x[i-(N+1)]+x[i-1-(N+1)])/(dz**2)
    #Set constants necessary to build the coefficient matrix
    d1=-0.25*Ck*lm*k0**(-3/2)*k1**2+0.5*Ck*lm*k0**(-1/2)*k2\
         +0.5*Ck*lm*k0**(-1/2)*Umean1**2-1.5*Ce*lm**(-1)*k0**(1/2)
    d2=Ck*lm*k0**(-1/2)*k1
    d3=Ck*lm*k0**(1/2)
    d4=2*Ck*lm*k0**(1/2)*Umean1
    db=-(-0.25*Ck*lm*k0**(-1/2)*k1**2-0.5*Ck*lm*k0**(1/2)*k2)\
         -(-1.5*Ck*lm*k0**(1/2)*Umean1**2+0.5*Ce*lm**(-1)*k0**(3/2))
    # Set the coefficient matrix and the B vector
    a[i][i-1-(N+1)]=...
    a[i][i+1-(N+1)]=...
    a[i][i-1]=...
    a[i][i]=...
    a[i][i+1]=...
    b[i]=...
# i=2N+1
a[2*N+1][2*N]=...
a[2*N+1][2*N+1]=...
b[2*N+1]=...

#print(a)

xnew = numpy.linalg.solve(a, b)
```

```
    # Calculate maximum norm errors for both momentum and turbulent kinetic energy
    ErrUmean=numpy.max(numpy.abs(numpy.divide(xnew[1:N+1]-x[1:N+1],x[1:N+1])))
    Errk=...

    print('Iteration=',iter,'ErrUmean=',ErrUmean,'Errk=',Errk)

    if ErrUmean < Err and Errk < Err:
        print('Solutions converged at iteration: ', iter)
        # Exit the loop
        break

    # Update solution
    x[:]=x[:]+alpha*(xnew[:]-x[:])

#Assign the X vector to the original Umean and k vectors
Umean[0:N+1]=...
k[0:N+1]=...

#Print the solution monitor as the velocity on top of the model domain
print('Maximum Umean=',Umean[N])

#Plot the mean velocity versus z
plt.plot(Umean, z)
plt.xlabel(' < U > [m s ^ -1]')
plt.ylabel('z [m]')
plt.title('Mean Velocity as Function of z')
plt.show()
# Plot the turbulent viscosity versus z
plt.plot(k, z)
plt.xlabel('k [m^2 s^-2]')
plt.ylabel('z [m]')
plt.title('Turbulent Kinetic Energy as Function of z')
plt.show()
```

Upon completing the code. You should get the following output from the console. For each iteration the convergence criteria is printed to the screen to monitor the solution behaviour. Note that with only a few tens of iterations using the Newton method, it is possible to converge to a solution with a relative error less than 1%.

```
...
Iteration= 43 ErrUmean= 0.00536326893688 Errk= 0.0123014708332
Iteration= 44 ErrUmean= 0.00482027210921 Errk= 0.0110574058777
Iteration= 45 ErrUmean= 0.00433285589024 Errk= 0.00994041815673
Solutions converged at iteration:  45
Maximum Umean= [ 10.24023021]
```

Run the code for all mesh levels and record the maximum velocity on the top of the domain on paper. Some example results are shown below for mesh levels 5 and 0. Observe that the solution curves are smoother for higher resolution simulations.

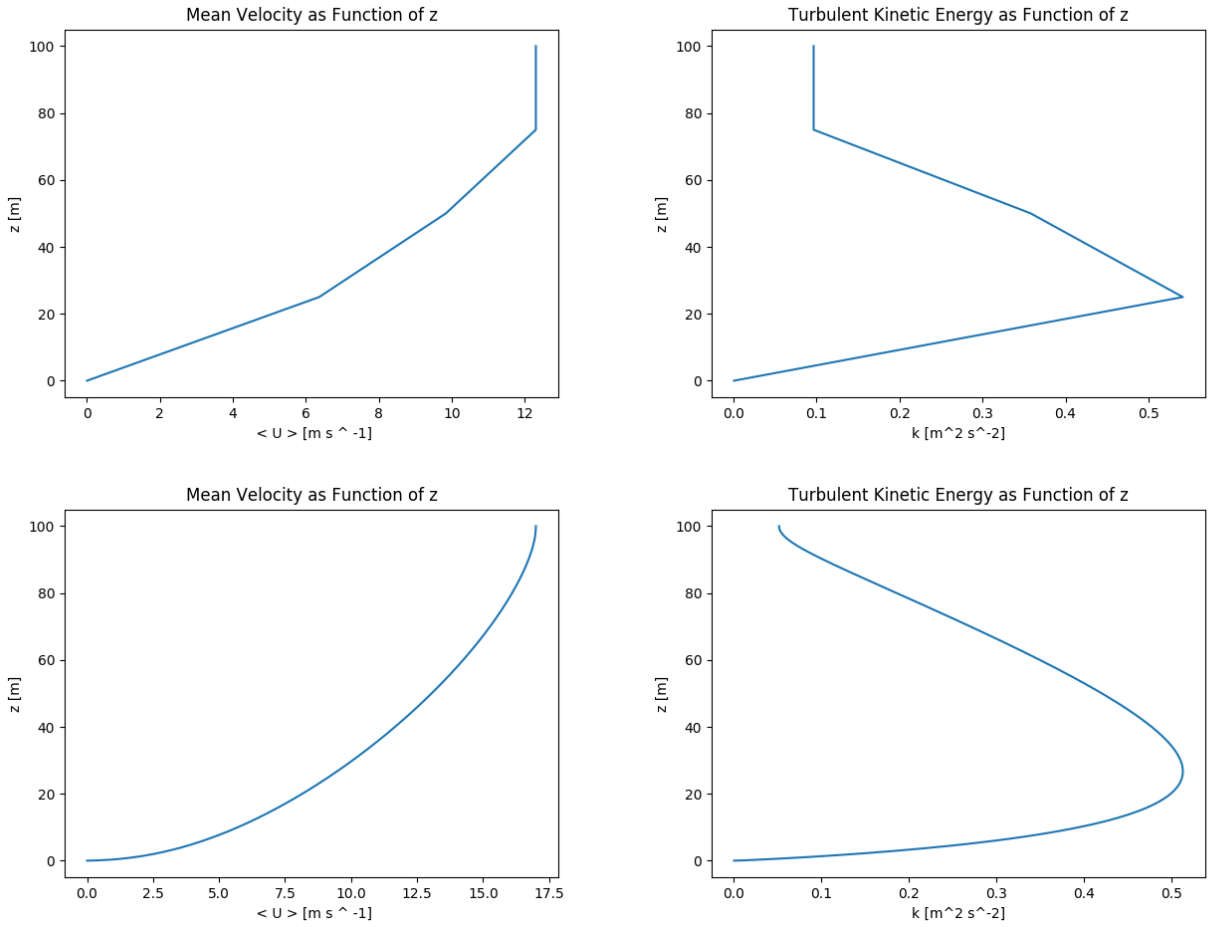


Figure 2: Momentum and turbulent kinetic energy for mesh level 5 (top) and mesh level 0 (bottom)

Next the GCI must be calculated, using the maximum velocity values you recorded on paper, complete the following script. This script indicates that the order of convergence p for maximum velocity on top of the domain is approximately 0.5 as shown in the figure below. The information obtained thus far can be used to calculate the GCI levels, ,1, 2, 3, 4, 5, for mesh levels 21, 32, 43, 54, 65. Note that on the GCI plot the value calculated for each successive pair of meshes, decreases GCI, indicating that the solution is approaching asymptotically to the exact solution.

```
#Now perform GCI analysis
#h for N=2, 4, 8, 16, 32, 64
h=numpy.zeros((6,1))
h[0]=50
h[1]=25
h[2]=12.5
h[3]=6.25
h[4]=3.125
```

```
h[5]=1.5625


#fexact calculated for N=1000
fexact=numpy.zeros((6,1))
fexact[0]=17.0
fexact[1]=17.0
fexact[2]=...
fexact[3]=...
fexact[4]=...
fexact[5]=...

#fh for N=2, 4, 8, 16, 32, 62
fh=numpy.zeros((6,1))
fh[0]=...
fh[1]=...
fh[2]=...
fh[3]=...
fh[4]=...
fh[5]=...

#Calculate absolute error
absE=numpy.zeros((6,1))
absE=numpy.abs(fh-fexact)

#Calculate the log of h and absolute E
logh=numpy.log(h)
logabsE=...

#Plot the mean velocity versus z
plt.plot(logh, logabsE)
plt.xlabel('log(h)')
plt.ylabel('log(abs(E))')
plt.title('Absolute Error versus Grid Discretization')
plt.show()

#GCI Safety factor based on at least 3 levels of mesh
Fs=1.25

#Set mesh refinement ratio and order of convergence
r=2
p=0.5

#Assume N=2, 4, 8, 16, 32, 62 correspond to mesh levels 6, 5, 4, 3, 2, 1
#GCI for levels 65, 54, 43, 32, 21
GCI=numpy.zeros((5,1))
```

```
for i in range(0,5):
    GCI[i]=Fs*(fh[i+1]-fh[i])/fh[i]/(r**p-1)

#Plot the mean velocity versus z
plt.plot([5, 4, 3, 2, 1], GCI, 'bs')
plt.xlabel('GCI Level n [mn]: 5 [65], 4 [54], 3 [43], 2 [32], 1 [21]')
plt.ylabel('GCI Level n [mn]')
plt.title('Grid Convergence Index (GCI) versus Grid Levels mn')
plt.show()
```

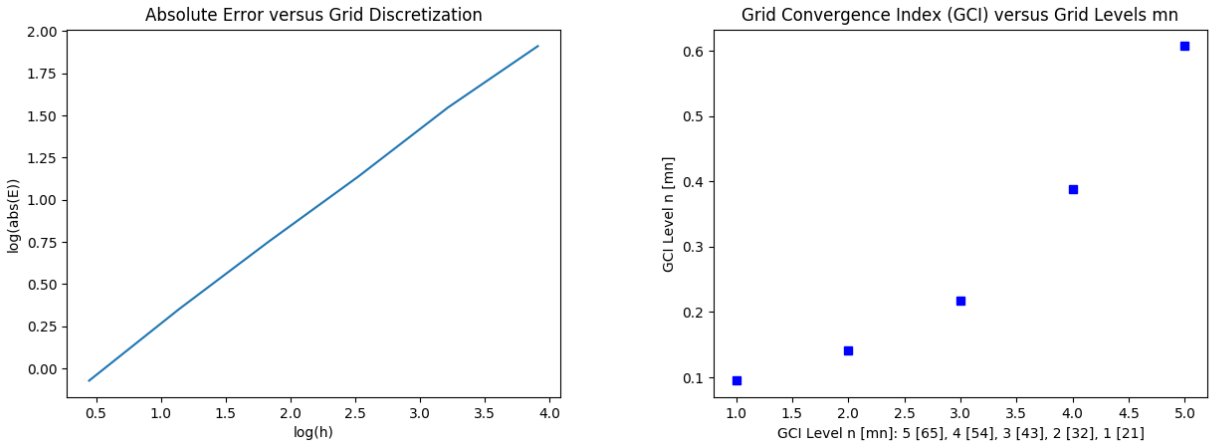


Figure 3: Order of convergence (left) and GCI (right) plots

Try to answer the following questions.

- How can the order of convergence be determined from the plots above?
- Comment on why the order of convergence is approximately 0.5 for maximum velocity in the domain, i.e. on top of the domain?
- How do you compare the results of this simulation to the results obtained by the mixing length model?

ENGG*6790: Theory and Applications of Turbulence

1D Momentum and Turbulent Kinetic Energy Equations over Flat Surface Formulated as a Transient Model

Amir A. Aliabadi

March 14, 2019

1 Introduction

In the lectures the turbulent kinetic energy model was introduced as one transport equation to predict the turbulent kinetic energy. This turbulent kinetic energy was then used to formulate turbulent viscosity so that the momentum equation can be solved. The turbulent kinetic energy equation is given as

$$\underbrace{\frac{\overline{D}k}{\overline{D}t}}_{\text{Material Derivative}} \equiv \underbrace{\frac{\partial k}{\partial t}}_{\text{Storage}} + \underbrace{\langle U\rangle.\nabla k}_{\text{Advection}} = \underbrace{\nabla.\left(\frac{\nu_T}{\sigma_k}\nabla k\right)}_{\text{Energy Flux Divergence}} + \underbrace{\mathcal{P}}_{\text{Production}} - \underbrace{\epsilon}_{\text{Dissipation}}, \tag{1}$$

$$\begin{cases} \nu_T = ck^{1/2}\ell_m, \\ \epsilon = C_D\frac{k^{3/2}}{\ell_m}, \\ \ell_m(\mathbf{x},t) \text{ known.} \end{cases}$$

This model can be employed to develop a one-dimensional transport model for momentum and turbulent kinetic energy. Suppose we use the Cartesian coordinate system with coordinate axes of x, y, and z, and velocities corresponding to these axes being $U = \langle U\rangle + u$, $V = \langle V\rangle + v$, $W = \langle W\rangle + w$, respectively, as shown in the schematic. We can assume that mean flow is only in the x direction parallel to the surface and that the direction z is normal to the surface so that $\langle V\rangle = \langle W\rangle = 0$. However, the mean velocity $\langle U\rangle$ will change as a function of time. Assume that the modified pressure has a constant gradient in the x direction, the 1D momentum equation then simplifies to

$$\underbrace{\frac{\partial\langle U\rangle}{\partial t}}_{\text{Storage}} = \underbrace{\frac{\partial}{\partial z}\left(\nu_T\frac{\partial\langle U\rangle}{\partial z}\right)}_{\text{Surface Forces and Reynolds Stress}} - \underbrace{\tau}_{\text{Modified Pressure Forces}} \tag{2}$$

The one-dimensional turbulent kinetic energy equation can be developed as follows

$$\underbrace{\frac{\partial k}{\partial t}}_{\text{Storage}} = \underbrace{\frac{\partial}{\partial z}\left(\frac{\nu_T}{\sigma_k}\frac{\partial k}{\partial z}\right)}_{\text{Energy Flux Divergence}} + \underbrace{\nu_T\left(\frac{\partial \langle U\rangle}{\partial z}\right)^2}_{\text{Shear Production}} - \underbrace{\epsilon}_{\text{Dissipation}} \tag{3}$$

where the energy flux divergence was discussed in the lectures. This term ensures that the resulting model transport equation for k yields smooth solutions, and that a boundary condition can be imposed on k everywhere in the boundary of the domain. Otherwise the model may diverge if other transport mechanisms for k are much smaller than this term. The shear production term, is an example of a production term $\mathcal{P}$, that contributes to the generation of the turbulent kinetic energy. Here, when there is non-zero mean velocity gradient, turbulent kinetic energy is generated. The dissipation term is responsible for consuming turbulent kinetic energy down the energy cascade.

To close the turbulence model we can assume that the turbulent Prandtl number is unity, i.e. $\sigma_k = 1$. We can model turbulent viscosity, dissipation rate, and the appropriate mixing length as follows.

$$\begin{cases} \nu_T = C_k \ell_m k^{1/2}, \\ \epsilon = C_\epsilon \ell_m^{-1} k^{3/2}, \\ \ell_m = \kappa z/(1 + \frac{\kappa z}{\ell_0}). \end{cases}$$

where $\kappa = 0.41$ is the von Kármán constant, and ℓ_0 is the maximum mixing length. This formulation for mixing length has the nice property that it is bounded between zero and ℓ_0, which is physically sound since mixing length increases linearly in the log-law sublayer near a wall but cannot increase indefinitely in the interior of the domain. This formulation results in

$$\begin{cases} z \to 0 & \ell_m \to \kappa z \\ z \to \infty & \ell_m \to \ell_0 \end{cases}$$

So we have two equations: momentum and turbulent kinetic energy. We can eliminate ν_T and ϵ from the momentum and turbulent kinetic energy equations by direct substitutions and simplifications using the chain rule. So the two equations can be re-expressed as

$$\begin{aligned} \frac{\partial \langle U\rangle}{\partial t} &= \frac{\partial}{\partial z}\left(C_k \ell_m k^{1/2}\frac{\partial \langle U\rangle}{\partial z}\right) - \tau \\ &= 0.5 C_k \ell_m k^{-1/2}\frac{\partial k}{\partial z}\frac{\partial \langle U\rangle}{\partial z} + C_k \ell_m k^{1/2}\frac{\partial^2 \langle U\rangle}{\partial z^2} - \tau \end{aligned} \tag{4}$$

$$\frac{\partial k}{\partial t} = \frac{\partial}{\partial z}\left(C_k \ell_m k^{1/2} \frac{\partial k}{\partial z}\right) + C_k \ell_m k^{1/2} \left(\frac{\partial \langle U \rangle}{\partial z}\right)^2 - C_\epsilon \ell_m^{-1} k^{3/2}$$
$$= 0.5 C_k \ell_m k^{-1/2} \left(\frac{\partial k}{\partial z}\right)^2 + C_k \ell_m k^{1/2} \frac{\partial^2 k}{\partial z^2} + C_k \ell_m k^{1/2} \left(\frac{\partial \langle U \rangle}{\partial z}\right)^2 - C_\epsilon \ell_m^{-1} k^{3/2}. \quad (5)$$

As can be seen the two equations are extremely non-linear. They involve non-integer powers of the unknowns and their derivatives. They also involve the multiplication of the unknowns and derivative. These equations can be linearized and solved using a finite difference scheme. Figure below shows the finite difference representation of the solution spaces for momentum and turbulent kinetic energy.

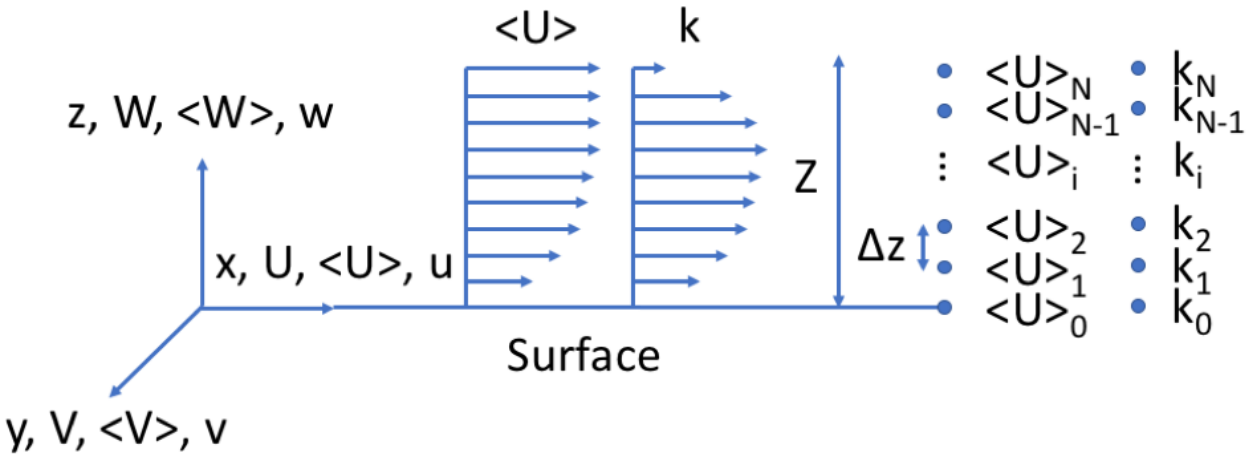


Figure 1: Schematic of 1D flow over flat surface using Cartesian coordinate system; finite difference representation of the 1D flow.

For notational convenience we can represent derivatives by superscripts and subsequently re-express the momentum and turbulent kinetic energy equations using this notation. Note that we keep the time derivatives with their original notation since this derivative involves having solutions at different timesteps, which later have to be considered for the implicit Euler method for the transient model.

$$\begin{cases} \langle U \rangle^{(0)} = \langle U \rangle, \langle U \rangle^{(1)} = \frac{\partial \langle U \rangle}{\partial z}, \langle U \rangle^{(2)} = \frac{\partial^2 \langle U \rangle}{\partial z^2} \\ k^{(0)} = k, k^{(1)} = \frac{\partial k}{\partial z}, k^{(2)} = \frac{\partial^2 k}{\partial z^2} \end{cases}$$

$$\frac{\partial \langle U \rangle}{\partial t} = \underbrace{0.5 C_k \ell_m (k^{(0)})^{-1/2} k^{(1)} \langle U \rangle^{(1)}}_{f_{1,mom.}} + \underbrace{C_k \ell_m (k^{(0)})^{1/2} \langle U \rangle^{(2)}}_{f_{2,mom.}} - \tau \quad (6)$$

$$\frac{\partial k}{\partial t} = \underbrace{0.5 C_k \ell_m (k^{(0)})^{-1/2} (k^{(1)})^2}_{f_{1,tke.}} + \underbrace{C_k \ell_m (k^{(0)})^{1/2} k^{(2)}}_{f_{2,tke.}} + \underbrace{C_k \ell_m (k^{(0)})^{1/2} (\langle U \rangle^{(1)})^2}_{f_{3,tke.}} \underbrace{- C_\epsilon \ell_m^{-1} (k^{(0)})^{3/2}}_{f_{4,tke.}}. \quad (7)$$

where each non-linear term in the equations have been renamed by a function f, Next, each f function can be replaced by its approximate using the Newton method expressing the function around an arbitrary point z_i. Beginning with the momentum equation the f function approximations are

$$
\begin{aligned}
f_{1,mom.} \approx\ & f_{1,mom.}(z_i) \\
& + \frac{\partial f_{1,mom.}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
& + \frac{\partial f_{1,mom.}}{\partial k^{(1)}}|_{z_i} \left[k^{(1)}(z) - k^{(1)}(z_i)\right] \\
& + \frac{\partial f_{1,mom.}}{\partial \langle U\rangle^{(1)}}|_{z_i} \left[\langle U\rangle^{(1)}(z) - \langle U\rangle^{(1)}(z_i)\right] \\
& = 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) \\
& - 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) [k^{(0)}(z) - k^{(0)}(z_i)] \\
& + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(1)}(z_i) [k^{(1)}(z) - k^{(1)}(z_i)] \\
& + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \left[\langle U\rangle^{(1)}(z) - \langle U\rangle^{(1)}(z_i)\right] \\
& = -0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) \\
& - 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) k^{(0)}(z) \\
& + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(1)}(z_i) k^{(1)}(z) \\
& + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z)
\end{aligned} \tag{8}
$$

$$
\begin{aligned}
f_{2,mom.} \approx\ & f_{2,mom.}(z_i) \\
& + \frac{\partial f_{2,mom.}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
& + \frac{\partial f_{2,mom.}}{\partial \langle U\rangle^{(2)}}|_{z_i} \left[\langle U\rangle^{(2)}(z) - \langle U\rangle^{(2)}(z_i)\right] \\
& = C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U\rangle^{(2)}(z_i) \\
& + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(2)}(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
& + C_k \ell_m (k^{(0)})^{1/2}(z_i) \left[\langle U\rangle^{(2)}(z) - \langle U\rangle^{(2)}(z_i)\right] \\
& = -0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(2)}(z_i) k^{(0)}(z_i) \\
& + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(2)}(z_i) k^{(0)}(z) \\
& + C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U\rangle^{(2)}(z)
\end{aligned} \tag{9}
$$

$$
\begin{aligned}
f_{1,tke.} &\approx f_{1,tke}(z_i) \\
&+ \frac{\partial f_{1,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{1,tke}}{\partial k^{(1)}}|_{z_i} \left[k^{(1)}(z) - k^{(1)}(z_i)\right] \\
&= 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(k^{(1)})^2(z_i) \\
&- 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i)(k^{(1)})^2(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \left[k^{(1)}(z) - k^{(1)}(z_i)\right] \\
&= -0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(k^{(1)})^2(z_i) \\
&- 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i)(k^{(1)})^2(z_i) k^{(0)}(z) \\
&+ C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) k^{(1)}(z)
\end{aligned} \tag{10}
$$

$$
\begin{aligned}
f_{2,tke.} &\approx f_{2,tke}(z_i) \\
&+ \frac{\partial f_{2,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{2,tke}}{\partial k^{(2)}}|_{z_i} \left[k^{(2)}(z) - k^{(2)}(z_i)\right] \\
&= C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(2)}(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ C_k \ell_m (k^{(0)})^{1/2}(z_i) \left[k^{(2)}(z) - k^{(2)}(z_i)\right] \\
&= -0.5 C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(2)}(z_i) k^{(0)}(z) \\
&+ C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z)
\end{aligned} \tag{11}
$$

$$
\begin{aligned}
f_{3,tke.} &\approx f_{3,tke}(z_i) \\
&+ \frac{\partial f_{3,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{3,tke}}{\partial \langle U \rangle^{(1)}}|_{z_i} \left[\langle U \rangle^{(1)}(z) - \langle U \rangle^{(1)}(z_i)\right] \\
&= C_k \ell_m (k^{(0)})^{1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ 2 C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U \rangle^{(1)}(z_i) \left[\langle U \rangle^{(1)}(z) - \langle U \rangle^{(1)}(z_i)\right] \\
&= -1.5 C_k \ell_m (k^{(0)})^{1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) k^{(0)}(z) \\
&+ 2 C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U \rangle^{(1)}(z_i) \langle U \rangle^{(1)}(z)
\end{aligned} \tag{12}
$$

$$
\begin{aligned}
f_{4,tke.} &\approx f_{4,tke}(z_i) \\
&+ \frac{\partial f_{4,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&= -C_\epsilon \ell_m^{-1} (k^{(0)})^{3/2}(z_i) \\
&- 1.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{1/2}(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&= 0.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{3/2}(z_i) \\
&- 1.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{1/2}(z_i) k^{(0)}(z)
\end{aligned} \tag{13}
$$

Now we can write the linearized forms of the momentum and turbulent kinetic energy equations. We will still keep the storage term, i.e. the time derivative, in differential form.

$$
\frac{\partial \langle U \rangle}{\partial t} = c_1 + c_2 \langle U \rangle^{(1)} + c_3 \langle U \rangle^{(2)} + c_4 k^{(0)} + c_5 k^{(1)} \tag{14}
$$

$$
\begin{cases}
c_1 = -0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U \rangle^{(1)}(z_i) - 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U \rangle^{(2)}(z_i) k^{(0)}(z_i) - \tau \\
c_2 = 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \\
c_3 = C_k \ell_m (k^{(0)})^{1/2}(z_i) \\
c_4 = -0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) k^{(1)}(z_i) \langle U \rangle^{(1)}(z_i) + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U \rangle^{(2)}(z_i) \\
c_5 = 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U \rangle^{(1)}(z_i)
\end{cases}
$$

$$
\frac{\partial k}{\partial t} = d_1 + d_2 k^{(0)} + d_3 k^{(1)} + d_4 k^{(2)} + d_5 \langle U \rangle^{(1)} \tag{15}
$$

$$
\begin{cases}
d_1 = -0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i) (k^{(1)})^2(z_i) - 0.5 C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z_i) \\
-1.5 C_k \ell_m (k^{(0)})^{1/2}(z_i) (\langle U \rangle^{(1)})^2(z_i) + 0.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{3/2}(z_i) \\
d_2 = -0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) (k^{(1)})^2(z_i) + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(2)}(z_i) \\
+0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) (\langle U \rangle^{(1)})^2(z_i) - 1.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{1/2}(z_i) \\
d_3 = C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \\
d_4 = C_k \ell_m (k^{(0)})^{1/2}(z_i) \\
d_5 = 2 C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U \rangle^{(1)}(z_i)
\end{cases}
$$

Now consider that we want to represent the linearized momentum and turbulent kinetic energy equations using finite differences. Consider a vertical discretization Δz as shown in the figure and temporal discretization Δt. Using central differencing the spatial derivatives can be replaced by values at indices $i-1$, i, and $i+1$. In addition, the time derivatives can be replaced by values at time levels n and $n+1$. If we use the implicit Euler method, i.e. computing spatial derivatives at

time level $n+1$, we can replace the momentum and turbulent kinetic energy equations by their discretized versions as follows

$$\frac{\langle U\rangle_i^{n+1}-\langle U\rangle_i^n}{\Delta t}=c_1+c_2\frac{\langle U\rangle_{i+1}^{n+1}-\langle U\rangle_{i-1}^{n+1}}{2\Delta z}+c_3\frac{\langle U\rangle_{i+1}^{n+1}-2\langle U\rangle_i^{n+1}+\langle U\rangle_{i-1}^{n+1}}{(\Delta z)^2}+c_4k_i^{n+1}+c_5\frac{k_{i+1}^{n+1}-k_{i-1}^{n+1}}{2\Delta z} \tag{16}$$

$$\frac{k_i^{n+1}-k_i^n}{\Delta t}=d_1+d_2k_i^{n+1}+d_3\frac{k_{i+1}^{n+1}-k_{i-1}^{n+1}}{2\Delta z}+d_4\frac{k_{i+1}^{n+1}-2k_i^{n+1}+k_{i-1}^{n+1}}{(\Delta z)^2}+d_5\frac{\langle U\rangle_{i+1}^{n+1}-\langle U\rangle_{i-1}^{n+1}}{2\Delta z} \tag{17}$$

Note that in the implicit Euler method we assume that the values of $\langle U\rangle_i$ and k_i are known at time level n, and one must solve for values at time level $n+1$. As a result, the above equations must be rearranged as follows

$$\left(\frac{c_2\Delta t}{2\Delta z}-\frac{c_3\Delta t}{(\Delta z)^2}\right)\langle U\rangle_{i-1}^{n+1}+\left(1+\frac{2c_3\Delta t}{(\Delta z)^2}\right)\langle U\rangle_i^{n+1}+\left(-\frac{c_2\Delta t}{2\Delta z}-\frac{c_3\Delta t}{(\Delta z)^2}\right)\langle U\rangle_{i+1}^{n+1}$$
$$+\left(\frac{c_5\Delta t}{2\Delta z}\right)k_{i-1}^{n+1}-c_4k_i^{n+1}+\left(-\frac{c_5\Delta t}{2\Delta z}\right)k_{i+1}^{n+1}=\Delta t(c_1+\langle U\rangle_i^n) \tag{18}$$

$$\left(\frac{d_3\Delta t}{2\Delta z}-\frac{d_4\Delta t}{(\Delta z)^2}\right)k_{i-1}^{n+1}+\left(1+\frac{2d_4\Delta t}{(\Delta z)^2}\right)k_i^{n+1}+\left(-\frac{d_3\Delta t}{2\Delta z}-\frac{d_4\Delta t}{(\Delta z)^2}\right)k_{i+1}^{n+1}$$
$$+\left(\frac{d_5\Delta t}{2\Delta z}\right)\langle U\rangle_{i-1}^{n+1}+\left(-\frac{d_5\Delta t}{2\Delta z}\right)\langle U\rangle_{i+1}^{n+1}=\Delta t(d_1+k_i^n) \tag{19}$$

As can be seen the unknowns $\langle U\rangle_i^{n+1}$ and k_i^{n+1} appear in both discretized momentum and turbulent kinetic energy equations. Therefore, these equations should be combined to arrive at a linear system of equations and subsequently solved using a linear algebra solver. Let us define the unknowns vector $\mathbf{X}$ such that

$$\mathbf{X}=\begin{bmatrix}x_0\\x_1\\\vdots\\x_{N-1}\\x_N\\x_{N+1}\\x_{N+2}\\\vdots\\x_{2N}\\x_{2N+1}\end{bmatrix}=\begin{bmatrix}\langle U\rangle_0^{n+1}\\\langle U\rangle_1^{n+1}\\\vdots\\\langle U\rangle_{N-1}^{n+1}\\\langle U\rangle_N^{n+1}\\k_0^{n+1}\\k_1^{n+1}\\\vdots\\k_{N-1}^{n+1}\\k_N^{n+1}\end{bmatrix}$$

As can be seen the first half of vector $\mathbf{X}$ contains the $\langle U \rangle_i^{n+1}$ solutions and the second half of vector $\mathbf{X}$ contains the k_i^{n+1} solutions. Note that k_i^{n+1} maps to x_{N+1+i}. Now we need $2N+2$ linear equations to solve for $\mathbf{X}$, i.e.

$$\begin{cases} \text{Equation 0: } a_{0,0}x_0 + a_{0,1}x_1 + ... + a_{0,2N}x_{2N} + a_{0,2N+1}x_{2N+1} = b_0 \\ \text{Equation 1: } a_{1,0}x_0 + a_{1,1}x_1 + ... + a_{1,2N}x_{2N} + a_{1,2N+1}x_{2N+1} = b_1 \\ ... \\ \text{Equation } i\text{: } a_{i,0}x_0 + a_{i,1}x_1 + ... + a_{i,2N}x_{2N} + a_{i,2N+1}x_{2N+1} = b_i \\ ... \\ \text{Equation } 2N\text{: } a_{2N,0}x_0 + a_{2N,1}x_1 + ... + a_{2N,2N}x_{2N} + a_{2N,2N+1}x_{2N+1} = b_{2N} \\ \text{Equation } 2N+1\text{: } a_{2N+1,0}x_0 + a_{2N+1,1}x_1 + ... + a_{2N+1,2N}x_{2N} + a_{2N+1,2N+1}x_{2N+1} = b_{2N+1} \end{cases}$$

Our next task is to identify $a_{i,j}$ and b_i. These can be inferred from the discretized momentum and turbulent kinetic energy equations. $a_{i,j}$ are mostly zero except for where there is a non-zero coefficient in the corresponding equations. The first N equations (equation 0, equation 1 ... equation N) are the momentum equations. The boundary condition for $\langle U \rangle$ at the surface provides the zeroth equation, i.e.

$$\begin{cases} x_0 = 0 \\ a_{0,0} = 1 \\ b_0 = 0 \end{cases}$$

The next $i : 1 \rightarrow N-1$ equations correspond to the momentum equation in the interior of the domain, so the coefficients can be obtained as follows

$$\begin{cases} \left(\frac{c_2\Delta t}{2\Delta z} - \frac{c_3\Delta t}{(\Delta z)^2}\right) x_{i-1} + \left(1 + \frac{2c_3\Delta t}{(\Delta z)^2}\right) x_i + \left(-\frac{c_2\Delta t}{2\Delta z} - \frac{c_3\Delta t}{(\Delta z)^2}\right) x_{i+1} \\ + \left(-\frac{c_5\Delta t}{2\Delta z}\right) x_{i-1+N+1} - c_4 x_{i+N+1} + \left(\frac{c_5\Delta t}{2\Delta z}\right) x_{i+1+N+1} = \Delta t(c_1 + \langle U \rangle_i^n) \\ a_{i,i-1} = \frac{c_2\Delta t}{2\Delta z} - \frac{c_3\Delta t}{(\Delta z)^2} \\ a_{i,i} = 1 + \frac{2c_3\Delta t}{(\Delta z)^2} \\ a_{i,i+1} = -\frac{c_2\Delta t}{2\Delta z} - \frac{c_3\Delta t}{(\Delta z)^2} \\ a_{i,i-1+N+1} = \frac{c_5\Delta t}{2\Delta z} \\ a_{i,i+N+1} = -c_4 \\ a_{i,i+1+N+1} = -\frac{c_5\Delta t}{2\Delta z} \\ b_i = \Delta t(c_1 + \langle U \rangle_i^n) \end{cases}$$

Note that where the turbulent kinetic energy term appears the index is shifted by $N+1$. The boundary condition for $\langle U \rangle$ at the top of the domain provides the N^{th} equation, i.e.

$$\begin{cases} x_{N-1} - x_N = 0 \\ a_{N,N-1} = 1 \\ a_{N,N} = -1 \\ b_N = 0 \end{cases}$$

The boundary condition for k at the surface provides the $N+1^{th}$ equation, i.e.

$$\begin{cases} x_{N+1} = 0 \\ a_{N+1,N+1} = 1 \\ b_{N+1} = 0 \end{cases}$$

The next $i : N+2 \rightarrow 2N$ equations correspond to the turbulent kinetic energy equation in the interior of the domain, so the coefficients can be obtained as follows

$$\begin{cases} \left(\frac{d_3\Delta t}{2\Delta z} - \frac{d_4\Delta t}{(\Delta z)^2}\right) x_{i-1} + \left(1 + \frac{2d_4\Delta t}{(\Delta z)^2}\right) x_i + \left(-\frac{d_3\Delta t}{2\Delta z} - \frac{d_4\Delta t}{(\Delta z)^2}\right) x_{i+1} \\ + \left(-\frac{d_5\Delta t}{2\Delta z}\right) x^{n+1}_{i-1-(N+1)} + \left(\frac{d_5\Delta t}{2\Delta z}\right) x^{n+1}_{i+1-(N+1)} = \Delta t(d_1 + k^n_{i-(N+1)}) \\ a_{i,i-1-(N+1)} = \frac{d_5\Delta t}{2\Delta z} \\ a_{i,i+1-(N+1)} = -\frac{d_5\Delta t}{2\Delta z} \\ a_{i,i-1} = \frac{d_3\Delta t}{2\Delta z} - \frac{d_4\Delta t}{(\Delta z)^2} \\ a_{i,i} = 1 + \frac{2d_4\Delta t}{(\Delta z)^2} \\ a_{i,i+1} = -\frac{d_3\Delta t}{2\Delta z} - \frac{d_4\Delta t}{(\Delta z)^2} \\ b_i = \Delta t(d_1 + k^n_{i-(N+1)}) \end{cases}$$

Note that where the momentum term appears the index is shifted by $-(N+1)$.The boundary condition for k at the top of the domain provides the $2N+1^{th}$ equation. The vertical gradient of the turbulent kinetic energy must be zero, i.e.

$$\begin{cases} x_{2N} - x_{2N+1} = 0 \\ a_{2N+1,2N} = 1 \\ a_{2N+1,2N+1} = -1 \\ b_{2N+1} = 0 \end{cases}$$

Finally, we have arrived at linear system of equations that can be solved to provide the unknowns. This system is given as follows

$$\mathbf{AX} = \mathbf{B} \tag{20}$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,2N} & a_{0,2N+1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,2N} & a_{1,2N+1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{2N,0} & a_{2N,1} & \dots & a_{2N,2N} & a_{2N,2N+1} \\ a_{2N+1,0} & a_{2N+1,1} & \dots & a_{2N+1,2N} & a_{2N+1,2N+1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{2N} \\ x_{2N+1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{2N} \\ b_{2N+1} \end{bmatrix}$$

Note that this system of equations must be solved at every time level. Once the solutions at every time level is obtained, it can then be used as initial condition and the system must be solved again for the next time level.

At each time level, note that the coefficients identified, themselves, depend on the solution. As a result the solution to the system of equations above must be found iteratively. That is, an initial guess for the solution vector $\mathbf{X}$ must be assumed. Then the system of equations must be solved iteratively. The best initial guess is the value of the solution in the previous time level. After each iteration the solutions must be updated. Subsequently, the system of equations must be solved again, until the difference between successive solutions is less than a specified error. For this purpose, the maximum norm can be considered. Suppose that a relative error of $Err = 0.01$ is specified for either the momentum or turbulent kinetic energy solution. Also suppose the x_i and $x_i^{(new)}$ represent two successive solutions for a specific point. The iteration can be stopped if the following conditions are met

$$\begin{cases} L_{\infty,mom.} = \max\left(|\frac{x_0^{(new)} - x_0}{x_0}|, |\frac{x_1^{(new)} - x_1}{x_1}|, ..., |\frac{x_N^{(new)} - x_N}{x_N}| \right) < Err \\ L_{\infty,tke} = \max\left(|\frac{x_{N+1}^{(new)} - x_{N+1}}{x_{N+1}}|, |\frac{x_{N+2}^{(new)} - x_{N+2}}{x_{N+2}}|, ..., |\frac{x_{2N+1}^{(new)} - x_{2N+1}}{x_{2N+1}}| \right) < Err \end{cases}$$

When solving a system of equations iteratively, it is sometimes more stable to only partially update a solution after each iteration. This is known as under relaxation. Consider that ϕ^{n-1} is the solution space found in the previous iteration and ϕ^{new} is the newly found solution. With the under relaxation factor $0 < \alpha < 1$, the solution can be updated for the next iteration such that

$$\phi^n = \phi^{n-1} + \alpha(\phi^{new} - \phi^{n-1}). \tag{21}$$

Particularly, whenever solving non-linear system of equations, this method improves stability of obtaining a numerical solution.

The transient simulation is desired for 4 combinations of τ and ℓ_0 shown in table below. Case 1 represents a low pressure gradient and small mixing length. Case 2 represents a high pressure gradient and small mixing length. Case 3 represents a low pressure gradient and large mixing length. Case 4 represents a high pressure gradient and large mixing length.

Table 1: Simulation cases with varying amount of horizontal pressure gradient and mixing length.

Case	τ [m s^{-2}]	ℓ_0 [m]
1	-0.005	10
2	-0.01	10
3	-0.005	20
4	-0.01	20

2 Python Script

Complete the following code.

