Project Based Engineering Instrumentation With CircuitPython

A Brief Textbook Presented to the
Student Body of the University of South Alabama

Last Update: June 28, 2022

Manuscript Changes

1. Original tutorials in Google Docs created
2. Tutorials moved to LaTeX on this Github
3. December 21st, 2021 - Updated links for manuscript and hardware
4. Tutorials purchased by Tangibles that Teach and moved to url `https://tangibles-that-teach.gitbook.io/instrumentation-lab-manual/-MbMx70LQzRmEG_hS7Ld/`
5. May 30th, 2022 - Tangibles that teach went out of business and chapter began the move to Github
6. June 28th, 2022 - Work began on a Chromebook. Unfortunately the Figures folder is not back up on Git. As such a main_latest.pdf has been created that's the latest full version. The main.pdf is the version created by the Chromebook so it has new chapters but none of the older chapters. Figure files are now backup on Git but only figures from the 'Voltage Potentiometer' are currently there.

Changes Needed

1. All chapters from TTT need to be moved here
2. More theory is needed in this book or direction to further reading for the students
3. The new kit has a CPB - Might need to add a chapter explaining the difference and maybe even having the students getting the bluetooth to work.
4. The photo of the button lab could be better.
5. The circuit photo for the LSM6DS33 is not correct
6. Equations on thermistor need to be expanded
7. Equations relating voltage from protocol to Lux needs to be included
8. Example plots for light, sound, acceleration, etc needs to be expanded
9. Pendulum lab must be done in one of two ways. Either the pendulum is attached to a potentiometer or the CPX is mounted to the end of a string and data logged on board the CPX itself. The potentiometer is nice because you can record data with your laptop but the string idea is cool because you use the accelerometer. In either case you can make some really long pendulums
10. 3D printing a disc with holes on the outside to eventually mount to a shaft would be a really cool angular velocity sensor lab. Tangibles that teach could easily include a 3D printed disc that can mount to a pencil for ease of rotation. Could also include the CAD drawings so students can print more or even edit the design for better or worse performance.
11. Buying some load cells with the HMC converter and including them in the kit would add a whole lot different labs
12. Buying some magnetometers to measure magnetic field and do some sensor fusion would be neat. Could do roll, pitch and yaw calibration if we included a magnetometer.
13. Right now a lab just on roll and pitch estimation would be possible. Pendulum lab pretty much introduces them to this but could easily do an rc aircraft lab where they build an aircraft out of foam with an elevator and aileron so that the servos responds to roll and pitch change. There is a lab right now with just pitch but perhaps we could add roll to it.
14. Another cool project idea would be for the students to take temperature and light data on a cloudy day. Then have them infer if the amount of sunlight affected the temperature of the thermistor. They could plot the data with light on the x-axis and temperature on the y-axis and draw conclusions based on the plot they generate.
15. Could also have them take temperature and light data over the course of a whole day to plot sunrise and sunset and watch the ambient temperature rise and fall
16. For the aliasing lab, have the students sample as fast as possible and obtain the natural frequency of the system. Then have them sample at 1.0, 2.0 and 3.0 times the natural frequency they obtained. I originally picked 1,10 and 100 as arbitrary sampling frequencies and it would have been better to do 2,4 and 6.
17. On cool lab would be to take light data during sunset and watch light and temperature plummet.
18. Add this lab on frequency for notes

Acknowledgements

The author, Dr. Carlos Montalvo would like to acknowledge a few key members who made this textbook possible. First and foremost I would like to thank Adafruit for their entire ecosystem of electronics, tutorials, blogs and forums. Much of what I have learned here to teach Instrumentation was from Adafruit and the Adafruit Learn system and specifically people like Lady Ada and John Park who have helped shape CircuitPython and the Circuit-Playground Express to what it is today. I would also like to thank Dr. Saami Yazdani for creating the blueprint for Instrumentation at my university by creating a laboratory environment for an otherwise totally theoretical course. His course was the foundation for this textbook and for that I thank him for showing the way. I'd like to also thank and acknowledge Tangibles that Teach for giving me the opportunity to morph this loose set of projects into a textbook that can be used for multiple universities and classrooms and of course help students learn and acquire knowledge through creating.

About this textbook

This textbook has been designed with the student and faculty member in mind. First, this textbook goes hand in hand with Engineering Instrumentation taught at the undergraduate level at many universities. The course begins with simple plotting and moves into data analysis, calibration and more complex instrumentation techniques such as active filtering and aliasing. This course is designed to get students away from their pen and paper and build something that blinks and moves as well as learn to process real data that they themselves acquire. There is no theory in these projects. It is all applied using the project based learning method. Students will be tasked with downloading code, building circuitry, taking data all from the ground up. By the end of this course students will be well versed in the desktop version of Python while also the variant CircuitPython designed specifically for microelectronics from Adafruit. After this course students will be able to understand Instrumentation at the fundamental level as well as generate code that can be used in future projects and research to take and analyze data. Python is such a broad and useful language that it will be very beneficial for any undergraduate student to learn this language. To the professors using this textbook, 1 credit hour labs are often hard to work into a curriculum and "live" demonstrations in the classroom cost time and money that take away from other faculty duties. I've created this kit and textbook to be completely stand-alone. Students simply need to purchase the required materials and follow along with the lessons. These lessons can be picked apart and taught sequentially or individually on a schedule suited to the learning speed of the course. I hope whomever reads and learns from this textbook will walk away with an excitement to tinker, code and build future projects using microelectronics and programming.

Contents

1 Measuring Voltage Across a Potentiometer

1.1 Parts List

1. Laptop
2. CPX/CPB
3. USB Cable
4. Potentiometer
5. Resistor (the Ohms depends on how large your potentiometer is)
6. Breadboard

1.2 Learning Objectives

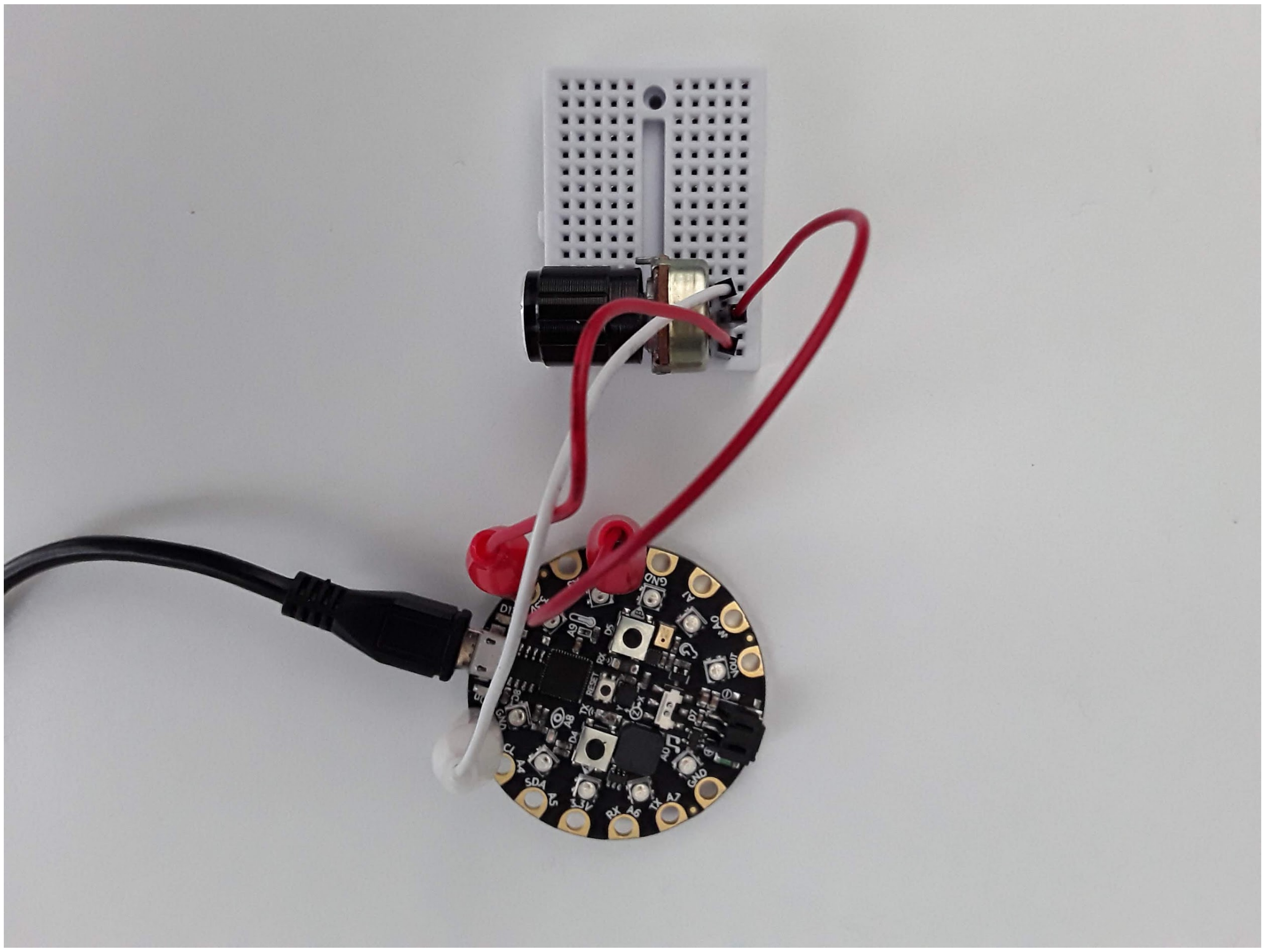
1. Understand voltage division of resistors in series
2. Measure an analog signal on the CircuitPlayground
3. Understand the binary measurement done by the analog to digital conversion (ADC)

1.3 Getting Started

At this point you've learned about analog to digital converters (ADC). It turns out that the CPX has 8 analog ports hooked up to a 3.3V logic 16 bit ADC. The input range on the ADC is 0 to 3.3V and the output range is 0 to 65536 which is 2^{16} hence 16 bits. In order to get accustomed to the ADC on the CPX, we're going to do a simple example where we measure the voltage drop across a potentiometer. You can read about potentiometers online if you wish. Basically though, a potentiometer is a variable resistance resistor that changes resistance by turning a knob. The knob changes the connection point of a wire and thus the length of the wire. This in turn changes the resistance. Potentiometers come in all shapes and sizes. Here are some examples.

Here's my circuit all hooked up. Two legs are connected to 3.3V and GND while the middle leg of the potentiometer is connected to pin A2.

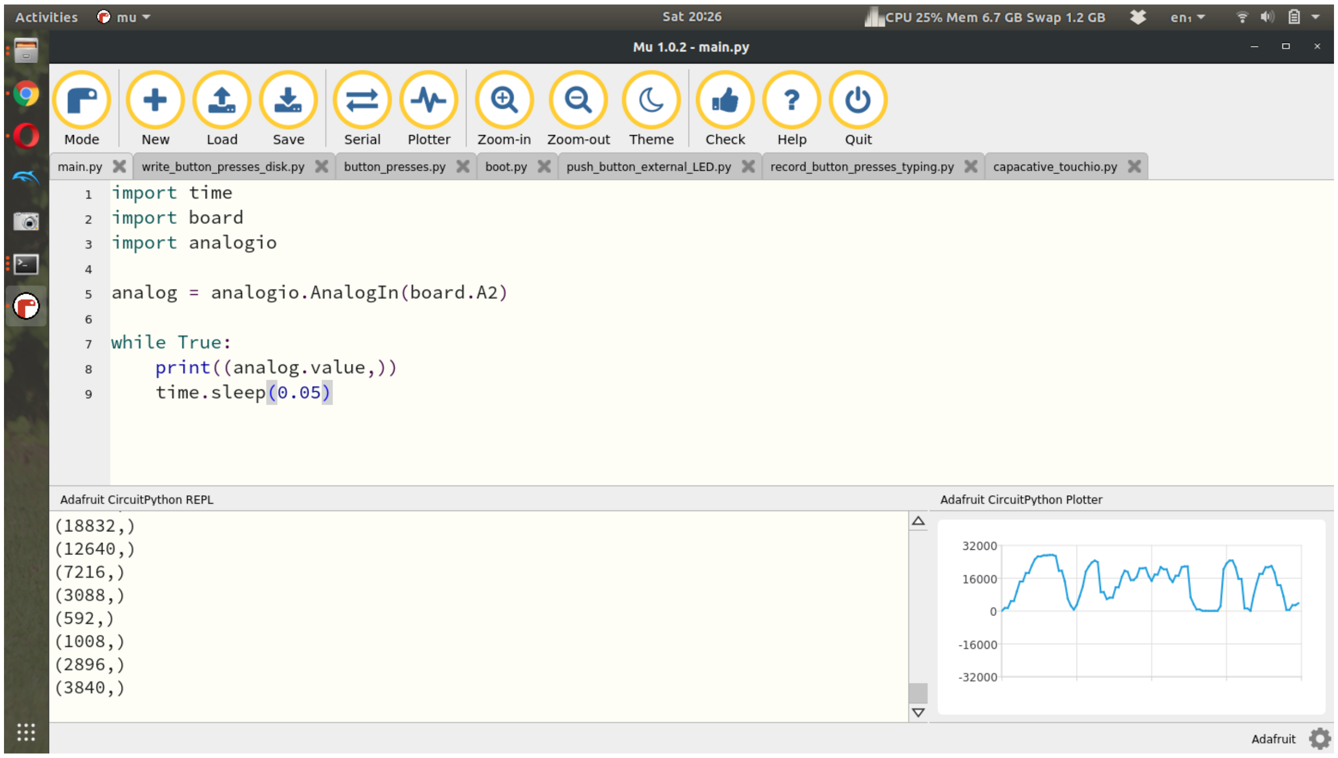
CAUTION!!!: Some potentiometers do not have enough resistance when turned all the way down. I suggest that you put a resistor in between the third leg and ground. Some experimenters have melted plastic or gotten really hot. One student even blew up a potentiometer.



