

Environmental Fluid Mechanics

Introduction to Python Programming

Amir A. Aliabadi

October 23, 2021

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.



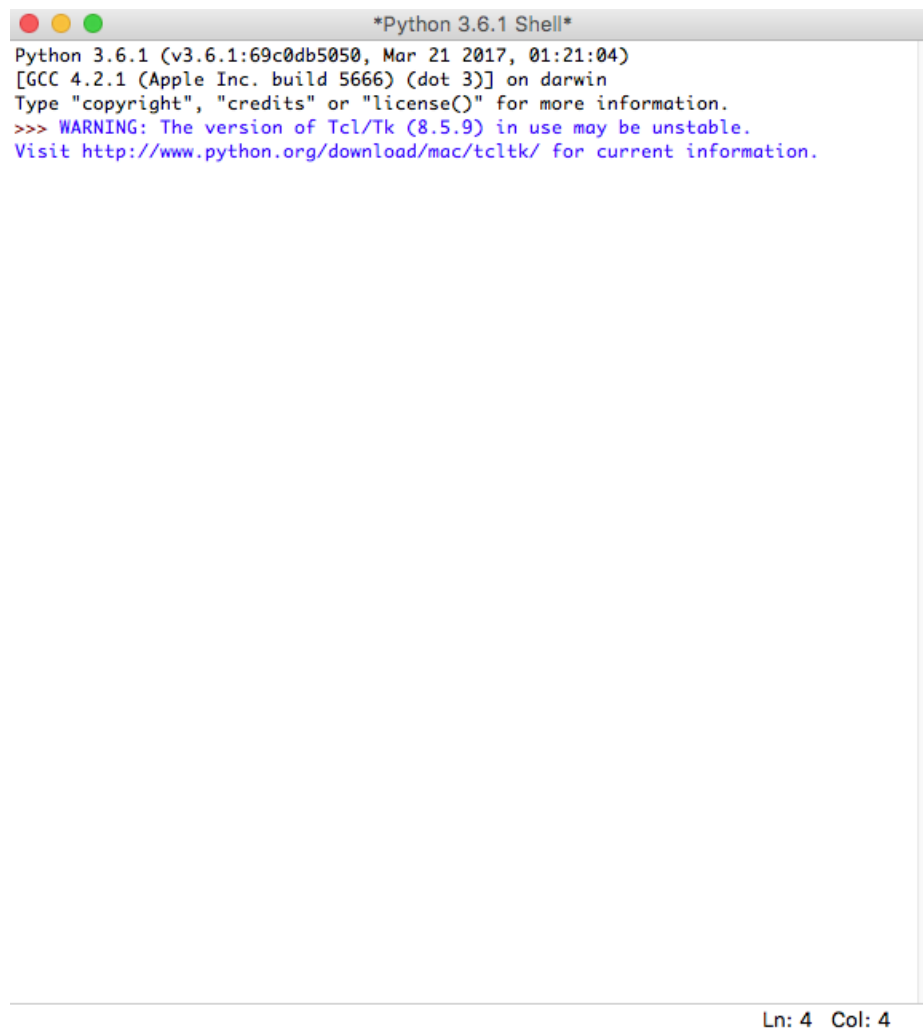
Figure 1: The logo of **Python** programming language.

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.



```
*Python 3.6.1 Shell*
Python 3.6.1 (v3.6.1:69c0db5050, Mar 21 2017, 01:21:04)
[GCC 4.2.1 (Apple Inc. build 5666) (dot 3)] on darwin
Type "copyright", "credits" or "license()" for more information.
>>> WARNING: The version of Tcl/Tk (8.5.9) in use may be unstable.
Visit http://www.python.org/download/mac/tcltk/ for current information.
```

Ln: 4 Col: 4

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: Opening 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**.

```
#Introduction to python programming
import numpy

#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.

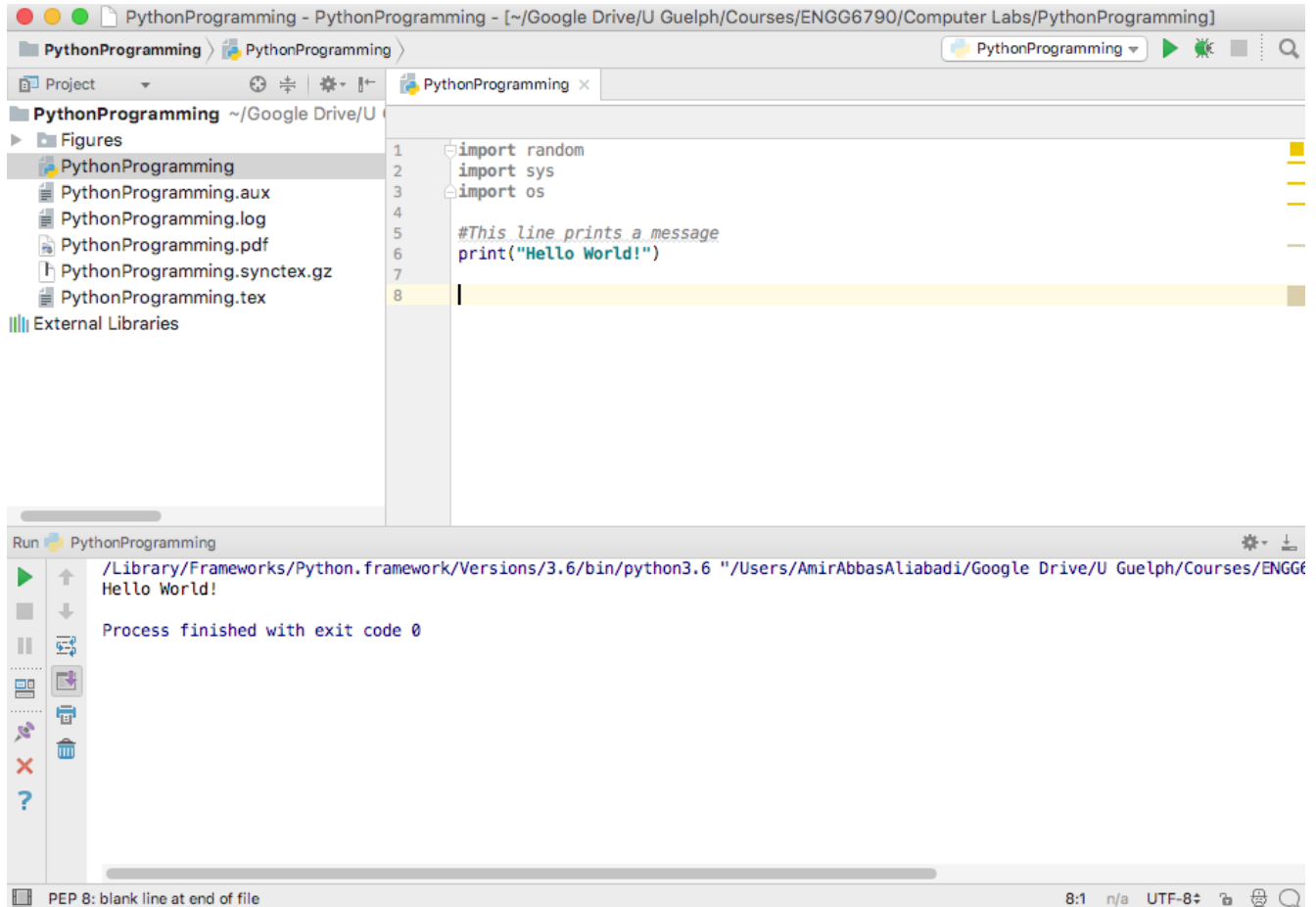


Figure 4: A simple program to print `Hello World!` in the output console.

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 interpreters sorted 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

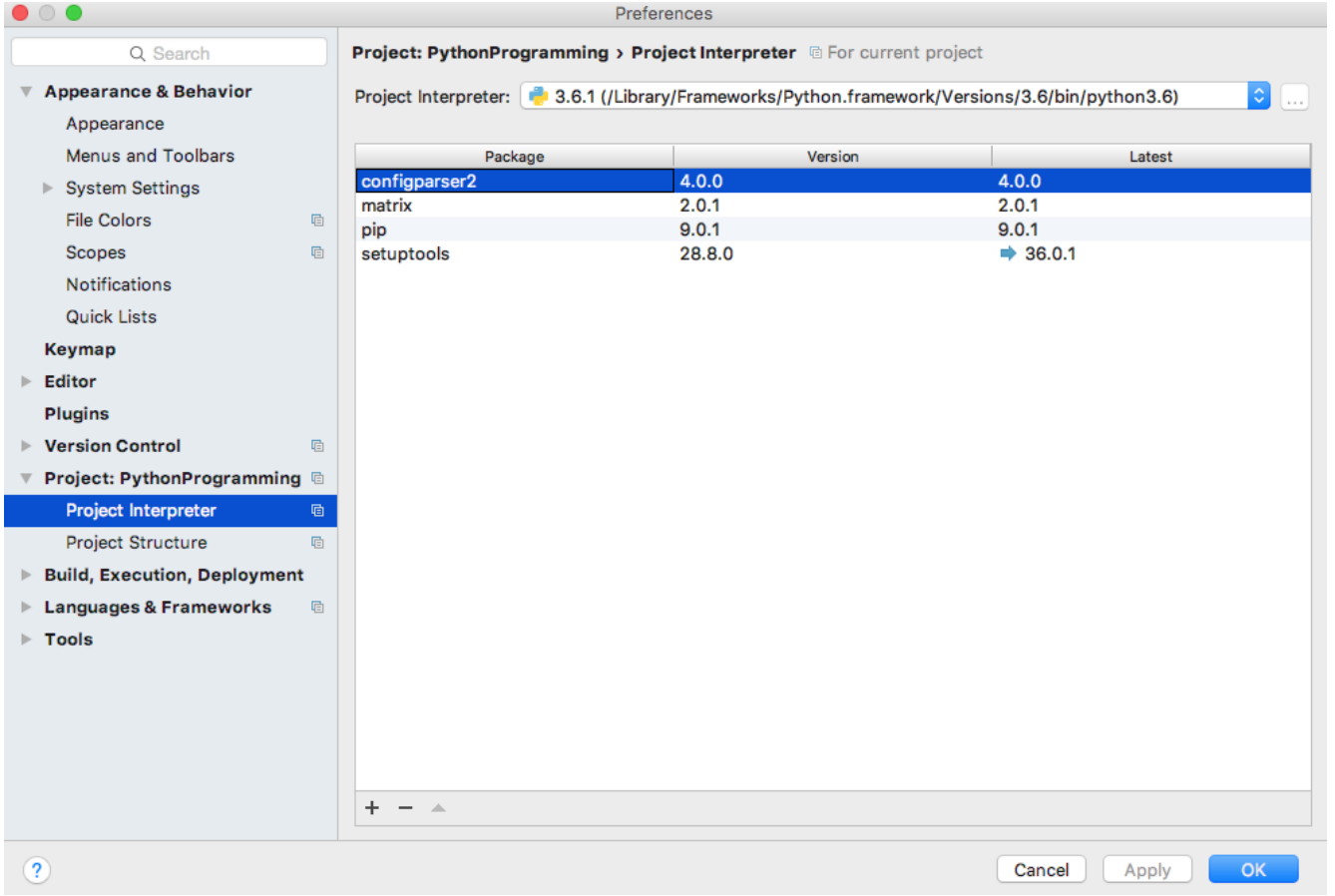


Figure 5: Preferences window and the Project Interpreter.

bottom **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.

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

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

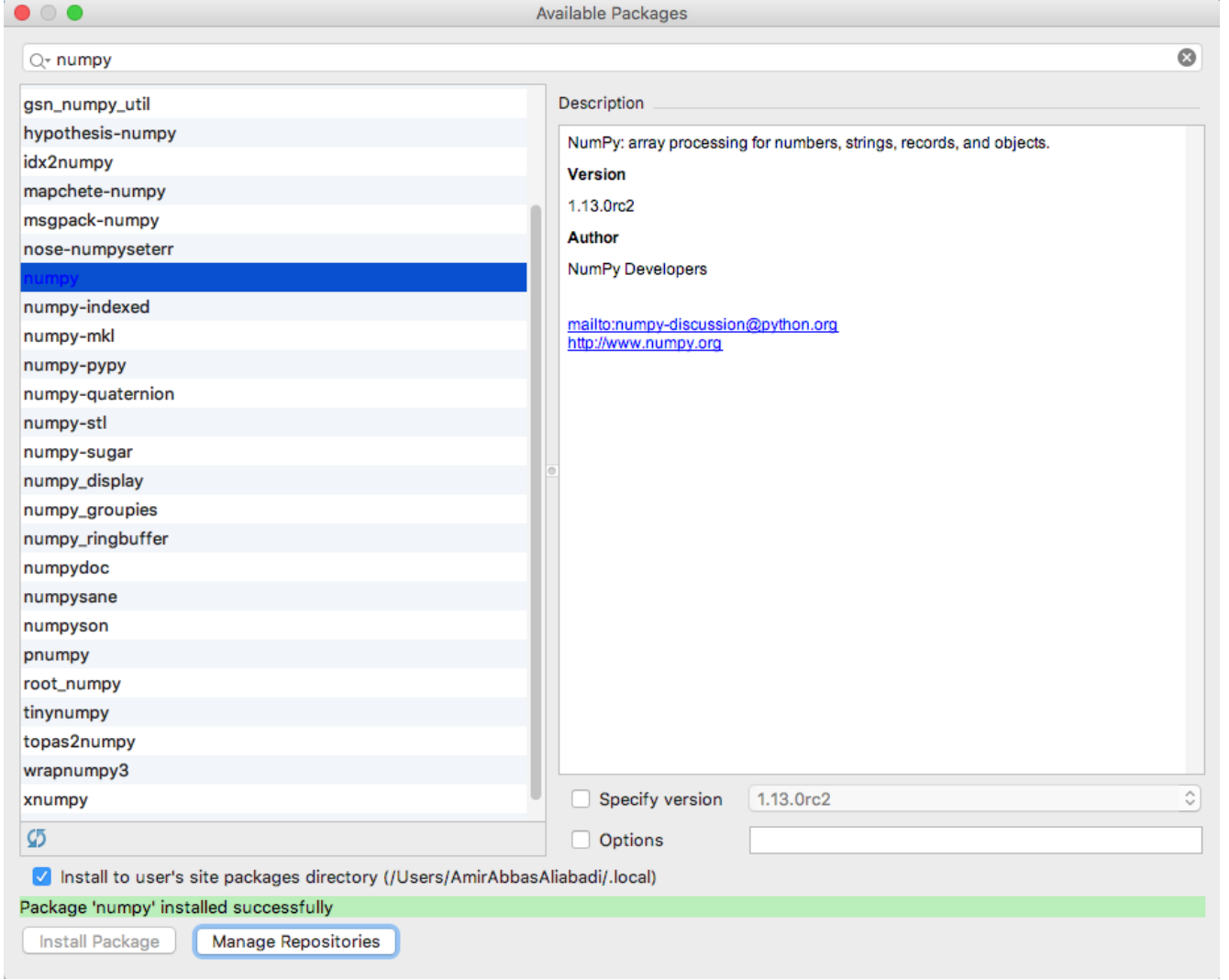


Figure 6: The numpy interpreter.

$$U = \langle U \rangle + u \quad (2)$$

$$V = \langle V \rangle + v \quad (3)$$

$$W = \langle W \rangle + w. \quad (4)$$

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

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

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.

