Project Based Engineering Instrumentation With CircuitPython

Carlos Montalvo*
University of South Alabama
Mobile, AL

Marine Feron Leabeater †
Georgia College and State University
Milledgeville, GA

Lisa Schibelius‡
Virginia Polytechnic Institute and State University
Blacksburg, VA

December 5, 2025

*Montalvo: William B. Burnsed Jr. Department of Mechanical, Aerospace and Biomedical Engineering, Associate Professor
†Leabeater: Lounsbury College of Education, EdD Graduate Student
‡Schibelius: Department of Engineering Education, PhD Graduate Research Assistant

Current Edition

This manuscript was last updated on December 5, 2025. The latest edition can be found on Github

Manuscript Changes

1. Original tutorials in Google Docs created
2. December 21st, 2021 - Updated links for manuscript and hardware
3. Tutorials purchased by Tangibles that Teach
4. May 30th, 2022 - Tangibles that teach went out of business and chapter began the move to Github
5. 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.
6. July 2nd, 2022 - All figures backed up and latest manuscript completed
7. October 18th, 2022 - A pedometer lab has been added.
8. November 8th, 2022 - Edited the Servo and feedback control servo lab to be one big lab with 3 parts
9. January 8th, 2023 - Removed all mentions of Tangibles that Teach in the main body of the text.
10. February 6th, 2023 - Changed an assignment description for LEDs and push buttons
11. February 22nd, 2023 - Moved the Bluetooth module to be before the modules lab so this will end up being a few updates to make those projects more uniform. I also created a backup called Bluetooth original in case you wanted to go back to the other version.
12. June 16th, 2023 - Cleaned up the "changes needed" list and updated Method 3 with some new software updates.
13. August 9th, 2023 - Edited the preamble for assignments and made a note in the servo lab
14. September 5th, 2023 - Updated hyperlinks to show up on Microsoft Edge using underlines and different color text.
15. September 7th, 2023 - Fixed the TL;DR section
16. September 15th, 2023 - Updated a link in the DAQ lab
17. September 20th, 2023 - Made the modules installation section a standalone tex file
18. October 4th, 2023 - Added a new potentiometer photo
19. November 16th, 2023 - Edited the lists of parts
20. November 22nd, 2023 - Edited acceleration lab
21. December 5th, 2023 - Edited the troubleshooting section
22. January 12th, 2024 - Edited chapters 1 and 2
23. January 22nd, 2024 - Edited the bootloader update notes in the troubleshooting chatper
24. January 25th, 2024 - More edits to the bootloader notes
25. March 20th, 2024 - Added a preamble about where this textbook is located
26. April 27th, 2024 - Previously edited servo lab and pendulum lab just the assignment portion and then added this change log to the manuscript
27. May 7th, 2024 - Title page changed and many sections moved around. Added a new results and discussion section following data analysis from course surveys of this course.
28. August 12th, 2024 - Added a new requirement for the servo lab
29. September 12th, 2024 - A few edits to assignment descriptions
30. October 25th, 2024 - Updated the photocell lab to include the number of data points for the histogram
31. November 11th, 2024 - Updated Method 3 quick list
32. March 26th, 2025 - Updated servo, pitot probe and photocell assignments
33. May 24th, 2025 - Made a few changes to chapters 2 and 19 to transition between the projects and the results and discussion about lab at home frameworks and project based teaching. Also overhauled the feedback control project.
34. June 16th, 2025 - Updated the assignment directions for all projects.
35. June 17th, 2025 - Added more theory to the photocell, thermistor and pitot probe assignment
36. June 20th, 2025 - Added more figures and wiring diagrams to many assignments to help students wire up their circuits.
37. June 21st, 2025 - Added a few problems from a standard instrumentation textbook. Probably need more statistics questions
38. July 2nd, 2025 - Moved changes needed to the preamble. Added learning objectives and parts list for a few projects that had them missing.
39. September 26th, 2025 - Updated the potentiometer assignment to include converting voltage to angle in degrees.

40. October 1st, 2025 - Updated the thermistor modules assignment with the thermistor equations so it's easy to find in two places.
41. November 12th, 2025 - Fixed an equation, link and figure in the feedback control assignment
42. November 14th, 2025 - Fixed a broken link
43. November 20th, 2025 - Added a GPS requirement to the integrating acceleration lab
44. December 5th, 2025 - Added the settling time to the list of parameter for the time constant assignment and edited the overall assignment directions to have a better description of the video requirements. Also edited the servo second order systems equation for an aircraft to be stable since the equations were unstable open loop. I also moved the changes needed to Github. I also edited the purchase assignment to have bold text for some important information.

Future Project Ideas

1. PhotoInterrupter Lab (Angular Velocity) - 3D print a disc with holes on the outside to eventually mount to a rotating shaft
2. Load Cell Calibration Lab - load cells with the HMC converter
3. Sunlight/Clouds - Take temperature and light data on a cloudy day. Draw correlations between sunlight and temperature
4. Sunrise/Sunset - Take temperature and light data over the course of a whole day
5. PLC Lab - Get a PLC kit and do something with it

Changes Needed

Needed changes are now tracked on Github

Contents

1 Introduction

This textbook describes an instrumentation kit used for an Instrumentation and Experimental Methods course at the undergraduate level. This textbook has been designed with the student and faculty member in mind. The kit contains the CircuitPlayground Express/Bluefruit running CircuitPython and is used to teach fundamentals of instrumentation and provide a hands-on way of learning. Engineering is usually taught in a traditional lecture format, involving theory in the classroom, homework outside of class, and routine examinations. Progressive forms of learning such as flipped classrooms and project based learning (PBL) have created new and fun ways for professors to interact with students and for students to be more involved in their learning. PBL provides a student-centered method of teaching and learning by posing problems for students to solve with the solution of the project being the primary goal and the theory a secondary goal.

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. The authors hope that whomever reads and learns from this textbook will walk away with an excitement to tinker, code and build future projects using microelectronics and programming. The implementation of this kit and overall teaching method has received many positive reviews from students and is reflected in anonymous course evaluations.

Students can learn in many ways with a variety of different omodalities [1]. As such, instructors can choose how to present course material and have students develop content-specific skill sets. College students enter the classroom with existing skills in multi-disciplinary learning that is practiced at the secondary school level [2]. College and university professors have the opportunity to use these existing skills as a foundation for their own instructional practices. Traditional lectures provide students with lectures on theory. The instructor then provides examinations to the students based on that theory. Using this format, an instructor can focus on mathematical principles, the so-called building blocks of engineering or provide specific applications to these models [3]. In a traditional lecture format, the instructor presents theory to the students with academic problems rarely encountered in a students future career, building invaluable theoretical knowledge of concepts that appear disconnected from practical knowledge applied in the field [4]. Historically in engineering education, there has been tension between theory and application [5]. Applications include case studies or practical engineering problems that emulate future careers in engineering. This provides students with engineering phenomena that they can see and hear. They can reason through problems intuitively, memorize facts using demos, build mathematical models, or build tangible objects themselves. With these different presentation methods, students are exposed to engineering phenomena in multiple ways rather than just on the whiteboard[3]. Project Based Learning (PBL) provides an alternative for conventional teaching by facilitating problem solving for students in a group setting that requires communication, critical thinking, and creativity. These types of problems have shown that heightened learning can happen when students interact with tangible objects that represent the theory they are presented with in the classroom [6]. Rather than learning an equation for heat transfer along a one-dimensional pipe, the students can create a pipe with thermocouples and plot data from multiple sensors along the pipe. This connection to a tangible object reinforces learning and allows students to form bridges in comprehension between their theoretical knowledge and practical application (practical knowledge) of those theories in the field[7]. Research has shown that a classroom that creates an integrated curriculum increases student satisfaction which immediately correlates to satisfactory student performance and increased graduation rates [8]. This is an example of the benefits of active student engagement versus passive student engagement. While the traditional lecture format is vital to students understandings, it is understood that application of the concepts covered in lecture are required to reach higher levels of learning [9].

Hands-on projects where students interact with tangible objects is not a new form of teaching. The laboratory environment has been around for decades. However, typically the classroom environment is separate from the laboratory environment. There has been little synergy between the two learning environments. A Mechanical

Engineering degree is likely to have around three to four labs in various disciplines. In these courses, students perform an experiment in groups. They take and analyze data as well as create a report documenting their findings. Many of these laboratory experiments of course are choreographed for the student by the instructor. They follow a script and perform the experiment without building anything themselves. The students take no ownership over the experiment and there is no creativity built-in. Over the years, the nature of laboratories has changed including the lack of clear learning objectives [10]. Furthermore, this laboratory environment requires a significant amount of instrumentation and hardware to implement. For example, a shock tube or steam pump can cost tens of thousands of dollars. The maintenance and up-keep alone is not practical for smaller institutions. The teaching investment required to prepare the lab every year and the lab itself (often on the order of three hours) can be a time consuming task for a tenure track faculty member who often has a large research load.

The so-called "Lab at Home" hardware kits are becoming more and more common in the classroom to ease the burden on the instructor and institution itself[11]. These kits are small enough to be purchased and shipped to a student for them to perform experiments remotely rather than in the classroom itself[12]. They can also be brought to the classroom to be used as a personal demo aid[13]. In this sense, the take-home lab kit serves as a bridge between theory based lectures and a laboratory setting[14]. This allows both instructors and students to engage with content in a way that promotes enduring understandings and practical application of theoretical knowledge[15]. Since the kits can be used remotely, they can also be used for distance learning courses or other asynchronous activities as well as during the COVID-19 pandemic.

Two home kits in particular are directly related to instrumentation and circuits. Cyganski and Nicoletti[15] for example created a new curriculum for first year electrical engineering students by creating live demonstrations for the students. Manijikian and Simmons[13] however, combined a popular commercial microcomputer board with their own custom-designed interface board in a kit that students retain for the duration of the course for both in-lab and at-home assignments. The take-home kits however were based on the Motorola 68HC11EVB microcomputer which is programmed in assembly language. Assembly language is a rather difficult language to learn at the undergraduate level. Even with such a difficult learning curve, however, student feedback on their approach was positive[13].

It is clear that using a lab at home kit is useful for the instrumentation classroom, however the programming language to be used is a subject of debate in faculty meetings and computing committees. There are multiple languages in use today of varying complexities and use cases. There are scripting languages like Python, Ruby and MATLAB, object oriented languages like Java and C++ and compilation languages like Fortran and C. Note, this is not an exhaustive list. A recent study showed that scripting languages (Python, Ruby, MATLAB) enable writing more concise code while compilation languages (C, Fortran) create smaller executables [16]. MATLAB is often considered in engineering given its success in industry and its ability to perform numerical simulations and plotting with ease. Python, however, has become more and more popular with numerical toolboxes like numpy and plotting toolboxes like matplotlib that are free and easy to install in integrated development environments like Thonny or Spyder [17]. The Tiobe Index of Programming[18] has the top 3 programming languages listed as Python, C, and Java. MATLAB is #20 on the list. In 2004, Hans Fangohr wrote that MATLAB is much better suited than C for engineering computing but the best choice in terms of clarity and functionality is provided by Python [19]. It seems then that it would be more practical for educators even in engineering to teach the most popular language. This helps with transferability of skills and has future implications for a students' career. The scripting aspect of Python lowers complexity and allows students to learn the language quickly to apply it as a tool rather than getting stuck memorizing syntax and compilation rules. The language also comes at no cost to the students. This is a plus for students already bearing heavy financial burdens to attend a university which includes tuition and institutional fees.

Given the success of other kits in many classes and the popularity of Python, the University of South Alabama (USA) has implemented such a kit for Instrumentation & Experimental Methods (ME 316) using CircuitPython. CircuitPython is a derivative of Python written for embedded systems and designed to simplify experimenting and learning to code [20]. The learning objectives for ME316 include: statistics, dynamic response of measurement systems, operational amplifiers, signal conditioners and fundamentals of microprocessors. This course could be taught with theory as the main focus of the lecture. However, this course includes the use of an instrumentation kit and Project Based Learning methods in addition to theory in the classroom. Each student taking the course purchases the kit and downloads a free accompanying project list [21]. Every Friday, the students complete one of the multiple projects. The following week they submit a report which includes a demo of the working project in the form of photos and videos, any plots generated if they are required to take data, all code used and a small write-up explaining their findings. The sections that follow describe the kit in more detail as well as student evaluations who have taken the course.

Note that, the Adafruit Learn page contains many tutorials on how to operate the CircuitPlayground[22].

However, Adafruit sells much more than just the CPX and thus it is often difficult for anyone to find the correct tutorial needed for the CPX. A simple search on the Adafruit Learn System yields 21 results for "servo" just on the first page with over 48 pages of results. The tutorial needed to run a servo with CircuitPython is the 10th result and it's for a different development board. Although the software works with the CPX, it does not explicitly say so. Due to the complexity of some of these systems, documentation was made and freely given to the community[21]. This documentation was custom built using a combination of multiple sources across the internet. All software for the kit is also on Github in a separate repository. Each chapter in the book contains one project for the students to complete on their own. A list of chapters and a brief description of the assignment is shown in the table of contents. Currently there are about 20 projects with multiple subsections for the students to work on during the "Funday Friday" lab days.

2 Course Description

The course is designed to coincide with lecture content in a standard engineering instrumentation course. The students are guided through numerous projects. The specific course learning objectives, which include statistics, dynamic responses, operational amplifiers, and electronic circuits both analog and digital are shown below:

1. Apply statistical concepts of error and uncertainty analysis using normal,"t" distribution and the χ^2 distribution
2. Use propagation of error methods to determine the uncertainty of calculated quantities
3. Apply the concepts of harmonic response to predict the response of measuring systems to input signals
4. Apply the fundamentals of operational amplifiers and electronic circuits to design signal conditioning circuits including
5. Apply microprocessor fundamentals to gain an understanding of digital circuits used for digital to analog and analog to digital conversion

This course is not going to be a traditional lecture format and that hands-on projects will be assigned every Friday using the lab at home kit. The projects every Friday are created to enhance the course learning objectives as dictated above.

The course is taught on a 2+1 style where the course meets three days a week for 50 minutes each on Monday, Wednesday and Friday. On Monday and Wednesday the class is similar to what would typically be found in a standard lecture format. The instructor goes through theory and examples on a white board. On Fridays, the students come to class with their hardware kits and laptops to perform one of 20+ projects defined in the tutorials in this textbook. Fridays are nicknamed "Funday Fridays" to differentiate the format of the class and to get students excited about the projects. On Fridays, the instructor can walk around the classroom and assist students on their projects as well as highlight fundamental concepts. This is done by showing them what they learned on a whiteboard with their lab at home kit. During the COVID-19 pandemic when Universities were meeting via Zoom, class time on Friday was used to assist students on their projects while they showed their circuits via webcam in a virtual space. Since all students purchased their own kit, there was no issue with students getting hands-on experience even during COVID-19.

It is important to note here that Project Based Learning (PBL) in this course did not replace existing teaching frameworks; rather, PBL supplemented existing standard teaching measures by allowing students to apply theory to practice. The role of connectivism comes from the online networking tools that were added to the curriculum. The tutorials and assignments as well as example code are all hosted online on Github [23]. The availability of these tutorials and software opens the students to an ecosystem of free and open source software. Furthermore, the Adafruit Learning System is also used occasionally which opens them to an ecosystem of hardware and software tutorials as well as Adafruit IO (an internet of things site) and the Adafruit Forums [22]. The Adafruit Forums allow the students to comment and communicate with other groups outside the university that utilize the same hardware kits. If the instructor cannot find a solution online in a Wiki or Tutorial page, a student or the instructor can post on the forum and wait for a response from the larger Adafruit community.

Students in this course are also required to do a final project where they must create something new by using the microcontroller in the kit plus three other electrical components. One of the three components must be something not originally included in the kit. The students may also work in a group. PBL typically consists of ownership, creativity, collaboration and critical thinking. These four aspects are clearly a big part of this lab at home kit and the final project. They can work in groups: collaboration. They must make something new: creativity. They must build the item themselves: ownership. They must apply fundamental principles of the course: critical thinking. In addition to PBL aspects, the students are exposed to the larger network of learning through Github, the Adafruit

Learning system and Forums as well as general Wikis online. This network of learning is seen in the connectivism framework. The chapters that follow detail the weekly projects. The chapters that follow the weekly projects explain results from anonymous course surverys from students taking the class as well as conclusions drawn from the authors and faculty who have taught the course.

3 Purchase Equipment

3.1 Parts List

1. Laptop with internet connectivity
2. Credit/Debit Card to Purchase items

3.2 Learning Objectives

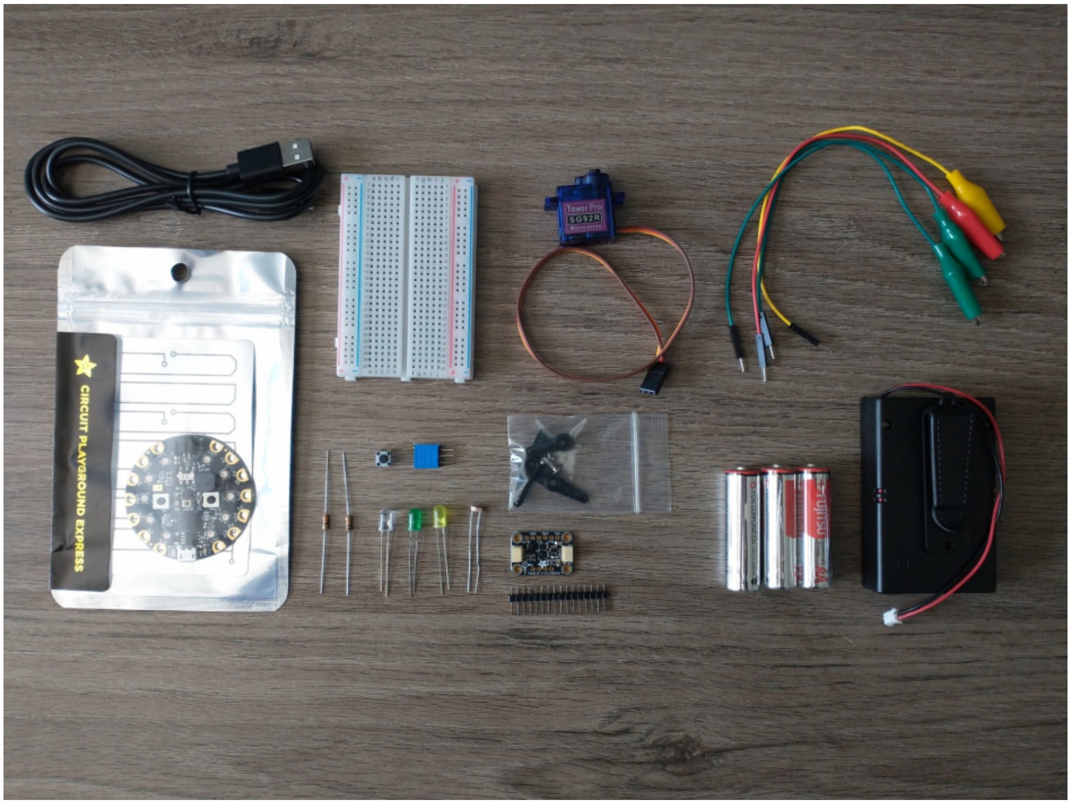
1. Understand the world of microcontrollers
2. Learn about the Adafruit eco-system
3. Learn about the Circuit Playground Express/Bluefruit

3.3 CircuitPython Kit

In this class you're going to build some circuits that will enhance your learning experience. Rather than just solving problems by hand you're going to take and analyze data. Over the summer of 2020, I began to work with Tangibles that Teach and at the time they bundled all components together. Unfortunately, that company has gone out of business and as such for the time being you must build the kit yourself. Here is a list of all the components in the kit

1. Circuit Playground Bluefruit or Express + Included USB Cable
2. Micro Servo
3. Potentiometer
4. Photocell
5. Two Resistors (330 Ohm and 1K Ohm)
6. Alligator Clips x3
7. External Battery Pack
8. AAA Batteries x3
9. Breadboard
10. Push Button
11. LEDs x2
12. Pitot Probe
13. LSM6DS33 + LIS3MDL - 9 DoF IMU (Optional)

Below is also a photo of all the components listed above. When you get your kit familiarize yourself with all of the components. I created an unboxing video on Youtube for you to take a look. The hardware kit is a collection of multiple items. At the time of this writing the entire kit cannot be purchased however the CircuitPlayground Express (CPX) can be purchased on Adafruit[24]. A photo of the kit is shown in the figure below. The photo shows the CPX on the left inside it's protective zip lock bag.



The CPX is a relatively new development board that operates in the same style as Arduino[25]. That is, the student can create a relatively simple script and then flash the software to the CPX via USB. The Arduino operates in the same way. The benefit of the CPX vs the Arduino is that the software is CircuitPython which is indistinguishable from Python. As mentioned earlier, Python is in the top 3 programming languages as stated by the Tiobe Index of Programming[18]. Other benefits include the components built in to the CPX. The CPX itself contains a built in microphone, accelerometer, light sensor, speaker, three push buttons, slide switch and infrared sensor. The CircuitPlayground Express (CPX) also contains 10 smart NeoPixel LEDs and a ring of copper plated pads that can be attached to the included alligator clips and breadboard. Students can also elect to purchase the CircuitPlayground Bluefruit (CPB) which does not have an infrared sensor but does contain bluetooth functionality to send data wirelessly to the Adafruit Connect App which runs on Apple/Android[26].

3.4 Assignment

Your assignment for this module is to purchase the required equipment for your class. **Note that the specific equipment will be instructor dependent so be sure you discuss with the instructor of your course before buying everything.** At a minimum you must purchase a CPX or CPB but the rest of the items in your kit will depend on which modules your instructor wants you to do during the semester. **Also, the CPX/CPB requires a microUSB cable and that cable must have a data line. Many USB cables are just power and ground.**

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%

5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. A figure of your receipt of your purchases this semester (If you already own the CPX/CPB you may just take a photo of you holding the device) -60%
2. Using the number of students in your class, compute the probability that at least one CPX/CPB will be shipped DOA (dead on arrival) assuming a failure rate of 1%. - 20%

4 Download Python for Desktop

4.1 Parts List

1. Laptop
2. Python Integrated Development Environment (IDE) (Thonny or Spyder - Do not use Python IDLE)

4.2 Learning Objectives

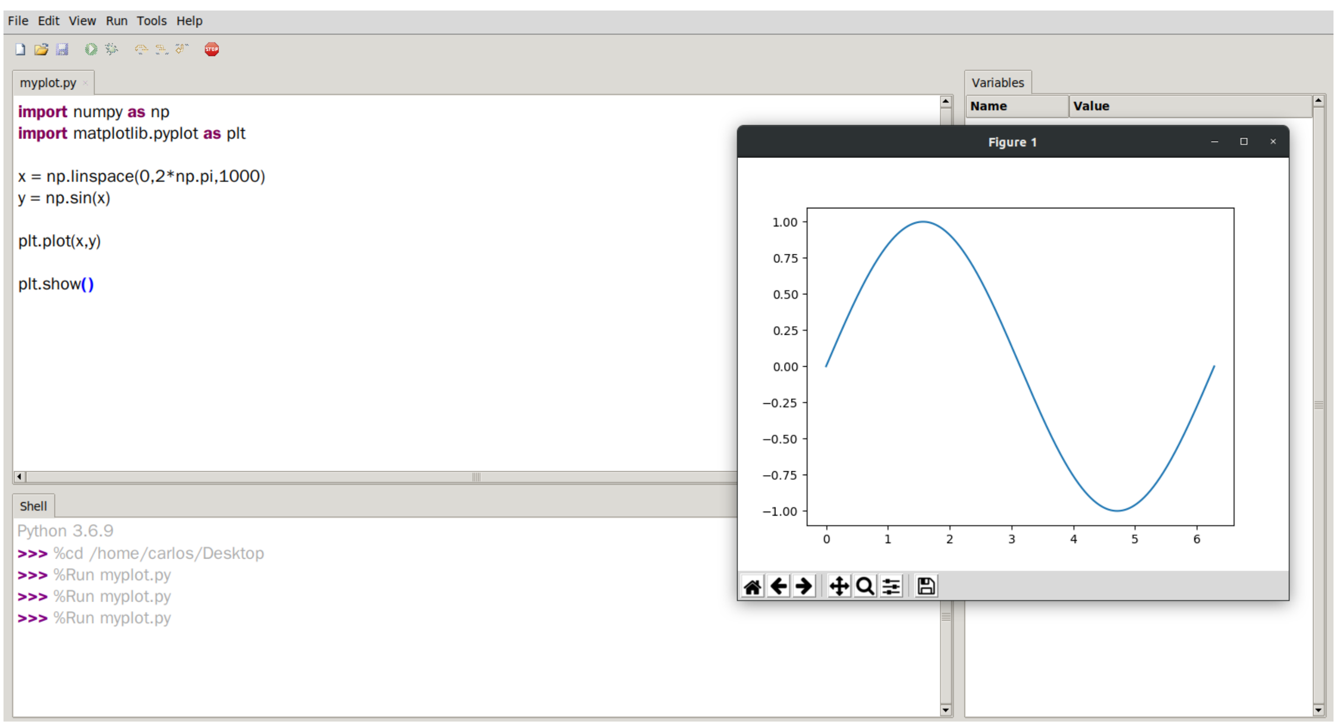
1. How to download and use a Python IDE (Thonny or Spyder is suggested. Do not use Python IDLE)
2. Plot a simple function in Python

As you learn Instrumentation throughout the semester, you will be tasked with creating computer programs on the Circuit Playground Express (CPX). The CPX itself has it's own RAM, CPU, HDD and many sensors. Your CPX is kind of like a mini computer! You can plug the CPX into your computer via USB and access the hard drive (HDD) from your own computer. When you program on the CPX you need to write programs on the CPX itself so that the mini computer can run the program you wrote. The CPX knows how to read multiple different languages but in this class we are going to write everything in the Python language which has been ported to the CPX and called CircuitPython. Since we have to write everything in CircuitPython we need to first learn how to program some things in Python. You can easily download Python by itself but it's nice to get what's called an Integrated Development Environment (IDE). This way you can practice writing Python code on your computer while you wait for your purchases to arrive in the mail.

So which IDE can you download and which is recommended? I recommend two IDEs. They are listed below. I recommend getting either one. If you just Google "Python download" you will find a humongous list of editors (Scratch, Anaconda, Canopy, Eclipse, PyDev, etc). It's easy to get lost when searching for something so broad. You've been warned.

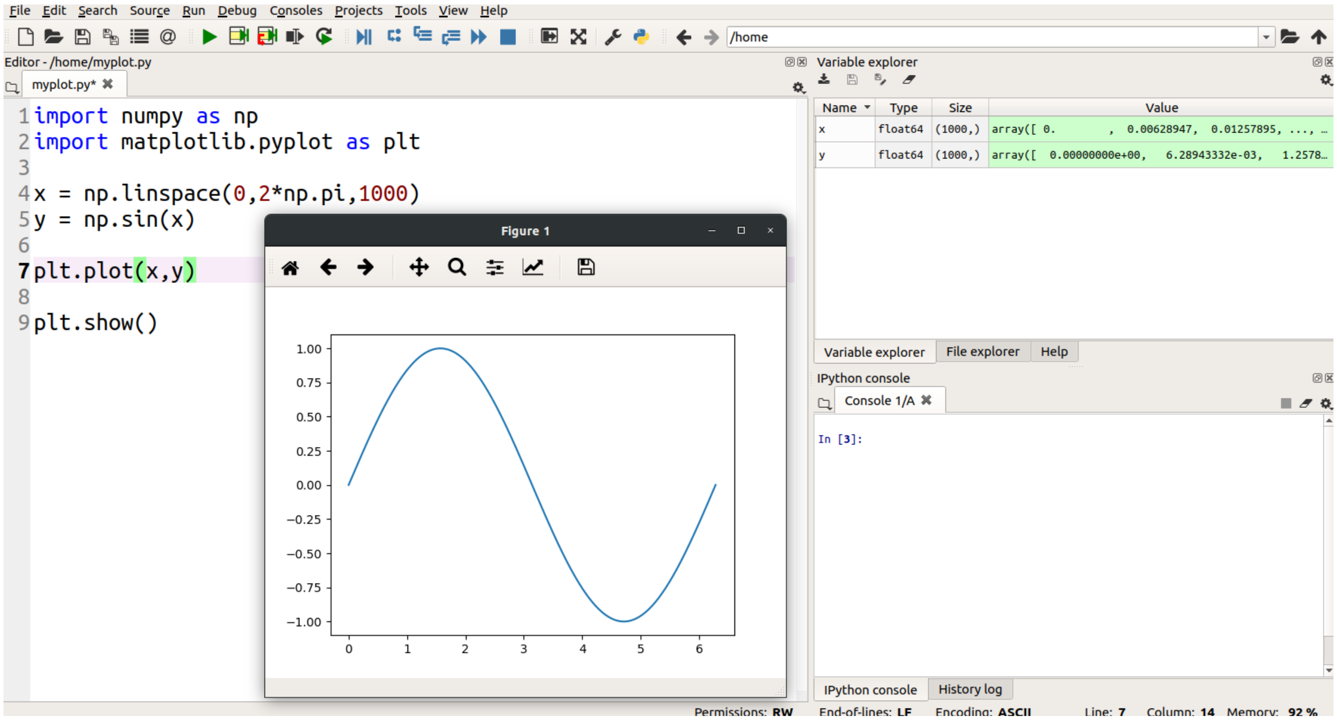
4.3 Thonny

Thonny - `https://thonny.org/` - Youtube video on how to install



4.4 Spyder

Spyder - `https://www.spyder-ide.org/` - Youtube video on how to install

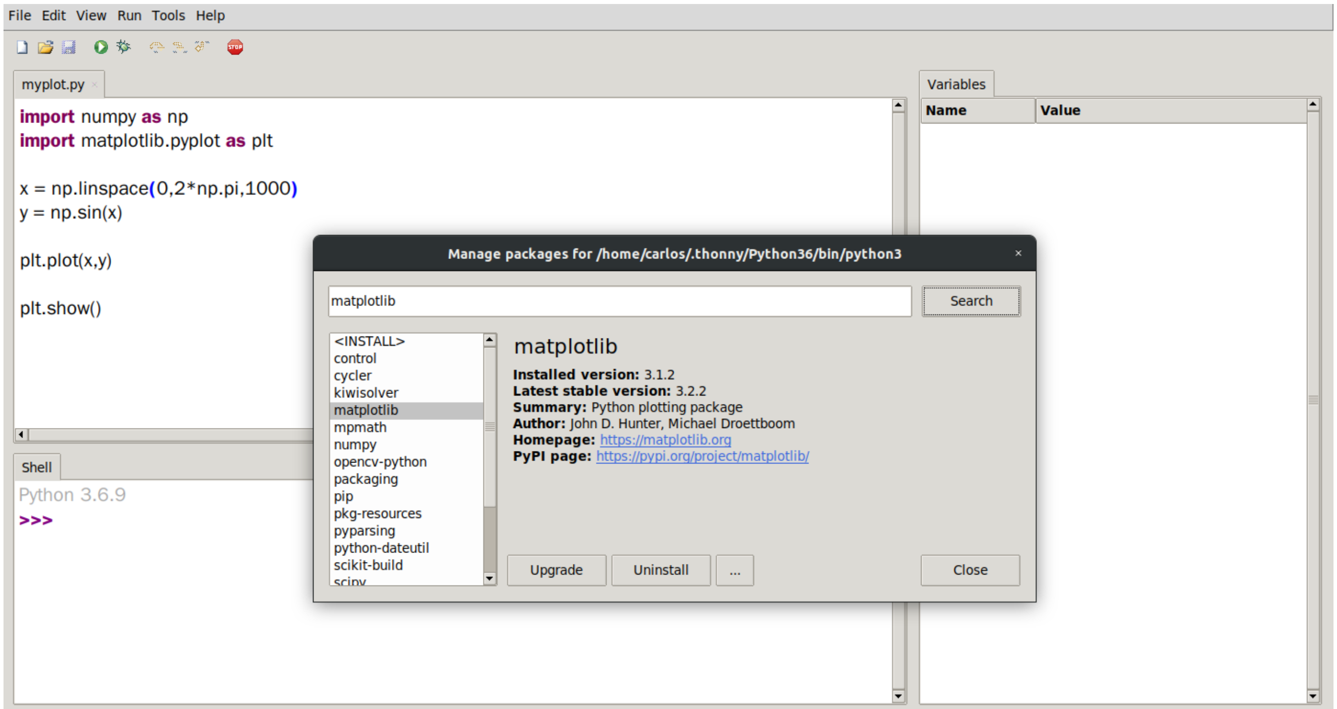


4.5 Other Options

It is possible to use Google Colab if you want to collaborate on Python projects or even get apps for your phone (Pydroid or Pythonista depending on Android or iPhone). You'll need to download 32 bit or 64 bit but which one? Well you need to figure out how many bits your computer has. This is a great thing to Google. Type the following: "do I have a 32 bit or 64 bit computer" into Google. I'm willing to bet you have a 64 bit computer but you may as well check. We'll learn about the difference between 32 and 64 bit computers when we get to the projects on Binary.

4.6 Setting up your IDE

Once you have Thonny or Spyder installed you need to install numpy and matplotlib which are modules within Python that allow us to do some extra things like numerical computation with Python (numpy) and Matlab style Plotting libraries (matplotlib). I explain how to install modules in my Youtube videos above; however, you need to head over to Tools¿Manage Packages in Thonny. You can see in the image below I already have version 3.1.2 but I can upgrade to 3.2.2



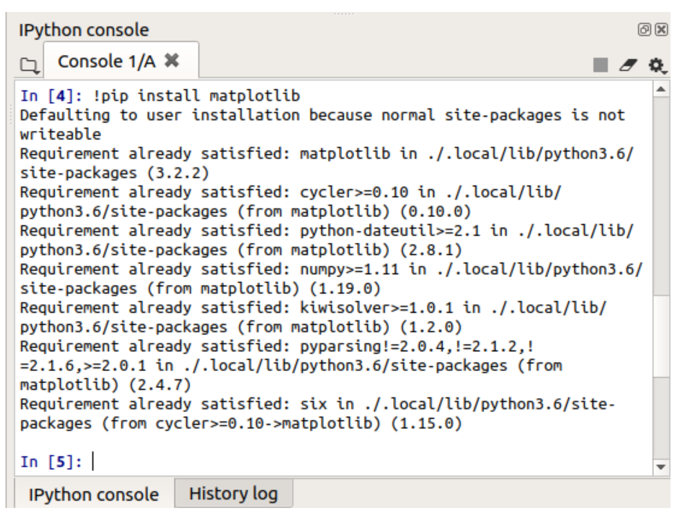
If numpy or matplotlib is not already included in Spyder then you need to type the following into the Python Console in the lower right hand corner of Spyder which is called the IPython console.

```
!pip install matplotlib
```

If that doesn't work try

```
!pip3 install matplotlib
```

You can see in the output example below that I already have matplotlib installed as it says "requirement already satisfied". Assuming you have a valid internet connection it will install the necessary module.



4.7 Scripting

Once you have numpy and matplotlib it's time to make a plot. I have a pretty comprehensive youtube video on how to plot in matplotlib but if you prefer text I will walk through a simple example.

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

x = np.linspace(0,2*np.pi,1000)
y = np.sin(x)

plt.plot(x,y)

plt.show()
```

The code above will plot a sine wave from 0 to 2pi. The two lines at the top are importing the numpy and matplotlib modules you installed earlier. When they are imported we give them shorter names so it's easier to reference them so numpy will now be called np and matplotlib.pyplot will be called plt. The next two lines then create a vector "x" from 0 to 2pi using 1000 data points. The next line then uses the sine function to create the vector "y". Finally "x" and "y" are plotted and the figure is instructed to pop up on your screen using the show() function.

4.8 Built-In Help Function and dir()

Running code will always create syntax errors. Typing your syntax error into Google will yield so many results you might get lost. Sometimes it helps to know how to learn things just from your computer. For example, type in the commands below in the IPython console or the Shell.
import numpy as np
dir(np)

```
>>> import numpy as np
>>> dir(np)
['ALLOW_THREADS', 'AxisError', 'BUFSIZE', 'CLIP', 'ComplexWarning', 'DataSource', 'ERR_CALL', 'ERR_
DEFAULT', 'ERR_IGNORE', 'ERR_LOG', 'ERR_PRINT', 'ERR_RAISE', 'ERR_WARN', 'FLOATING_POINT_SUPP
ORT', 'FPE_DIVIDEBYZERO', 'FPE_INVALID', 'FPE_OVERFLOW', 'FPE_UNDERFLOW', 'False_', 'Inf', 'Infinit
y', 'MAXDIMS', 'MAY_SHARE_BOUNDS', 'MAY_SHARE_EXACT', 'MachAr', 'ModuleDeprecationWarning', '
NAN', 'NINF', 'NZERO', 'NaN', 'PINF', 'PZERO', 'RAISE', 'RankWarning', 'SHIFT_DIVIDEBYZERO', 'SHIFT_I
NVALID', 'SHIFT_OVERFLOW', 'SHIFT_UNDERFLOW', 'ScalarType', 'Tester', 'TooHardError', 'True_', 'UFU
NC_BUFSIZE_DEFAULT', 'UFUNC_PYVALS_NAME', 'VisibleDeprecationWarning', 'WRAP', '_NoValue', '_U
FUNC_API', '__NUMPY_SETUP__', '__all__', '__builtins__', '__cached__', '__config__', '__doc__', '__file
__', '__git_revision__', '__loader__', '__name__', '__package__', '__path__', '__spec__', '__version__',
'_add_newdoc_ufunc', '_distributor_init', '_globals', '_mat', '_pytesttester', 'abs', 'absolute', 'absolute
_import', 'add', 'add_docstring', 'add_newdoc', 'add_newdoc_ufunc', 'alen', 'all', 'allclose', 'alltrue', '
amax', 'amin', 'angle', 'any', 'append', 'apply_along_axis', 'apply_over_axes', 'arange', 'arccos', 'arcc
osh', 'arcsin', 'arcsinh', 'arctan', 'arctan2', 'arctanh', 'argmax', 'argmin', 'argpartition', 'argsort', 'argw
here', 'around', 'array', 'array2string', 'array_equal', 'array_equiv', 'array_repr', 'array_split', 'array_str',
'asanyarray', 'asarray', 'asarray_chkfinite', 'ascontiguousarray', 'asfarray', 'asfortranarray', 'asmatrix', '
asscalar', 'atleast_1d', 'atleast_2d', 'atleast_3d', 'average', 'bartlett', 'base_repr', 'binary_repr', 'binco
unt', 'bitwise_and', 'bitwise_not', 'bitwise_or', 'bitwise_xor', 'blackman', 'block', 'bmat', 'bool', 'bool8',
 'bool_', 'broadcast', 'broadcast_arrays', 'broadcast_to', 'busday_count', 'busday_offset', 'busdaycalen
dar', 'byte', 'byte_bounds', 'bytes0', 'bytes_', 'c_', 'can_cast', 'cast', 'cbrt', 'cdouble', 'ceil', 'cfloat', 'c
har', 'character', 'chararray', 'choose', 'clip', 'clongdouble', 'clongfloat', 'column_stack', 'common_typ
```

I included a photo of the output from the dir function. You'll notice there are a ton of functions in numpy. Every function in Python has a

```
__doc__
```

function. That's two underscores followed by "doc" and then another two underscores. If you're ever curious about what a particular function does you can just run the command below again in the IPython console or Shell. In this example I'm looking at what *arctan2* does.

```
print(np.arctan2.__doc__)
```

```
>>> print(np.arctan2.__doc__)
arctan2(x1, x2, /, out=None, *, where=True, casting='same_kind', order='K', dtype=None,
subok=True[, signature, extobj])

Element-wise arc tangent of ``x1/x2`` choosing the quadrant correctly.

The quadrant (i.e., branch) is chosen so that ``arctan2(x1, x2)`` is
the signed angle in radians between the ray ending at the origin and
passing through the point (1,0), and the ray ending at the origin and
passing through the point (`x2`, `x1`).  (Note the role reversal: the
"`y`-coordinate" is the first function parameter, the "`x`-coordinate"
is the second.)  By IEEE convention, this function is defined for
`x2` = +/-0 and for either or both of `x1` and `x2` = +/-inf (see
Notes for specific values).

This function is not defined for complex-valued arguments; for the
so-called argument of complex values, use `angle`.