```
import random
import sys
import os
import numpy
import matplotlib.pyplot as plt

#Define under-relaxation factor
alpha=0.7

#Define horizontal pressure gradient divided by density [m s^-2]
tau=-0.005

#Define von Karman constant
kappa=0.41

#Define maximum mixing length [m]
l0=10

#Define turbulence model constants, Martilli et al. (2002)
Ck=0.4
Ce=0.71

#Define maximum iteration number
MaxIter=10

#Define relative error
Err=0.01

#Define z axis from 0 to Z with dz increments
Z=100       #[m]
N=20
dz=Z/N      #[m]
z=numpy.linspace(0, Z, N+1)
```

```
#Define t vector from 0 to T with dt increments
T=100         #[s]
Nt=100
dt=T/Nt     #[s]
t=numpy.linspace(0, T, Nt+1)

#Define and initialize a mean velocity vector [m s^-1]
Uinitial=0.01
Umean=numpy.zeros((N+1,Nt+1))
for i in range(0, N+1):
    Umean[i][0]=Uinitial


#Define and initialize turbulent kinetic energy [m^2 s^-s]
kinitial=0.01
k=numpy.zeros((N+1,Nt+1))
for i in range(0, N+1):
    k[i][0]=kinitial

#Define unknown vector X, coefficient matrix A, and vector B, in AX=B
x=numpy.zeros((2*N+2,1))
xnew=numpy.zeros((2*N+2,1))
b=numpy.zeros((2*N+2,1))
a=numpy.zeros((2*N+2,2*N+2))

#Initialize solution vector X
for i in range(0, N+1):
    x[i]=Umean[i][0]
    x[i+N+1]=k[i][0]

#Iterate for time levels
for n in range(1, Nt+1):
    for iter in range(1, MaxIter):
        #Momentum equations
        #i = 0
        a[0][0] = 1
        b[0] = 0
        #i = 1 to N-1
        for i in range(1, N):
            #Calculate derivatives by finite differences for the current i index
            #Remember to shift indices by N+1 if needed
            lm=kappa*z[i]/(1+(kappa*z[i])/l0)
            k0=x[i+N+1]
            k1=(x[i+1+N+1]-x[i-1+N+1])/(2*dz)
            k2=(x[i+1+N+1]-2*x[i+N+1]+x[i-1+N+1])/(dz**2)
```

```
    Umean0=x[i]
    Umean1=(x[i+1]-x[i-1])/(2*dz)
    Umean2=(x[i+1]-2*x[i]+x[i-1])/(dz**2)
    # Set constants necessary to build the coefficient matrix
    c1=-0.25*Ck*lm*k0**(-1/2)*k1*Umean1-0.5*Ck*lm*k0**(-1/2)*Umean2*k0-tau
    c2=0.5*Ck*lm*k0**(-1/2)*k1
    c3=Ck*lm*k0**(1/2)
    c4=-0.25*Ck*lm*k0**(-3/2)*k1*Umean1+0.5*Ck*lm*k0**(-1/2)*Umean2
    c5=0.5*Ck*lm*k0**(-1/2)*Umean1
    # Set the coefficient matrix and the B vector
    a[i][i-1]=c2*dt/(2*dz)-c3*dt/(dz**2)
    a[i][i]=1+2*c3*dt/(dz**2)
    a[i][i+1]=-c2*dt/(2*dz)-c3*dt/(dz**2)
    a[i][i-1+N+1]=c5*dt/(2*dz)
    a[i][i+N+1]=-c4
    a[i][i+1+N+1]=-c5*dt/(2*dz)
    b[i]=dt*(c1+Umean[i][n-1])
#i=N
a[N][N-1]=1
a[N][N]=-1
b[N]=0


#Turbulent kinetic energy equations
#i=N+1
a[N+1][N+1]=...
b[N+1]=...
#i=N+2 to 2N
for i in range(N+2,2*N+1):
    #Calculate derivatives by finite differences for the current i index
    # Shift indices by -(N+1) if needed
    lm=kappa*z[i-(N+1)]/(1+(kappa*z[i-(N+1)])/l0)
    k0=x[i]
    k1=(x[i+1]-x[i-1])/(2*dz)
    k2=(x[i+1]-2*x[i]+x[i-1])/(dz**2)
    Umean0=x[i-(N+1)]
    Umean1=(x[i+1-(N+1)]-x[i-1-(N+1)])/(2*dz)
    Umean2=(x[i+1-(N+1)]-2*x[i-(N+1)]+x[i-1-(N+1)])/(dz**2)
    #Set constants necessary to build the coefficient matrix
    d1=-0.25*Ck*lm*k0**(-1/2)*k1**2-0.5*Ck*lm*k0**(1/2)*k2\
        -1.5*Ck*lm*k0**(1/2)*Umean1**2+0.5*Ce*lm**(-1)*k0**(3/2)
    d2=-0.25*Ck*lm*k0**(-3/2)*k1**2+0.5*Ck*lm*k0**(-1/2)*k2\
        +0.5*Ck*lm*k0**(-1/2)*Umean1**2-1.5*Ce*lm**(-1)*k0**(1/2)
    d3=Ck*lm*k0**(-1/2)*k1
    d4=Ck*lm*k0**(1/2)
    d5=2*Ck*lm*k0**(1/2)*Umean1
```

```
            # Set the coefficient matrix and the B vector
            a[i][i-1-(N+1)]=d5*dt/(2*dz)
            a[i][i+1-(N+1)]=-d5*dt/(2*dz)
            a[i][i-1]=d3*dt/(2*dz)-d4*dt/(dz**2)
            a[i][i]=1+2*d4*dt/(dz**2)
            a[i][i+1]=-d3*dt/(2*dz)-d4*dt/(dz**2)
            b[i]=dt*(d1+k[i-(N+1)][n-1])
        #i=2N+1
        a[2*N+1][2*N]=...
        a[2*N+1][2*N+1]=...
        b[2*N+1]=...

        xnew=numpy.linalg.solve(a,b)

        #Calculate maximum norm errors for both momentum and turbulent kinetic energy
        ErrUmean=numpy.max(numpy.abs(numpy.divide(xnew[1:N+1]-x[1:N+1],x[1:N+1])))
        Errk=...

        print('t=',t[n],' Iteration=',iter,' ErrUmean=',ErrUmean,' Errk=',Errk)

        if ErrUmean < Err and Errk < Err:
            print('\n')
            # Exit the loop
            break

        # Update solution
        x[:] = x[:]+alpha*(xnew[:]-x[:])

    #Before next time level,
    #assign the X vector to the original Umean and k vectors
    for i in range(0, N+1):
        Umean[i][n]=x[i]
        k[i][n]=x[i+N+1]

#Plot the mean velocity versus z
plt.plot(Umean[:,1],z,label='t='+str(t[1])+' s')
plt.plot(Umean[:,25],z,label='t='+str(t[25])+' s')
plt.plot(Umean[:,50],z,label='t='+str(t[50])+' s')
plt.plot(Umean[:,75],z,label='t='+str(t[75])+' s')
plt.plot(Umean[:,100],z,label='t='+str(t[100])+' s')
plt.xlabel('<U> [m s^-1]')
plt.ylabel('z [m]')
plt.title('Mean Velocity as Function of z')
plt.legend()
plt.show()
```

```
#Plot the turbulent kinetic energy versus z
plt.plot(...
plt.plot(...
plt.plot(...
plt.plot(...
plt.plot(...
plt.xlabel('k [m^2 s^-2]')
plt.ylabel('z [m]')
plt.title('Turbulent Kinetic Energy as Function of z')
plt.legend()
plt.show()
```

Upon completing the code. You should get the following output from the console as well as the following figures. For each timestep the convergence criteria is printed to the screen to monitor the solution behaviour. Note that with only a few iterations using the Newton method, it is possible to converge to a solution with a relative error less than 1%.

```
...
t= 99.0  Iteration= 1  ErrUmean= 0.0126885840809  Errk= 0.0337588457361
t= 99.0  Iteration= 2  ErrUmean= 0.00377306284665  Errk= 0.0114400382904
t= 99.0  Iteration= 3  ErrUmean= 0.00112893716837  Errk= 0.00395004802126