```
#Introduction to python programming
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 [m2 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 [m2 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 [m2 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

```

Environmental Fluid Mechanics

One-point Turbulent Statistics

Amir A. Aliabadi

February 14, 2022

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,

$$U = \langle U \rangle + u \quad (1)$$

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

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

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,

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

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

$$\text{cov}(U, V) = \langle uv \rangle \quad (4)$$

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,

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

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

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

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

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 $\text{var}(T)$, $\text{cov}(W, T)$, $\text{cov}(U, T)$, $\text{cov}(U, W)$, 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 [Aliabadi et al., 2019, Aliabadi et al., 2021]. 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 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



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)

conducted at a sampling frequency of 4 Hz. Figure below shows the campaign site and the roof anemometer.

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

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 subtracted) 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.

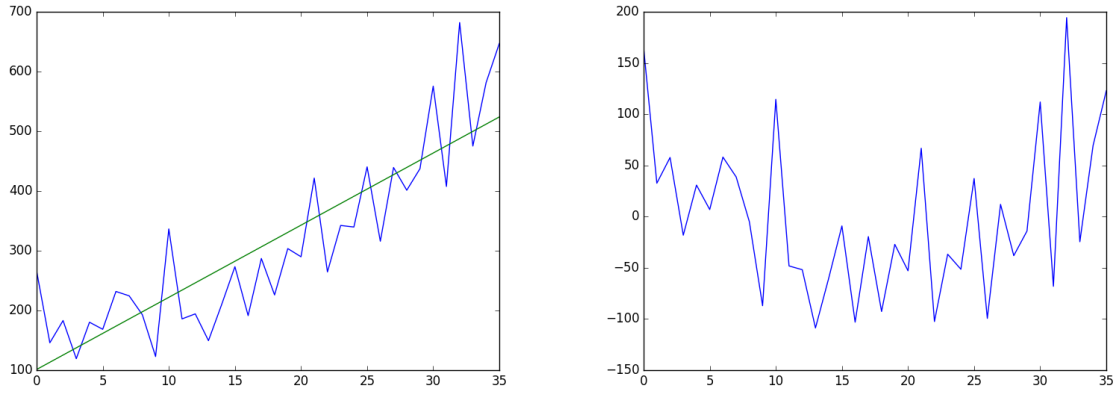


Figure 2: A trended time series (left) and a detrended time series (right); turbulent statistics must always be calculated after removing a trend from a time series, or else spurious statistics may be reported.

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 numpy
```

```
#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] = ...
    timeHavg[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().timestamp()`. 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 velocities, i.e. U_{avg} , V_{avg} , W_{avg} , or $(U_{avg}^2 + V_{avg}^2)^{0.5}$, 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?

References