Parameters
----------
x1 : array_like, real-valued
    `y`-coordinates.
x2 : array_like, real-valued
    `x`-coordinates. If ``x1.shape != x2.shape``, they must be broadcastable to a commo
n shape (which becomes the shape of the output).
out : ndarray, None, or tuple of ndarray and None, optional
```

You'll see that arctan2 takes 2 input arguments "x1" and "x2". I didn't include the entire output but if you continue to scroll through the output it will even include examples on how to use the function.

Another way to learn certain functions is by visiting the appropriate documentation. The Numpy Docs website for example has all the documentation you need for Numpy. Navigating that website you can find the same documentation for arctan2.

As a last resort you can always Google "how to compute the inverse tangent 2 function in Python". Note though that there is so much content out there on Google that you could easily get lost. Still, there's also so much information that the answers are out there for just about anything.

So you have three methods for finding out how to program in python. The dir and *__doc__* functions in Python, using the appropriate documentation online and of course Google. I'm lumping Youtube in with Google which is also another way to learn information although when I want to find information quickly I just use the documentation. It's the best in my opinion.

4.9 Assignment

Your assignment for this project is to plot the equation below from 0 to 10 seconds and include that Figure in your report. You must add a grid and label the x-axis 'Time (sec)' and the y-axis 'Temperature (F)'

$$T(t) = 60(1 - e^{-5t}) + 30 \tag{1}$$

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%

4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%
5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. Include a screenshot of your Python IDE (Thonny or Spyder is suggested. Do no use Python IDLE) with your code pasted into the workspace - 40%
2. Include the plot of temperature vs time being sure to save the figure so it is in high resolution - 40%

5 Getting Started with the CPX/CPB

5.1 Parts List

1. Laptop
2. Mu
3. CPX (or CPB) UF2 File
4. CPX(or CPB)
5. USB Cable (with a data line. Not all USB cables have data lines)

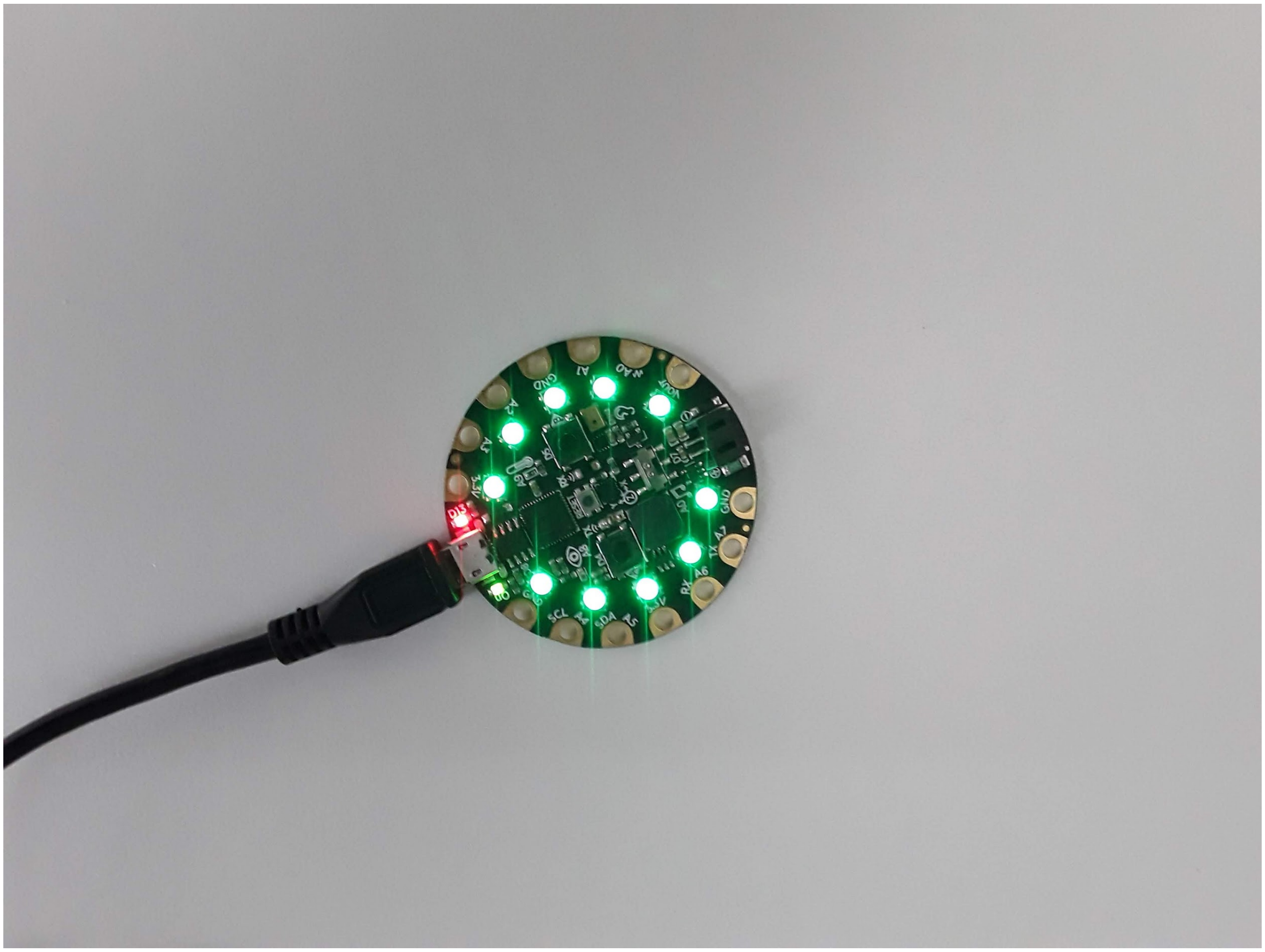
5.2 Learning Objectives

1. Learn how to setup your Circuit Playground Express
2. Learn how to install Mu
3. Learn how to code with Mu
4. Learn about hardware and software debugging
5. Learn the difference between Python and CircuitPython

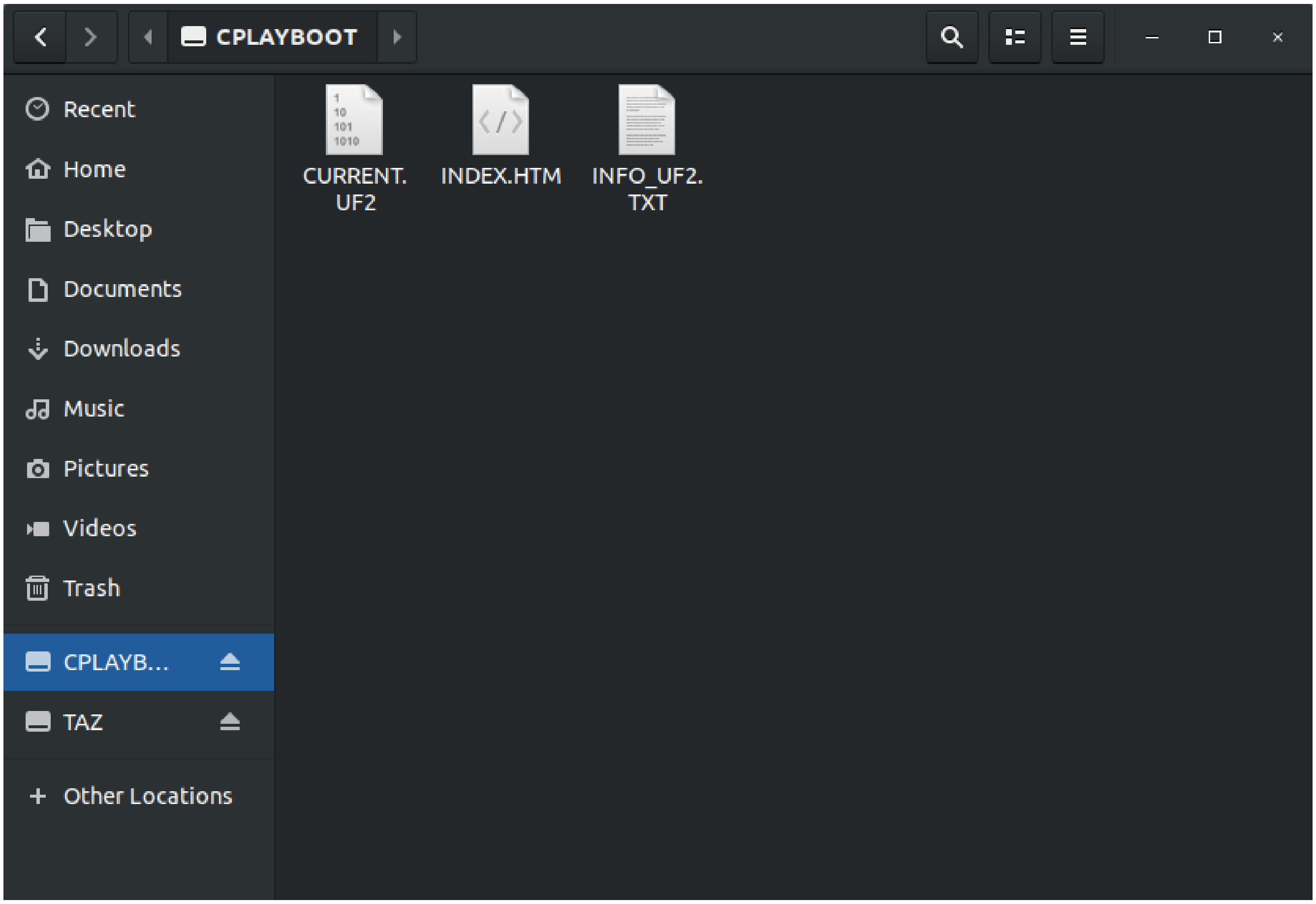
5.3 Setting up your Circuit Playground

By now you hopefully have your Circuit Playground (CPX) and it's time to get your CPX up and running. A very in depth and detailed tutorial can be found on the Adafruit Learn site. The text below is a summary of what you need to do to get the CPX up and running.

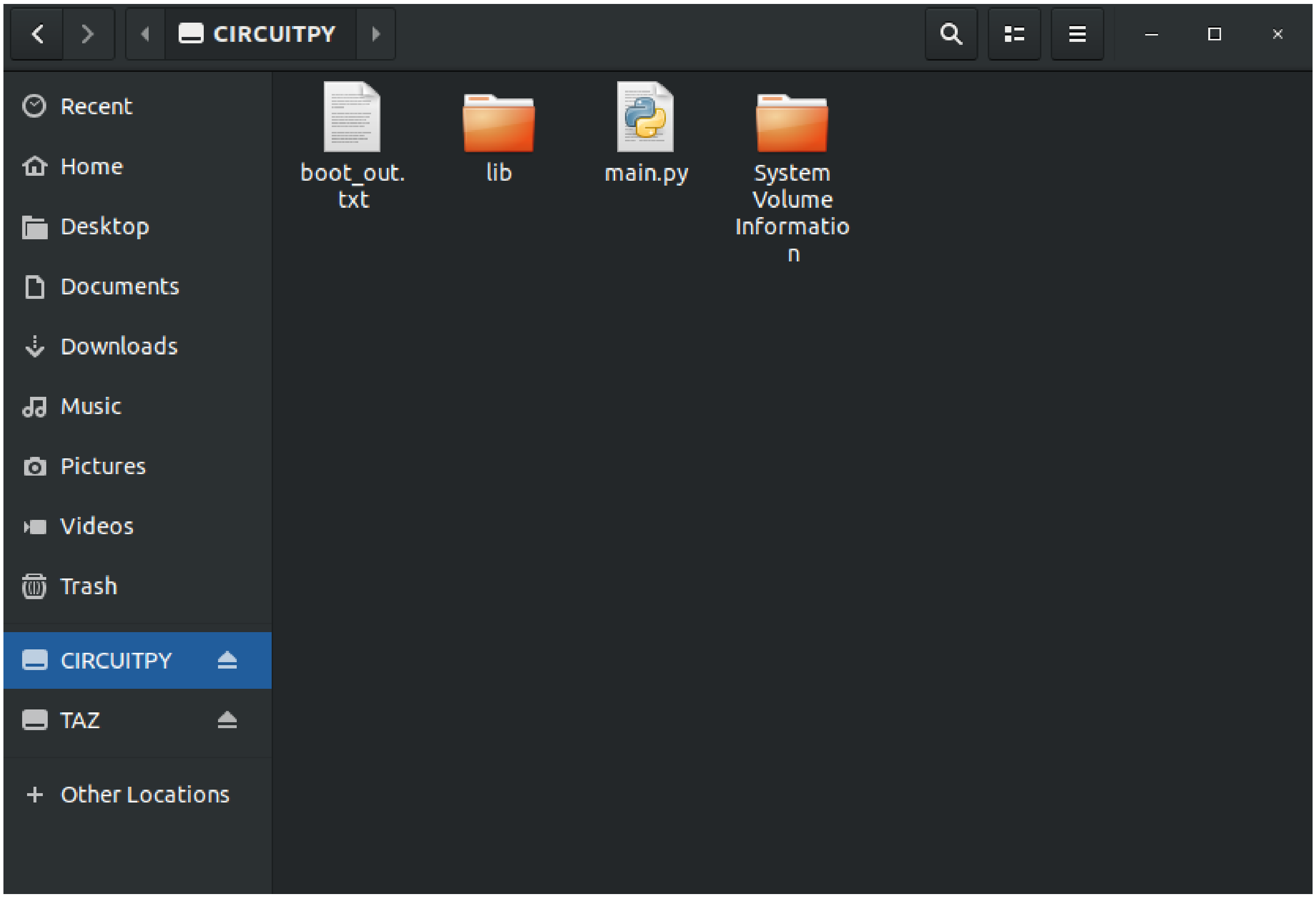
When you get your CPX and plug it into the computer via USB it actually won't run Python just yet. First you need to double click the reset button (the button in the center. It says RESET above the button) and put it into boot mode. All the neopixels (the ring lights on the CPX) will light up green.



Something called CPLAYBOOT will pop up on your computer just like a USB stick or external harddrive. A couple files with be in there but it doesn't matter what they say right now.



You then need to download what's called a UF2 file and transfer it onto the CPX. Note if you purchased a kit with the Bluefruit you need to download a different UF2. Make sure you get the right one. Once the UF2 is downloaded you need to drag the UF2 over to the CPLAYBOOT drive on your computer. After a bit of time a USB drive called CIRCUITPY will pop up as a flash drive on your computer. The CPX is now like a USB stick with 2MB of storage. **Note that if you have an old bootloader you must update the bootloader.**

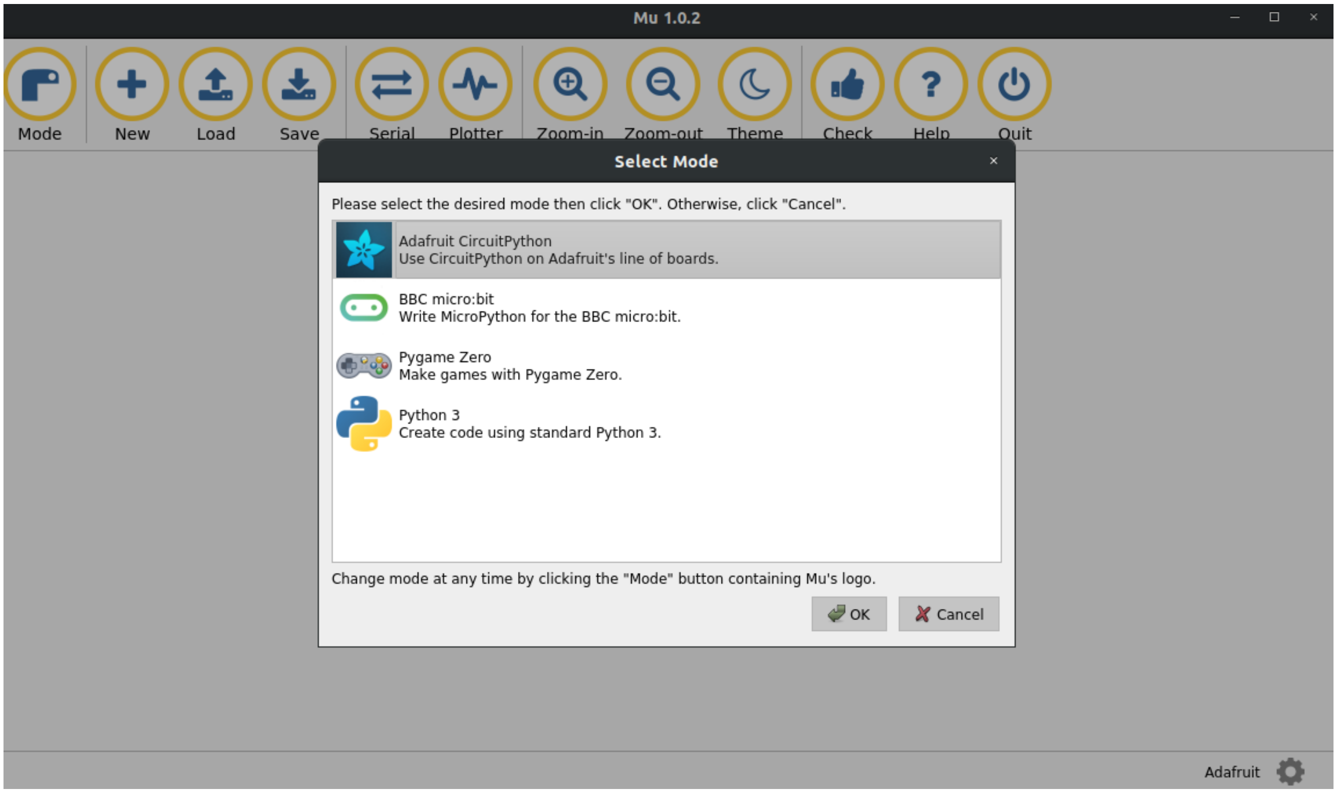


At this point the CPX is like a mini computer. If you put Python code on the "flash drive" it will run python code. Since I've done this before, there are a number of files already on my CPX. You may have some or none of these. The "boot_out.txt" file will tell you the version of CircuitPython you have on the CPX. Mine says this:

```
Adafruit CircuitPython 5.3.0 on 2020-04-29; Adafruit CircuitPlayground Express with samd21g18
```

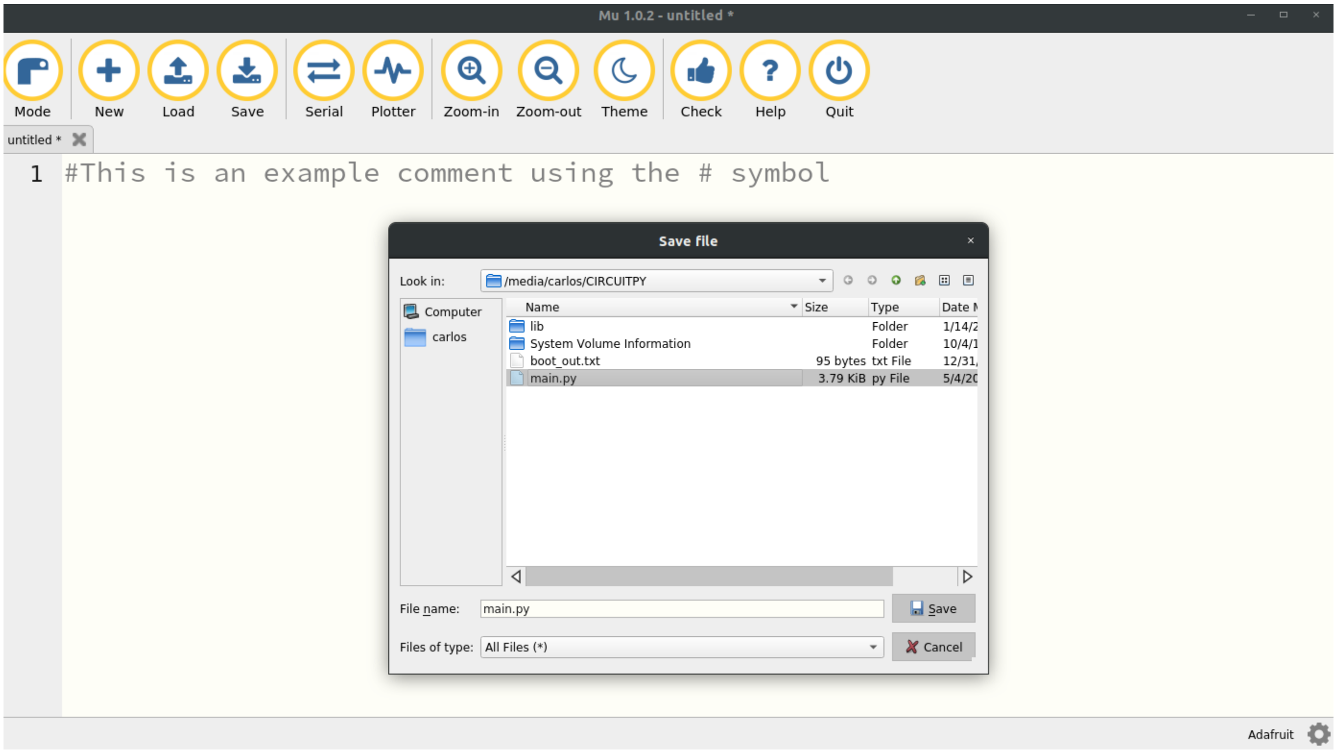
Which means I have CircuitPython version 5.3.0 last updated on April 29th, 2020. The CPX itself is using the ATSAMD21G18 microprocessor. The folder lib is a folder with extra libraries that you may need to install later. The file main.py is the Python file that the CPX is currently running. **The CPX can store as many Python files as possible for a 2MB flash drive but it will only run ONE Python script at a time and that file must be named main.py** The folder *System_Volume_Information* is a file management folder that we will never use.

If you want the CPX to run code you simply need to edit the file *main.py* or if that file does not exist you just need to create it. You could just open Notepad or any other text app (Sublime,TexWrangler, Emacs, Vi, Nano, Gedit, Notepad++, Wordpad, VSCode, etc) but the CPX has alot of debugging options and it is recommended to use a program called "Mu". Mus is a good way to write and debug code on the CPX. Note that Mu is only used to program the CPX. If you want to run Python code on your laptop you need to use Thonny or Spyder (or whatever other IDE you downloaded). If you want to run Python code on your CPX you must use Mu. Once you've downloaded and installed Mu and open it up it will look like this (Note it's possible the the software gets updated from the time this book is published. As such be sure to select the Circuitpython Mode for board development).



Make sure to select the “Adafruit CircuitPython” option. If the software has been updated and that option no longer exists, be sure to select the option that says Circuitpython for board development.

Alright, let’s start writing code! If you have a file called *main.py* on your CPX click the Load button and load *main.py* (make sure to load the main.py that is stored on your CPX and not somewhere else on your computer.) If a file *main.py* does not exist on your CPX simply click the New button and then Save the file as *main.py* (again make sure you save it to the CPX and not to your computer)



You'll see that I am accessing the CIRCUITPY drive and saving the file as main.py The file itself is empty and just has a comment using the # symbol. At this point since the file is blank the CPX won't do anything.

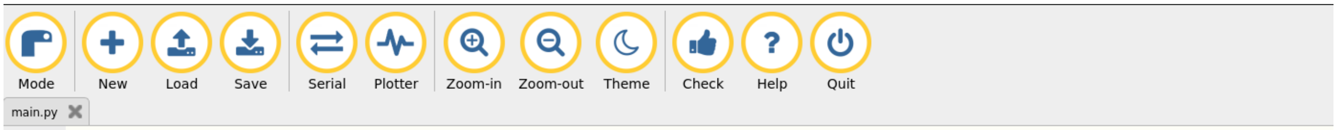
Now this is really important. Your CPX is a USB stick that can hold as many Python files as 2MB will allow but it can only run or execute one python script at a time. Furthermore it will only run two types of files. It will run code.py if it exists and if it can't find code.py it will run main.py If the CPX can't find main.py or code.y it will just not do anything. If you have two versions of main.py or a combination of main.py or code.py it will run one of them and not the other. Make sure you only have one version of main.py or code.by but not both! Some common things to check if your CPX isn't working.

1. Make sure you're using Mu in the right Mode
2. Make sure main.py or code.py is on the CIRCUITPY drive and not somewhere on your computer.
3. Make sure you are editing the right file in Mu. Do you have two versions of main.py?
4. Are you editing using Thonny or Spyder?
5. Are you editing a file on your computer? Make sure you are writing to the CIRCUITPY drive.
6. Unplug the CPX, close Mu and try again.

So let's get the CPX to do something simple like blink an LED. I have an entire Github devoted to Microelectronics. Specifically I have a folder with all of my Circuit Playground files. The easiest program to run is the blink.py script. I've attached a screenshot of the script below.

```python
import board
import digitalio
import time

led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT

while True:
    print(time.monotonic())
    led.value = True
    time.sleep(0.5)
    led.value = False
    time.sleep(0.5)
    led.value = True
    time.sleep(0.1)
    led.value = False
    time.sleep(0.1)
```

We will talk about what this code is doing later on. For now copy and paste these 17 or so lines of code and paste them into Mu specifically the *main.py* script. It will hopefully look like this.



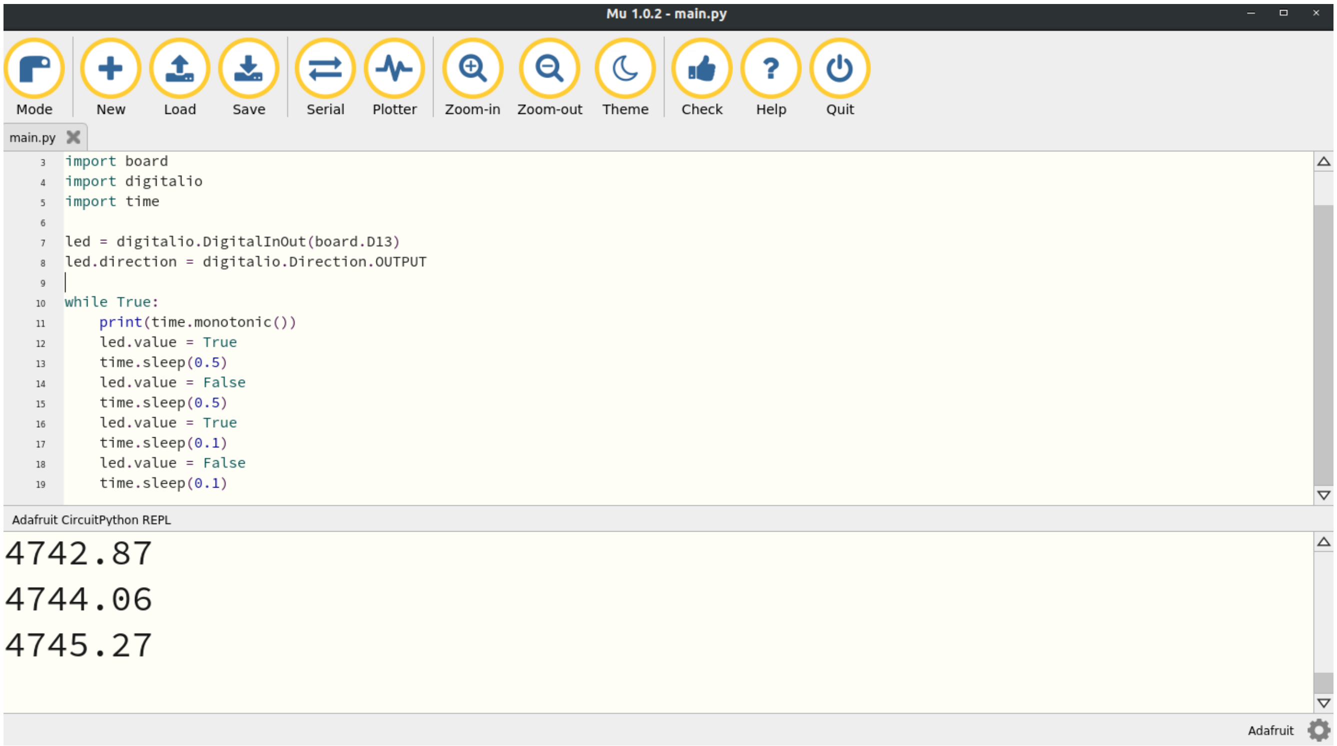

```
1  #This is an example comment using the # symbol
2  
3  import board
4  import digitalio
5  import time
6  
7  led = digitalio.DigitalInOut(board.D13)
8  led.direction = digitalio.Direction.OUTPUT
9  
10 while True:
11     print(time.monotonic())
12     led.value = True
13     time.sleep(0.5)
14     led.value = False
15     time.sleep(0.5)
16     led.value = True
17     time.sleep(0.1)
18     led.value = False
19     time.sleep(0.1)
```

Make sure to save. You can click *Zoom-In* and *Zoom-out* to zoom in and out to change the font size and see more of the output. If the blink code is working you will see a red led labeled D13 on the CPX blink back and forth. D13 stands for digital pin 13. You'll notice there are analog pins labeled A5 and A6 among other numbers. Let's talk about the code a bit more and explain why it's doing what it's doing. The three lines at the top of the code are *import* commands to import different modules just like we did for *numpy* and *matplotlib*. In this case the modules being imported are *board*, *digitalio* and *time*. The module *board* is used to import the layout of the CPX so we can access different pins on the CPX. The module *digitalio* stands for digital input output which means we are inputting and outputting digital signals. Since we can combine this module with the *board* module we will be able to output digital signals to different pins on the CPX board. Remember that PCB stands for printed circuit board so *board* implies we are accessing pins on the CPX PCB. Hopefully that makes sense. The final module we are importing is *time* which acts just like the *time* module on your desktop computer. It will let us access the CPX's internal clock.

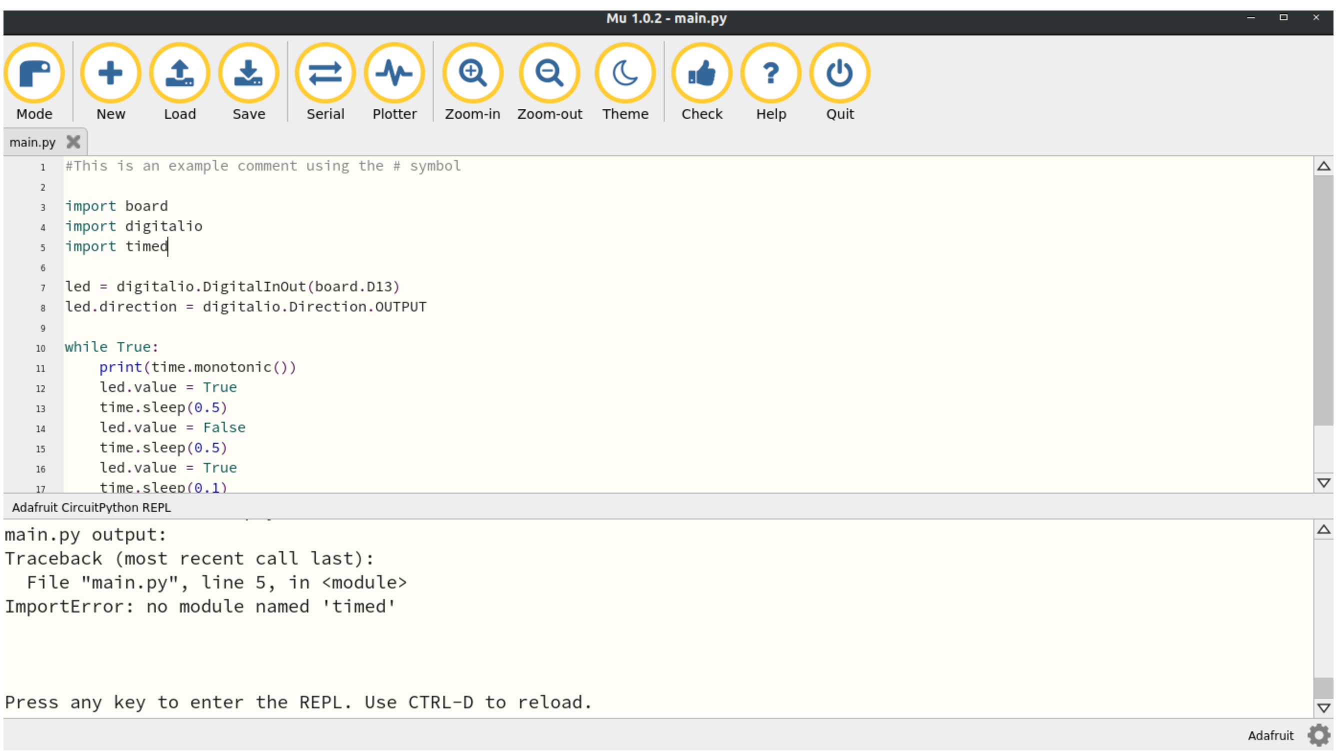
Moving along, lines 7 and 8 create an LED object using the *board* module and *digitalio*. It's a long line of code that basically says, create a variable called *led* that lets us output a digital signal to pin D13. We also set the direction of the LED to output since we only want to write to the LED.

Lines 10 through 19 kick off an infinite loop that never ends. The line that says *while True:* means loop while *True*. Well *True* is always true which means it will loop forever. The colon at the end of the line tells Python that the loop condition statement ends and to begin looping from 11 through 19.

Line 11 specifically says *print(time.monotonic())*. First the *print()* function is used to print things so that you and I can see it. Rather than just seeing a blinking LED we want to see the time printed. The *time.monotonic()* is using the module *time* which we imported and using a function from that module called *monotonic()* which calls the internal clock of the CPX. So how do you see the output from the print statement? Hit the *Serial* button on Mu and you will hopefully see some output. Here's what it looks like on my machine.



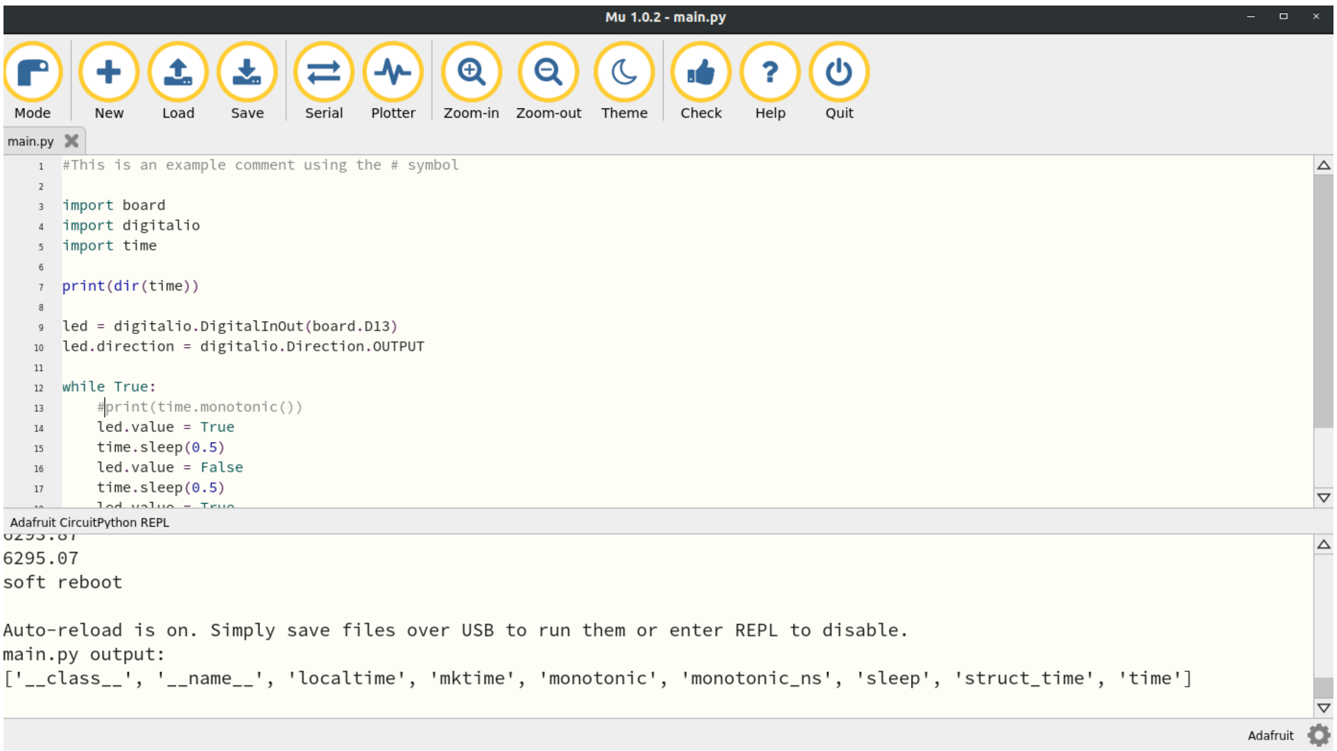
In this case you can see the time printed every time it goes through the loop. You may even see an error. This *Serial* button is great for debugging because it will tell you the error in your code. For example, in the picture below I have an error in my code and the *Serial* output is letting me know.



In this case I have an error on line 5. It's saying there is no module named *timed*. The reason that module doesn't exist is because the module is actually *time* not *timed*. You can use the *Serial* monitor to check on your program and see what errors you may have. Ok so there are two more lines of code to discuss. They are *led.value = True* and *time.sleep(0.5)*.

These lines of code are repeated throughout the while loop and do two things. First, the *led.value* either sets the value of the LED to True which turns the light on or False which turns the light off. The LED is digital which means the signal can either be on or off. There's no in between for digital signals. The *time.sleep()* function tells the CPX to pause for half a second. You can change the number in the parentheses if you want to change length of time the code pauses. Note that the CPX completely pauses. That is no code runs during a sleep.

If you've gotten the LED to blink you're all set for this lab. However, I'd like to you learn a few more things about documentation. Just like Python on your desktop you can lookup the documentation on the CPX itself. For example, I've added a print statement to print the directory of time.



In this case I've added *print(dir(time))* to line 7. The output shows that the *time* module has 9 functions including *monotonic()*. Unfortunately CircuitPython does not have *__doc__* functions built in which means if you want to learn about a specific function, you need to visit the documentation for CircuitPython. Here's the specific documentation for the time module. A lot of example code from the Adafruit Learning System is also on Github.

Finally, in order to keep practicing using Python on your desktop I want you to modify the code above to run on your desktop computer. You're going to have to modify a few things. First, make sure to open Thonny or Spyder depending on which version of Python IDE you downloaded. Then, only import the *time* module. All the other modules don't exist on your desktop. Also, get rid of all the lines of code that blink the LED. We just want to print time in a while loop. Finally, the time module on your desktop uses a function called time.time() (unless you have Python3 installed) so when you print time make sure to use that module instead. Again visit the documentation for time for Python if you want to learn more. Thonny by default will load Python version 3 but it's possible you may have Python version 2 so make sure you look up the documentation for the appropriate version of Python. After searching through the documentation you can use a function called asctime() on your desktop. This is the output I get in Thonny when I add a sleep of 1 second. The exercise for you is to do something similar.