There is a relevant Adafruit Learn Tutorial to help with the *analogio* module but I'll explain the minimum required here to get some analog values plotted in *Plotter* and Python on your computer. First let's take a look at some simple example code to read an analog signal and plot it using the *Plotter*.

```
1  import time
2  import board
3  import analogio
4
5  analog = analogio.AnalogIn(board.A2)
6
7  while True:
8      print((analog.value,))
9      time.sleep(0.05)
```

In the example code above, lines 1-3 again import the necessary modules with *analogio* being the new module here. Line 5 creates the analog object by attaching pin A2 to the analog function. Lines 7-9 then simple read the analog value and print it to *Serial* and the *Plotter*. Running this code on my laptop and turning the knob on the potentiometer produces this output. My potentiometer has a very large knob on the front and is easy to turn. Some potentiometers have a small screw on top that you need to turn with a screwdriver. Turning the screw or the knob results in chaning the resistance and therefore changing the voltage read by the CPX.



```
import time
import board
import analogio

analog = analogio.AnalogIn(board.A2)

while True:
    print((analog.value,))
    time.sleep(0.05)
```

```
(18832,)
(12640,)
(7216,)
(3088,)
(592,)
(1008,)
(2896,)
(3840,)
```

For this lab I want you to spin the potentiometer all the way to one side and then the other while recording time and the analog value. I then want you to plot the data with time on the x-axis and voltage on the y-axis. Remember to convert a digital output to voltage you just need to use the equation below where D is the raw value from the analog port. 3.3V is the range of the ADC and 2^{16} is the maximum value the ADC can represent.

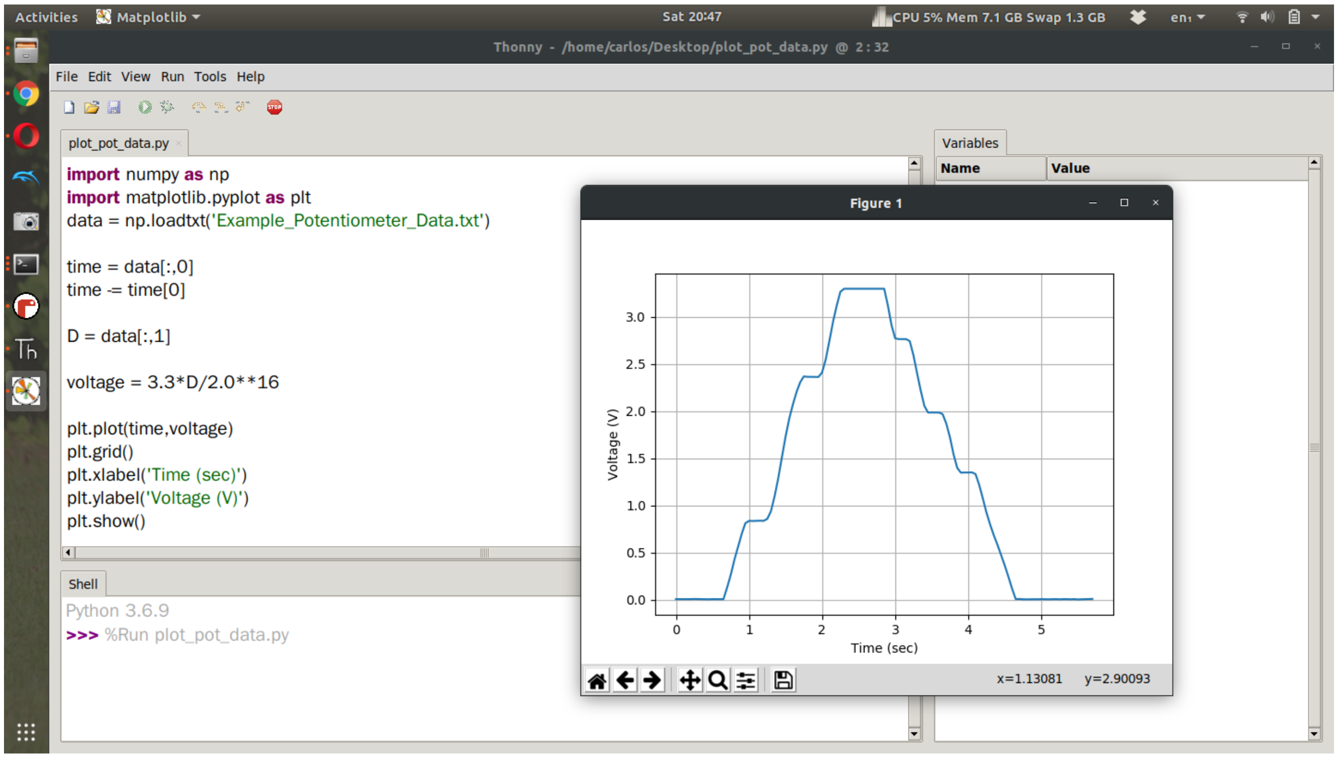
$$V = \frac{3.3D}{2^{16}} \tag{1}$$

After doing this experiment myself, this is the plot I obtain. The code is not provided as reading data and plotting has been discussed in a previous lab (See chapter **??**). From the screenshot though you can see how I convert the digital output to an analog signal.

NOTE THAT ON LINE 6 IT READS

```
time -= time[0]
```

Notice the minus sign in front of the equal sign. That effects a lot.



```
import numpy as np
import matplotlib.pyplot as plt
data = np.loadtxt('Example_Potentiometer_Data.txt')

time = data[:,0]
time -= time[0]

D = data[:,1]

voltage = 3.3*D/2.0**16

plt.plot(time,voltage)
plt.grid()
plt.xlabel('Time (sec)')
plt.ylabel('Voltage (V)')
plt.show()
```

```
Python 3.6.9
>>> %Run plot_pot_data.py
```

Your assignment for this lab is to do the same as I've done above. Wire up the potentiometer, read the analog signal and plot it in Python on your desktop computer. I've made some youtube videos on first just creating the circuit and plotting the data and then another video where I write data to the CPX using method 3.

1.4 Assignment

Once you've done that upload a PDF with all of the photos and text below included. My recommendation is for you to create a Word document and insert all the photos and text into the document. Then export the Word document to a PDF. For videos I suggest uploading the videos to Google Drive, turn on link sharing and include a link in your PDF.

1. Include a video of you wiggling the potentiometer and watching the digital signal in the Plotter in Mu go up and down (make sure your face is in the video at some point and you state your name) - 50%
2. Embed your plot of voltage vs time in a document and also include your code In Python both from the CPX and in Thonny or Spyder to plot - 50%

2 Wind Speed from Pitot Probe

2.1 Parts List

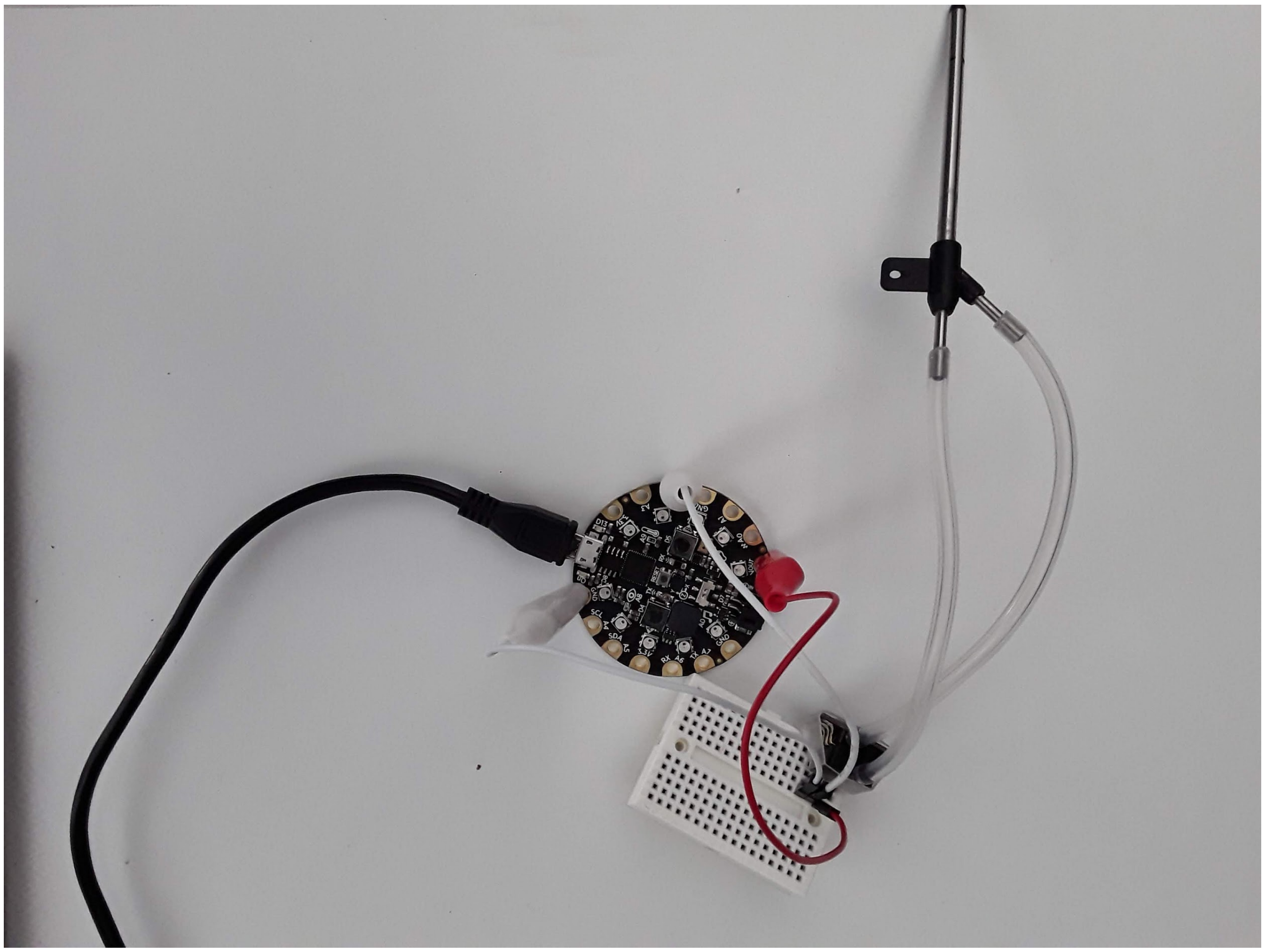
1. Laptop
2. CPX/CPB
3. USB Cable
4. Alligator Clips (x3)
5. Pitot Probe (Not included in kit at the moment so will need to buy this separately or borrow one)
6. Breadboard

2.2 Learning Objectives

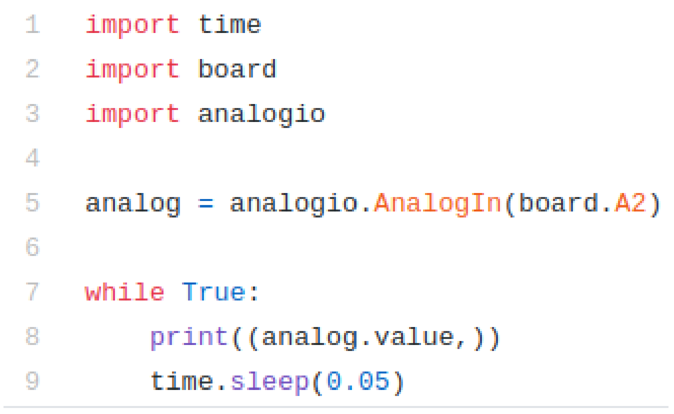
1. Understand how pitot probes works
2. Understand the relationship between a voltage signal from a pitot probe to a pressure value
3. Understand the relationship between pressure and windspeed

2.3 Getting Started

Although a CPX has numerous sensors built in, you can easily augment the capabilities of the CPX using either I2C or just the ADC on board the CPX. In this lab, if you purchased a pitot probe you will be able to do this assignment. Since you don't need the pitot probe for very long you can always borrow one from some other team. Let's talk about the hardware and the wiring to get this to work.