- [Aliabadi et al., 2021] Aliabadi, A. A., Moradi, M., and Byerley, R. A. E. (2021). The budgets of turbulence kinetic energy and heat in the urban roughness sublayer. *Environmental Fluid Mechanics*, 21(4):843–884.
- [Aliabadi et al., 2019] Aliabadi, A. A., Moradi, M., Clement, D., Lubitz, W. D., and Gharabaghi, B. (2019). Flow and temperature dynamics in an urban canyon under a comprehensive set of wind directions, wind speeds, and thermal stability conditions. *Environ. Fluid Mech.*, 19(1):81–109.

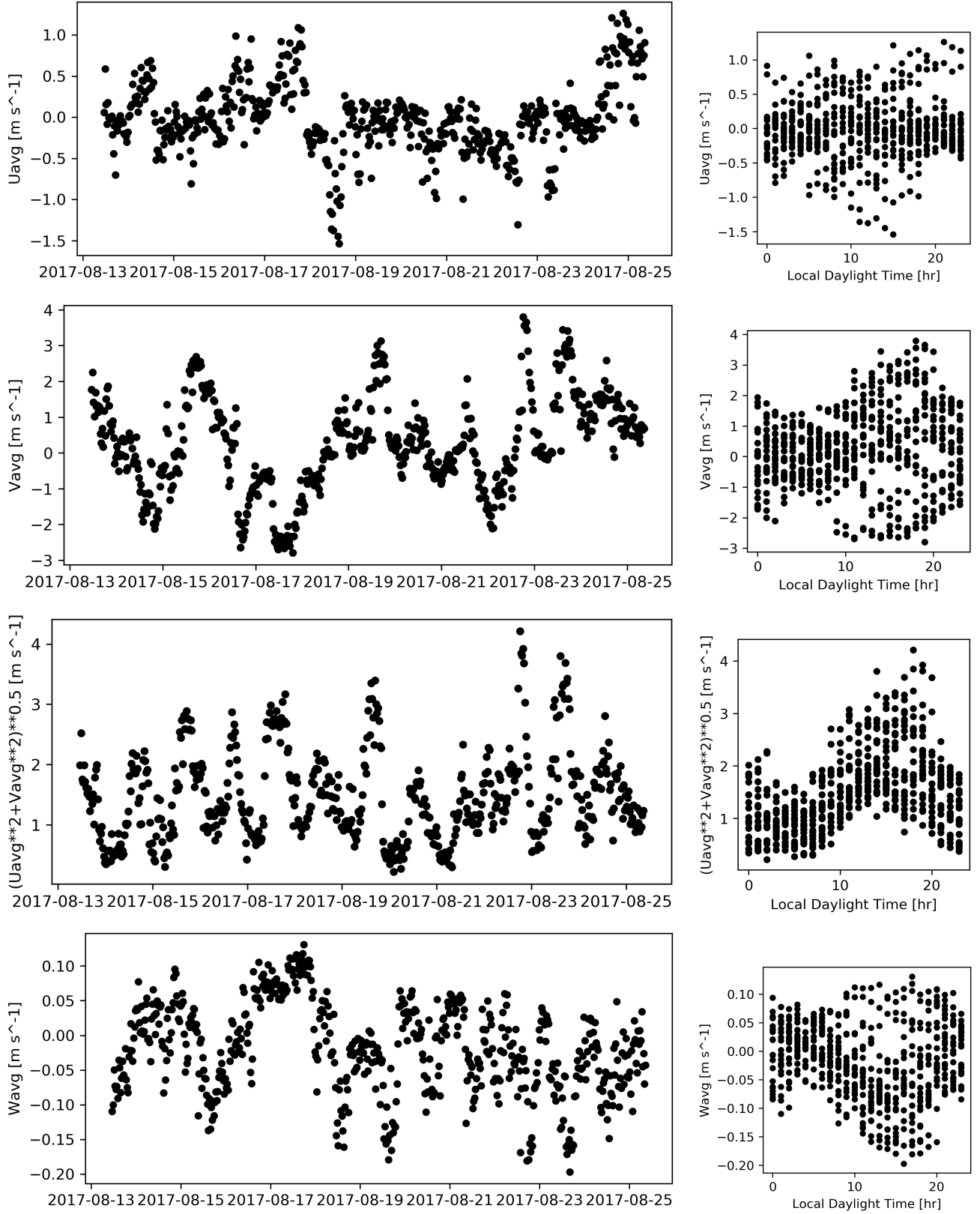


Figure 3: Mean velocities and mean wind speed in the horizontal direction.

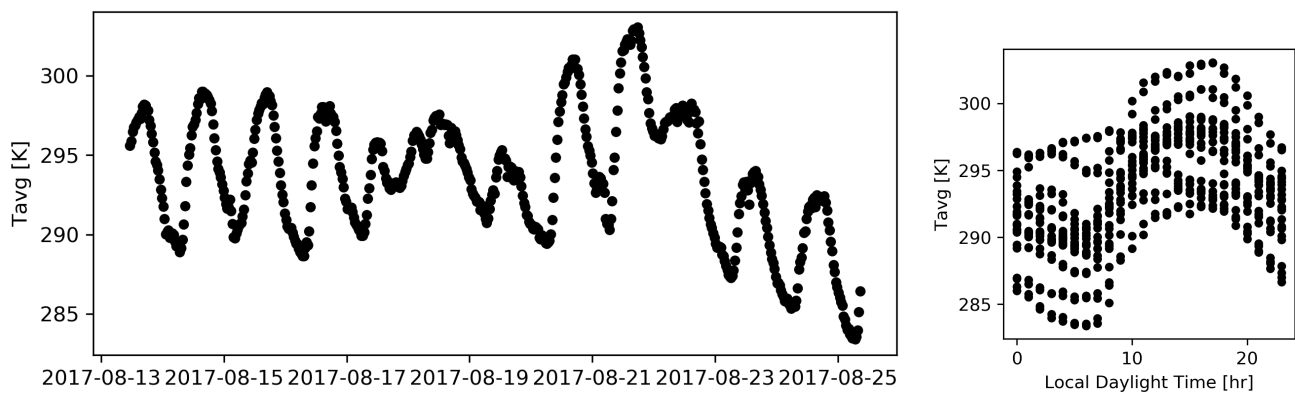


Figure 4: Mean temperature.

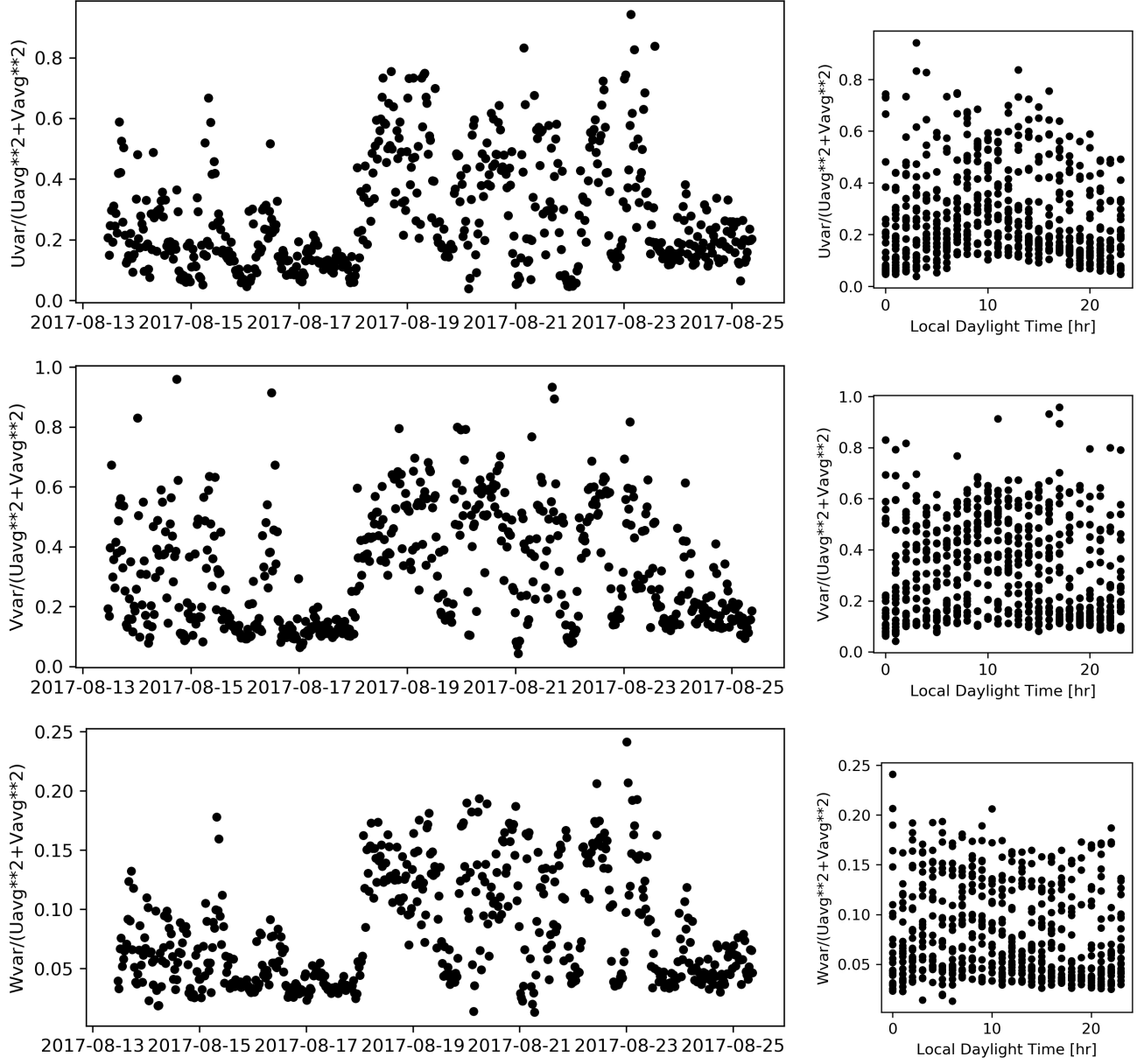


Figure 5: Normalized variances of velocity components.

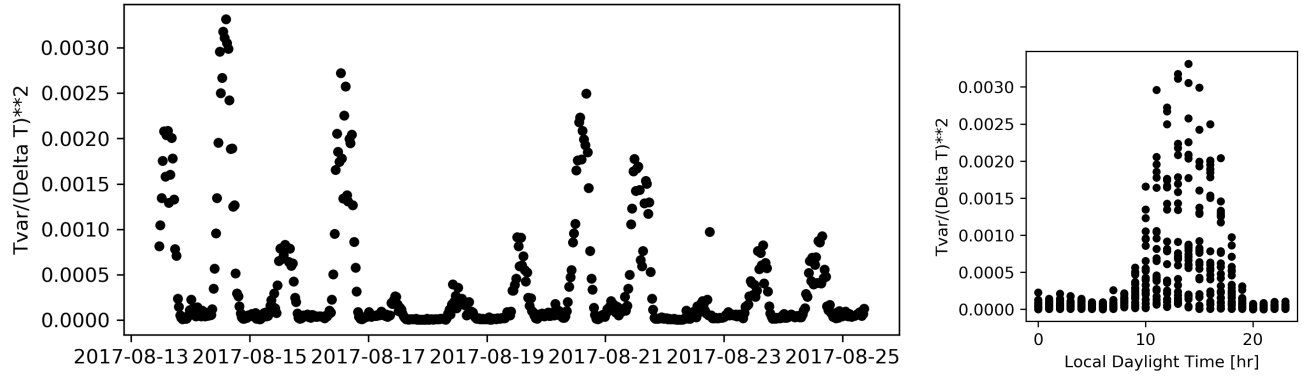


Figure 6: Normalized variance of temperature.

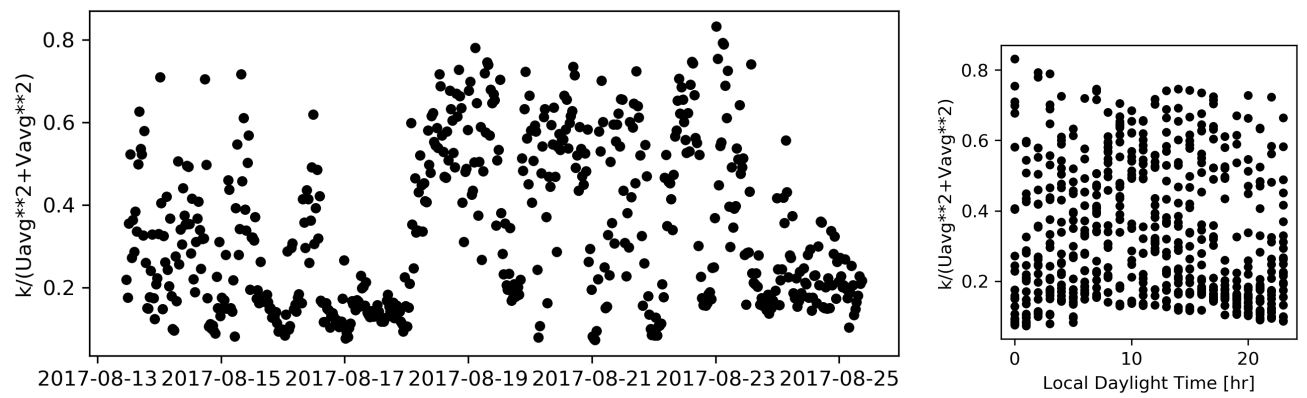


Figure 7: Normalized turbulent kinetic energy.

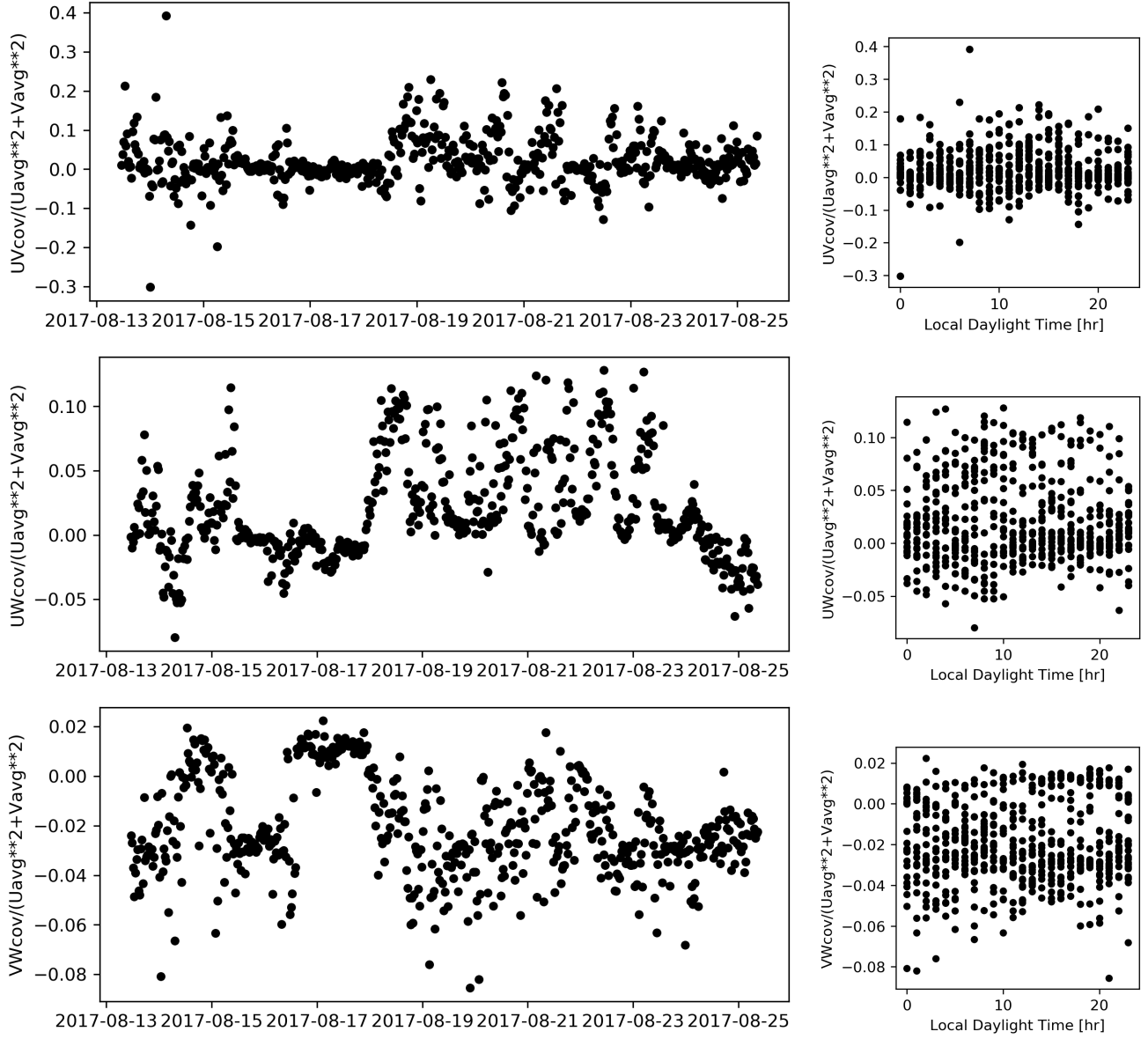


Figure 8: Normalized turbulent kinematic mass fluxes.

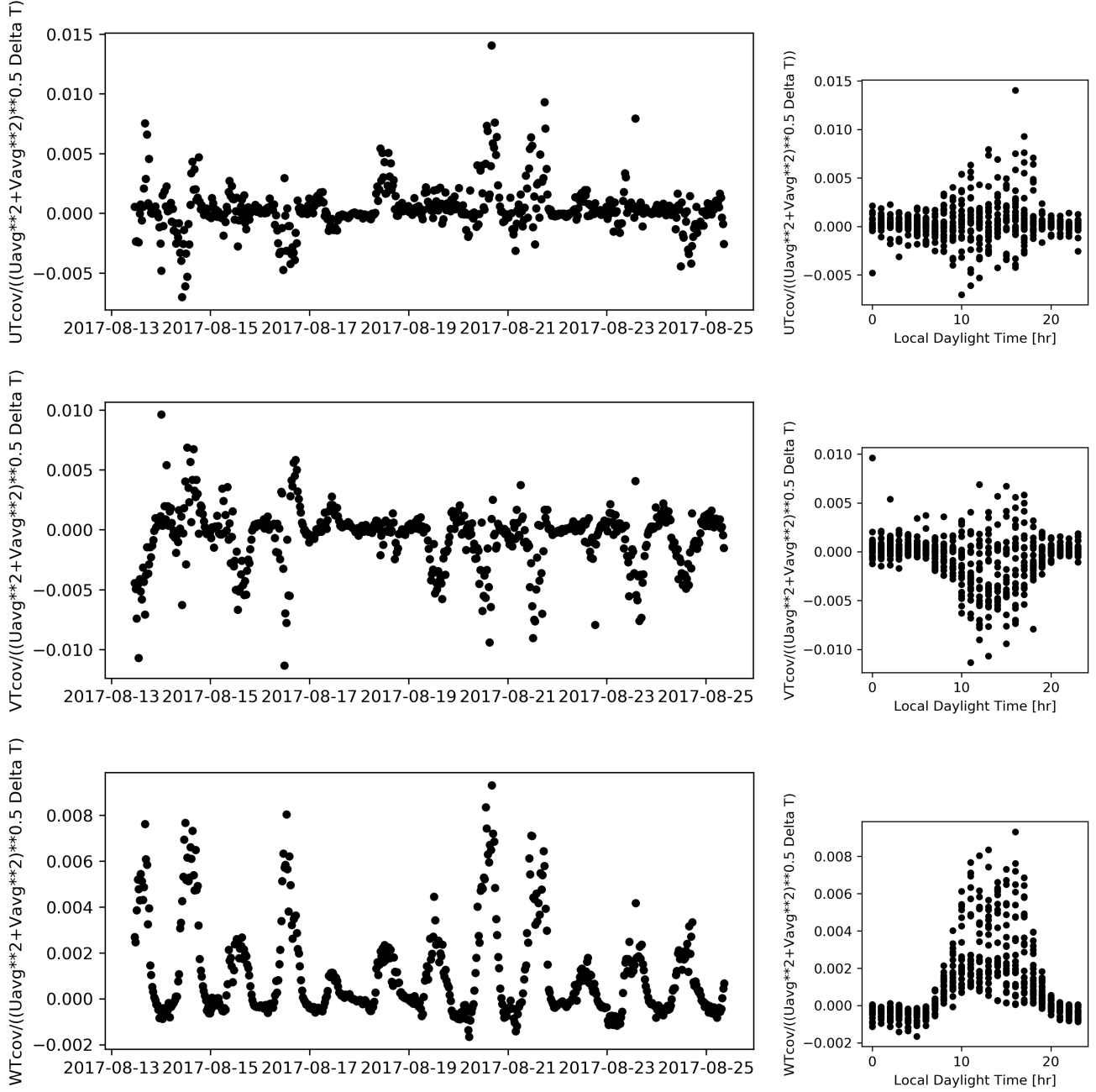


Figure 9: Normalized turbulent kinematic heat fluxes.

Environmental Fluid Mechanics

Discrete Fourier Transform Analysis in Time and Frequency Domains

Amir A. Aliabadi

January 31, 2022

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:

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

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

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

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

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

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

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

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.

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

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 [Aliabadi et al., 2016a, Aliabadi et al., 2016b]. Figure 1 shows the Polar 6 aircraft with a meteorological sensor installed under its wing to measure the aerodynamic and thermodynamic variables of the atmosphere.

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

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

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



Figure 1: The Polar 6 research aircraft with a meteorological sensor installed under its wing.

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.

#Discrete Fourier Tranform Analysis of wind velocity vector components measured using an a