```
>>> %Run myprint.py

Thu Jul  2 10:42:37 2020
Thu Jul  2 10:42:38 2020
Thu Jul  2 10:42:39 2020
Thu Jul  2 10:42:40 2020
Thu Jul  2 10:42:41 2020
Thu Jul  2 10:42:42 2020
Thu Jul  2 10:42:43 2020
Thu Jul  2 10:42:44 2020
```

5.4 TL;DR

1. Plug in your CPX and double tap it to go into reset mode. CPLAYBOOT will mount to your computer. (Some CPXes you need to hold the button down instead of double tap).
2. Download the UF2 file.
3. Drag the UF2 file to your CPLAYBOOT drive. After a few seconds CIRCUITPY will mount.
4. You need to then download Mu
5. Open Mu and make sure to select the mode Adafruit CircuitPython (or just CircuitPython if you have a new version)
6. Open main.py (or code.py) in Mu from the CIRCUITPY drive in Mu. (It's possible you have code.py on your drive instead of main.py which is fine. Just open one or the other)
7. Copy the blink.py script into main.py (or code.py)
8. Once you have the script running, modify the script to run on your Desktop using Spyder or Thonny

There is an accompanying youtube video to help you see me perform the 8 steps above.

5.5 Assignment

For this assignment you must appropriately set up your CPX/CPB, get the blink code to run and include some screenshots of relevant steps. **Note for the blink code, modify blink pattern to be different from the default on Github**. The specific items requested are shown below.

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%
5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. Include a screenshot of your computer showing the CPLAYBOOT drive on your computer - 20%
2. Include a screenshot of your computer showing the CIRCUITPY drive on your computer - 20%
3. Include a screenshot of Mu with the Serial monitor open showing the output of the blink code. Remember to modify the blink pattern - 20%
4. Include a photo of your CPX with the red led on - 20%

6 Troubleshooting Guide

Is your CPX/CPB broken or not running code? Read below.

1. Your CPX is a USB stick that can hold as many Python files as 2MB will allow but it can only run or execute one python script at a time. It will look for code.py and then main.py and that's it.
2. Make sure you're using Mu in the right Mode (CircuitPython)
3. Make sure main.py or code.py is on the CIRCUITPY drive and not somewhere on your computer.
4. Make sure you are editing the right file in Mu. Do you have two versions of main.py or perhaps main.py and code.py?
5. Do you have boot.py on there when you don't need it?
6. Are you editing using Thonny or Spyder? You're supposed to use Mu.
7. Are you editing a file on your computer? Make sure you are writing to the CIRCUITPY drive.
8. Do you have the right modules in your *lib* folder?
9. Unplug the CPX, close Mu and try again.
10. First, try and reset the CPX to CPLAYBOOT and reflash the UF2 to see if that fixes it.
11. If you have a linux computer you can "sudo screen /dev/ttyACM0" and then run "import storage" and then "storage.erase_filesystem()"
12. If that doesn't work sometimes you just need to completely erase the CIRCUITPY drive so head over to this troubleshooting guide and follow some of the steps they tell you.
13. As a last resort you can try to download these UF2s and hopefully it will fix all errors and mistakes - CPX, CPB
14. If you're having an issue with an old bootloader be sure to update the bootloader for the CPB or alternatively the bootloader for the CPX. At the time of this writing here is the command you need to upload the new bootloader the CPB. If you have version 0.4.0 or later you can use the UF2 method.

    ```
    adafruit-nrfutil --verbose dfu serial --package
    circuitplayground_nrf52840_bootloader-0.8.2_s140_6.1.1.zip -p /dev/ttyACM0
    -b 115200 --singlebank --touch 1200
    ```

    For linux users you will need a few things

    (a) You need pip3 $->$ $ sudo apt install python3-pip
    (b) You need adafruit-nrfutil $->$ $ pip3 install --user adafruit-nrfutil
    (c) adafruit-nrfutil gets installed into ~/.local/bin
    (d) You need to add your user to dialout $->$ $ sudo usermod -a -G your_username
    (e) Restart your computer

    For the CPX there is a UF2 that you transfer over to the CPLAYBOOT drive. At the time of this writing the version you're looking for is *update-bootloader-circuitplay_m0-v3.15.0.uf2*.
15. Are you running out of memory? To figure out how much RAM you have on your CPX/CPB you need to *import gc* then run *gc.collect()* and then *print(gc.mem_free())*. At the time of this writing, the CPX has about 17 KB of RAM and the CPB has around 140 KB of RAM.

7 External LEDs and Push Buttons

7.1 Parts List

1. Laptop
2. CPX + USB Cable
3. 2 Alligator clips
4. Push Button
5. Breadboard
6. LED (Light Emitting Diode) (x3 in case you fry some)
7. Resistor (300 to 1000 Ohms)

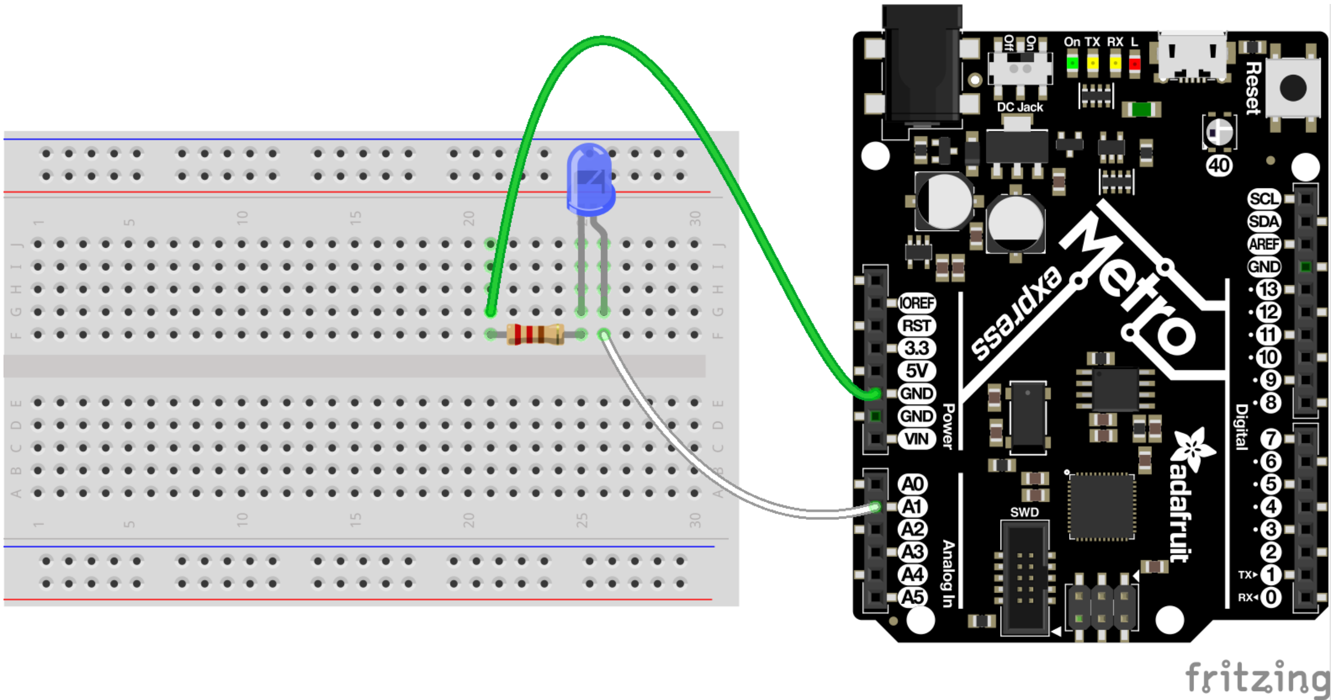
7.2 Learning Objectives

1. The VOUT and 3.3V pin are always "ON" even when code is not running on the CPX. So long as your CPX is plugged in via USB or a Lipo Battery
2. LEDs are Light Emitting **Diodes** which means current only flows in one direction
3. LEDs need resistors in series otherwise they will get too hot and burn up
4. Breadboard pinout diagrams
5. Analog pins can be controlled by simply using the digitalio module
6. LEDs can be hooked up to analog pins and set to blink by changing the board pin

In this project we're going to use the same blink code as before but modify it to blink an external LED. The purpose of this lab is to familiarize yourself with the pins on the CPX and create a simple circuit using the 5V (VOUT) pin on the CPX and one of the Analog pins. Your laptop has a battery with something between 10 to 20V. There are DC to DC converters in your laptop that provide 5V to your USB ports. These USB ports can be used to power your CPX as you have done in the past few labs.

If you purchased the optional battery pack you can also power the CPX using 3 AA batteries. These batteries nominally have 1.5 V but fully charged it's actually something like 1.8 V. So 1.8 times 3 is 5.4V which is enough to power the CPX. If you have the battery pack and some AA batteries, give it a try. If you still have the blink code from the last project on board you'll see the D13 LED blink as before. You won't be able to see the serial print() output as before but that code will be running which is why D13 is blinking. **I have noticed that some of the battery packs have power and ground wires swapped. If the battery pack doesn't work it may be because those two wires are backwards.**

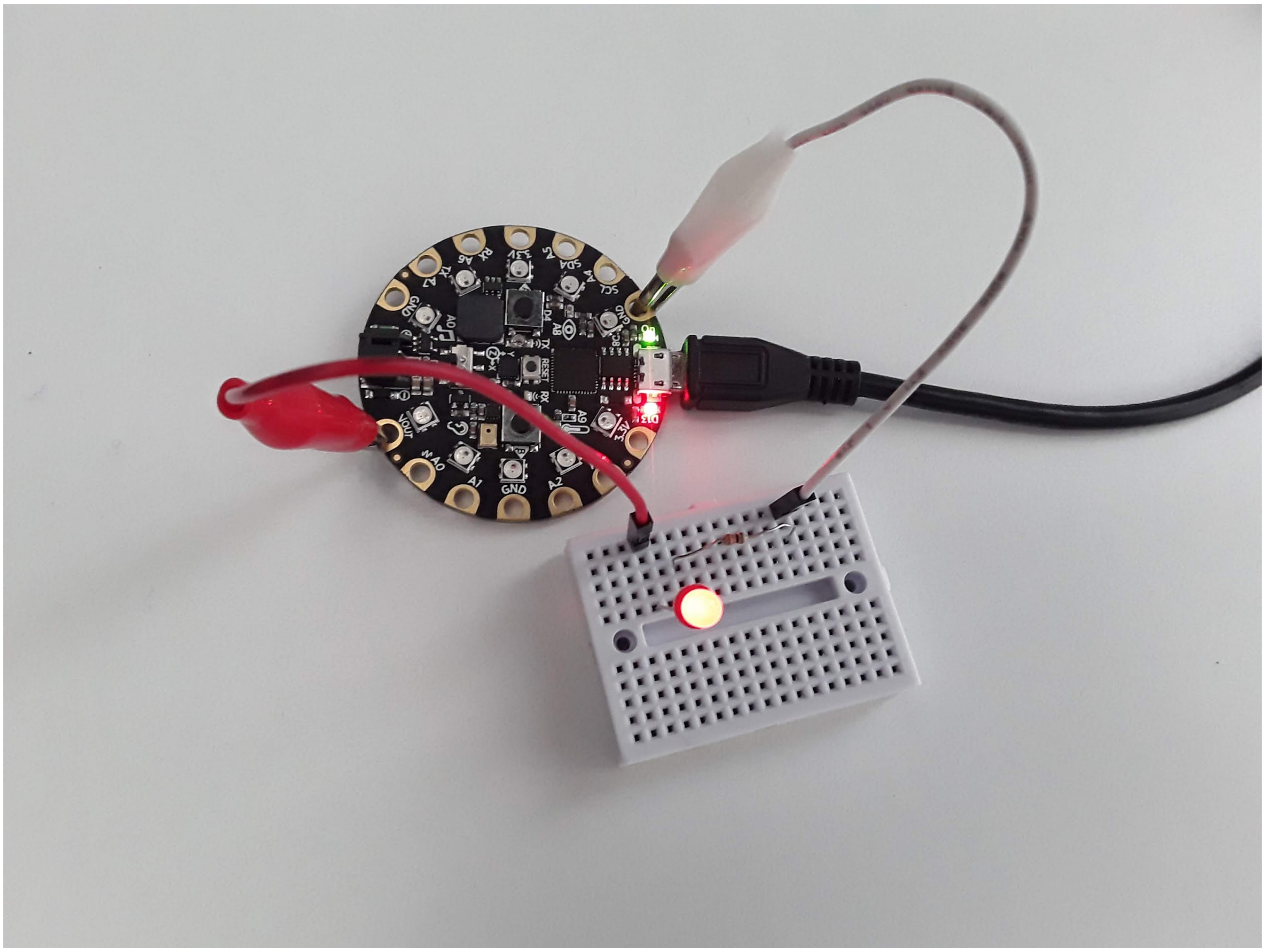
The CPX itself uses 3.3V logic which means when it converts numbers to binary a 0 (False) represents 0 volts and a 1 (True) represents 3.3V. The CPX has ports that are labeled various things. GND stands for ground and you need to hook the negative end of your circuit to this and it also has VOUT which supplies 5V to any circuit you build. Hook the positive end of your circuit to the VOUT pin. There is also a port labeled 3.3V and obviously that outputs 3.3V. The Figure below shows an Arduino Metro hooked up to an LED on a breadboard (Courtesy of Tony Dicola).



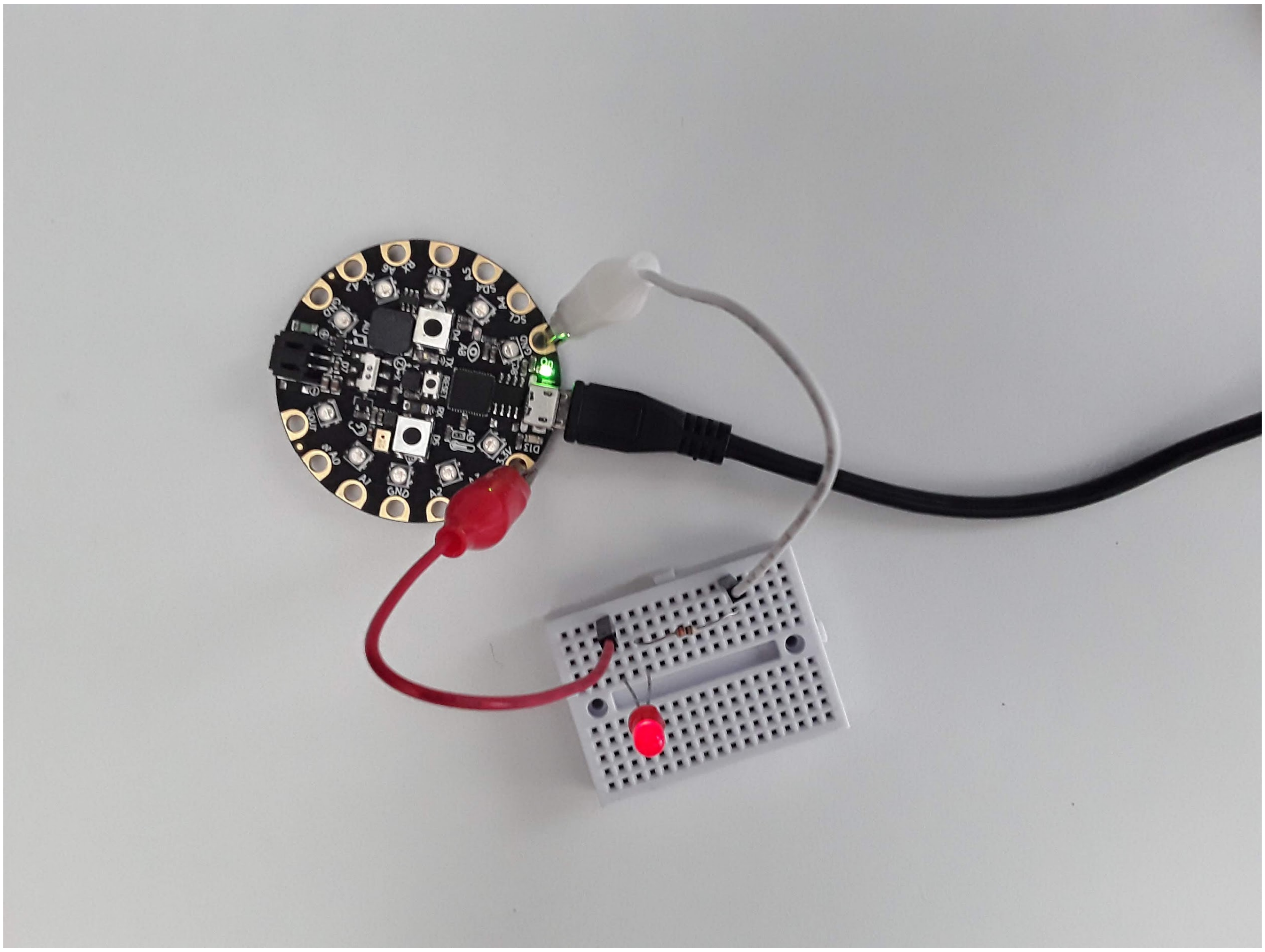
If you're not familiar with how breadboards work I would recommend watching this video on how breadboards work. Your lab today specifically involves an external LED as shown above. You can read about LEDs more online if you wish. **Remember that the long leg of the LED is the positive end and the short leg is the negative end.** The task today is to wire an LED up to the CPX in multiple ways. The figure above shows the LED connected to the A1 pin but we will be wiring up the LED to multiple pins on the CPX/CPB to gain understanding in how the circuit works. **Note, whenever you modify a circuit on the breadboard, always be sure to remove power from the CPX. You can damage multiple components if you're not careful.**

7.3 LED with no Code

For this part we are going to light up the LED without the use of any code on the CPX. First, wire up the circuit with the positive end connected to 5V. This is how my circuit looks. Make sure to use a resistor between 300 and 1000 Ohms. An LED does not have that much internal resistance so you need a resistor in series with an LED to reduce the amount of current flowing through the LED or the entire LED will fry. If you use a resistor that has too much impedance the LED just won't turn on because the voltage/current through the LED will be below the activation voltage of the circuit.

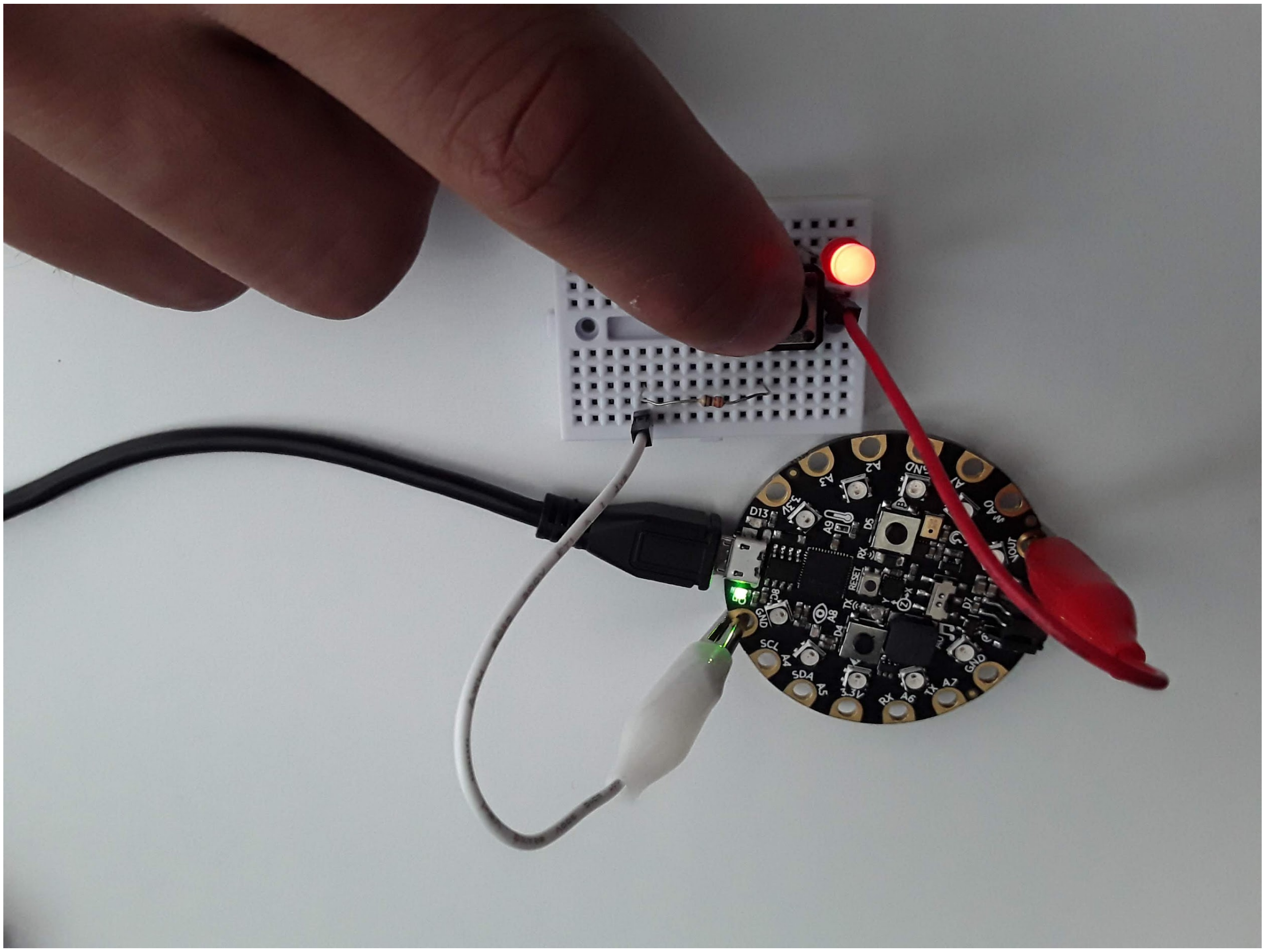


Once you have that circuit working, wire up the circuit again with the 3.3V output. Do you notice anything different when you hook up the circuit with different pins? Here's my circuit. Do you notice something different about the intensity of the LED? Why is it different?



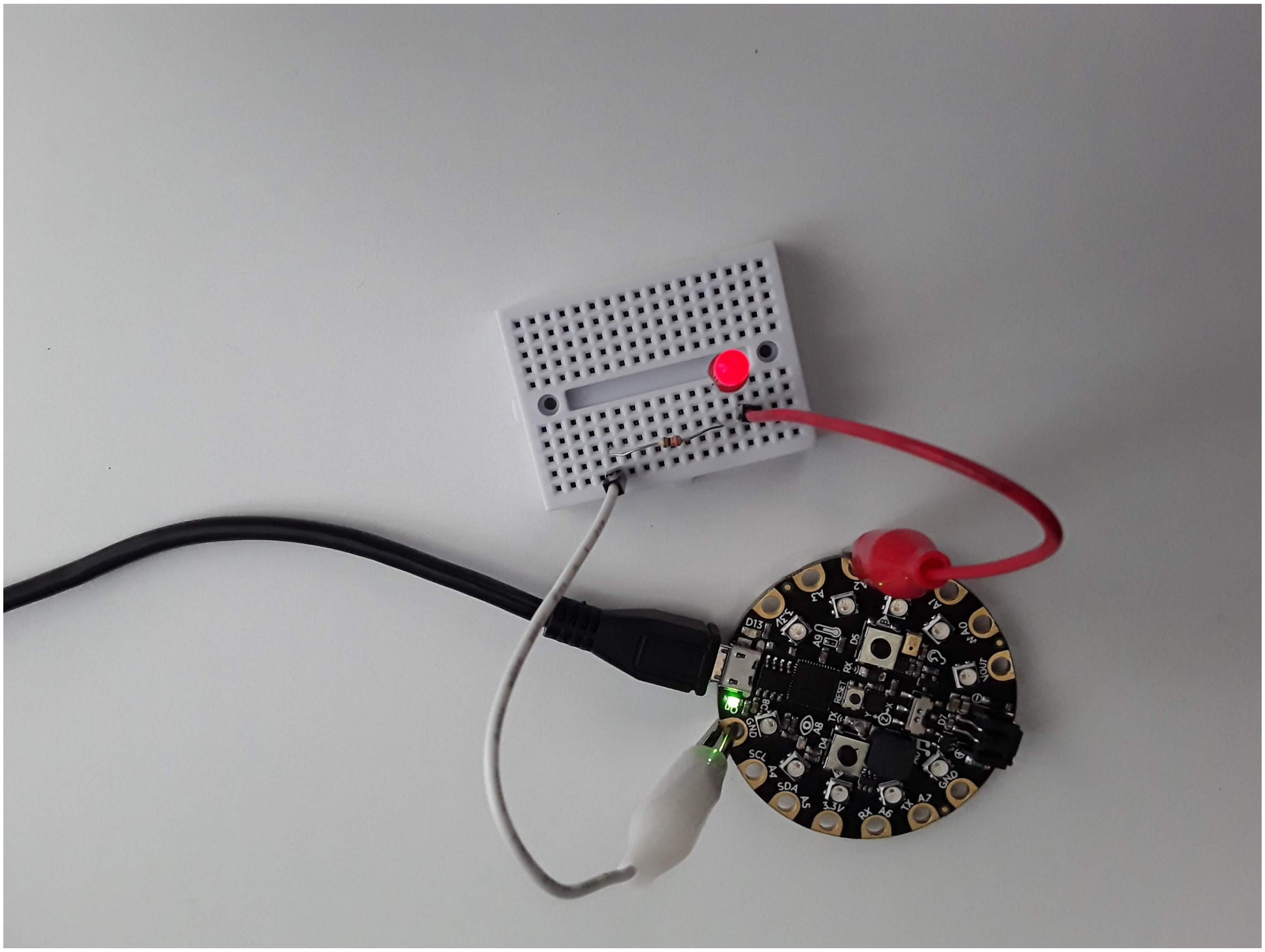
7.4 LED with a push button

Now that we understand breadboards a bit, we're now going to manually blink the LED using a push button placed onto the breadboard and have it act like a switch. Therefore, when the button is pressed, the LED will turn on and when the button is released the LED will turn off. The button just acts like a wire so you can plug in the button anywhere in the circuit.



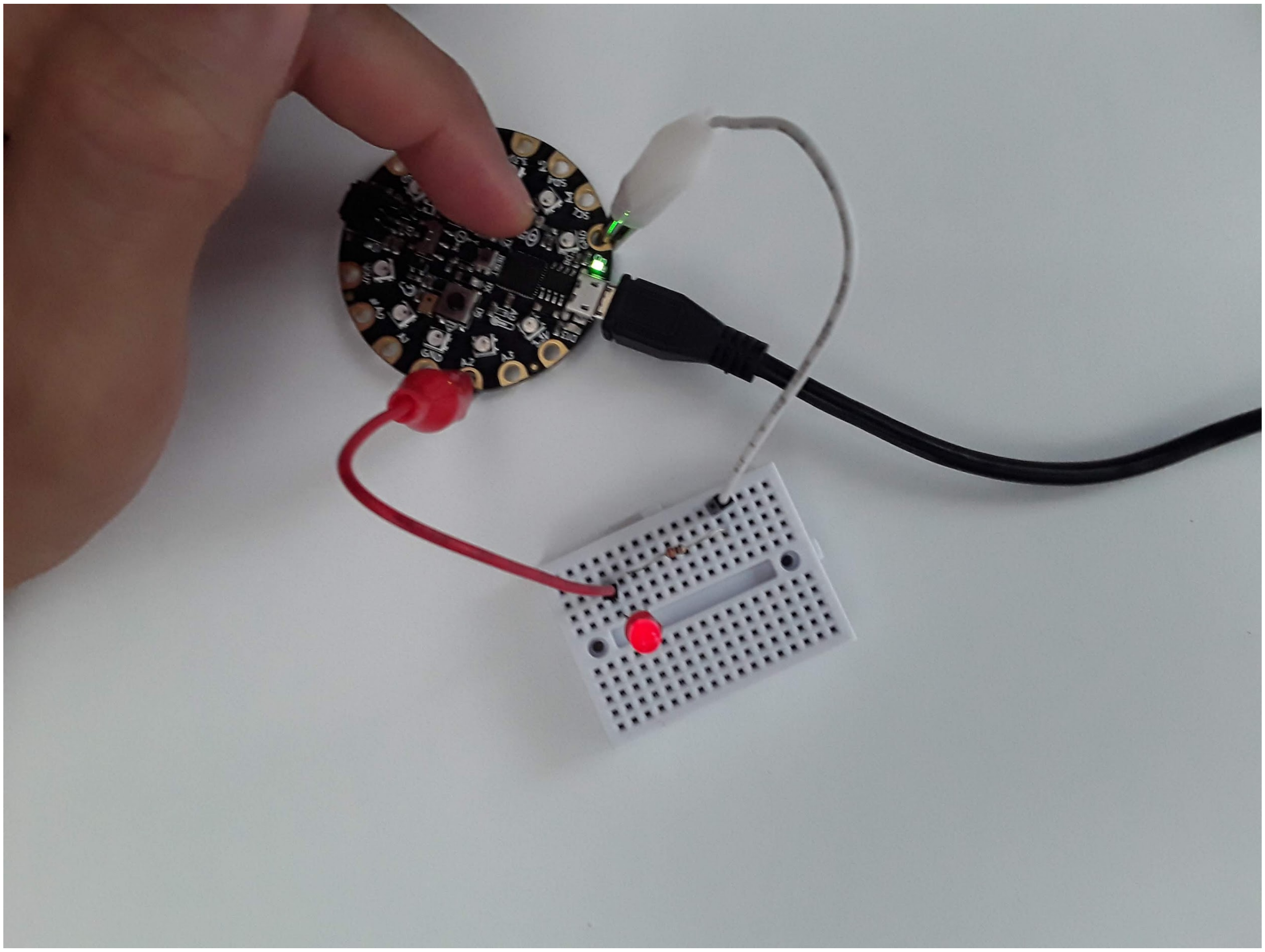
7.5 LED with code

Next I want you to remove the button from the circuit and wire up the LED like you had it when the positive end was connected to VOUT or 3.3V. Except this time I want you to hook the positive end of the circuit to pin A2. Then edit your blink code to blink pin A2. Take a look at the blink code. Right now the code is blinking pin D13. How do you think you need to change the code to blink pin A2? Here's what my circuit looks like for this one. I won't include code for this one since you just need to change one line of code.



7.6 LED with CPX button

Finally, I want to use one of the buttons on the CPX to blink the LED hooked up to pin A2. For this code to work you first need to detect a button press and then tell the program to change the light from True to False depending on what it's current status is. This one is a bit more difficult so I'll include the code here and discuss the code itself. Here is my circuit (identical) to the previous one with Button A on the CPX pressed down.



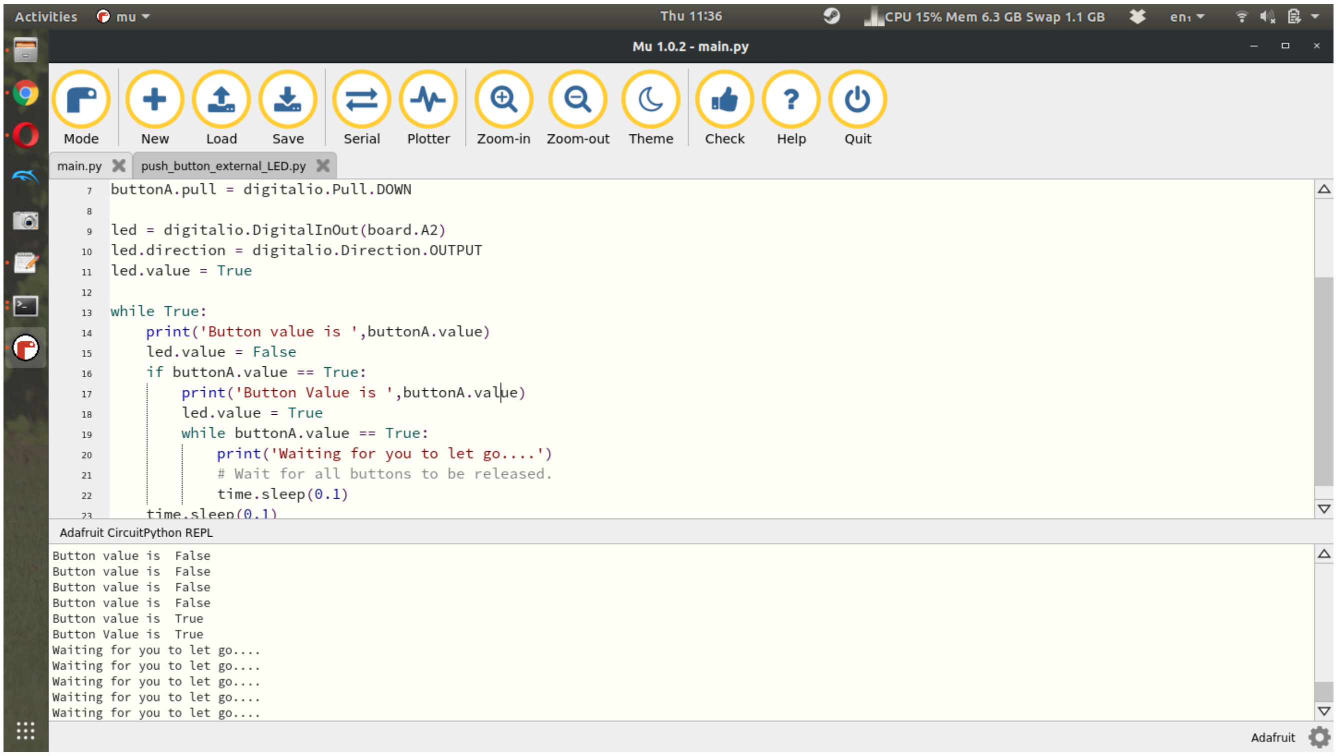
Alright so how do we detect a button press? Well the documentation on this is not so straight forward. What we want to do is detect the INPUT of a digital signal and then do something if we detect that signal. Here's the code I created to get it to work.

```
1  import board
2  import digitalio
3  import time
4
5  buttonA = digitalio.DigitalInOut(board.BUTTON_A)
6  buttonA.direction = digitalio.Direction.INPUT
7  buttonA.pull = digitalio.Pull.DOWN
8
9  led = digitalio.DigitalInOut(board.A2)
10 led.direction = digitalio.Direction.OUTPUT
11 led.value = True
12
13 while True:
14     print('Button value is ',buttonA.value)
15     led.value = False
16     if buttonA.value == True:
17         print('Button Value is ',buttonA.value)
18         led.value = True
19         while buttonA.value == True:
20             print('Waiting for you to let go....')
21             # Wait for all buttons to be released.
22             time.sleep(0.1)
23     time.sleep(0.1)
```

The code above is an image. You could type exactly what you see and hope you don't type any errors (which is highly unlikely) or you could look for my code on my Github. Have you bookmarked this link yet? I recommend you do so!!! So you're making a code about Buttons....hmmm. Click that Github link. Where do you think the code about Buttons is located? Anywho, let's talk about the code.

The first 3 lines are exactly the same as before. Line 5 through 7 are similar to creating the "led" variable except we're using *BUTTON_A* as the board value and setting the direction to *INPUT*. Finally we're setting the pull direction to *DOWN*. This means the button acts like a pull down resistor and when it's pressed the value of the button goes *HIGH*.

Lines 9-11 are the same as before. We create an led variable and tell the CPX that the led is hooked up to pin A2. We then start the while loop on line 13. First if you click the Serial button you'll see the text 'Button value is False'. It's False because *buttonA.value* is not being pressed. On line 15 the led is set to False (turned off). On line 16 the button value is checked using an if statement as to whether or not the button is pressed (True). If the button is pressed, the user will be notified that the button is pressed on line 17 and the led will be set to True (on) in line 18. Lines 19-22 are while loop that will notify the Serial monitor that you must let go of the button before the code can continue to the main while loop. The *time.sleep* functions are there to make sure a human can operate the button without code running faster than a human can press a button. When I press the button down here is the output I get from the *Serial* monitor.



Here you'll see 4 lines that say "Button value is False" and then two lines that say "Button value is True" followed by 5 lines that say "Waiting for you to let go...". See if you can get this code to work and play with it and modify it as you see fit. By the way, the LED connected to pin D13 has this exact same circuitry, an LED a resistor, it's all just soldered to the PCB so you don't have to build it using a breadboard. Hopefully now you have some appreciation for buttons and LEDs!!

7.7 Assignment

For this specific project you must get your LED to turn on and off using buttons as well as software. The different scenarios are shown below and the grading rubric is presented after this list.

1. Get your LED to turn on with the VOUT pin.
2. Get your LED to turn on with the 3.3V pin.
3. Get your LED to blink automatically using an analog pin on the CPX/CPB.
4. Get your LED to turn on using a button on the breadboard.
5. Get your LED to blink automatically by pressing a button on the CPX/CPB.

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%

5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix ”A” and ”B” will result in a zero in the assignment. Appendix ”A” will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix ”B” will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. Include a photo of your circuit with your LED turned on using VOUT. Based on the resistance in your resistor and LED, what is the voltage drop and current through the LED? - 10%
2. Include a photo of your circuit with your LED turned on using 3.3V. Based on the resistance in your resistor and LED, what is the voltage drop and current through the LED? - 10%
3. The USB port from your computer is 5V. The DC to DC converter on the CPX/CPB converts that 5V to 3.3V. Compute the Gain (G) for this converter both in magnitude and in dB.
4. Include a photo of your circuit with your LED turned on using a button on your breadboard - 10%
5. Include a photo of your circuit with your LED turned on using an analog pin and then also include a screenshot of your Mu code showing the same analog pin used to blink your LED - 20%
6. Include a photo of your circuit with your LED turned on by pressing a button on the CPX/CPB and then also include a screenshot of your Mu code showing the line of code used to read the button press and blink the LED. - 20%

8 Using the CPX/CPB as a Data Acquisition System (DAQ)

8.1 Parts List

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

8.2 Learning Objectives

1. Record real time measurements from the CircuitPlayground using the Serial monitor
2. Learn how to use the typing module to type data directly into a spreadsheet or text file
3. Learn how to log data on the on board memory of the CircuitPlayground

8.3 Extra Help

This is a pretty hard project to do with multiple different methods. After you've read through this document I suggest you watch a youtube I created on Logging Data with the Circuit Playground Express. I have also made video where I log temperature and accelerometer data using on board memory with the CircuitPlayground Bluefruit.

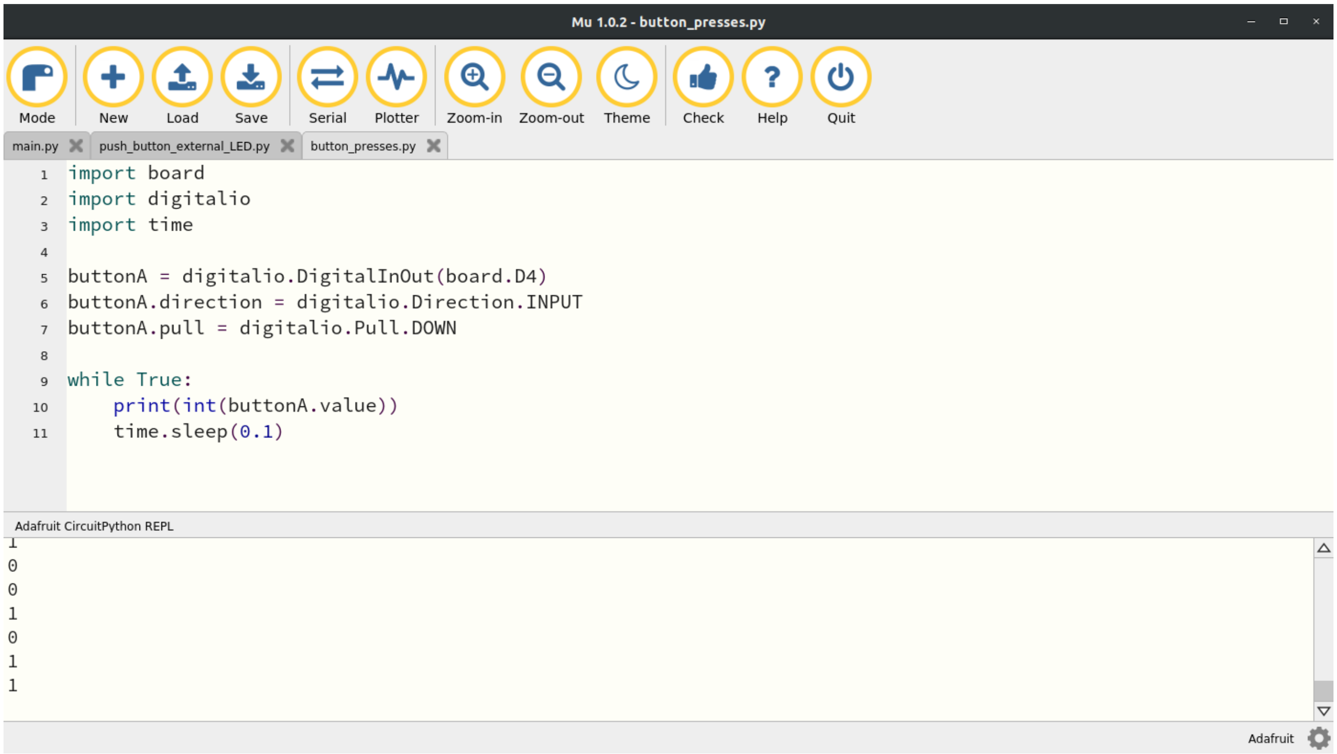
8.4 Getting Started

Taking data is the core of any instrumentation project. Data Acquisition Systems or DAQ for short come in all shapes and sizes. Believe it or not the CPX can be used as a crude and cheap DAQ. The CPX can easily take temperature data and monitor the temperature in a greenhouse or take humidity readings of a plant to monitor soil content. Before we learn about the different sensors on board the CPX, we want to make sure we can store that data later rather than just having it spit data out via the serial monitor. For starters though let's get the CPX to print out something simple like button presses since we've touched on that already. The code I'm using is shown below and can also be found on Github.

```
import board
import digitalio
import time

buttonA = digitalio.DigitalInOut(board.D4)
buttonA.direction = digitalio.Direction.INPUT
buttonA.pull = digitalio.Pull.DOWN

while True:
    print(int(buttonA.value))
    time.sleep(0.1)
```

The code is pretty similar to what I had in the past. I import board, digitalio, and time. I create a buttonA object using the digitalio library to record button presses. I then enter into a while loop print the buttonA.value.

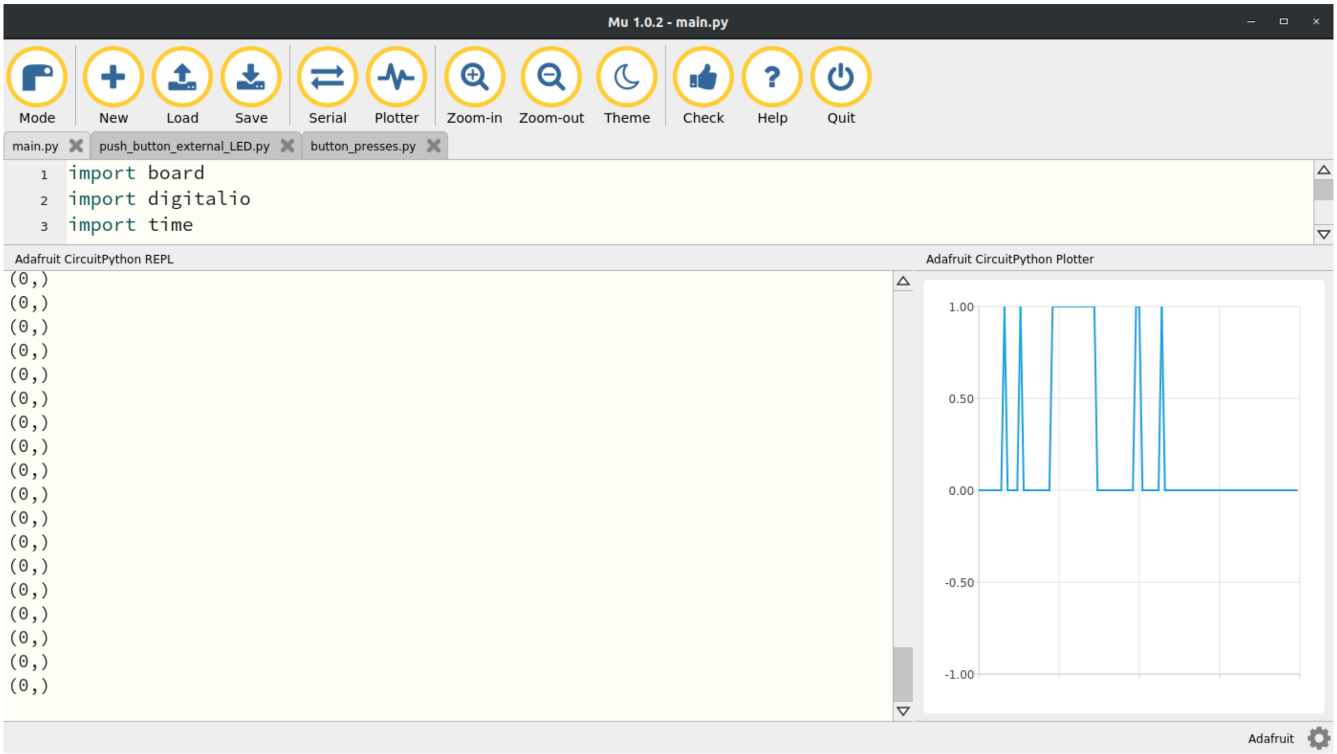
The difference here is that I use the int() function to convert the buttonA.value to an integer. The reason why I do this is because buttonA.value is a boolean. It is either True or False. An integer though is a number and thus a value of False is 0 and True is 1. If you open the serial monitor and push the A button down a few times you'll see some zeros and 1's.



Mu also has a really neat builtin plotter. You'll see next to the Serial button there is a button called Plotter. If you click that button now nothing will pop up on the screen. Unfortunately in order to plot using the Plotter you need to modify the print() statement to this:

```
print((int(buttonA.value),))
```

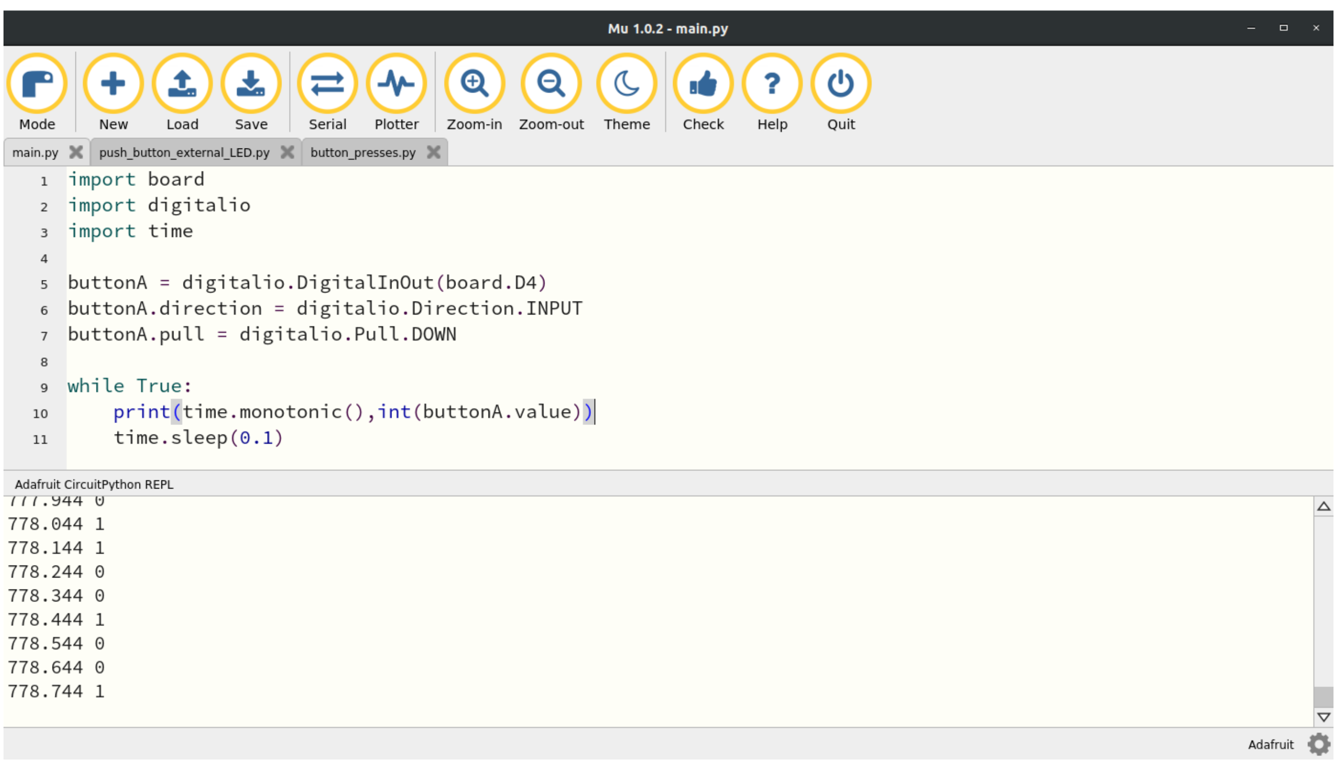
Notice the extra parentheses and the comma. Now if you click Plotter you'll see something like this. You'll notice that the print statement now has commas in it and the Plotter is recording button presses.



The problem with this is we still can't save the recorded data anywhere. Before we get into saving data let's first edit the print statement again to get rid of the Plotter by removing the extra parentheses and add time.monotonic() that way we can keep track of when a button was pressed. My print statement looks like this now:

```
print(time.monotonic(),int(buttonA.value))
```

Looking at the serial monitor now you'll see that time is being printed alongside the button presses.



Now we are in a position where we can record some data and save it to our computer. There are 4 ways to record data. I call the first, Method1 and you basically just copy and paste from the serial monitor, Method2 where you have the CPX/CPB type data into a spreadsheet and Method3 where you log data internally onto the CPX/CPB itself. The 4th method called Method4 utilizes the Bluetooth Module. Since that has it's own issues there is a completely separate section on how to explain Bluetooth (See section 9). Note you can only do Bluetooth if you have the Circuit Playground Bluefruit (CPB).

8.5 Method 1 - Copying Serial Monitor Data

If you open up the serial monitor you can see the data output. If you unplug the CPX while it's taking data the code will stop. **Note: In newer versions, unplugging your CPX will result in a loss of data. If this happens try pressing CTRL+C after you click the REPL window.** With the code stopped you can select all the data in the Serial monitor and then copy and paste the data into a text file on your computer. You can actually copy this into a new file on Thonny and just save it as a *.txt file. Once you have the data in a text file you can proceed to plotting in Python on your desktop which I discuss in the last section.Here's some example data in Gedit which is a simple text editing program.

8.6 Method 2 - Automatically Populate a Spreadsheet

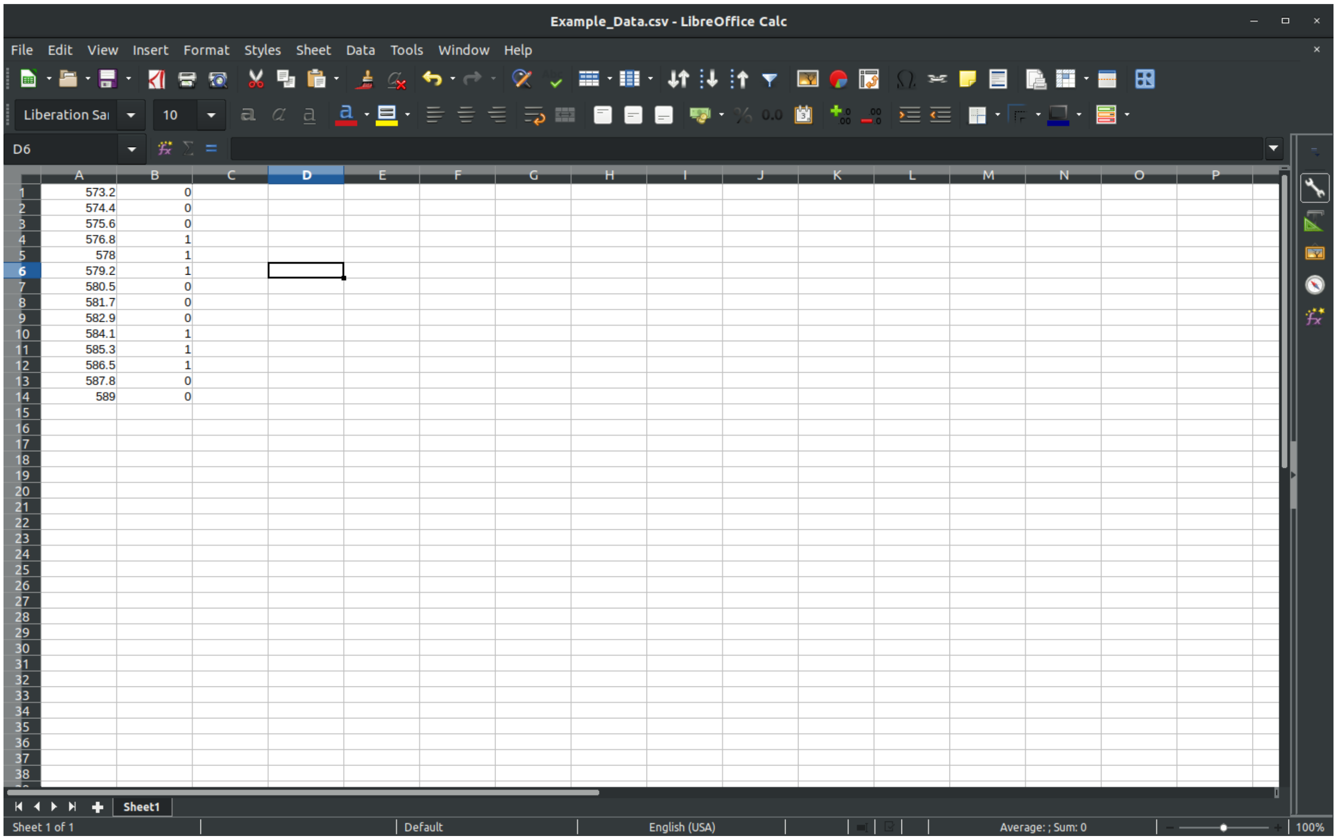
The downside with the above method of course is if you have a ton of data to record you could lose the data or run into a massive copy and paste issue. The second option is to use this module called keyboard which takes control of your keyboard on your desktop computer and actively types your data into a spreadsheet. The code is very extensive but I'll include the simple one here so we can discuss it. Below are the first 30 lines of code. The first 6 lines of code are just comments since I heavily adopted this code from the Adafruit Learn System. My version of this code can be found on my Github. Lines 8 - 14 are import commands as we've seen previously. The regular import modules board, time and digitalio are imported but we are also importing the Keyboard module so that the CPX can takeover our keyboard. Lines 16-22 create two buttons. First we create buttonA attached to pin D4 and then a switch attached to pin D7. If you look on the CPX there is a switch labeled D7. Before you copy this code onto the CPX make sure you move the switch towards the ear looking symbol. Lines 26-28 created the keyboard object. We are going to call it layout for this example code.