The pitot probe has two pressure taps that measure ambient pressure and stagnation pressure. These taps move through two silicon tubes to a pressure transducer that has a strain gauge in separating both pressures. When the pressure on one side of the transducer is larger than the other, it will flex the membrane and create strain. This strain runs through a wheatstone bridge with a voltage offset to the pin labeled analog. The transducer has 3 pins, +5V, GND and Analog. It is pretty straightforward how to wire this up but remember that +5V needs to go to VOUT, GND to GND and Analog to any analog pin. I chose pin A2. At that point it's very simple to just print the analog signal in bits to Serial. I've done this below. The code is the same analog code that we've used in the past.



```python
import time
import board
import analogio

analog = analogio.AnalogIn(board.A2)

while True:
    print((analog.value,))
    time.sleep(0.05)
```

The goal of the experiment is to take pitot probe data for 15 seconds with no wind, then 15 seconds of data with a fan on and then 15 seconds of no wind data. You'll need to use one of the datalogging methods (See chapter **??**) to log both time and pitot probe analog value. Once you have that data, import the data into Python on your desktop computer and convert the signal to windspeed. In order to convert the analog value to windspeed you need to first convert the analog value to voltage. Remember that the ADC on the CPX is going to convert the analog signal from the pitot probe to a digital output. So use this equation first to convert the analog signal (which I call D) to voltage. The 2 to the 16th power represents the 16 bit ADC.

$$V = \frac{3.3D}{2^{16}} \tag{2}$$

Before converting Voltage to windspeed we need to first subtract off the bias from the pitot probe. I explain this process in this accelerometer video. I've done this project before and have posted a video on Youtube about Converting Pitot Probe Data to Windspeed. *There is a typo in the video. V1 is supposed to have a sqrt()*. Once you have computed the bias you can compute the change in pressure using the equation below which converts the change in voltage to Pascals (V_b is the voltage bias you obtain when the wind is off). The data sheet states that the voltage is linearly proportional to pressure in Pascals which is nice.

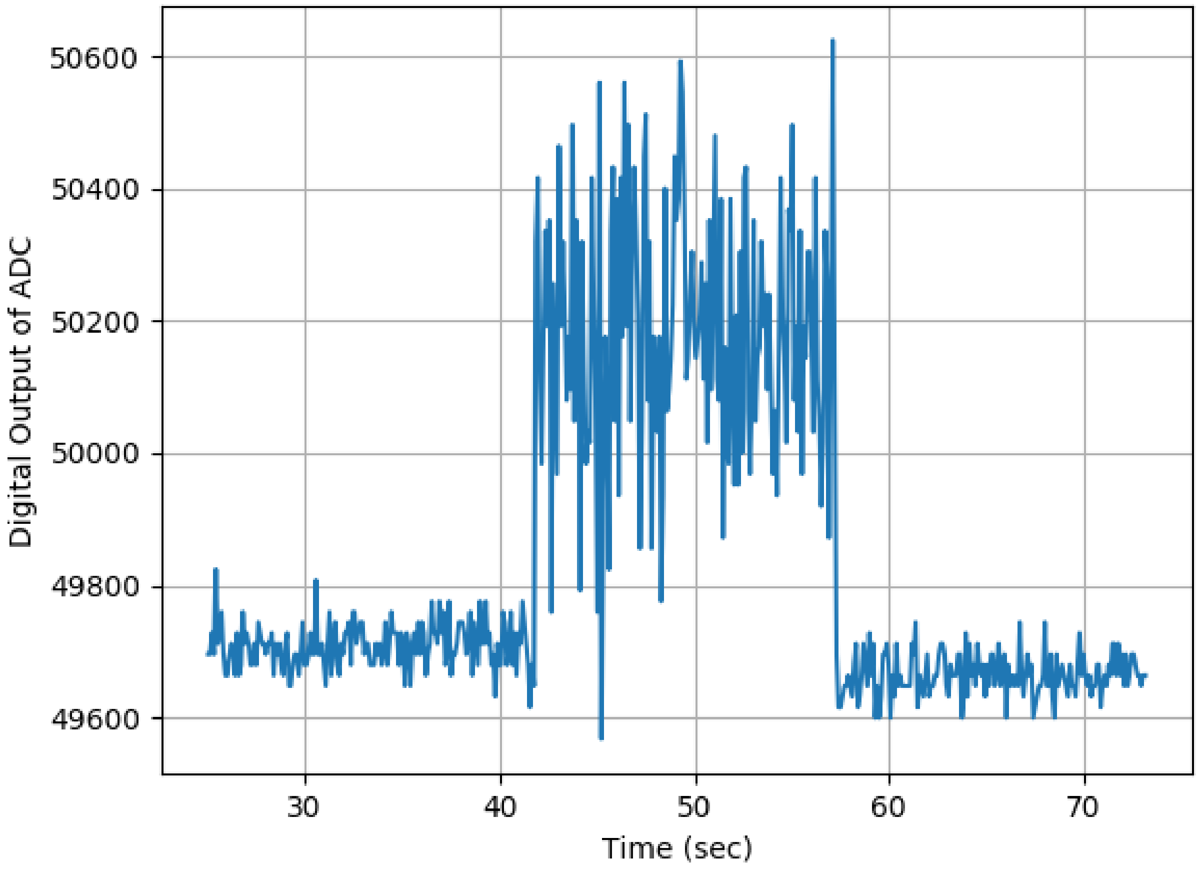
$$P = 1000(V - V_b) \tag{3}$$

Using pressure, the equation below can be used to compute windspeed, where U is the windspeed and the density at sea-level (ρ) is 1.225 kg/m^3.

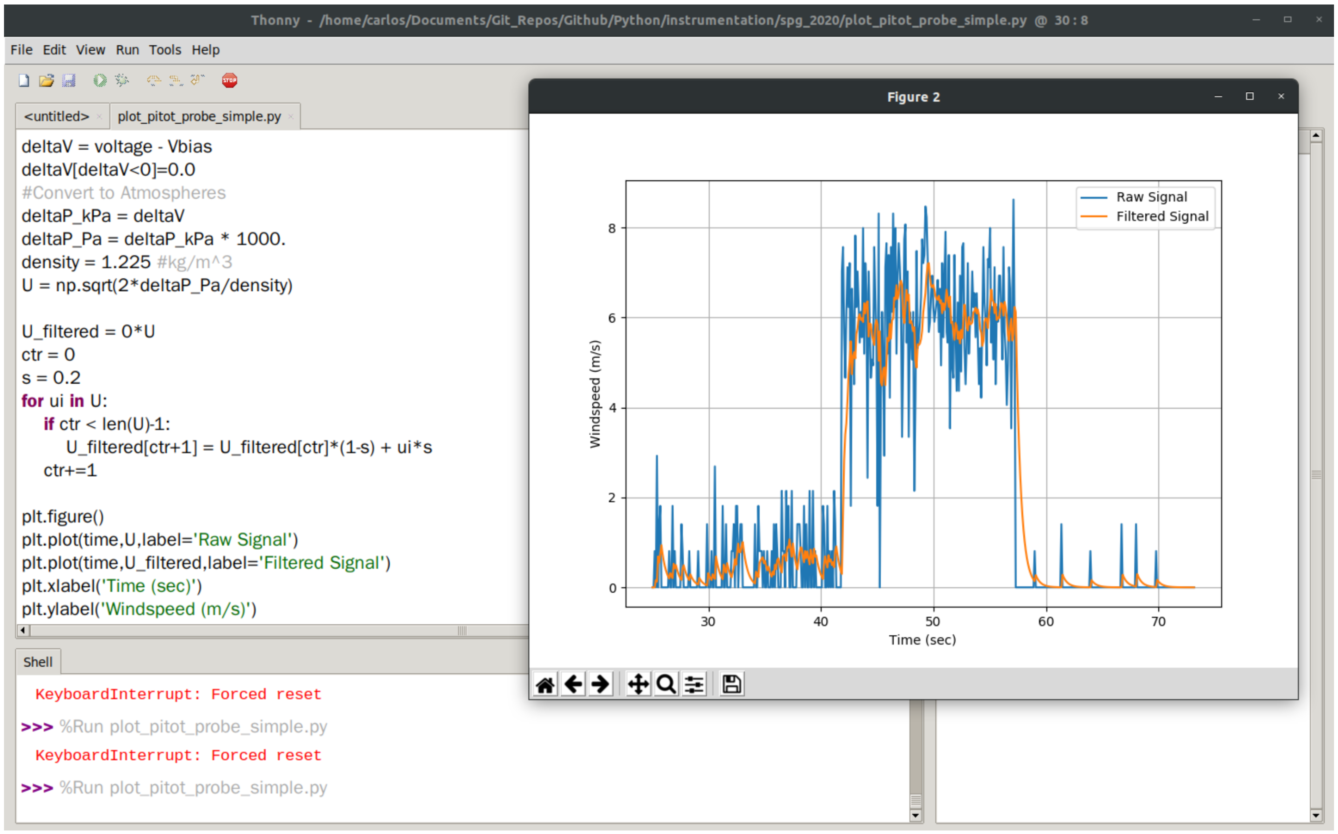
$$U = \sqrt{\frac{2P}{\rho}} \tag{4}$$

Using your data, create a plot of windspeed with time on the x-axis and windspeed on the y-axis. Some steps that might help you as you complete this project. First, have Mu plot the voltage coming from the pitot probe. If you've done everything right it will not be zero. The data sheet says there's an offset voltage of 2.5V so you will hopefully get something around 50,000 when you don't blow into the pitot probe. 50,000 multiplied by $3.3/2^16$ is around 2.5V. Make a note of that average value you get so you can subtract it off later. Once you've verified you're reading the pitot probe correctly, blow into the pitot probe and using the Plotter or Serial, verify that the analog signal increases. If the signal decreases, it means the pressure taps on the pressure transducer are backwards and you need to flip them. Either that or just flip the sign in your plotting routine on your computer but flipping the tubes might be easier for you. Hopefully when you do this lab you will get some data that looks like this. In this

Figure you'll see that when the fan wasn't running the signal was something around 49,800 which is fine. It means your bias is around 2.5 volts. Every pitot probe and circuit will be different. You can then convert this signal to voltage then and then pressure and then finally wind speed.



The code to accomplish this is relatively simple and a portion of the code is shown below. You'll see that when I subtracted the bias from the voltage I also zeroed out any negative values. That is, any delta voltage less than zero was set to zero. A couple of things about this chart. The data from the pitot probe is super noisy which means attaching a complementary filter is probably a good idea provided you don't over filter the signal and run into aliasing issues. You can see that I implemented an offline complementary filter and plotted it in the orange line which helps the noise issue quite a bit. You'll also notice that the noise is about 2 m/s. It turns out that pitot probes are actually not very accurate lower than about 2 m/s. They would be great for an airplane or you driving down the highway but they wouldn't be very good to take wind data outside on a calm day.


```
deltaV = voltage - Vbias
deltaV[deltaV<0]=0.0
#Convert to Atmospheres
deltaP_kPa = deltaV
deltaP_Pa = deltaP_kPa * 1000.
density = 1.225 #kg/m^3
U = np.sqrt(2*deltaP_Pa/density)

U_filtered = 0*U
ctr = 0
s = 0.2
for ui in U:
    if ctr < len(U)-1:
        U_filtered[ctr+1] = U_filtered[ctr]*(1-s) + ui*s
    ctr+=1

plt.figure()
plt.plot(time,U,label='Raw Signal')
plt.plot(time,U_filtered,label='Filtered Signal')
plt.xlabel('Time (sec)')
plt.ylabel('Windspeed (m/s)')
```

```
KeyboardInterrupt: Forced reset
>>> %Run plot_pitot_probe_simple.py
KeyboardInterrupt: Forced reset
>>> %Run plot_pitot_probe_simple.py
```
Thonny - /home/carlos/Documents/Git_Repos/Github/Python/instrumentation/spg_2020/plot_pitot_probe_simple.py @ 30 : 8
File Edit View Run Tools Help
<untitled>
plot_pitot_probe_simple.py
Shell
Figure 2
Raw Signal
Filtered Signal
Windspeed (m/s)
Time (sec)
0
2
4
6
8
30
40
50
60
70

2.4 Assignment

Once you've done that upload a PDF with all of the photos and text below included. My recommendation is for you to create a Word document and insert all the photos and text into the document. Then export the Word document to a PDF. For videos I suggest uploading the videos to Google Drive, turn on link sharing and include a link in your PDF.