t= 100.0  Iteration= 1  ErrUmean= 0.0125437463153  Errk= 0.0344018672795
t= 100.0  Iteration= 2  ErrUmean= 0.00373036893349  Errk= 0.0116863554115
t= 100.0  Iteration= 3  ErrUmean= 0.00111619600402  Errk= 0.0040461622013
```

Try to answer the following questions.

- What are the effects of horizontal modified pressure gradient and maximum mixing length on the solutions?
- At what height is the turbulent kinetic energy at its maximum?
- How do you compare the results of this simulation to the results obtained by the mixing length model?
- Try to change other simulation variables such as `Z`, `N`, `T`, and `Nt` until the code diverges. Can you explain why you are not getting a converged solution in these cases?

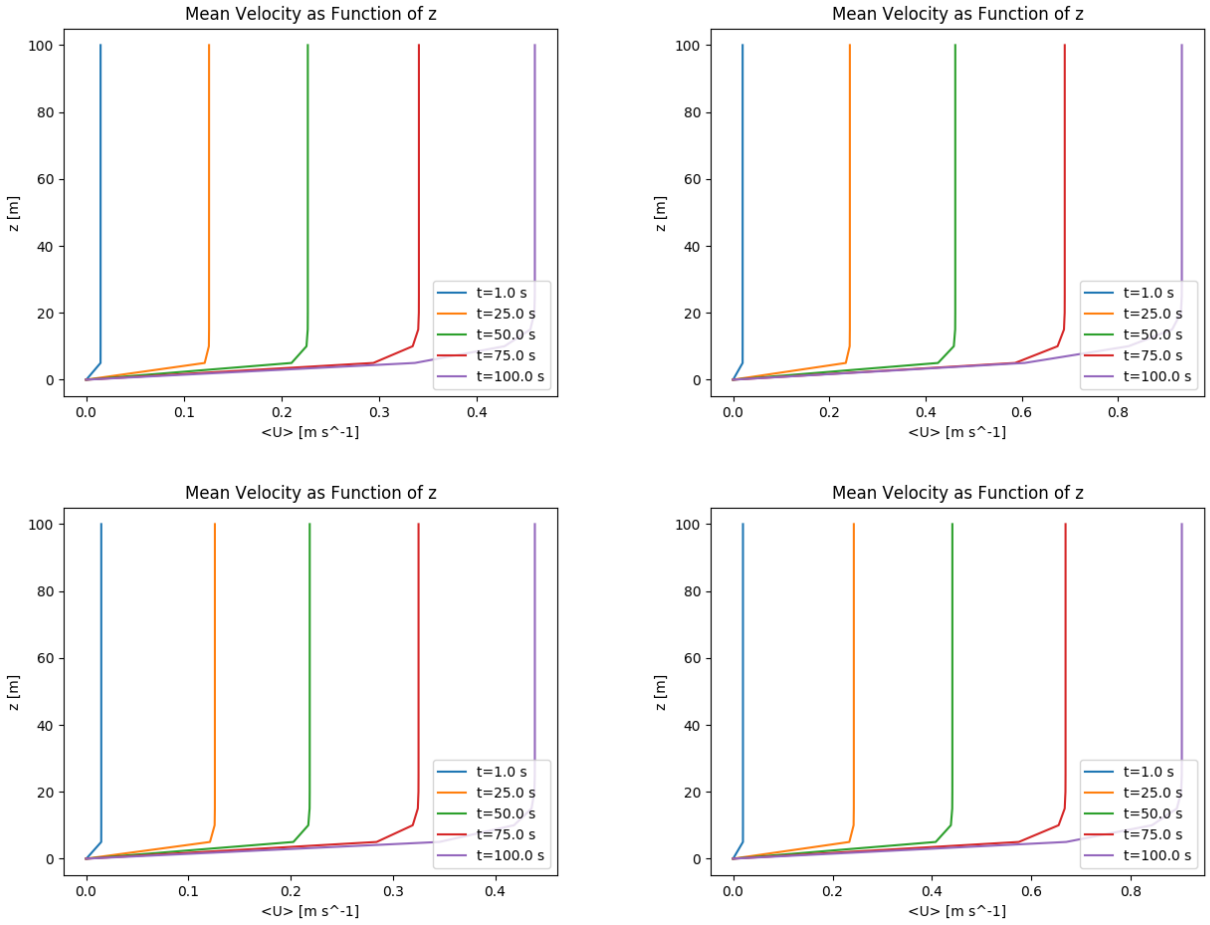


Figure 2: Momentum for Case1 (top left), Case 2 (top right), Case 3 (bottom left), Case 4 (bottom right)

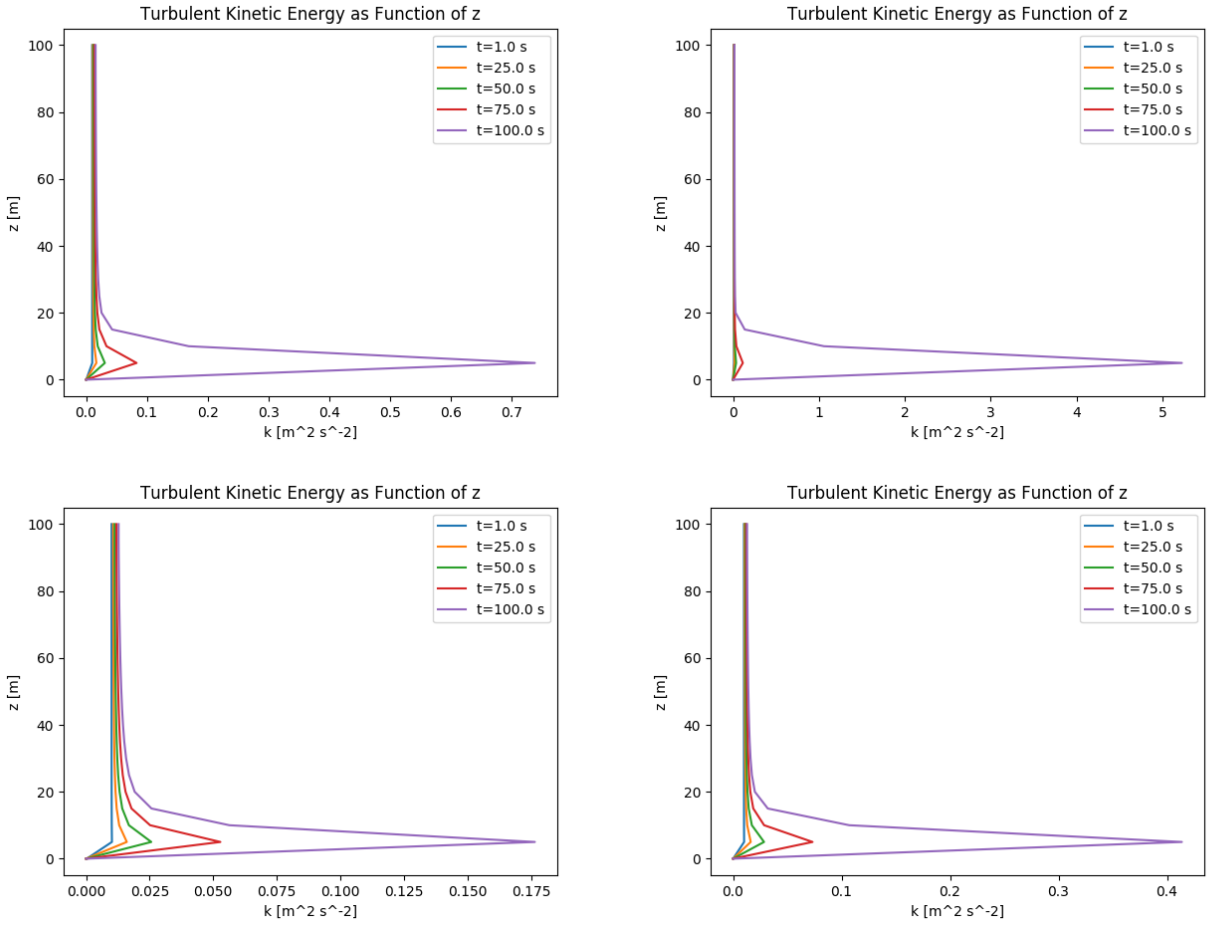


Figure 3: Turbulent viscosity for Case1 (top left), Case 2 (top right), Case 3 (bottom left), Case 4 (bottom right)

ENGG*6790: Theory and Applications of Turbulence

1D Momentum, Turbulent Kinetic Energy, and Heat Equations over Flat Surface with Steady Formulation

Amir A. Aliabadi

March 14, 2019

1 Introduction

In the lectures the transport equation for heat (temperature) was obtained using the gradient diffusion hypothesis. This hypothesis gives the transport equation and the effective diffusivity as

$$\underbrace{\frac{\overline{D}\langle T\rangle}{\overline{D}t}}_{\text{Material Derivative of Mean}} = \underbrace{\nabla.(\Gamma_{eff}\nabla\langle T\rangle)}_{\text{Diffusion of Mean}}, \tag{1}$$

$$\underbrace{\Gamma_{eff}(\mathbf{x},t)}_{\text{Effective Diffusivity}} = \underbrace{\Gamma}_{\text{Molecular Diffusivity}} + \underbrace{\Gamma_T(\mathbf{x},t)}_{\text{Turbulent Diffusivity}} \approx \Gamma_T(\mathbf{x},t), \tag{2}$$

where the molecular diffusivity has been ignored in a highly turbulent field in comparison to the turbulent diffusivity. Assuming that the turbulent Prandtl number is approximately equal to one the effective diffusivity for the heat equation can be given as

$$\Gamma_T = \frac{\nu_T}{Pr_T} \approx \frac{\nu_T}{1} = \nu_T. \tag{3}$$

This model can be employed to develop a one-dimensional transport model for heat transport under steady state conditions. Suppose we use the Cartesian coordinate system with coordinate axes of x, y, and z, and velocities corresponding to these axes being $U = \langle U\rangle + u$, $V = \langle V\rangle + v$, $W = \langle W\rangle + w$, respectively, as shown in the schematic. We can assume that mean flow is only in the x direction parallel to the surface and that the direction z is normal to the surface so that $\langle V\rangle = \langle W\rangle = 0$. The transport equation for heat can be given by

$$0 = \underbrace{\frac{\partial}{\partial z}\left(\nu_T \frac{\partial \langle T \rangle}{\partial z}\right)}_{\text{Diffusion of Mean}} - \underbrace{\gamma}_{\text{Heat Sink or Source}}, \tag{4}$$

where the term γ is a sink or source for temperature by uniform cooling or heating in the domain. The turbulent viscosity can be obtained solving the transport equations for momentum and the turbulent kinetic energy.

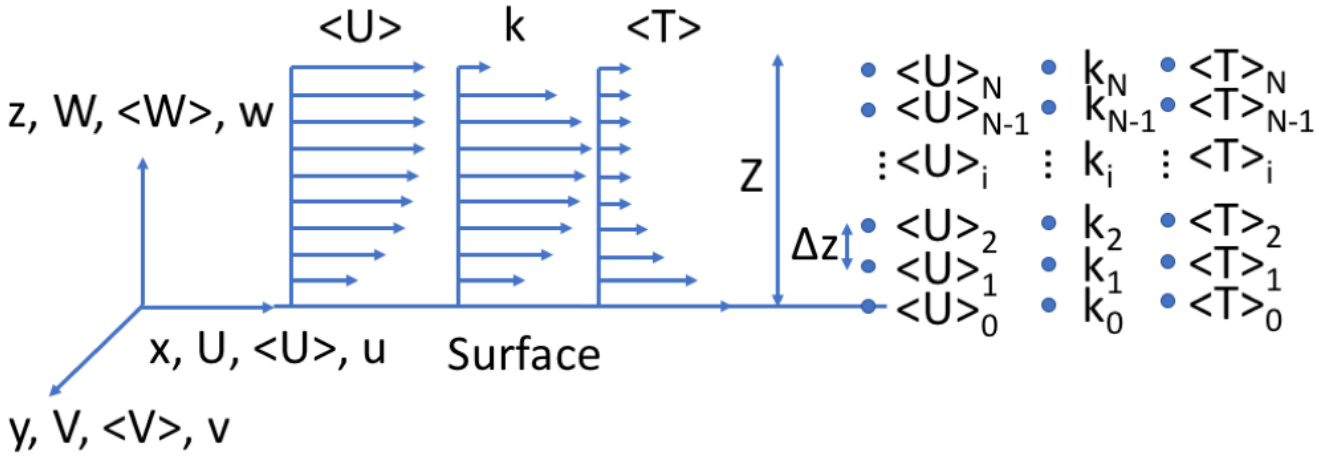


Figure 1: Schematic of 1D flow and heat transport over flat surface using Cartesian coordinate system; finite difference representation of the 1D flow and heat transport.

In the lectures the turbulent kinetic energy model was introduced as one transport equation to predict the turbulent kinetic energy. This turbulent kinetic energy was then used to formulate turbulent viscosity so that the momentum equation can be solved. The turbulent kinetic energy equation is given as

$$\underbrace{\frac{\overline{D}k}{\overline{D}t}}_{\text{Material Derivative}} \equiv \underbrace{\frac{\partial k}{\partial t}}_{\text{Storage}} + \underbrace{\langle U \rangle . \nabla k}_{\text{Advection}} = \underbrace{\nabla . \left(\frac{\nu_T}{\sigma_k} \nabla k\right)}_{\text{Energy Flux Divergence}} + \underbrace{\mathcal{P}}_{\text{Production}} - \underbrace{\epsilon}_{\text{Dissipation}}, \tag{5}$$

$$\begin{cases} \nu_T = c k^{1/2} \ell_m, \\ \epsilon = C_D \frac{k^{3/2}}{\ell_m}, \\ \ell_m(\mathbf{x}, t) \text{ known}. \end{cases}$$

Assume that the modified pressure has a constant gradient in the x direction, the 1D momentum equation then simplifies to

$$0 = \underbrace{\frac{\partial}{\partial z}\left(\nu_T \frac{\partial \langle U \rangle}{\partial z}\right)}_{\text{Surface Forces and Reynolds Stress}} - \underbrace{\tau}_{\text{Modified Pressure Forces}} \tag{6}$$

The one-dimensional turbulent kinetic energy equation can be developed as follows

$$0 = \underbrace{\frac{\partial}{\partial z}\left(\frac{\nu_T}{\sigma_k}\frac{\partial k}{\partial z}\right)}_{\text{Energy Flux Divergence}} + \underbrace{\nu_T\left(\frac{\partial \langle U\rangle}{\partial z}\right)^2}_{\text{Shear Production}} \underbrace{-\frac{g}{T_0}\frac{\nu_T}{Pr_T}\frac{\partial \langle T\rangle}{\partial z}}_{\text{Buoyant Production or Sink}} - \underbrace{\epsilon}_{\text{Dissipation}} \tag{7}$$

where the energy flux divergence was discussed in the lectures. This term ensures that the resulting model transport equation for k yields smooth solutions, and that a boundary condition can be imposed on k everywhere in the boundary of the domain. Otherwise the model may diverge if other transport mechanisms for k are much smaller than this term. The shear production term, is an example of a production term $\mathcal{P}$, that contributes to the generation of the turbulent kinetic energy. Here, when there is non-zero mean velocity gradient, turbulent kinetic energy is generated. The buoyant production or sink term, is an example of a production term $\mathcal{P}$, that contributes to the generation or sink of the turbulent kinetic energy. Here, when there is non-zero mean temperature gradient, turbulent kinetic energy is generated, when there is negative vertical gradient for mean temperature, or sunk, when there is positive vertical gradient for mean temperature. T_0 is an average temperature in the domain that can be assumed as a constant. The dissipation term is responsible for consuming turbulent kinetic energy down the energy cascade.

To close the turbulence model we can assume that the turbulent Prandtl number is unity, i.e. $Pr_T = \sigma_k = 1$. We can model turbulent viscosity, dissipation rate, and the appropriate mixing length as follows.

$$\begin{cases} \nu_T = C_k \ell_m k^{1/2}, \\ \epsilon = C_\epsilon \ell_m^{-1} k^{3/2}, \\ \ell_m = \kappa z/(1 + \frac{\kappa z}{\ell_0}). \end{cases}$$

where $\kappa = 0.41$ is the von Kármán constant, and ℓ_0 is the maximum mixing length. This formulation for mixing length has the nice property that it is bounded between zero and ℓ_0, which is physically sound since mixing length increases linearly in the log-law sublayer near a wall but cannot increase indefinitely in the interior of the domain. This formulation results in

$$\begin{cases} z \to 0 & \ell_m \to \kappa z \\ z \to \infty & \ell_m \to \ell_0 \end{cases}$$

So we have three equations: momentum, turbulent kinetic energy, and heat. We can eliminate ν_T and ϵ from these equations by direct substitutions and simplifications using the chain rule. So the equations can be re-expressed as

$$\begin{aligned} 0 &= \frac{\partial}{\partial z}\left(C_k \ell_m k^{1/2}\frac{\partial \langle U\rangle}{\partial z}\right) - \tau \\ &= 0.5 C_k \ell_m k^{-1/2}\frac{\partial k}{\partial z}\frac{\partial \langle U\rangle}{\partial z} + C_k \ell_m k^{1/2}\frac{\partial^2 \langle U\rangle}{\partial z^2} - \tau, \end{aligned} \tag{8}$$

$$
\begin{aligned}
0 &= \frac{\partial}{\partial z}\left(C_k \ell_m k^{1/2} \frac{\partial k}{\partial z}\right) + C_k \ell_m k^{1/2} \left(\frac{\partial \langle U \rangle}{\partial z}\right)^2 - C_\epsilon \ell_m^{-1} k^{3/2} \\
&= 0.5 C_k \ell_m k^{-1/2} \left(\frac{\partial k}{\partial z}\right)^2 + C_k \ell_m k^{1/2} \frac{\partial^2 k}{\partial z^2} + C_k \ell_m k^{1/2} \left(\frac{\partial \langle U \rangle}{\partial z}\right)^2 - g T_0^{-1} C_k \ell_m k^{1/2} \frac{\partial \langle T \rangle}{\partial z} - C_\epsilon \ell_m^{-1} k^{3/2},
\end{aligned} \tag{9}
$$

$$
\begin{aligned}
0 &= \frac{\partial}{\partial z}\left(C_k \ell_m k^{1/2} \frac{\partial \langle T \rangle}{\partial z}\right) - \gamma \\
&= 0.5 C_k \ell_m k^{-1/2} \frac{\partial k}{\partial z} \frac{\partial \langle T \rangle}{\partial z} + C_k \ell_m k^{1/2} \frac{\partial^2 \langle T \rangle}{\partial z^2} - \gamma.
\end{aligned} \tag{10}
$$

As can be seen these equations are extremely non-linear. They involve non-integer powers of the unknowns and their derivatives. They also involve the multiplication of the unknowns and derivatives. These equations can be linearized and solved using a finite difference scheme. Figure above shows the finite difference representation of the solution spaces for momentum, turbulent kinetic energy, and temperature.

For notational convenience we can represent derivatives by superscripts and subsequently re-express the equations using this notation.

$$
\begin{cases}
\langle U \rangle^{(0)} = \langle U \rangle, \langle U \rangle^{(1)} = \frac{\partial \langle U \rangle}{\partial z}, \langle U \rangle^{(2)} = \frac{\partial^2 \langle U \rangle}{\partial z^2} \\
k^{(0)} = k, k^{(1)} = \frac{\partial k}{\partial z}, k^{(2)} = \frac{\partial^2 k}{\partial z^2} \\
\langle T \rangle^{(0)} = \langle T \rangle, \langle T \rangle^{(1)} = \frac{\partial \langle T \rangle}{\partial z}, \langle T \rangle^{(2)} = \frac{\partial^2 \langle T \rangle}{\partial z^2}
\end{cases}
$$

$$
0 = \underbrace{0.5 C_k \ell_m (k^{(0)})^{-1/2} k^{(1)} \langle U \rangle^{(1)}}_{f_{1,mom.}} + \underbrace{C_k \ell_m (k^{(0)})^{1/2} \langle U \rangle^{(2)}}_{f_{2,mom.}} - \tau, \tag{11}
$$

$$
\begin{aligned}
0 = &\underbrace{0.5 C_k \ell_m (k^{(0)})^{-1/2} (k^{(1)})^2}_{f_{1,tke.}} + \underbrace{C_k \ell_m (k^{(0)})^{1/2} k^{(2)}}_{f_{2,tke.}} \\
&+ \underbrace{C_k \ell_m (k^{(0)})^{1/2} (\langle U \rangle^{(1)})^2}_{f_{3,tke.}} \underbrace{- g T_0^{-1} C_k \ell_m (k^{(0)})^{1/2} \langle T \rangle^{(1)}}_{f_{4,tke.}} \underbrace{- C_\epsilon \ell_m^{-1} (k^{(0)})^{3/2}}_{f_{5,tke.}},
\end{aligned} \tag{12}
$$

$$
0 = \underbrace{0.5 C_k \ell_m (k^{(0)})^{-1/2} k^{(1)} \langle T \rangle^{(1)}}_{f_{1,hea.}} + \underbrace{C_k \ell_m (k^{(0)})^{1/2} \langle T \rangle^{(2)}}_{f_{2,hea.}} - \gamma. \tag{13}
$$

where each non-linear term in the equations have been renamed by a function f, Next, each f function can be replaced by its approximate using the Newton method expressing the function around an arbitrary point z_i. Beginning with the momentum equation the f function approximations are

$$
\begin{aligned}
f_{1,mom.} &\approx f_{1,mom.}(z_i) \\
&+ \frac{\partial f_{1,mom.}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{1,mom.}}{\partial k^{(1)}}|_{z_i} \left[k^{(1)}(z) - k^{(1)}(z_i)\right] \\
&+ \frac{\partial f_{1,mom.}}{\partial \langle U\rangle^{(1)}}|_{z_i} \left[\langle U\rangle^{(1)}(z) - \langle U\rangle^{(1)}(z_i)\right] \\
&= 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) \\
&- 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) [k^{(0)}(z) - k^{(0)}(z_i)] \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(1)}(z_i) [k^{(1)}(z) - k^{(1)}(z_i)] \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \left[\langle U\rangle^{(1)}(z) - \langle U\rangle^{(1)}(z_i)\right] \\
&= -0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) \\
&- 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) k^{(0)}(z) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(1)}(z_i) k^{(1)}(z) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z)
\end{aligned} \tag{14}
$$

$$
\begin{aligned}
f_{2,mom.} &\approx f_{2,mom.}(z_i) \\
&+ \frac{\partial f_{2,mom.}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{2,mom.}}{\partial \langle U\rangle^{(2)}}|_{z_i} \left[\langle U\rangle^{(2)}(z) - \langle U\rangle^{(2)}(z_i)\right] \\
&= C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U\rangle^{(2)}(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(2)}(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ C_k \ell_m (k^{(0)})^{1/2}(z_i) \left[\langle U\rangle^{(2)}(z) - \langle U\rangle^{(2)}(z_i)\right] \\
&= -0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(2)}(z_i) k^{(0)}(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(2)}(z_i) k^{(0)}(z) \\
&+ C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U\rangle^{(2)}(z)
\end{aligned} \tag{15}
$$

$$
\begin{aligned}
f_{1,tke.} &\approx f_{1,tke}(z_i) \\
&+ \frac{\partial f_{1,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{1,tke}}{\partial k^{(1)}}|_{z_i} \left[k^{(1)}(z) - k^{(1)}(z_i)\right] \\
&= 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(k^{(1)})^2(z_i) \\
&- 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i)(k^{(1)})^2(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \left[k^{(1)}(z) - k^{(1)}(z_i)\right] \\
&= -0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(k^{(1)})^2(z_i) \\
&- 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i)(k^{(1)})^2(z_i) k^{(0)}(z) \\
&+ C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) k^{(1)}(z)
\end{aligned} \tag{16}
$$

$$
\begin{aligned}
f_{2,tke.} &\approx f_{2,tke}(z_i) \\
&+ \frac{\partial f_{2,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{2,tke}}{\partial k^{(2)}}|_{z_i} \left[k^{(2)}(z) - k^{(2)}(z_i)\right] \\
&= C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(2)}(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ C_k \ell_m (k^{(0)})^{1/2}(z_i) \left[k^{(2)}(z) - k^{(2)}(z_i)\right] \\
&= -0.5 C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(2)}(z_i) k^{(0)}(z) \\
&+ C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z)
\end{aligned} \tag{17}
$$

$$
\begin{aligned}
f_{3,tke.} &\approx f_{3,tke}(z_i) \\
&+ \frac{\partial f_{3,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{3,tke}}{\partial \langle U \rangle^{(1)}}|_{z_i} \left[\langle U \rangle^{(1)}(z) - \langle U \rangle^{(1)}(z_i)\right] \\
&= C_k \ell_m (k^{(0)})^{1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ 2 C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U \rangle^{(1)}(z_i) \left[\langle U \rangle^{(1)}(z) - \langle U \rangle^{(1)}(z_i)\right] \\
&= -1.5 C_k \ell_m (k^{(0)})^{1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) k^{(0)}(z) \\
&+ 2 C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U \rangle^{(1)}(z_i) \langle U \rangle^{(1)}(z)
\end{aligned} \tag{18}
$$

$$
\begin{aligned}
f_{4,tke.} &\approx f_{4,tke}(z_i) \\
&+ \frac{\partial f_{4,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{4,tke}}{\partial \langle T\rangle^{(1)}}|_{z_i} \left[\langle T\rangle^{(1)}(z) - \langle T\rangle^{(1)}(z_i)\right] \\
&= -gT_0^{-1}C_k\ell_m(k^{(0)})^{1/2}(z_i)\langle T\rangle^{(1)}(z_i) \\
&- 0.5gT_0^{-1}C_k\ell_m(k^{(0)})^{-1/2}(z_i)\langle T\rangle^{(1)}(z_i)\left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&- gT_0^{-1}C_k\ell_m(k^{(0)})^{1/2}(z_i)\left[\langle T\rangle^{(1)}(z) - \langle T\rangle^{(1)}(z_i)\right] \\
&= 0.5gT_0^{-1}C_k\ell_m(k^{(0)})^{1/2}(z_i)\langle T\rangle^{(1)}(z_i) \\
&- 0.5gT_0^{-1}C_k\ell_m(k^{(0)})^{-1/2}(z_i)\langle T\rangle^{(1)}(z_i)k^{(0)}(z) \\
&- gT_0^{-1}C_k\ell_m(k^{(0)})^{1/2}(z_i)\langle T\rangle^{(1)}(z)
\end{aligned} \tag{19}
$$

$$
\begin{aligned}
f_{5,tke.} &\approx f_{4,tke}(z_i) \\
&+ \frac{\partial f_{4,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&= -C_\epsilon\ell_m^{-1}(k^{(0)})^{3/2}(z_i) \\
&- 1.5C_\epsilon\ell_m^{-1}(k^{(0)})^{1/2}(z_i)\left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&= 0.5C_\epsilon\ell_m^{-1}(k^{(0)})^{3/2}(z_i) \\
&- 1.5C_\epsilon\ell_m^{-1}(k^{(0)})^{1/2}(z_i)k^{(0)}(z)
\end{aligned} \tag{20}
$$

$$
\begin{aligned}
f_{1,hea.} &\approx f_{1,hea.}(z_i) \\
&+ \frac{\partial f_{1,hea.}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{1,hea.}}{\partial k^{(1)}}|_{z_i} \left[k^{(1)}(z) - k^{(1)}(z_i)\right] \\
&+ \frac{\partial f_{1,hea.}}{\partial \langle T\rangle^{(1)}}|_{z_i} \left[\langle T\rangle^{(1)}(z) - \langle T\rangle^{(1)}(z_i)\right] \\
&= 0.5C_k\ell_m(k^{(0)})^{-1/2}(z_i)k^{(1)}(z_i)\langle T\rangle^{(1)}(z_i) \\
&- 0.25C_k\ell_m(k^{(0)})^{-3/2}(z_i)k^{(1)}(z_i)\langle T\rangle^{(1)}(z_i)[k^{(0)}(z) - k^{(0)}(z_i)] \\
&+ 0.5C_k\ell_m(k^{(0)})^{-1/2}(z_i)\langle T\rangle^{(1)}(z_i)[k^{(1)}(z) - k^{(1)}(z_i)] \\
&+ 0.5C_k\ell_m(k^{(0)})^{-1/2}(z_i)k^{(1)}(z_i)\left[\langle T\rangle^{(1)}(z) - \langle T\rangle^{(1)}(z_i)\right] \\
&= -0.25C_k\ell_m(k^{(0)})^{-1/2}(z_i)k^{(1)}(z_i)\langle T\rangle^{(1)}(z_i) \\
&- 0.25C_k\ell_m(k^{(0)})^{-3/2}(z_i)k^{(1)}(z_i)\langle T\rangle^{(1)}(z_i)k^{(0)}(z) \\
&+ 0.5C_k\ell_m(k^{(0)})^{-1/2}(z_i)\langle T\rangle^{(1)}(z_i)k^{(1)}(z) \\
&+ 0.5C_k\ell_m(k^{(0)})^{-1/2}(z_i)k^{(1)}(z_i)\langle T\rangle^{(1)}(z)
\end{aligned} \tag{21}
$$

$$
\begin{aligned}
f_{2,hea.} \approx & f_{2,hea.}(z_i) \\
& + \frac{\partial f_{2,hea.}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
& + \frac{\partial f_{2,hea.}}{\partial \langle T\rangle^{(2)}}|_{z_i} \left[\langle T\rangle^{(2)}(z) - \langle T\rangle^{(2)}(z_i)\right] \\
& = C_k \ell_m (k^{(0)})^{1/2}(z_i)\langle T\rangle^{(2)}(z_i) \\
& + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)\langle T\rangle^{(2)}(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
& + C_k \ell_m (k^{(0)})^{1/2}(z_i) \left[\langle T\rangle^{(2)}(z) - \langle T\rangle^{(2)}(z_i)\right] \\
& = -0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)\langle T\rangle^{(2)}(z_i) k^{(0)}(z_i) \\
& + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)\langle T\rangle^{(2)}(z_i) k^{(0)}(z) \\
& + C_k \ell_m (k^{(0)})^{1/2}(z_i)\langle T\rangle^{(2)}(z)
\end{aligned} \tag{22}
$$

Now we can write the linearized forms of the equations.

$$
c_1 \langle U\rangle^{(1)} + c_2 \langle U\rangle^{(2)} + c_3 k^{(0)} + c_4 k^{(1)} = c_b \tag{23}
$$

$$
\begin{cases}
c_1 = 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \\
c_2 = C_k \ell_m (k^{(0)})^{1/2}(z_i) \\
c_3 = -0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) k^{(1)}(z_i)\langle U\rangle^{(1)}(z_i) + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)\langle U\rangle^{(2)}(z_i) \\
c_4 = 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)\langle U\rangle^{(1)}(z_i) \\
c_b = -\left[-0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i)\langle U\rangle^{(1)}(z_i) - 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)\langle U\rangle^{(2)}(z_i) k^{(0)}(z_i) - \tau\right]
\end{cases}
$$

$$
d_1 k^{(0)} + d_2 k^{(1)} + d_3 k^{(2)} + d_4 \langle U\rangle^{(1)} + d_5 \langle T\rangle^{(1)} = d_b \tag{24}
$$

$$
\begin{cases}
d_1 = -0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i)(k^{(1)})^2(z_i) + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(2)}(z_i) \\
+0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(\langle U\rangle^{(1)})^2(z_i) - 0.5 g T_0^{-1} C_k \ell_m (k^{(0)})^{-1/2}(z_i)\langle T\rangle^{(1)}(z_i) \\
-1.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{1/2}(z_i) \\
d_2 = C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \\
d_3 = C_k \ell_m (k^{(0)})^{1/2}(z_i) \\
d_4 = 2 C_k \ell_m (k^{(0)})^{1/2}(z_i)\langle U\rangle^{(1)}(z_i) \\
d_5 = -g T_0^{-1} C_k \ell_m (k^{(0)})^{1/2}(z_i) \\
d_b = -\left[-0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(k^{(1)})^2(z_i) - 0.5 C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z_i)\right] \\
-\left[-1.5 C_k \ell_m (k^{(0)})^{1/2}(z_i)(\langle U\rangle^{(1)})^2(z_i) + 0.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{3/2}(z_i)\right] \\
-\left[0.5 g T_0^{-1} C_k \ell_m (k^{(0)})^{1/2}(z_i)\langle T\rangle^{(1)}(z_i)\right]
\end{cases}
$$

$$e_1\langle T\rangle^{(1)} + e_2\langle T\rangle^{(2)} + e_3 k^{(0)} + e_4 k^{(1)} = e_b \tag{25}$$

$$\begin{cases} e_1 = 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \\ e_2 = C_k \ell_m (k^{(0)})^{1/2}(z_i) \\ e_3 = -0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) k^{(1)}(z_i) \langle T\rangle^{(1)}(z_i) + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle T\rangle^{(2)}(z_i) \\ e_4 = 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle T\rangle^{(1)}(z_i) \\ e_b = -\left[-0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle T\rangle^{(1)}(z_i) - 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle T\rangle^{(2)}(z_i) k^{(0)}(z_i) - \gamma\right] \end{cases}$$

Now consider that we want to represent the linearized momentum and turbulent kinetic energy equations using finite differences. Consider a vertical discretization Δz as shown in the figure. Using central differencing the spatial derivatives can be replaced by values at indices $i-1$, i, and $i+1$. We can replace the equations by their discretized versions as follows

$$c_1 \frac{\langle U\rangle_{i+1} - \langle U\rangle_{i-1}}{2\Delta z} + c_2 \frac{\langle U\rangle_{i+1} - 2\langle U\rangle_i + \langle U\rangle_{i-1}}{(\Delta z)^2} + c_3 k_i + c_4 \frac{k_{i+1} - k_{i-1}}{2\Delta z} = c_b, \tag{26}$$

$$d_1 k_i + d_2 \frac{k_{i+1} - k_{i-1}}{2\Delta z} + d_3 \frac{k_{i+1} - 2k_i + k_{i-1}}{(\Delta z)^2} + d_4 \frac{\langle U\rangle_{i+1} - \langle U\rangle_{i-1}}{2\Delta z} + d_5 \frac{\langle T\rangle_{i+1} - \langle T\rangle_{i-1}}{2\Delta z} = d_b, \tag{27}$$

$$e_1 \frac{\langle T\rangle_{i+1} - \langle T\rangle_{i-1}}{2\Delta z} + e_2 \frac{\langle T\rangle_{i+1} - 2\langle T\rangle_i + \langle T\rangle_{i-1}}{(\Delta z)^2} + e_3 k_i + e_4 \frac{k_{i+1} - k_{i-1}}{2\Delta z} = e_b. \tag{28}$$

The above equations must be rearranged as follows.

$$\begin{aligned} &\left(-\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2}\right)\langle U\rangle_{i-1} + \left(-\frac{2c_2}{(\Delta z)^2}\right)\langle U\rangle_i + \left(\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2}\right)\langle U\rangle_{i+1} \\ &\qquad + \left(-\frac{c_4}{2\Delta z}\right) k_{i-1} + c_3 k_i + \left(\frac{c_4}{2\Delta z}\right) k_{i+1} = c_b, \end{aligned} \tag{29}$$

$$\begin{aligned} &\left(-\frac{d_2}{2\Delta z} + \frac{d_3}{(\Delta z)^2}\right) k_{i-1} + \left(d_1 - \frac{2d_3}{(\Delta z)^2}\right) k_i + \left(\frac{d_2}{2\Delta z} + \frac{d_3}{(\Delta z)^2}\right) k_{i+1} \\ &\qquad + \left(-\frac{d_4}{2\Delta z}\right)\langle U\rangle_{i-1} + \left(\frac{d_4}{2\Delta z}\right)\langle U\rangle_{i+1} \\ &\qquad + \left(-\frac{d_5}{2\Delta z}\right)\langle T\rangle_{i-1} + \left(\frac{d_5}{2\Delta z}\right)\langle T\rangle_{i+1} = d_b, \end{aligned} \tag{30}$$

$$\left(-\frac{e_1}{2\Delta z}+\frac{e_2}{(\Delta z)^2}\right)\langle T\rangle_{i-1}+\left(-\frac{2e_2}{(\Delta z)^2}\right)\langle T\rangle_i+\left(\frac{e_1}{2\Delta z}+\frac{e_2}{(\Delta z)^2}\right)\langle T\rangle_{i+1}$$
$$+\left(-\frac{e_4}{2\Delta z}\right)k_{i-1}+e_3k_i+\left(\frac{e_4}{2\Delta z}\right)k_{i+1}=e_b. \tag{31}$$

As can bee seen the unknowns $\langle U\rangle_i$, k_i, and $\langle T\rangle_i$ appear in the discretized equations simultaneously. Therefore, these equations should be combined to arrive at a linear system of equations and subsequently solved using a linear algebra solver. Let us define the unknowns vector $\mathbf{X}$ such that

$$\mathbf{X}=\begin{bmatrix} x_0\\ x_1\\ \vdots\\ x_{N-1}\\ x_N\\ x_{N+1}\\ x_{N+2}\\ \vdots\\ x_{2N}\\ x_{2N+1}\\ x_{2N+2}\\ x_{2N+3}\\ \vdots\\ x_{3N+1}\\ x_{3N+2}\end{bmatrix}=\begin{bmatrix} \langle U\rangle_0\\ \langle U\rangle_1\\ \vdots\\ \langle U\rangle_{N-1}\\ \langle U\rangle_N\\ k_0\\ k_1\\ \vdots\\ k_{N-1}\\ k_N\\ \langle T\rangle_0\\ \langle T\rangle_1\\ \vdots\\ \langle T\rangle_{N-1}\\ \langle T\rangle_N\end{bmatrix}$$

As can be seen the first third of vector $\mathbf{X}$ contains the $\langle U\rangle_i$ solutions, the second third of vector $\mathbf{X}$ contains the k_i solutions, and the last third of vector $\mathbf{X}$ contains the $\langle T\rangle_i$ solutions. Note that k_i maps to x_{i+N+1} and $\langle T\rangle_i$ maps to x_{i+2N+2}. Now we need $3N+3$ linear equations to solve for $\mathbf{X}$, i.e.

$$\begin{cases}
\text{Equation 0: } a_{0,0}x_0+a_{0,1}x_1+...+a_{0,3N+1}x_{3N+1}+a_{0,3N+2}x_{3N+2}=b_0\\
\text{Equation 1: } a_{1,0}x_0+a_{1,1}x_1+...+a_{1,3N+1}x_{3N+1}+a_{1,3N+2}x_{3N+2}=b_1\\
...\\
\text{Equation } i\text{: } a_{i,0}x_0+a_{i,1}x_1+...+a_{i,3N+1}x_{3N+1}+a_{i,3N+2}x_{3N+2}=b_i\\
...\\
\text{Equation } 3N+1\text{: } a_{3N+1,0}x_0+a_{3N+1,1}x_1+...+a_{3N+1,3N+1}x_{3N+1}+a_{3N+1,3N+2}x_{3N+2}=b_{3N+1}\\
\text{Equation } 3N+2\text{: } a_{3N+2,0}x_0+a_{3N+2,1}x_1+...+a_{3N+2,3N+1}x_{3N+1}+a_{3N+2,3N+2}x_{3N+2}=b_{3N+2}
\end{cases}$$

Our next task is to identify $a_{i,j}$ and b_i. These can be inferred from the discretized equations. $a_{i,j}$ are mostly zero except for where there is a non-zero coefficient in the corresponding equations.

The first N equations (equation 0, equation 1, ... equation N) are the momentum equations. The boundary condition for $\langle U \rangle$ at the surface provides the zeroth equation, i.e.

$$\begin{cases} x_0 = 0 \\ a_{0,0} = 1 \\ b_0 = 0 \end{cases}$$

The next $i : 1 \rightarrow N-1$ equations correspond to the momentum equation in the interior of the domain, so the coefficients can be obtained as follows

$$\begin{cases} \left(-\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2}\right) x_{i-1} + \left(-\frac{2c_2}{(\Delta z)^2}\right) x_i + \left(\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2}\right) x_{i+1} \\ + \left(-\frac{c_4}{2\Delta z}\right) x_{i-1+N+1} + c_3 x_{i+N+1} + \left(\frac{c_4}{2\Delta z}\right) x_{i+1+N+1} = c_b \\ a_{i,i-1} = -\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2} \\ a_{i,i} = -\frac{2c_2}{(\Delta z)^2} \\ a_{i,i+1} = \frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2} \\ a_{i,i-1+N+1} = -\frac{c_4}{2\Delta z} \\ a_{i,i+N+1} = c_3 \\ a_{i,i+1+N+1} = \frac{c_4}{2\Delta z} \\ b_i = c_b \end{cases}$$

Note that where the turbulent kinetic energy term appears the index is shifted by $N+1$. The zero-gradient boundary condition for $\langle U \rangle$ at the top of the domain provides the N^{th} equation, i.e.

$$\begin{cases} x_{N-1} - x_N = 0 \\ a_{N,N-1} = 1 \\ a_{N,N} = -1 \\ b_N = 0 \end{cases}$$

The boundary condition for k at the surface provides the $N+1^{th}$ equation, i.e.

$$\begin{cases} x_{N+1} = 0 \\ a_{N+1,N+1} = 1 \\ b_{N+1} = 0 \end{cases}$$

The next $i : N+2 \rightarrow 2N$ equations correspond to the turbulent kinetic energy equation in the interior of the domain, so the coefficients can be obtained as follows

$$
\begin{cases}
\left(-\frac{d_2}{2\Delta z}+\frac{d_3}{(\Delta z)^2}\right)x_{i-1}+\left(d_1-\frac{2d_3}{(\Delta z)^2}\right)x_i+\left(\frac{d_2}{2\Delta z}+\frac{d_3}{(\Delta z)^2}\right)x_{i+1} \\
+\left(-\frac{d_4}{2\Delta z}\right)x_{i-1-(N+1)}+\left(\frac{d_4}{2\Delta z}\right)x_{i+1-(N+1)}+\left(-\frac{d_5}{2\Delta z}\right)x_{i-1+(N+1)}+\left(\frac{d_5}{2\Delta z}\right)x_{i+1+(N+1)}=d_b \\
a_{i,i-1-(N+1)}=-\frac{d_4}{2\Delta z} \\
a_{i,i+1-(N+1)}=\frac{d_4}{2\Delta z} \\
a_{i,i-1}=-\frac{d_2}{2\Delta z}+\frac{d_3}{(\Delta z)^2} \\
a_{i,i}=d_1-\frac{2d_3}{(\Delta z)^2} \\
a_{i,i+1}=\frac{d_2}{2\Delta z}+\frac{d_3}{(\Delta z)^2} \\
a_{i,i-1+(N+1)}=-\frac{d_5}{2\Delta z} \\
a_{i,i+1+(N+1)}=\frac{d_5}{2\Delta z} \\
b_i=d_b
\end{cases}
$$

Note that where the momentum term appears the index is shifted by $-(N+1)$ and where the temperature term appears the index is shifted by $(N+1)$. The boundary condition for k at the top of the domain provides the $2N+1^{th}$ equation. The vertical gradient of the turbulent kinetic energy must be zero, i.e.

$$
\begin{cases}
x_{2N}-x_{2N+1}=0 \\
a_{2N+1,2N}=1 \\
a_{2N+1,2N+1}=-1 \\
b_{2N+1}=0
\end{cases}
$$

The boundary condition for $\langle T\rangle$ at the surface provides the $2N+2^{th}$ equation, i.e.

$$
\begin{cases}
x_{2N+2}=T_s \\
a_{2N+2,2N+2}=1 \\
b_{2N+2}=T_s
\end{cases}
$$

The next $i:2N+3\rightarrow 3N+1$ equations correspond to the heat equation in the interior of the domain, so the coefficients can be obtained as follows

$$
\begin{cases}
\left(-\frac{e_1}{2\Delta z}+\frac{e_2}{(\Delta z)^2}\right)x_{i-1}+\left(-\frac{2e_2}{(\Delta z)^2}\right)x_i+\left(\frac{e_1}{2\Delta z}+\frac{e_2}{(\Delta z)^2}\right)x_{i+1} \\
+\left(-\frac{e_4}{2\Delta z}\right)x_{i-1-(N+1)}+e_3x_{i-(N+1)}+\left(\frac{e_4}{2\Delta z}\right)x_{i+1-(N+1)}=e_b \\
a_{i,i-1-(N+1)}=-\frac{e_4}{2\Delta z} \\
a_{i,i-(N+1)}=e_3 \\
a_{i,i+1-(N+1)}=\frac{e_4}{2\Delta z} \\
a_{i,i-1}=-\frac{e_1}{2\Delta z}+\frac{e_2}{(\Delta z)^2} \\
a_{i,i}=-\frac{2e_2}{(\Delta z)^2} \\
a_{i,i+1}=\frac{e_1}{2\Delta z}+\frac{e_2}{(\Delta z)^2} \\
b_i=e_b
\end{cases}
$$

The boundary condition for $\langle T\rangle$ at the top of the domain provides the $3N+2^{th}$ equation. The vertical gradient of mean temperature must be zero, i.e.

$$
\begin{cases}
x_{3N+1}-x_{3N+2}=0 \\
a_{3N+2,3N+1}=1 \\
a_{3N+2,3N+2}=-1 \\
b_{3N+2}=0
\end{cases}
$$

Finally, we have arrived at linear system of equations that can be solved to provide the unknowns. This system is given as follows

$$
\mathbf{AX}=\mathbf{B} \tag{32}
$$

$$
\begin{bmatrix}
a_{0,0} & a_{0,1} & \dots & a_{0,2N} & a_{0,2N+1} \\
a_{1,0} & a_{1,1} & \dots & a_{1,2N} & a_{1,2N+1} \\
\vdots & \vdots & \vdots & \vdots & \vdots \\
a_{3N+1,0} & a_{3N+1,1} & \dots & a_{3N+1,3N+1} & a_{3N+1,3N+2} \\
a_{3N+2,0} & a_{3N+2,1} & \dots & a_{3N+2,3N+1} & a_{3N+2,3N+2}
\end{bmatrix}
\begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{3N+1} \\ x_{3N+2} \end{bmatrix}
=
\begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{3N+1} \\ b_{3N+2} \end{bmatrix}
$$

Note that the coefficients identified, themselves, depend on the solution. As a result the solution to the system of equations above must be found iteratively. That is, an initial guess for the solution vector $\mathbf{X}$ must be assumed. Then the system of equations must be solved iteratively. After each iteration the solutions must be updated. The system of equations must be solved iteratively until the difference between successive solutions is less than a specified error. For this purpose, the maximum norm can be considered. Suppose that a relative error of $Err=0.01$ is specified for any solution. Also suppose the x_i and $x_i^{(new)}$ represent two successive solutions for a specific point. The iteration can be stopped if the following conditions are met

$$\begin{cases} L_{\infty,mom.} = \max\left(|\frac{x_0^{(new)} - x_0}{x_0}|, |\frac{x_1^{(new)} - x_1}{x_1}|, ..., |\frac{x_N^{(new)} - x_N}{x_N}| \right) < Err \\ L_{\infty,tke.} = \max\left(|\frac{x_{N+1}^{(new)} - x_{N+1}}{x_{N+1}}|, |\frac{x_{N+2}^{(new)} - x_{N+2}}{x_{N+2}}|, ..., |\frac{x_{2N+1}^{(new)} - x_{2N+1}}{x_{2N+1}}| \right) < Err \\ L_{\infty,hea.} = \max\left(|\frac{x_{2N+2}^{(new)} - x_{2N+2}}{x_{2N+2}}|, |\frac{x_{2N+3}^{(new)} - x_{2N+3}}{x_{2N+3}}|, ..., |\frac{x_{3N+2}^{(new)} - x_{3N+2}}{x_{3N+2}}| \right) < Err \end{cases}$$

When solving a system of equations iteratively, it is sometimes more stable to only partially update a solution after each iteration. This is known as under relaxation. Consider that ϕ^{n-1} is the solution space found in the previous iteration and ϕ^{new} is the newly found solution. With the under relaxation factor $0 < \alpha < 1$, the solution can be updated for the next iteration such that

$$\phi^n = \phi^{n-1} + \alpha(\phi^{new} - \phi^{n-1}). \tag{33}$$

Particularly, whenever solving non-linear system of equations, this method improves stability of obtaining a numerical solution.

The simulation is desired for 4 combinations of τ and γ shown in table below. Case 1 represents a low pressure gradient and heat sink. Case 2 represents a high pressure gradient and heat sink. Case 3 represents a low pressure gradient and heat gain. Case 4 represents a high pressure gradient and heat gain.

Table 1: Simulation cases with varying amount of horizontal pressure gradient and heat sink/source.

Case	τ [m s^{-2}]	γ [K s^{-1}]
1	−0.005	0.005
2	−0.015	0.005
3	−0.005	−0.005
4	−0.015	−0.005

2 Python Script

Complete the following code to calculate the steady-state solutions for momentum, turbulent kinetic energy, and temperature.