```
1   # Circuit Playground Express Data Logger
2   # Log data to a spreadsheet on-screen
3   # Open Spreadsheet beforehand and position to start (A,1)
4   # Use slide switch to start and stop sensor readings
5   # Time values are seconds since board powered on (relative time)
6   # Adapted from Adafruit Forums
7
8   import time
9   import digitalio
10  import board
11  import usb_hid
12  from adafruit_hid.keyboard import Keyboard
13  from adafruit_hid.keycode import Keycode
14  from adafruit_hid.keyboard_layout_us import KeyboardLayoutUS
15
16  buttonA = digitalio.DigitalInOut(board.D4)
17  buttonA.direction = digitalio.Direction.INPUT
18  buttonA.pull = digitalio.Pull.DOWN
19
20  switch = digitalio.DigitalInOut(board.D7)
21  switch.direction = digitalio.Direction.INPUT
22  switch.pull = digitalio.Pull.UP
23
24  # Set the keyboard object!
25  # Sleep for a bit to avoid a race condition on some systems
26  time.sleep(1)
27  kbd = Keyboard(usb_hid.devices)
28  layout = KeyboardLayoutUS(kbd)  # US is currently only option.
29
30  print("Time\tButton Value")  # Print column headers
```

The next 30 lines are shown below. Lines 32-35 define a function. Functions in Python have a pretty standard structure. The keyword def is used to denote that the next line is a definition for a function. The name of the function is *slow_write()*. The input to the function is string which ironically enough is a string object. Line 33-35 define what the function does. Line 33 sets up a for loop where the code loops through each character in the string. Everytime it gets to a new character it will use your keyboard to type that character using the layout.write(c) command. The time.sleep(0.02) is just to slow down the keyboard so your computer can keep up. That function is defined above the standard while True: statement on line 37 but is called on line 42. You'll see there is a *slow_write(output)* on line 42. In this case output is a string and it's sent to the function *slow_write()*. So in this case we have a function that can write a string so we just need to take data and then write it using our keyboard. Line 38 is an if statement that will only be true if the switch on pin D7 is pushed towards the music note on the CPX. If the switch is not thrown the code will move to the else statement on line 52 and tell the user that you need to flip the switch. If the switch is thrown line 40 will take data for us. First it will record the time.monotonic() and store it as a floating point number using the %0.1f designation which means that it will store 1 decimal as a %floating point number for f.

```python
32 def slow_write(string):   # Typing should not be too fast for
33     for c in string:      # the computer to be able to accept
34         layout.write(c)
35         time.sleep(0.02)   # use 1/5 second pause between characters
36 
37 while True:
38     if switch.value == True:
39         # Turn on the LED to show we're logging
40         output = "%0.1f\t%d" % (time.monotonic(), int(buttonA.value))
41         print(output)          # Print to serial monitor
42         slow_write(output)    # Print to spreadsheet
43 
44         kbd.press(Keycode.DOWN_ARROW)  # Code to go to next row
45         time.sleep(0.01)
46         kbd.release_all()
47         for _ in range(2):
48             kbd.press(Keycode.LEFT_ARROW)
49             time.sleep(0.005)
50             kbd.release_all()
51             #time.sleep(0.025)  # Wait a bit more for Google Sheets
52     else:
53         print('Not logging. Flip Switch to start logging')
54     # Change 0.1 to whatever time you need between readings
55     time.sleep(1.0)
```

The second number in the string is an integer or a base 10 (decimal) integer designated by the %d part of the format. The integer is int(buttonA.value). You'll see a n̂ between the formatted numbers which is a tab. The tab is there to tab between cells in a spreadsheet. Line 41 will print the output string to the Serial monitor and it will also type the contents of the string. Very important here. When you flip the switch on the CPX your keyboard will start typing in whatever active window is selected. If you don't have a spreadsheet opened and active (selected), the keyboard will just begin typing in whatever window is open. Make sure you have a spreadsheet program open and ready to go. Lines 44-51 tell the keyboard to hit the *DOWN_ARROW* on your keyboard to move to the next row and the *LEFT_ARROW* twice to move back to the first column. Line 55 is a sleep to only log data once a second. I ran this code for a bit and had it type into LibreOffice Calc which is a free spreadsheet program. Google Sheets or Microsoft Excel will also work just fine.



You'll see that the first column is time with 1 decimal point and the second column is the button press values. At this point you must click *Save As...* and save the document as a CSV which stands for Comma Separated Value. Once you have the file saved you can proceed to plotting in Python on your Desktop.

8.7 Method 3 - Logging Data Directly to on board memory

The problem with the above 2 methods is that you need a laptop to log data in the field. It would be nice if you could use the optional battery pack and just have the CPX log data on the CPX itself. This is the most complex way but in my opinion the best way. In order to get this to work you need to allow the drive on the CPX to have read/write permissions. This requires you to load a piece of software called boot.py and put it on the CPX. I have this software on my Github. The software is shown below. The first 10 lines are probably very familiar. Import some modules and then create a switch object. Line 13 is where all of the storage permissions are changed. If the flip is switched towards the A button, the storage module is used to allow you to write to the CPX. The problem here is that if you do this, you won't be able to edit code. I'll explain the procedure here in a minute. As always, the relevant Adafruit tutorial is on the Adafruit Learn System if you want to read more about it. Again make sure you store this file onto the CIRCUITPY drive and save it as boot.py

```python
import board
import digitalio
import storage

# For Gemma M0, Trinket M0, Metro M0/M4 Express, ItsyBitsy M0/M4 Express
#switch = digitalio.DigitalInOut(board.D2)
# switch = digitalio.DigitalInOut(board.D5)  # For Feather M0/M4 Express
switch = digitalio.DigitalInOut(board.D7)  # For Circuit Playground Express
switch.direction = digitalio.Direction.INPUT
switch.pull = digitalio.Pull.UP

# If the switch pin is connected to ground CircuitPython can write to the drive
storage.remount("/", switch.value)
if (switch.value):
    print('Storage changed')
else:
    print('Change the switch')
```

In addition to storing the file boot.py you'll need to edit your main.py script to only log data when the switch is moved towards the B button. The software to record button presses on disk is shown below and as always on my Github. In this software we again see the standard commands. Lines 1-3 import all the modules we need and then 5-15 create a switch, a button and an LED. In this case we're using the LED soldered to the board. Line 17-20 check to see if the user has flipped the switch. If the switch is False the storage module on boot.py will allow the drive to act like a data logger and it will open a file called *Test_Data.txt* for writing ('w'). If the switch is True then the user will be notified that the file has not been opened for writing. Lines 22 through 33 include the infinite while loop. Line 23 turns the LED on and line 24 prints out the current time and the button value in integer form. If the switch value is False the program will create an output string by converting all numbers to strings using the str function. Notice that there is a *str("\n")* at the end of the output variable which tells the computer to write a new line of data to the file. Lines 28 and 29 write the output to the file from line 18 and then flush the output which means the CPX waits for the data to be fully written before moving on. It also turns the LED off so we know the CPX took data even when we aren't looking at the Serial monitor. If the switch value is true it means that the we never opened the data file and thus we tell the user we aren't logging data and it's time to flip the switch and hit reset. **NOTE: The figure below is from an old version of the code. The newest version produces the same outcome. The only difference is that the D13 LED blinks when the code is running and the first neopixel toggles 3 different colors when the system is logging data. This allows for extra user information when operating Method 3 without a computer and the serial monitor. Note that these additions require the use of neopixel.mpy module.**

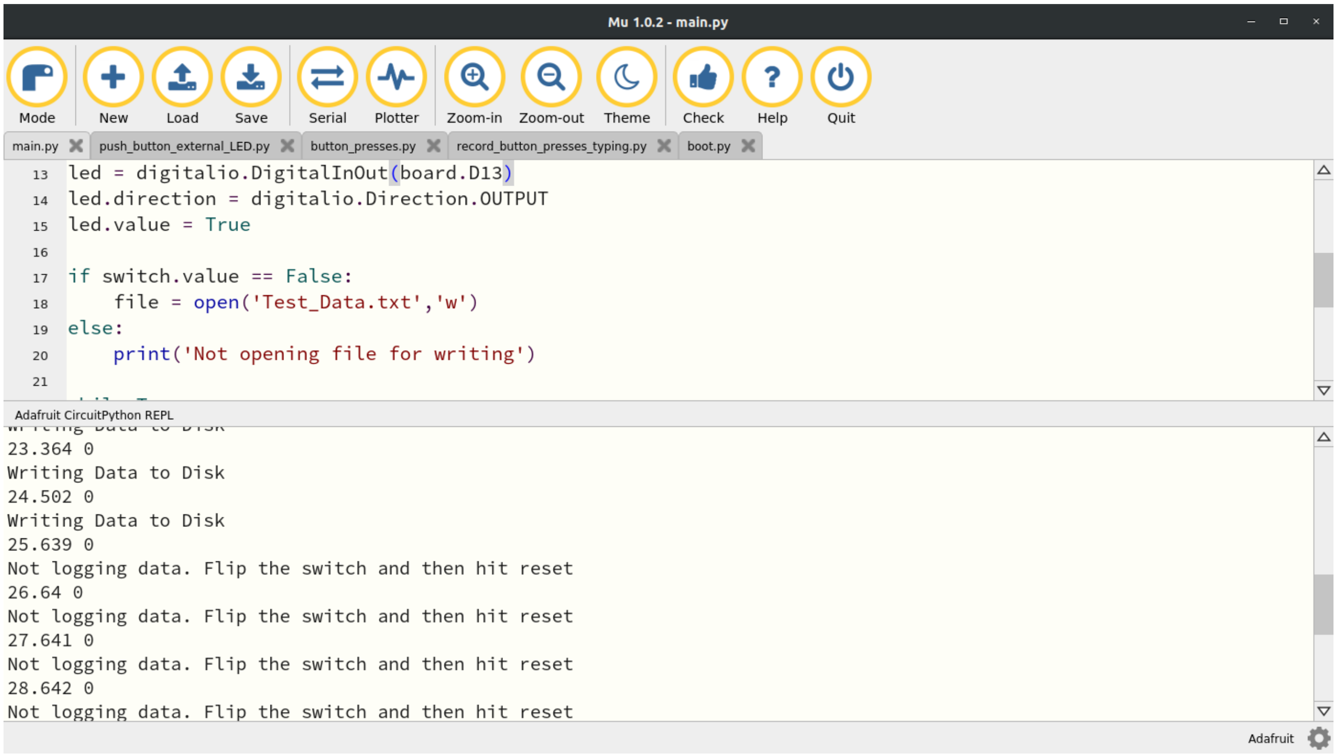
8.8 Installing Modules

One issue you're going to run into when you run the codes below is that you won't have some of the modules on your CPB/CPX. To fix this you need to download the CircuitPython Libraries. You need to download the appropriate version: 6.x, 7.x or 8.x. How do you know what version of CircuitPython you have? Well head over to your CIRCUITPY drive and open the boot_out.txt file and it will tell you the version. Note that this is the same version as the .UF2 file installed back in the Getting Started labs (See Chapter 5). When you download the modules it will download a .zip file. Extract the .zip file on your desktop computer and then open the *lib* folder on your desktop and your CIRCUITPY. You then need to transfer the modules (ONLY THE ONES YOU NEED) from your desktop to your CPX/CPB *lib* folder. The reason why you can't copy the entire folder is because the CPB/CPX only has 2MB of flash and the CircuitPython download is 4.1 MB at the time of this writing.

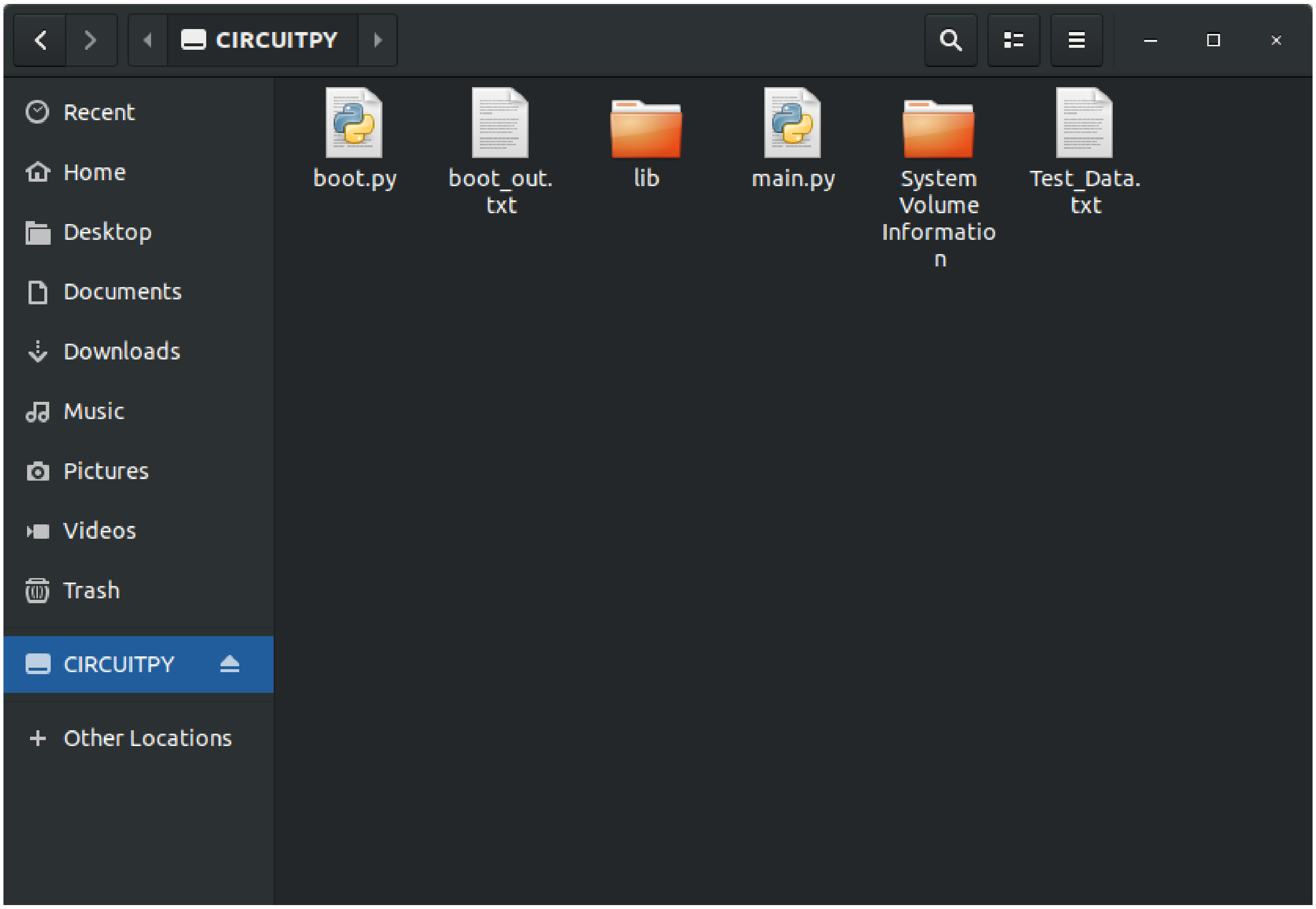
```python
import time
import board
import digitalio

switch = digitalio.DigitalInOut(board.D7)
switch.direction = digitalio.Direction.INPUT
switch.pull = digitalio.Pull.UP

buttonA = digitalio.DigitalInOut(board.D4)
buttonA.direction = digitalio.Direction.INPUT
buttonA.pull = digitalio.Pull.DOWN

led = digitalio.DigitalInOut(board.D13)
led.direction = digitalio.Direction.OUTPUT
led.value = True

if switch.value == False:
    file = open('Test_Data.txt','w')
else:
    print('Not opening file for writing')

while True:
    led.value = True
    print(time.monotonic(),int(buttonA.value))
    if switch.value == False:
        print('Writing Data to Disk')
        output = str(time.monotonic()) + " " + str(int(buttonA.value)) + str('\n')
        file.write(output)
        file.flush()
        led.value = False
    else:
        print('Not logging data. Flip the switch and then hit reset')
    time.sleep(1) #sleep for so many seconds between measurements
```

So here is the flow of what you want to do for method 3.

1. Unplug the CPX
2. Flip the switch towards the A button.
3. Plug in the CPX and save the boot.py and main.py files. Remember you can only save Python scripts when the switch is flipped towards the A button.
4. When you are ready to start recording data, flip the switch towards the B button. If you're looking at the Serial monitor, the software will throw an error. Just ignore it and hit the reset button. When your computer recognizes the CPX you can turn the Serial monitor on and off.
5. When you are done taking data simply slide the switch over towards the A button and hit reset again. This is what my Serial monitor looks like when I do this. You'll see that I was writing to disk for like 25 seconds and then I flipped the switch back towards the A button.



With the switch flipped and data taken, open your folder manager and take a look at the CIRCUITPY drive. This is what mine looks like. You'll see I have two Python files and a file *Test_Data.txt* with all my data in it.



If you open the *Test_Data.txt* file you will hopefully see data in it.

At this point you can copy this text file over to your desktop computer and proceed to the Python plotting portion. Ok so let's recap method 3 one more time.

1. Unplug CPX (or remove power)
2. Slide switch to A
3. Plug in CPX (or provide battery power) and wait for system to fully boot up
4. Slide switch to B
5. Hit Reset and wait for system to fully boot up
6. Take data for however long you want
7. Slide switch to A when you're ready to stop taking data
8. Hit Reset and wait for system to fully boot up
9. Remove power if you're on battery power
10. Plug CPX into computer if not already connected
11. Transfer data file to computer

8.9 Method 4 - Logging Data on a Cell Phone using Bluetooth (CPB Only)

As mentioned in the introduction it's possible to have the Circuit Playground send data wirelessly to a cell phone using Bluetooth provided you have Bluetooth setup and a smart phone with the Adafruit Connect App. Bluetooth is explained in detail in its own section (See Section 9). Method 4 is a valid form of logging data it just requires a cell phone to be powered the entire time and it must be within 30 feet of the Circuit Playground at all times. This also only works on the CPB since the CPX does not have a Bluetooth transmitter.

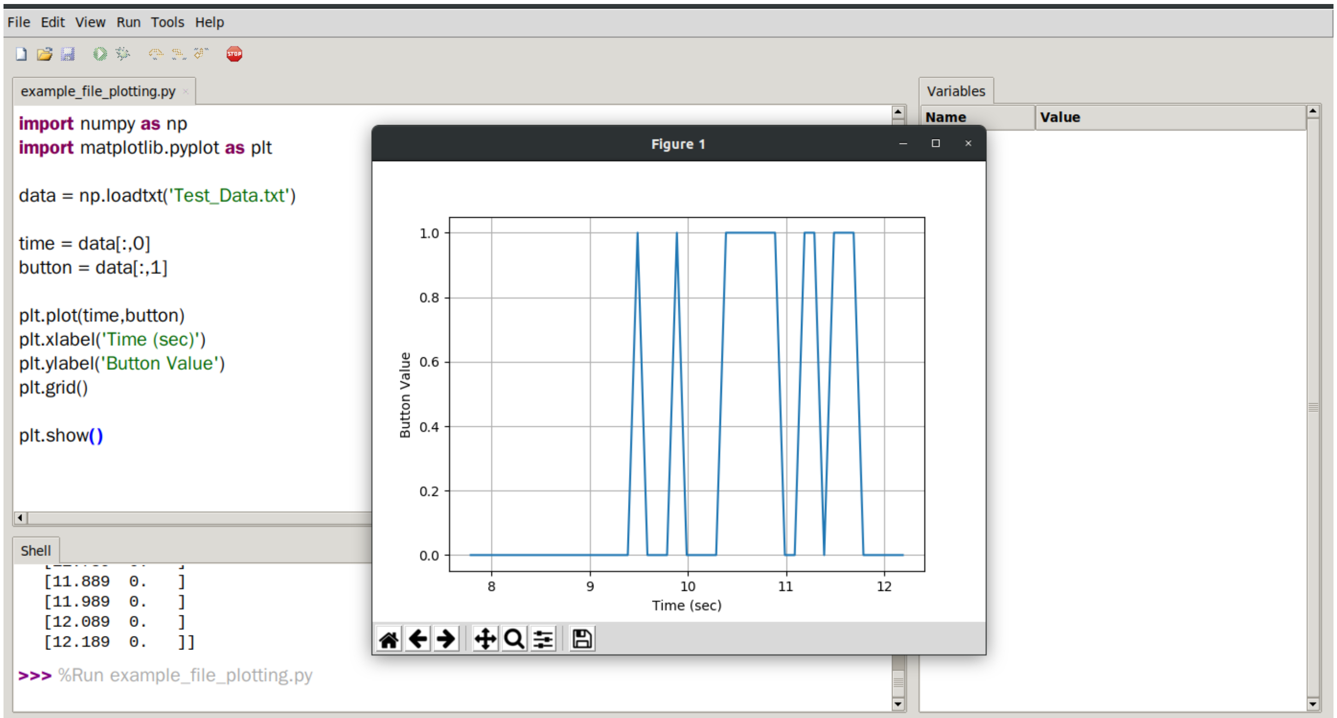
8.10 Plotting Logged Data

Alright so there you have it. I have explained 4 methods to datalogging. Here are the methods again in summary.

1. Print data to Serial and copy and paste
2. Use the Keyboard module to save data to a spreadsheet
3. Access the storage of your CPX and write data to a text file on the CPX
4. Send data wirelessly to a Cell Phone using Bluetooth - CPB Only (See Section 9)

All methods will work but some will obviously have their pros and cons. I suggest you get comfortable with 1 method and use that for the remainder of the semester. Whatever option you choose though will provide you with a data file that you can read in Python on your desktop computer to plot. The simplest way to import data is by using the loadtxt function from the module numpy. Here is some very simple code to plot data from a text file. I also have a Youtube video explaining how to plot a text file if you'd rather watch something.

When you plot make sure your *Test_Data.txt* file is in the same folder as your plotting script in Thonny or Spyder. Here's my example code (this code is not on Github but you only need 3 or 4 lines of code to plot).



In this example lines 1 and 2 import numpy and matplotlib. Line 4 imports data from the *Test_Data.txt* file and then 6 and 7 save the first and second columns into time and button. The remaining lines plot the data and create x and y labels as well as a grid. Hopefully now you are well versed in taking data and plotting in Python.

8.11 Assignment

For this project you must use method 1, 2, 3 or 4 to save time and button presses to a text file. You must then plot the button presses as a function of time. Remember to add x and y labels to all figures.

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%
5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. Include a snippet (5 lines) of your data file - 40%
2. Include a plot of your button presses with time on the x-axis and button presses on the y-axis (no screenshots) - 40%

9 Bluetooth on the CircuitPlayground Bluefruit - Method 4

9.1 Parts List

1. Smart Phone
2. Adafruit BLE Connect App (Play Store/App Store)
3. CircuitPlayground Bluefruit
4. USB Cable
5. Laptop

9.2 Learning Objectives

1. Understand the bluetooth module on the CircuitPlayground Bluefruit
2. Learn how to send data via the Bluefruit to your smart phone
3. Understand how to plot data sent via UART

9.3 Extra Help

You might find plotting data via bluetooth to be rather difficult and it was pretty difficult for me until I learned that you can export data as a txt file rather than a csv file. Before I learned how to do that I put together a 4 part series describing everything in this module. Worst case you can just watch the third video in the series. The video is 30 minutes but the first 8 minutes goes through setting up the bluetooth module and the rest of the video is just on plotting the exported csv data which took me some time. Note that exporting data as a txt file is the preferred method as parsing the file is way easier.

9.4 Installing Modules

One issue you're going to run into when you run the codes below is that you won't have some of the modules on your CPB/CPX. To fix this you need to download the CircuitPython Libraries. You need to download the appropriate version: 6.x, 7.x or 8.x. How do you know what version of CircuitPython you have? Well head over to your CIRCUITPY drive and open the boot_out.txt file and it will tell you the version. Note that this is the same version as the .UF2 file installed back in the Getting Started labs (See Chapter 5). When you download the modules it will download a .zip file. Extract the .zip file on your desktop computer and then open the *lib* folder on your desktop and your CIRCUITPY. You then need to transfer the modules (ONLY THE ONES YOU NEED) from your desktop to your CPX/CPB *lib* folder. The reason why you can't copy the entire folder is because the CPB/CPX only has 2MB of flash and the CircuitPython download is 4.1 MB at the time of this writing.

9.5 Getting Started

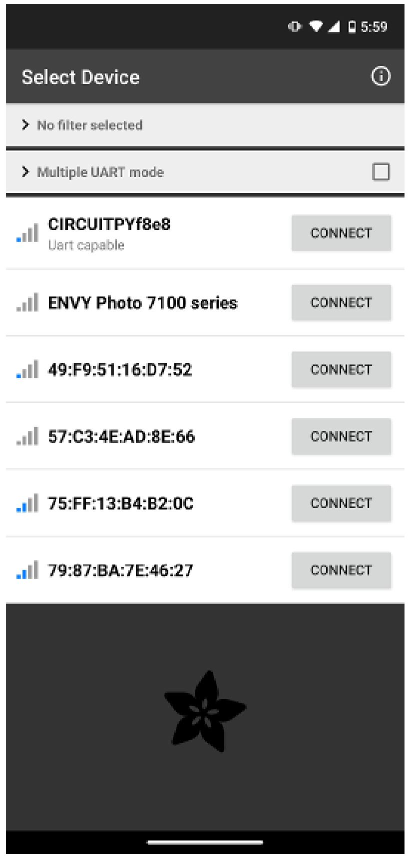
I mentioned in the DAQ project that there is technically a Method4. This is because with bluetooth you can send data from your phone to your Circuitplayground Bluefruit (CPB) and you can also send data to your smart phone. Once the data is on your phone you can export the data to a text file. That basically means you can use the bluetooth module as another method to save data from the CPB. There is a lot you can do with bluetooth but the bottom line is that All code required for this module is on my Github. First we're going to run the bluetooth_uart_button.py script which sends button data to your smart phone via something called UART which is a type of serial communication. It's beyond the scope of this lesson but serial is digital as opposed to analog which is done using the AnalogIn functions (See Chapter 10)

```
1  # CircuitPython Bluefruit LE Connect Button Send Example
2  ###Import Stuff
3  import board
4  import time
5  import analogio
6  from adafruit_ble import BLERadio
7  from adafruit_ble.advertising.standard import ProvideServicesAdvertisement
8  from adafruit_ble.services.nordic import UARTService
9  import busio
10 import digitalio
11 import math
12
13 ####Setup blue tooth
14 ble = BLERadio()
15 uart_server = UARTService()
16 advertisement = ProvideServicesAdvertisement(uart_server)
17 print('Bluetooth Setup')
18
19 ##SETUP Button
20 ##Button Presses
21 buttonA = digitalio.DigitalInOut(board.BUTTON_A)
22 buttonA.direction = digitalio.Direction.INPUT
23 buttonA.pull = digitalio.Pull.DOWN
24 print('Button Setup')
25 start_time = time.monotonic()
```

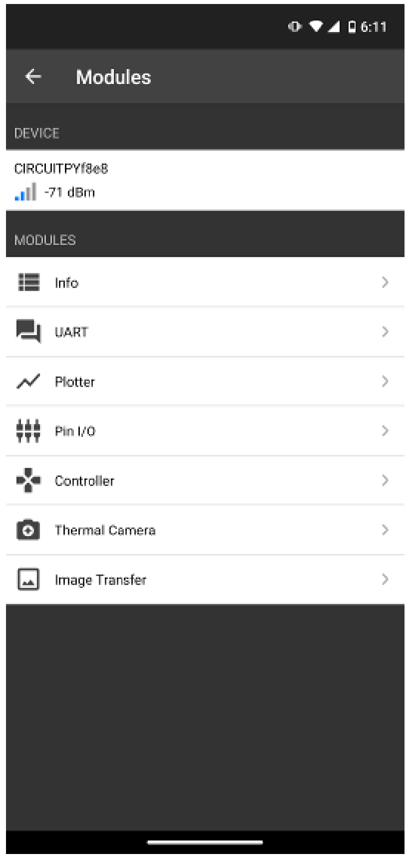
Lines 3-11 import a ton of modules. You'll recognize many of them like analogio, and time but the new ones are the ones that say ble. These are the bluetooth modules required for the CPB. Lines 21-24 setup the button so we can log the button via bluetooth and Lines 14-17 kick of the BLERadio object and the UARTService() to send data. Line 25 also grabs the current time before the infinite while loop that way the timer starts closer to zero.