1. If you borrowed a pitot probe return the pitot probe - Pass/Fail - If you don't return the pitot probe you receive a zero
2. Include a video of you taking data and explaining the circuit (make sure you are in the video) - 50%
3. Include a plot of the raw analog signal vs time just like I did above - 20%
4. Include a plot of windspeed vs time as I did above - 20%
5. Filter your signal using an offline complementary filter and include it in your plot like I did above - 10%

3 Circuit Playground (CPX/CPB) Modules

3.1 Parts List

1. Laptop
2. CPX/CPB
3. USB Cable

3.2 Learning Objectives

1. Understand the different sensors on the Circuitplayground
2. Learn the difference between high level and low level control
3. Get more practice plotting data from onboard sensors

3.3 Extra Help

If you need extra help on this assignment I have uploaded a youtube video where I read the temperature and accelerometer from the CircuitPlayground Bluefruit

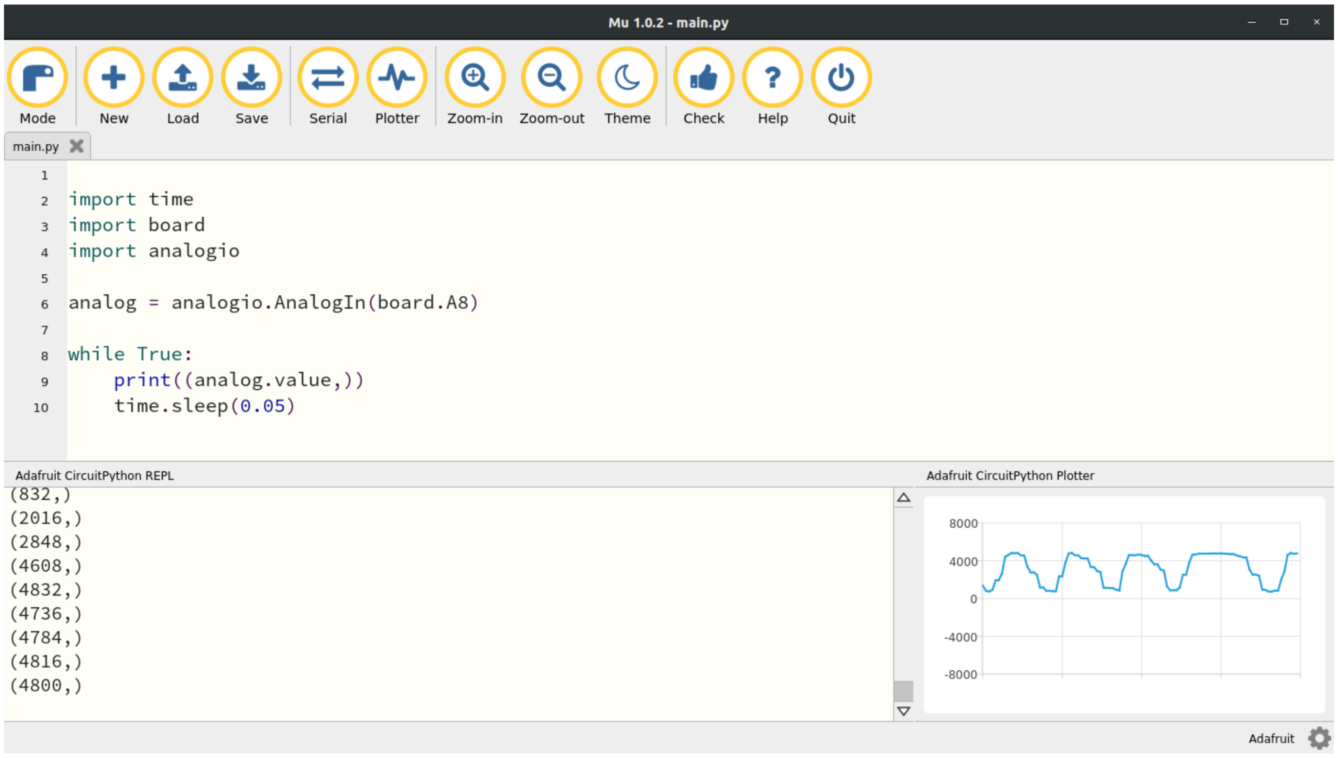
3.4 Getting Started

The CPX has numerous built-in sensors. These include a light sensor, an IR sensor, an accelerometer, a microphone, a speaker, some neopixels, a temperature sensor and 8 analog inputs with ADCs and even I2C (pronounced I squared C - it's a kind of serial communication) that you can use to easily hook up more sensors to it. We're not going to utilize all of these sensors since that would be a rather large project. Instead we're going to learn how to use the temperature, light and sound sensors as well as the accelerometer. For each of these examples there is a relatively easy way to access the sensors using a built-in module called *adafruit_circuitplayground.express* if you are using the CPX. If you are using the CPB you need to type *adafruit_circuitplayground.bluefruit.* It's a very nice module because it imports everything on the board. The problem is you can run into module conflicts. This happens when two different modules try to access the same pins on the CP. Sometimes if you import *adafruit_circuitplayground* you won't be able to import some other modules. **Note you might need to add the** *adafruit_circuitplayground* **library to your lib/ folder on your CIRCUITPY drive**. Due to this module conflict issue, there are some low level control commands you can use to access each of the sensors on board. We're obviously going to learn the low level control method first and then I'll show you how to access the sensors using the *adafruit_circuitplayground* module. **If you get a "currently in use" error it means you have a module conflict. Hence why I'm showing you the low level control method**.

3.5 Low Level Control

3.5.1 Light

The light sensor on the CPX is just a simple photocell wired in series with a resistor. There is a lab on photocells (See chapter 4 if you'd like to do that lab first to learn about photocells. The GND leg of the photocell is connected to pin A8. You can check the pin by looking at the graphic of an eye on the CPX and taking a look at the digital pin next to it. We've already learned how to access analog pins (See chapter 1) in a previous lab so just use the code from that lab and change the pin to A8. Here's what my code looks like when I change the pin to A8. I also brought the Plotter up and moved my finger in front of the light to make sure the light was working. Verify that your CPX responds the same way before moving on.



```python

import time
import board
import analogio

analog = analogio.AnalogIn(board.A8)

while True:
    print((analog.value,))
    time.sleep(0.05)
```

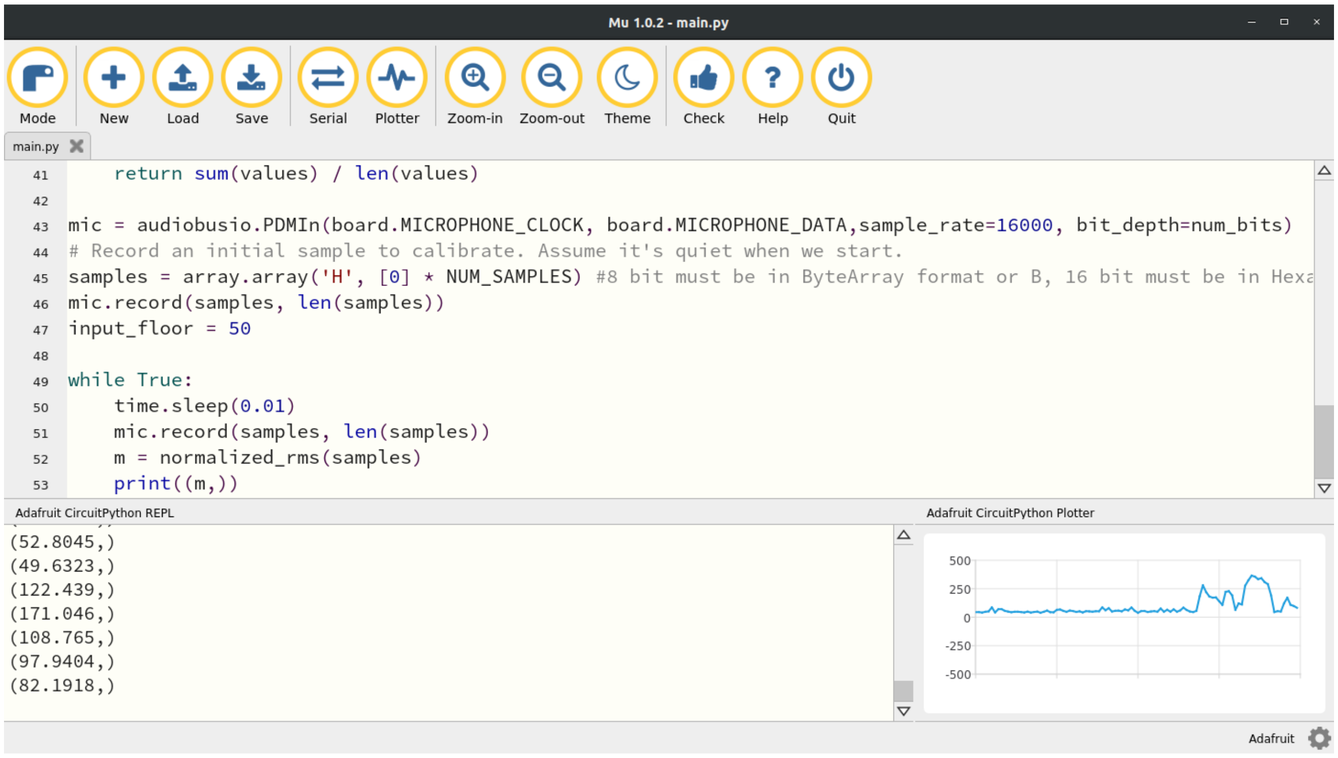
```
(832,)
(2016,)
(2848,)
(4608,)
(4832,)
(4736,)
(4784,)
(4816,)
(4800,)
```

3.5.2 Sound

The sound sensor uses the audiobusio library and creates a mic object using the (Pulse Density Modulation) PDM library. You have to set the sample rate and the number of bits to use to capture the data. We're going to set the bits to 16 to utilize the whole spectrum and then set the sample rate to 16 kHz. It's not quite 44.1 kHz like most modern microphones but it will do. After creating the mic object we have to compute some root mean squared values and thus two functions are defined before the while true loop in the code. The code itself is shown below. The code starts on line 22 because the first 22 lines are copyright from Dan Halbert, Kattni Rembor, and Tony DiCola from Adafruit Industries. I have edited the code a bit to fit my needs and uploaded my version to Github. In the code line 23-27 import standard modules as well as some new ones. The array module is used to create array like matrices. The math module is used to compute functions like cos, sin, and sqrt. Then of course the audiobusio module is used to create the mic object on line 42. Notice the two functions defined on 33 and 39 which create a function for computing the mean and for computing the normalized root mean square value of the data stream. Basically what's going to happen is we're going to record 160 samples as defined on line 160. So on line 43 we create a hexadecimal array (hexademical: base 16 hence the num_bits set to 16 on line 31) with 160 zeros. In the while true loop we're going to sleep for 0.01 seconds and then record some samples. Since we're sampling at 16 kHz the time it takes to record 160 samples is 160/16000 = 16/1600 = 1/100 = 0.01 seconds. Since we're taking 160 samples we need to compute some sort of average which is why the normalized root mean square value is computed on line 48.

```
22  # Circuit Playground Sound Meter
23  import array
24  import math
25  import audiobusio
26  import board
27  import time
28
29  # Number of samples to read at once.
30  NUM_SAMPLES = 160
31  num_bits = 16
32
33  def normalized_rms(values):
34      meanbuf = int(mean(values))
35      samples_sum = sum(float(sample - meanbuf) * (sample - meanbuf) for sample in values)
36      rms_mean = math.sqrt(samples_sum/len(values)) ##Notice that samples_sum = (sample-mean)**2
37      return rms_mean
38
39  def mean(values):
40      return sum(values) / len(values)
41
42  mic = audiobusio.PDMIn(board.MICROPHONE_CLOCK, board.MICROPHONE_DATA,sample_rate=16000, bit_depth=num_bits)
43  samples = array.array('H', [0] * NUM_SAMPLES)
44
45  while True:
46      time.sleep(0.01)
47      mic.record(samples, len(samples))
48      m = normalized_rms(samples)
49      print((m,))
```

When I run this code and talk normally into the microphone. I get this output in the Plotter. You'll notice that the data is pretty noisy in the beginning. It's possible we could increase the number of samples we take each loop by editing line 30 but that would slow down our code. So there's a tradeoff between filtering here and speed. That's something will investigate in some later labs.



```
    return sum(values) / len(values)

mic = audiobusio.PDMIn(board.MICROPHONE_CLOCK, board.MICROPHONE_DATA,sample_rate=16000, bit_depth=num_bits)
# Record an initial sample to calibrate. Assume it's quiet when we start.
samples = array.array('H', [0] * NUM_SAMPLES) #8 bit must be in ByteArray format or B, 16 bit must be in Hexa
mic.record(samples, len(samples))
input_floor = 50

while True:
    time.sleep(0.01)
    mic.record(samples, len(samples))
    m = normalized_rms(samples)
    print((m,))
```