```

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
fig = plt.figure(figsize=(7,5))
plt.plot(Lon,Lat)
plt.xlabel('Lon [deg]')
plt.ylabel('Lat [deg]')
plt.title('Aircraft Latitude versus Longitude')

```

```
plt.savefig('LatLon.png', bbox_inches='tight', dpi=300)
fig.show()
```

```
#Plot the aircraft altitude versus time
fig = plt.figure(figsize=(7,5))
plt.plot(t,Alt)
plt.xlabel('t [s]')
plt.ylabel('Alt [m]')
plt.title('Aircraft Altitude versus Time')
plt.savefig('Alt.png', bbox_inches='tight', dpi=300)
fig.show()
```

```
#Plot the frequency and energy spectra
fig = plt.figure(figsize=(7,5))
plt.plot(kappa,FU2,'ko')
plt.xlabel('kappa [m-1]')
plt.ylabel('FU2 [m2 s-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('FU2 Frequency Spectrum')
plt.savefig('FU2.png', bbox_inches='tight', dpi=300)
fig.show()
```

```
fig = plt.figure(figsize=(7,5))
plt.plot(kappa,EU,'ko')
plt.xlabel('kappa [m-1]')
plt.ylabel('EU [m2 s-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('EU Discrete Energy Spectrum')
plt.savefig('EU.png', bbox_inches='tight', dpi=300)
fig.show()
```

```
fig = plt.figure(figsize=(7,5))
plt.plot(kappa,FV2,'ro')
plt.xlabel('kappa [m-1]')
plt.ylabel('FV2 [m2 s-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('FV2 Frequency Spectrum')
plt.savefig('FV2.png', bbox_inches='tight', dpi=300)
fig.show()
```

```
fig = plt.figure(figsize=(7,5))
plt.plot(kappa,EV,'ro')
plt.xlabel('kappa [m-1]')
```

```

plt.ylabel('EV [m2 s-2']')
plt.xscale('log')
plt.yscale('log')
plt.title('EV Discrete Energy Spectrum')
plt.savefig('EV.png', bbox_inches='tight', dpi=300)
fig.show()

```

```

fig = plt.figure(figsize=(7,5))
plt.plot(kappa,FW2,'bo')
plt.xlabel('kappa [m-1']')
plt.ylabel('FW2 [m2 s-2']')
plt.xscale('log')
plt.yscale('log')
plt.title('FW2 Frequency Spectrum')
plt.savefig('FW2.png', bbox_inches='tight', dpi=300)
fig.show()

```

```

fig = plt.figure(figsize=(7,5))
plt.plot(kappa,EW,'bo')
plt.xlabel('kappa [m-1']')
plt.ylabel('EW [m2 s-2']')
plt.xscale('log')
plt.yscale('log')
plt.title('EW Discrete Energy Spectrum')
plt.savefig('EW.png', bbox_inches='tight', dpi=300)
fig.show()

```

```

#Plot the discrete energy spectrum for turbulent kinetic energy
fig = plt.figure(figsize=(7,5))
plt.plot(kappa,Ek,'go')
plt.xlabel('kappa [m-1']')
plt.ylabel('Ek [m2 s-2']')
plt.xscale('log')
plt.yscale('log')
plt.title('Ek Discrete Energy Spectrum')
plt.savefig('Ek.png', bbox_inches='tight', dpi=300)
fig.show()

```

```

#Plot absolute value of co-spectra and discrete energy co-spectra
fig = plt.figure(figsize=(7,5))
plt.plot(kappa,numpy.absolute(CoUV),'ko')
plt.xlabel('kappa [m-1']')
plt.ylabel('|CoUV| [m2 s-2']')
plt.xscale('log')
plt.yscale('log')
plt.title('CoUV Frequency Co-spectrum')

```

```
plt.savefig('CoUV.png', bbox_inches='tight', dpi=300)
fig.show()
```

```
fig = plt.figure(figsize=(7,5))
plt.plot(kappa,numpy.absolute(EUV),'ko')
plt.xlabel('kappa [m-1']')
plt.ylabel('|EUV| [m2 s-2']')
plt.xscale('log')
plt.yscale('log')
plt.title('EUV Discrete Energy Co-spectrum')
plt.savefig('EUV.png', bbox_inches='tight', dpi=300)
fig.show()
```

```
fig = plt.figure(figsize=(7,5))
plt.plot(kappa,numpy.absolute(CoUW),'ro')
plt.xlabel('kappa [m-1']')
plt.ylabel('|CoUW| [m2 s-2']')
plt.xscale('log')
plt.yscale('log')
plt.title('CoUW Frequency Co-spectrum')
plt.savefig('CoUW.png', bbox_inches='tight', dpi=300)
fig.show()
```

```
fig = plt.figure(figsize=(7,5))
plt.plot(kappa,numpy.absolute(EUW),'ro')
plt.xlabel('kappa [m-1']')
plt.ylabel('|EUW| [m2 s-2']')
plt.xscale('log')
plt.yscale('log')
plt.title('EUW Discrete Energy Co-spectrum')
plt.savefig('EUW.png', bbox_inches='tight', dpi=300)
fig.show()
```

```
fig = plt.figure(figsize=(7,5))
plt.plot(kappa,numpy.absolute(CoVW),'bo')
plt.xlabel('kappa [m-1']')
plt.ylabel('|CoVW| [m2 s-2']')
plt.xscale('log')
plt.yscale('log')
plt.title('CoVW Frequency Co-spectrum')
plt.savefig('CoVW.png', bbox_inches='tight', dpi=300)
fig.show()
```

```
fig = plt.figure(figsize=(7,5))
plt.plot(kappa,numpy.absolute(EVW),'bo')
plt.xlabel('kappa [m-1']')
```

```

plt.ylabel('|EVW| [m2 s-2]')
plt.xscale('log')
plt.yscale('log')
plt.title('EVW Discrete Energy Co-spectrum')
plt.savefig('EVW.png', bbox_inches='tight', dpi=300)
fig.show()

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.

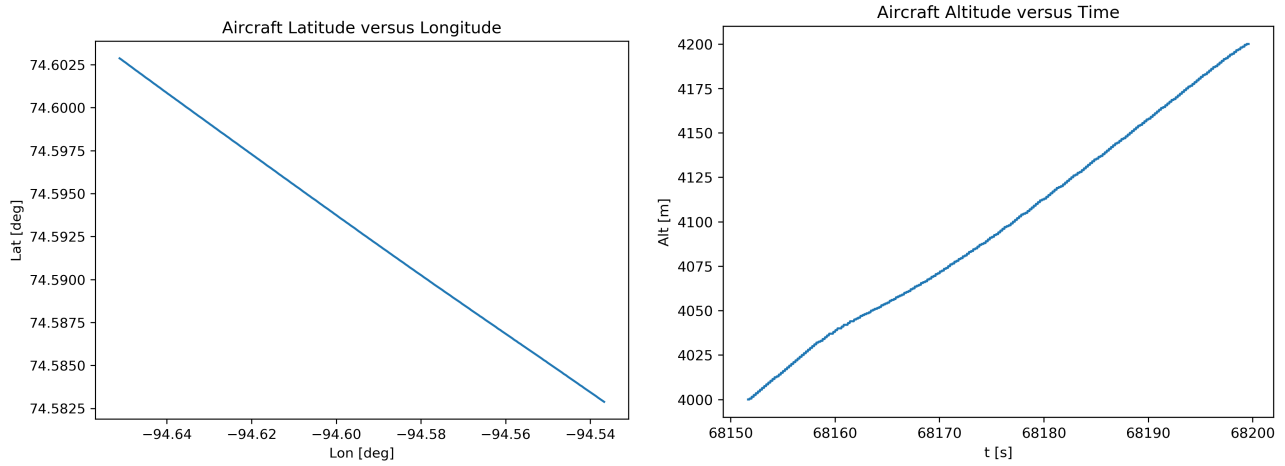


Figure 2: Plot of the aircraft movement; latitude vs. longitude (left); altitude vs. time (right).

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, -0.0082437328261500652],
[-0.11038926755049001, -0.0082437328261500652, 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 slope 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?

References

- [Aliabadi et al., 2016a] Aliabadi, A. A., Staebler, R. M., Liu, M., and Herber, A. (2016a). Characterization and parametrization of Reynolds stress and turbulent heat flux in the stably-stratified lower Arctic troposphere using aircraft measurements. *Bound.-Lay. Meteorol.*, 161(1):99–126.
- [Aliabadi et al., 2016b] Aliabadi, A. A., Thomas, J. L., Herber, A. B., Staebler, R. M., Leitch, W. R., Schulz, H., Law, K. S., Marelle, L., Burkart, J., Willis, M. D., Bozem, H., Hoor, P. M., Köllner, F., Schneider, J., Levasseur, M., and Abbatt, J. P. D. (2016b). Ship emissions measurement in the Arctic by plume intercepts of the Canadian Coast Guard icebreaker *Amundsen* from the *Polar 6* aircraft platform. *Atmos. Chem. Phys.*, 16(12):7899–7916.

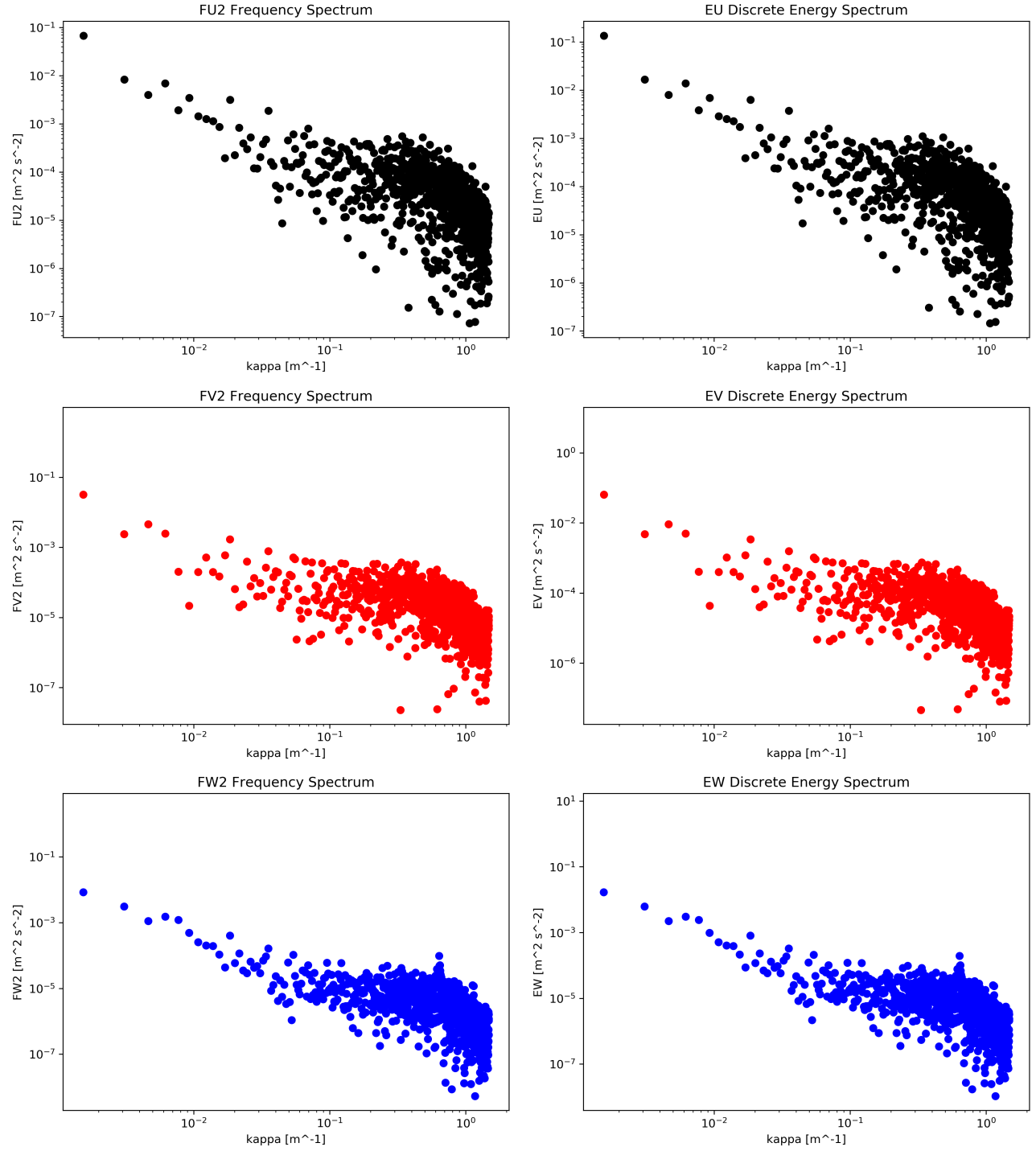


Figure 3: Plot of the discrete frequency and discrete spectral energy for U (top), V (middle), and W (bottom).