```
26 while True:
27     # Advertise when not connected.
28     print('Not connected')
29     print('Look for',ble.name)
30     ble.start_advertising(advertisement)
31     ##Keep looping until connection established
32     while not ble.connected:
33         pass
34     #Stop advertising once connected
35     print('Connected')
36     ble.stop_advertising()
37
38     ##Once connected
39     while ble.connected:
40         #Time =
41         t = time.monotonic()-start_time
42         #Button press
43         b = int(buttonA.value)
44
45         #Print to STDOUT
46         print((t,b))
47
48         #And send them over uart (which is basically serial) but this is _server
49         uart_server.write('{},{}\n'.format(t,b))
50
51         #Sleep for 5Hz
52         time.sleep(0.2)# Write your code here :-)
```

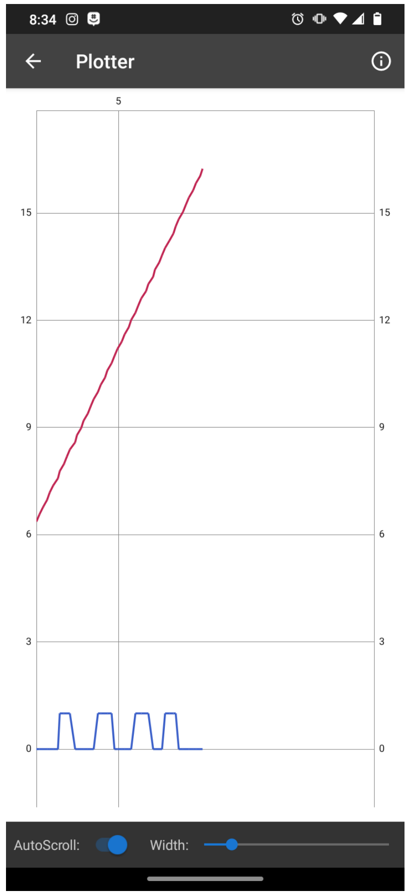
The code above is the infinite while loop which actually contains 2 while loops. Lines 28-33 prints the name of your bluetooth sensor and then starts advertising bluetooth to whoever is listening. It will then enter a while loop from 32-33 until bluetooth is connected. Once bluetooth is connected it will enter into the second while loop from line 39-52. In those lines 40-46 is responsible for taking all the necessary measurements and printing them to the serial monitor in Mu. In this case it's only printing the current time and the value of the button as an integer. *buttonA.value* is either True or False and the *int* function converts that to a 0 or a 1. Line 49 then sends the data over bluetooth using the UART server. You'll notice in this case the code is sending t,b by using the format variable and the 2 empty brackets. If you want to send more data you need to add more empty brackets and more variables to the format function. When you first save this script your CPB will not be connected and enter into an infinite while loop where it waits for your smart phone to connect. If you open your smart phone and open the Bluefruit Connect App the following screen will pop up.



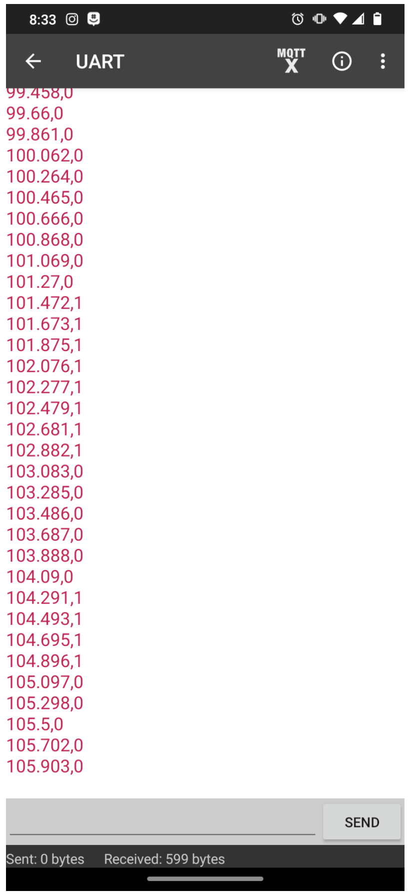
In this case there are numerous different bluetooth modules can be seen but the one you need to click is the one that says CIRCUITPYf8e8. You will have a different code after CIRCUITPY and you can figure out what your 4 digit code is by making sure you have the *print('Look for',ble.name)* in your code. Once you do that the CPB will begin sending time and the button press to your smart phone.



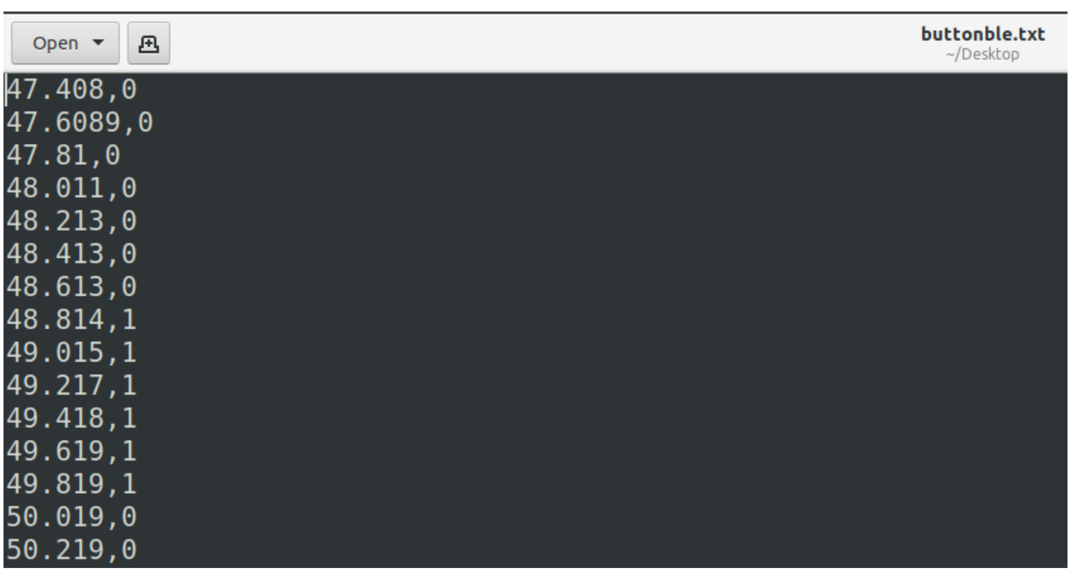
There are numerous items you can click. The Controller is very fun for creating a remote control robot but we're only going to go over the UART and Plotter tabs. If you click the plotter tab you will be greeted with a live screen of the data being sent.



In the photo above you can see three data streams that is coming directly from the CPB. The red line is time and the blue line is the button value. Notice the blue line goes from 0 to 1 which means I pressed the button a few times. The red line is always increasing which kind of messes up the plotter so you can always go back to your code in Mu and just send your data. This is great for live demonstrations and for debugging if you need to see data from an experiment and you don't have access to a laptop with Mu. If you hit the back arrow and then click UART you will see the raw data come in as text.



Again here you can see the 3 data streams separated by commas. The very neat thing with the UART tab is that you can click the three vertical dots in the upper right hand corner and click export to TXT. The easiest thing for me was to export the data to google drive and then download the data to my computer. Once I downloaded the TXT file to my computer and opened it the data file looked like this.



If you export the file as a CSV the data file will look completely different and it's much more complicated to plot. If you export the data as a TXT file you just need to use the np.loadtxt command to read in the data. Note you might have commas in your data file. If there are commas just use the CTRL+H command and replace all commas with spaces or use the np.loadtxt('buttonble.txt',delimiter=',') command. Plotting your button presses should be as simple as the previous lab thus plotting the button is left as an exercise to the reader.

9.6 Assignment

This project is similar to the Data Acquisition project only you must use method 4 to save time and button presses to a text file. You must then plot the button presses as a function of time. Remember to add x and y labels to all figures.

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%
5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. Include a screenshot from your phone showing the raw UART output - 20%
2. Include a snippet (5 lines) of your data file - 20%
3. Include a plot of your button presses with time on the x-axis and button presses on the y-axis (no screenshots) - 40%

10 Measuring Voltage Across a Potentiometer

10.1 Parts List

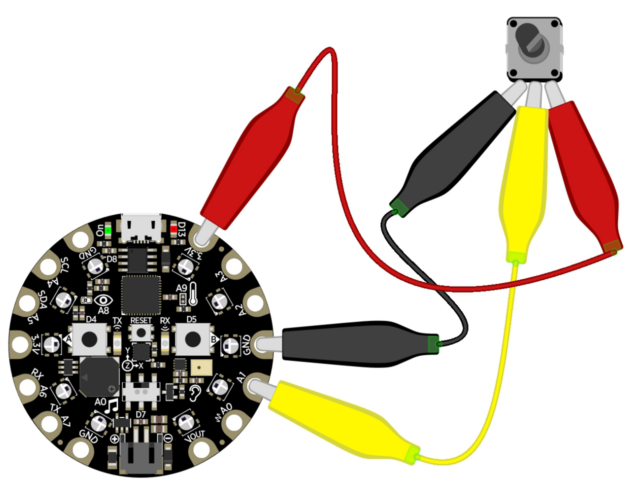
1. Laptop
2. CPX/CPB
3. USB Cable
4. Potentiometer
5. Resistor (the Ohms depends on how large your potentiometer is)
6. Breadboard

10.2 Learning Objectives

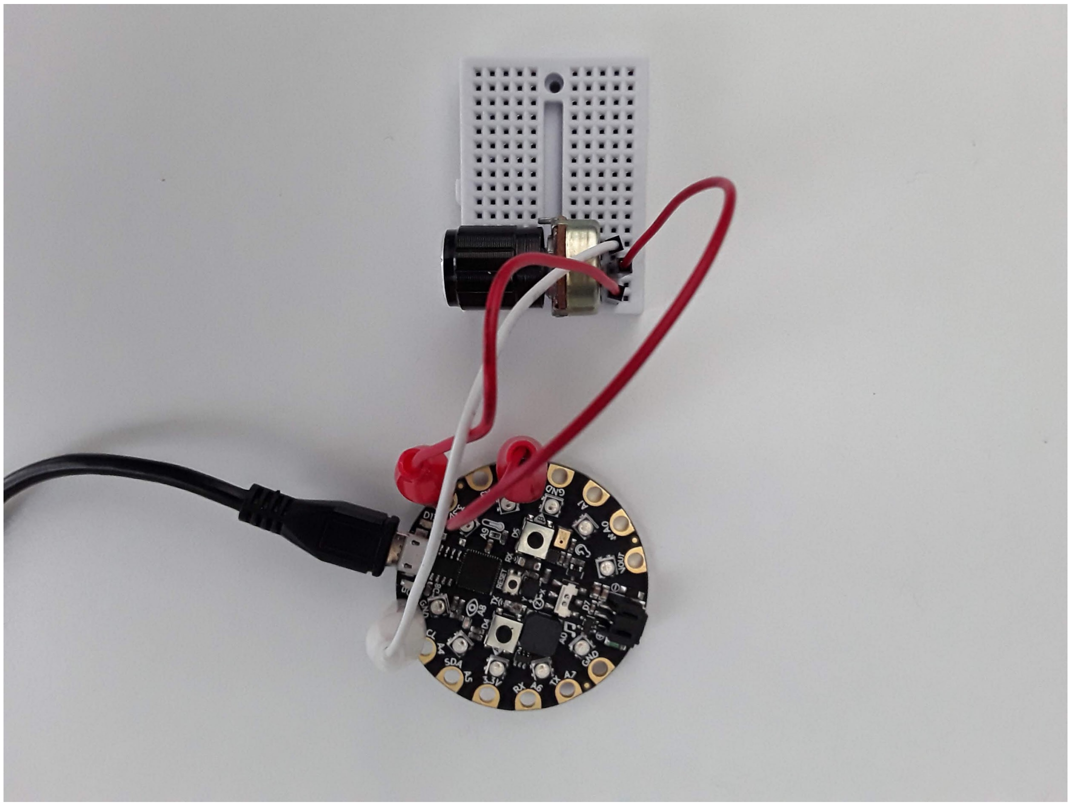
1. Understand voltage division of resistors in series
2. Measure an analog signal on the CircuitPlayground
3. Understand the binary measurement done by the analog to digital conversion (ADC)

10.3 Getting Started

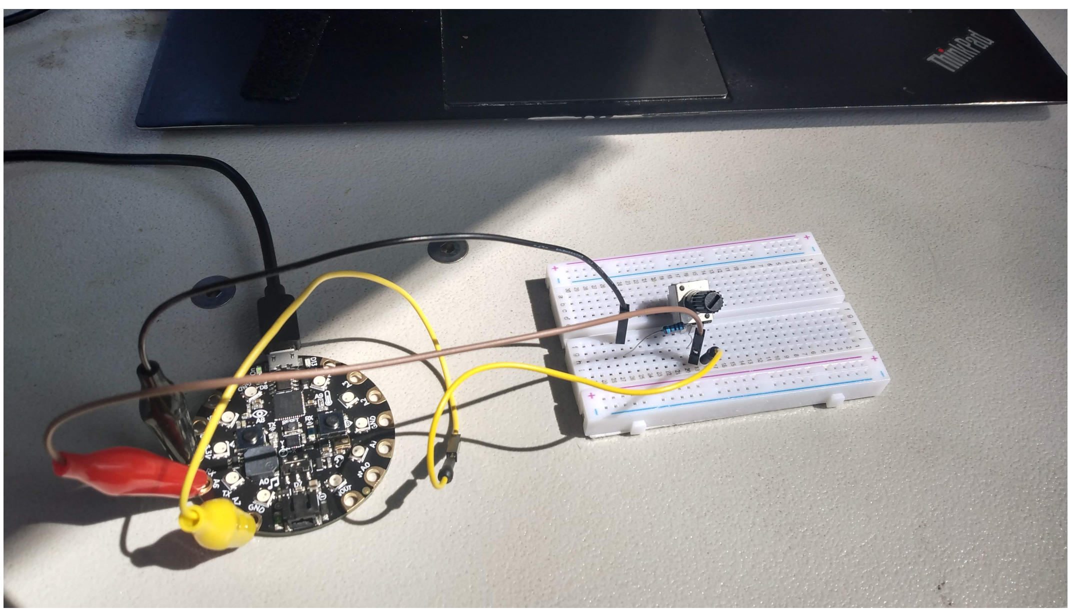
At this point you've learned about analog to digital converters (ADC). It turns out that the CPX has 8 analog ports hooked up to a 3.3V logic 16 bit ADC. The input range on the ADC is 0 to 3.3V and the output range is 0 to 65536 which is 2^{16} hence 16 bits. In order to get accustomed to the ADC on 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 potentiometer examples. I've done this lab with a few potentiometer. Ideally you'd like to have the potentiometer hooked up in series with another resistor so that you end up building a voltage divider but it' possible you can do it without it as shown in the figure below (Courtesy of Kattni Rembor).



Here's my circuit all hooked up without a resistor in series. Two legs are connected to 3.3V and GND while the middle leg of the potentiometer is connected to pin A2.



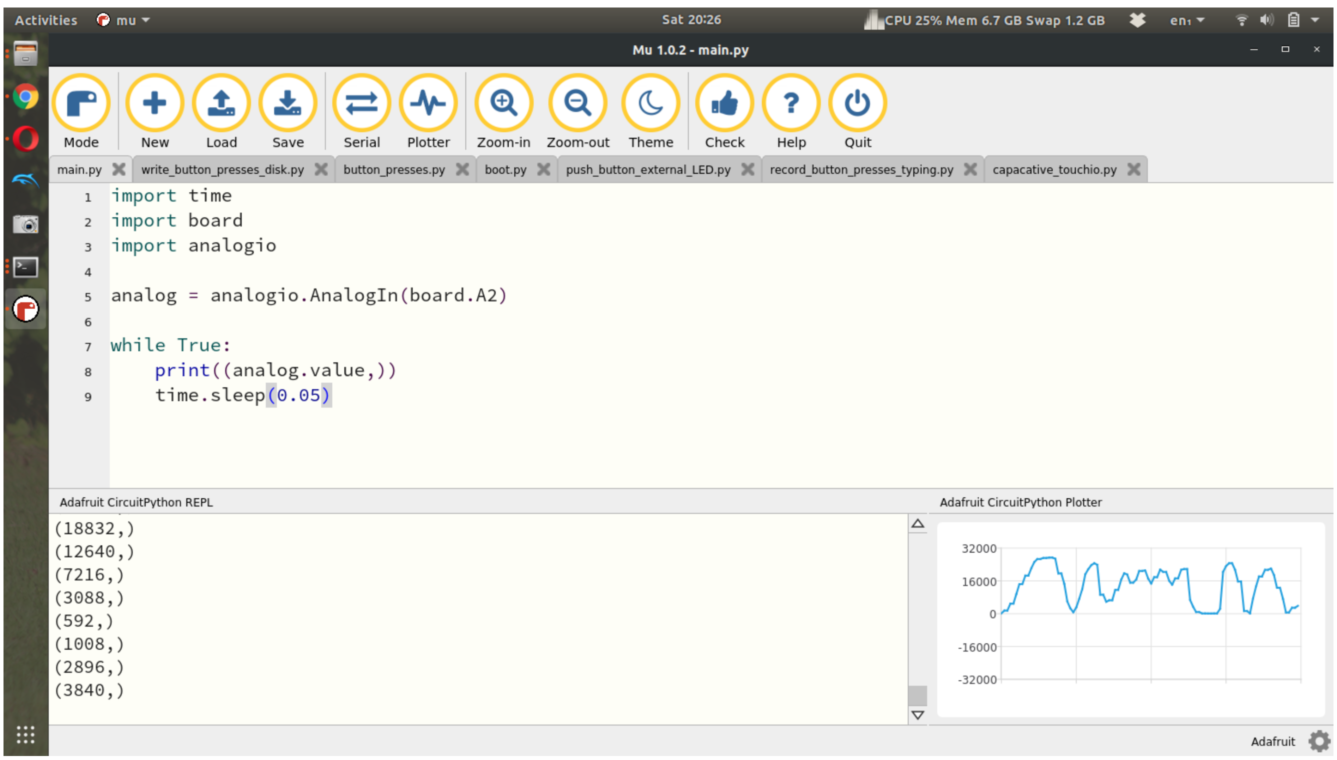
As I said before, some potentiometers do not have enough resistance when turned all the way down. I suggest that you put a resistor in between the third leg and ground. Some experimenters have melted plastic or gotten really hot. One student even blew up a potentiometer. Here is my circuit with a resistor in series.



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*.

```python
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.



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.

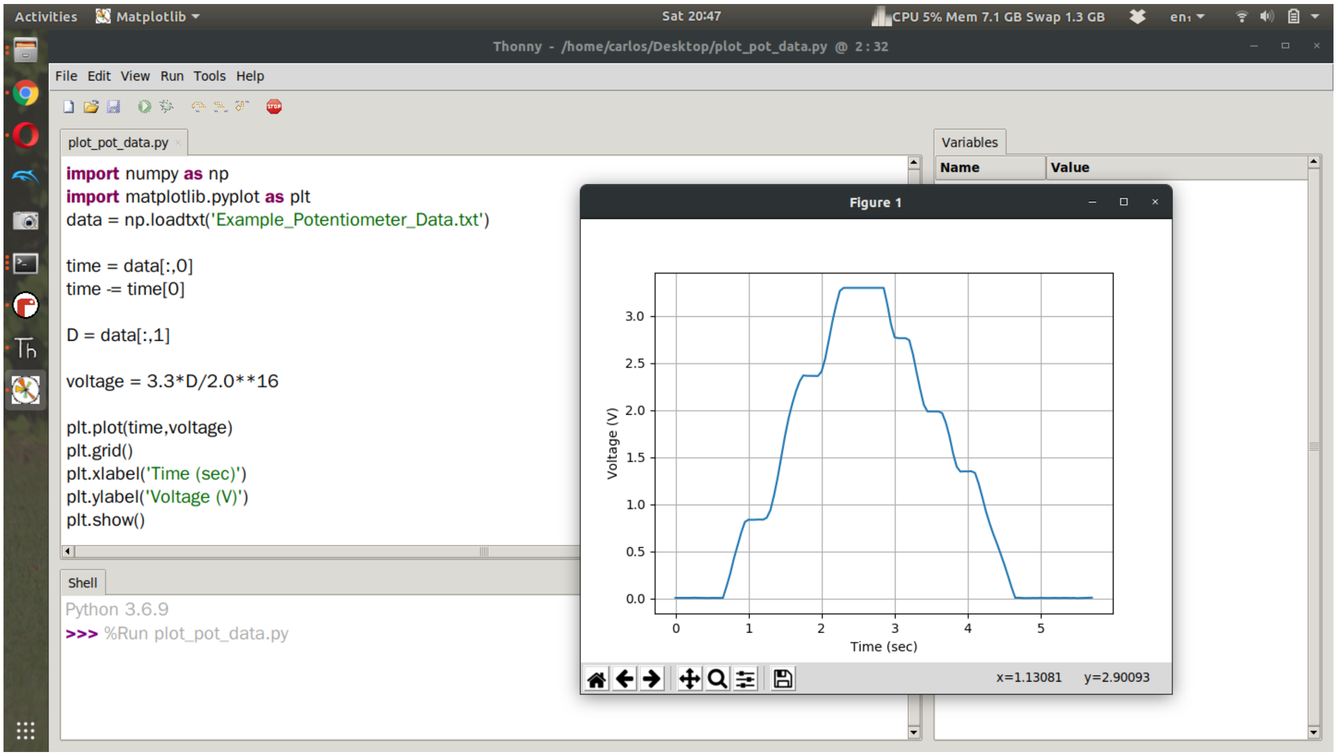
$$V = \frac{3.3D}{2^{16}} \tag{2}$$

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 8). 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.



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.

10.4 Assignment

For this assignment you are to wire up a potentiometer and read the voltage across the potentiometer using the analog to digital converter on the CPX. You are to record both time and voltage as you rotate the potentiometer and then plot that as a function of time. Specific requirements are shown below.

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%
5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. Include a photo of your circuit showing the potentiometer wired up to an analog pin on your CPX/CPB - 10%
2. Include a screenshot of Mu with the Plotter open showing the digital output of the potentiometer. The code in Mu also needs to also show the same analog pin as your potentiometer. - 10%
3. Based on the digital output from the potentiometer, compute the minimum and maximum voltage across the potentiometer. - 10%
4. Relate the voltage across your potentiometer to angle of the potentiometer in degrees and plot degrees on the x-axis and voltage across the potentiometer on the y-axis - 10%
5. Plot your digital output (raw potentiometer analog value) vs time - 10%
6. Then convert your digital output (Do) to voltage and plot that vs time - 10%
7. Finally convert your voltage to angle in degrees and plot that vs time - 20%

11 Wind Speed from Pitot Probe

11.1 Parts List

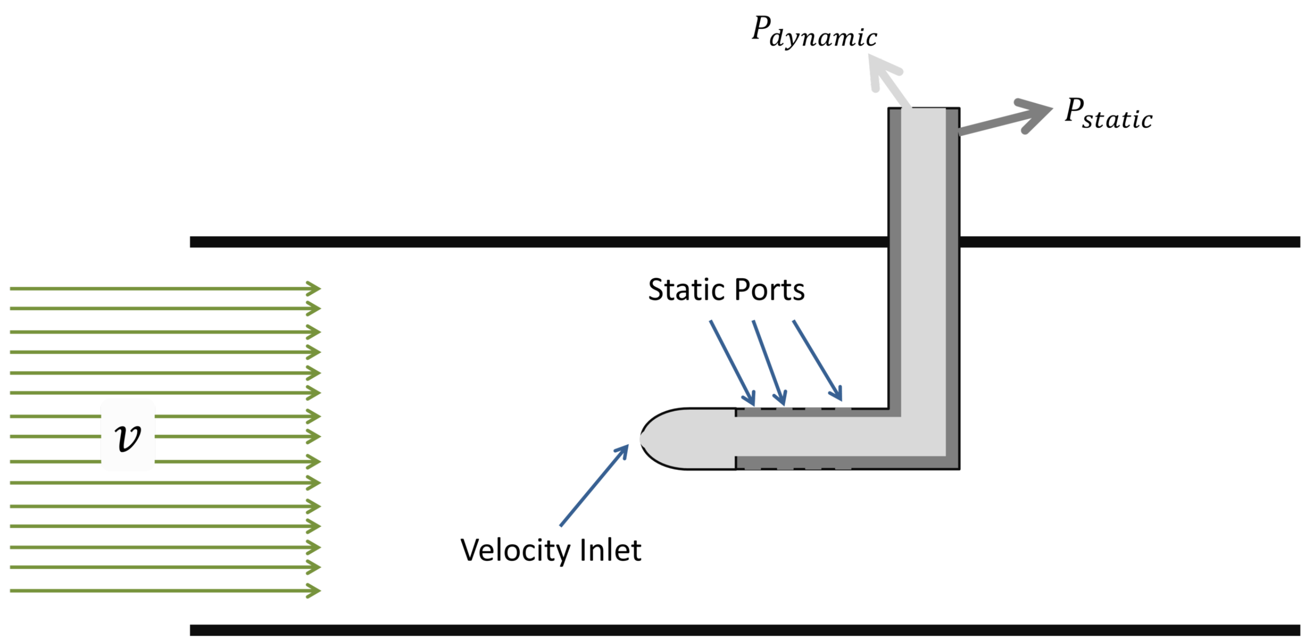
1. Laptop
2. CPX/CPB
3. USB Cable
4. Alligator Clips (x3)
5. Pitot Probe (Not included in kit at the moment so will need to buy this separately or borrow one)
6. Breadboard

11.2 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

11.3 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. First, pitot probes work by mechanically changing the dynamic pressure of the incoming airflow as shown in the figure below (Courtesy of Joshua Hrisko).



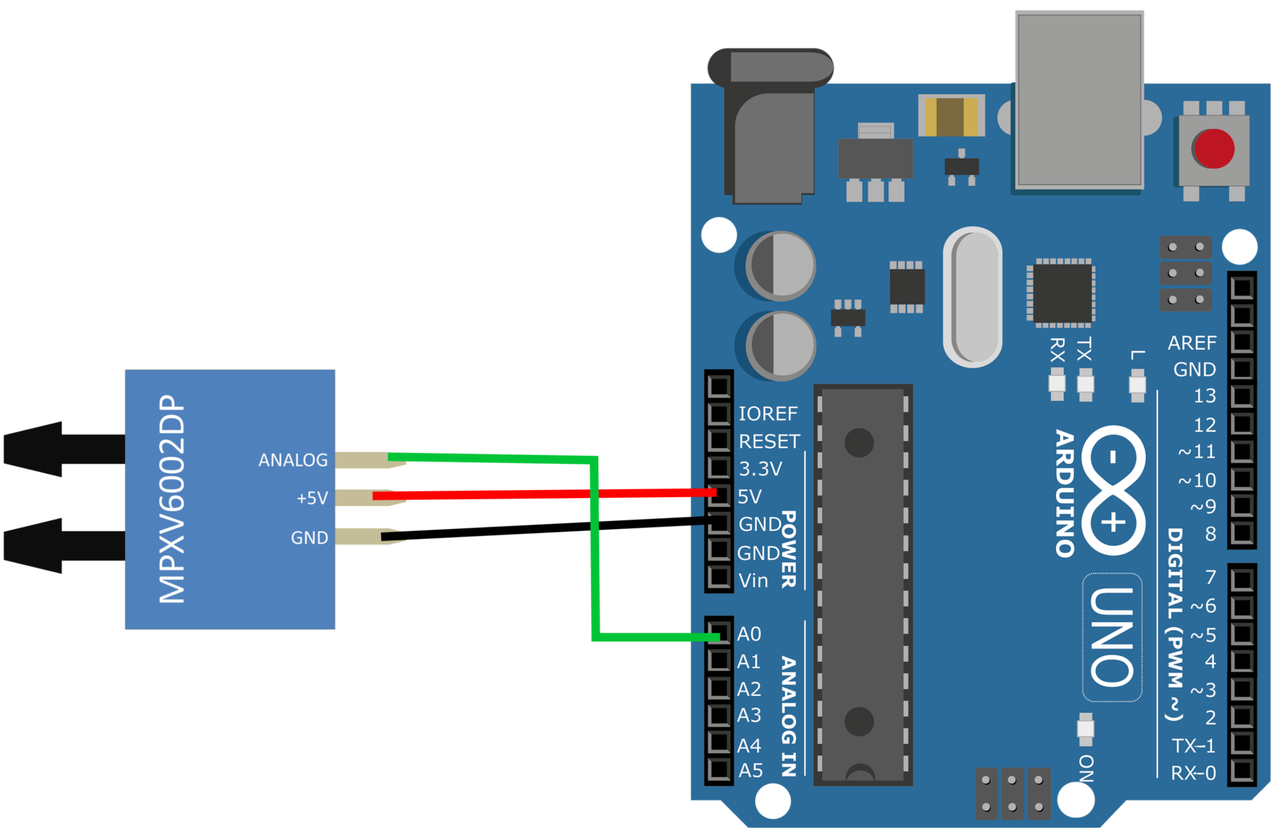
The pitot probe has two pressure taps that measure static (ambient) pressure and dynamic (stagnation) pressure. These taps move through two silicon tubes to a pressure transducer that has a strain gauge that separates both pressures. When the pressure on one side of the transducer is larger than the 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 analog signal from the analog pin will be denoted as V_{pitot}. The pressure transducer used in this lab is the MPXV7002 which is a differential pressure sensor. The datasheet indicates that the voltage offset of the pitot probe is 2.5V and the change in voltage is proportional to the change in pressure in units of kPa. This means that ΔP is given by the equation below.

$$\Delta P_{kPa} = V_{pitot} - 2.5 \tag{3}$$

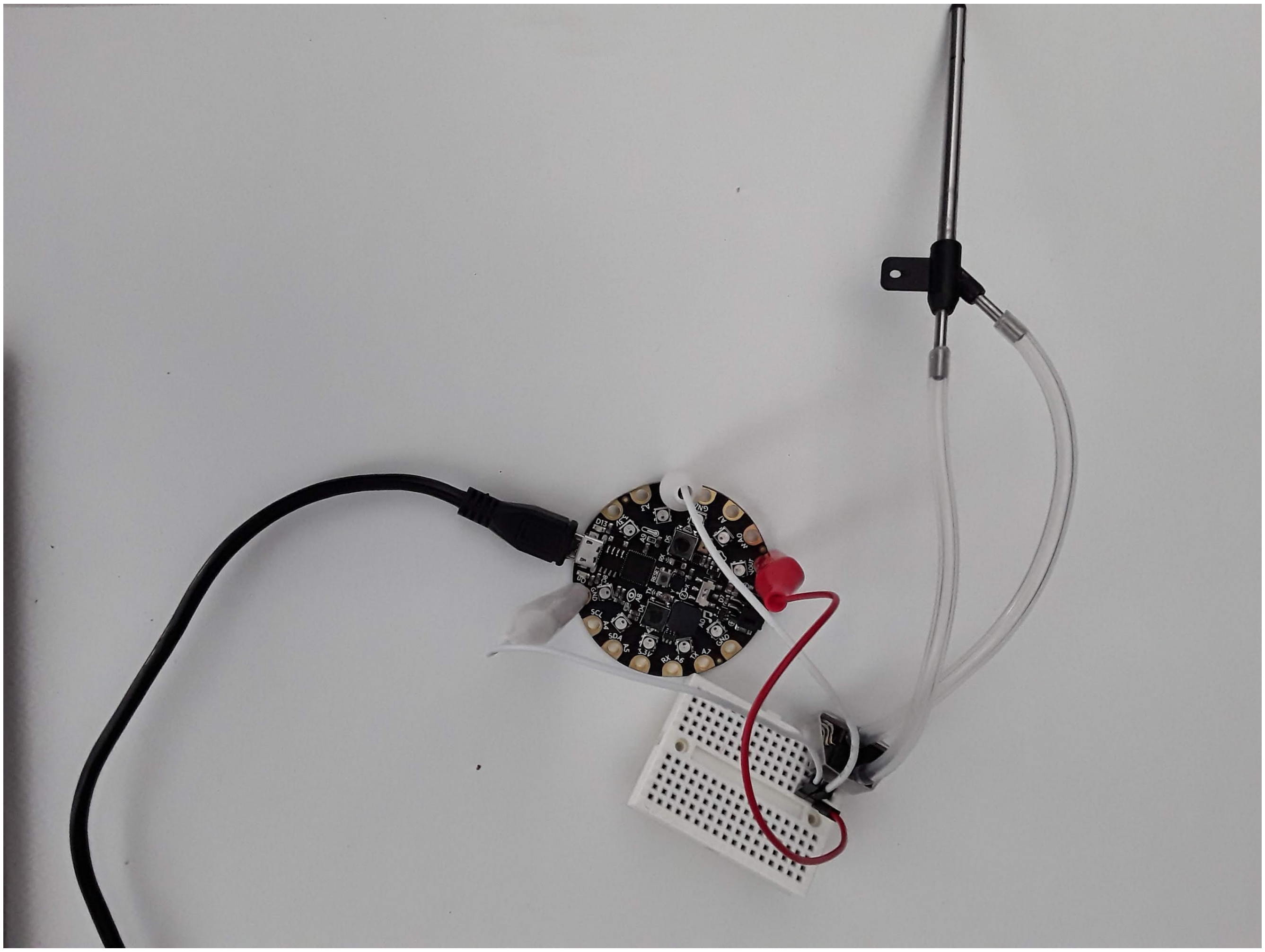
remembering that $V_{pitot} = 3.3D_o/65535$. Also note that 2.5V is the nominal voltage even though your sensor might have something slightly different like 2.4V or 2.6V. I explain the process of substracting off bias in this accelerometer video. To measure wind speed you can use a variation of Bernoulli's principle. Remember to convert the pressure P from kilopascals (kPa) to Pascals (Pa) by multiplying by 1000 before using it in this formula.

$$U = \sqrt{\frac{2\Delta P_{Pa}}{\rho}}$$

where U is the wind speed in m/s, ΔP is the differential pressure (in Pascals), and $\rho = 1.225\ kg/m^3$ is the density of air. I've done this pitot 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()).* As for wiring up the circuit itself, the transducer has 3 pins, +5V, GND and Analog. The figure below shows the transducer connected to an Arduino (Courtesy of Pinterest).



It is pretty straightforward how to wire this up to a CPX/CPB but remember that +5V needs to go to VOUT, GND to GND and Analog to any analog pin. I chose pin A2 as shown in the Figure below.

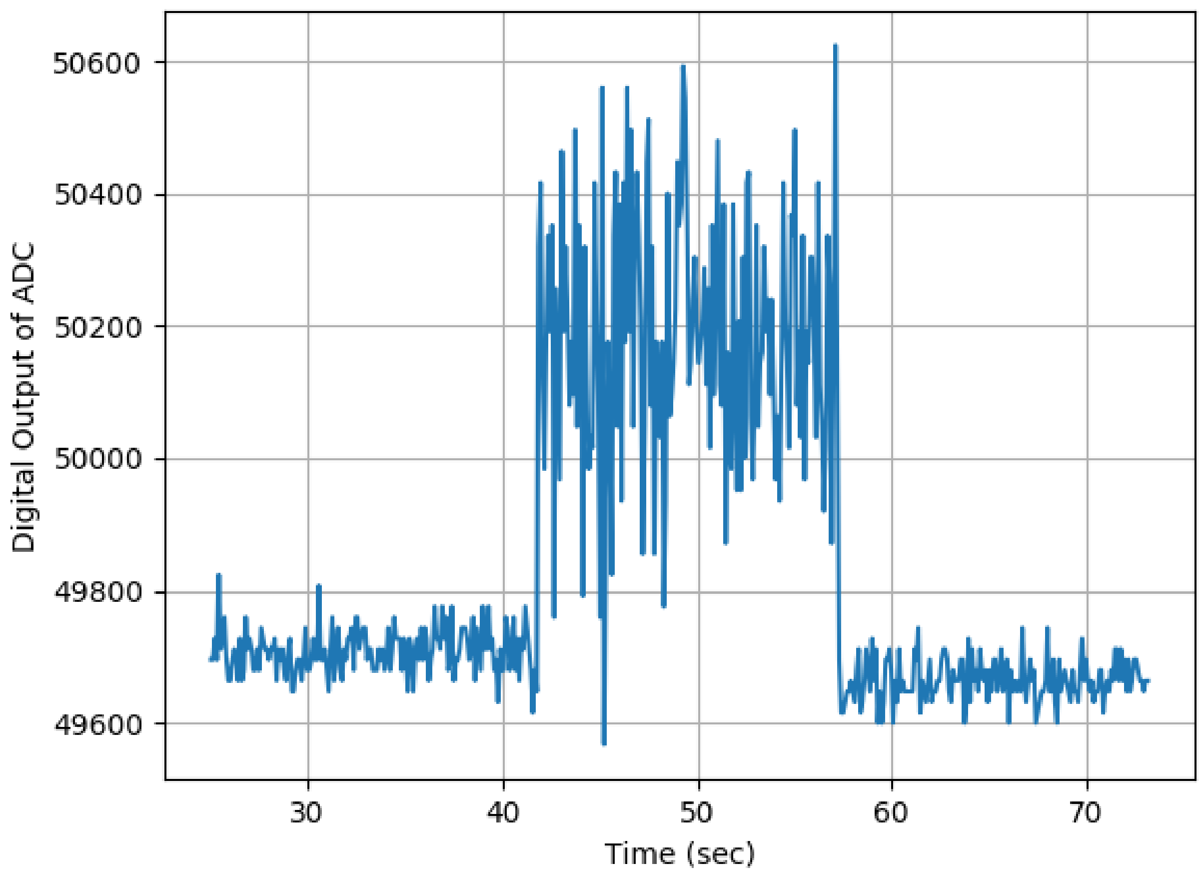


At that point it's very simple to just print the analog signal in bits to Serial. I've done this below. The code is the same analog code that we've used in the past.

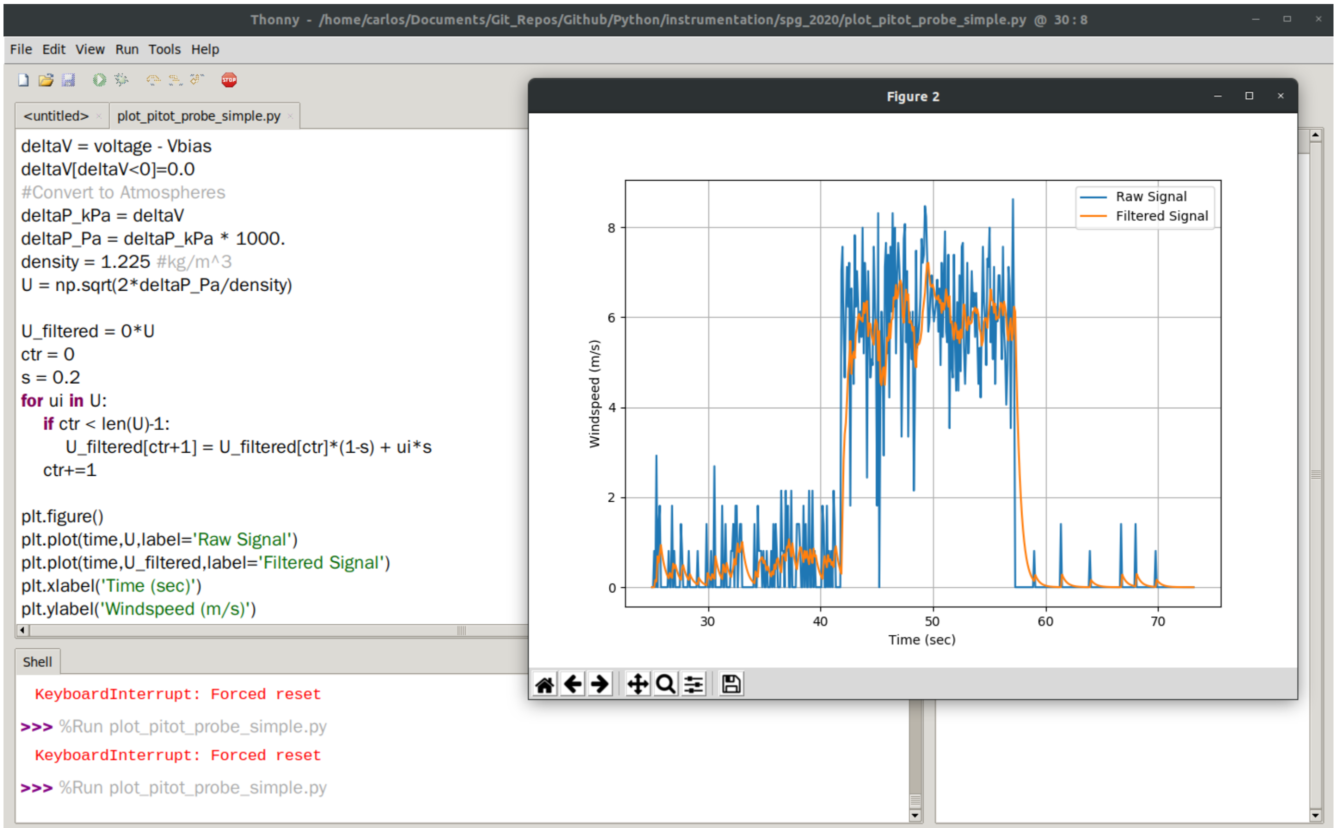
```
import time
import board
import analogio

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 8) 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 as explained above. 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.



11.4 Assignment

For this assignment you are to wire up a pitot probe and record time and raw analog signal from the pitot probe as you turn a fan on and off. My suggestion is you record at least 30 seconds of data with the fan off and then 30 seconds with the fan on and then again 30 seconds with the fan off. Specific requirements are shown below.

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%
5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. In addition to the standard format above, you must also return the pitot probe you borrowed in class - Pass/Fail
2. Include a photo of your circuit with your pitot probe wired up to an analog pin - 10%

3. Include a screenshot of Mu with the Plotter open showing the raw analog signal. The Mu code also needs to show the same analog pin as the circuit above - 10%
4. Include a plot of the raw analog signal vs time that clearly shows when the fan is on and off - 20%
5. Convert your raw signal to windspeed and filter your signal using an offline complementary filter. Plot both unfiltered and filtered windspeed on the same plot and include a legend. Make sure you add the saturation filter to prevent a negative in the square root - 20%
6. What is the maximum windspeed that the CPX/CPB can read? - 10%
7. The data sheet also suggests that you use a 470 pF capacitor to filter the output. Select two resistors for a low pass filter such that the DC Gain would be equal to 1. Assume the total impedance of both resistors is 2 $k\Omega$. Also compute the cutoff frequency. - 10%

12 Circuit Playground (CPX/CPB) Modules

12.1 Parts List

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

12.2 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

12.3 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

12.4 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**.

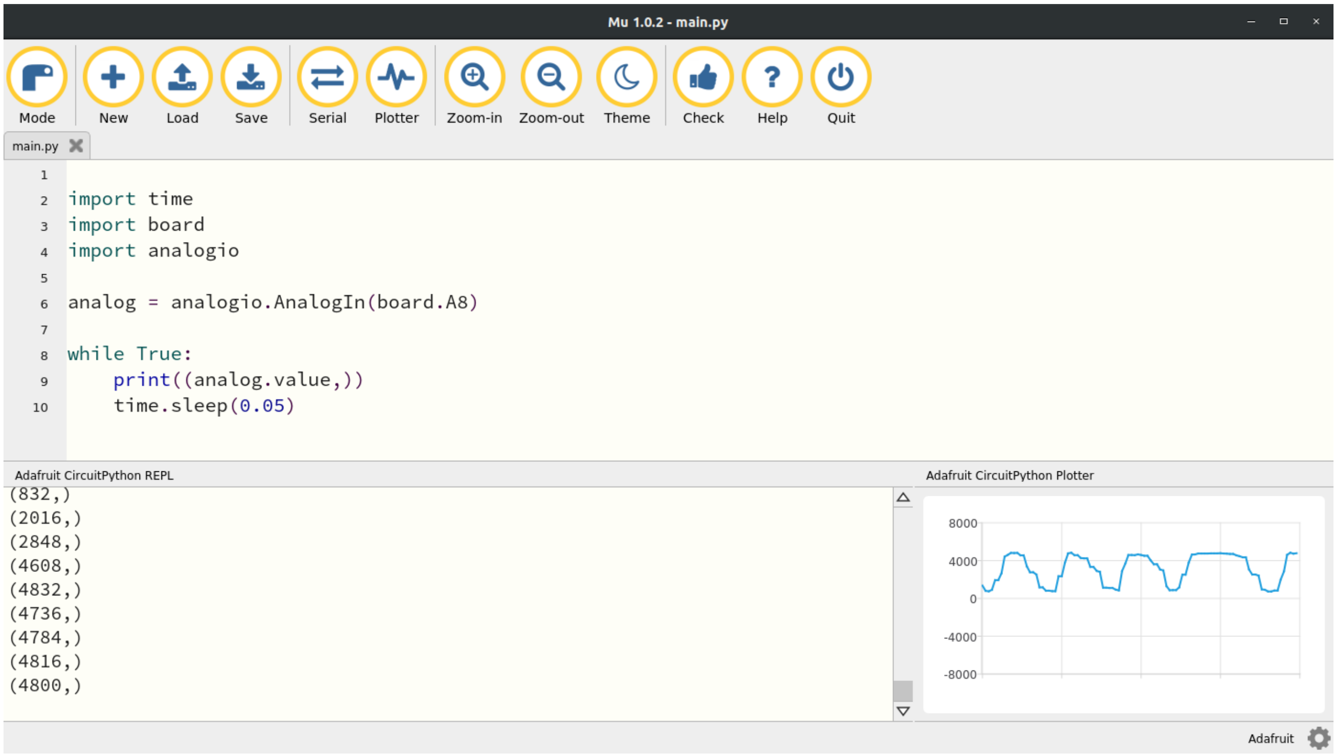
12.5 Installing Modules

One issue you're going to run into when you run the codes below is that you won't have some of the modules on your CPB/CPX. To fix this you need to download the CircuitPython Libraries. You need to download the appropriate version: 6.x, 7.x or 8.x. How do you know what version of CircuitPython you have? Well head over to your CIRCUITPY drive and open the boot_out.txt file and it will tell you the version. Note that this is the same version as the .UF2 file installed back in the Getting Started labs (See Chapter 5). When you download the modules it will download a .zip file. Extract the .zip file on your desktop computer and then open the *lib* folder on your desktop and your CIRCUITPY. You then need to transfer the modules (ONLY THE ONES YOU NEED) from your desktop to your CPX/CPB *lib* folder. The reason why you can't copy the entire folder is because the CPB/CPX only has 2MB of flash and the CircuitPython download is 4.1 MB at the time of this writing.

12.6 Low Level Control

12.6.1 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 16 if you'd like to do that lab first to learn more about photocells. This lab is designed to simply teach you about the onboard sensors rather than how they work. 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 10) 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.

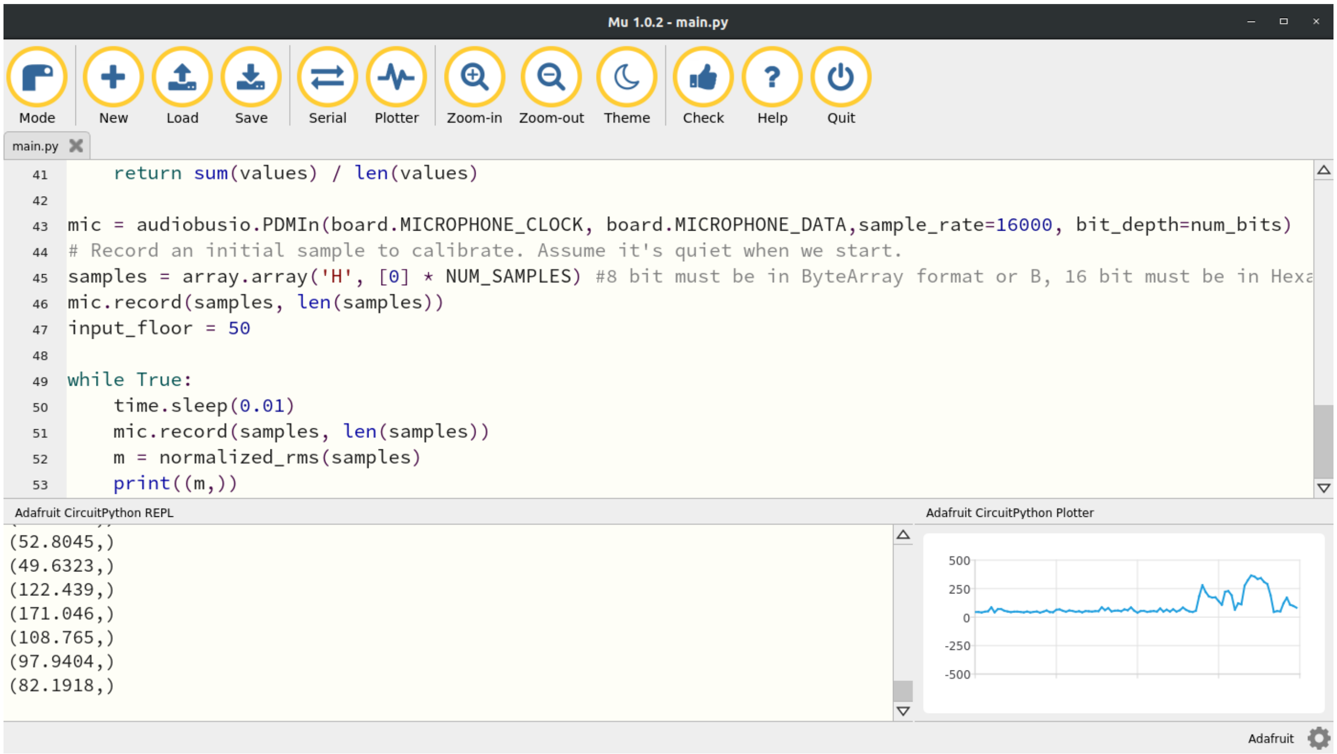


12.6.2 Sound

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 but then there are noticeable humps in the data. This is me saying something random into the microphone at normal volume. 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.



12.6.3 Temperature

The temperature sensor is actually a thermistor. A thermistor is basically a thermometer resistor which means the resistance depends on temperature. Since this thermistor on the CPX/CPB is connected to an analog pin, 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.

In this case, the thermistor is wired in a voltage divider circuit in series with a 10000 Ω resistor. This means that the voltage across the thermistor can be converted to its resistance using the voltage divider equation below similar to the equation used for the photocell.

$$V_{thermistor} = V_{out}\frac{R_{thermistor}}{R_{series} + R_{thermistor}} \tag{4}$$

Where $V_{out} = 3.3V$ and $R_{series} = 10000\ \Omega$. The equation above can be inverted to obtain the resistance in the thermistor which can then be converted to Kelvin. This means that the digital output from the analog to digital converter can be converted to voltage and then to resistance as has been done for various ADC labs in this textbook. Remember that $V_{thermistor} = V_{out}D_o/65535$ where D_o is the digital output from the ADC. Once you have the resistance from the thermistor, you can use a modified version of the Steinhart-Hart equation which is given below.

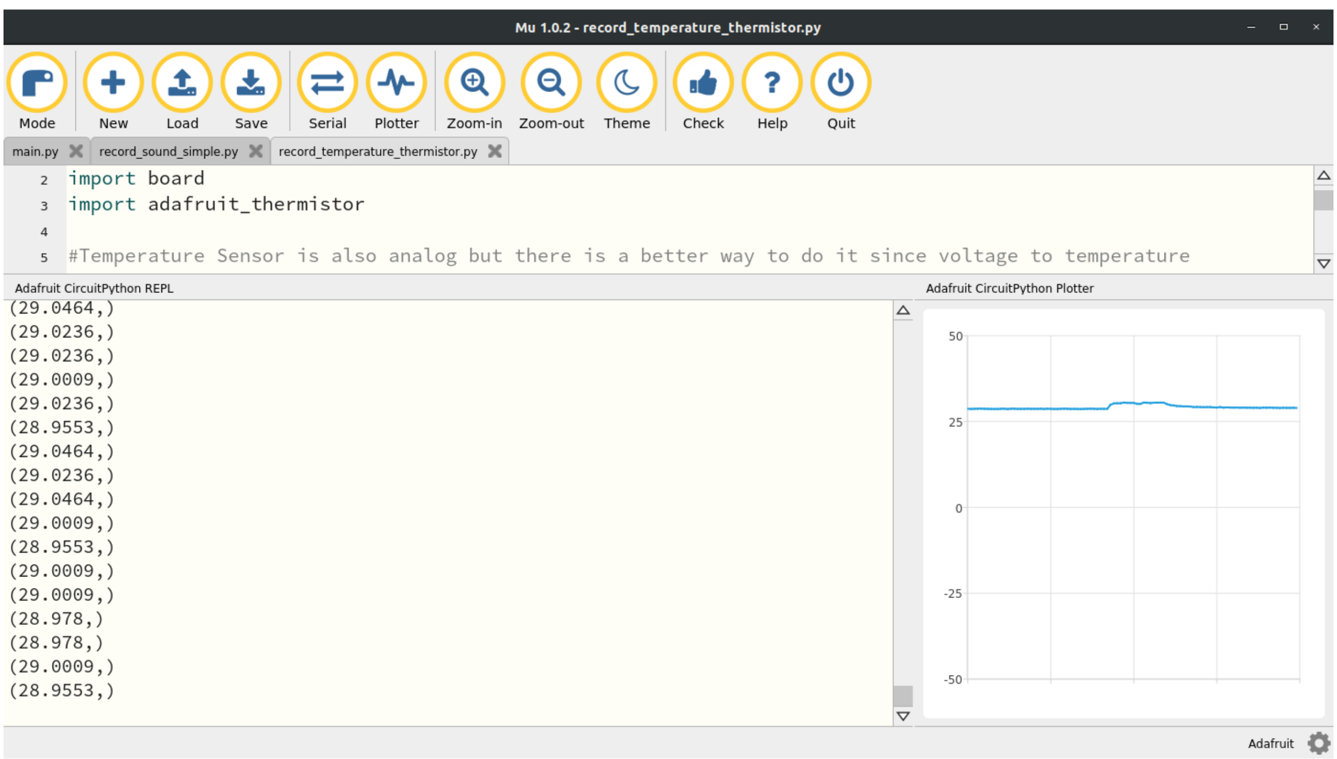
$$T_{thermistor} = \left(\frac{1}{T_0} + \frac{1}{\beta}log(R_{thermistor}/R_{series})\right)^{-1} \tag{5}$$

where $\beta = 3950$ is a heat transfer coefficient specific to the bulk semiconductor material over a given temperature range of interest and $T_0 = 298.15K$ is the nominal temperature of the semiconductor in Kelvin. Note that in the equation above, the output is in Kelvin.

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 but I have also summarized it above and in Chapter 18. For now though we will just appreciate the simplicity of the code below which is also on my Github.