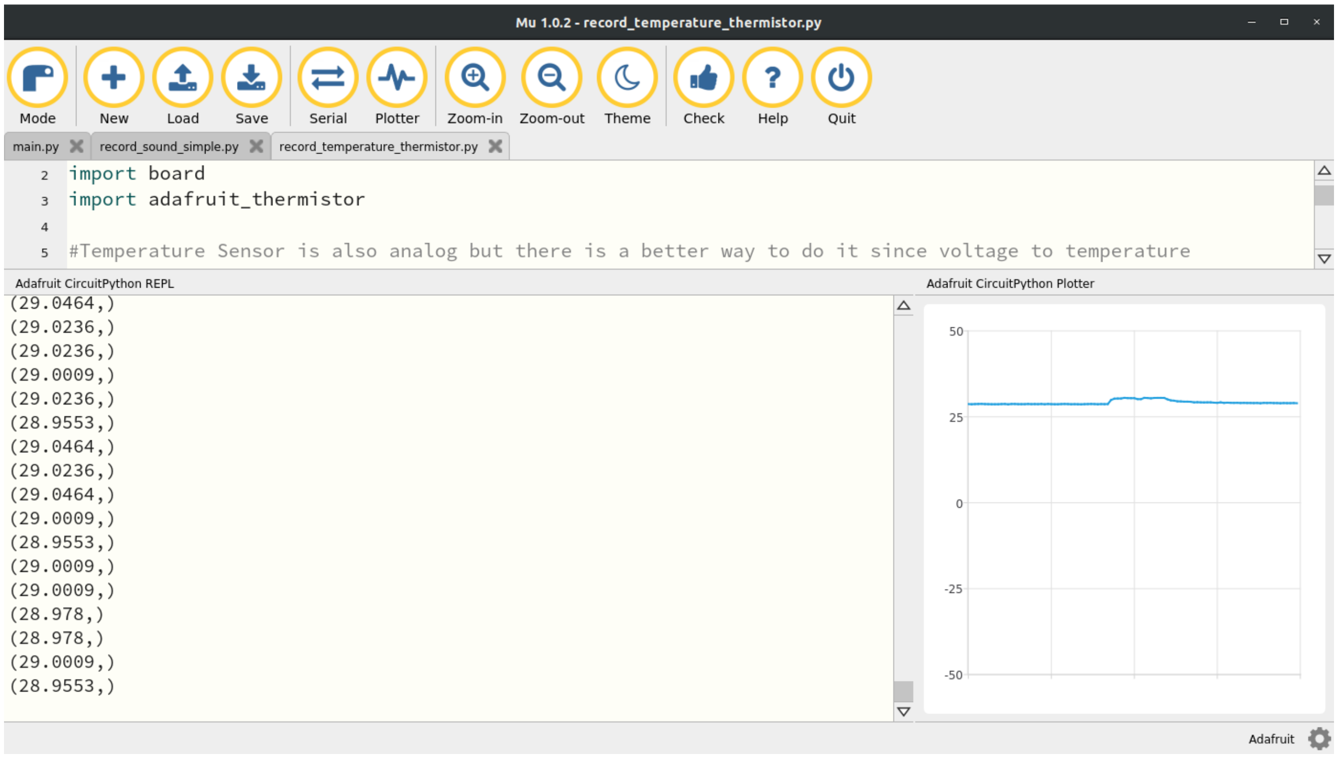
```
(52.8045,)
(49.6323,)
(122.439,)
(171.046,)
(108.765,)
(97.9404,)
(82.1918,)
```

3.5.3 Temperature

The temperature sensor is actually a thermistor. A thermistor is basically a thermometer resistor which means the resistance depends on temperature. This means that you can read the analog signal coming from the thermistor just by reading the analog signal from pin A9. If you look for the thermometer symbol on the CPX you'll see pin A9. Therefore, it is possible to just use the analogio library and just read in the analog voltage but in order to convert to celsius and then fahrenheit you need to use some heat transfer equations to convert the analog signal to celsius. Thankfully the folks at Adafruit have done it again with an *adafruit_thermistor* module. If you head over to their github on this module you'll see the relevant conversion under the definition temperature which at the time of this writing is on line 86. The Adafruit Learn system also does a bit of work to explain the conversion from voltage to temperature. For now though we will just appreciate the simplicity of the code below which is also on my Github.

```
1   import time
2   import board
3   import adafruit_thermistor
4
5   #Temperature Sensor is also analog but there is a better way to do it since voltage to temperature
6   #Is nonlinear and depends on series resistors and b_coefficient (some heat transfer values)
7   #thermistor = AnalogIn(board.A9) ##If you want analog
8   thermistor = adafruit_thermistor.Thermistor(board.A9, 10000, 10000, 25, 3950)
9
10  while True:
11      temp = thermistor.temperature
12      #temp = thermistor.value #if you want analog
13      print((temp,))
14      time.sleep(0.5)
```

As always lines 1-3 import the relevant modules and then line 8 create the thermistor object. You'll notice the input arguments are the pin which is A9 as well as the resistor values which are in series with the thermistor. These resistors are soldered to the PCB so they are fixed at 10 kOhms. The 25 is for the nominal resistance temperature in celsius of the thermistor and 3950 is the b coefficient which is a heat transfer property. Running this code and then placing my finger on the A9 symbol causes the temperature to rise just a bit. You'll notice the temperature rise quite quickly when I place my finger on the sensor but when I remove the sensor it takes some time before the sensor cools off. This has to do with the dynamic response of the sensor. We'll discuss this in some future labs on dynamic measurements. For now you can move on to the accelerometer.

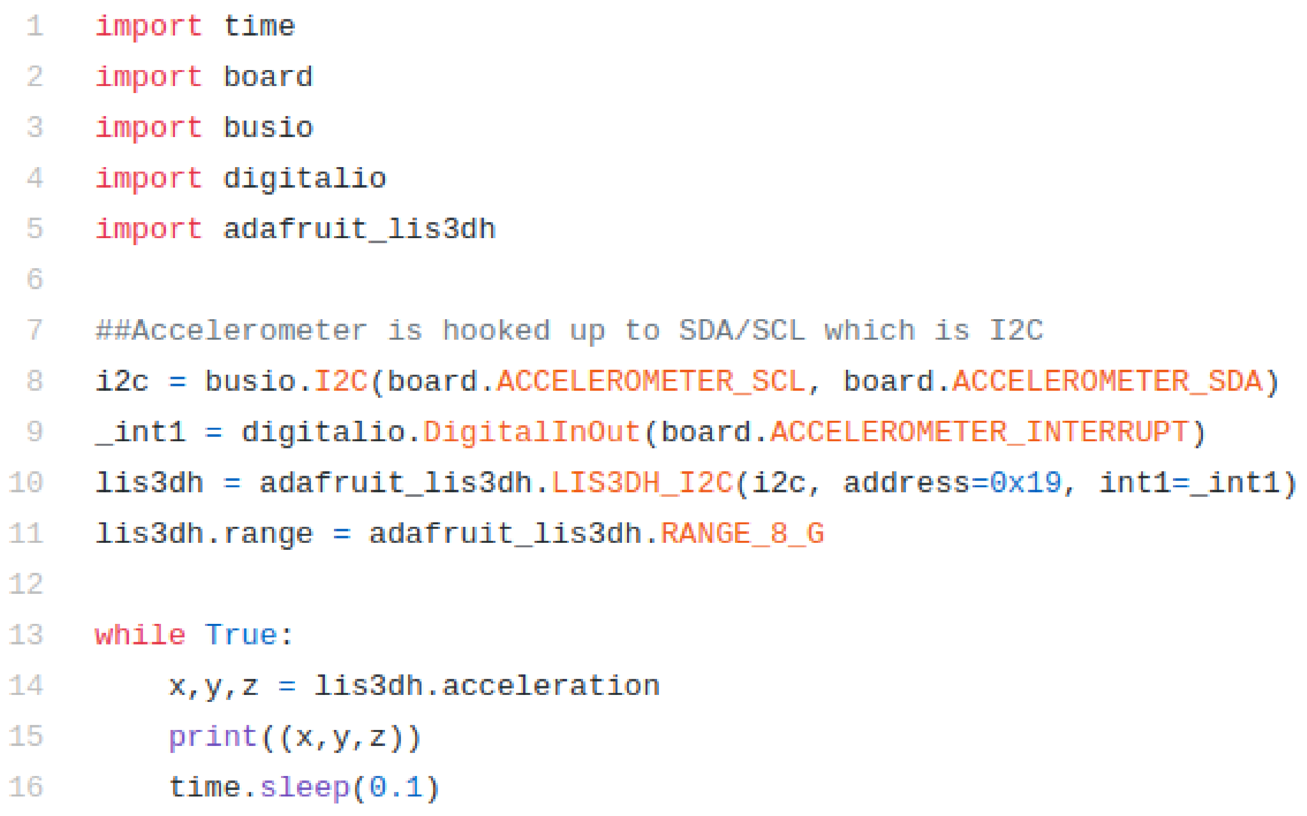

Mu 1.0.2 - record_temperature_thermistor.py
Mode
New
Load
Save
Serial
Plotter
Zoom-in
Zoom-out
Theme
Check
Help
Quit
main.py
record_sound_simple.py
record_temperature_thermistor.py
Adafruit CircuitPython REPL
Adafruit CircuitPython Plotter
50
25
0
-25
-50
Adafruit

```
import board
import adafruit_thermistor

#Temperature Sensor is also analog but there is a better way to do it since voltage to temperature
```

```
(29.0464,)
(29.0236,)
(29.0236,)
(29.0009,)
(29.0236,)
(28.9553,)
(29.0464,)
(29.0236,)
(29.0464,)
(29.0009,)
(28.9553,)
(29.0009,)
(29.0009,)
(28.978,)
(28.978,)
(29.0009,)
(28.9553,)
```

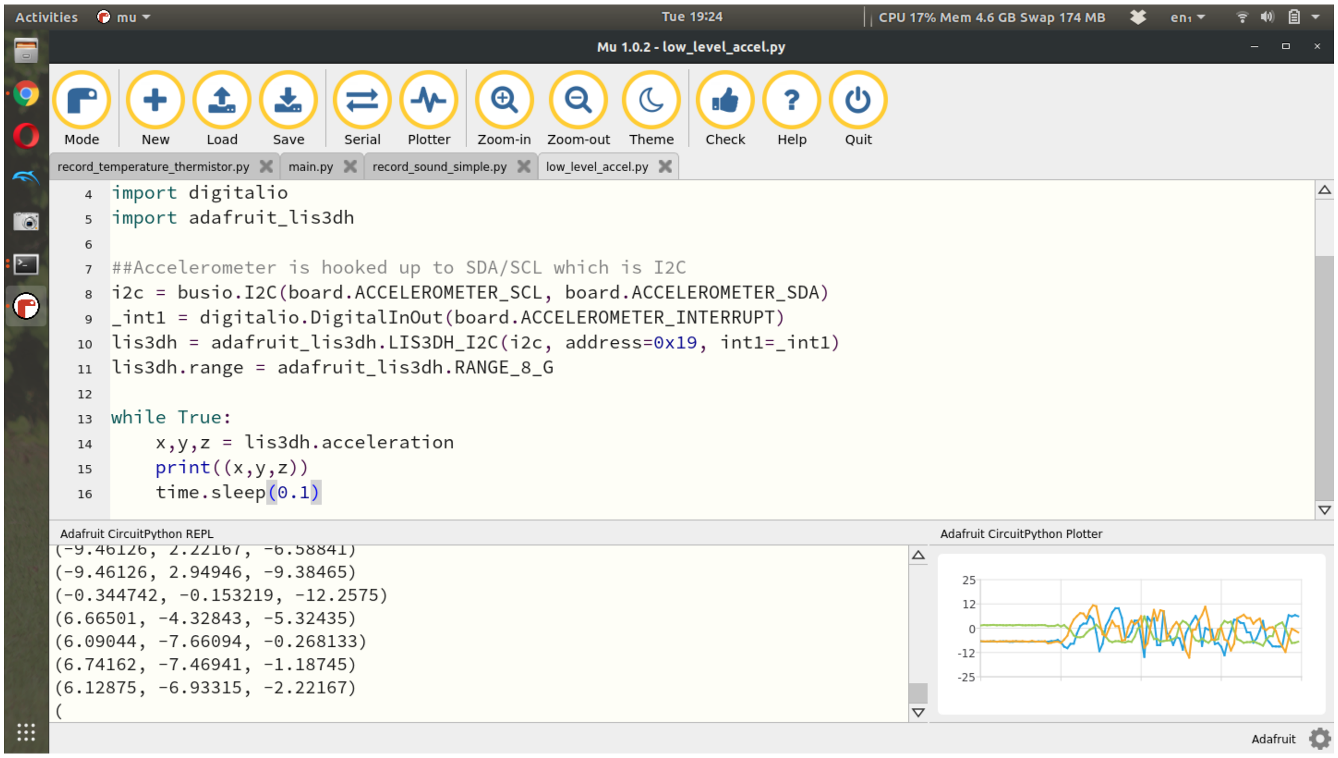
3.5.4 Accelerometer

The accelerometer is a 3-axis sensor. As such it is going to spit out not just 1 value but 3 values. Accelerations in x,y and z or North, East, Down or Forward, Side to Side, Up and Down. Since it's reading 3 values we can't just read 3 analog signals (we can but the accelerometer chip design didn't want to do that) so instead we're going to use the I2C (I like indigo and square C. So I squared C. Not 12C or one two C. It's I squared C) functionality. I2C is a type of serial communication that allows computers to send strings rather than numbers. It's a much more complex form of communication but since it's standard we can just use the busio module which contains the I2C protocol.



```
1   import time
2   import board
3   import busio
4   import digitalio
5   import adafruit_lis3dh
6
7   ##Accelerometer is hooked up to SDA/SCL which is I2C
8   i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
9   _int1 = digitalio.DigitalInOut(board.ACCELEROMETER_INTERRUPT)
10  lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c, address=0x19, int1=_int1)
11  lis3dh.range = adafruit_lis3dh.RANGE_8_G
12
13  while True:
14      x,y,z = lis3dh.acceleration
15      print((x,y,z))
16      time.sleep(0.1)
```