```
import random
import sys
import os
import numpy
import matplotlib.pyplot as plt
```

```
#Define gravitational acceleration [m s^-2]
g=9.81

#Define under-relaxation factor
alpha=0.1

#Define horizontal pressure gradient divided by density [m s^-2]
tau=-0.005

#Average temperature in the entire domain [K]
T0=300

#Surface temperature [K]
Ts=290

#Define heat sink or source [K s^-1]
gamma=0.005

#Define von Karman constant
kappa=0.41

#Define maximum mixing length [m]
l0=10

#Define turbulence model constants, Martilli et al. (2002)
Ck=0.4
Ce=0.71

#Define maximum iteration number
MaxIter=100

#Define relative error
Err=0.01

#Define z axis from 0 to Z with dz increments
Z=100       #[m]
N=100
dz=Z/N      #[m]
z=numpy.linspace(0,Z,N+1)

#Define and initialize a mean velocity vector [m s^-1]
Uinitial=1
Umean=numpy.zeros((N+1,1))
Umean[:]=Uinitial

#Define and initialize turbulent viscosity [m^2 s^-1]
```

```
kinitial=0.1
k=numpy.zeros((N+1,1))
k[:]=kinitial

#Define and initialize temperature [K]
Tinitial=300
Tmean=numpy.zeros((N+1,1))
Tmean[:]=Tinitial

#Define unknown vector X, coefficient matrix A, and vector B, in AX=B
x=numpy.zeros((3*N+3,1))
xnew=numpy.zeros((3*N+3,1))
b=numpy.zeros((3*N+3,1))
a=numpy.zeros((3*N+3,3*N+3))

#Initialize solution vector X
#This is a short syntax for for loop
x[0:N+1]=Umean[0:N+1]
x[N+1:2*N+2]=k[0:N+1]
x[2*N+2:3*N+3]=Tmean[0:N+1]

for iter in range(1, MaxIter):
    #Momentum equations
    #i=0
    a[0][0]=1
    b[0]=0
    #i=1 to N-1
    for i in range(1, N):
        #Calculate derivatives by finite differences for the current i index
        #Remember to shift indices by N+1 if needed
        lm=kappa*z[i]/(1+(kappa*z[i])/l0)
        k0=x[i+N+1]
        k1=(x[i+1+N+1]-x[i-1+N+1])/(2*dz)
        k2=(x[i+1+N+1]-2*x[i+N+1]+x[i-1+N+1])/(dz**2)
        Umean0=x[i]
        Umean1=(x[i+1]-x[i-1])/(2*dz)
        Umean2=(x[i+1]-2*x[i]+x[i-1])/(dz**2)
        # Set constants necessary to build the coefficient matrix
        c1=0.5*Ck*lm*k0**(-1/2)*k1
        c2=Ck*lm*k0**(1/2)
        c3=-0.25*Ck*lm*k0**(-3/2)*k1*Umean1+0.5*Ck*lm*k0**(-1/2)*Umean2
        c4=0.5*Ck*lm*k0**(-1/2)*Umean1
        cb=-(-0.25*Ck*lm*k0**(-1/2)*k1*Umean1-0.5*Ck*lm*k0**(-1/2)*Umean2*k0-tau)
        # Set the coefficient matrix and the B vector
        a[i][i-1]=-c1/(2*dz)+c2/(dz**2)
        a[i][i]=-2*c2/(dz**2)
```

```
    a[i][i+1]=c1/(2*dz)+c2/(dz**2)
    a[i][i-1+N+1]=-c4/(2*dz)
    a[i][i+N+1]=c3
    a[i][i+1+N+1]=c4/(2*dz)
    b[i]=cb
#i=N
a[N][N-1]=1
a[N][N]=-1
b[N]=0

#Kinetic energy equations
#i=N+1
a[N+1][N+1]=1
b[N+1]=0
#i=N+2 to 2N
for i in range(N+2, 2*N+1):
    #Calculate derivatives by finite differences for the current i index
    #Shift indices by -(N+1) or (N+1) if needed
    lm=kappa*z[i-(N+1)]/(1+(kappa*z[i-(N+1)])/l0)
    k0=x[i]
    k1=(x[i+1]-x[i-1])/(2*dz)
    k2=(x[i+1]-2*x[i]+x[i-1])/(dz ** 2)
    Umean0=x[i-(N+1)]
    Umean1=(x[i+1-(N+1)]-x[i-1-(N+1)])/(2*dz)
    Umean2=(x[i+1-(N+1)]-2*x[i-(N+1)]+x[i-1-(N+1)])/(dz**2)
    Tmean0=x[i+(N+1)]
    Tmean1=(x[i+1+(N+1)]-x[i-1+(N+1)])/(2*dz)
    Tmean2=(x[i+1+(N+1)]-2*x[i+(N+1)]+x[i-1+(N+1)])/(dz**2)
    #Set constants necessary to build the coefficient matrix
    d1=...
    d2=...
    d3=...
    d4=...
    d5=.
    db=...
    # Set the coefficient matrix and the B vector
    a[i][i-1-(N+1)]=...
    a[i][i+1-(N+1)]=...
    a[i][i-1]=...
    a[i][i]=...
    a[i][i+1]=...
    a[i][i-1+(N+1)]=...
    a[i][i+1+(N+1)]=...
    b[i]=db
# i=2N+1
a[2*N+1][2*N]=1
```

```
a[2*N+1][2*N+1]=-1
b[2*N+1]=0

#heat equations
#i=2N+2
a[2*N+2][2*N+2]=...
b[2*N+2]=...
#i=2N+3 to 3N+1
for i in range(2*N+3, 3*N+2):
    #Calculate derivatives by finite differences for the current i index
    #Remember to shift indices by -(N+1) or -(2N+2) if needed
    lm=...
    k0=...
    k1=...
    k2=...
    Tmean0=...
    Tmean1=...
    Tmean2=...
    # Set constants necessary to build the coefficient matrix
    e1=0.5*Ck*lm*k0**(-1/2)*k1
    e2=Ck*lm*k0**(1/2)
    e3=-0.25*Ck*lm*k0**(-3/2)*k1*Tmean1+0.5*Ck*lm*k0**(-1/2)*Tmean2
    e4=0.5*Ck*lm*k0**(-1/2)*Tmean1
    eb=-(-0.25*Ck*lm*k0**(-1/2)*k1*Tmean1-0.5*Ck*lm*k0**(-1/2)*Tmean2*k0-gamma)
    # Set the coefficient matrix and the B vector
    a[i][i-1-(N+1)]=-e4/(2*dz)
    a[i][i-(N+1)]=e3
    a[i][i+1-(N+1)]=e4/(2*dz)
    a[i][i-1]=-e1/(2*dz)+e2/(dz**2)
    a[i][i]=-2*e2/(dz**2)
    a[i][i+1]=e1/(2*dz)+e2/(dz**2)
    b[i]=eb
#i=3N+2
a[3*N+2][3*N+1]=1
a[3*N+2][3*N+2]=-1
b[3*N+2]=0

xnew = numpy.linalg.solve(a, b)

# Calculate maximum norm errors for all solutions
ErrUmean=numpy.max(numpy.abs(numpy.divide(xnew[1:N+1]-x[1:N+1],x[1:N+1])))
Errk=...
ErrT=...

print('Iteration=',iter,'ErrUmean=',ErrUmean,'Errk=',Errk,'ErrT=',ErrT)
```

```
    if ErrUmean < Err and Errk < Err and ErrT < Err:
        print('Solutions converged at iteration: ', iter)
        # Exit the loop
        break

    # Update solution
    x[:]=x[:]+alpha*(xnew[:]-x[:])

#Assign the X vector to the original Umean, k, and T vectors
Umean[0:N+1]=x[0:N+1]
k[0:N+1]=x[N+1:2*N+2]
Tmean[0:N+1]=x[2*N+2:3*N+3]

#Plot the mean velocity versus z
plt.plot(Umean, z)
plt.xlabel('<U> [m s ^ -1]')
plt.ylabel('z[m]')
plt.title('Mean Velocity as Function of z')
plt.show()
# Plot the turbulent viscosity versus z
plt.plot(k, z)
plt.xlabel('k [m^2 s^-2]')
plt.ylabel('z [m]')
plt.title('Turbulent Kinetic Energy as Function of z')
plt.show()
# Plot the turbulent viscosity versus z
plt.plot(Tmean, z)
plt.xlabel('<T> [K]')
plt.ylabel('z [m]')
plt.title('Mean Temperature as Function of z')
plt.show()
```

Upon completing the code. You should get the following output from the console. For each iteration the convergence criteria is printed to the screen to monitor the solution behaviour. Note that with only a few tens of iterations using the Newton method, it is possible to converge to a solution with a relative error less than 1%.