```python
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.

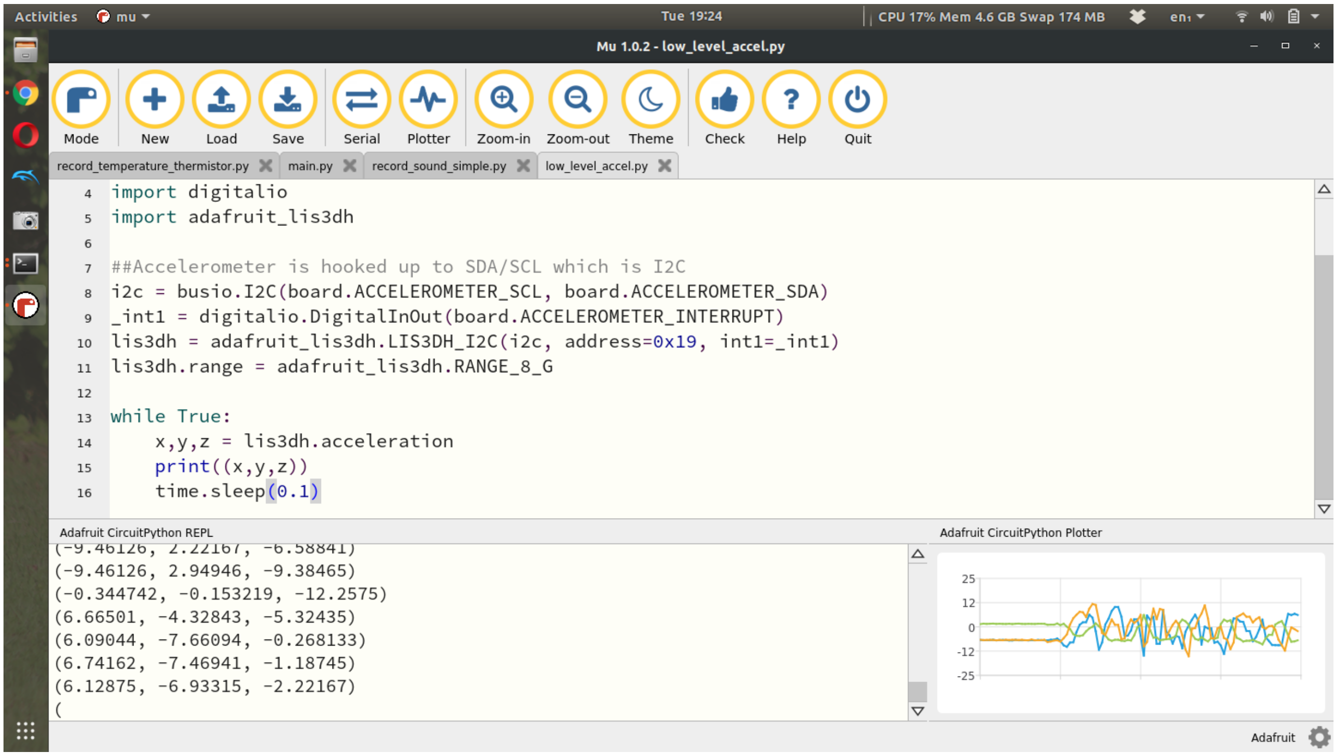


12.6.4 Accelerometer

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 2C like ”Squared 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.



12.7 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
```

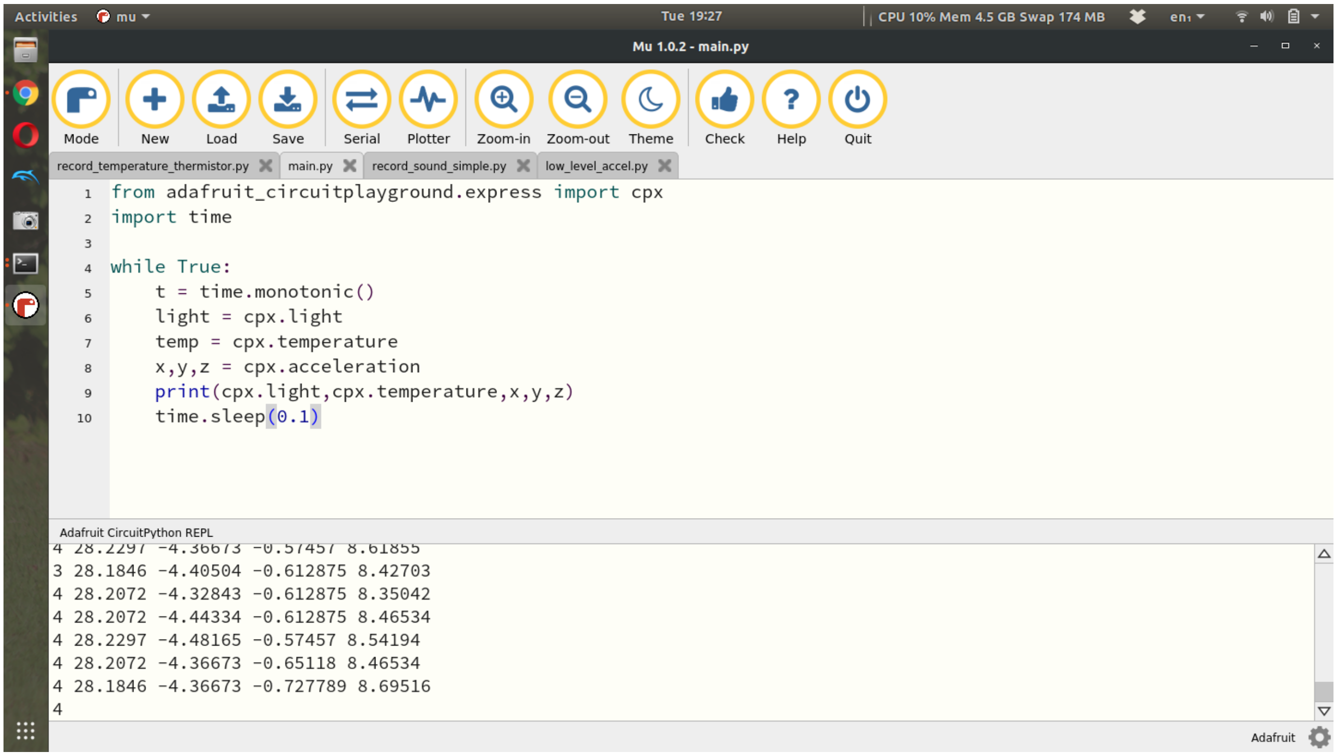
Note that you need to change that line to adafruit_circuitplayground.bluefruit import cpb. Then everywhere you see cpx you replace with cpb. This will import the cpx module into your working code. From here the commands to read different things are relatively simple. Here are the commands for all the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



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.

12.8 Assignment

Using either **low or high level control**, take at least 60 seconds of data using the microphone, photocell, accelerometer and thermistor on your CPX/CPB. Make sure to 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 (although this isn't a strict requirement).

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%
5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. Include 4 photos of Mu with the plotter open showing example data from each sensor. - 10% per photo
2. Include 4 plots with time on the x-axis and sensor data on the y-axis (No screenshots) - 10% per plot

13 Integrating Acceleration

13.1 Parts List

1. CPX/CPB
2. USB Cable
3. Laptop
4. Some sort of temporary adhesive
5. Automobile

13.2 Learning Objectives

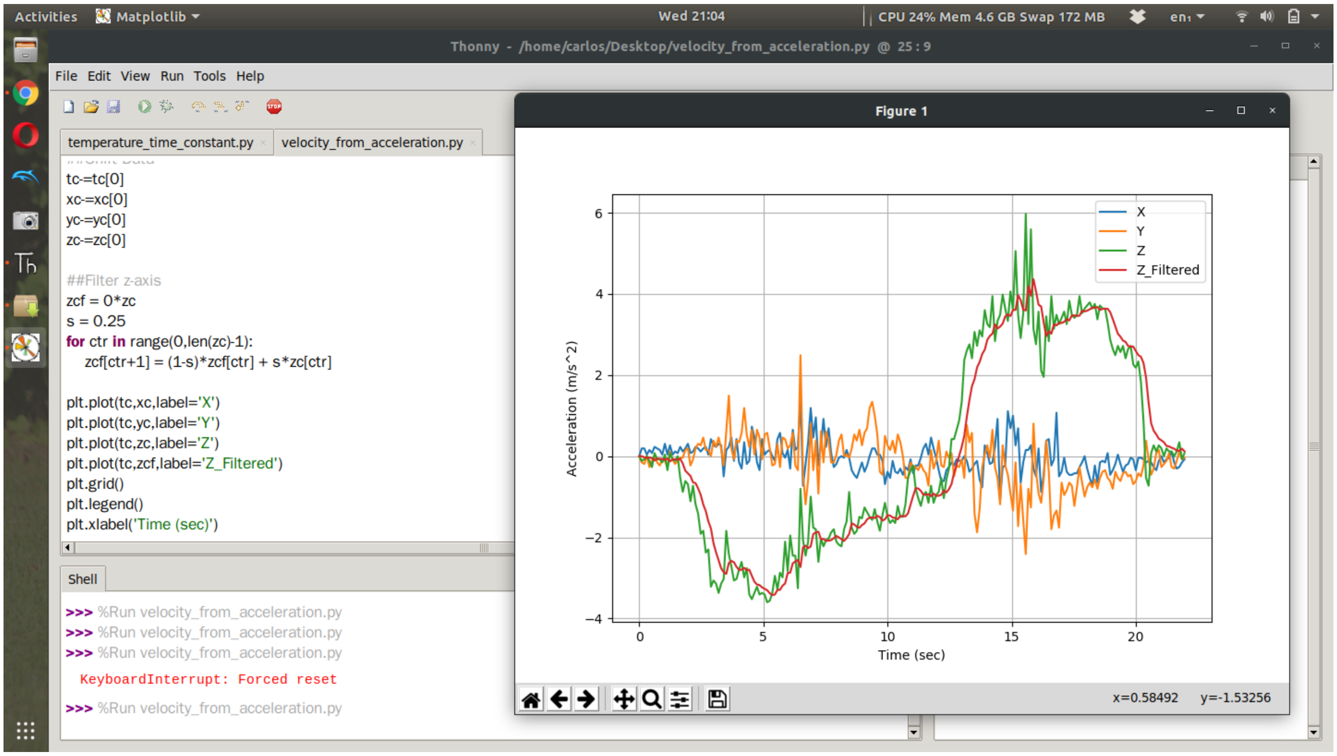
1. Taking acceleration data of real systems
2. Numerical Integration
3. The dangers of integration bias

13.3 Getting Started

The code for this lab is to have the CPX log acceleration. So when you're done with this lab you will hopefully have a data file with 4 columjns of data: time, acceleration x, acceleration y, acceleration z. The code I'm using is the same as the lab on accelerometers (See chatper 12.6.4). I'm using method 1 for datalogging so I'm just having it print to Serial (See chapter 8).

```
import time
import board
import busio
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)
```

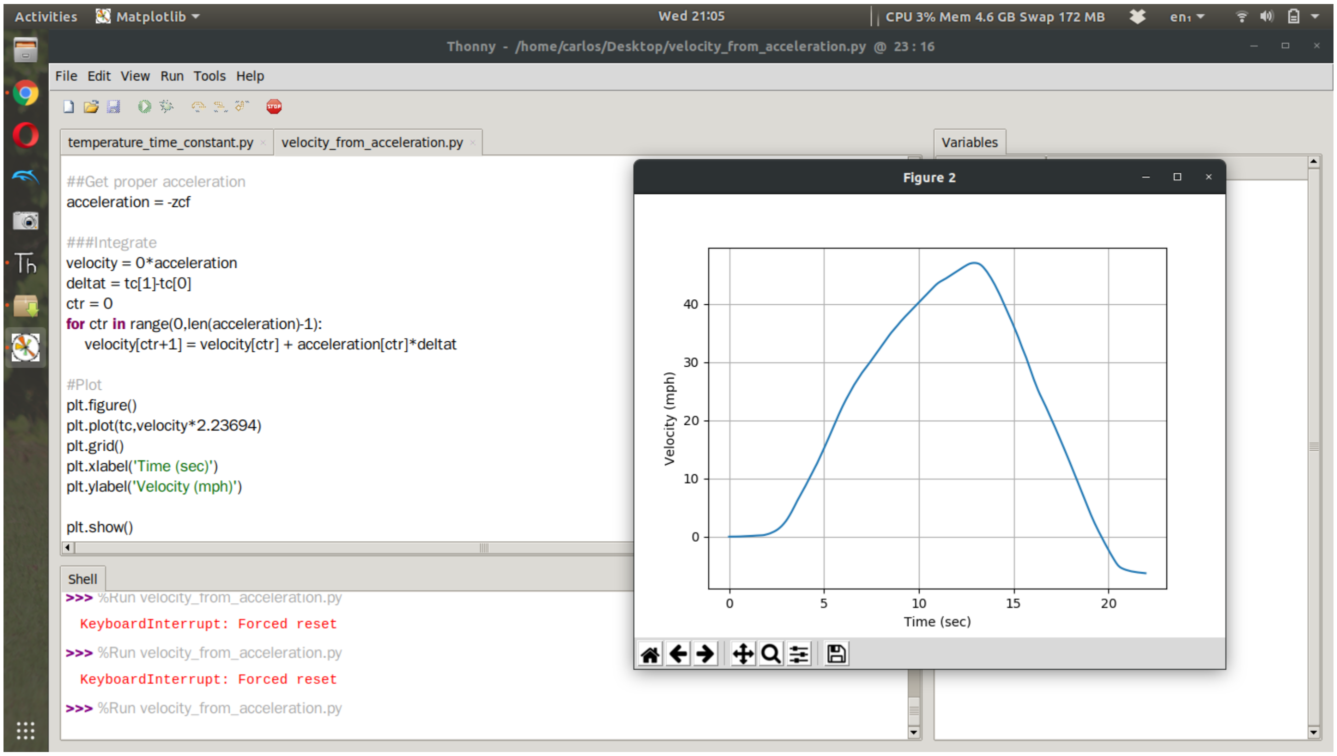
The code on Github has a sleep of 0.1 seconds but make sure to have the CPX take data as fast as possible. A sleep of 0.01 is probably good. You will probably get a lot of data points for this experiment. Once your code is working, place the CPX on your dashboard with one of the axes of the accelerometer pointing towards the nose of your car. Try and place the CPX on as flat a surface as possible. You can use 3M tape or duct tape or hot glue. Just make sure you don't damage your car and make sure the CPX is well anchored to the dashboard. This way when the car accelerates, the CPX will measure that acceleration. Note, if you'd like to do this with a bike or some other motor vehicle that is just fine. Just make sure to take pictures and videos when you do the experiment. I suggest you do this in a parking lot for safety reasons. I am not responsible for any damage done to your vehicle or anyone else because you are doing this project. Once you have the CPX anchored, accelerate your vehicle to 20 mph (or however fast you are comfortable driving) and then slam on the brakes. Once your data is logged, plot your acceleration in Python on your desktop computer. After doing the experiment myself, this is what my acceleration plot looks like. I had to clip the time series to only include the part from where I accelerated and decelerated quickly. I also subtracted the first data point from each accelerometer axis to zero it out and subtract off the bias. Since I took some data for a bit before I started moving I could have averaged the first few data points to obtain the bias. I've done this in a Youtube video if you're unsure what I mean. Instead just to get something working properly I went ahead and just used the first data point.



It's clear from the plots that the z axis was oriented towards the nose of the car. In this case I am going to have to flip the z-axis since the beginning is acceleration and the end is deceleration. I also through the acceleration in the z-axis through a complementary filter with a filter value of 0.25. I think it makes the acceleration profile a bit less jumpy. I then used a Reimann sum and integrated the acceleration data points to get velocity. The equation itself looks like this:

$$V_i = \sum_{n=0}^{i} (a_i - a_0)\Delta t \tag{6}$$

This of course assumes the initial velocity is zero. Notice that I take the individual acceleration points and subtract off the bias. Computing that summation by hand is pretty trivial but getting the code to work is another story. For a Reimann sum we're going to use a for loop where we loop through all the data points. The good news is that the time between data points is the same so we can just treat that as a constant. Once you have acceleration integrated you can plot velocity. This is what mine looks like after I did the experiment. According to my plot I accelerated to about 45 mph. I guess I can't lie in this instance. I said to accelerate to 20 mph but I really wanted to see a large change in acceleration so I punched it. Notice though that at the end of the time series the velocity is negative. This is because as time goes on you are integrating error and the error just gets worse.



This is why speedometers are used. They are just much more accurate than integrating acceleration which is prone to bias and drift. This folder on Github has some codes that will help with your project. **Note, that some of those codes have a bias filter, truncation filter and complementary filter. That code may not work for you and you may need to tune the filters for your specific data set. Make sure to understand what each filter does and think about how it applies to your data set otherwise your code may throw an error due to the differences in your data set.**

13.4 Assignment

For this assignment you are to find a safe place to accelerate and decelerate your vehicle while recording acceleration data on the Circuit Playground. I also want you to take GPS data using PhyPhox so you can compare data. I suggest using a temporary adhesive to secure your CPX/CPB to your dashboard (making sure it's level) and have a passenger in the car to help you record data as well as operate Phyphox and verify that you have a GPS lock. Again for safety's sake I suggest finding an empty parking lot and only accelerating to 20 mph or less.

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%
5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. Include a picture of your car and your passenger that is helping you record data. In your description write what speed you achieved in your experiment. - 10%
2. Include a photo of your CPX/CPB mounted to your vehicle indicating which axis of acceleration is pointing forward - 10%
3. Plot one axis of your accelerometer data vs time which clearly indicates when you accelerated and decelerated. In your description be sure to explain which axis you are plotting and any signal conditioners you applied to get your clean signal - 20%
4. Integrate acceleration and plot velocity as a function of time. Comment on whether or not the maximum velocity is the same as what you did in your actual car. Make sure to superimpose your Phyphox GPS speed on top of your plot to compare. You may need to have your raw GPS coordinates converted to velocity if it doesn't log speed natively. - 20%
5. Integrate the velocity and compute position. Plot your position as a function of time and include that in your report. Also include the position of your car using your GPS coordinates keeping in mind that you may need to convert your Lat/Lon coordinates to meters. Although you didn't measure how far you went, comment on the accuracy of the plot and whether or not you think you traveled that far especially given the superimposed data from your GPS. - 20%

14 Building a Pedometer using an Accelerometer

14.1 Parts List

1. Laptop
2. CPX/CPB
3. USB Cable
4. External Battery Pack

14.2 Learning Objectives

1. Understand how to run CPB/CPX while not tethered to a computer
2. Reinforce bluetooth tech for data transfer
3. Understand post-processing for debugging to be used for online calculations
4. Understand the fundamentals of how a pedometer works

14.3 Getting Started

A pedometer is a device that counts the number of steps. Typically these are worn as watches with popular brands like Garmin or Fitit owning the market share at the time of this writing. It turns out though that your phone and apps like Google Fit can also track steps just by being inside your pocket all day. The way they do this is by measuring the acceleration, angular velocity and potentially even the angles (using a magnetometer) to count steps. In this lab this we will just focus on attempting to get steps using accelerometer data.

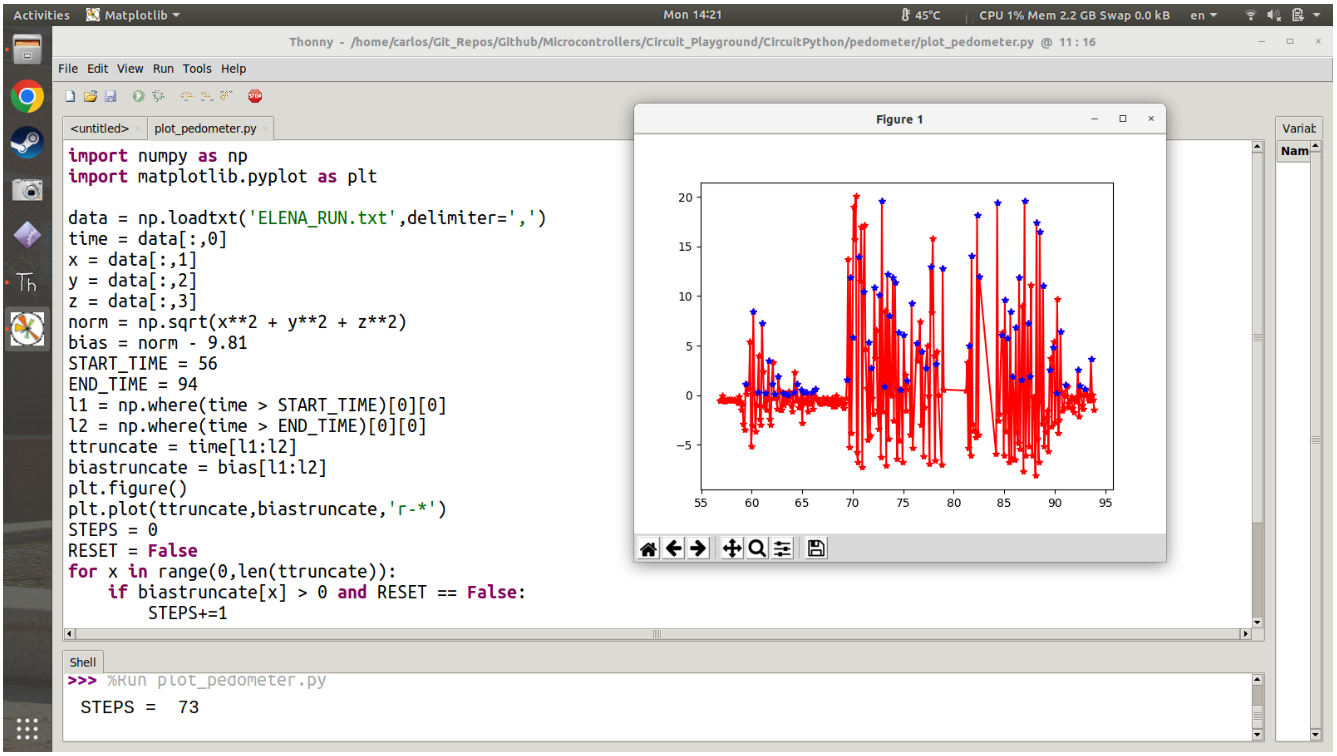
14.4 Gathering Accelerometer Data

First we need to make sure we can gather accelerometer data. The low level accelerometer code is relatively simple and is explained in the Modules lab (Section 12). In order to gather the accelerometer data while running you'll need to be able to operate the CPB/CPX untethered from a computer. This means you have to use Method 3 or 4 (Section 8 and 9). Remember that Method 1 and 2 require a computer and running with a computer would be difficult. Method 3 requires a lot of setup to log data directly to the disk so for this lab we will just use the Bluetooth module to send data directly to your phone. Note that if you have a CPX you will have to use Method 3. The best way to do this experiment is with a partner. Have the CPB/CPX measure acceleration and place the entire device with a battery pack inside the runners pocket. Then have your partner connect to the CPB with the Adafruit Connect app and log data using the UART and Export to txt function. Remember not to run too far because the Bluetooth signal distance is only about 30 feet. See if you can combine the Bluetooth code and the acceleration code into one code to send time and the 3-axis accelerometer data. If you're still having trouble, code for this lab can be found on Github. Note if you have a CPX you will need to combine the accelerometer code with the Method 3 version of data logging.

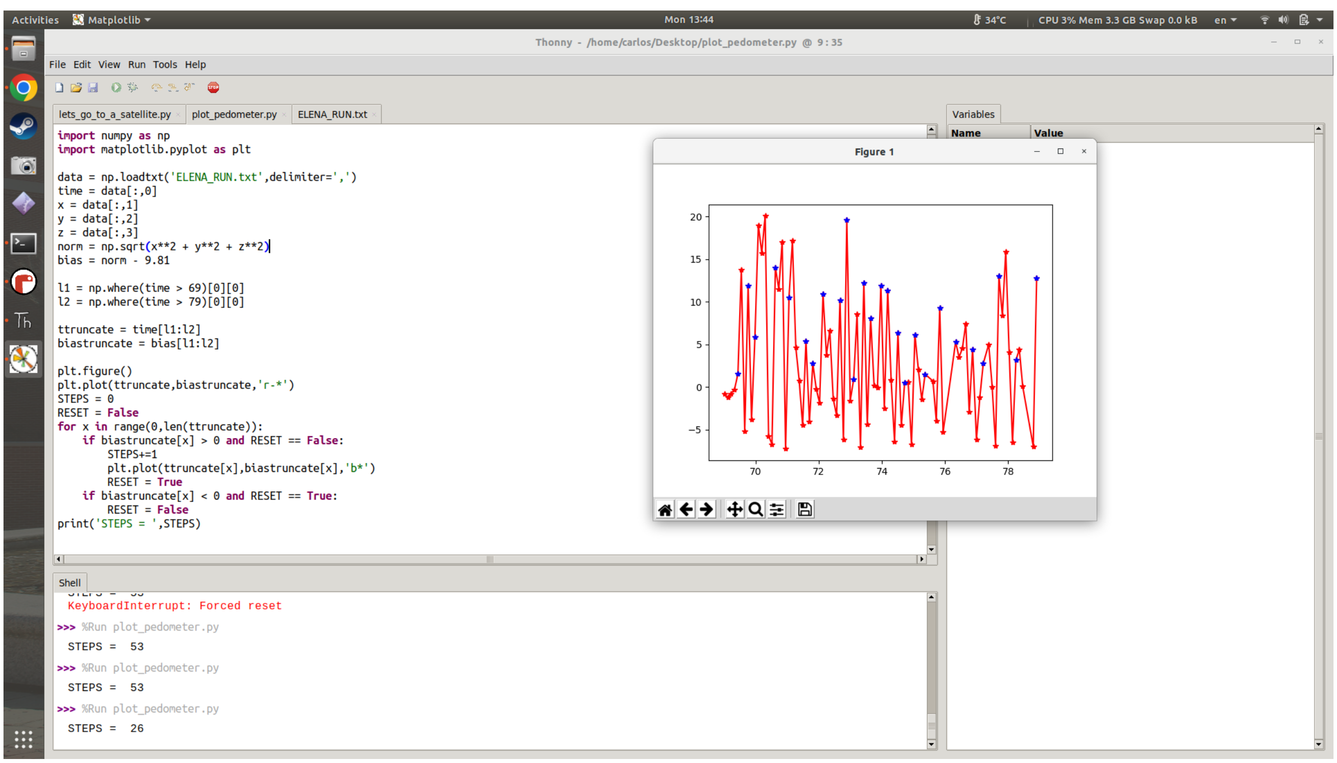
Running the CPX/CPB untethered does require a few extra steps besides writing the code. The first step is obviously to write the software that you want to run on the CPX/CPB. I recommend testing the code extensively while tethered to the computer so you can debug using the REPL. Once you're certain the code works you can disconnect the CPX/CPB and connect it to a battery pack. Once again I recommend testing the code with the battery back before you perform the experiment. In this experiment I used an external USB battery bank as shown in the photo below. My code also utilizes the neopixel library to turn on some LEDs.

14.5 Computing Number of Steps: Post-Processing

Using the hardware and software defined above I had my partner run down the hallway after I ensured there was a solid connection between the CPB and my smart phone. I then exported the data to a text file and plotted the raw data using Thonny.



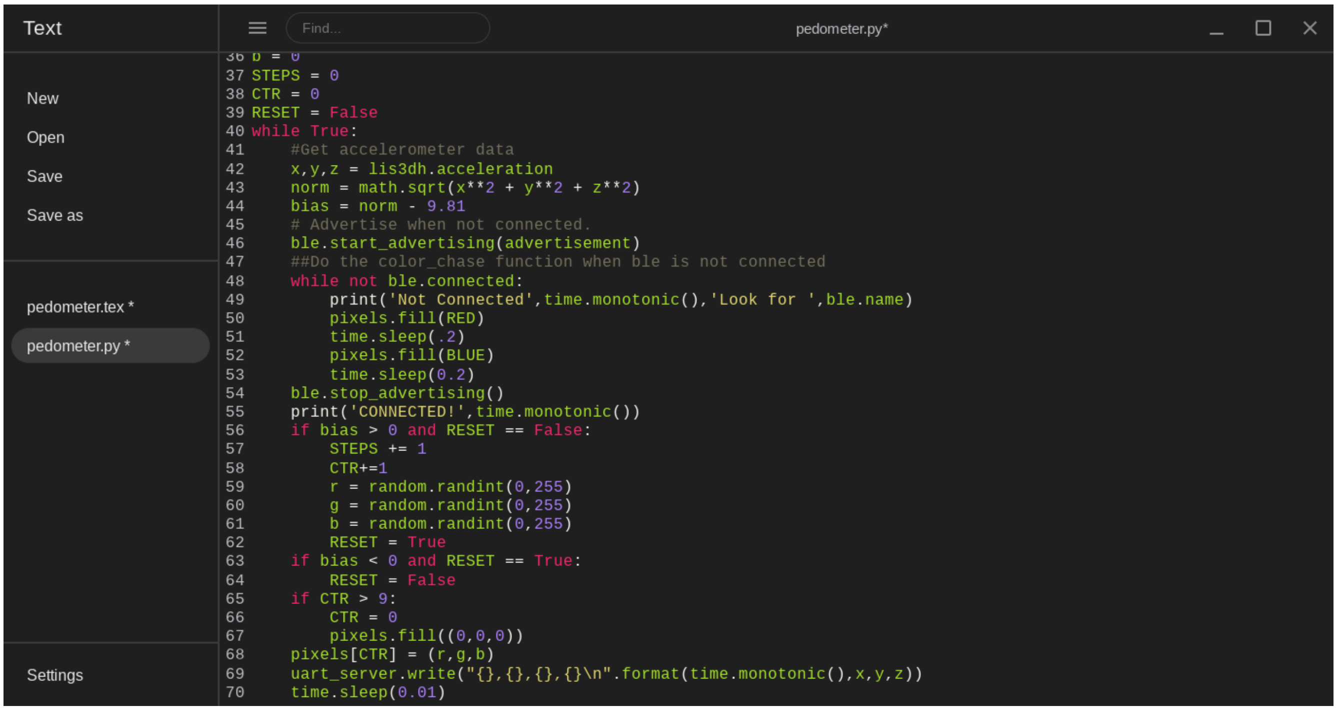
Upon inspecting the raw data it seems as though my partner began running around 68 seconds. At about 80 seconds my partner reached the end of the hallway and the CPB got a bit out of range. As such there is a gap in the data. The accelerometer streams return once my partner begins running back down the hallway. In order to simply look at the data of one run the data was truncated from 69 seconds to 79 seconds as shown in the Figure below.



In order to count steps the algorithm is fairly simple and not very robust but it does at least give you a sense of how data can be analyzed to obtain steps. First, the norm of the accelerometer data is computed and then the norm is substracted by 9.81 m/s^2. When glancing at the data the steps seem to be taken when the result of the norm-9.81 goes from positive to negative. It is possible that there is some aliasing in the data but for a simple experiment like this a rudimentary algorithm can be created. First the STEP counter is set to zero and then a for loop is created to loop through the data. When the data goes from positive to negative a STEP is created. This is done by using a RESET flag and checking whether or not the data becomes positive. This algorithm computes 26 steps which seems reasonable for the length of the hallway.

14.6 Computing Number of Steps: Online

The benefit of post-processing in Python is that the CPX/CPB run a almost identical derivative of Python called CircuitPython. This means that almost any line of code used in Python can be copied directly onto the CPX/CPB as shown in the Figure below.



You can see that the STEPS counter is set to zero before the infinite while loop and then after connection the RESET flag and bias value are checked for a switch from positive to negative. Some code is added to change the pixels on the CPB so that the user can see a step change. The raw data is still transmitted via bluetooth but it would be a neat exercise to have the CPB transmit the number of steps to a cell phone for instant feedback of the number of steps.

14.7 Assignment

For this assignment you are to run down the hallway while wearing your CPX/CPB and record acceleration. The acceleration signal will then be processed to compute the number of steps you took. While running, be sure to make a note on the number of steps you are taking.

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail

2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%
5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. Include a photo of your CPX/CPB with its own battery pack - 40%
2. Include a plot of accelerometer data vs time showing stars where the peaks are. In your description be sure to comment on how many steps the code outputted vs how many steps you took during your experiment - 40%

15 Inertial Measurement Unit - Accelerometer, Rate Gyro, Magnetometer

15.1 Parts List

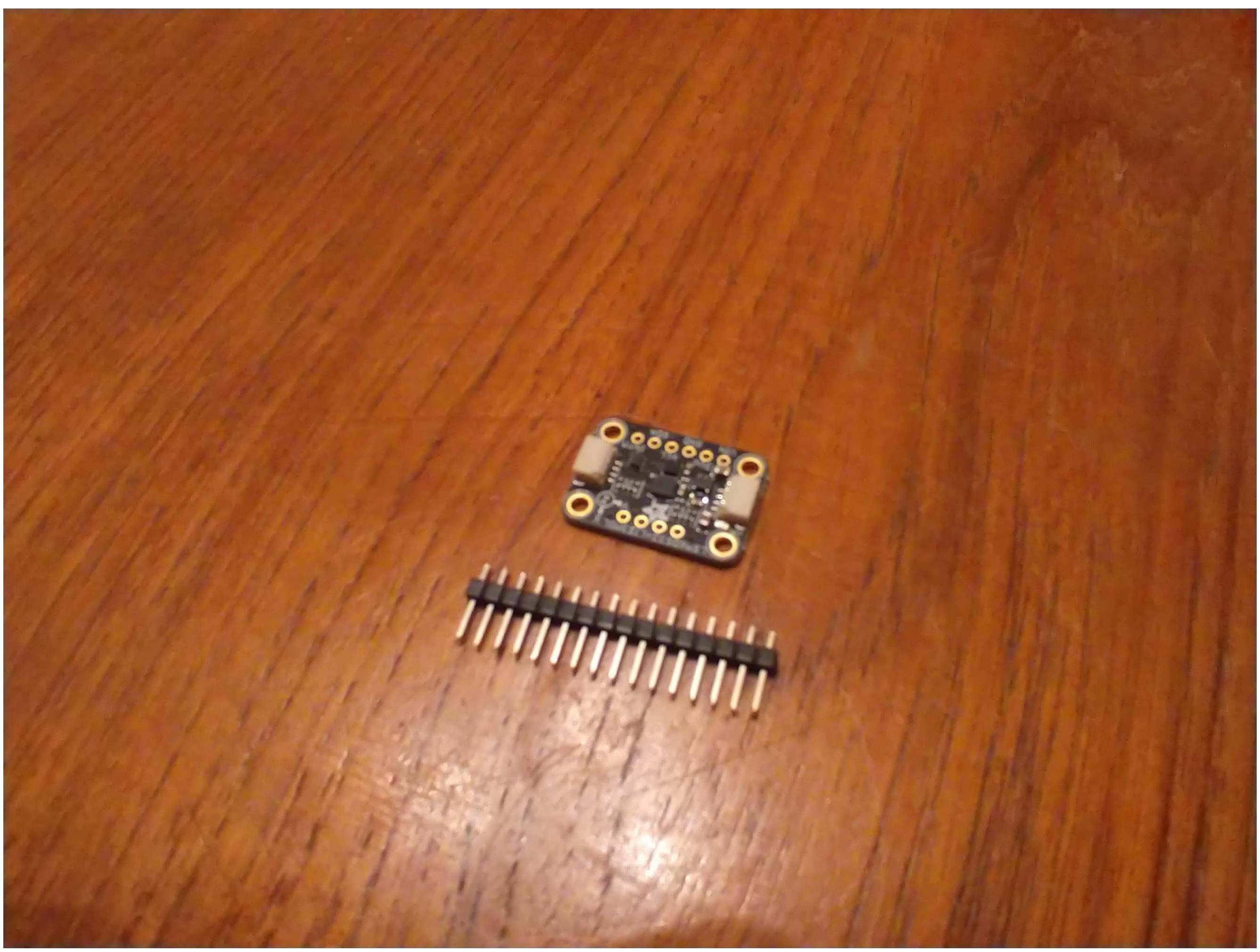
1. Laptop
2. CPX/CPB
3. USB Cable
4. LSM6DS33+LIS3MDL (Not included in kit. Note that other Inertial Measurement or "DOF" Sensors will work for this lab you will just have to change the code to accommodate the change in hardware)
5. Alligator Clips (x4)
6. Bread Board
7. Soldering iron

15.2 Learning Objectives

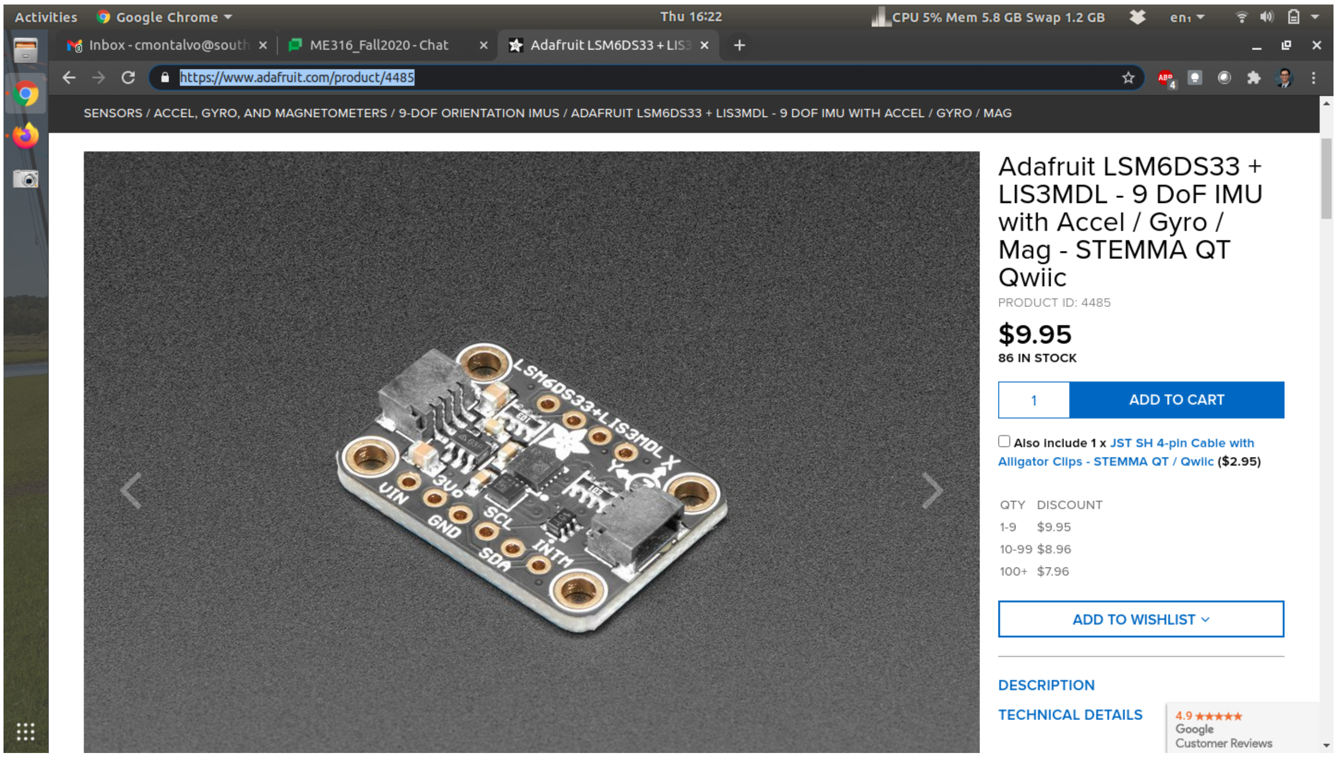
1. See and understand the concept of soldering
2. Understand the I2C protocol at a high level
3. Learn the components of an IMU (Accelerometer, Rate Gyro, Magnetometer)
4. Read IMU data and plot

15.3 Getting Started

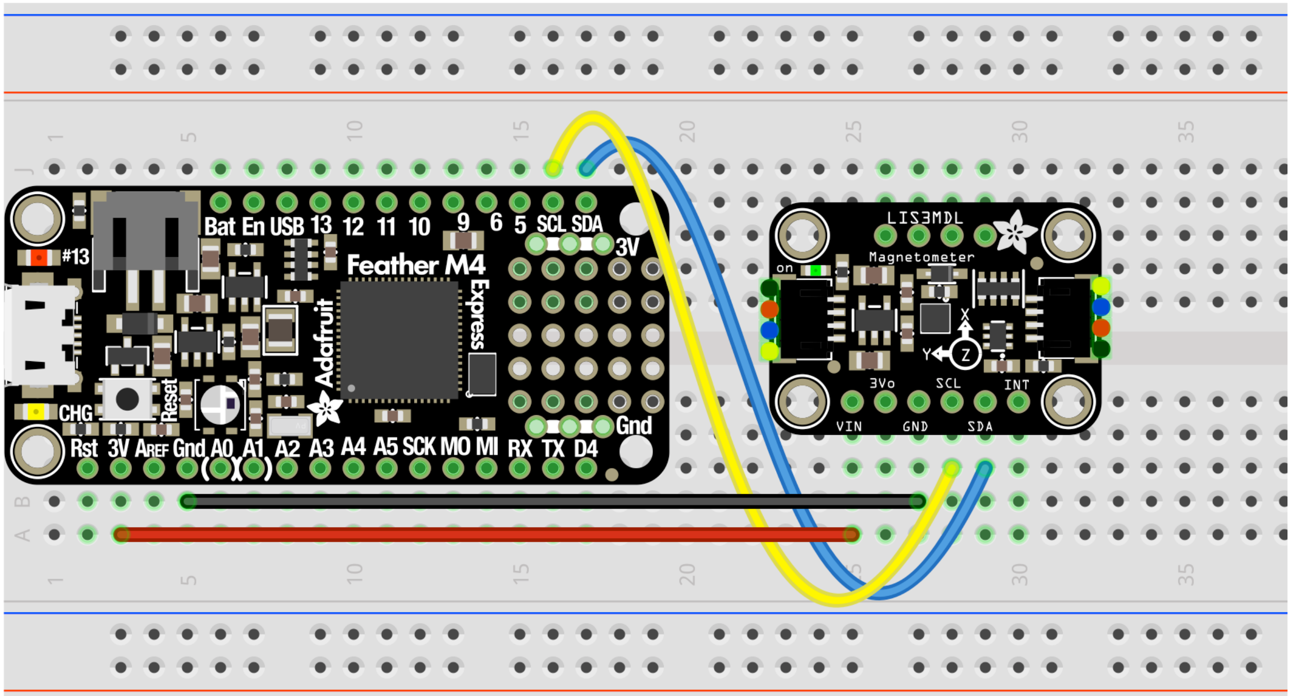
In this lab we're going to use this an external sensor to measure angular velocity and the magnetic field of the surrounding environment.



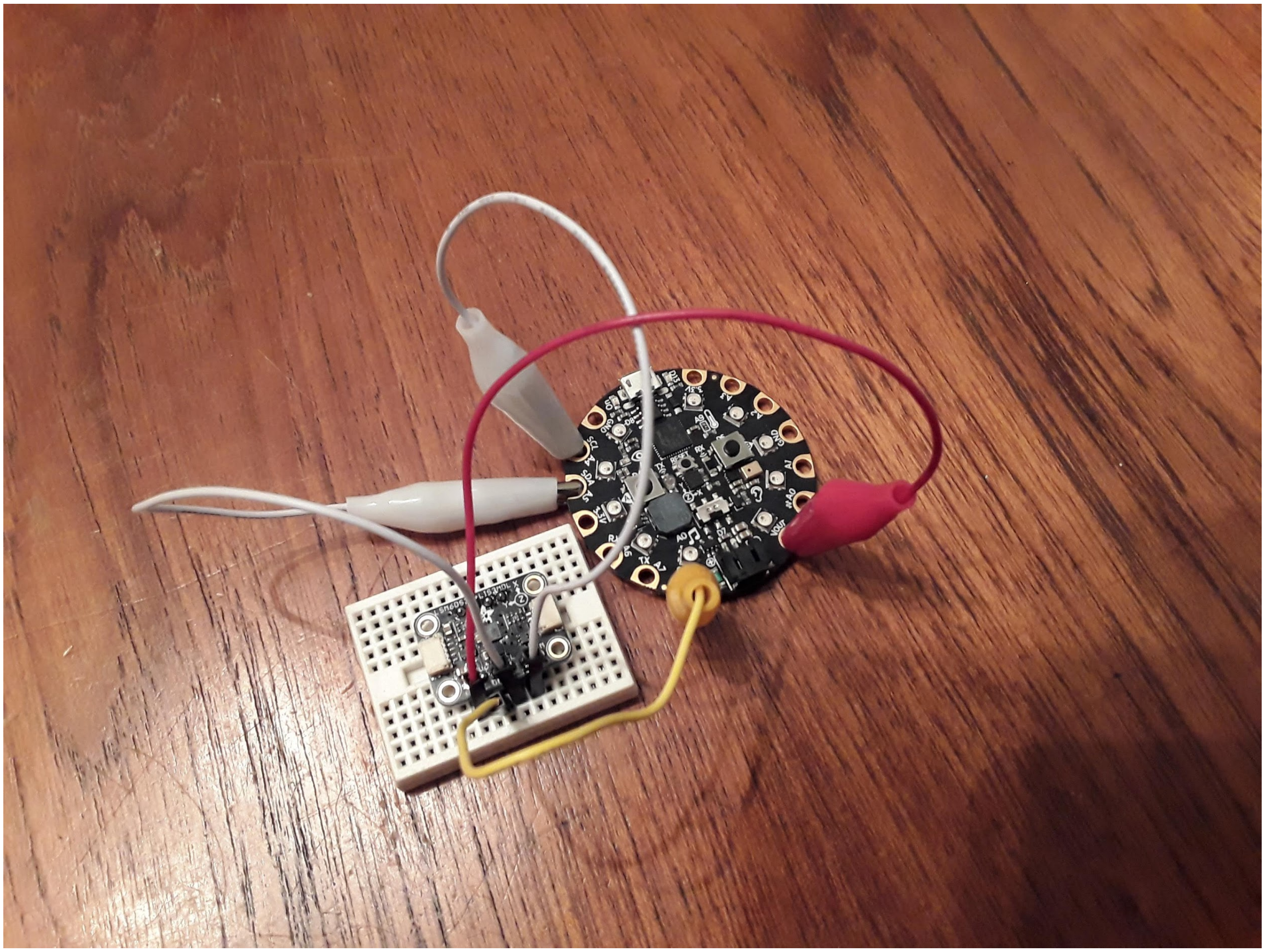
The sensor above is the LSM6DS33+LIS3MDL. You basically have 2 separate microchips in one. The first (LSM6DS33) is a 3-axis accelerometer and 3-axis rate gyro. The first measures acceleration and the second measures angular velocity. The LIS3MDL is a 3-axis magnetometer which measures magnetic fields. The three of these sensors put together (accelerometer, rate gyro, magnetometer) is called an IMU (Inertial Measurement Unit). It's actually possible to measure roll and pitch of an airplane and heading using the magnetometer. Combining the angular velocity of the rate gyro can create a complete attitude estimation algorithm for spacecraft. This sensor does not come standard in the current iteration of the kit. You can purchase one on Adafruit for only \$10 at the time of this writing. The interesting thing about this device is that you can actually purchase the LSM6DS33 and LIS3MDL separately but this breakout board has both chips on board. The goal of this lab is not necessarily to use this specific sensor but to understand IMUs and I2C protocol. Most if not all breakout boards on the Adafruit website use I2C communication. You'll know if the breakout board uses I2C if you find SDA/SCL pins on the board. It will also say it in the quick description of the sensor.



When you open the packaging of this breakout board you'll notice that the header pins are missing. First you'll need to cut a row of 4 and 6 for the top and bottom side of the board and **solder the header pins to the sensor**. If you're taking my class I can solder this for you or teach everyone about soldering during a lecture session of class. If you are taking this class elsewhere you have two options: try and find someone who can solder this real quick (only takes about 5 minutes) or buy your own soldering iron and try to solder yourself. Once the device is soldered you can "plug" it into a breadboard. The wiring for this system requires 4 wires. The figure below shows the IMU connected to Feather M4 (Courtesy of Bryan Siepert).



The only difference between the wiring diagram above and the CPX/CPB is that you will be using 4 alligator clips. The rest is straightforward. You need VOUT (5V) to run to (VIN), GND to GND and then SDA to SDA and SCL to SCL.



Believe it or not the photo above was actually wired wrong. I had SDA to SCL and SCL to SDA. You need to make sure you have the proper wires going to the correct pins or it won't work. SDA and SCL are 2 pins for something called I2C (pronounced I squared C) where I is I as in "I am a billy goat" or "I like to code". I2C is beyond the scope of this project but just know it's a type of serial communication that uses hexadecimal addresses.

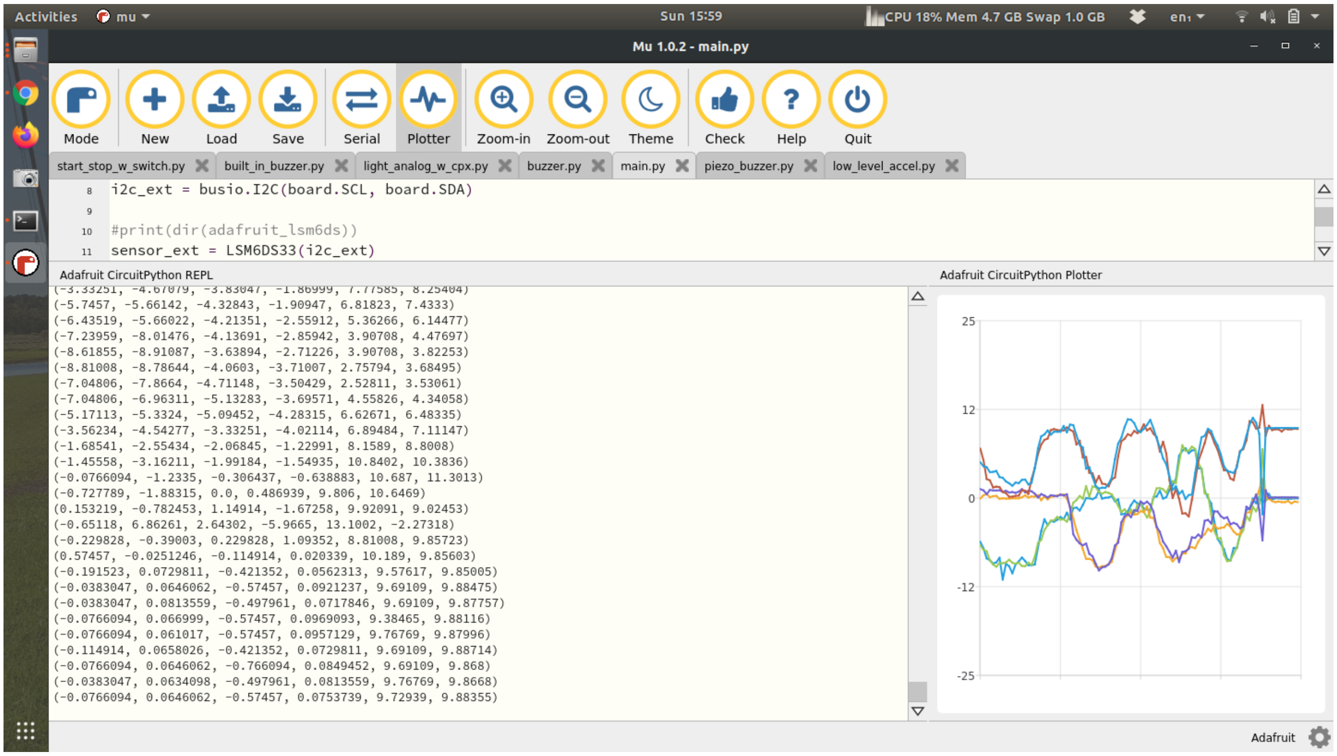
Once you have the circuit wired and soldered it's time to work on software. First, you want to make sure you have your Circuit Python UF2 up to date. In this example I'm using the 6.X version. Once I updated my UF2 I also updated my Circuit Python Libraries. The Circuit Python Libraries download as a zip file. You need to unzip the folder and go into the lib folder and grab the following python modules.

1. adafruit_bus_device
2. adafruit_register
3. lis3mdl
4. lsm6ds33

There will be folders for some and just floating .mpy files for others which are python modules that you can import just like time, board and busio as we've done in the past. Put those files into your lib folder on your CIRCUITPY drive. If the lib folder doesn't exist you just need to make one. Once you have the necessary modules you can run some example code. The Adafruit Learn page has a tutorial for the LSM6DS33. The problem with the tutorial is that it seems like it was written for the Raspberry or some other microcontroller. As such I had to find some example code on Adafruit's Github. After following both tutorials I was able to make my own script and upload it to my Github.

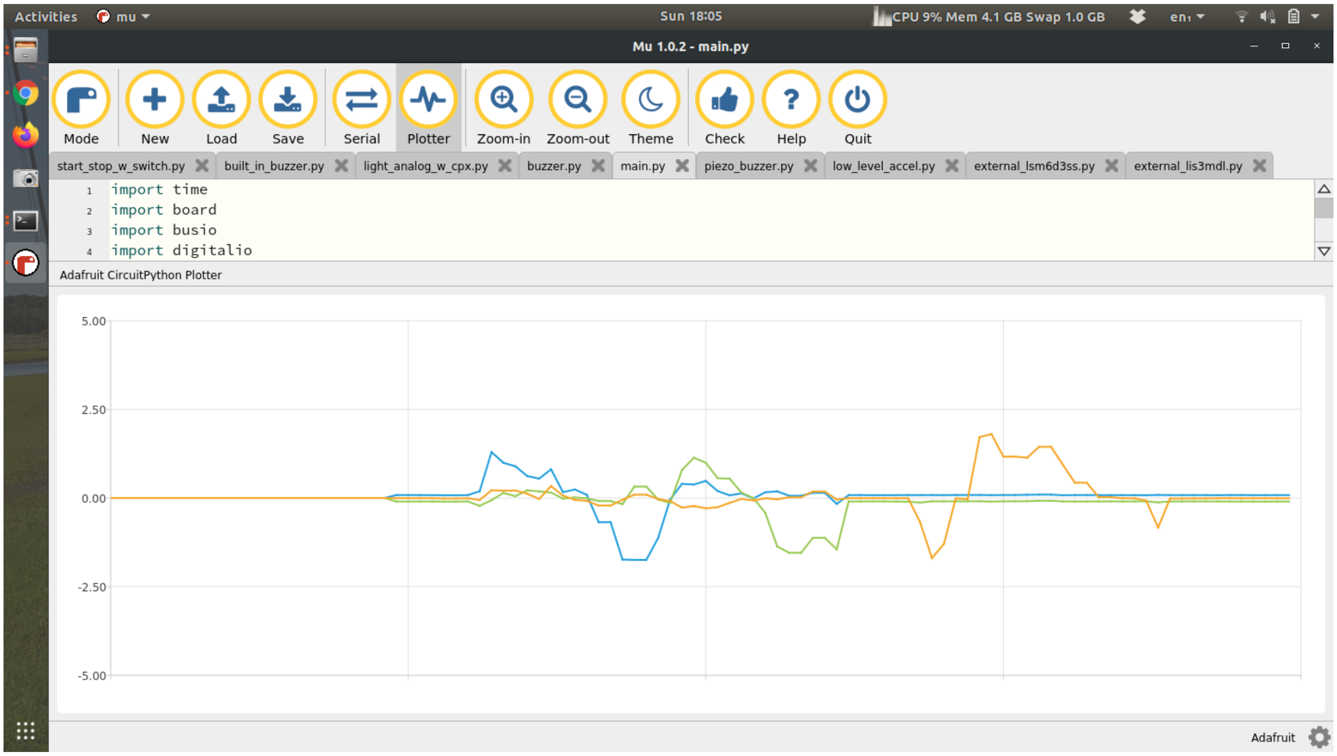
```
1  import time
2  import board
3  import busio
4  import digitalio
5  import adafruit_lis3dh
6  from adafruit_lsm6ds.lsm6ds33 import LSM6DS33
7
8  i2c_ext = busio.I2C(board.SCL, board.SDA)
9
10 #print(dir(adafruit_lsm6ds))
11 sensor_ext = LSM6DS33(i2c_ext)
12
13 ##Accelerometer is hooked up to SDA/SCL which is I2C
14 i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
15 _int1 = digitalio.DigitalInOut(board.ACCELEROMETER_INTERRUPT)
16 lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c, address=0x19, int1=_int1)
17 lis3dh.range = adafruit_lis3dh.RANGE_8_G
18
19 while True:
20     #x,y,z = lis3dh.acceleration
21     #xe,ye,ze = sensor_ext.acceleration
22     gx,gy,gz = sensor_ext.gyro
23     #print((x,xe,y,ye,z,ze))
24     print((gx,gy,gz))
25     time.sleep(0.1)
```

In the code above lines 1-6 import all the modules with line 5 importing the accelerometer on board the CPX and line 6 importing the external sensor wired up to SDA and SCL. Line 8 creates an I2C object using the SDA and SCL pins from the alligator clips and line 11 creates the sensor object. I also include lines 14-17 to include the onboard accelerometer. Notice I can access both sensors no problem. In the while loop line 20 checks the accelerometer on the CPX, line 21 checks the accelerometer on the breakout board and line 22 checks the angular velocity on the breakout board. Lines 23 and 24 print to serial and output to the plotter. Note that some lines are commented out because I wanted to try one thing at a time. With both accelerometers printing to the Plotter I could move the CPX and the breakout board in unison and get the following output.



Notice that there are 6 numbers printed and 6 lines on the plotter. Both the CPX and the breakout board have little XYZ cartesian coordinate systems. I had to line them up properly before I started moving them. My suggestion would be for you to get some hot glue or 3M tape and place both breakout board and CPX on some sort of hard material like plywood, masonite, or even a cutting board. Anything to keep everything together.

Once you've done this, try uncommenting the line of code that prints the angular velocity. When I do that and move the breakout board around I can measure the angular velocity of each axis. The units are in radians per second but it's pretty obvious just from the magnitude of the graph.



The final part is to get the LIS3MDL (magnetometer) to work. The starting point for me was the Adafruit Learn page, along with the simple example from Adafruit's Github. After that I was able to create my own code. The only difference in your code is that the address will be 0x1c instead of 0x1e.