In this code we see alot more imports than normal. In addition to the standard time, board and digitalio modules we need the busio module and the *adafruit_lis3dh*. You might think that LIS3DH is a very weird name for an accelerometer but it's actually the name of the chip on your CPX. The chip itself is very standard and is well documented on multiple websites. Here's one from ST. You can also buy the chip on a breakout board from Adafruit and then of course the Adafruit Learn site has plenty of tutorials on reading Accelerometer data in CircuitPython. As always I've learned what I can from the relevant tutorials and created my own simple version to read the accelerometer data and posted it to Github. I digress, lines 8-11 of the code do alot. It first uses the SCL and SDA pins to set up an I2C object which establishes serial communication to the accelerometer. Line 9 creates an interrupt which is beyond the scope of this course. Finally, line 10 creates the actual accelerometer object by sending it the I2C pins, the hexadecimal address in the I2C protocol and finally the interrupt pin. Line 11 then sets the range. Line 14 in the while loop is where the x,y and z values of the accelerometer are read and then promptly printed to Serial on line 15. If I run this code and shake the sensor a bit I can get all the values to vary. If you put the CPX on a flat surface, the Z axis will measure something close to 9.81. The units of the accelerometer are clearly in m/s^2.


Activities
mu
Tue 19:24
CPU 17% Mem 4.6 GB Swap 174 MB
Mu 1.0.2 - low_level_accel.py
Mode
New
Load
Save
Serial
Plotter
Zoom-in
Zoom-out
Theme
Check
Help
Quit
record_temperature_thermistor.py
main.py
record_sound_simple.py
low_level_accel.py
Adafruit CircuitPython REPL
Adafruit CircuitPython Plotter
25
12
0
-12
-25
Adafruit

```
import digitalio
import adafruit_lis3dh

##Accelerometer is hooked up to SDA/SCL which is I2C
i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
_int1 = digitalio.DigitalInOut(board.ACCELEROMETER_INTERRUPT)
lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c, address=0x19, int1=_int1)
lis3dh.range = adafruit_lis3dh.RANGE_8_G

while True:
    x,y,z = lis3dh.acceleration
    print((x,y,z))
    time.sleep(0.1)
```

```
(-9.46126, 2.22167, -6.58841)
(-9.46126, 2.94946, -9.38465)
(-0.344742, -0.153219, -12.2575)
(6.66501, -4.32843, -5.32435)
(6.09044, -7.66094, -0.268133)
(6.74162, -7.46941, -1.18745)
(6.12875, -6.93315, -2.22167)
(
```

3.6 High Level Control

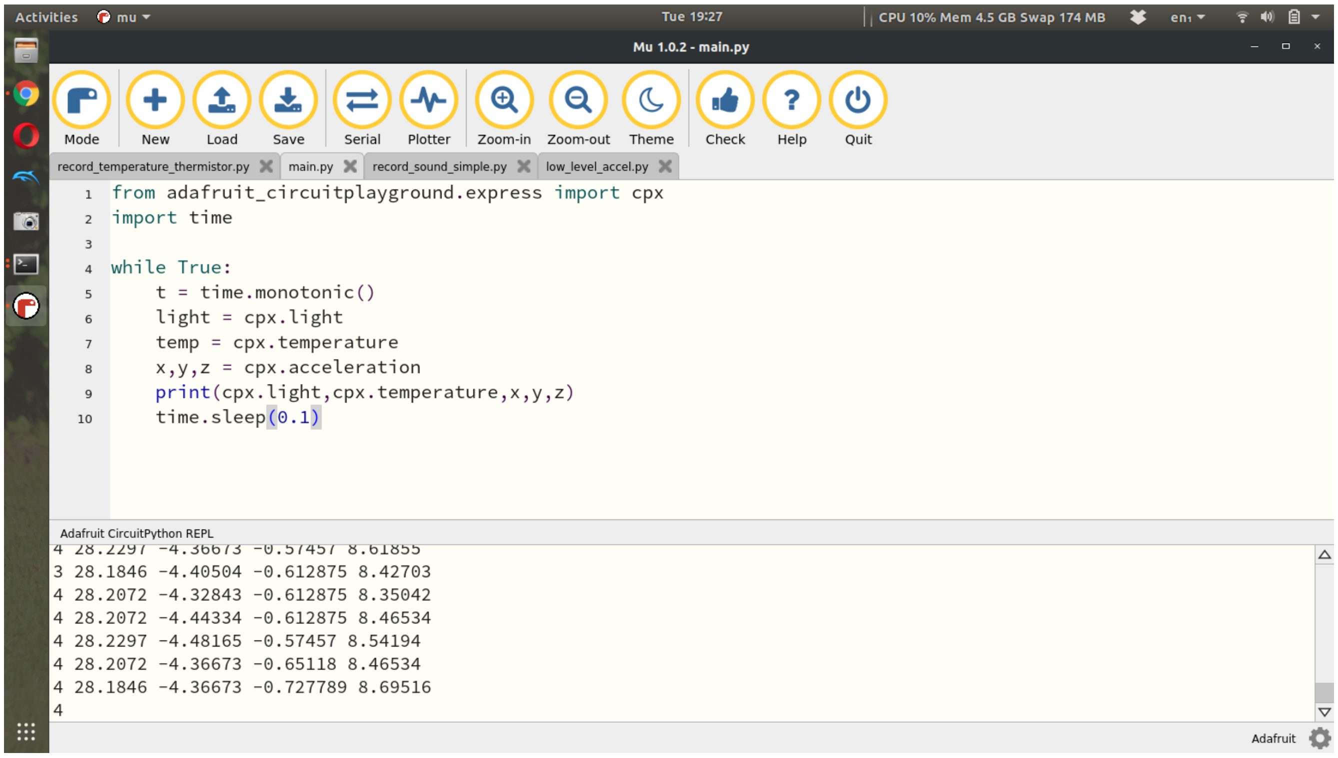
Alright so we've learned the hard way for all the sensors using low level control of the various sensors. Let's now import the simple adafruit_circuitplayground.express module. The Adafruit Learn site offers pretty much every example code snippet you'd ever need for all the different push buttons and sensors on the CPX. Head over there if you ever need something outside of the scope of this text. As I said before, the main module you need to import is done by adding the following to the top of your code

```
from adafruit_circuitplayground.express import cpx
```

Note that you need to change that line to adafruit_circuitplayground.bluefruit import cpb. Then everywhere you see cpx you replace with cpb. This will import the cpx module into your working code. From here the commands to read different things are relatively simple. Here are the commands for all the various sensors

```
light = cpx.light

x,y,z = cpx.acceleration

temperature = cpx.temperature
```

There unfortunately is no simple module for the sound sensor. You'll still need to use the low level control no matter what. According to Adafruit though, if you get the Circuit Playground Bluefruit there is a simple way to read the sound level. Implementing the various sensors into a while loop on my CPX looks like this.



```
from adafruit_circuitplayground.express import cpx
import time

while True:
    t = time.monotonic()
    light = cpx.light
    temp = cpx.temperature
    x,y,z = cpx.acceleration
    print(cpx.light,cpx.temperature,x,y,z)
    time.sleep(0.1)
```

```
4 28.2297 -4.36673 -0.57457 8.61855
3 28.1846 -4.40504 -0.612875 8.42703
4 28.2072 -4.32843 -0.612875 8.35042
4 28.2072 -4.44334 -0.612875 8.46534
4 28.2297 -4.48165 -0.57457 8.54194
4 28.2072 -4.36673 -0.65118 8.46534
4 28.1846 -4.36673 -0.727789 8.69516
4
```

I left out the sound sensor stuff just because it kind of messes with the simplicity of the code above. The adafruit_circuitplayground.express module outputs just as before except for the light sensor. In the low level control we simply computed the voltage across the photocell but the adafruit_circuitplayground.express module outputs data in Lux.

3.7 Assignment

Using either **low or high level control** take at least 60 seconds of data for each of the sensors above. Make sure log time and the raw sensor value at 1Hz or faster. To make the project more challenging, try and log all sensor data all at once. Import the data onto your desktop and make 4 plots total with time on the x-axis and the sensor data on the y-axis.

Once you've done that upload a PDF with all of the photos and text below included. My recommendation is for you to create a Word document and insert all the photos and text into the document. Then export the Word document to a PDF. For videos I suggest uploading the videos to Google Drive, turn on link sharing and include a link in your PDF.

1. Using the Plotter in Mu, take a video of you plotting light, sound, temperature and acceleration one at a time while varying light, sound, temperature and acceleration individually and watching the plot change in Mu (be sure to be in the video and introduce yourself) - 30%
2. Copy and paste your CPX code that you used to log data for each of the sensors - 20%
3. Copy and paste your Python desktop code that you used to plot data for each of the 4 sensors - 20%
4. Include 4 plots with time on the x-axis and sensor data on the y-axis (No screenshots) - 30%

4 Histograms and Normal Distribution of Photocell Readings

4.1 Parts List

1. Laptop
2. CPX/CPB
3. USB Cable
4. Photocell
5. Resistor (10 kOhm)
6. Alligator Clips (x3)
7. Breadboard

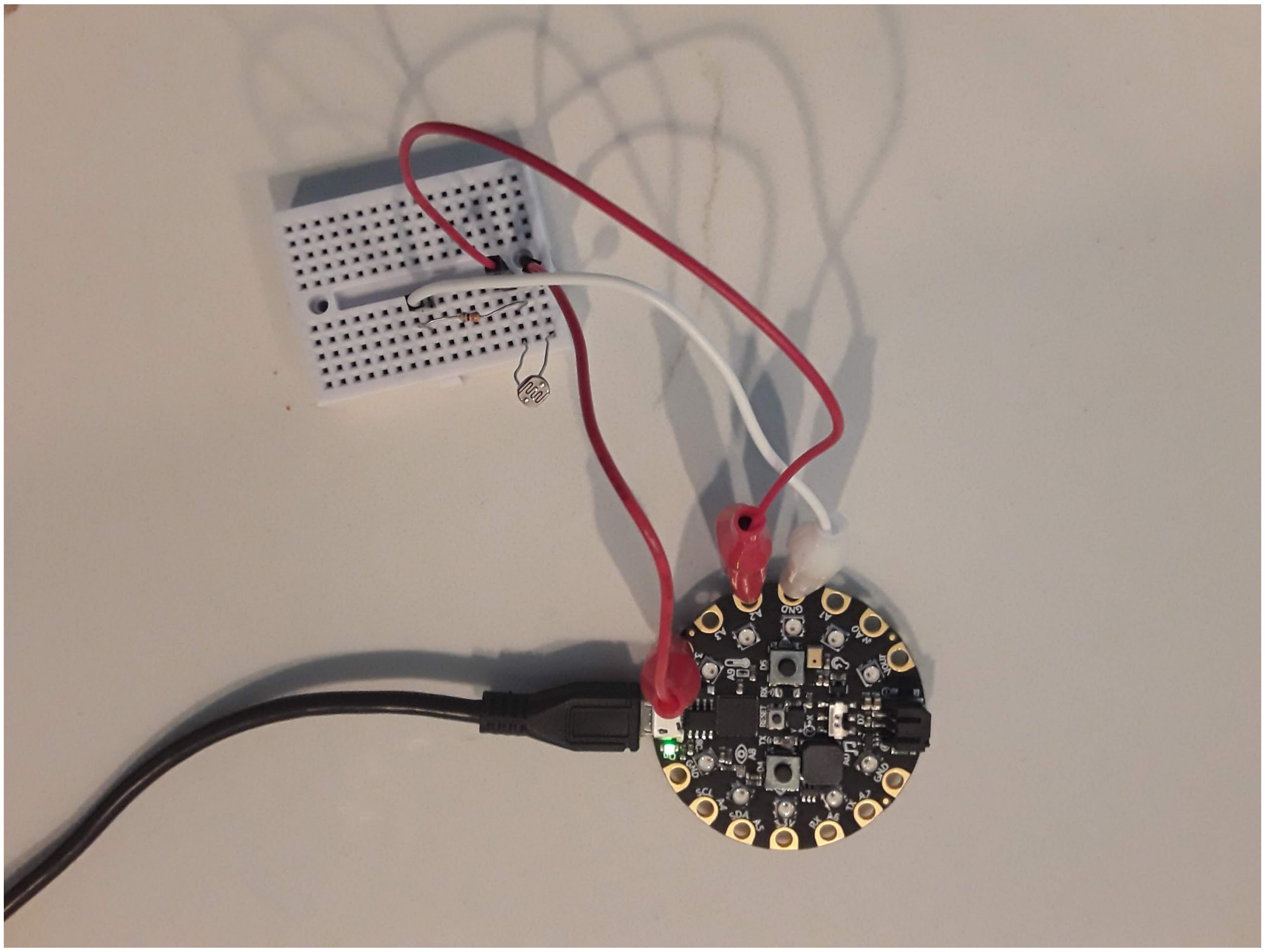
4.2 Learning Objectives

1. Understand how a photocell responds to light level by measuring the voltage across the photocell in different light conditions
2. Learn how to create histograms of a noisy data signal
3. Understand mean and standard deviation and how that applies to Normal distributions.