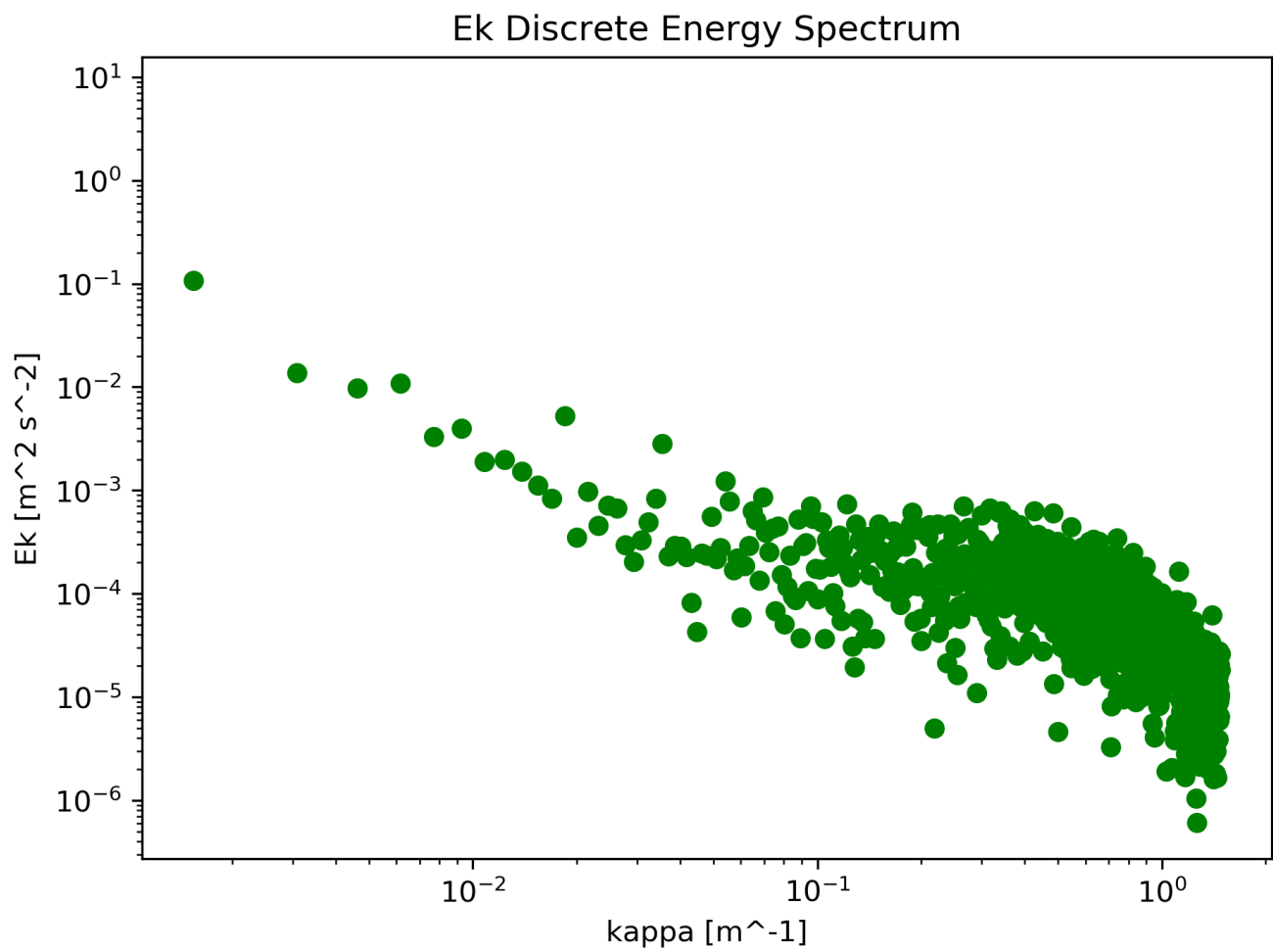


Figure 4: Plot of the discrete spectral energy for the turbulent kinetic energy.

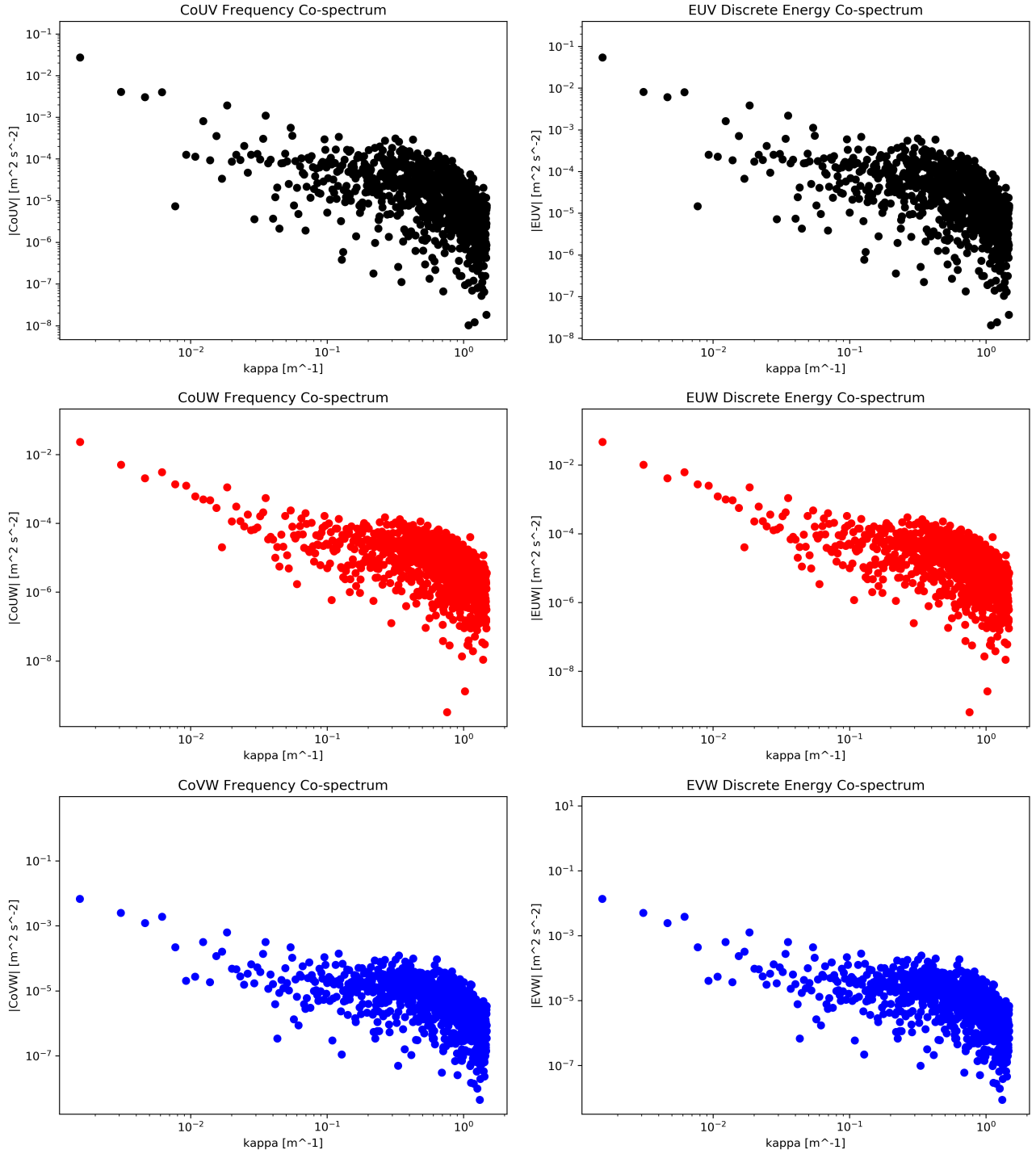


Figure 5: Plot of the absolute value of discrete frequency and discrete spectral energy for co-spectra *UV* (top), *UW* (middle), and *VW* (bottom).

Environmental Fluid Mechanics

Balloons

Amir A. Aliabadi

March 12, 2024

1 Introduction

In lectures we discuss the tethered balloon as an in-situ technique for measuring atmospheric variables. Figure 1 shows such a balloon that was deployed at the University of Guelph [Aliabadi et al., 2021] and at a mine field in northern Canada [Byerlay et al., 2020, Nambiar et al., 2020, Nahian et al., 2020]. This balloon is equipped with a mini weather station called TriSonica. It samples atmospheric wind velocity components (as measured according to the sensor's local coordinate system), heading, pitch, roll, and wind direction angles (also as measured according to the sensor's local coordinate system), at a sampling frequency of 10 Hz. The goal of this computer assignment is to transform the wind velocity vector components from the sensor's local coordinate system to a fixed inertial reference frame.

Since balloons typically rotate about three axes once levitated in air, their measurements of wind velocity vector components should be referenced to a frame with fixed coordinate axes directions along east, north, and normal to the earth surface using rotation matrices. This is made possible by use of the heading, roll, and pitch angles measured by sensors on board. Suppose that the sensor has a coordinate system in which the $+x$ axis is from local north N_T of the sensor to local south S_T , the $+y$ axis is from local east E_T to local west W_T , and the $+z$ axis is downward, i.e. toward the earth surface. This local coordinate system is *right handed*. Likewise, the velocities measured by the sensor are positive along these axes. Let the sensor's coordinate system be shown by x_T , y_T , and z_T and the corresponding velocities be shown by U_T , V_T , and W_T [ms^{-1}]. The sensor measures heading h [$^\circ$], positive clockwise with respect to magnetic north N_M . It measures the pitch angle p [$^\circ$], which is a positive downward rotation of x_T about y_T , and the roll angle r [$^\circ$], which is a positive downward rotation of y_T about x_T . The goal is to transform this coordinate system, by means of rotation matrices, to align with a reference frame of the earth with x_F pointing from west to east, y_F pointing from south to north, and z_F pointing away upward from the surface of the earth. This coordinate system is also *right handed*. The resulting velocity transformation will provide U_F , V_F , and W_F [ms^{-1}] in the final coordinate system.

As depicted in Figure 2, the first rotation should be about the local y_T axis by an angle $\gamma = p$ [$^\circ$]. This transformation results in an intermediate coordinate system x_1 , y_1 , and z_1 so that the x_1 axis will be aligned with the horizon, i.e. parallel to the earth surface. Note that the figure is

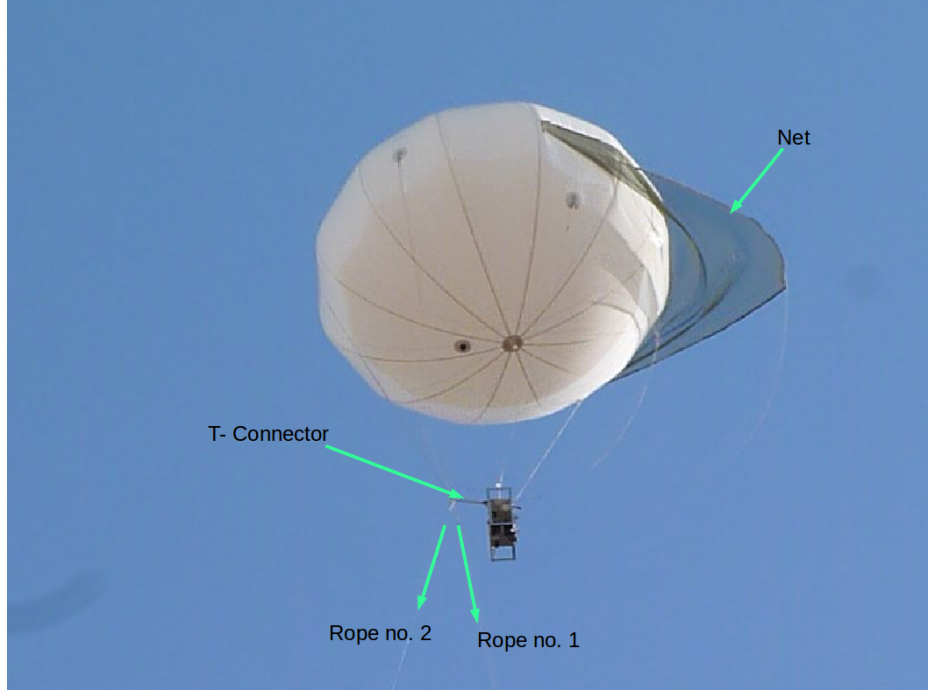


Figure 1: A tethered balloon system for atmospheric measurements controlled using two ropes.

viewed normal to $y_T = y_1$ and that in this coordinate system z_1 is still not yet normal to the earth surface and that y_1 is still not yet aligned with the horizon. This transformation is given by

$$\begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} = \begin{bmatrix} \cos(\gamma) & 0 & \sin(\gamma) \\ 0 & 1 & 0 \\ -\sin(\gamma) & 0 & \cos(\gamma) \end{bmatrix} \begin{bmatrix} x_T \\ y_T \\ z_T \end{bmatrix} = R_{y,\gamma} \begin{bmatrix} x_T \\ y_T \\ z_T \end{bmatrix}. \quad (1)$$

As depicted in Figure 3, the second rotation should be about x_1 axis by an angle $\eta = r$ [°]. This transformation results in another intermediate coordinate system x_2 , y_2 , and z_2 so that now the y_2 axis will be aligned with the horizon, i.e. parallel to the earth surface. Note that the figure is viewed normal to $x_1 = x_2$ and that in this coordinate system z_2 is normal to the earth surface. This transformation is given by

$$\begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(\eta) & -\sin(\eta) \\ 0 & \sin(\eta) & \cos(\eta) \end{bmatrix} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix} = R_{x,\eta} \begin{bmatrix} x_1 \\ y_1 \\ z_1 \end{bmatrix}. \quad (2)$$