```python
1   import time
2   import board
3   import busio
4   import digitalio
5   #import adafruit_lis3dh
6   #from adafruit_lsm6ds.lsm6ds33 import LSM6DS33
7   import adafruit_lis3mdl
8
9   i2c_ext = busio.I2C(board.SCL, board.SDA)
10
11  #print(dir(adafruit_lsm6ds))
12  #sensor_ext = LSM6DS33(i2c_ext,address=0x6a)
13  mag = adafruit_lis3mdl.LIS3MDL(i2c_ext,address=0x1e)
14
15  ##Accelerometer is hooked up to SDA/SCL which is I2C
16  #i2c = busio.I2C(board.ACCELEROMETER_SCL, board.ACCELEROMETER_SDA)
17  #_int1 = digitalio.DigitalInOut(board.ACCELEROMETER_INTERRUPT)
18  #lis3dh = adafruit_lis3dh.LIS3DH_I2C(i2c, address=0x19, int1=_int1)
19  #lis3dh.range = adafruit_lis3dh.RANGE_8_G
20
21  while True:
22      #x,y,z = lis3dh.acceleration
23      #xe,ye,ze = sensor_ext.acceleration
24      #gx,gy,gz = sensor_ext.gyro
25      bx,by,bz = mag.magnetic
26      #print((x,y,z))
27      #print((x,xe,y,ye,z,ze))
28      #print((xe,ye,ze))
29      #print((gx,gy,gz))
30      print((bx,by,bz))
31      time.sleep(1.0)
```

The code is almost identical to the code before except all the LIS3DH and LSM6DS33 code is commented out. Instead I have code to grab the magnetometer (LIS3MDL) at address 0x1E. Line 25 then calls the magnetometer and prints it. Note that the accelerometer can be used to obtain pitch and roll angles and the magnetometer can be used to obtain the yaw angle. This is done via trigonometry and is shown in the equations below.

$$\theta = -sin^{-1}(\bar{a}_x) \tag{7}$$

$$\phi = tan^{-1}\left(\frac{\bar{a}_y}{\bar{a}_z}\right) \tag{8}$$

$$\psi = tan^{-1}\left(\frac{\bar{\beta}_z s_\phi - \bar{\beta}_y c_\phi}{\bar{\beta}_x c_\phi + \bar{\beta}_y s_\theta s_\phi + \bar{\beta}_z c_\phi s_\theta}\right) \tag{9}$$

In the equations above β is the magnetic field along all 3 axes and a is the accelerations along all 3 axes. Shorthand is used for $cos(\eta) = c_\eta$ and $sin(\eta) = s_\eta$. The $\bar{\eta}$ notation is used to indicate a normalization of the vector. That is $\bar{\eta} = \eta/||\eta||$ where $||\eta||$ is the norm of a 3 dimensional vector $||\eta|| = \sqrt{\eta_x^2 + \eta_y^2 + \eta_z^2}$. The derivation for these angles from accelerometers and magnetometers is quite involved and requires the knowledge of rotation matrices. That derivation is in another textbook called Aerospace Mechanics.

15.4 Assignment

For this assignment you are to wire up the external IMU and get data from it. You need to mount the CPX/CPB and the IMU to some sort of hard surface so that when you move the CPX/CPB the IMU moves as well. Make sure the axis of the IMU and the CPX/CPB are oriented in the same direction. With system mounted to a hard surface, perform doublet manuevers on each axis for a total of 3 doublets. A doublet is where you rotate the system to +90 degrees and then -90 degrees and then back to zero typically taking around 3 seconds for the entire maneuever. Using the accelerometer, compute the roll and pitch angle in degrees. Then use the magnetometer to compute the yaw angle. Using the roll, pitch and yaw angles, take a derivative to compute the angular velocity. Finally, take the angular velocity data and integrate it to obtain the pitch, roll and yaw angles.

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%
5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. Include a photo of your CPX/CPB mounted to a hard surface with the IMU on a breadboard. Be sure to explain in your description about your axis system for both sensors. - 10%
2. Include a screenshot of Mu showing the Plotter open and all 3 angular velocity axes - 10%
3. Include a plot of both accelerometers for the 3 doublet manuevers. - 10%
4. Include a plot of angular velocity data for the 3 doublet manuevers. Also plot the derivative of the pitch, roll and yaw angles on top of this plot and add a legend to clearly indicate which is which. - 20%
5. Include a plot of magnetometer data for the 3 doublet maneuevers - 10%
6. Plot the roll, pitch and yaw angles in degrees from the accelerometer/magnetometer as well as the integrated rate gyro angles. Add a legend to clearly indicate which line is which. - 20%

16 Histograms and Normal Distribution of Photocell Readings

16.1 Parts List

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

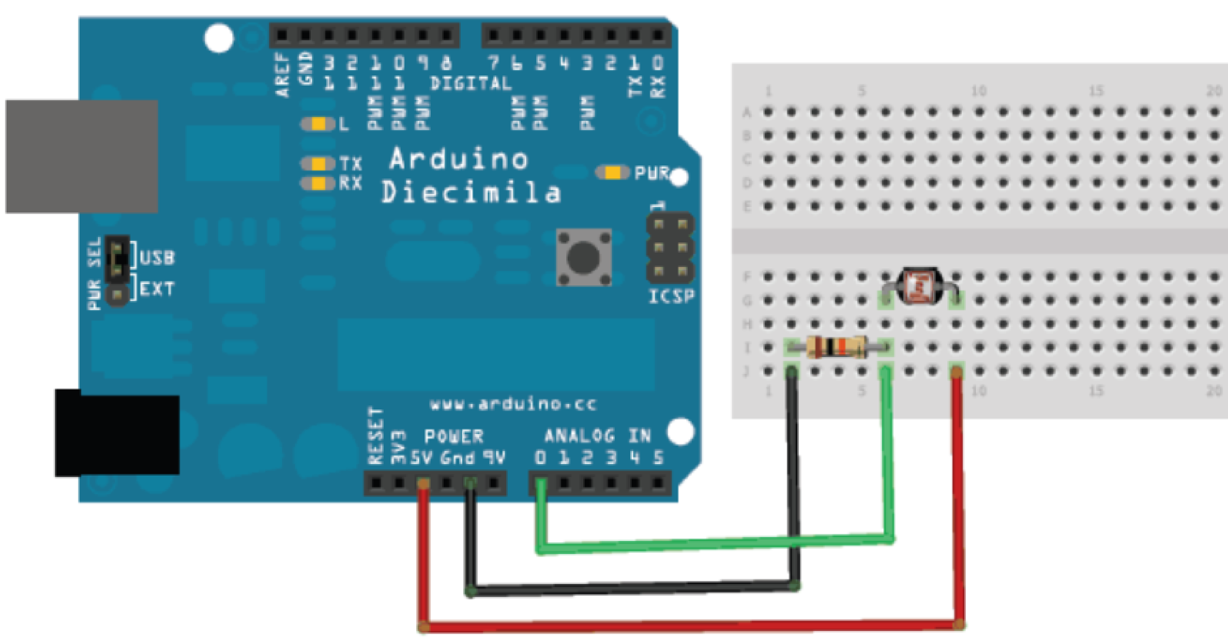
16.2 Learning Objectives

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.

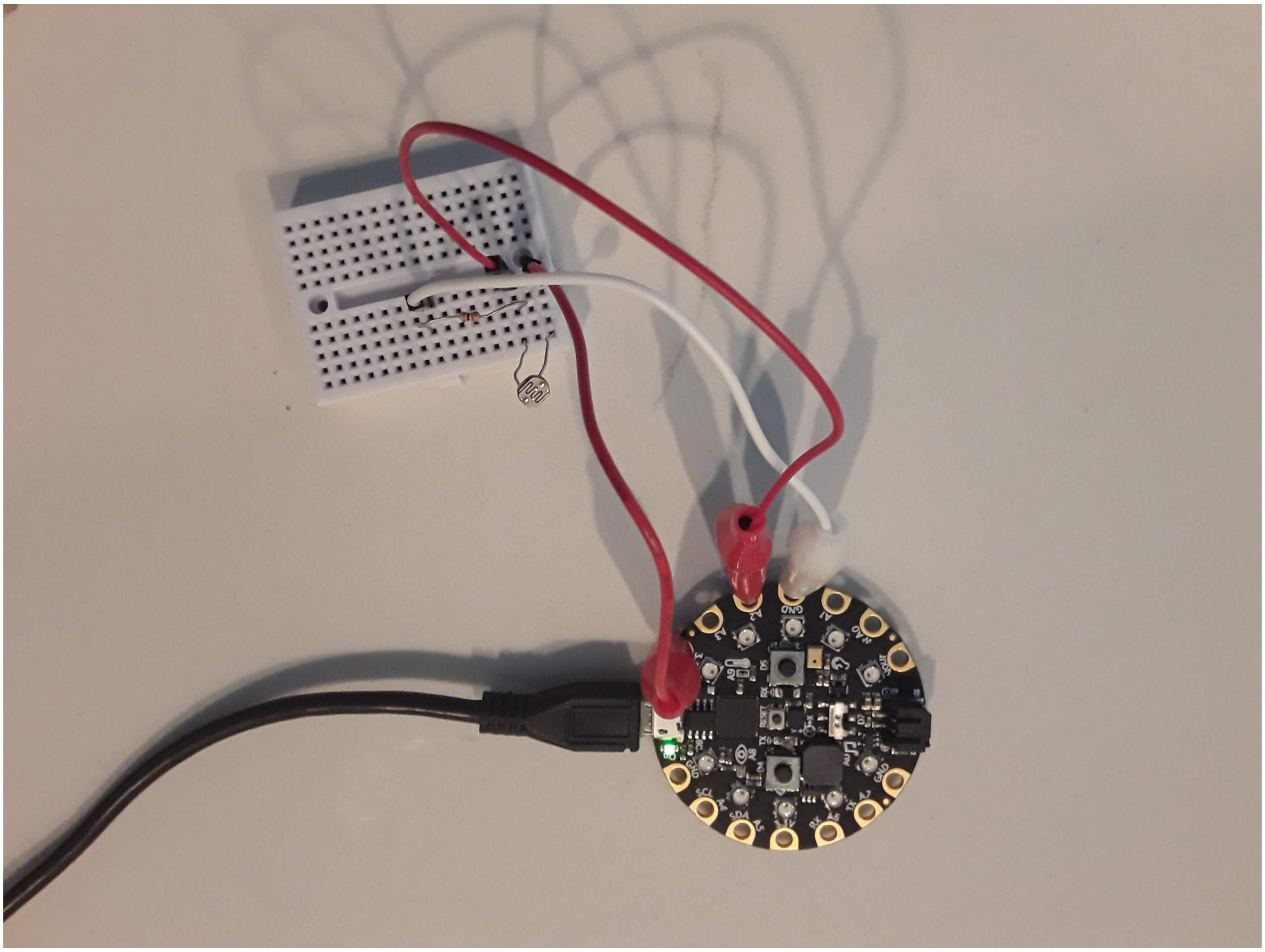
16.3 Getting Started

This lab is going to be similar to the potentiometer lab (See chapter 10). We are going to use a photocell though to vary the resistance instead of a potentiometer. A photocell (or Light Dependent Resistor, LDR) is a sensor whose resistance changes significantly based on the intensity of light falling on its sensitive surface. In dimmer light, its resistance is high, and in brighter light, its resistance is low. By integrating it into a simple voltage divider circuit similar to what you did for the potentiometer lab, you'll be able to read the voltage across the photocell and correlate that to light level.

Wiring a photocell is similar to a potentiometer and remember that you need a resistor in series with the photocell to create the voltage divider circuit. 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 photo below also shows a voltage divider circuit of a photocell using an Arduino (Courtesy of Lady Ada herself!).



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 was already presented and when I did the lab my example circuit is shown below. An alligator clip is connected to 3.3V on the CPX and the other end is connected to either end of the photocell. The photocell is then in series with a resistor and the other end of the resistor is connected to GND via another alligator clip. Finally, another alligator clip from pin A2 is plugged into the same row on the breadboard as the resistor and photocell.



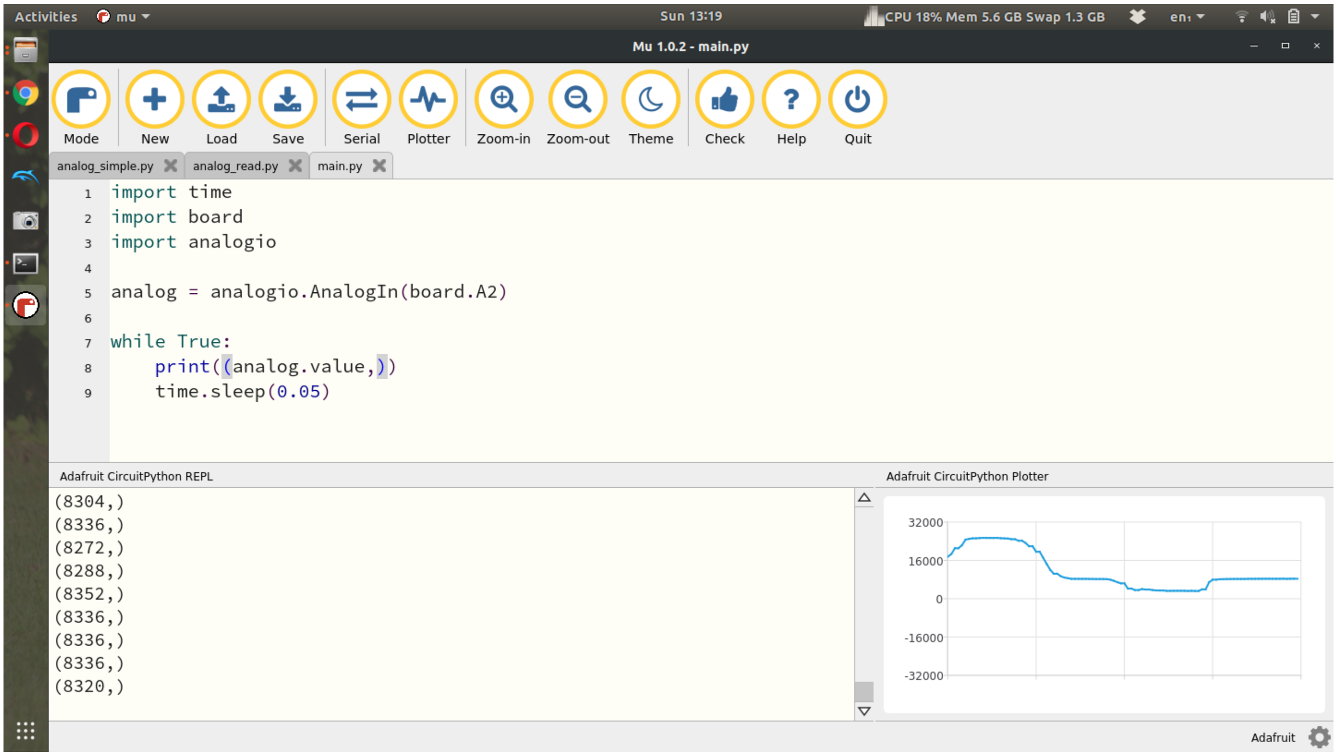
Building this circuit as a voltage divider will create the standard voltage divider law given by the equation below.

$$V_{photocell} = V_{out}\frac{R_{photocell}}{R_{series} + R_{photocell}} \tag{10}$$

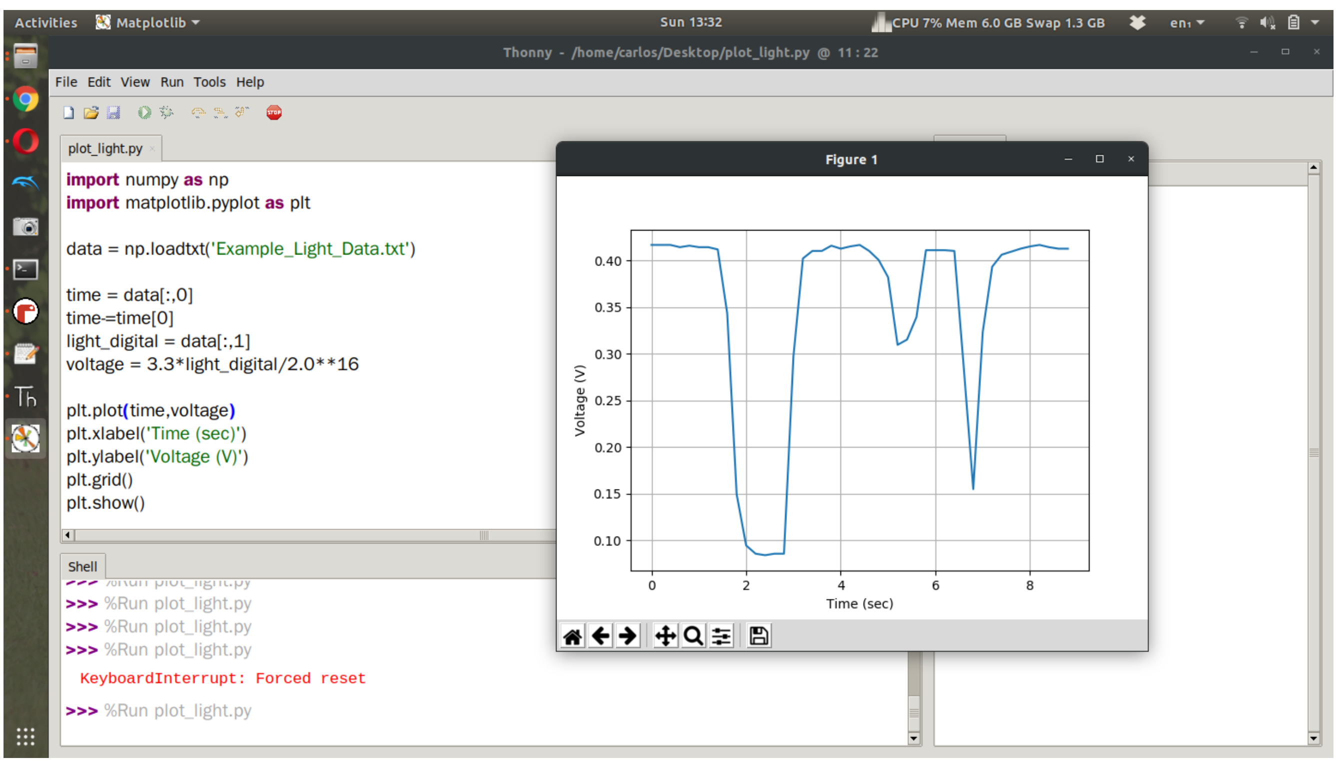
In this case V_{out} is 3.3V and R_{series} is dependant on the resistor you use in your circuit. So, if you measure $V_{photocell}$ using the ADC on the CPX/CPB you'll be able to solve for the resistance in the photocell $R_{photocell}$. The resistance can then be used to determine the lux value. Some sources suggest that Lux is a power law given by the equation below [27].

$$Lux = 500000R_{photocell}^{-0.7} \tag{11}$$

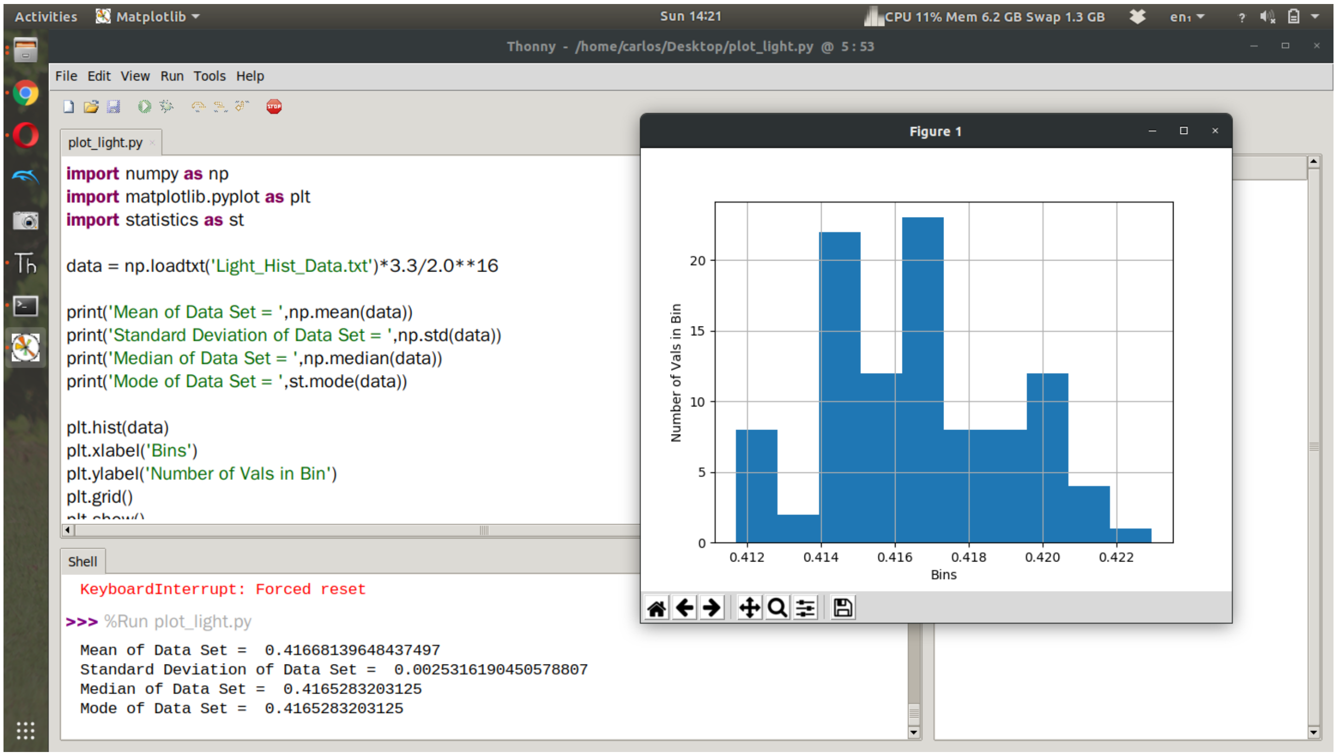
where $R_{photocell}$ is the resistance of the photocell in Ohms. Once you have the circuit wired properly you can use the same code as the potentiometer lab. The example screenshot below shows the analog signal below showing a high spike where I placed a flashlight over the photocell and then a low 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.



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.



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 sunlight if you're outside. If you're completing this assignment late you can use a flashlight as your third light source. 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.



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]
```