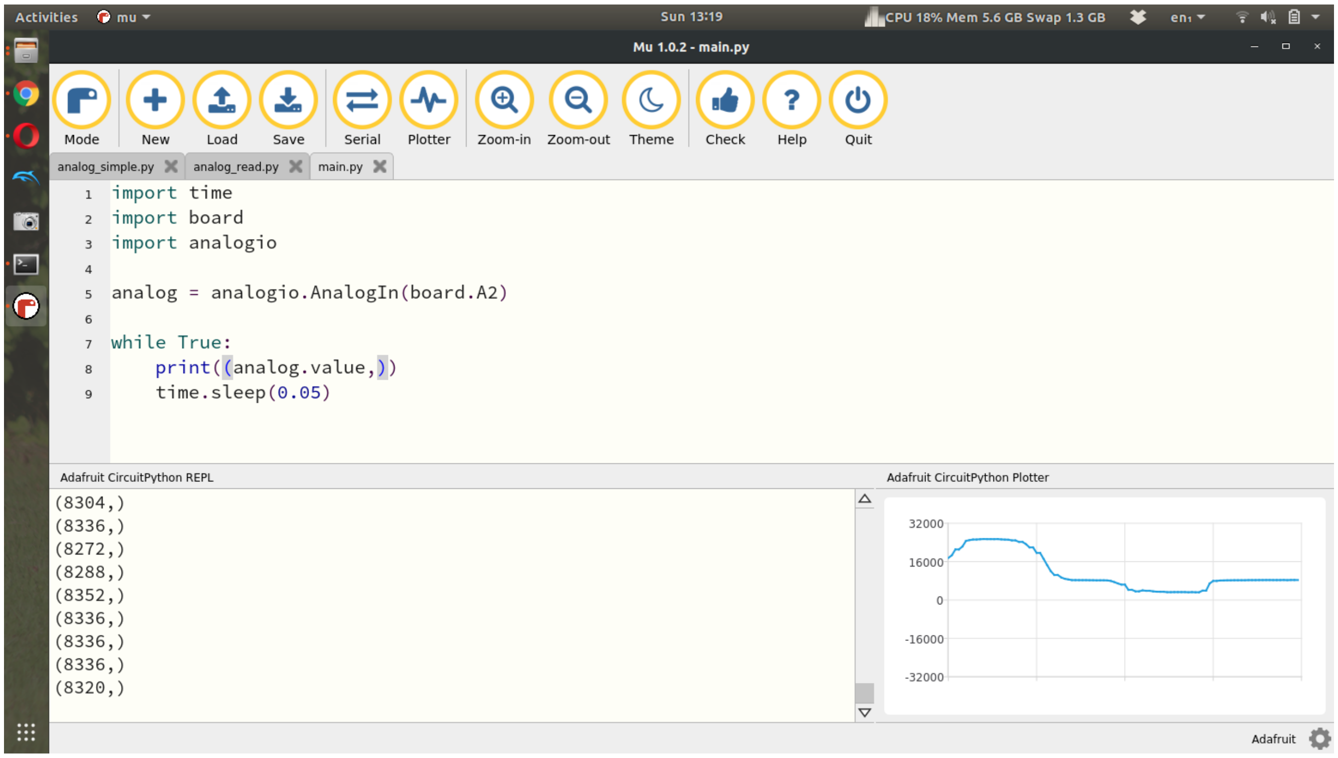
4.3 Getting Started

This lab is going to be similar to the potentiometer lab (See chapter 1). We are going to use a photocell though to vary the resistance instead of a potentiometer. Photocells are cool because they change their resistance solely based on the light intensity hitting the sensor rather than twisting a knob like the potentiometer.

Wiring a photocell is similar to a potentiometer except that you need to add a resistor in series with the photocell. There is a relevant Adafruit Tutorial on Photocells and the code required to measure the voltage if you'd like to read more about it. The lab this week requires you to do the same as the potentiometer lab. I'd like you to wire up the circuit, take data at the low and high value of the photocell by covering the sensor with your finger and then shining a light on it and plotting the entire data set in Python on your desktop computer. The wiring diagram is shown below. Have an alligator clip connected to 3.3V on the CPX and have it connected to either end of the photocell. Then place a resistor in series with the photocell and route the free end of the resistor to GND.



Then take another alligator clip from pin A2 and plug it into the same row on the breadboard as the resistor and photocell. Once you have the circuit wired properly you can use the same code as the potentiometer lab. The example screenshot below shows the analog signal below showing a high spike where I placed a flashlight over the photocell and then a low spot where I covered the photocell with my finger. Remember that you can use any Analog pin on the CPX provided you change line 5 to the same pin.

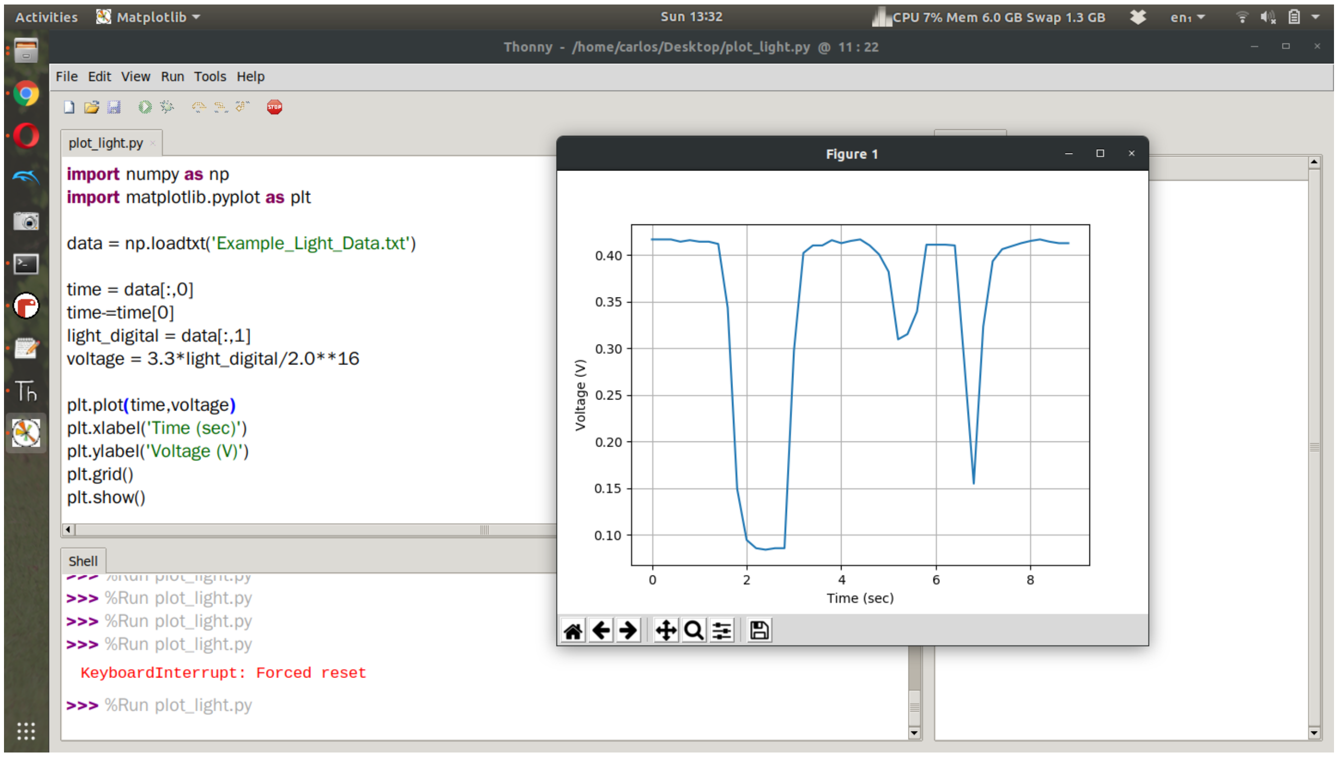


```
import time
import board
import analogio

analog = analogio.AnalogIn(board.A2)

while True:
    print((analog.value,))
    time.sleep(0.05)
```

Adafruit CircuitPython REPL

```
(8304,)
(8336,)
(8272,)
(8288,)
(8352,)
(8336,)
(8336,)
(8336,)
(8320,)
```

Once you've gotten some example data you can plot the result in Python as you did for the potentiometer lab. Here's what your plot may look like.

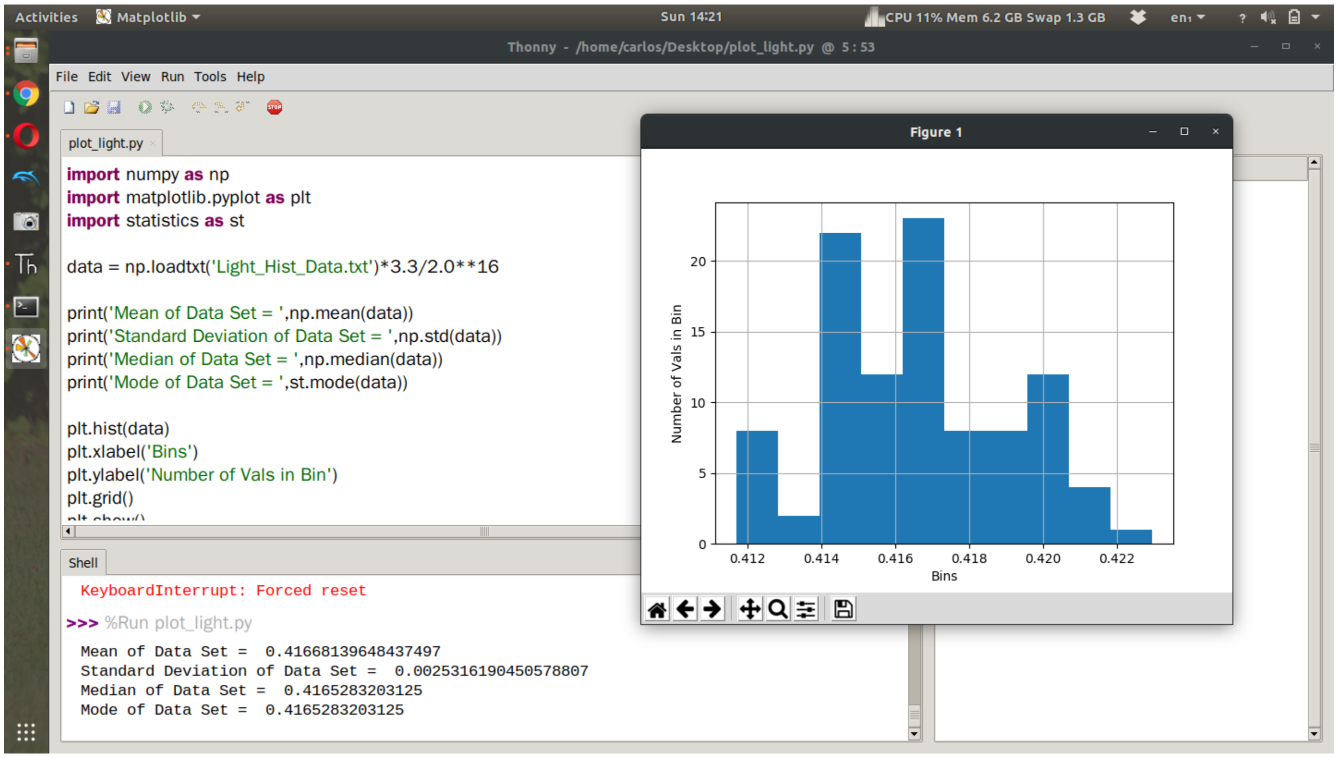


```
import numpy as np
import matplotlib.pyplot as plt

data = np.loadtxt('Example_Light_Data.txt')

time = data[:,0]
time-=time[0]
light_digital = data[:,1]
voltage = 3.3*light_digital/2.0**16

plt.plot(time,voltage)
plt.xlabel('Time (sec)')
plt.ylabel('Voltage (V)')
plt.grid()
plt.show()
```

```
>>> %Run plot_light.py
>>> %Run plot_light.py
>>> %Run plot_light.py
  KeyboardInterrupt: Forced reset
>>> %Run plot_light.py
```

Thonny - /home/carlos/Desktop/plot_light.py @ 11 : 22
File Edit View Run Tools Help
plot_light.py
Shell
Figure 1
0.40
0.35
0.30
0.25
0.20
0.15
0.10
Voltage (V)
0
2
4
6
8
Time (sec)

If you noticed the data you obtained even when the light source was constant was quite noisy. What I'd like you to do for part 2 of this lab is take 100 data points with the photocell with as constant of a light source as possible. Do this for three different light ranges. Low Light, ambient light and then a flashlight. With the three different data streams, create a histogram of the data with appropriate labels and compute the mean, median, and standard deviation of the data stream. Creating a histogram in Python is fairly simple and I have a Youtube Video to supplement this tutorial. I also have another video where I get mean and median values for accelerometer data. Still, here is my example code showing code to get mean, median, and standard deviation as well as create the histogram. Notice in my code I imported the statistics module to compute the mode. Although it worked in my code, it's not typical to compute the mode of a continuous variable because often times you will not ever get the same value twice. Still, feel free to compute the mode if you so desire.