As depicted in Figure 4, the third rotation should be about z_2 axis by an angle $\alpha = (\delta + h + 90) \% 360$ [°], where δ [°] is the magnetic declination of the earth, which is dependent on a specific latitude and longitude. For the current site $\delta = 13.5^\circ$. Here the modulus with 360 is taken since the heading angle can vary from 0 to 360°. This transformation results in another intermediate coordinate system x_3 , y_3 , and z_3 so that now the y_3 axis will be aligned from north to south and the x_3 axis will be aligned from west to east. Note that the figure is viewed normal to $z_2 = z_3$. This

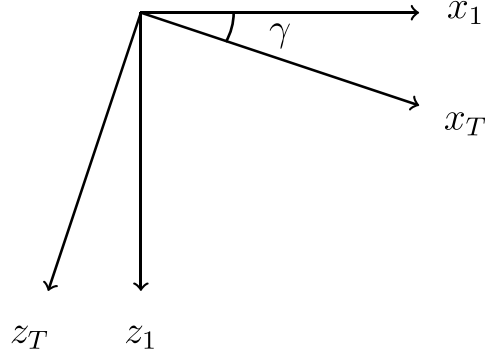


Figure 2: Rotation about y_T by $\gamma = p$ [$^\circ$]; figure viewed normal to the $y_T = y_1$ axis

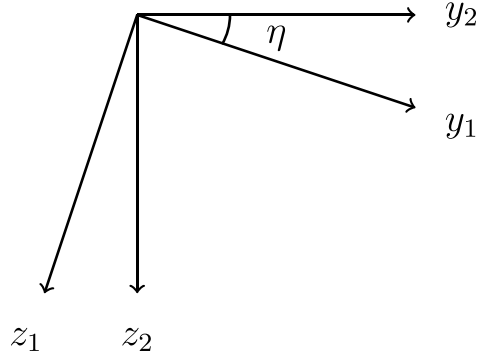


Figure 3: Rotation about x_1 by $\eta = r$ [$^\circ$]; figure viewed normal to the $x_1 = x_2$ axis

transformation is given by

$$\begin{bmatrix} x_3 \\ y_3 \\ z_3 \end{bmatrix} = \begin{bmatrix} \cos(\alpha) & -\sin(\alpha) & 0 \\ \sin(\alpha) & \cos(\alpha) & 0 \\ 0 & 0 & 1 \end{bmatrix} \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix} = R_{z,\alpha} \begin{bmatrix} x_2 \\ y_2 \\ z_2 \end{bmatrix}. \quad (3)$$

As depicted in Figure 5, the fourth and final rotation should be about x_3 axis by an angle 180° . This transformation results in the final coordinate system with x_F , y_F , and z_F axes, which point west-to-east, south-to-north, and normal upward from the earth surface, respectively. This transformation is given by

$$\begin{bmatrix} x_F \\ y_F \\ z_F \end{bmatrix} = \begin{bmatrix} 1 & 0 & 0 \\ 0 & \cos(180^\circ) & -\sin(180^\circ) \\ 0 & \sin(180^\circ) & \cos(180^\circ) \end{bmatrix} \begin{bmatrix} x_3 \\ y_3 \\ z_3 \end{bmatrix} = R_{x,180^\circ} \begin{bmatrix} x_3 \\ y_3 \\ z_3 \end{bmatrix}. \quad (4)$$

In compressed form, the entire coordinate transformation can be shown as follows. This immediately implies a similar transformation for the measured velocities by the sensor.

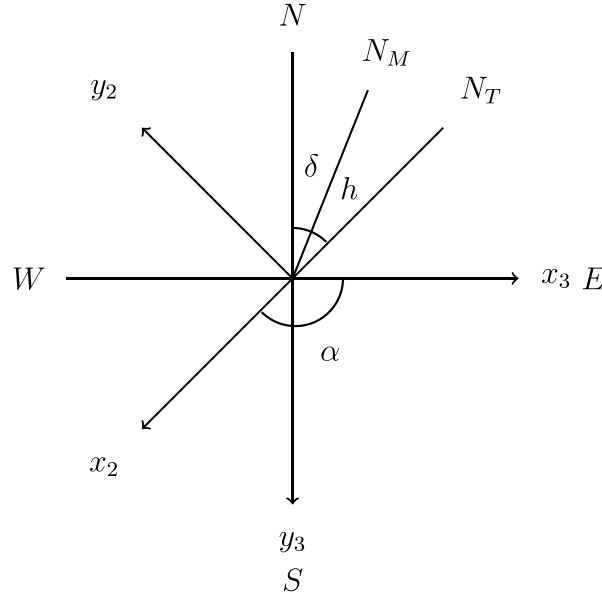


Figure 4: Rotation about z_2 by $\alpha = (\delta + h + 90)\%360$ [$^\circ$]; figure viewed normal to the $z_2 = z_3$ axis

$$\begin{bmatrix} x_F \\ y_F \\ z_F \end{bmatrix} = R_{x,180^\circ} R_{z,\alpha} R_{x,\eta} R_{y,\gamma} \begin{bmatrix} x_T \\ y_T \\ z_T \end{bmatrix}, \quad (5)$$

$$\begin{bmatrix} U_F \\ V_F \\ W_F \end{bmatrix} = R_{x,180^\circ} R_{z,\alpha} R_{x,\eta} R_{y,\gamma} \begin{bmatrix} U_T \\ V_T \\ W_T \end{bmatrix}. \quad (6)$$

2 Python Script

The script begins by defining the magnetic declination angle and the input file name. This file is associated with one among many of the balloon launches in the mine field. The magnetic declination angle could vary from one location of the earth to another. It is also known to vary over long periods of time. Rumor has it that in prehistoric times the magnetic north and south poles were swapped!

```
#Implement rotation matrices to convert TriSonica's measurements from
# TriSonica's coordinate system to a fixed inertial coordinate system
```

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

```
#Magnetic declination [degrees]
delta = 13.5
```

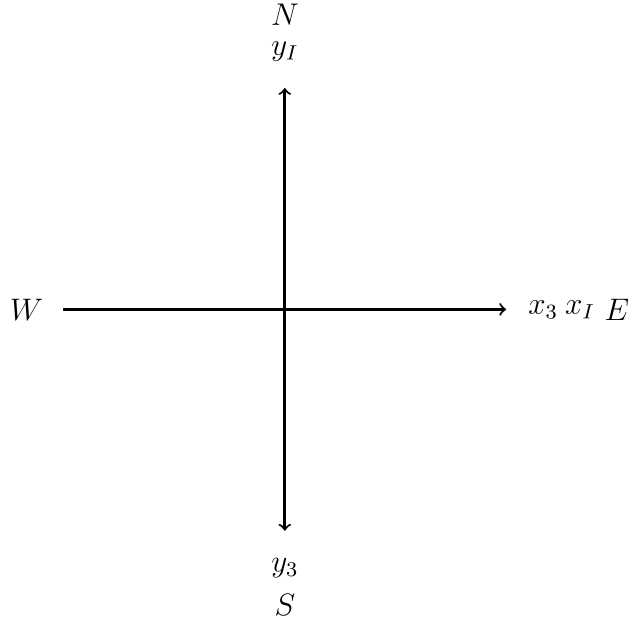



Figure 5: Rotation about x_3 by 180° ; figure viewed normal to the z_3 axis

```
#Define file name
```

```
fileNameData = "Balloon-Data-Calibrated.txt"
```

In the next step we read the wind direction angle, calibrated U, V, and W components of the wind velocity vector, roll angle, and pitch angle, all measured with respect to the sensor's local coordinate system. Note that many columns of data are ignored for our analysis. Also note that the vertical wind velocity vector is multiplied by minus one to ensure that the positive vertical velocity points down toward the earth surface.

```
#Load all data; note: skip the U and V columns
```

```
data = np.loadtxt(fileNameData)
```

```
#Wind direction in local TriSonica's coordinate system, clockwise from local North
```

```
WD = data[:,8]
```

```
#TriSonica U [m s-1] assume local North to South positive
```

```
UCal = data[:,10]
```

```
#TriSonica V [m s-1] assume local East to West positive
```

```
VCal = data[:,12]
```

```
#TroSonica W [m s-1] MUST BE positive downward, so multiply by -1
```

```
WCal = -1*data[:,14]
```

```
Roll = data[:,19]
```

```
Pitch = data[:,20]
```

```
#Magnetic heading positive clockwise from local North
```

```
Heading = data[:,21]
```

```
N = np.size(WD)
```

Next we correct the heading angle with the magnetic declination angle. We also convert all angles from degrees to radians.

```
#Rotation angle about local z axis to align local x close to inertial x
alpha = (np.pi/180)*(((delta + Heading + 90) % 360))

#Rotation angle about local y axis to align local x,z close to inertial x,z
gamma = (np.pi/180)*Pitch

#Rotation angle about local x axis to align local y,z close to inertial y,z
eta = (np.pi/180)*Roll
```

In the next step, we define the rotation matrices. It is good practice to use extensive commenting to prevent mistakes in defining the rotation matrices.

```
#Rzalpha: Rotation matrix to rotate about local z axis to align local x close to inertial x
# [cos alpha   -sin alpha   0]
# [sin alpha    cos alpha   0]
# [0            0           1]

Rzalpha11 = np.cos(alpha)
Rzalpha12 = -np.sin(alpha)
Rzalpha13 = 0

Rzalpha21 = np.sin(alpha)
Rzalpha22 = np.cos(alpha)
Rzalpha23 = 0

Rzalpha31 = 0
Rzalpha32 = 0
Rzalpha33 = 1

#Rxpi: Rotation matrix to rotate about local x axis to align local z close to inertial z
# [1      0      0      ]
# [0      cos pi  -sin pi]
# [0      sin pi   cos pi ]

Rxpi11 = ...
Rxpi12 = ...
Rxpi13 = ...

Rxpi21 = ...
Rxpi22 = ...
Rxpi23 = ...

Rxpi31 = ...
```

```

Rxpi32 = ...
Rxpi33 = ...

#Rygamma: Rotation matrix to rotate about local y axis to align local x,z close to inertial
# [cos gamma    0      sin gamma]
# [0           1      0          ]
# [-sin gamma   0      cos gamma]

Rygamma11 = ...
Rygamma12 = ...
Rygamma13 = ...

Rygamma21 = ...
Rygamma22 = ...
Rygamma23 = ...

Rygamma31 = -np.sin(gamma)
Rygamma32 = 0
Rygamma33 = np.cos(np.pi)

#Rxeta: Rotation matrix to rotate about local x axis to align local y,z close to inertial
# [1      0      0      ]
# [0      cos eta  -sin eta]
# [0      sin eta  cos eta ]

Rxeta11 = ...
Rxeta12 = ...
Rxeta13 = ...

Rxeta21 = ...
Rxeta22 = ...
Rxeta23 = ...

Rxeta31 = ...
Rxeta32 = ...
Rxeta33 = ...

```

Once the rotation matrices is defined, it is time to perform the transformations one by one. This is achieved via the following code.

```

# Perform rotation 1
U1 = Rygamma11 * UCal + Rygamma12 * VCal + Rygamma13 * WCal
V1 = Rygamma21 * UCal + Rygamma22 * VCal + Rygamma23 * WCal
W1 = Rygamma31 * UCal + Rygamma32 * VCal + Rygamma33 * WCal

# Perform rotation 2

```

```

U2 = Rxeta11 * U1 + Rxeta12 * V1 + Rxeta13 * W1
V2 = Rxeta21 * U1 + Rxeta22 * V1 + Rxeta23 * W1
W2 = Rxeta31 * U1 + Rxeta32 * V1 + Rxeta33 * W1

```

```

# Perform rotation 3

```

```

U3 = Ralpha11 * U2 + Ralpha12 * V2 + Ralpha13 * W2
V3 = Ralpha21 * U2 + Ralpha22 * V2 + Ralpha23 * W2
W3 = Ralpha31 * U2 + Ralpha32 * V2 + Ralpha33 * W2

```

```

# Perform rotation 4

```

```

# Finally, after these 4 rotations, we have the velocities in the inertial reference frame

```

```

U4 = Rxi11 * U3 + Rxi12 * V3 + Rxi13 * W3
V4 = Rxi21 * U3 + Rxi22 * V3 + Rxi23 * W3
W4 = Rxi31 * U3 + Rxi32 * V3 + Rxi33 * W3

```

In the final step, we perform the plotting to visualize the angles and wind velocity vector components. Note that the horizontal axis in all the plots is simply the sample number.

```

fig = plt.figure(figsize=(10,2))
plt.plot(Heading,'ko',markersize=1,markeredgecolor='k',markerfacecolor='k')
plt.axhline(y=0, color='k', linestyle='-')
plt.xlabel('Data Point',fontsize=12)
plt.ylabel('Heading [deg]',fontsize=6)
plt.savefig('HeadingAngle.pdf', bbox_inches='tight', dpi=600)
fig.show()

```

```

fig = plt.figure(figsize=(10,2))
plt.plot(Pitch,'ko',markersize=1,markeredgecolor='k',markerfacecolor='k')
plt.axhline(y=0, color='k', linestyle='-')
plt.xlabel('Data Point',fontsize=12)
plt.ylabel('Pitch [deg]',fontsize=6)
plt.savefig('PitchAngle.pdf', bbox_inches='tight', dpi=600)
fig.show()

```

```

fig = plt.figure(figsize=(10,2))
plt.plot(Roll,'ko',markersize=1,markeredgecolor='k',markerfacecolor='k')
plt.axhline(y=0, color='k', linestyle='-')
plt.xlabel('Data Point',fontsize=12)
plt.ylabel('Roll [deg]',fontsize=6)
plt.savefig('RollAngle.pdf', bbox_inches='tight', dpi=600)
fig.show()