```
...
Iteration= 43 ErrUmean= 0.0061785224409 Errk= 0.0111982631439 ErrT= 0.000830386432384
Iteration= 44 ErrUmean= 0.00555284890133 Errk= 0.0100670331501 ErrT= 0.000747160626933
Iteration= 45 ErrUmean= 0.00499124350421 Errk= 0.00905111247775 ErrT= 0.00067229321339
Solutions converged at iteration:  45
```

Try to answer the following questions.

- What is the effect of changing the τ on the momentum, turbulent kinetic energy, and tem-

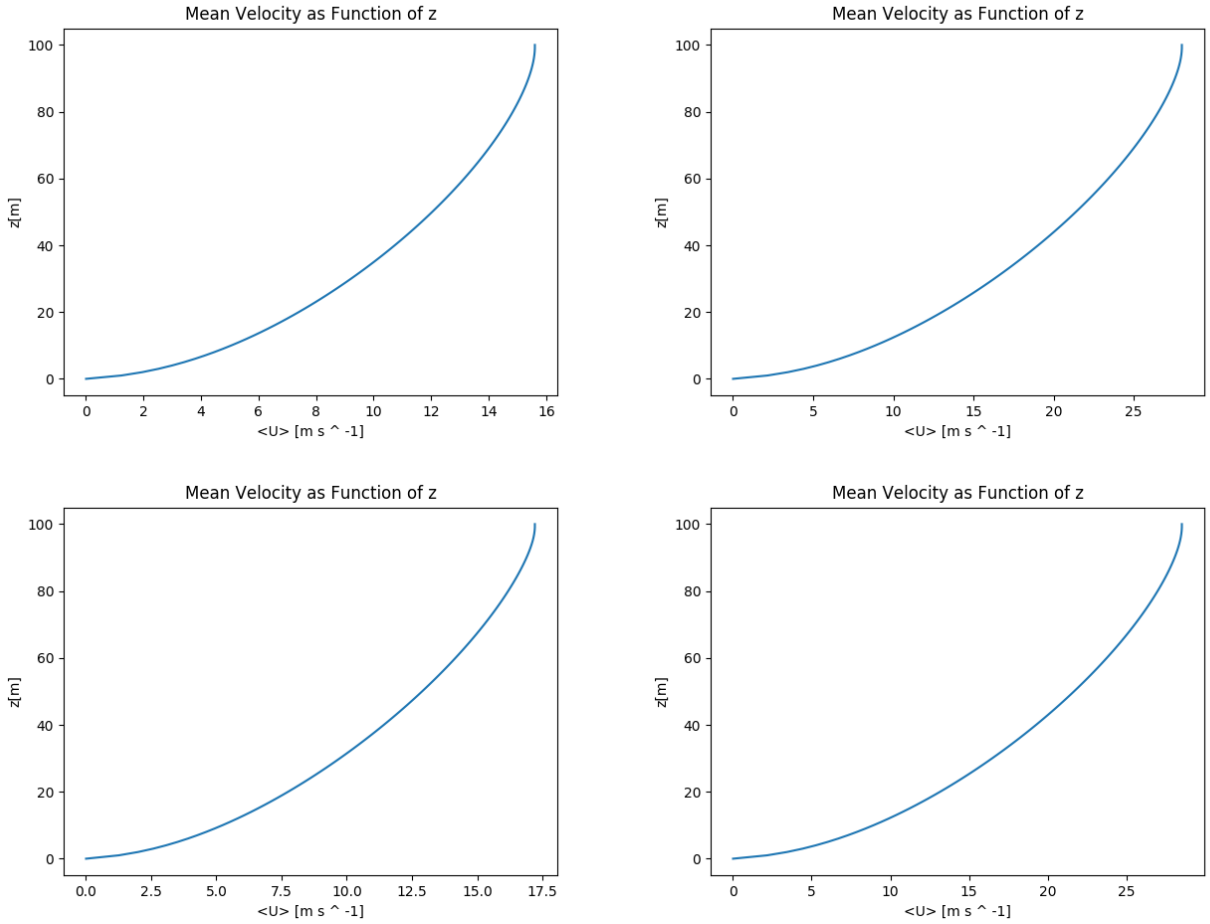


Figure 2: Momentum solution for case 1 (top left), case 2 (top right), case 3 (bottom left), and case 4 (bottom right)

perature solutions? Which solutions are affected the most.

- What is the effect of changing the γ on the momentum, turbulent kinetic energy, and temperature solutions? Which solutions are affected the most.
- Set $\gamma = 0$, i.e. eliminate heat sink or source in the heat equation. Can you explain the temperature profile that results?
- Increase the under-relaxation factor α to 0.9 when solving for the momentum and turbulent kinetic energy equations? Can you obtain a converged solution? Reason why.

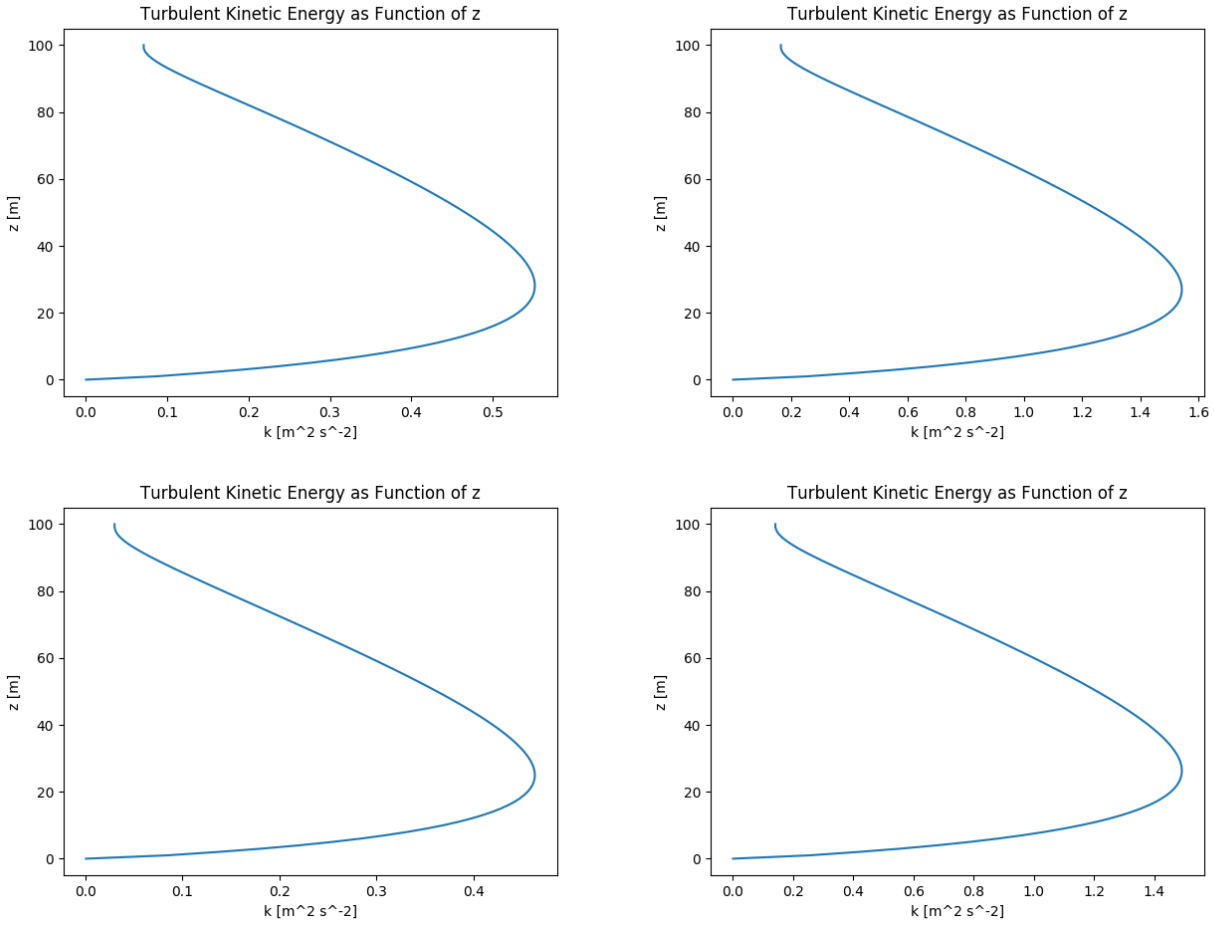


Figure 3: Turbulent kinetic energy solution for case 1 (top left), case 2 (top right), case 3 (bottom left), and case 4 (bottom right)

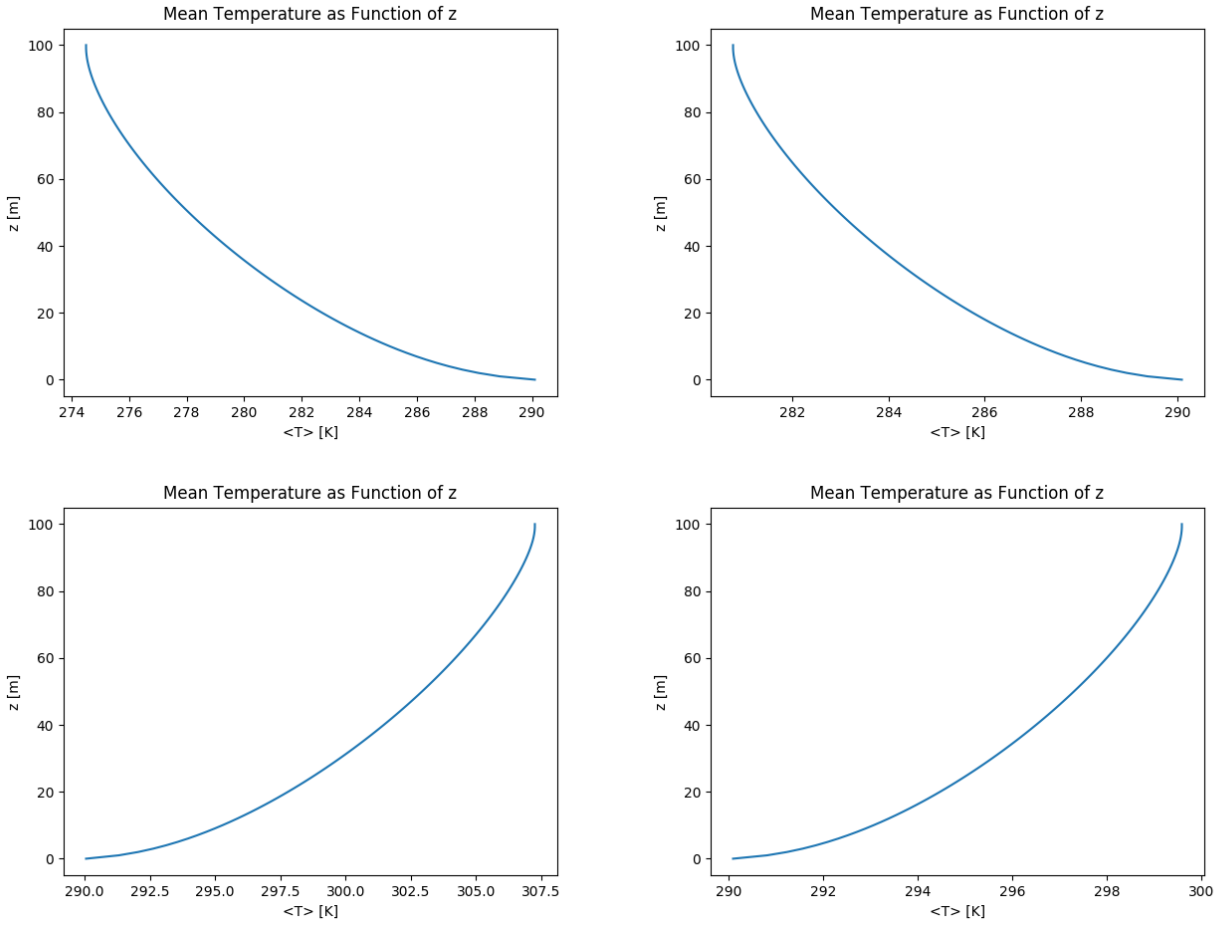


Figure 4: Temperature solution for case 1 (top left), case 2 (top right), case 3 (bottom left), and case 4 (bottom right)

ENGG*6790: Theory and Applications of Turbulence

1D Momentum, Turbulent Kinetic Energy, and Passive Scalar Equations over Flat Surface with Steady and Transient Formulations

Amir A. Aliabadi

March 14, 2019

1 Introduction

In the lectures the transport equation for a passive scalar was obtained using the gradient diffusion hypothesis. This hypothesis gives the transport equation and the effective diffusivity as

$$\underbrace{\frac{\overline{D}\langle\phi\rangle}{\overline{D}t}}_{\text{Material Derivative of Mean}} = \underbrace{\nabla.(\Gamma_{eff}\nabla\langle\phi\rangle)}_{\text{Diffusion of Mean}}. \tag{1}$$

$$\underbrace{\Gamma_{eff}(\mathbf{x},t)}_{\text{Effective Diffusivity}} = \underbrace{\Gamma}_{\text{Molecular Diffusivity}} + \underbrace{\Gamma_T(\mathbf{x},t)}_{\text{Turbulent Diffusivity}} \approx \Gamma_T(\mathbf{x},t), \tag{2}$$

where the molecular diffusivity has been ignored in a highly turbulent field in comparison to the turbulent diffusivity. Assuming that the turbulent Schmidt number is approximately equal to one the effective diffusivity for the passive scalar equation can be given as

$$\Gamma_T = \frac{\nu_T}{Sc_T} \approx \frac{\nu_T}{1} = \nu_T. \tag{3}$$

This model can be employed to develop a one-dimensional transport model for the passive scalar transport under transient conditions. Suppose we use the Cartesian coordinate system with coordinate axes of x, y, and z, and velocities corresponding to these axes being $U = \langle U\rangle + u$, $V = \langle V\rangle + v$, $W = \langle W\rangle + w$, respectively, as shown in the schematic. We can assume that mean flow is only in the x direction parallel to the surface and that the direction z is normal to the surface so that $\langle V\rangle = \langle W\rangle = 0$. The transport equation for the passive scalar can be given by

$$\underbrace{\frac{\partial\langle\phi\rangle}{\partial t}}_{\text{Storage}} = \underbrace{\frac{\partial}{\partial z}\left(\nu_T \frac{\partial\langle\phi\rangle}{\partial z}\right)}_{\text{Diffusion of Mean}}, \tag{4}$$

where the turbulent viscosity can be obtained solving the transport equations for momentum and the turbulent kinetic energy. One convenient approach is to solve the momentum and turbulent kinetic energy equations under steady-state condition first, Subsequently, once the turbulent viscosity or alternatively the kinetic energy is obtained, one can solve the transport equation for the passive scalar under transient conditions. This is perfectly reasonable because the passive scalar equation, whether solved under transient or steady state conditions, does not influence the momentum and turbulent kinetic energy equations.

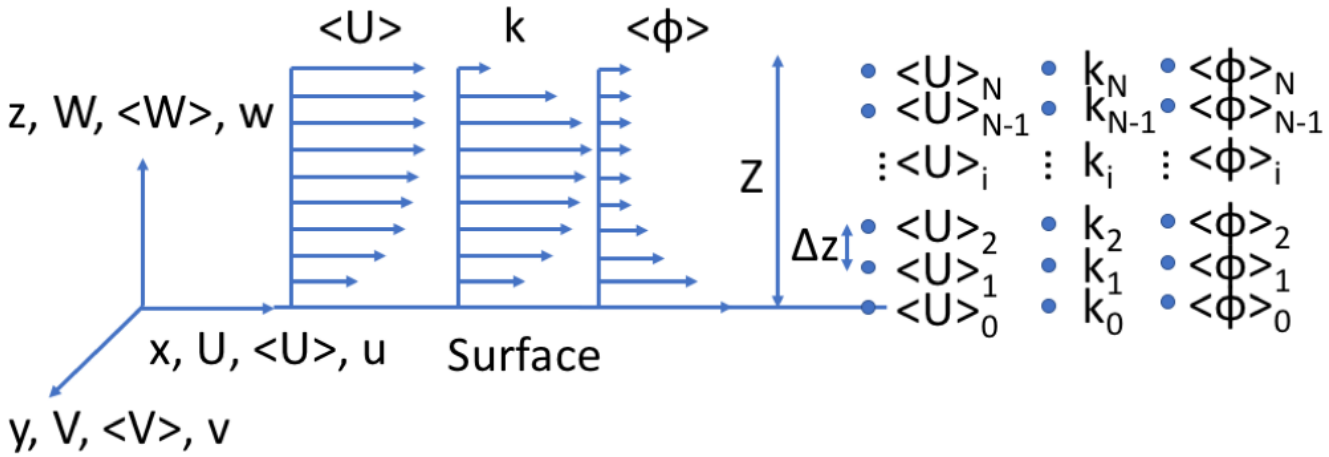


Figure 1: Schematic of 1D flow and passive scalar transport over flat surface using Cartesian coordinate system; finite difference representation of the 1D flow and passive scalar transport.

1.1 Steady-state Solution of the Momentum and Turbulent Kinetic Energy Equations

We first direct our attention to solve the momentum and turbulent kinetic energy equations under steady-state conditions. In the lectures the turbulent kinetic energy model was introduced as one transport equation to predict the turbulent kinetic energy. This turbulent kinetic energy was then used to formulate turbulent viscosity so that the momentum equation can be solved. The turbulent kinetic energy equation is given as

$$\underbrace{\frac{\overline{D}k}{\overline{D}t}}_{\text{Material Derivative}} \equiv \underbrace{\frac{\partial k}{\partial t}}_{\text{Storage}} + \underbrace{\langle U\rangle.\nabla k}_{\text{Advection}} = \underbrace{\nabla.\left(\frac{\nu_T}{\sigma_k}\nabla k\right)}_{\text{Energy Flux Divergence}} + \underbrace{\mathcal{P}}_{\text{Production}} - \underbrace{\epsilon}_{\text{Dissipation}}, \tag{5}$$

$$\begin{cases} \nu_T = ck^{1/2}\ell_m, \\ \epsilon = C_D \frac{k^{3/2}}{\ell_m}, \\ \ell_m(\mathbf{x}, t) \text{ known.} \end{cases}$$

Assume that the modified pressure has a constant gradient in the x direction, the 1D momentum equation then simplifies to

$$0 = \underbrace{\frac{\partial}{\partial z}\left(\nu_T \frac{\partial \langle U \rangle}{\partial z}\right)}_{\text{Surface Forces and Reynolds Stress}} - \underbrace{\tau}_{\text{Modified Pressure Forces}} \tag{6}$$

The one-dimensional turbulent kinetic energy equation can be developed as follows

$$0 = \underbrace{\frac{\partial}{\partial z}\left(\frac{\nu_T}{\sigma_k}\frac{\partial k}{\partial z}\right)}_{\text{Energy Flux Divergence}} + \underbrace{\nu_T \left(\frac{\partial \langle U \rangle}{\partial z}\right)^2}_{\text{Shear Production}} - \underbrace{\epsilon}_{\text{Dissipation}} \tag{7}$$

where the energy flux divergence was discussed in the lectures. This term ensures that the resulting model transport equation for k yields smooth solutions, and that a boundary condition can be imposed on k everywhere in the boundary of the domain. Otherwise the model may diverge if other transport mechanisms for k are much smaller than this term. The shear production term, is an example of a production term $\mathcal{P}$, that contributes to the generation of the turbulent kinetic energy. Here, when there is non-zero mean velocity gradient, turbulent kinetic energy is generated. The dissipation term is responsible for consuming turbulent kinetic energy down the energy cascade.

To close the turbulence model we can assume that the turbulent Prandtl number is unity, i.e. $\sigma_k = 1$. We can model turbulent viscosity, dissipation rate, and the appropriate mixing length as follows.

$$\begin{cases} \nu_T = C_k \ell_m k^{1/2}, \\ \epsilon = C_\epsilon \ell_m^{-1} k^{3/2}, \\ \ell_m = \kappa z / (1 + \frac{\kappa z}{\ell_0}). \end{cases}$$

where $\kappa = 0.41$ is the von Kármán constant, and ℓ_0 is the maximum mixing length. This formulation for mixing length has the nice property that it is bounded between zero and ℓ_0, which is physically sound since mixing length increases linearly in the log-law sublayer near a wall but cannot increase indefinitely in the interior of the domain. This formulation results in

$$\begin{cases} z \to 0 & \ell_m \to \kappa z \\ z \to \infty & \ell_m \to \ell_0 \end{cases}$$

So we have two equations: momentum and turbulent kinetic energy. We can eliminate ν_T and ϵ from the momentum and turbulent kinetic energy equations by direct substitutions and simplifications using the chain rule. So the two equations can be re-expressed as

$$\begin{aligned} 0 &= \frac{\partial}{\partial z}\left(C_k \ell_m k^{1/2} \frac{\partial \langle U \rangle}{\partial z}\right) - \tau \\ &= 0.5 C_k \ell_m k^{-1/2} \frac{\partial k}{\partial z}\frac{\partial \langle U \rangle}{\partial z} + C_k \ell_m k^{1/2} \frac{\partial^2 \langle U \rangle}{\partial z^2} - \tau \end{aligned} \tag{8}$$

$$\begin{aligned} 0 &= \frac{\partial}{\partial z}\left(C_k \ell_m k^{1/2} \frac{\partial k}{\partial z}\right) + C_k \ell_m k^{1/2}\left(\frac{\partial \langle U \rangle}{\partial z}\right)^2 - C_\epsilon \ell_m^{-1} k^{3/2} \\ &= 0.5 C_k \ell_m k^{-1/2}\left(\frac{\partial k}{\partial z}\right)^2 + C_k \ell_m k^{1/2} \frac{\partial^2 k}{\partial z^2} + C_k \ell_m k^{1/2}\left(\frac{\partial \langle U \rangle}{\partial z}\right)^2 - C_\epsilon \ell_m^{-1} k^{3/2}. \end{aligned} \tag{9}$$

As can be seen the two equations are extremely non-linear. They involve non-integer powers of the unknowns and their derivatives. They also involve the multiplication of the unknowns and derivatives. These equations can be linearized and solved using a finite difference scheme. Figure below shows the finite difference representation of the solution spaces for momentum and turbulent kinetic energy.

For notational convenience we can represent derivatives by superscripts and subsequently re-express the momentum and turbulent kinetic energy equations using this notation.

$$\begin{cases} \langle U \rangle^{(0)} = \langle U \rangle, \langle U \rangle^{(1)} = \frac{\partial \langle U \rangle}{\partial z}, \langle U \rangle^{(2)} = \frac{\partial^2 \langle U \rangle}{\partial z^2} \\ k^{(0)} = k, k^{(1)} = \frac{\partial k}{\partial z}, k^{(2)} = \frac{\partial^2 k}{\partial z^2} \end{cases}$$

$$0 = \underbrace{0.5 C_k \ell_m (k^{(0)})^{-1/2} k^{(1)} \langle U \rangle^{(1)}}_{f_{1,mom.}} + \underbrace{C_k \ell_m (k^{(0)})^{1/2} \langle U \rangle^{(2)}}_{f_{2,mom.}} - \tau \tag{10}$$

$$0 = \underbrace{0.5 C_k \ell_m (k^{(0)})^{-1/2} (k^{(1)})^2}_{f_{1,tke.}} + \underbrace{C_k \ell_m (k^{(0)})^{1/2} k^{(2)}}_{f_{2,tke.}} + \underbrace{C_k \ell_m (k^{(0)})^{1/2} (\langle U \rangle^{(1)})^2}_{f_{3,tke.}} \underbrace{- C_\epsilon \ell_m^{-1} (k^{(0)})^{3/2}}_{f_{4,tke.}}. \tag{11}$$

where each non-linear term in the equations have been renamed by a function f, Next, each f function can be replaced by its approximate using the Newton method expressing the function around an arbitrary point z_i. Beginning with the momentum equation the f function approximations are

$$
\begin{aligned}
f_{1,mom.} &\approx f_{1,mom.}(z_i) \\
&+ \frac{\partial f_{1,mom.}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{1,mom.}}{\partial k^{(1)}}|_{z_i} \left[k^{(1)}(z) - k^{(1)}(z_i)\right] \\
&+ \frac{\partial f_{1,mom.}}{\partial \langle U\rangle^{(1)}}|_{z_i} \left[\langle U\rangle^{(1)}(z) - \langle U\rangle^{(1)}(z_i)\right] \\
&= 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) \\
&- 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) [k^{(0)}(z) - k^{(0)}(z_i)] \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(1)}(z_i) [k^{(1)}(z) - k^{(1)}(z_i)] \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \left[\langle U\rangle^{(1)}(z) - \langle U\rangle^{(1)}(z_i)\right] \\
&= -0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) \\
&- 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z_i) k^{(0)}(z) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(1)}(z_i) k^{(1)}(z) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U\rangle^{(1)}(z)
\end{aligned} \tag{12}
$$

$$
\begin{aligned}
f_{2,mom.} &\approx f_{2,mom.}(z_i) \\
&+ \frac{\partial f_{2,mom.}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{2,mom.}}{\partial \langle U\rangle^{(2)}}|_{z_i} \left[\langle U\rangle^{(2)}(z) - \langle U\rangle^{(2)}(z_i)\right] \\
&= C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U\rangle^{(2)}(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(2)}(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ C_k \ell_m (k^{(0)})^{1/2}(z_i) \left[\langle U\rangle^{(2)}(z) - \langle U\rangle^{(2)}(z_i)\right] \\
&= -0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(2)}(z_i) k^{(0)}(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U\rangle^{(2)}(z_i) k^{(0)}(z) \\
&+ C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U\rangle^{(2)}(z)
\end{aligned} \tag{13}
$$

$$
\begin{aligned}
f_{1,tke.} &\approx f_{1,tke}(z_i) \\
&+ \frac{\partial f_{1,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{1,tke}}{\partial k^{(1)}}|_{z_i} \left[k^{(1)}(z) - k^{(1)}(z_i)\right] \\
&= 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(k^{(1)})^2(z_i) \\
&- 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i)(k^{(1)})^2(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \left[k^{(1)}(z) - k^{(1)}(z_i)\right] \\
&= -0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(k^{(1)})^2(z_i) \\
&- 0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i)(k^{(1)})^2(z_i) k^{(0)}(z) \\
&+ C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) k^{(1)}(z)
\end{aligned} \tag{14}
$$

$$
\begin{aligned}
f_{2,tke.} &\approx f_{2,tke}(z_i) \\
&+ \frac{\partial f_{2,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{2,tke}}{\partial k^{(2)}}|_{z_i} \left[k^{(2)}(z) - k^{(2)}(z_i)\right] \\
&= C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(2)}(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ C_k \ell_m (k^{(0)})^{1/2}(z_i) \left[k^{(2)}(z) - k^{(2)}(z_i)\right] \\
&= -0.5 C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(2)}(z_i) k^{(0)}(z) \\
&+ C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z)
\end{aligned} \tag{15}
$$

$$
\begin{aligned}
f_{3,tke.} &\approx f_{3,tke}(z_i) \\
&+ \frac{\partial f_{3,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ \frac{\partial f_{3,tke}}{\partial \langle U \rangle^{(1)}}|_{z_i} \left[\langle U \rangle^{(1)}(z) - \langle U \rangle^{(1)}(z_i)\right] \\
&= C_k \ell_m (k^{(0)})^{1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&+ 2 C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U \rangle^{(1)}(z_i) \left[\langle U \rangle^{(1)}(z) - \langle U \rangle^{(1)}(z_i)\right] \\
&= -1.5 C_k \ell_m (k^{(0)})^{1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) \\
&+ 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i)(\langle U \rangle^{(1)})^2(z_i) k^{(0)}(z) \\
&+ 2 C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U \rangle^{(1)}(z_i) \langle U \rangle^{(1)}(z)
\end{aligned} \tag{16}
$$

$$
\begin{aligned}
f_{4,tke.} &\approx f_{4,tke}(z_i) \\
&+ \frac{\partial f_{4,tke}}{\partial k^{(0)}}|_{z_i} \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&= -C_\epsilon \ell_m^{-1} (k^{(0)})^{3/2}(z_i) \\
&- 1.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{1/2}(z_i) \left[k^{(0)}(z) - k^{(0)}(z_i)\right] \\
&= 0.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{3/2}(z_i) \\
&- 1.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{1/2}(z_i) k^{(0)}(z)
\end{aligned} \tag{17}
$$

Now we can write the linearized forms of the momentum and turbulent kinetic energy equations.

$$
c_1 \langle U \rangle^{(1)} + c_2 \langle U \rangle^{(2)} + c_3 k^{(0)} + c_4 k^{(1)} = c_b \tag{18}
$$

$$
\begin{cases}
c_1 = 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \\
c_2 = C_k \ell_m (k^{(0)})^{1/2}(z_i) \\
c_3 = -0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) k^{(1)}(z_i) \langle U \rangle^{(1)}(z_i) + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U \rangle^{(2)}(z_i) \\
c_4 = 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U \rangle^{(1)}(z_i) \\
c_b = -\left[-0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \langle U \rangle^{(1)}(z_i) - 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) \langle U \rangle^{(2)}(z_i) k^{(0)}(z_i) - \tau\right]
\end{cases}
$$

$$
d_1 k^{(0)} + d_2 k^{(1)} + d_3 k^{(2)} + d_4 \langle U \rangle^{(1)} = d_b \tag{19}
$$

$$
\begin{cases}
d_1 = -0.25 C_k \ell_m (k^{(0)})^{-3/2}(z_i) (k^{(1)})^2(z_i) + 0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(2)}(z_i) \\
+0.5 C_k \ell_m (k^{(0)})^{-1/2}(z_i) (\langle U \rangle^{(1)})^2(z_i) - 1.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{1/2}(z_i) \\
d_2 = C_k \ell_m (k^{(0)})^{-1/2}(z_i) k^{(1)}(z_i) \\
d_3 = C_k \ell_m (k^{(0)})^{1/2}(z_i) \\
d_4 = 2 C_k \ell_m (k^{(0)})^{1/2}(z_i) \langle U \rangle^{(1)}(z_i) \\
d_1 = -\left[-0.25 C_k \ell_m (k^{(0)})^{-1/2}(z_i) (k^{(1)})^2(z_i) - 0.5 C_k \ell_m (k^{(0)})^{1/2}(z_i) k^{(2)}(z_i)\right] \\
-\left[-1.5 C_k \ell_m (k^{(0)})^{1/2}(z_i) (\langle U \rangle^{(1)})^2(z_i) + 0.5 C_\epsilon \ell_m^{-1} (k^{(0)})^{3/2}(z_i)\right]
\end{cases}
$$

Now consider that we want to represent the linearized momentum and turbulent kinetic energy equations using finite differences. Consider a vertical discretization Δz as shown in the figure. Using central differencing the spatial derivatives can be replaced by values at indices $i-1$, i, and $i+1$. We can replace the momentum and turbulent kinetic energy equations by their discretized versions as follows

$$
c_1 \frac{\langle U \rangle_{i+1} - \langle U \rangle_{i-1}}{2\Delta z} + c_2 \frac{\langle U \rangle_{i+1} - 2\langle U \rangle_i + \langle U \rangle_{i-1}}{(\Delta z)^2} + c_3 k_i + c_4 \frac{k_{i+1} - k_{i-1}}{2\Delta z} = c_b \tag{20}
$$

$$d_1 k_i + d_2 \frac{k_{i+1} - k_{i-1}}{2\Delta z} + d_3 \frac{k_{i+1} - 2k_i + k_{i-1}}{(\Delta z)^2} + d_4 \frac{\langle U \rangle_{i+1} - \langle U \rangle_{i-1}}{2\Delta z} = d_b \tag{21}$$

The above equations must be rearranged as follows.

$$\begin{aligned}\left(-\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2}\right) \langle U \rangle_{i-1} + \left(-\frac{2c_2}{(\Delta z)^2}\right) \langle U \rangle_i + \left(\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2}\right) \langle U \rangle_{i+1} \\ + \left(-\frac{c_4}{2\Delta z}\right) k_{i-1} + c_3 k_i + \left(\frac{c_4}{2\Delta z}\right) k_{i+1} = c_b\end{aligned} \tag{22}$$

$$\begin{aligned}\left(-\frac{d_2}{2\Delta z} + \frac{d_3}{(\Delta z)^2}\right) k_{i-1} + \left(d_1 - \frac{2d_3}{(\Delta z)^2}\right) k_i + \left(\frac{d_2}{2\Delta z} + \frac{d_3}{(\Delta z)^2}\right) k_{i+1} \\ + \left(-\frac{d_4}{2\Delta z}\right) \langle U \rangle_{i-1} + \left(\frac{d_4}{2\Delta z}\right) \langle U \rangle_{i+1} = d_b\end{aligned} \tag{23}$$

As can bee seen the unknowns $\langle U \rangle_i$ and k_i appear in both discretized momentum and turbulent kinetic energy equations. Therefore, these equations should be combined to arrive at a linear system of equations and subsequently solved using a linear algebra solver. Let us define the unknowns vector $\mathbf{X}$ such that

$$\mathbf{X} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \\ x_N \\ x_{N+1} \\ x_{N+2} \\ \vdots \\ x_{2N} \\ x_{2N+1} \end{bmatrix} = \begin{bmatrix} \langle U \rangle_0 \\ \langle U \rangle_1 \\ \vdots \\ \langle U \rangle_{N-1} \\ \langle U \rangle_N \\ k_0 \\ k_1 \\ \vdots \\ k_{N-1} \\ k_N \end{bmatrix}$$

As can be seen the first half of vector $\mathbf{X}$ contains the $\langle U \rangle_i$ solutions and the second half of vector $\mathbf{X}$ contains the k_i solutions. Note that k_i maps to x_{i+N+1}. Now we need $2N + 2$ linear equations to solve for $\mathbf{X}$, i.e.

$$\begin{cases} \text{Equation 0: } a_{0,0}x_0 + a_{0,1}x_1 + ... + a_{0,2N}x_{2N} + a_{0,2N+1}x_{2N+1} = b_0 \\ \text{Equation 1: } a_{1,0}x_0 + a_{1,1}x_1 + ... + a_{1,2N}x_{2N} + a_{1,2N+1}x_{2N+1} = b_1 \\ ... \\ \text{Equation } i\text{: } a_{i,0}x_0 + a_{i,1}x_1 + ... + a_{i,2N}x_{2N} + a_{i,2N+1}x_{2N+1} = b_i \\ ... \\ \text{Equation } 2N\text{: } a_{2N,0}x_0 + a_{2N,1}x_1 + ... + a_{2N,2N}x_{2N} + a_{2N,2N+1}x_{2N+1} = b_{2N} \\ \text{Equation } 2N+1\text{: } a_{2N+1,0}x_0 + a_{2N+1,1}x_1 + ... + a_{2N+1,2N}x_{2N} + a_{2N+1,2N+1}x_{2N+1} = b_{2N+1} \end{cases}$$

Our next task is to identify $a_{i,j}$ and b_i. These can be inferred from the discretized momentum and turbulent kinetic energy equations. $a_{i,j}$ are mostly zero except for where there is a non-zero coefficient in the corresponding equations. The first N equations (equation 0, equation 1, ... equation N) are the momentum equations. The boundary condition for $\langle U \rangle$ at the surface provides the zeroth equation, i.e.

$$\begin{cases} x_0 = 0 \\ a_{0,0} = 1 \\ b_0 = 0 \end{cases}$$

The next $i : 1 \rightarrow N-1$ equations correspond to the momentum equation in the interior of the domain, so the coefficients can be obtained as follows

$$\begin{cases} \left(-\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2}\right) x_{i-1} + \left(-\frac{2c_2}{(\Delta z)^2}\right) x_i + \left(\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2}\right) x_{i+1} \\ + \left(-\frac{c_4}{2\Delta z}\right) x_{i-1+N+1} + c_3 x_{i+N+1} + \left(\frac{c_4}{2\Delta z}\right) x_{i+1+N+1} = c_b \\ a_{i,i-1} = -\frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2} \\ a_{i,i} = -\frac{2c_2}{(\Delta z)^2} \\ a_{i,i+1} = \frac{c_1}{2\Delta z} + \frac{c_2}{(\Delta z)^2} \\ a_{i,i-1+N+1} = -\frac{c_4}{2\Delta z} \\ a_{i,i+N+1} = c_3 \\ a_{i,i+1+N+1} = \frac{c_4}{2\Delta z} \\ b_i = c_b \end{cases}$$

Note that where the turbulent kinetic energy term appears the index is shifted by $N+1$. The boundary condition for $\langle U \rangle$ at the top of the domain provides the N^{th} equation, i.e.

$$\begin{cases} x_{N-1} - x_N = 0 \\ a_{N,N-1} = 1 \\ a_{N,N} = -1 \\ b_N = 0 \end{cases}$$

The boundary condition for k at the surface provides the $N+1^{th}$ equation, i.e.

$$\begin{cases} x_{N+1} = 0 \\ a_{N+1,N+1} = 1 \\ b_{N+1} = 0 \end{cases}$$

The next $i : N+2 \rightarrow 2N$ equations correspond to the turbulent kinetic energy equation in the interior of the domain, so the coefficients can be obtained as follows

$$\begin{cases} \left(-\frac{d_2}{2\Delta z} + \frac{d_3}{(\Delta z)^2}\right) x_{i-1} + \left(d_1 - \frac{2d_3}{(\Delta z)^2}\right) x_i + \left(\frac{d_2}{2\Delta z} + \frac{d_3}{(\Delta z)^2}\right) x_{i+1} \\ + \left(-\frac{d_4}{2\Delta z}\right) x^{n+1}_{i-1-(N+1)} + \left(\frac{d_4}{2\Delta z}\right) x^{n+1}_{i+1-(N+1)} = d_b \\ a_{i,i-1-(N+1)} = -\frac{d_4}{2\Delta z} \\ a_{i,i+1-(N+1)} = \frac{d_4}{2\Delta z} \\ a_{i,i-1} = -\frac{d_2}{2\Delta z} + \frac{d_3}{(\Delta z)^2} \\ a_{i,i} = d_1 - \frac{2d_3}{(\Delta z)^2} \\ a_{i,i+1} = \frac{d_2}{2\Delta z} + \frac{d_3}{(\Delta z)^2} \\ b_i = d_b \end{cases}$$

Note that where the momentum term appears the index is shifted by $-(N+1)$. The boundary condition for k at the top of the domain provides the $2N+1^{th}$ equation. The vertical gradient of the turbulent kinetic energy must be zero, i.e.

$$\begin{cases} x_{2N} - x_{2N+1} = 0 \\ a_{2N+1,2N} = 1 \\ a_{2N+1,2N+1} = -1 \\ b_{2N+1} = 0 \end{cases}$$

Finally, we have arrived at linear system of equations that can be solved to provide the unknowns. This system is given as follows

$$\mathbf{AX} = \mathbf{B} \tag{24}$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,2N} & a_{0,2N+1} \\ a_{1,0} & a_{1,1} & \dots & a_{1,2N} & a_{1,2N+1} \\ \vdots & \vdots & \vdots & \vdots & \vdots \\ a_{2N,0} & a_{2N,1} & \dots & a_{2N,2N} & a_{2N,2N+1} \\ a_{2N+1,0} & a_{2N+1,1} & \dots & a_{2N+1,2N} & a_{2N+1,2N+1} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{2N} \\ x_{2N+1} \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_{2N} \\ b_{2N+1} \end{bmatrix}$$

Note that the coefficients identified, themselves, depend on the solution. As a result the solution to the system of equations above must be found iteratively. That is, an initial guess for the solution vector $\mathbf{X}$ must be assumed. Then the system of equations must be solved iteratively. After each iteration the solutions must be updated. The system of equations must be solved iteratively until the difference between successive solutions is less than a specified error. For this purpose, the maximum norm can be considered. Suppose that a relative error of $Err = 0.01$ is specified for either the momentum or turbulent kinetic energy solution. Also suppose the x_i and $x_i^{(new)}$ represent two successive solutions for a specific point. The iteration can be stopped if the following conditions are met

$$\begin{cases} L_{\infty,mom.} = \max\left(|\frac{x_0^{(new)} - x_0}{x_0}|, |\frac{x_1^{(new)} - x_1}{x_1}|, ..., |\frac{x_N^{(new)} - x_N}{x_N}| \right) < Err \\ L_{\infty,tke} = \max\left(|\frac{x_{N+1}^{(new)} - x_{N+1}}{x_{N+1}}|, |\frac{x_{N+2}^{(new)} - x_{N+2}}{x_{N+2}}|, ..., |\frac{x_{2N+1}^{(new)} - x_{2N+1}}{x_{2N+1}}| \right) < Err \end{cases}$$

When solving a system of equations iteratively, it is sometimes more stable to only partially update a solution after each iteration. This is known as under relaxation. Consider that ϕ^{n-1} is the solution space found in the previous iteration and ϕ^{new} is the newly found solution. With the under relaxation factor $0 < \alpha < 1$, the solution can be updated for the next iteration such that

$$\phi^n = \phi^{n-1} + \alpha(\phi^{new} - \phi^{n-1}). \tag{25}$$

Particularly, whenever solving non-linear system of equations, this method improves stability of obtaining a numerical solution.

1.2 Transient Solution of the Passive Scalar Transport Equation

Next we focus our attention to solve the transient passive scalar transport equation. For notational convenience we can represent derivatives by superscripts and subsequently re-express the passive scalar transport equation using this notation. Note that we keep the time derivative with its original notation since this derivative involves having solutions at different timesteps, which later have to be considered for the implicit Euler method for the transient model.

$$\begin{cases} \langle\phi\rangle^{(0)} = \langle\phi\rangle, \langle\phi\rangle^{(1)} = \frac{\partial\langle\phi\rangle}{\partial z}, \langle\phi\rangle^{(2)} = \frac{\partial^2\langle\phi\rangle}{\partial z^2} \\ k^{(0)} = k, k^{(1)} = \frac{\partial k}{\partial z} \end{cases}$$

$$\frac{\partial\langle\phi\rangle}{\partial t} = 0.5C_k\ell_m(k^{(0)})^{-1/2}k^{(1)}\langle\phi\rangle^{(1)} + C_k\ell_m(k^{(0)})^{1/2}\langle\phi\rangle^{(2)}. \tag{26}$$

Note that having values for $k^{(0)}$ and $k^{(1)}$ already, this equation is already in the linear form and one does not need to use the Newton method to linearize it. This equation can be expressed as

$$\frac{\partial\langle\phi\rangle}{\partial t} = c_1\langle\phi\rangle^{(1)} + c_2\langle\phi\rangle^{(2)} \tag{27}$$

$$\begin{cases} c_1 = 0.5C_k\ell_m(k^{(0)})^{-1/2}k^{(1)} \\ c_2 = C_k\ell_m(k^{(0)})^{1/2} \end{cases}$$

Now consider that we want to represent the passive scalar transport equation using finite differences. Consider a vertical discretization Δz as shown in the figure and temporal discretization Δt. Using central differencing the spatial derivatives can be replaced by values at indices $i-1$, i, and $i+1$. In addition, the time derivative can be replaced by values at time levels n and $n+1$. If we use the implicit Euler method, i.e. computing spatial derivatives at time level $n+1$, we can replace the equation by its discretized versions as follows

$$\frac{\langle\phi\rangle_i^{n+1} - \langle\phi\rangle_i^n}{\Delta t} = c_1\frac{\langle\phi\rangle_{i+1}^{n+1} - \langle\phi\rangle_{i-1}^{n+1}}{2\Delta z} + c_2\frac{\langle\phi\rangle_{i+1}^{n+1} - 2\langle\phi\rangle_i^{n+1} + \langle\phi\rangle_{i-1}^{n+1}}{(\Delta z)^2} \tag{28}$$

Note that in the implicit Euler method we assume that the values of $\langle\phi\rangle_i$ are known at time level n, and one must solve for values at time level $n+1$. As a result, the above equation must be rearranged as follows

$$\left(\frac{c_1\Delta t}{2\Delta z} - \frac{c_2\Delta t}{(\Delta z)^2}\right)\langle\phi\rangle_{i-1}^{n+1} + \left(1 + \frac{2c_2\Delta t}{(\Delta z)^2}\right)\langle\phi\rangle_i^{n+1} + \left(-\frac{c_1\Delta t}{2\Delta z} - \frac{c_2\Delta t}{(\Delta z)^2}\right)\langle\phi\rangle_{i+1}^{n+1} = \langle\phi\rangle_i^n. \tag{29}$$

Now let us define the unknowns vector $\mathbf{X}$ such that

$$\mathbf{X} = \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_{N-1} \\ x_N \end{bmatrix} = \begin{bmatrix} \langle\phi\rangle_0^{n+1} \\ \langle\phi\rangle_1^{n+1} \\ \vdots \\ \langle\phi\rangle_{N-1}^{n+1} \\ \langle\phi\rangle_N^{n+1} \end{bmatrix}$$

Now we need $N+1$ linear equations to solve for $\mathbf{X}$, i.e.

$$\begin{cases} \text{Equation 0: } a_{0,0}x_0 + a_{0,1}x_1 + ... + a_{0,N}x_N = b_0 \\ \text{Equation 1: } a_{1,0}x_0 + a_{1,1}x_1 + ... + a_{1,N}x_N = b_1 \\ ... \\ \text{Equation } i\text{: } a_{i,0}x_0 + a_{i,1}x_1 + ... + a_{i,N}x_N = b_i \\ ... \\ \text{Equation } N-1\text{: } a_{N-1,0}x_0 + a_{N-1,1}x_1 + ... + a_{N-1,N}x_N = b_{N-1} \\ \text{Equation } N\text{: } a_{N,0}x_0 + a_{N,1}x_1 + ... + a_{N,N}x_N = b_N \end{cases}$$

Our next task is to identify $a_{i,j}$ and b_i. These can be inferred from the discretized passive scalar equation. $a_{i,j}$ are mostly zero except for where there is a non-zero coefficient in the corresponding equations. The boundary condition for $\langle\phi\rangle$ at the surface provides the zeroth equation, assuming a fixed concentration, at the surface, i.e.

$$\begin{cases} x_0 = 0 \\ a_{0,0} = 1 \\ b_0 = \phi_s \end{cases}$$

The next $i : 1 \rightarrow N-1$ equations correspond to the interior of the domain, so the coefficients can be obtained as follows

$$\begin{cases} \left(\frac{c_1\Delta t}{2\Delta z} - \frac{c_2\Delta t}{(\Delta z)^2}\right) x_{i-1} + \left(1 + \frac{2c_2\Delta t}{(\Delta z)^2}\right) x_i + \left(-\frac{c_1\Delta t}{2\Delta z} - \frac{c_2\Delta t}{(\Delta z)^2}\right) x_{i+1} = \langle\phi\rangle_i^n \\ a_{i,i-1} = \left(\frac{c_1\Delta t}{2\Delta z} - \frac{c_2\Delta t}{(\Delta z)^2}\right) \\ a_{i,i} = \left(1 + \frac{2c_2\Delta t}{(\Delta z)^2}\right) \\ a_{i,i+1} = \left(-\frac{c_1\Delta t}{2\Delta z} - \frac{c_2\Delta t}{(\Delta z)^2}\right) \\ b_i = \langle\phi\rangle_i^n \end{cases}$$

The boundary condition for $\langle\phi\rangle$ at the top of the domain provides the N^{th} equation, assuming zero gradient, i.e.

$$\begin{cases} x_{N-1} - x_N = 0 \\ a_{N,N-1} = 1 \\ a_{N,N} = -1 \\ b_N = 0 \end{cases}$$

Finally, we have arrived at linear system of equations that can be solved to provide the unknowns. This system is given as follows

$$\mathbf{AX} = \mathbf{B} \tag{30}$$

$$\begin{bmatrix} a_{0,0} & a_{0,1} & \dots & a_{0,N} \\ a_{1,0} & a_{1,1} & \dots & a_{1,N} \\ \vdots & \vdots & \vdots & \vdots \\ a_{N,0} & a_{N,1} & \dots & a_{N,N} \end{bmatrix} \begin{bmatrix} x_0 \\ x_1 \\ \vdots \\ x_N \end{bmatrix} = \begin{bmatrix} b_0 \\ b_1 \\ \vdots \\ b_N \end{bmatrix}$$

Note that this system of equations must be solved at every time level. Once the solutions at every time level is obtained, it can then be used as initial condition and the system must be solved again for the next time level. At each time level, note that the coefficients identified, themselves, do not depend on the solution. As a result the solution to the system of equations above does not have to be found iteratively.

The transient simulation is desired for 4 combinations of τ and ℓ_0 shown in table below, while the value of $\phi_s = 1$ fixed. Case 1 represents a low pressure gradient and small mixing length. Case 2 represents a high pressure gradient and small mixing length. Case 3 represents a low pressure gradient and large mixing length. Case 4 represents a high pressure gradient and large mixing length.

Table 1: Simulation cases with varying amount of horizontal pressure gradient and mixing length

Case	τ [m s^{-2}]	ℓ_0 [m]
1	-0.005	10
2	-0.01	10
3	-0.005	20
4	-0.01	20

2 Python Script

Complete the following code to calculate the steady-state solutions for momentum and the turbulent kinetic energy.