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

data = np.loadtxt('Light_Hist_Data.txt')*3.3/2.0**16

print('Mean of Data Set = ',np.mean(data))
print('Standard Deviation of Data Set = ',np.std(data))
print('Median of Data Set = ',np.median(data))
print('Mode of Data Set = ',st.mode(data))

plt.hist(data)
plt.xlabel('Bins')
plt.ylabel('Number of Vals in Bin')
plt.grid()
plt.show()
```

```
KeyboardInterrupt: Forced reset
>>> %Run plot_light.py

  Mean of Data Set =  0.41668139648437497
  Standard Deviation of Data Set =  0.0025316190450578807
  Median of Data Set =  0.4165283203125
  Mode of Data Set =  0.4165283203125
```

Again make sure to convert to voltage before you plot that way you can see what the noise level is in volts. Notice that I import the data and convert to voltage all in one line. However, note that my text file only has 1 column of data. It's possible your data has time in the first column and light value in the second column at which point you will need to extract the second column first and then convert to voltage. If you have two columns of data you'll need to add a few things. First, don't convert to voltage when you import the data.

```
data = np.loadtxt('Light_Hist_Data.txt')
```

Then extract the second column (assuming your photocell readings are in that column)

```
second_column = data[:,1]
```

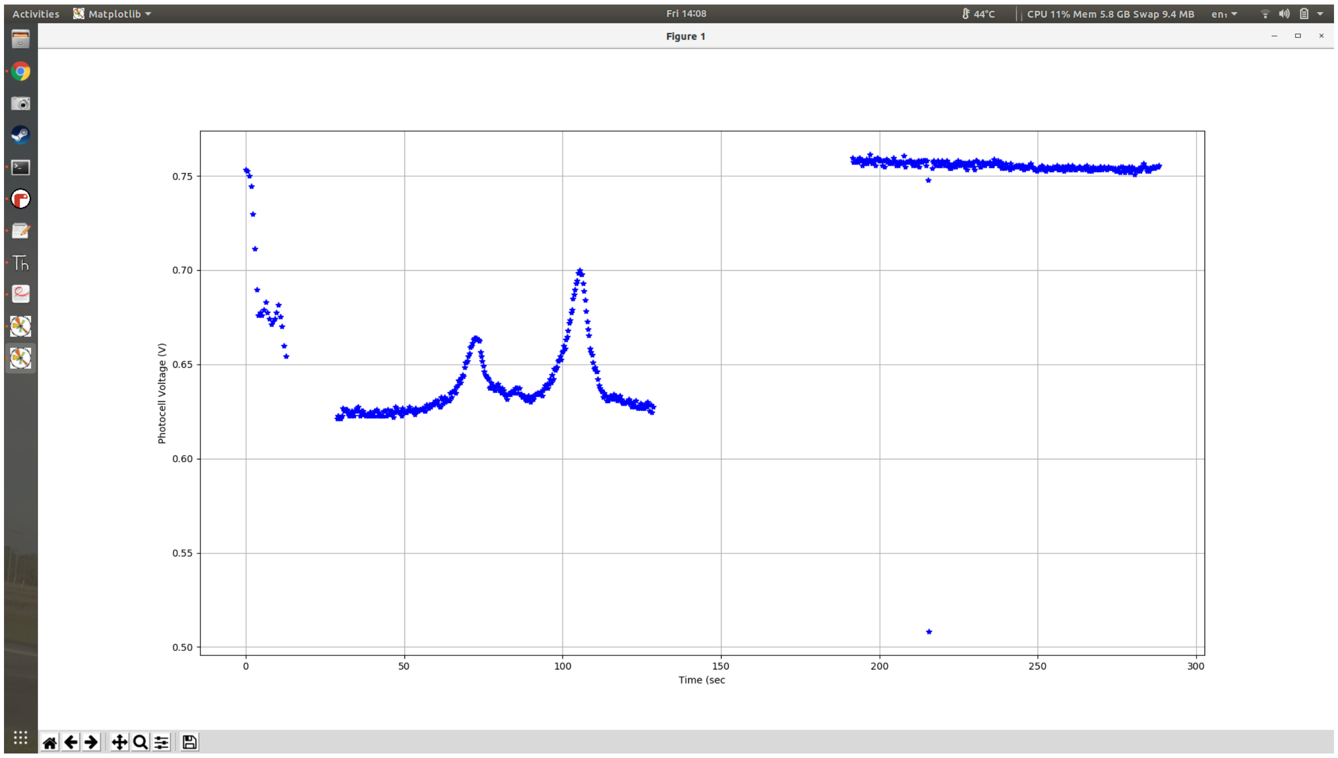
Finally convert to voltage

```
voltage = second_column*3.3/2.0**16
```

Then replace data in the rest of your code with voltage. Remember to remove st.mode since it does not work all ofthe time for continuous data sets.

4.4 Throwing Out Outliers

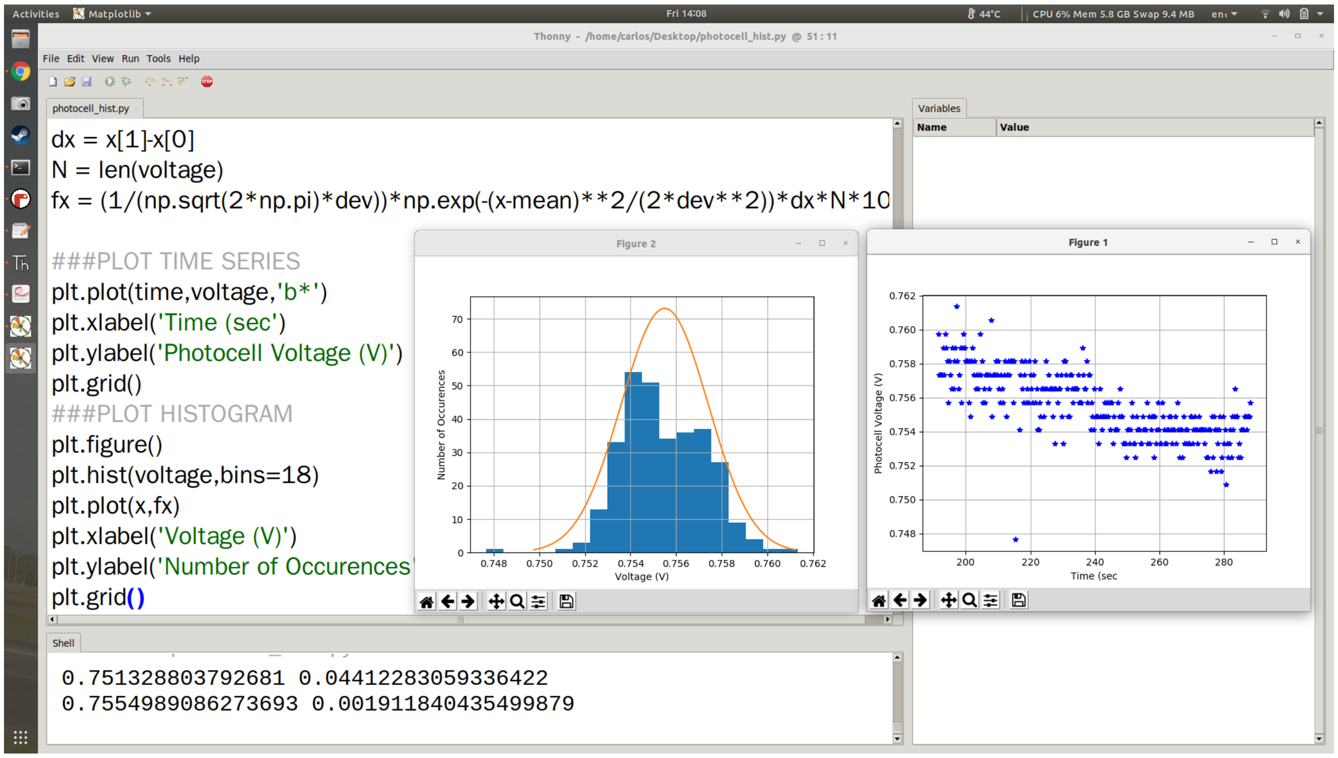
When I ran this experiment for a second time my CPX started and stopped 3 separate times. You'll see in the time series plot below that the voltage dipped in the first set and the second data set had some weird bumps probably from me changing tabs on my chrome tab. The photocell was close to my computer so that effected it. Thankfully the 3rd data set looked pretty good.



The only problem with the 3rd data set is that I put my hand over it for testing purposes. Because of that I had to remove those outliers. To do that I computed the current mean and standard deviation and then threw out all data points that were 3 standard deviations away from the mean. The code looks like this.

```
##COMPUTE CURRENT MEAN AND DEV

mean = np.mean(voltage)

dev = np.std(voltage)

print(mean,dev)

time = time[voltage > mean - 3*dev]

voltage = voltage[voltage > mean - 3*dev]

time = time[voltage < mean + 3*dev]

voltage = voltage[voltage < mean + 3*dev]

###COMPUTE NEW MEAN,STD

mean = np.mean(voltage)

dev = np.std(voltage) print(mean,dev)
```

Once I did all that clean up I was able to get a nice time series plot of my data.



```
dx = x[1]-x[0]
N = len(voltage)
fx = (1/(np.sqrt(2*np.pi)*dev))*np.exp(-(x-mean)**2/(2*dev**2))*dx*N*10

###PLOT TIME SERIES
plt.plot(time,voltage,'b*')
plt.xlabel('Time (sec)')
plt.ylabel('Photocell Voltage (V)')
plt.grid()
###PLOT HISTOGRAM
plt.figure()
plt.hist(voltage,bins=18)
plt.plot(x,fx)
plt.xlabel('Voltage (V)')
plt.ylabel('Number of Occurences'
plt.grid()
```

```
0.751328803792681 0.04412283059336422
0.7554989086273693 0.001911840435499879
```

4.5 Normal Distribution

I also was able to plot the Normal Gaussian Distribution on top of the histogram. You can see that in the left plot in orange. The code to do that is shown below where the 72 in the plot is the "height" of the histogram. Note that your histogram will have a different height and you will need to get that specifically from your plot.

```
###COMPUTE THE NORMAL DISTRIBUTION

x = np.linspace(-3*s+mu,3*s+mu,100)

pdf = 1.0/(s*np.sqrt(2*np.pi))*np.exp((-(x-mu)**2)/(2.0*s**2)) * (s*np.sqrt(2*np.pi)) * 72
```

The equations above make a time series from +-3 standard deviations from the mean and then plot the PDF of a normal Gaussian distribution. The only extra thing you have to do is multiply by (s*np.sqrt(2*np.pi)) * 72 which first causes the height of the PDF to be 1 and then multiply by 72 which again is the height of the histogram which will be different for your system.

4.6 Assignment

Turning in this assignment Once you've done that upload a PDF with all of the photos and text below included. My recommendation is for you to create a Word document and insert all the photos and text into the document. Then export the Word document to a PDF. For videos I suggest uploading the videos to Google Drive, turn on link sharing and include a link in your PDF.

4.6.1 Part 1

1. Include a video of you varying light conditions and watching the digital signal in the Plotter in Mu go up and down (make sure your face is in the video at some point and you state your name) - 50%
2. Include your Python code in the appendix along with your data plotted in a Figure. Make sure to plot the voltage and not the digital output - 50%

4.6.2 Part 2

1. Include a video of your circuit and explain how you captured the data to store on your computer (make sure your face is in the video at some point and you state your name) - 25%
2. Include the mean, median, and standard deviation of all light levels in volts - 25%
3. Include 3 histogram plots of low light, ambient light and high light levels. On top of the histogram I want you to plot the normal Gaussian distribution to see how close your histogram is to a Gaussian distribution - 50%