```

```

fig = plt.figure(figsize=(10,2))
plt.plot(WD,'ko',markersize=1,markeredgecolor='k',markerfacecolor='k')
plt.axhline(y=0, color='k', linestyle='-')
plt.xlabel('Data Point',fontsize=12)

```

```

plt.ylabel('WD TriSonica [deg]',fontsize=6)
plt.savefig('WindDirectionAngleSensor.pdf', bbox_inches='tight', dpi=600)
fig.show()

fig = plt.figure(figsize=(10,2))
plt.plot(UCal,'ko',markersize=1,markeredgecolor='k',markerfacecolor='k',label='UCal')
plt.plot(VCal,'bo',markersize=1,markeredgecolor='b',markerfacecolor='b',label='Vcal')
plt.axhline(y=0, color='k', linestyle='-')
plt.legend(loc='upper center')
plt.xlabel('Data Point',fontsize=12)
plt.ylabel('UCal, VCal [m s-1]',fontsize=6)
plt.savefig('UCalVCalSensor.pdf', bbox_inches='tight', dpi=600)
plt.ylim((-10,10))
fig.show()

fig = plt.figure(figsize=(10,2))
plt.plot(U1,'ko',markersize=1,markeredgecolor='k',markerfacecolor='k',label='U1')
plt.plot(V1,'bo',markersize=1,markeredgecolor='b',markerfacecolor='b',label='V1')
plt.axhline(y=0, color='k', linestyle='-')
plt.legend(loc='upper center')
plt.xlabel('Data Point',fontsize=12)
plt.ylabel('U1, V1 [m s-1]',fontsize=6)
plt.savefig('U1V1.pdf', bbox_inches='tight', dpi=600)
plt.ylim((-10,10))
fig.show()

...

fig = plt.figure(figsize=(10,2))
plt.plot(U4,'ko',markersize=1,markeredgecolor='k',markerfacecolor='k',label='U4')
plt.plot(V4,'bo',markersize=1,markeredgecolor='b',markerfacecolor='b',label='V4')
plt.axhline(y=0, color='k', linestyle='-')
plt.legend(loc='upper center')
plt.xlabel('Data Point',fontsize=12)
plt.ylabel('U4, V4 [m s-1]',fontsize=6)
plt.savefig('U4V4FixedInertialFrame.pdf', bbox_inches='tight', dpi=600)
plt.ylim((-10,10))
fig.show()

fig = plt.figure(figsize=(10,2))
plt.plot(WCal,'ko',markersize=1,markeredgecolor='k',markerfacecolor='k',label='WCal')
plt.plot(W1,'bo',markersize=1,markeredgecolor='b',markerfacecolor='b',label='W1')
plt.plot(W2,'ro',markersize=1,markeredgecolor='r',markerfacecolor='r',label='W2')
plt.plot(W3,'yo',markersize=1,markeredgecolor='y',markerfacecolor='y',label='W3')
plt.plot(W4,'go',markersize=1,markeredgecolor='g',markerfacecolor='g',label='W4')
plt.axhline(y=0, color='k', linestyle='-')

```

```
plt.legend(loc='upper center')
plt.xlabel('Data Point',fontsize=12)
plt.ylabel('WCal,1,2,3,4 [m s-1]',fontsize=6)
plt.savefig('VerticalWind.pdf', bbox_inches='tight', dpi=600)
plt.ylim((-5,5))
fig.show()

plt.show()
```

Complete the above lines of code. After successfully running the script, Figures 6, 7, and 8 should be obtained. Try to answer the following questions.

- The pitch and roll angles are generally close to zero. What does this indicate with respect to the balloon's stability in wind?
- The heading angle of the balloon indicates that the sensor is oriented toward the south and oscillates from south west to south east with respect to the magnetic north. Meanwhile, considering the horizontal wind velocity vector component measurements with respect to the sensor's local coordinate system, the magnitude of UCal is large and its sign is positive, while VCal is close to zero. What does this indicate with respect to the actual wind direction with respect to the fixed inertial coordinate system?
- Given you answer to the previous item, what should you expect to get for horizontal velocity vector components U4 and V4 with respect to the fixed inertial coordinate system. Do the results meet your expectation?
- With regard to the vertical velocity vector component, explain the main important feature in the results.

References

- [Aliabadi et al., 2021] Aliabadi, A. A., Moradi, M., and Byerlay, R. A. E. (2021). The budgets of turbulence kinetic energy and heat in the urban roughness sublayer. *Environmental Fluid Mechanics*, 21(4):843–884.
- [Byerlay et al., 2020] Byerlay, R. A. E., Nambiar, M. K., Nazem, A., Nahian, M. R., Biglarbegian, M., and Aliabadi, A. A. (2020). Measurement of land surface temperature from oblique angle airborne thermal camera observations. *Int. J. Remote Sens.*, 41(8):3119–3146.
- [Nahian et al., 2020] Nahian, M. R., Nazem, A., Nambiar, M. K., Byerlay, R., Mahmud, S., Seguin, A. M., Robe, F. R., Ravenhill, J., and Aliabadi, A. A. (2020). Complex meteorology over a complex mining facility: Assessment of topography, land use, and grid spacing modifications in WRF. *Journal of Applied Meteorology and Climatology*, 59(4):769–789.
- [Nambiar et al., 2020] Nambiar, M. K., Byerlay, R. A. E., Nazem, A., Nahian, M. R., Moradi, M., and Aliabadi, A. A. (2020). A Tethered Air Blimp (TAB) for observing the microclimate over a complex terrain. *Geosci. Instrum. Meth. Data Syst.*, 9(1):193–211.

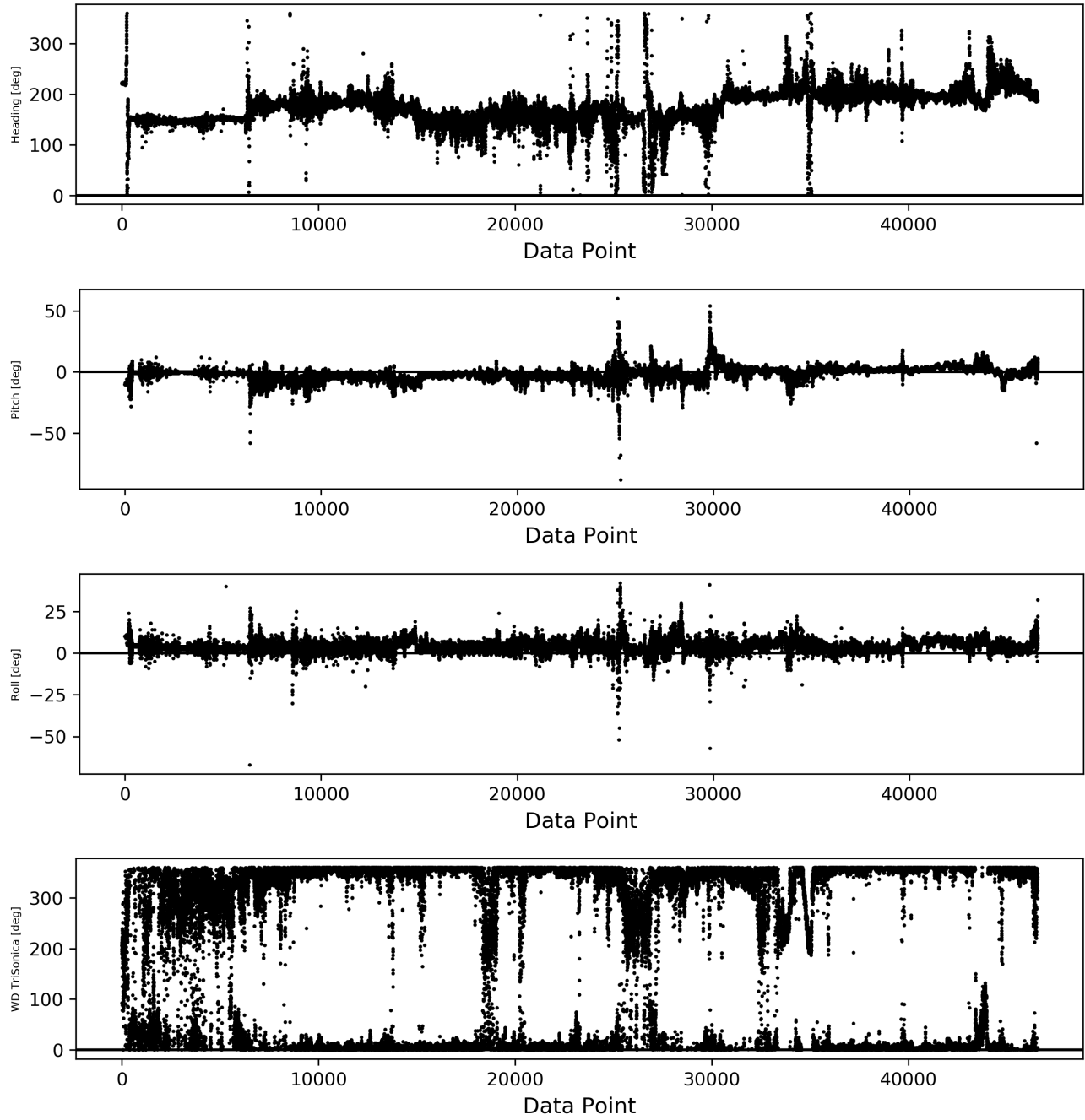


Figure 6: Heading, Pitch, Roll, and Wind Direction Angles as measured by the meteorological sensor on the balloon according to the local coordinate system of the sensor.

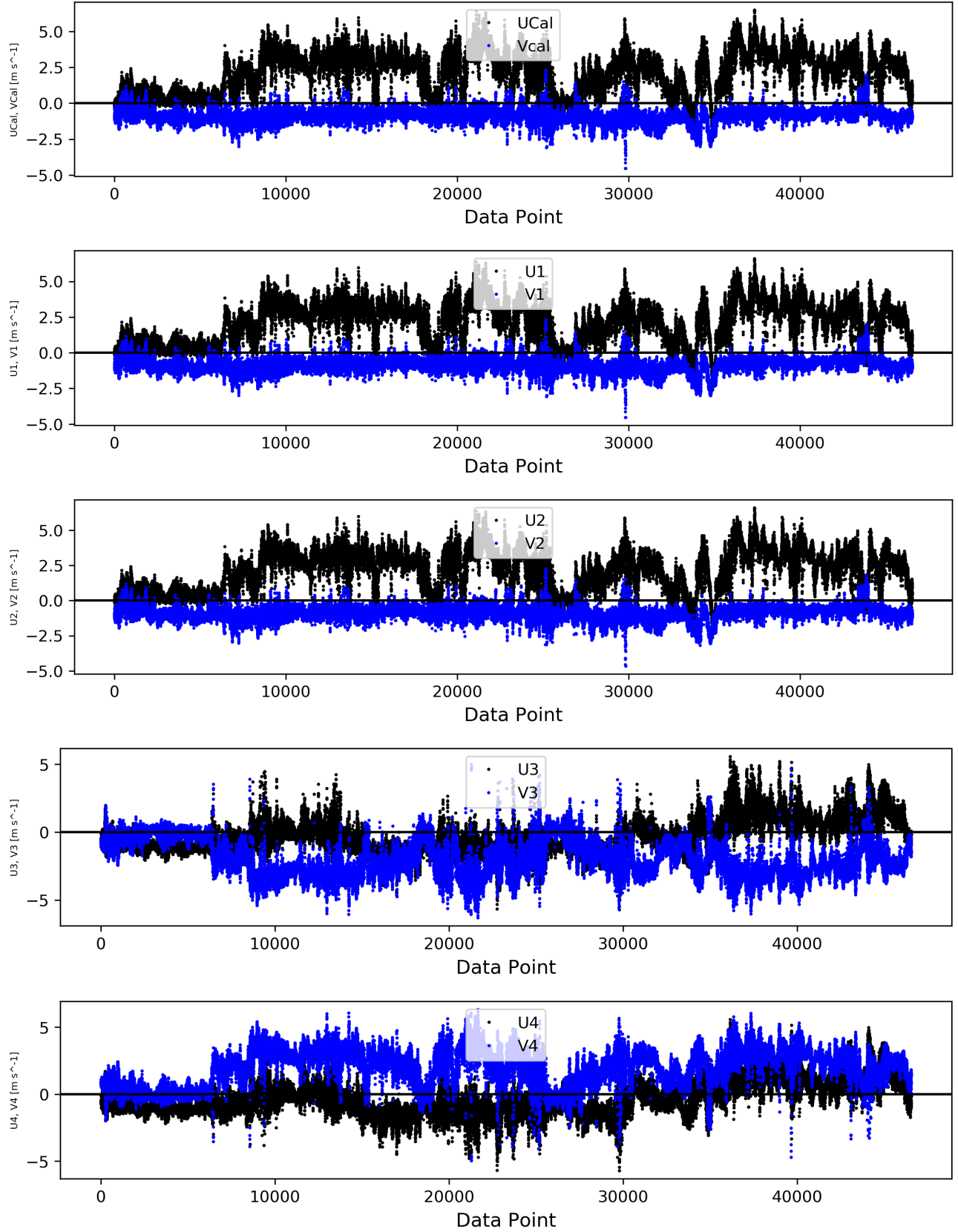


Figure 7: Horizontal velocity vector components of wind as measured according to the local coordinate system of the sensor, throughout the coordinate transformation process, and as transformed according to the fixed inertial frame.