```
import random
import sys
import os
import numpy
import matplotlib.pyplot as plt

#Define under-relaxation factor
alpha=0.1

#Define horizontal pressure gradient divided by density [m s^-2]
```

```
tau=-0.005

#Define von Karman constant
kappa=0.41

#Define maximum mixing length [m]
l0=10

#Define turbulence model constants, Martilli et al. (2002)
Ck=0.4
Ce=0.71

#Define maximum iteration number
MaxIter=100

#Define relative error
Err=0.01

#Define z axis from 0 to Z with dz increments
Z=100       #[m]
N=100
dz=Z/N      #[m]
z=numpy.linspace(0,Z,N+1)

#Define and initialize a mean velocity vector [m s^-1]
Uinitial=1
Umean=numpy.zeros((N+1,1))
Umean[:]=Uinitial

#Define and initialize turbulent viscosity [m^2 s^-1]
kinitial=0.1
k=numpy.zeros((N+1,1))
k[:]=kinitial

#Define unknown vector X, coefficient matrix A, and vector B, in AX=B
x=numpy.zeros((2*N+2,1))
xnew=numpy.zeros((2*N+2,1))
b=numpy.zeros((2*N+2,1))
a=numpy.zeros((2*N+2,2*N+2))

#Initialize solution vector X
#This is a short syntax for for loop
x[0:N+1]=Umean[0:N+1]
x[N+1:2*N+2]=k[0:N+1]

for iter in range(1, MaxIter):
```

```
#Momentum equations
#i=0
a[0][0]=1
b[0]=0
#i=1 to N-1
for i in range(1, N):
    #Calculate derivatives by finite differences for the current i index
    #Remember to shift indices by N+1 if needed
    lm=kappa*z[i]/(1+(kappa*z[i])/l0)
    k0=x[i+N+1]
    k1=(x[i+1+N+1]-x[i-1+N+1])/(2*dz)
    k2=(x[i+1+N+1]-2*x[i+N+1]+x[i-1+N+1])/(dz**2)
    Umean0=x[i]
    Umean1=(x[i+1]-x[i-1])/(2*dz)
    Umean2=(x[i+1]-2*x[i]+x[i-1])/(dz**2)
    # Set constants necessary to build the coefficient matrix
    c1=0.5*Ck*lm*k0**(-1/2)*k1
    c2=Ck*lm*k0**(1/2)
    c3=-0.25*Ck*lm*k0**(-3/2)*k1*Umean1+0.5*Ck*lm*k0**(-1/2)*Umean2
    c4=0.5*Ck*lm*k0**(-1/2)*Umean1
    cb=-(-0.25*Ck*lm*k0**(-1/2)*k1*Umean1-0.5*Ck*lm*k0**(-1/2)*Umean2*k0-tau)
    # Set the coefficient matrix and the B vector
    a[i][i-1]=...
    a[i][i]=...
    a[i][i+1]=...
    a[i][i-1+N+1]=...
    a[i][i+N+1]=...
    a[i][i+1+N+1]=...
    b[i]=...
#i=N
a[N][N-1]=...
a[N][N]=...
b[N]=...

#Kinetic energy equations
#i=N+1
a[N+1][N+1]=1
b[N+1]=0
#i=N+2 to 2N
for i in range(N+2, 2*N+1):
    #Calculate derivatives by finite differences for the current i index
    #Shift indices by -(N+1) if needed
    lm=kappa*z[i-(N+1)]/(1+(kappa*z[i-(N+1)])/l0)
    k0=x[i]
    k1=(x[i+1]-x[i-1])/(2*dz)
    k2=(x[i+1]-2*x[i]+x[i-1])/(dz ** 2)
```

```
        Umean0=x[i-(N+1)]
        Umean1=(x[i+1-(N+1)]-x[i-1-(N+1)])/(2*dz)
        Umean2=(x[i+1-(N+1)]-2*x[i-(N+1)]+x[i-1-(N+1)])/(dz**2)
        #Set constants necessary to build the coefficient matrix
        d1=-0.25*Ck*lm*k0**(-3/2)*k1**2+0.5*Ck*lm*k0**(-1/2)*k2\
             +0.5*Ck*lm*k0**(-1/2)*Umean1**2-1.5*Ce*lm**(-1)*k0**(1/2)
        d2=Ck*lm*k0**(-1/2)*k1
        d3=Ck*lm*k0**(1/2)
        d4=2*Ck*lm*k0**(1/2)*Umean1
        db=-(-0.25*Ck*lm*k0**(-1/2)*k1**2-0.5*Ck*lm*k0**(1/2)*k2)\
             -(-1.5*Ck*lm*k0**(1/2)*Umean1**2+0.5*Ce*lm**(-1)*k0**(3/2))
        # Set the coefficient matrix and the B vector
        a[i][i-1-(N+1)]=...
        a[i][i+1-(N+1)]=...
        a[i][i-1]=...
        a[i][i]=...
        a[i][i+1]=...
        b[i]=...
    # i=2N+1
    a[2*N+1][2*N]=...
    a[2*N+1][2*N+1]=...
    b[2*N+1]=...

    xnew = numpy.linalg.solve(a, b)

    # Calculate maximum norm errors for both momentum and turbulent kinetic energy
    ErrUmean=numpy.max(numpy.abs(numpy.divide(xnew[1:N+1]-x[1:N+1],x[1:N+1])))
    Errk=...

    print('Iteration=',iter,'ErrUmean=',ErrUmean,'Errk=',Errk)

    if ErrUmean < Err and Errk < Err:
        print('Solutions converged at iteration: ', iter)
        # Exit the loop
        break

    # Update solution
    x[:]=x[:]+alpha*(xnew[:]-x[:])

#Assign the X vector to the original Umean and k vectors
Umean[0:N+1]=...
k[0:N+1]=...

#Plot the mean velocity versus z
plt.plot(Umean, z)
plt.xlabel('<U> [m s^-1]')
```

```
plt.ylabel('z[m]')
plt.title('Mean Velocity as Function of z')
plt.show()
# Plot the turbulent viscosity versus z
plt.plot(k, z)
plt.xlabel('k [m^2 s^-2]')
plt.ylabel('z [m]')
plt.title('Turbulent Kinetic Energy as Function of z')
plt.show()
```

Upon completing the code. You should get the following output from the console. For each iteration the convergence criteria is printed to the screen to monitor the solution behaviour. Note that with only a few tens of iterations using the Newton method, it is possible to converge to a solution with a relative error less than 1%.

```
...
Iteration= 43 ErrUmean= 0.00536326893688 Errk= 0.0123014708332
Iteration= 44 ErrUmean= 0.00482027210921 Errk= 0.0110574058777
Iteration= 45 ErrUmean= 0.00433285589024 Errk= 0.00994041815673
Solutions converged at iteration:  45
```

Now complete the following code to find a transient solution for the passive scalar transport equation.

```
#Now simulate the transient passive scalar transport equation
#Define t vector from 0 to T with dt increments
T=1000         #[s]
Nt=100
dt=T/Nt     #[s]
t=numpy.linspace(0, T, Nt+1)

#Define and initialize a mean passive scalar concentration
phis=1
phimean=numpy.zeros((N+1,Nt+1))
phimean[0][0]=phis

#Define unknown vector X, coefficient matrix A, and vector B, in AX=B
x=numpy.zeros((N+1,1))
xnew=numpy.zeros((N+1,1))
b=numpy.zeros((N+1,1))
a=numpy.zeros((N+1,N+1))

#Initialize solution vector X
for i in range(0, N+1):
    x[i]=phimean[i][0]
```

```
#Iterate for time levels
for n in range(1, Nt+1):

    #passive scalar equations
    #i=0
    a[0][0]=1
    b[0]=phis
    #i=1 to N-1
    for i in range(1, N):
        #Calculate derivatives by finite differences for the current i index
        lm=kappa*z[i]/(1+(kappa*z[i])/l0)
        k0=k[i]
        k1=(k[i+1]-k[i-1])/(2*dz)
        # Set constants necessary to build the coefficient matrix
        c1=...
        c2=...
        # Set the coefficient matrix and the B vector
        a[i][i-1]=...
        a[i][i]=...
        a[i][i+1]=...
        b[i]=phimean[i][n-1]
    #i=N
    a[N][N-1]=...
    a[N][N]=...
    b[N]=...

    xnew=numpy.linalg.solve(a,b)

    #Update solution
    x[:]=xnew[:]

    #Before next time level,
    #assign the X vector to the original phimean matrix
    for i in range(0, N+1):
        phimean[i][n]=x[i]

#Plot the mean passive scalar versus z
plt.plot(phimean[:,1],z,label='t='+str(t[1])+' s')
plt.plot(phimean[:,25],z,label='t='+str(t[25])+' s')
plt.plot(phimean[:,50],z,label='t='+str(t[50])+' s')
plt.plot(phimean[:,75],z,label='t='+str(t[75])+' s')
plt.plot(phimean[:,100],z,label='t='+str(t[100])+' s')
plt.xlabel('<phi>')
plt.ylabel('z [m]')
plt.title('Mean Passive Scalar as Function of z')
plt.legend()
```