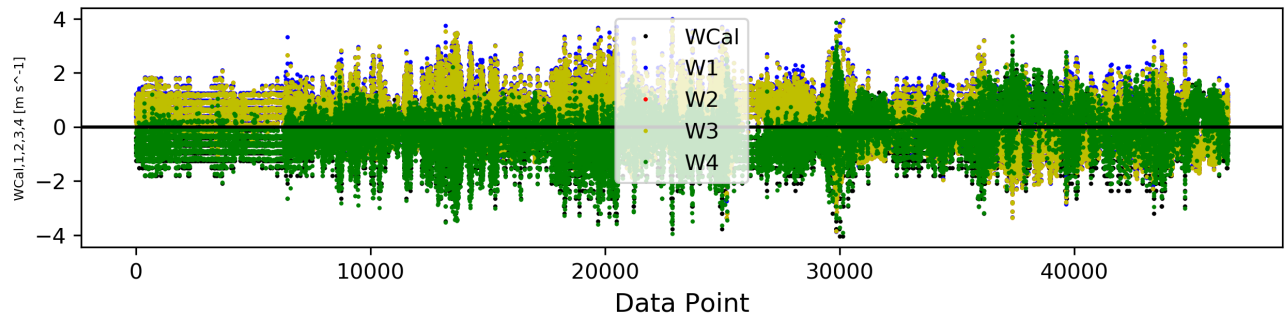


Figure 8: Vertical velocity vector component of wind as measured according to the local coordinate system of the sensor, throughout the coordinate transformation process, and as transformed according to the fixed inertial frame.

Environmental Fluid Mechanics

SOnic Detection And Ranging (SODAR)

Amir A. Aliabadi

January 31, 2022

1 Introduction

In lectures we discuss the SOnic Detection And Ranging (SODAR) as a class of atmospheric measurement techniques that rely on the transmission and echo of sonic sound waves to measure wind speed and direction in the atmosphere. Figure 1 shows a mini Sodar device that probes atmospheric winds from 20 to 200 m elevation at a height resolution of up to 10 m. This Sodar instrument was installed in the Turfgrass Institute, Guelph, in the summer of 2018 [Aliabadi et al., 2021] and at a mine field in northern Canada in summer and fall of 2019 [Kia et al., 2021] for wind profile measurements. It sampled one atmospheric profile every half hour. In this computer assignment a selected part of the data set is used to classify wind speeds at selected altitudes of 40, 60, 80, 100, and 120 m for selected diurnal times of 0000, 0600, 1200, and 1800 Eastern Standard Time (EST).

2 Python Script

Five text files are provided which contain the Sodar data. These are "Sodarh40.txt", "Sodarh60.txt", "Sodarh80.txt", "Sodarh100.txt", and "Sodarh120.txt". The most important entries from these text files are the hour of measurement and the measured wind speed. The python script begins by reading the five input text files.

```
#Import mini SODAR data and plot wind profiles classified according to diurnal time and he
import numpy
import matplotlib.pyplot as plt

inputFileNameSodarh40="Sodarh40.txt"
inputFileNameSodarh60="Sodarh60.txt"
inputFileNameSodarh80="Sodarh80.txt"
inputFileNameSodarh100="Sodarh100.txt"
inputFileNameSodarh120="Sodarh120.txt"
```

The next step is to load the data into vectors. It is convenient to store each of the hours and



Figure 1: A mini SODAR deployed at the Turfgrass Insititute, Guelph, in the summer of 2018 for wind speed and direction profile measurements from 20 to 200 m elevation.

elevations of measurements in a separate vector. After loading the data, we can print the number of data points in each vector to make sure they are of the same size.

```
#Load data into vectors
```

```
dataSodarh40 = numpy.loadtxt(inputFileNameSodarh40)
HourSodarh40=dataSodarh40[:,3]
SSodarh40=dataSodarh40[:,6]
NSodarh40 = numpy.size(HourSodarh40)
```

```
dataSodarh60 = numpy.loadtxt(inputFileNameSodarh60)
HourSodarh60=dataSodarh60[:,3]
SSodarh60=dataSodarh60[:,6]
NSodarh60 = numpy.size(HourSodarh60)
```

```
...
```

```
print('The size of datasets: ', NSodarh40, NSodarh60, NSodarh80, NSodarh100, NSodarh120)
```

Next, we define new vectors into which we classify measurements that occurred in a specific hour and elevation. We should initialize the vectors with the value of Not a Number (nan) or `numpy.nan`. This is required because the Sodar device can occasionally return nan if it cannot analyze the echo of the transmitted sound signal.

```

#Define classified vectors
#For four diurnal times: 0000, 0600, 1200, and 1800 Eastern Standard Time (EST)
#For five elevations: 40m, 60m, 80m, 100m, and 120m

#Make vectors to contain wind speed
#Initialize values to Not A Number (nan)
SSodarh40t0000=numpy.zeros((NSodarh40))
SSodarh40t0000[:]=numpy.nan
SSodarh60t0000=numpy.zeros((NSodarh40))
SSodarh60t0000[:]=numpy.nan
SSodarh80t0000=numpy.zeros((NSodarh40))
SSodarh80t0000[:]=numpy.nan
SSodarh100t0000=numpy.zeros((NSodarh40))
SSodarh100t0000[:]=numpy.nan
SSodarh120t0000=numpy.zeros((NSodarh40))
SSodarh120t0000[:]=numpy.nan

SSodarh40t0600=numpy.zeros((NSodarh40))
SSodarh40t0600[:]=numpy.nan
SSodarh60t0600=numpy.zeros((NSodarh40))
SSodarh60t0600[:]=numpy.nan
SSodarh80t0600=numpy.zeros((NSodarh40))
SSodarh80t0600[:]=numpy.nan
SSodarh100t0600=numpy.zeros((NSodarh40))
SSodarh100t0600[:]=numpy.nan
SSodarh120t0600=numpy.zeros((NSodarh40))
SSodarh120t0600[:]=numpy.nan

...

```

Next we employ a series of conditional statements in a loop to assign measurements associated with a particular hour to the classifying vector. Note that after this assignment, many entries of the classifying vectors are still nan, but that is fine.

```

#Classify data based on diurnal time and elevation
for n in range(0, NSodarh40):
    if (HourSodarh40[n] == 0):
        SSodarh40t0000[n] = SSodarh40[n]
        SSodarh60t0000[n] = SSodarh60[n]
        SSodarh80t0000[n] = SSodarh80[n]
        SSodarh100t0000[n] = SSodarh100[n]
        SSodarh120t0000[n] = SSodarh120[n]
    elif (HourSodarh40[n] == 6):
        SSodarh40t0600[n] = SSodarh40[n]
        SSodarh60t0600[n] = SSodarh60[n]
        SSodarh80t0600[n] = SSodarh80[n]

```

```

SSodarh100t0600[n] = SSodarh100[n]
SSodarh120t0600[n] = SSodarh120[n]
elif (HourSodarh40[n] == 12):
...
elif (HourSodarh40[n] == 18):
...

```

Finally, we can plot our results using the `plt.boxplot()` command. Before employing this command, the data can be reduced by removing the nan values. This can be accomplished in a very compact coding syntax shown below.

```

#Plot results

fig = plt.figure(figsize=(6,6))
SSodar0000All = [SSodarh40t0000[~numpy.isnan(SSodarh40t0000)],
SSodarh60t0000[~numpy.isnan(SSodarh60t0000)],
SSodarh80t0000[~numpy.isnan(SSodarh80t0000)],
SSodarh100t0000[~numpy.isnan(SSodarh100t0000)],
SSodarh120t0000[~numpy.isnan(SSodarh120t0000)]]
plt.boxplot(SSodar0000All, 0, '', 0)
ax = fig.add_subplot(111)
ax.set_yticklabels(['40', '60', '80', '100', '120'])
ax.tick_params(axis='both', which='major', labelsize=13)
plt.ylabel('z [m]',fontsize=20)
plt.xlabel('S [m s-1]',fontsize=20)
plt.xlim((0,17))
plt.tight_layout()
plt.savefig('SSodart0000.pdf', bbox_inches='tight', dpi=600)
fig.show()

...

plt.show()

```

Complete the above lines of code. After successfully running the script, Figure 2 should be obtained. Try to answer the following questions.

- Open each of the text files `Sodarh40.txt` and `Sodarh120.txt` and briefly examine their contents. Which file contains more nan values? why?
- Of the wind speed profile plots shown, which elevation shows greater diurnal variability in wind speed?
- Of the wind speed profile plots shown, at which diurnal time is the profile of the wind speed more uniformly distributed with altitude?

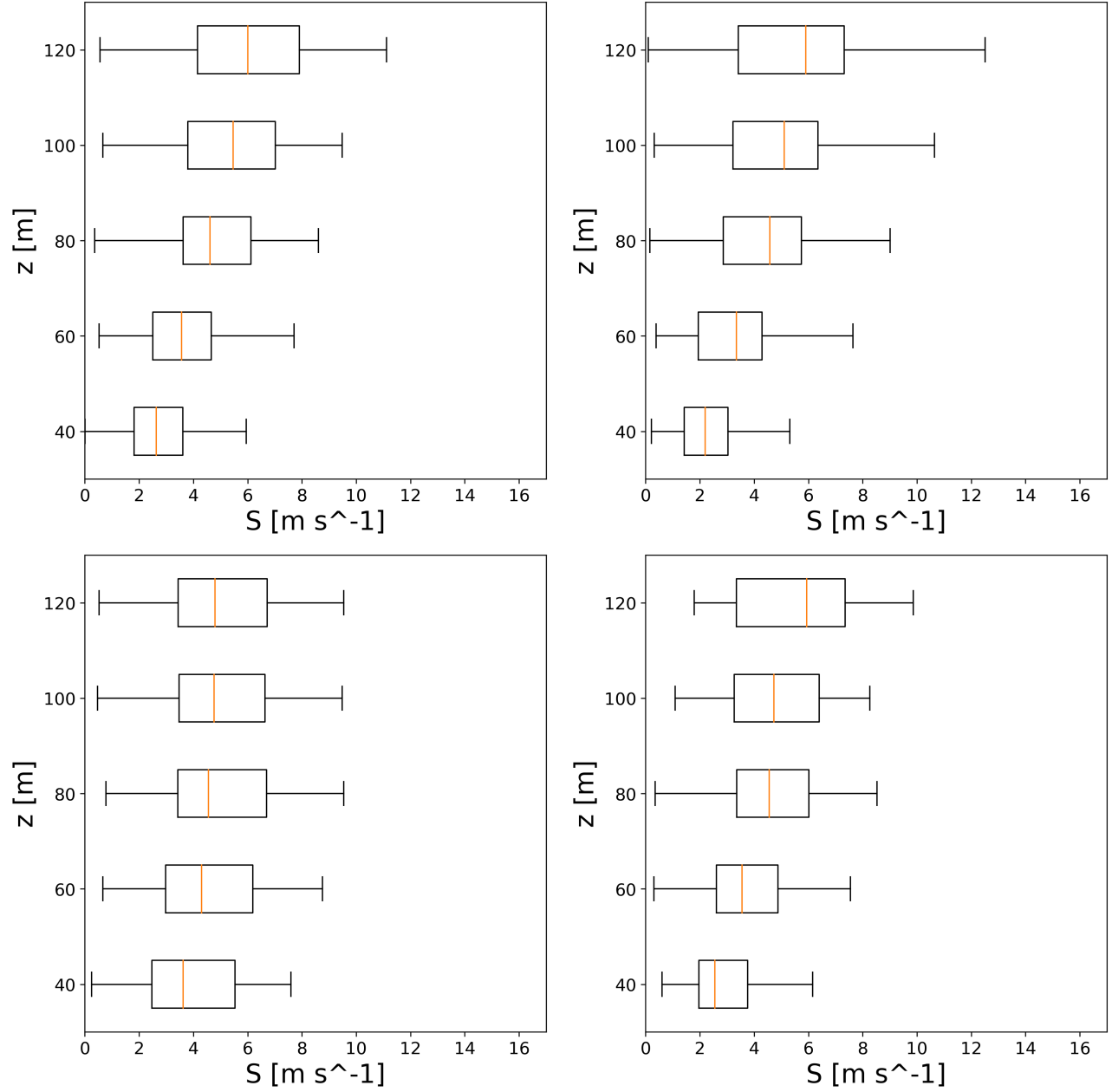


Figure 2: Box plots showing the 5th, 25th, 50th, 75th, and 95th percentiles of wind speed at five altitudes (40, 60, 80, 100, and 120 m); data are classified for four diurnal times 0000 (top left), 0600 (top right), 1200 (bottom left), and 1800 (bottom right), Eastern Standard Time (EST).

References

- [Aliabadi et al., 2021] Aliabadi, A. A., Moradi, M., and Byerlay, R. A. E. (2021). The budgets of turbulence kinetic energy and heat in the urban roughness sublayer. *Environmental Fluid Mechanics*, 21(4):843–884.
- [Kia et al., 2021] Kia, S., Flesch, T. K., Freeman, B. S., and Aliabadi, A. A. (2021). Atmospheric transport over open-pit mines: The effects of thermal stability and mine depth. *J. Wind. Eng. Ind. Aerodyn.*, 214:104677.