```
plt.show()
```

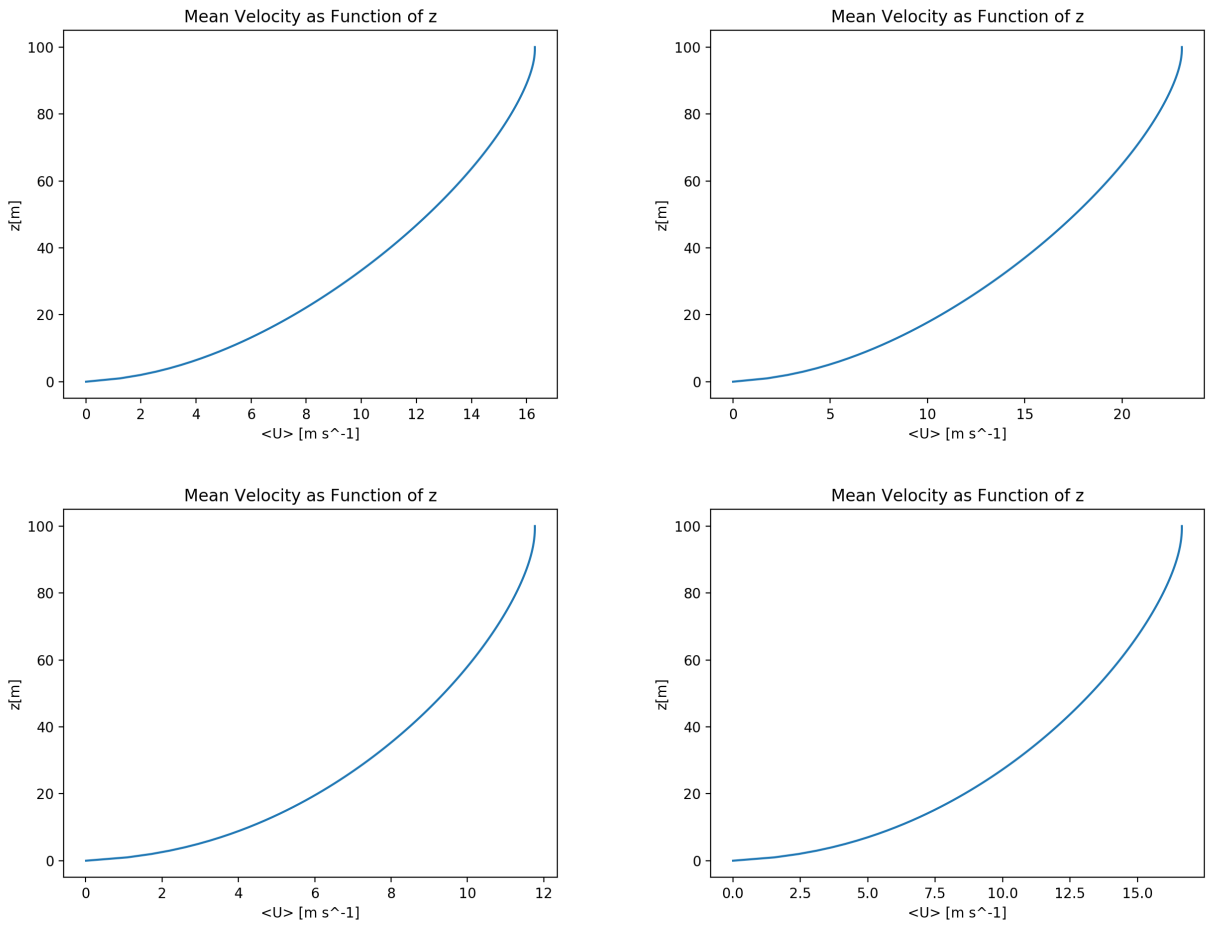


Figure 2: Momentum solution for case 1 (top left), case 2 (top right), case 3 (bottom left), and case 4 (bottom right)

Try to answer the following questions.

- Discuss the effects of horizontal pressure gradient and maximum mixing length on the solutions.
- Why did we not iterate at every time level when solving the passive scalar transport equation?
- What is the effect of changing the timestep Δt on the passive scalar solution?
- What is the effect of changing the spatial discretization Δz on the passive scalar solution?
- Increase the under-relaxation factor α to 0.9 when solving for the momentum and turbulent kinetic energy equations? Can you obtain a converged solution? Reason why.

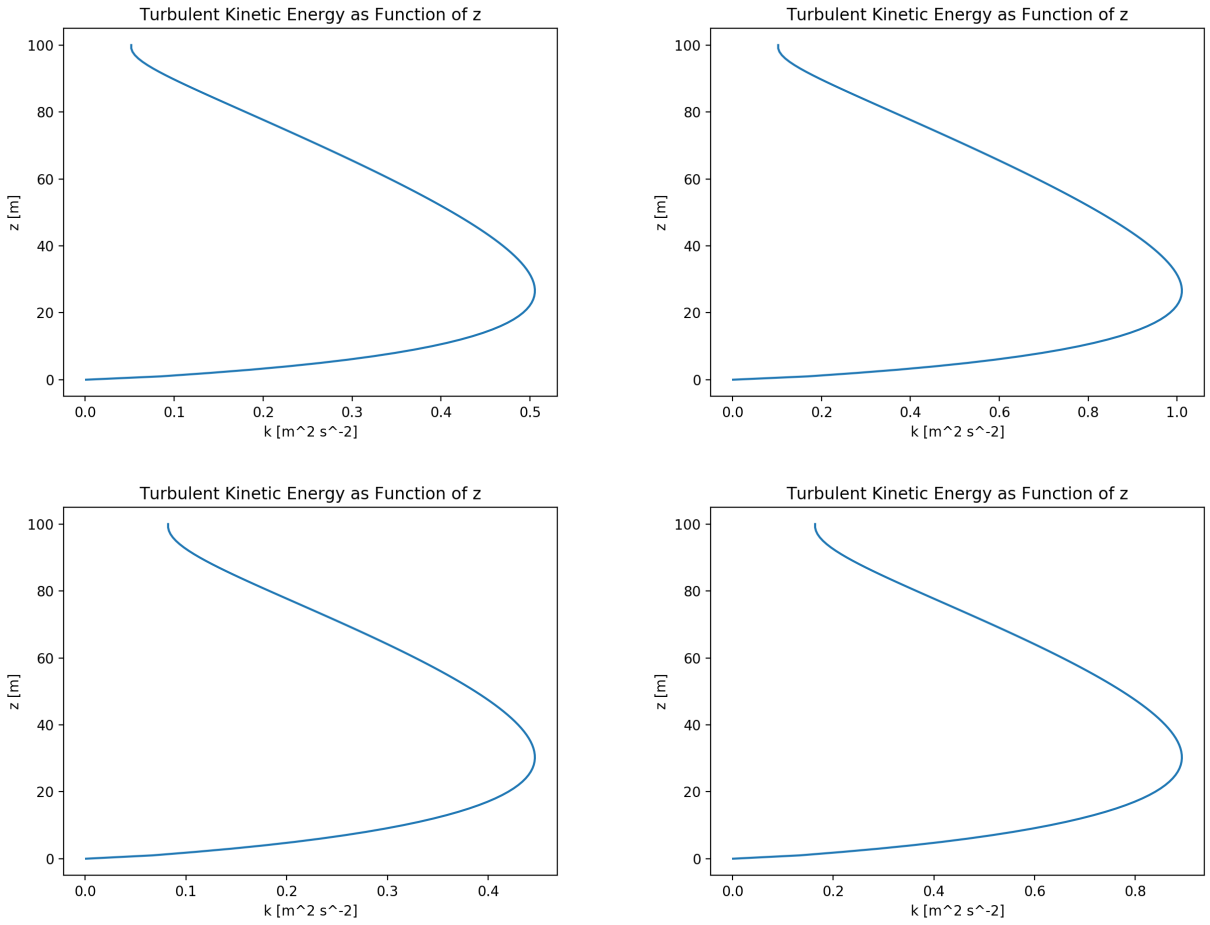


Figure 3: Turbulent kinetic energy solution for case 1 (top left), case 2 (top right), case 3 (bottom left), and case 4 (bottom right)

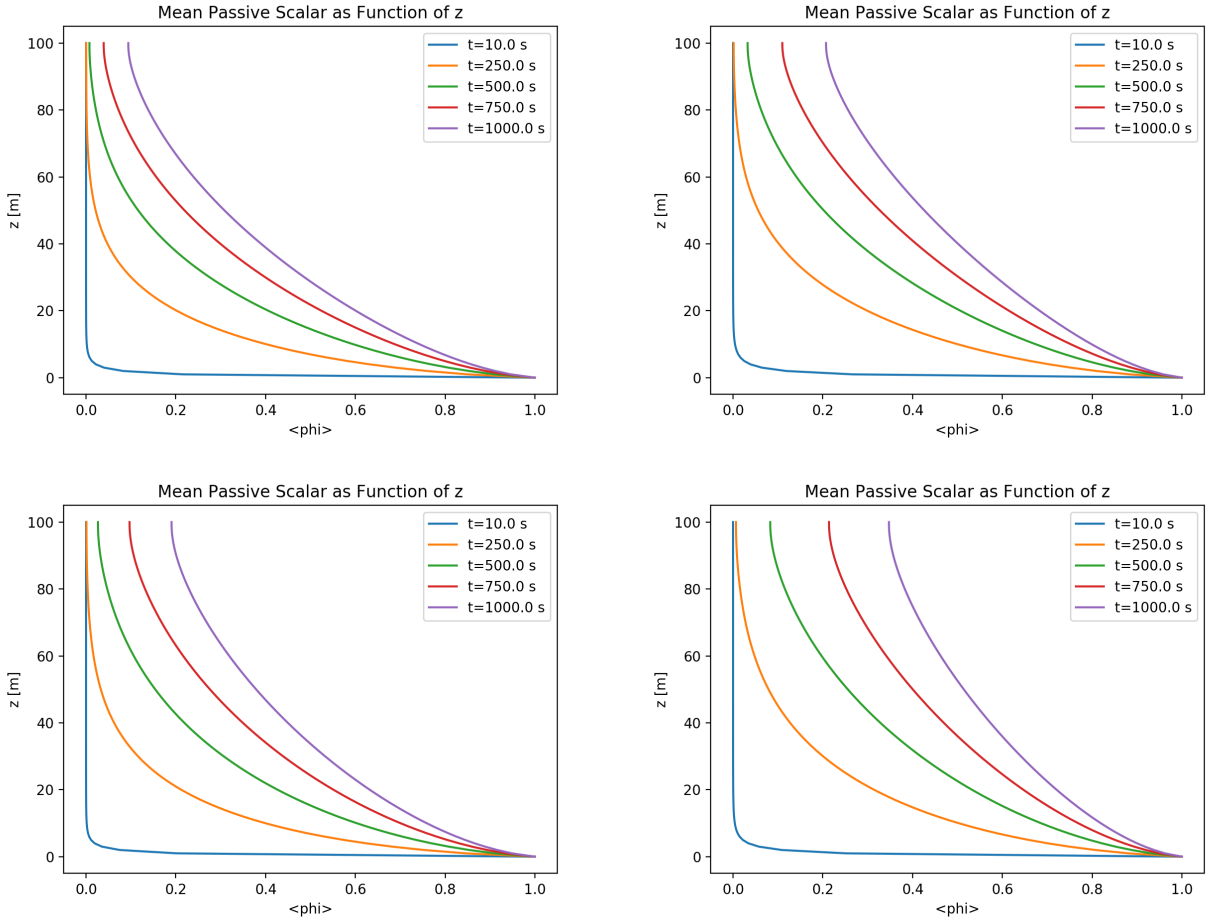


Figure 4: Passive scalar solution for case 1 (top left), case 2 (top right), case 3 (bottom left), and case 4 (bottom right)

ENGG*6790: Theory and Applications of Turbulence

A Simple Wall Model (1) for
1D Momentum and Turbulent Kinetic Energy Equations over Flat Surface

Amir A. Aliabadi

March 22, 2019

1 Introduction

Various wall functions were introduced in lectures. The purpose of this lab is to implement a simple wall function in the 1D momentum and turbulent kinetic energy equations. In addition, it is desired to investigate the sensitivity of the solutions as a function of the length of the first grid near the wall.

In previous labs, the equation for momentum and the effective viscosity, the sum of the molecular viscosity and the turbulent viscosity, were introduced as

$$\underbrace{\frac{\overline{D}}{\overline{D}t}\langle U_j\rangle}_{\text{Material Derivative of Mean}} = \underbrace{\frac{\partial}{\partial x_i}\left[\nu_{eff}\left(\frac{\partial\langle U_i\rangle}{\partial x_j}+\frac{\partial\langle U_j\rangle}{\partial x_i}\right)\right]}_{\text{Surface Forces and Reynolds Stress}} - \underbrace{\frac{1}{\rho}\frac{\partial}{\partial x_j}(\langle p\rangle+\frac{2}{3}\rho k)}_{\text{Modified Pressure}}, \tag{1}$$

$$\underbrace{\nu_{eff}(\mathbf{x},t)}_{\text{Effective Viscosity}} = \underbrace{\nu}_{\text{Molecular Viscosity}} + \underbrace{\nu_T(\mathbf{x},t)}_{\text{Turbulent Viscosity}} . \tag{2}$$

In addition, in the lectures the turbulent kinetic energy model was introduced as one transport equation to predict the turbulent kinetic energy. This turbulent kinetic energy was then used to formulate turbulent viscosity so that the momentum equation can be solved. The turbulent kinetic energy equation is given as

$$\underbrace{\frac{\overline{D}k}{\overline{D}t}}_{\text{Material Derivative}} \equiv \underbrace{\frac{\partial k}{\partial t}}_{\text{Storage}} + \underbrace{\langle U\rangle.\nabla k}_{\text{Advection}} = \underbrace{\nabla.\left(\frac{\nu_T}{\sigma_k}\nabla k\right)}_{\text{Energy Flux Divergence}} + \underbrace{\mathcal{P}}_{\text{Production}} - \underbrace{\epsilon}_{\text{Dissipation}} , \tag{3}$$

$$\begin{cases} \nu_T = ck^{1/2}\ell_m, \\ \epsilon = C_D \frac{k^{3/2}}{\ell_m}, \\ \ell_m(\mathbf{x}, t) \text{ known.} \end{cases}$$

This model can be employed to develop a one-dimensional transport model for momentum and turbulent kinetic energy under steady-state conditions. Suppose we use the Cartesian coordinate system with coordinate axes of x, y, and z, and velocities corresponding to these axes being $U = \langle U \rangle + u$, $V = \langle V \rangle + v$, $W = \langle W \rangle + w$, respectively, as shown in the schematic. We can assume that mean flow is only in the x direction parallel to the surface and that the direction z is normal to the surface so that $\langle V \rangle = \langle W \rangle = 0$.

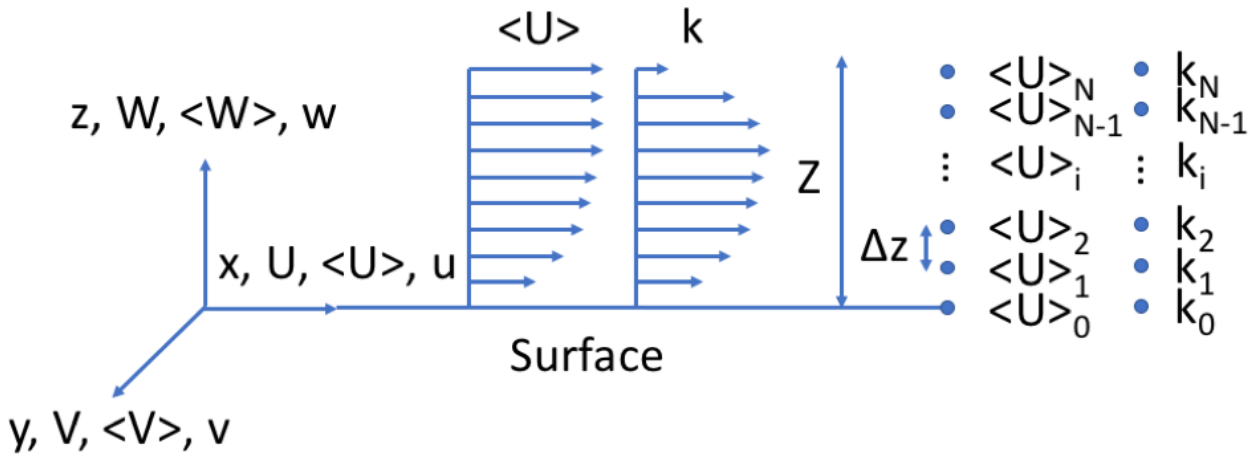


Figure 1: Schematic of 1D flow over flat surface using Cartesian coordinate system; finite difference representation of the 1D flow.

Assume that the modified pressure has a constant gradient in the x direction, the 1D momentum equation then simplifies to

$$0 = \underbrace{\frac{\partial}{\partial z}\left(\nu_T \frac{\partial \langle U \rangle}{\partial z}\right)}_{\text{Surface Forces and Reynolds Stress}} - \underbrace{\tau}_{\text{Modified Pressure Forces}} \tag{4}$$

The one-dimensional turbulent kinetic energy equation can be developed as follows

$$0 = \underbrace{\frac{\partial}{\partial z}\left(\frac{\nu_T}{\sigma_k}\frac{\partial k}{\partial z}\right)}_{\text{Energy Flux Divergence}} + \underbrace{\nu_T \left(\frac{\partial \langle U \rangle}{\partial z}\right)^2}_{\text{Shear Production}} - \underbrace{\epsilon}_{\text{Dissipation}} \tag{5}$$

where the energy flux divergence was discussed in the lectures. This term ensures that the resulting model transport equation for k yields smooth solutions, and that a boundary condition can be

imposed on k everywhere in the boundary of the domain. Otherwise the model may diverge if other transport mechanisms for k are much smaller than this term. The shear production term, is an example of a production term $\mathcal{P}$, that contributes to the generation of the turbulent kinetic energy. Here, when there is non-zero mean velocity gradient, turbulent kinetic energy is generated. The dissipation term is responsible for consuming turbulent kinetic energy down the energy cascade.

To close the turbulence model we can assume that the turbulent Prandtl number is unity, i.e. $\sigma_k = 1$. We can model turbulent viscosity, dissipation rate, and the appropriate mixing length as follows.

$$\begin{cases} \nu_T = C_k \ell_m k^{1/2}, \\ \epsilon = C_\epsilon \ell_m^{-1} k^{3/2}, \\ \ell_m = \kappa z / (1 + \frac{\kappa z}{\ell_0}). \end{cases}$$

where $\kappa = 0.41$ is the von Kármán constant, and ℓ_0 is the maximum mixing length. This formulation for mixing length has the nice property that it is bounded between zero and ℓ_0, which is physically sound since mixing length increases linearly in the log-law sublayer near a wall but cannot increase indefinitely in the interior of the domain. This formulation results in

$$\begin{cases} z \to 0 & \ell_m \to \kappa z \\ z \to \infty & \ell_m \to \ell_0 \end{cases}$$

So we have two equations: momentum and turbulent kinetic energy. We can eliminate ν_T and ϵ from the momentum and turbulent kinetic energy equations by direct substitutions and simplifications using the chain rule. So the two equations can be re-expressed as

$$\begin{aligned} 0 &= \frac{\partial}{\partial z}\left(C_k \ell_m k^{1/2} \frac{\partial \langle U \rangle}{\partial z} \right) - \tau \\ &= 0.5 C_k \ell_m k^{-1/2} \frac{\partial k}{\partial z} \frac{\partial \langle U \rangle}{\partial z} + C_k \ell_m k^{1/2} \frac{\partial^2 \langle U \rangle}{\partial z^2} - \tau \end{aligned} \tag{6}$$

$$\begin{aligned} 0 &= \frac{\partial}{\partial z}\left(C_k \ell_m k^{1/2} \frac{\partial k}{\partial z} \right) + C_k \ell_m k^{1/2} \left(\frac{\partial \langle U \rangle}{\partial z} \right)^2 - C_\epsilon \ell_m^{-1} k^{3/2} \\ &= 0.5 C_k \ell_m k^{-1/2} \left(\frac{\partial k}{\partial z} \right)^2 + C_k \ell_m k^{1/2} \frac{\partial^2 k}{\partial z^2} + C_k \ell_m k^{1/2} \left(\frac{\partial \langle U \rangle}{\partial z} \right)^2 - C_\epsilon \ell_m^{-1} k^{3/2}. \end{aligned} \tag{7}$$

These equations would apply to the interior of the domain. However, in the first node within the domain, i.e. the node adjacent to the wall (p), we should apply a wall function. This requires solving another transport equation for momentum. The concept of the wall function is to apply a value for the Reynolds stresses, or shear stresses, at the first computational node, instead of

applying a value for the mean momentum. As a result we should first look at the mean momentum equation without the turbulent viscosity hypothesis:

$$\underbrace{\frac{\overline{D}\langle U_j\rangle}{\overline{D}t}}_{\text{Material Derivative of Mean}} = \underbrace{\nu\nabla^2\langle U_j\rangle}_{\text{Surface Forces}} - \underbrace{\frac{\partial\langle u_i u_j\rangle}{\partial x_i}}_{\text{Reynolds Stresses}} - \underbrace{\frac{1}{\rho}\frac{\partial\langle p\rangle}{\partial x_j}}_{\text{Normal and Body Forces}} . \tag{8}$$

Since a wall function is being used, node p is in the turbulent boundary layer and we can neglect the surface forces due to molecular viscosity ν. In fact we can assume ν_T is 100 or 1000 times ν. We would also re-express the normal and body forces in terms of a modified pressure. Applying these changes to the steady 1D momentum equation we will obtain

$$0 = \underbrace{-\frac{\partial\langle uw\rangle}{\partial z}}_{\text{Reynolds Stress}} - \underbrace{\tau}_{\text{Modified Pressure Forces}} + \frac{2}{3}k \tag{9}$$

The term $\frac{2}{3}k$ must appear because the modified pressure requires $-\frac{2}{3}k$ to be added to the pressure gradient term. When we write the discretized version of this equation, we need the Reynolds stress at the wall as well as the Reynolds stress on the upper node. The Reynolds stress on the upper node can be approximated by the turbulent viscosity hypothesis, while the Reynolds stress on the wall comes from the wall function. Let us discuss the wall function first.

It must be recalled that Reynolds stress near the wall, friction velocity, and shear stress at the wall are related by

$$-\langle uw\rangle = u_\tau^2 = \frac{\tau_w}{\rho}. \tag{10}$$

It should also be remembered that it is possible to relate shear stress near the wall to the turbulent kinetic energy by

$$-\langle uw\rangle = u_\tau^2 = C_k k. \tag{11}$$

We first calculate a nominal friction velocity to approximate the friction velocity using the value of turbulent kinetic energy at node p

$$u_\tau^* = C_k^{1/2} k_p^{1/2}. \tag{12}$$

Next we calculate z_p^* using the location of the first node away from the wall, i.e. z_p, as follows

$$z_p^* = \frac{z_p u_\tau^*}{\nu}. \tag{13}$$

This allows us to find a nominal mean velocity using the log-law of the wall

$$\langle U \rangle_p^* = u_\tau^* \left(\frac{1}{\kappa} \ln z_p^* + B \right). \tag{14}$$

Finally we can express the Reynolds stress at node p using

$$-\langle uw \rangle_p = u_\tau^{*2} \frac{\langle U \rangle_p}{\langle U \rangle_p^*} \tag{15}$$

We can make multiple substitutions to express the Reynolds stress at node p only as a function of unknown momentum and turbulent kinetic energy solution variables

$$-\langle uw \rangle_p = \frac{C_k^{1/2} k_p^{1/2} \langle U \rangle_p}{\frac{1}{\kappa} \ln \left(\frac{C_k^{1/2} k_p^{1/2} z_p}{\nu} \right) + B}. \tag{16}$$

This equation can be solved iteratively along with other equations. However, in practice, because the calculation of Reynolds stress at node p is within an overall iterative procedure for the entire flow field, at each overall iteration, the equation for Reynolds stress at node p is solved only once, with the right-hand-side values taken to be those of the previous overall iteration.

It may be noticed that solving the new momentum equation is not very convenient as it requires a separate discretization scheme, involving the discretization of the derivative of the Reynolds stress. An alternative approach is to modify the turbulent viscosity for the first node, i.e. with an effective viscosity, ν_{eff}, that ensures the correct friction velocity or Reynolds stress, even though the velocity gradient is erroneous. In other words, for the first computational node, we can use

$$\nu_{eff} = \frac{-\langle uw \rangle_p}{\frac{\partial U}{\partial z}} \simeq -\langle uw \rangle_p \frac{2\Delta z}{\langle U \rangle_2 - \langle U \rangle_0} \tag{17}$$

We can calculate this effective viscosity once per iteration and use it in the simpler momentum equation

$$0 = \nu_{eff} \frac{d^2 \langle U \rangle}{dz^2} - \tau. \tag{18}$$

Using a finite difference scheme we can assume a uniform vertical discretization of Δz and approximate the second derivative of $\langle U \rangle$ with

$$\left(\frac{d^2 \langle U \rangle}{dz^2} \right)_1 \approx \frac{\langle U \rangle_2 - 2\langle U \rangle_1 + \langle U \rangle_0}{(\Delta z)^2}. \tag{19}$$

Therefore, the finite difference representation of the 1D momentum equation can be provided using the following equation, which can be rearranged to the following form for simplicity

$$0 = \nu_{eff} \frac{\langle U \rangle_2 - 2\langle U \rangle_1 + \langle U \rangle_0}{(\Delta z)^2} - \tau. \tag{20}$$

$$\langle U \rangle_0 - 2\langle U \rangle_1 + \langle U \rangle_2 = \frac{\tau(\Delta z)^2}{\nu_{eff}}. \tag{21}$$

So the momentum equation for the first computational node is discretized. We wish to simulate flow over flat surface on 9 mesh levels from very fine to very coarse with and without using the simple wall function. We desire to know how much the momentum solution on the top of the domain will change as a result of using coarser and coarser meshes, and if the use of the simple wall function will help to reduce solution change as a result of coarsening the mesh.

Table 1: Simulation cases with various mesh resolutions

Mesh Level	N
1	1000
2	500
3	250
4	100
5	50
6	25
7	10
8	5
9	2

2 Python Script

We first implement the script without using a wall function. Complete the following script and then run it for the 9 levels of mesh. Please record the value of the momentum solution on top of the domain for each simulation. You can see that the portion of the code responsible for implementing the simple wall function is commented out.

```
import random
import sys
import os
import numpy
import matplotlib.pyplot as plt

#Define under-relaxation factor
alpha=0.1
```

```
#Define horizontal pressure gradient divided by density [m s^-2]
tau=-0.005

#Define von Karman constant
kappa=0.41

#Define constant for the log-law of the wall
B=5.2

#Define air kinematic viscosity [m^2 s^-1]
nu=1.5e-5

#Define maximum mixing length [m]
l0=10

#Define turbulence model constants, Martilli et al. (2002)
Ck=0.4
Ce=0.71

#Define maximum iteration number
MaxIter=1000

#Define relative error
Err=0.01

#Define z axis from 0 to Z with dz increments
Z=100       #[m]
N=2
dz=Z/N      #[m]
z=numpy.linspace(0,Z,N+1)

#Define and initialize a mean velocity vector [m s^-1]
Uinitial=1
Umean=numpy.zeros((N+1,1))
Umean[:]=Uinitial

#Define and initialize turbulent kinetic energy [m^2 s^-s]
kinitial=0.1
k=numpy.zeros((N+1,1))
k[:]=kinitial

#Define unknown vector X, coefficient matrix A, and vector B, in AX=B
x=numpy.zeros((2*N+2,1))
xnew=numpy.zeros((2*N+2,1))
b=numpy.zeros((2*N+2,1))
```

```
a=numpy.zeros((2*N+2,2*N+2))

#Initialize solution vector X
#This is a short syntax for for loop
x[0]=0
x[1:N+1]=Umean[1:N+1]
x[N+1:2*N+2]=k[0:N+1]

for iter in range(1, MaxIter):
    #Momentum equations
    #i=0
    a[0][0]=1
    b[0]=0

    '''
    #i=1
    #Calculate the effective viscosity in the first node using wall function
    uTauStar=(Ck**0.5)*(x[1+N+1]**0.5)
    zpStar=dz*uTauStar/nu
    UmeanpStar=uTauStar*((1/kappa)*numpy.log(zpStar)+B)
    ReynoldsStressp=(uTauStar**2)*x[1]/UmeanpStar
    nuEff=ReynoldsStressp*(2*dz)/(x[2]-x[0])
    a[1][0]=...
    a[1][1]=...
    a[1][2]=...
    b[1]=...
    '''

    #i=1 to N-1
    for i in range(1, N):
        #Calculate derivatives by finite differences for the current i index
        #Remember to shift indices by N+1 if needed
        lm=kappa*z[i]/(1+(kappa*z[i])/l0)
        k0=x[i+N+1]
        k1=(x[i+1+N+1]-x[i-1+N+1])/(2*dz)
        k2=(x[i+1+N+1]-2*x[i+N+1]+x[i-1+N+1])/(dz**2)
        Umean0=x[i]
        Umean1=(x[i+1]-x[i-1])/(2*dz)
        Umean2=(x[i+1]-2*x[i]+x[i-1])/(dz**2)
        # Set constants necessary to build the coefficient matrix
        c1=0.5*Ck*lm*k0**(-1/2)*k1
        c2=Ck*lm*k0**(1/2)
        c3=-0.25*Ck*lm*k0**(-3/2)*k1*Umean1+0.5*Ck*lm*k0**(-1/2)*Umean2
        c4=0.5*Ck*lm*k0**(-1/2)*Umean1
        cb=-(-0.25*Ck*lm*k0**(-1/2)*k1*Umean1-0.5*Ck*lm*k0**(-1/2)*Umean2*k0-tau)
        # Set the coefficient matrix and the B vector
```

```
    a[i][i-1]=-c1/(2*dz)+c2/(dz**2)
    a[i][i]=-2*c2/(dz**2)
    a[i][i+1]=c1/(2*dz)+c2/(dz**2)
    a[i][i-1+N+1]=-c4/(2*dz)
    a[i][i+N+1]=c3
    a[i][i+1+N+1]=c4/(2*dz)
    b[i]=cb
#i=N
a[N][N-1]=1
a[N][N]=-1
b[N]=0

#Kinetic energy equations
#i=N+1
a[N+1][N+1]=1
b[N+1]=0
#i=N+2 to 2N
for i in range(N+2, 2*N+1):
    #Calculate derivatives by finite differences for the current i index
    #Shift indices by -(N+1) if needed
    lm=kappa*z[i-(N+1)]/(1+(kappa*z[i-(N+1)])/l0)
    k0=x[i]
    k1=(x[i+1]-x[i-1])/(2*dz)
    k2=(x[i+1]-2*x[i]+x[i-1])/(dz ** 2)
    Umean0=x[i-(N+1)]
    Umean1=(x[i+1-(N+1)]-x[i-1-(N+1)])/(2*dz)
    Umean2=(x[i+1-(N+1)]-2*x[i-(N+1)]+x[i-1-(N+1)])/(dz**2)
    #Set constants necessary to build the coefficient matrix
    d1=-0.25*Ck*lm*k0**(-3/2)*k1**2+0.5*Ck*lm*k0**(-1/2)*k2\
         +0.5*Ck*lm*k0**(-1/2)*Umean1**2-1.5*Ce*lm**(-1)*k0**(1/2)
    d2=Ck*lm*k0**(-1/2)*k1
    d3=Ck*lm*k0**(1/2)
    d4=2*Ck*lm*k0**(1/2)*Umean1
    db=-(-0.25*Ck*lm*k0**(-1/2)*k1**2-0.5*Ck*lm*k0**(1/2)*k2)\
         -(-1.5*Ck*lm*k0**(1/2)*Umean1**2+0.5*Ce*lm**(-1)*k0**(3/2))
    # Set the coefficient matrix and the B vector
    a[i][i-1-(N+1)]=-d4/(2*dz)
    a[i][i+1-(N+1)]=d4/(2*dz)
    a[i][i-1]=-d2/(2*dz)+d3/(dz**2)
    a[i][i]=d1-2*d3/(dz**2)
    a[i][i+1]=d2/(2*dz)+d3/(dz**2)
    b[i]=db
# i=2N+1
a[2*N+1][2*N]=1
a[2*N+1][2*N+1]=-1
b[2*N+1]=0
```

```
    xnew = numpy.linalg.solve(a, b)

    # Calculate maximum norm errors for both momentum and turbulent kinetic energy
    ErrUmean=numpy.max(numpy.abs(numpy.divide(xnew[1:N+1]-x[1:N+1],x[1:N+1])))
    Errk=numpy.max(numpy.abs(numpy.divide(xnew[N+2:2*N+1]-x[N+2:2*N+1],x[N+2:2*N+1])))

    print('Iteration=',iter,'ErrUmean=',ErrUmean,'Errk=',Errk)

    if ErrUmean < Err and Errk < Err:
        print('Solutions converged at iteration: ', iter)
        # Exit the loop
        break

    # Update solution
    x[:]=x[:]+alpha*(xnew[:]-x[:])

#Assign the X vector to the original Umean and k vectors
Umean[0:N+1]=x[0:N+1]
k[0:N+1]=x[N+1:2*N+2]

#Print the solution monitor as the velocity on top of the model domain
print('Highest Level Umean=',Umean[N])
```

Now change the script so the simple wall function can be used. You need to remove the comments. Note that the variable of iteration for the loop following the wall function must now start from 2. The fragment of the code to be changed is given below.

```
    ...
    #Momentum equations
    #i=0
    a[0][0]=1
    b[0]=0

    #i=1
    #Calculate the effective viscosity in the first node using wall function
    uTauStar=(Ck**0.5)*(x[1+N+1]**0.5)
    zpStar=dz*uTauStar/nu
    UmeanpStar=uTauStar*((1/kappa)*numpy.log(zpStar)+B)
    ReynoldsStressp=(uTauStar**2)*x[1]/UmeanpStar
    nuEff=ReynoldsStressp*(2*dz)/(x[2]-x[0])
    a[1][0]=...
    a[1][1]=...
    a[1][2]=...
    b[1]=...
```

```
#i=2 to N-1
for i in range(2, N):
...
```

After running the code with the implemented wall function for the 9 levels of mesh, we can obtain the following values for momentum solution on top of the domain. Table below summarizes the solutions for both without and with using wall functions.

Table 2: Simulation cases with various mesh resolutions

Mesh Level	N	$\langle U \rangle_N$ [m s^{-1}]	$\langle U \rangle_N$ [m s^{-1}]
		Without Wall Function	With Wall Function 1
1	1000	17.0	17.0
2	500	16.9	16.8
3	250	16.7	16.6
4	100	16.3	16.4
5	50	15.9	16.1
6	25	15.3	15.7
7	10	14.2	15.2
8	5	12.9	14.7
9	2	10.2	12.4

Try to answer the following questions.

- Suppose that level 1 ($N = 1000$) provides the most accurate solution. Which series of simulations shows less variation in the solution by coarsening the mesh?
- Is the effect of using wall functions more noticeable on the fine mesh or coarse mesh?
- Calculate the relative error for the momentum solution between each mesh level and the first level (finest) for each series of simulations. Which series exhibits smaller errors at each mesh level? the series with or without using wall functions?
- If you are limited by computational resources and must use a relatively coarse mesh for a simulation job, would you use wall functions?

ENGG*6790: Theory and Applications of Turbulence

A Simple Wall Model (2) for
1D Momentum and Turbulent Kinetic Energy Equations over Flat Surface

Amir A. Aliabadi

March 22, 2019

1 Introduction

A simple wall function (1) was used in a previous lab. In this lab we implement another simple wall function (2) for comparison. Again, we develop a one-dimensional transport model for momentum and turbulent kinetic energy under steady-state conditions. Suppose we use the Cartesian coordinate system with coordinate axes of x, y, and z, and velocities corresponding to these axes being $U = \langle U \rangle + u$, $V = \langle V \rangle + v$, $W = \langle W \rangle + w$, respectively, as shown in the schematic. We can assume that mean flow is only in the x direction parallel to the surface and that the direction z is normal to the surface so that $\langle V \rangle = \langle W \rangle = 0$.

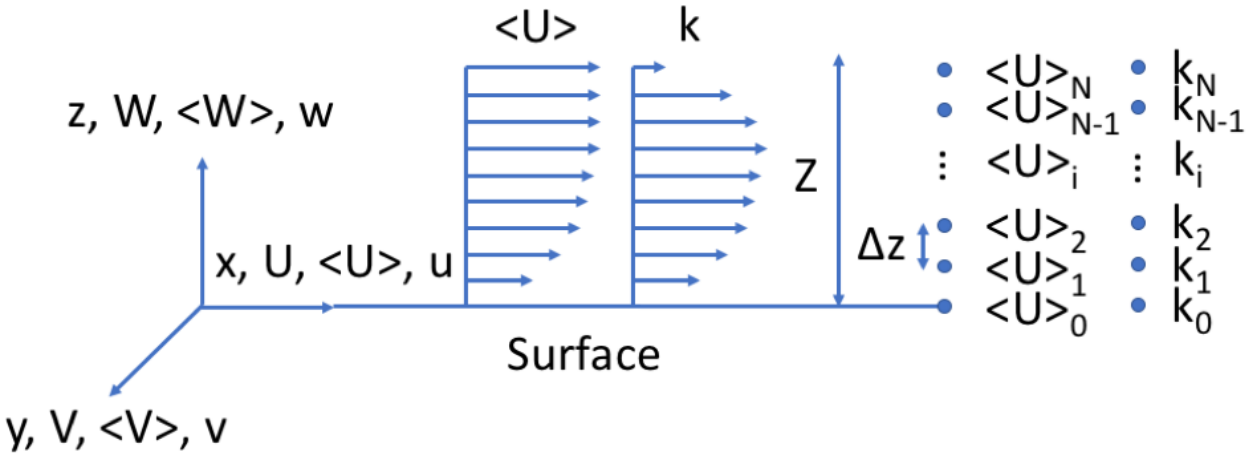


Figure 1: Schematic of 1D flow over flat surface using Cartesian coordinate system; finite difference representation of the 1D flow.

Before introducing wall function (2), it must be recalled that Reynolds stress near the wall, friction velocity, and shear stress at the wall are related by

$$-\langle uw \rangle = u_\tau^2 = \frac{\tau_w}{\rho}. \tag{1}$$

It should also be remembered that it is possible to relate shear stress near the wall to the turbulent kinetic energy by

$$-\langle uw \rangle = u_\tau^2 = C_k k. \tag{2}$$

The new wall function evaluates a nominal friction velocity iteratively by successively solving the following equation as part of the overall solver iteration

$$u_\tau^* = \frac{\langle U \rangle_p^*}{\left(\frac{1}{\kappa} \ln z_p^* + B\right)} \tag{3}$$

where $\langle U \rangle_p^*$ is the most recent solution at node p for momentum. Of course z_p^* itself is a function of the nominal friction velocity such that

$$z_p^* = \frac{z_p u_\tau^*}{\nu} \tag{4}$$

which requires that nominal friction velocity be initialized before the iterations. A convenient way for this initialization is to use the relationship between friction velocity and the turbulent kinetic energy, i.e. $u_\tau^2 = C_k k$, at the beginning of the simulation.

Again, at each iteration, we modify the turbulent viscosity for the first node, i.e. with an effective viscosity, ν_{eff}, that ensures the correct friction velocity or Reynolds stress, even though the velocity gradient is erroneous. In other words, for the first computational node, we can use

$$\nu_{eff} = \frac{-\langle uw \rangle_p}{\frac{\partial U}{\partial z}} = \frac{u_\tau^{*2}}{\frac{\partial U}{\partial z}} \simeq -\langle uw \rangle_p \frac{2\Delta z}{\langle U \rangle_2 - \langle U \rangle_0} = u_\tau^{*2} \frac{2\Delta z}{\langle U \rangle_2 - \langle U \rangle_0} \tag{5}$$

We can calculate this effective viscosity once per iteration and use it in the simpler momentum equation

$$0 = \nu_{eff} \frac{d^2 \langle U \rangle}{dz^2} - \tau. \tag{6}$$

Next, instead of evaluating the nominal friction velocity from the turbulent kinetic energy at node p, we evaluate the turbulent kinetic energy at node p from the most recent update of the nominal friction velocity

$$k_p = \frac{u_\tau^{*2}}{C_k}. \tag{7}$$

Using a finite difference scheme we can assume a uniform vertical discretization of Δz and approximate the second derivative of $\langle U \rangle$ with

$$\left(\frac{d^2\langle U\rangle}{dz^2}\right)_1 \approx \frac{\langle U\rangle_2 - 2\langle U\rangle_1 + \langle U\rangle_0}{(\Delta z)^2}. \tag{8}$$

Therefore, the finite difference representation of the 1D momentum equation can be provided using the following equation, which can be rearranged to the following form for simplicity

$$0 = \nu_{eff}\frac{\langle U\rangle_2 - 2\langle U\rangle_1 + \langle U\rangle_0}{(\Delta z)^2} - \tau. \tag{9}$$

$$\langle U\rangle_0 - 2\langle U\rangle_1 + \langle U\rangle_2 = \frac{\tau(\Delta z)^2}{\nu_{eff}}. \tag{10}$$

So the momentum equation for the first computational node is discretized. Likewise, the turbulent kinetic energy equation for the first computational node is discretized as

$$k_1 = \frac{-\langle uw\rangle_p}{C_k} = \frac{u_\tau^{*2}}{C_k} \tag{11}$$

We wish to simulate flow over flat surface on 9 mesh levels from very fine to very coarse with this new simple wall function. We desire to know how much the momentum solution on the top of the domain will change as a result of using coarser and coarser meshes, and if the use of this simple wall function will help to reduce solution change as a result of coarsening the mesh.

Table 1: Simulation cases with various mesh resolutions

Mesh Level	N
1	1000
2	500
3	250
4	100
5	50
6	25
7	10
8	5
9	2

2 Python Script

Complete the following script and then run it for the 9 levels of mesh. Please record the value of the momentum solution on top of the domain for each simulation.

```
import random
import sys
import os
import numpy
import matplotlib.pyplot as plt

#Define under-relaxation factor
alpha=0.1

#Define horizontal pressure gradient divided by density [m s^-2]
tau=-0.005

#Define von Karman constant
kappa=0.41

#Define constant for the log-law of the wall
B=5.2

#Define air kinematic viscosity [m^2 s^-1]
nu=1.5e-5

#Define maximum mixing length [m]
l0=10

#Define turbulence model constants, Martilli et al. (2002)
Ck=0.4
Ce=0.71

#Define maximum iteration number
MaxIter=1000

#Define relative error
Err=0.01

#Define z axis from 0 to Z with dz increments
Z=100       #[m]
N=100
dz=Z/N      #[m]
z=numpy.linspace(0,Z,N+1)
```

```
#Define and initialize a mean velocity vector [m s^-1]
Uinitial=1
Umean=numpy.zeros((N+1,1))
Umean[:]=Uinitial

#Define and initialize turbulent kinetic energy [m^2 s^-s]
kinitial=0.1
k=numpy.zeros((N+1,1))
k[:]=kinitial

#Define and initialize friction velocity
uTauStar=(Ck**0.5)*(k[1]**0.5)

#Define unknown vector X, coefficient matrix A, and vector B, in AX=B
x=numpy.zeros((2*N+2,1))
xnew=numpy.zeros((2*N+2,1))
b=numpy.zeros((2*N+2,1))
a=numpy.zeros((2*N+2,2*N+2))

#Initialize solution vector X
#This is a short syntax for for loop
x[0]=0
x[1:N+1]=Umean[1:N+1]
x[N+1:2*N+2]=k[0:N+1]

for iter in range(1, MaxIter):
    #Momentum equations
    #i=0
    a[0][0]=1
    b[0]=0

    #i=1
    #Calculate the effective viscosity in the first node using wall function
    zpStar=dz*uTauStar/nu
    UmeanpStar=x[1]
    #Update nominal friction velocity based on the latest solutions
    uTauStar=UmeanpStar/((1/kappa)*numpy.log(zpStar)+B)
    nuEff=(uTauStar**2)*(2*dz)/(x[2]-x[0])
    a[1][0]=...
    a[1][1]=...
    a[1][2]=...
    b[1]=...

    #i=2 to N-1
    for i in range(2, N):
        #Calculate derivatives by finite differences for the current i index
```

```
    #Remember to shift indices by N+1 if needed
    lm=kappa*z[i]/(1+(kappa*z[i])/l0)
    k0=x[i+N+1]
    k1=(x[i+1+N+1]-x[i-1+N+1])/(2*dz)
    k2=(x[i+1+N+1]-2*x[i+N+1]+x[i-1+N+1])/(dz**2)
    Umean0=x[i]
    Umean1=(x[i+1]-x[i-1])/(2*dz)
    Umean2=(x[i+1]-2*x[i]+x[i-1])/(dz**2)
    # Set constants necessary to build the coefficient matrix
    c1=0.5*Ck*lm*k0**(-1/2)*k1
    c2=Ck*lm*k0**(1/2)
    c3=-0.25*Ck*lm*k0**(-3/2)*k1*Umean1+0.5*Ck*lm*k0**(-1/2)*Umean2
    c4=0.5*Ck*lm*k0**(-1/2)*Umean1
    cb=-(-0.25*Ck*lm*k0**(-1/2)*k1*Umean1-0.5*Ck*lm*k0**(-1/2)*Umean2*k0-tau)
    # Set the coefficient matrix and the B vector
    a[i][i-1]=-c1/(2*dz)+c2/(dz**2)
    a[i][i]=-2*c2/(dz**2)
    a[i][i+1]=c1/(2*dz)+c2/(dz**2)
    a[i][i-1+N+1]=-c4/(2*dz)
    a[i][i+N+1]=c3
    a[i][i+1+N+1]=c4/(2*dz)
    b[i]=cb
#i=N
a[N][N-1]=1
a[N][N]=-1
b[N]=0

#Kinetic energy equations
#i=N+1
a[N+1][N+1]=1
b[N+1]=0

#i=N+2
a[N+2][N+2]=...
b[N+2]=...

#i=N+3 to 2N
for i in range(N+3, 2*N+1):
    #Calculate derivatives by finite differences for the current i index
    #Shift indices by -(N+1) if needed
    lm=kappa*z[i-(N+1)]/(1+(kappa*z[i-(N+1)])/l0)
    k0=x[i]
    k1=(x[i+1]-x[i-1])/(2*dz)
    k2=(x[i+1]-2*x[i]+x[i-1])/(dz ** 2)
    Umean0=x[i-(N+1)]
    Umean1=(x[i+1-(N+1)]-x[i-1-(N+1)])/(2*dz)
```

```
        Umean2=(x[i+1-(N+1)]-2*x[i-(N+1)]+x[i-1-(N+1)])/(dz**2)
        #Set constants necessary to build the coefficient matrix
        d1=-0.25*Ck*lm*k0**(-3/2)*k1**2+0.5*Ck*lm*k0**(-1/2)*k2\
             +0.5*Ck*lm*k0**(-1/2)*Umean1**2-1.5*Ce*lm**(-1)*k0**(1/2)
        d2=Ck*lm*k0**(-1/2)*k1
        d3=Ck*lm*k0**(1/2)
        d4=2*Ck*lm*k0**(1/2)*Umean1
        db=-(-0.25*Ck*lm*k0**(-1/2)*k1**2-0.5*Ck*lm*k0**(1/2)*k2)\
             -(-1.5*Ck*lm*k0**(1/2)*Umean1**2+0.5*Ce*lm**(-1)*k0**(3/2))
        # Set the coefficient matrix and the B vector
        a[i][i-1-(N+1)]=-d4/(2*dz)
        a[i][i+1-(N+1)]=d4/(2*dz)
        a[i][i-1]=-d2/(2*dz)+d3/(dz**2)
        a[i][i]=d1-2*d3/(dz**2)
        a[i][i+1]=d2/(2*dz)+d3/(dz**2)
        b[i]=db
    # i=2N+1
    a[2*N+1][2*N]=1
    a[2*N+1][2*N+1]=-1
    b[2*N+1]=0

    xnew = numpy.linalg.solve(a, b)

    # Calculate maximum norm errors for both momentum and turbulent kinetic energy
    ErrUmean=numpy.max(numpy.abs(numpy.divide(xnew[1:N+1]-x[1:N+1],x[1:N+1])))
    Errk=numpy.max(numpy.abs(numpy.divide(xnew[N+2:2*N+1]-x[N+2:2*N+1],x[N+2:2*N+1])))

    print('Iteration=',iter,'ErrUmean=',ErrUmean,'Errk=',Errk)

    if ErrUmean < Err and Errk < Err:
        print('Solutions converged at iteration: ', iter)
        # Exit the loop
        break

    # Update solution
    x[:]=x[:]+alpha*(xnew[:]-x[:])

#Assign the X vector to the original Umean and k vectors
Umean[0:N+1]=x[0:N+1]
k[0:N+1]=x[N+1:2*N+2]

#Print the solution monitor as the velocity on top of the model domain
print('Highest Level Umean=',Umean[N])
```

After running the code with the implemented wall function for the 9 levels of mesh, we can obtain the following values for momentum solution on top of the domain. Table below also summarizes

the solutions for simulations from a previous lab where no wall function and wall function (1) were used.

Table 2: Simulation cases with various mesh resolutions

Mesh Level	N	$\langle U \rangle_N$ [m s^{-1}] Without Wall Function	$\langle U \rangle_N$ [m s^{-1}] With Wall Function 1	$\langle U \rangle_N$ [m s^{-1}] With Wall Function 2
1	1000	17.0	17.0	17.1
2	500	16.9	16.8	17.1
3	250	16.7	16.6	17.1
4	100	16.3	16.4	17.2
5	50	15.9	16.1	17.3
6	25	15.3	15.7	17.6
7	10	14.2	15.2	18.4
8	5	12.9	14.7	19.1
9	2	10.2	12.4	14.0

Try to answer the following questions.

- Suppose that level 1 ($N = 1000$) provides the most accurate solution. Which series of simulations shows less variation in the solution by coarsening the mesh?
- Is the effect of using wall function (2) more noticeable on the fine mesh or coarse mesh?
- Calculate the relative error for the momentum solution between each mesh level and the first level (finest) for each series of simulations. Which series exhibits smaller errors at each mesh level? the series with or without using wall functions?
- If you are limited by computational resources and must use a relatively coarse mesh for a simulation job, would you use wall function (1) or wall function (2)?

ENGG*6790: Theory and Applications of Turbulence

Box Filtering a 1D Velocity Field

Amir A. Aliabadi

November 11, 2017

1 Introduction

In the lectures on Large-eddy Simulation (LES) the idea of filtering a velocity field was introduced. In this lab a box filter with width Δ will be used to filter an instantaneous 1D velocity field $U(x)$ at a given time. The box filter $G(r)$ is given as

$$\begin{cases} \frac{1}{\Delta} & \text{if } |x-r| < \frac{1}{2}\Delta \\ 0 & \text{otherwise} \end{cases}$$

where the filter simply gives the average of $U(x')$ at x' in the interval $x - \frac{1}{2}\Delta < x' < x + \frac{1}{2}\Delta$. We will generate a noisy velocity field using superimposed sine waves at different frequencies and amplitudes consisting of 100 data points. The filtering will be performed by averaging 3, 9, and 27 velocities around each velocity point.

2 Python Script

Complete the following script to generate the instantaneous velocity field. The command `numpy.sin()` gives the sine of an argument in radians. Note that number π can be generated using the command `numpy.pi`. After executing this code you can obtain the velocity field as shown in the figure below.

```
import random
import sys
import os
import numpy
import matplotlib.pyplot as plt

#Define x axis from 0 to X
X=100       #[m]
```

```
N=100
x=numpy.linspace(0, X, N+1)

#Define box filter width as an odd number of indices greater than or equal to 3
nFilter=3

#Define a 1D velocity field by superimposing sine functions [m s^-1]
U=numpy.sin(1*2*numpy.pi*x/X)+\
  (1/2)*numpy.sin(4*2*numpy.pi*x/X)+\
  (1/4)*numpy.sin(8*2*numpy.pi*x/X)+\
  (1/8)*numpy.sin(16*2*numpy.pi*x/X)+\
  (1/16)*numpy.sin(32*2*numpy.pi*x/X)+\
  (1/32)*numpy.sin(64*2*numpy.pi*x/X)

#Plot the velocity versus x
plt.plot(x,U)
plt.xlabel('x [m]')
plt.ylabel('U [m s^-1]')
plt.title('Velocity as Function of x')
plt.show()
```

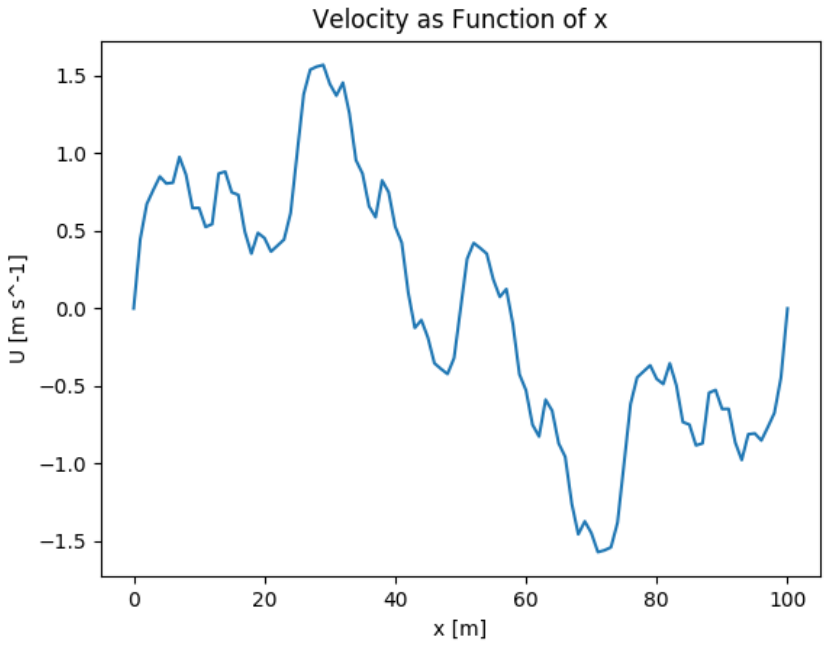


Figure 1: Plot of the instantaneous 1D velocity field versus x.

Next, complete the following script, which calculates a box filtered velocity field. Note that in this code Δ is not actually used. Instead, the velocity field is averaged having the number of data points in the filter, which is the same as the `nFilter` variable. For the nested `for` loops, we have carefully defined the starting index and finishing index for each loop. For the outer loop, we need

to start and finish at `i` indices for which we can access indices before and after, as required for averaging. For the inner loop we define the starting and finishing `j` indices in such a way that the average is centred at the current `i` index.

```
#Define and initialize the 1D filtered velocity field
UFilter=numpy.zeros((N+1))

#Box filter the 1D velocity field
#Outer iteration for velocity at each x value
#Inner iteration for averaging the velocity in the box width

#Calculate start and finish indices for the center of the box
#Must have these indices as integers
iStart=int((nFilter-1)/2)
iFinish=int(N-((nFilter-1)/2)+1)

for i in range(iStart, iFinish):
    # Calculate start and finish indices for each box
    # Must have these indices as integers
    jStart=int(i-(nFilter-1)/2)
    jFinish=int(i+(nFilter-1)/2+1)
    for j in range (jStart, jFinish):
        UFilter[i]=UFilter[i]+...

#Plot the filtered velocity versus x
plt.plot(x,UFilter)
plt.xlabel('x [m]')
plt.ylabel('UFilter [m s^-1]')
plt.title('Filtered Velocity as Function of x')
plt.show()
```

Upon completing the code. You should get the following figures. Note that you should re-run the code with different values for `nFilter`.

Try to answer the following questions.

- Describe the effect of the filter width on the filtered velocity.
- Why are the ends of the x domain for velocity field clipped, i.e. pinched to zero?

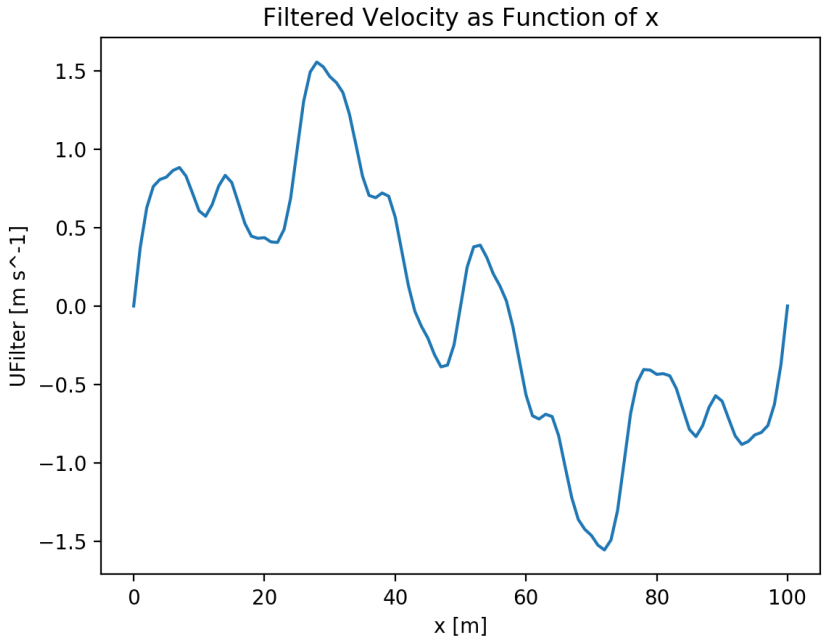


Figure 2: Plot of the filtered 1D velocity field versus x for `nFilter` equal to 3.

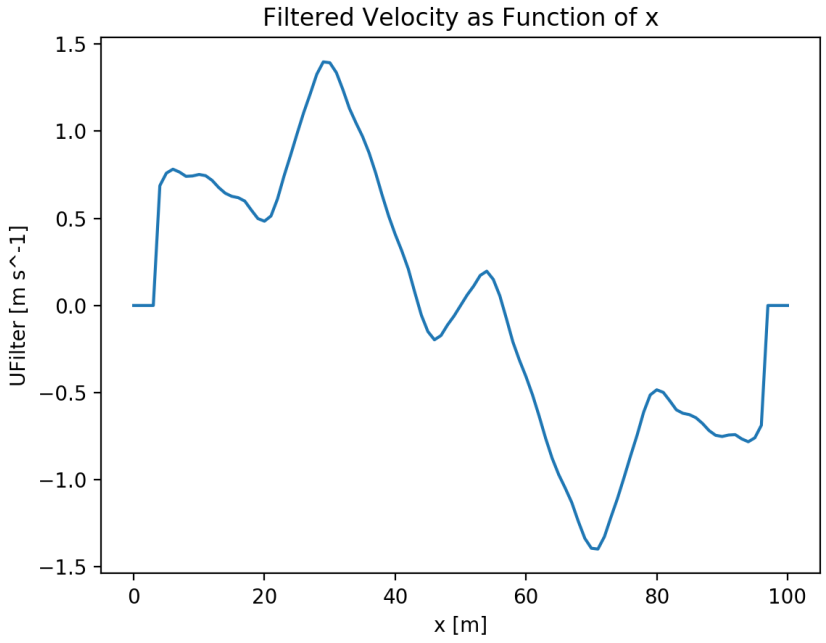


Figure 3: Plot of the filtered 1D velocity field versus x for `nFilter` equal to 9.

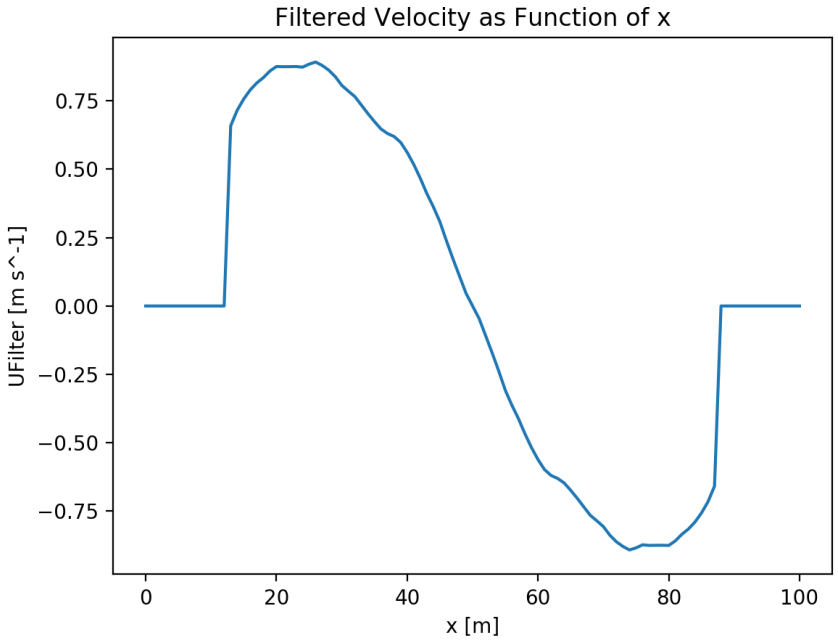


Figure 4: Plot of the filtered 1D velocity field versus x for `nFilter` equal to 27.