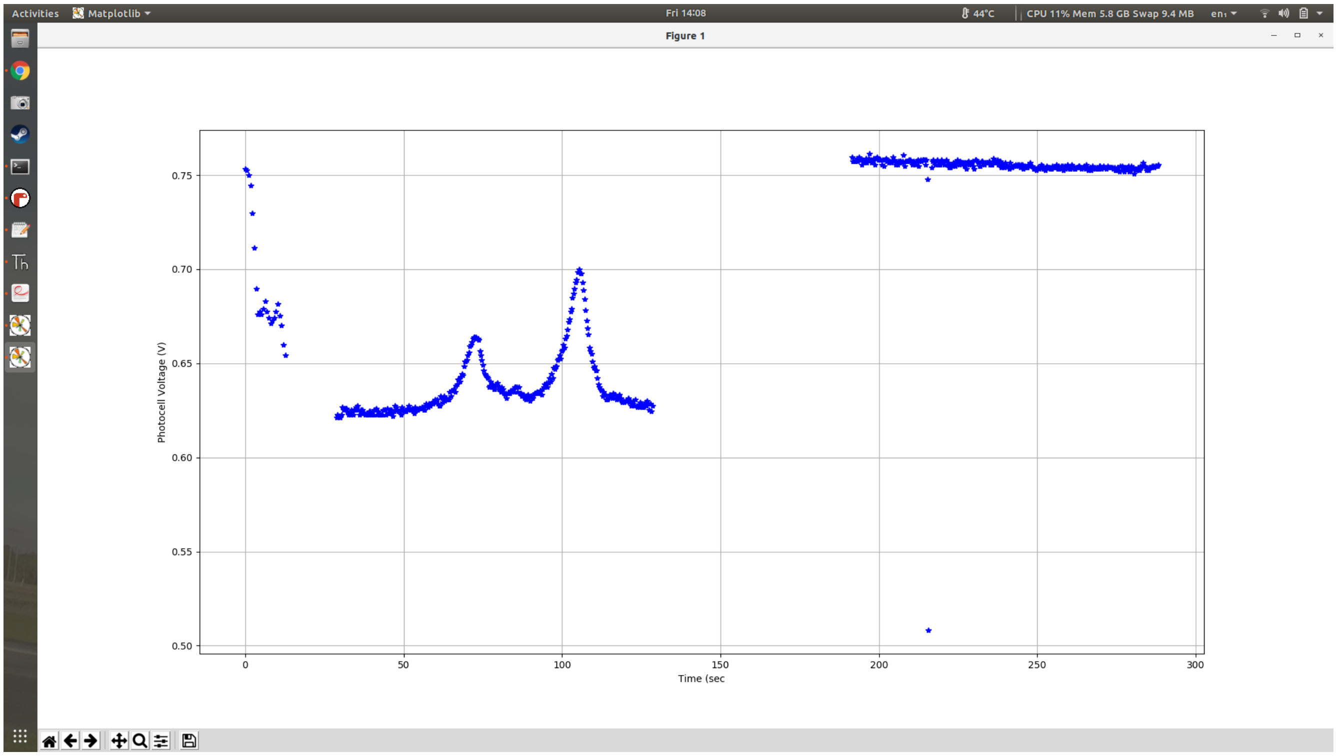
Finally convert to voltage

```
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.

16.4 Throwing Out Outliers

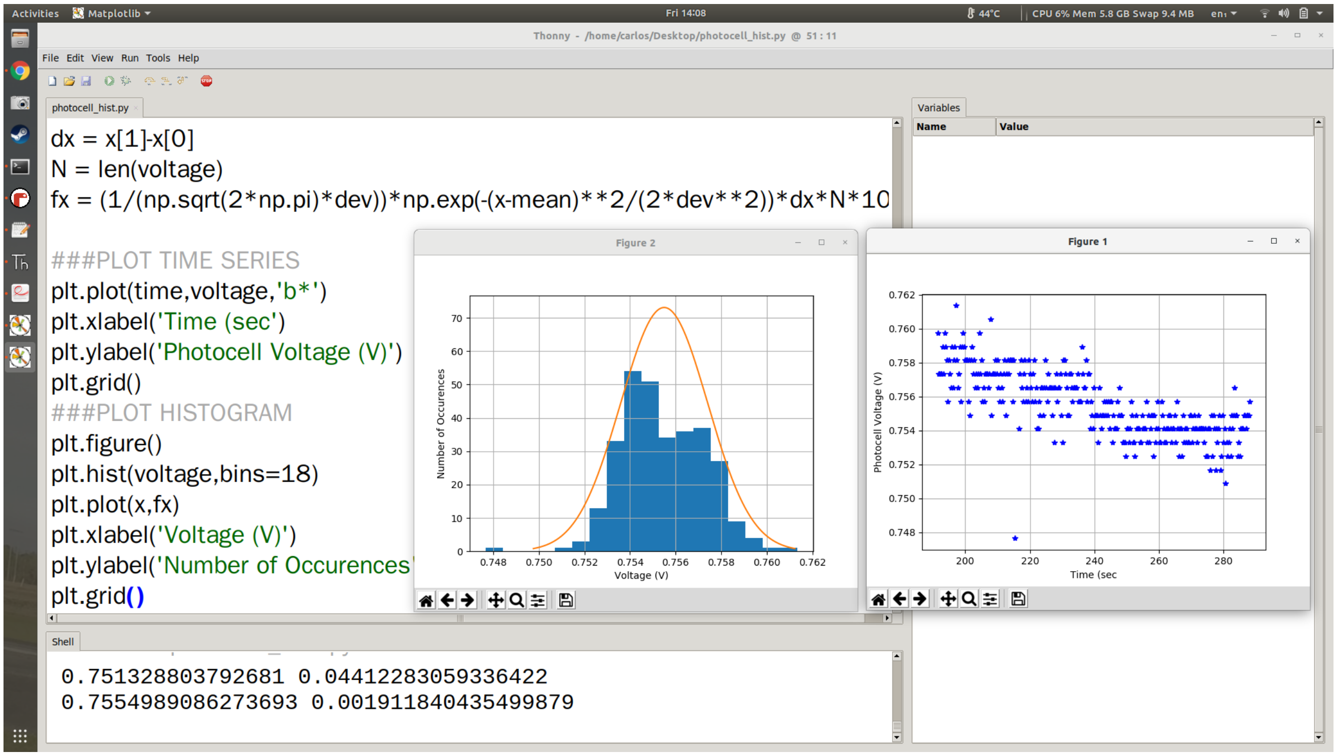
When I ran this experiment for a second time my CPX started and stopped 3 separate times. You'll see in the time series plot below that the voltage dipped in the first set and the second data set had some weird bumps probably from me changing tabs on my chrome tab. The photocell was close to my computer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



16.5 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.

16.6 Assignment

For this assignment you are to build another voltage divider circuit using a photocell and measure the voltage across the photocell using the analog to digital converter on the CPB/CPX. First take 30 seconds of data where you cover the photocell with your hand and also shine a flashlight on it. This will represent the min and max values of Lux as well as the min and max values of the resistance in your photocell. After that you are to run some statistics on the photocell by taking 1000 data points three separate times at three different light levels. The three light levels are low light (hand covered), ambient light (office light or shade light), and high light (flashlight or sunlight). For each set of 1000 data points, create a histogram, compute the mean, median, standard deviation and plot a Gaussian distribution curve on top of the histogram.

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%
5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. Include a photo of your circuit and write the value of the resistor that is in series with your photocell. Also, explain how you varied the light levels - 10%
2. Plot voltage, resistance (converted from voltage) and Lux (converted from resistance) of the photocell vs time for your first 30 seconds of data where you randomly vary light conditions. - 20%
3. Include the mean, median, and standard deviation of all light levels in Lux - 10%
4. Using the mean, standard deviation and number of samples, compute your 90% confidence interval for your 3 light levels - 10%
5. Include 3 histogram plots of low light, ambient light and high light levels in units of Lux. On top of the histogram plot the normal Gaussian distribution to see how close your histogram is to a Gaussian distribution - 30%

17 Servo Calibration (PulseWidthModulation(PWM) = f(angle)) and Feedback Control

17.1 Parts List

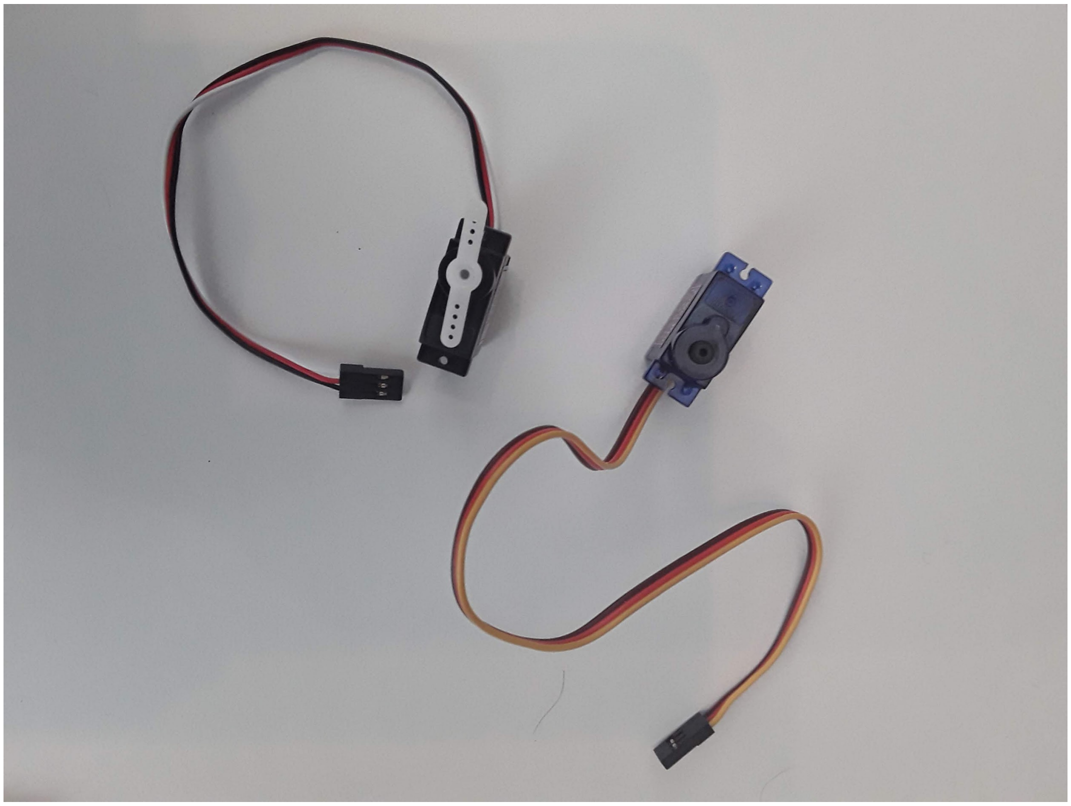
1. CPX/CPB
2. USB Cable
3. Laptop
4. protractor or piece of paper
5. servo
6. Alligator clips (x3)

17.2 Learning Objectives

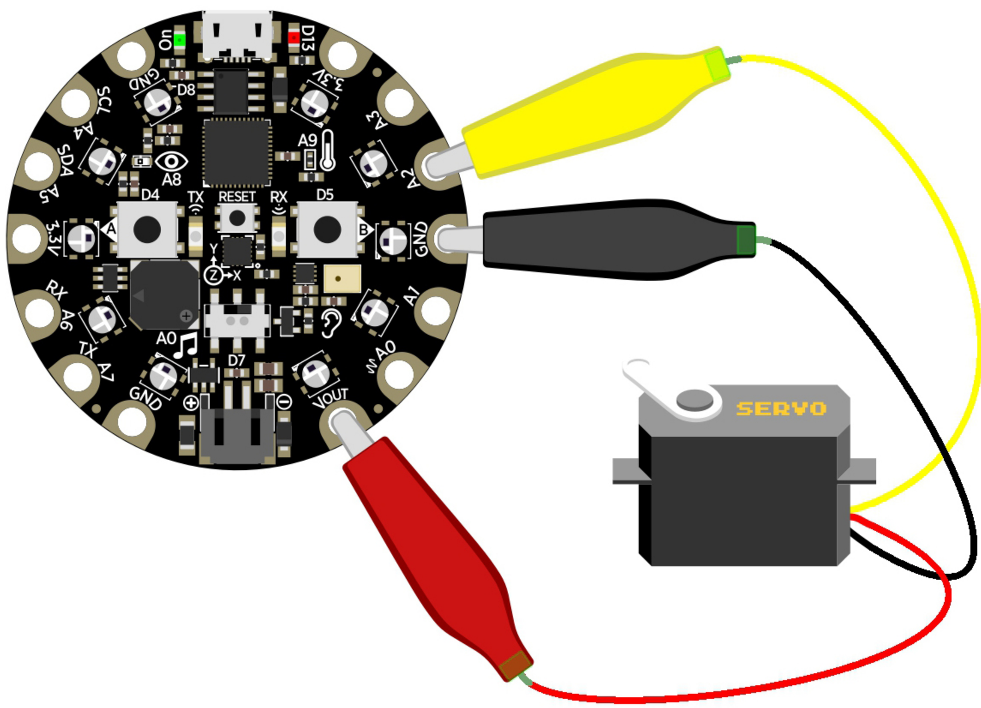
1. Understand what a PWM signal is and how it affects a servo
2. Understand the inner workings of a servo and how to make it move
3. Practice first order regression and calibration techniques
4. Understand how to compute roll and pitch from an accelerometer
5. Learn the fundamental concepts of feedback control
6. Learn how to Combine two codes (servo code and accelerometer code into one)

17.3 Getting Started

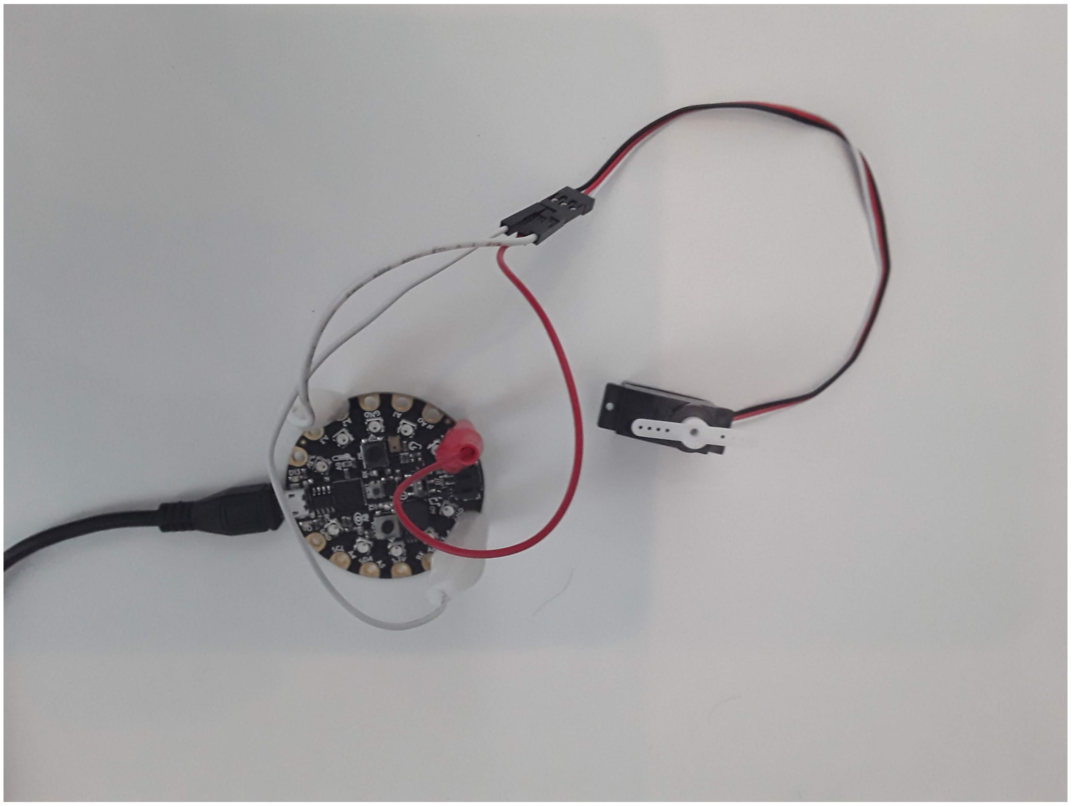
For this lab we are going to learn how to drive a servo. Servos accept what are called PWM signals which are basically square waves of varying frequency. The servo itself has a microprocessor on board that turns a DC motor based on the incoming PWM frequency which is typically called a duty cycle. The DC motor runs through some gear to rotate a shaft. Because of this rotation, you can make a number of things move! Servos are used for all sorts of things, opening doors, deflecting control surfaces on RC aircraft and many more! The neat thing about PWM signals is that they can not only move servos but they can also drive speed controllers to turn 3 phase motors and even change the light intensity of LEDs. Servos typically come in two different color schemes as shown below.



As you can see, servos have 3 pins (GND, PWR, SIGNAL). A wiring diagram of a servo connected to the CPX/CPB is shown below (Couresy of Kattni Rembor).



The brown or black wire is GND, the red wire needs to go to a 5V signal so for the CPX it needs to go to the VOUT pin and the yellow or white wire is the signal wire which need to go to an analog port on the CPX that supports PWM signals. Which ones support PWM signals? Adafruit Learn has a great description of which pins support PWM signals but as a quick check you can use the following analog pins: A1, A2, A3, A6 and A7. In my circuit I just picked pin A2 since I've been using it so much in the past.



Very important: It is recommended to power a servo through an external power supply instead of the CPX. Servos can draw a lot of current and the CPX although it supports 5V can only provide so much power (P). Remember that P = VI so if P is low it means current is low. If the servo pulls more current than the CPX can provide the CPX will "brown out" which means it will go into a safe-mode setting. If you have the AA power supply you may consider doing that. For small servos you hopefully won't have any issues.

As I said before, a servo takes in a square wave. The square wave has a duty cycle in units of microseconds. If you send a roughly 500 us square wave to a servo it will rotate all the way to the left. If you send a roughly 2500 us signal to the servo, it will turn all the way to the right. The code to send PWM signals has been thoroughly explained in the Adafruit Learn system. I also have a simple servo.py script on Github. In this code as usual the top 3 lines are used to import the necessary modules. The pulseio module is used here to create a servo object on line 6 by connecting to pin A2. Make sure to change the pin to whatever pin you have the signal wire hooked up to. Lines 9-12 create a function that pulse in milliseconds and compute the duty cycle of PWM signal. Lines 16-19 then kick off an infinite while loop where a 800 us signal is sent to the servo and then a 2000 us signal is sent using a for loop which starts on line 16. You'll see servo command on line 18 which is responsible for sending the microsecond signal to the servo. The function servo_duty_cycle converts the pulse in milliseconds to a duty cycle. The value is then passed to the attribute of the servo object servo.duty_cycle. If you put this code on the CPX and run the code you will hopefully see your servo turning left and right in 1 second intervals. I did this project myself and posted a YouTube video about it.

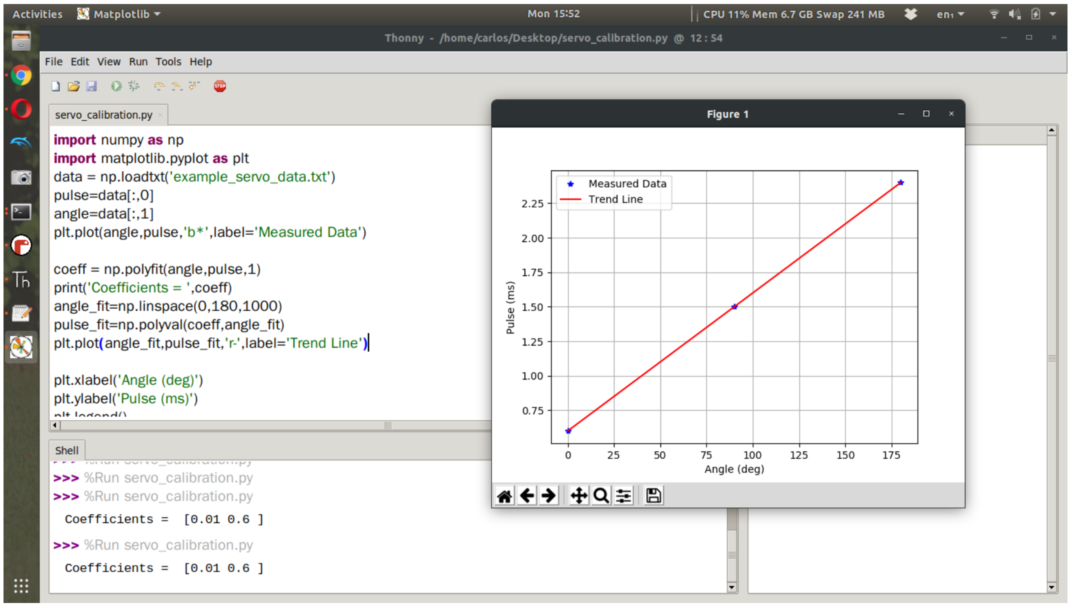
Besides making the servo move back and forth I'd like you to vary the pulses on line 16 **SLOWLY** until the servo can't move any farther. This line of code is a for loop which loops through the array currently showing [0.8,2.0]. If you change that array to [0.9,1.2,1.5,1.8] the servo will move to a pulse in milliseconds of 0.9, 1.2, 1.5 and then 1.8. The for loop is a great way to loop through multiple commands. Using this array, determine the minimum pulse you can send to the servo and the maximum pulse you can send to the servo. If the servo makes a funny noise it means you sent a signal outside the bounds so try a different signal. Hence the need for moving the pulse signal slowly. If you change the array to just 1 number [0.8] the servo will just move to 1 angle and stay there forever. **NOTE THAT IN CIRCUITPYTHON VERSION 7.0.0 YOU HAVE TO USE THE PWMIO LIBRARY INSTEAD OF THE PULSEIO LIBRARY**.

```python
import time
import board
import pulseio

# Initialize PWM output for the servo (on pin A2):
servo = pulseio.PWMOut(board.A2, frequency=50)

# Create a function to simplify setting PWM duty cycle for the servo:
def servo_duty_cycle(pulse_ms, frequency=50):
    period_ms = 1.0 / frequency * 1000.0
    duty_cycle = int(pulse_ms / (period_ms / 65535.0))
    return duty_cycle

# Main loop will run forever moving between 1.0 and 2.0 mS long pulses:
while True:
    for pulse_ms in [0.8,2.0]:
        print('Milli Second Pulse = ',pulse_ms)
        servo.duty_cycle = servo_duty_cycle(pulse_ms)
        time.sleep(1.0)
```

If you notice, sending a pulse signal in microseconds moved the servo to a specific angle. Thing is I would like to be able to move the servo to a specific angle rather than having to just guess and check like we did in the last lab. So what we're going to do is start at the minimum pulse signal you computed in the last project and then change the servo pulse in equal increments until we reach the maximum servo pulse signal. When I did this lab I found that 0.6 ms was just about the smallest I could get the servo to move. We're going to call this 0 degrees. My maximum pulse signal ended up being about 2.4 ms. So I want you to test 10 different points between your specific maximum and minimum value which will hopefully be different for all of you. Everytime you test a pulse I want to measure the angle the servo makes with the minimum value being 0 degrees. Create a table of data with two columns. In the first column put pulse in milliseconds and in the second column put angle of servo in degrees. Use a protractor to measure the angle. If you don't have a protractor make one or you can download a picture and hold your servo up to the screen. I did this project with just 3 data points and here are my data points. Again you need to have around 10 data points

Pulse (ms)	Angle (Degrees)
0.6	0
1.5	90
2.4	180

Take your table of data and put it into a spreadsheet and save the data as a CSV or simply put your data into a text file. Since I only had 3 data points I just put them into a text file. Plot the data in Python on your desktop computer with servo angle on the x-axis and duty cycle on the y axis. Using the data, determine if the data set is linear, quadratic or cubic. Fit a trend line to the data and plot your trend line on top of the data. If you need help with trend lines in Python you can watch this video I posted on Youtube. I also made a helpful python script with some fictitious data on Github that fits the data with linear and quadratic fits. Here is my data plotted alongside the trendline in Python. I sort of made up the data and made it perfect on purpose so my trendline is perfect. Yours will not be so perfect.



You'll notice that I first import the data from the text file using the np.loadtxt function and then I use the polyfit and polyval functions to create the trendline. The polyfit function requires you to give it the X and Y axes and the order of the trendline which since the trend line is linear I sent it a 1 but you could easily do 2 for quadratic or 3 for cubic. I then print the coefficients which are [0.01 0.6]. This means my trend line looks like this.

$$Pulse = 0.6 + 0.01 * Angle \tag{12}$$

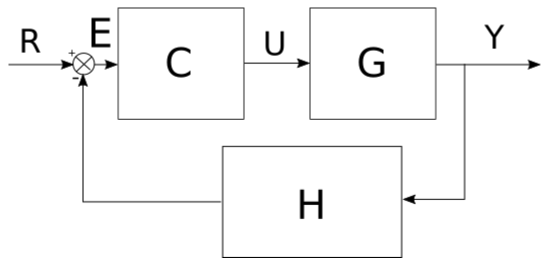
Where Pulse is in ms and Angle is in degrees. Now you have an equation where you can use angle in degrees to compute the pulse in milliseconds. In the code I've posted I then use np.linspace to create 1000 data points from 0 to 180 degrees and then use the polyval function to compute the pulse for all 1000 angles I created using the linspace command. I finally plot the trend line in red and use the remaining part of the script to create labels and legends. Once you have this plot write down your coefficients and create an equation like I did above. Again your numbers won't be so neat. Once you have this equation, return to Mu and create a function using the def keyword that takes in an angle as an input and then returns a pulse signal in milli seconds. It will look sometime like this

```
def angle2pulse(angle):
        return 0.6 + 0.01*angle
```

Note: Functions in python must be after all your imports but before your while loop. If you put this function inside your while loop the code will not work. Using that equation, modify your servo.py script to have the servo move through the following angles, 0, 45,90,135 (your servo may not travel to 180 degrees). Verify that your equation is working correctly by placing your protractor below the servo. Much of the code required for this project is not included because it is left as an exercise for the student.

17.4 Feedback Control

Feedback control can be its own course or multiple courses but can be broken down into a few simple steps. The goal of feedback control is to drive the "state" of a "system" to a desired "command" by sending a "control" signal to the "system". The figure below shows a standard block diagram for a dynamic system.



In this figure, R is the reference signal or the commanded signal. The reference signal feeds into a summation block where the reference signal is substracted from the measured signal. The measured signal is the output of the H block which represents the sensor. The output of the summation block is the error signal E. The error signal feeds into the controller C and outputs the control signal U to the plant system G. The output of the plant G is Y or the state. The state Y feeds into the sensor block H to measure this signal and out the measured signal to the summation block. I explain this in much more detail in a controls overview Youtube video.

An example feedback control system could be employed for the longitudinal pitch dynamics of an airplane. In this case the aircraft pitches through the angle θ and the elevator δ_e is used to control the pitch angle. Typically the pitch angle is measured with an angle sensor or even an accelerometer. The elevator is then typically controlled with a servo. Thus, for this lab we are going to make the bare bones circuitry required for pitch angle control of an airplane. The "system" then is the aircraft. The "state" is the "pitch" angle (measured by the CPX), the "command" is the desired pitch angle (programmed by you the pilot in command) and the "control" signal is the elevator command (actuated by a servo). I'm using the same circuit I created in parts 1 and 2.

To measure the pitch angle using the accelerometer we are going to use the same equation that we used when measuring the angle of a pendulum (See chapter 19). I'm going to rotate the CPX in the X/Z plane which can be done by rotating the USB cable where it plugs into the CPX. In this case we can ignore the Y axis data. If you print the raw accelerometer data, you'll notice that when you place the CPX directly onto a flat surface, the x and y axes read a value around 0, while the z axis reads around gravity ($9.81 m/s^2$). If you then rotate the sensor clockwise 90 degrees, the x axis is reading about gravity while the z axis is now zero. This means we can form a triangle and get the angle using these two axes using the equation below which gives angle in degrees.

$$\theta = tan^{-1}(x/z)\frac{180}{\pi} \tag{13}$$

On the CPX specifically we want to import the math module and use the atan2 function. This measurement of the pitch angle would represent the H block in the figure above.

The elevator on an airplane is a control surface responsible for pitching the aircraft up and down. For this example we are going to assume that the desired pitch angle is 0^o. This means our "error" signal is going to be 0 minus the pitch angle $e = \theta_C - \theta = 0 - \theta = -\theta$. Our "control" signal will be the angle of the elevator. As I said, we are going to use a servo to control the elevator so this just means we need a way to relate our "error" signal to servo pulse width. There are a few steps here before we can move on. First, we need to relate our "error" signal to the "control" signal which will be the elevator pitch angle. I've made a table below to explain what I mean.

Pitch (deg)	Error (deg)	Elevator (deg)
-90	90	+90
0	0	0
+90	-90	-90

This table basically says that if the aircraft is level with a pitch angle of zero I want the elevator to be zero as well. If the aircraft pitches down, I want the elevator to pitch up and counteract that rotation. Using these three data points I can create a simple equation to relate elevator angle to pitch angle.

$$\delta_e = -\theta \tag{14}$$

This is a simplified version of proportional control however. In reality the control signal should be given by the equation below.

$$\delta_e = Ce \tag{15}$$

where C is the controller and e is the error signal. If we replace the error signal with $e = \theta_c - \theta$ and the controller with just proportional gain k_p we arrive at the equation below.

$$\delta_e = kp(\theta_c - \theta) \tag{16}$$

It's easy to see that if $k_p = 1$ and $\theta_c = 0$ the equation for the elevator would simplify to equation 14 Now that we have the elevator pitch angle we need to relate this to the servo angle. Servos can only move from 0 to 180 degrees which means we can't have the servo go negative. Thus we need to offset the elevator angle to the servo angle. Again we can make a table here.

Elevator (deg)	Servo (deg)
-90	0
0	90
+90	180

This also results in a simple equation to relate servo angle (s) to elevator angle (δ_e).

$$s = \delta_e + 90 \tag{17}$$

Finally, we can then use our calibration coefficients (See chapter 17) to relate servo angle to pulse width. When I calibrated my servo I obtained the following equation where μ is the pulse in PWM.

$$\mu = 0.6 + 0.01s \tag{18}$$

So now I have an equation that relates the pitch angle from the airplane θ to the elevator deflection angle δ_e. I can then relate the servo deflection angle to servo deflection angle (s) and then finally the servo deflection angle to PWM signal μ. With these 3 equations I can now program my servo to respond to changes in the pitch angle of the CPX.

Using the accelerometer to measure pitch and the servo to deflect the elevator I can put the entire system together. Pictured below is my code which again is also online on Github. **Note that the version on Github is constantly edited and as such will be slightly different than the version below.**

```python
import time
import board
import busio
import digitalio
import adafruit_lis3dh
import math
import pwmio

# Initialize PWM output for the servo (on pin A2):
servo = pwmio.PWMOut(board.A6, frequency=50)

# Create a function to simplify setting PWM duty cycle for the servo:
def servo_duty_cycle(pulse_ms, frequency=50):
    period_ms = 1.0 / frequency * 1000.0
    duty_cycle = int(pulse_ms / (period_ms / 65535.0))
    return duty_cycle

##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
```

The first 22 lines here will hopefully seem familiar. Line 1-7 are import commands of all the various modules needed. Lines 10-13 create the definition that converts pulse width to duty cycle. Line 16 creates the servo and lines 19-22 create the accelerometer. Hopefully this is a good example of combining different codes together to get a more complex piece of software. Lines 24-45 include a very long while loop. I will try and go through each line. Line 26 grabs the accelerometer data on the CPX. Line 28 uses the x and z axis accelerometer data and converts the values to pitch angle using the atan2 function in the math module which was imported on line 4. Line 30 computes the elevator pitch angle and line 32 computes the servo deflection angle. Line 34-37 is a type of signal conditioner called a saturation filter. Basically, I don't want the servo to break because I tried to make the servo rotate more than 180 degrees or less than 0 degrees. So I created two if statements that restrict the servo to be within these two values. If the servo angle is less than 0 as stated on line 34, the servo angle is set to 0 on line 35. If the servo angle is greater than 180 as stated in line 36 the servo angle is set to 180.

```
24 while True:
25     #Get Accelerometer
26     x,y,z = lis3dh.acceleration
27     #Get theta
28     theta = math.atan2(x,z)*180/3.141592654
29     #Get elevator angle
30     de = -theta
31     #Get servo angle
32     s = de + 90.0
33     #Saturation Filter
34     if s < 0:
35         s = 0.
36     if s > 180:
37         s = 180.
38     #Get pulse width
39     pulse_ms = 0.6 + 0.01*s
40     #Get duty cycle
41     duty_cycle = servo_duty_cycle(pulse_ms)
42     #Actuate Servo
43     servo.duty_cycle = duty_cycle
44     print(theta,de,s,pulse_ms)
45     time.sleep(0.1)
```

Line 39 uses the calibration equation from the previous experiment to convert servo angle to pulse width. You'll need to replace these numbers with your servo since all servos are different. Line 41 uses the definition created on lines 10-13 to convert pulse width to duty cycle. Line 43 makes the servo move. Line 44 prints everything to Serial for debugging purposes and line 45 pauses the script for 0.1 seconds which helps with some twitchiness in the servo. When I did this I didn't have to program a complementary filter so I guess the servo may have it's own low pass filter. Either way this circuit is ready to be placed on an aircraft. Whether or not it is effective is a completely different story. We could take an entire aircraft design course or an undergraduate controls and systems dynamics course but to start let's just simulate this control system and see how it does. First we need to write the dynamic equation for pitch of an aircraft. That is shown below.

$$\frac{J}{q_\infty S\bar{c}}\ddot{\theta} - \frac{\bar{c}}{2V}C_{mq}\dot{\theta} - C_{m\alpha}\theta = C_{m\delta_e}\delta_e + C_{m0} \tag{19}$$

In the equation above J is the inertia of the aircraft, $C_{m0}, C_{mq}, C_{m\alpha}$ and $C_{m\delta_e}$ are non-dimensional coefficients, $\bar{c}$ is the mean aerodynamic chord, V is the velocity, $\bar{c}$ is the mean aerodynamic chord, S is the planform area of the wing and q_∞ is the dynamic pressure. The dynamic pressure is given by the equation below

$$q_\infty = \frac{1}{2}\rho V^2 \tag{20}$$

where ρ is the air density. To simulate this system use the table below.

$C_{m\alpha}$	-2.19	C_{mq}	-24.45	$C_{m\delta_e}$	-1.15
$\bar{c}$	0.2286 m	S	0.34 m^2	J	0.1232 $kg-m^2$
V	15 m/s	ρ	1.225 kg/m^3	C_{m0}	-0.076

Now that the dynamics are written, the second order equation can be integrated using a standard numerical tool. In this case we'll assume the angular velocity is zero and the initial pitch angle is 5 degrees remembering to convert this to radians first. Note that proportional control is not sufficient to stabilize your aircraft but I'll leave that discussion to your controls professor.

17.5 Assignment

This servo project can be split into 2 projects at your instructors discretion. The first part involves learning how to rotate the servo and then calibrating the servo so that degrees of rotation can be related to PWM in microseconds while the second part deals with feedback control. In part 1 ensure that you can rotate the servo and then experimentally determine the minimum and maximum PWM signal you can send to the servo before it stops rotating anymore. Then create a calibration equation and plot with degrees on the x-axis and PWM on the y-axis. Take at least 10 data points for your calibration equation and use a protractor on your computer screen to obtain angle in degrees of your servo. Your calibration equation needs to be linear. In part 2 you are to use the angle obtained by the accelerometer to feedback to the servo using proportional control. You are to do this for a k_p value of 1 and 2. In addition, you are to simulate the second order system above for 10 seconds with varying values of k_p and plotting the results. **Note, for this lab, once you have the minimum and maximum values of your servo, you need to add a saturation filter into your code so that the value sent to your servo is always within these bounds. Otherwise you could damage or even break your servo.**

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%
5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

Part 1

1. Include 2 photos of your servo with one photo rotated to the maximum and the other photo rotated to the minimum degrees. Also, include a screenshot of Mu showing the PWM pulse in the serial monitor for both rotations. - 10%
2. Report the minimum and maximum values of your servo in a table. Be sure to report the PWM signals in milliSeconds as well as the angle of the servo in degrees. Give at least 2 decimal points in your calculation - 10%
3. Include your raw calibration data with PWM signal in one column and degrees in the other column - 10%
4. Include a Figure of your calibration data plotted in Python with your trend line on top. Put your r^2 value in the title of the plot - 20%
5. Write your regression equation as servo_pulse = m*servo_angle + b being sure to replace m and b with your actual values. Remember to round to appropriate significant figures - 10%
6. Using your calibration equation imbedded in your Mu code, compute the pulse required to rotate your servo to 90 degrees. Report on how close it is to 1.5 ms. Include a photo of your servo rotated to 45 degrees and a screenshot of your calibration equation placed into Mu with your pulse value in the Serial monitor. The code must also show your saturation filter as well - 20%

Part 2

1. Include an annotated photo of your circuit showing the accelerometer, wiring, and servo. - 20%
2. Include a photo of your CPX rotated to 45 degrees and your servo rotated with $k_p = 1$ and 2. Have your servo held up to your protractor when rotated so readers can see the angle at which your servo is rotated. - 40%
3. Simulate the second order system above for 10 seconds with $k_p = -1$ and $k_p = -10$ for a $\theta_c = 45^o$. Plot θ vs time for both values of proportional gain. - 20%

18 Time Constant of a Thermistor

18.1 Parts List

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

18.2 Learning Objectives

1. Learn the basic form of a first order system
2. Learn the basic solution of a first order system
3. Understand settling time and how that effects engineering
4. Applied Estimation of a First Order system

18.3 Getting Started

For this lab we're going to use the thermistor located on Pin A9. In a previous lab we measured the analog signal from the pin and also used the *adafruit_thermistor.py* module that converts the voltage from the thermistor to celsius. Here though we're going to explain a bit more about the thermistor. In this case, the thermistor is wired in a voltage divider circuit in series with a 10000 Ω resistor. This means that the voltage across the thermistor can be converted to its resistance using the voltage divider equation below similar to the equation used for the photocell.

$$V_{thermistor} = V_{out}\frac{R_{thermistor}}{R_{series} + R_{thermistor}} \tag{21}$$

Where $V_{out} = 3.3V$ and $R_{series} = 10000\ \Omega$. The equation above can be inverted to obtain the resistance in the thermistor which can then be converted to Kelvin. This means that the digital output from the analog to digital converter can be converted to voltage and then to resistance as has been done for various ADC labs in this textbook. Remember that $V_{thermistor} = V_{out}D_o/65535$ where D_o is the digital output from the ADC. Once you have the resistance from the thermistor, you can use a modified version of the Steinhart-Hart equation which is given below.

$$T_{thermistor} = \left(\frac{1}{T_0} + \frac{1}{\beta}log(R_{thermistor}/R_{series})\right)^{-1} \tag{22}$$

where $\beta = 3950$ is a heat transfer coefficient specific to the bulk semiconductor material over a given temperature range of interest and $T_0 = 298.15K$ is the nominal temperature of the semiconductor in Kelvin. Note that in the equation above, the output is in Kelvin.

For this lab though, we're not just going to measure the temperature from the thermistor, we're going to analyze the dynamics of the thermistor and how it responds to an abrupt change in temperature. It is said that a themistor can be modeled as a first order system. The basic form of a first order system can be shown below.

$$\dot{T} = \sigma(T_f - T) \tag{23}$$

where T is our state variable and $\dot{T}$ is the derivative of our state. For this example let's assume that T is temperature. In this case T_f is an external temperature that is either higher or lower than current temperature causing the derivative of temperature to be non-zero. You can see in this case that once the temperature of the system is equal to the external forcing temperature, the derivative of the temperature of the system goes to zero. This implies that the system has reached equilibrium. The variable σ has units of Hz or $1/sec$. The inverse of σ is τ the time constant.

$$\tau = \frac{1}{\sigma} \tag{24}$$

The time constant of the system is related to how quickly the system responds to change or even how long it takes for the system to reach equilibrium. Reaching equilibrium quantitatively happens when the state has changed 98%. In other words the state of the system is within 2% of the equilibrium state. This is called the settling time T_s. The time constant is related to the settling time using the equation below.

$$\tau = \frac{T_s}{4} \tag{25}$$

The only question now of course is what is the solution $T(t)$ or rather the temperature as a function of time. In this case, the solution to the above dynamics equation can be solved using standard first order differential equation techniques to obtain the solution below.

$$T(t) = (T_0 - T_f)e^{-\sigma t} + T_f \tag{26}$$

Looking at the equation now you can see some properties right away. When $t = 0$, the first term reduces to $(T_0 - T_f)$ which means the initial temperature is T_0 or the initial temperature. When $t \to \infty$, the first term drops to zero and thus the temperature is T_f or final temperature.

18.4 Temperature Change Ideas

In this lab we're going to get the time constant of the thermistor on board the CPX. If you take data on the CPX and walk outside or place the thermistor directly into a fridge, the temperature change will not be immediate. Remember that the thermistor is a resistor that changes with temperature. The ADC on the CPX converts the voltage across the thermistor to temperature. The Adafruit Learn system does a bit of work to explain the conversion from voltage to temperature. My version of the code is also on my Github.

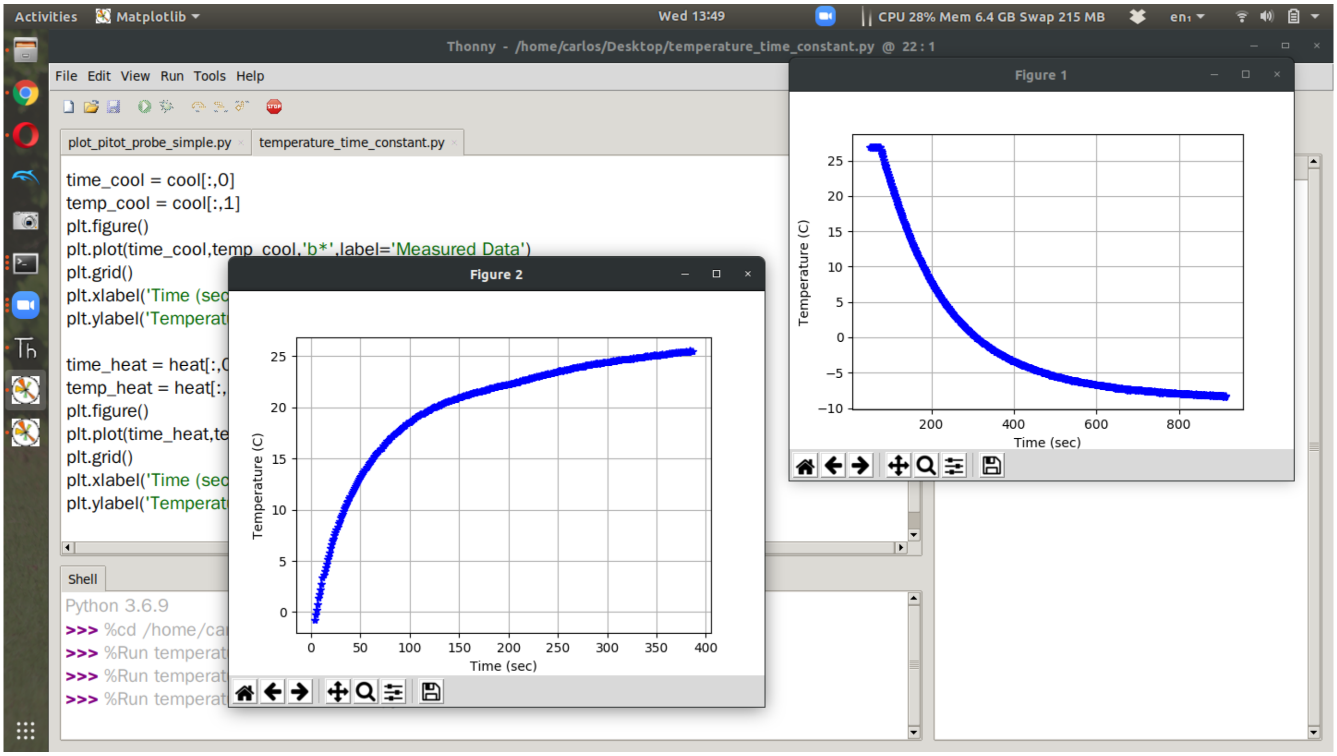
```
import time
import board
import adafruit_thermistor

#Temperature Sensor is also analog but there is a better way to do it since voltage to temperature
#Is nonlinear and depends on series resistors and b_coefficient (some heat transfer values)
#thermistor = AnalogIn(board.A9) ##If you want analog
thermistor = adafruit_thermistor.Thermistor(board.A9, 10000, 10000, 25, 3950)

while True:
    temp = thermistor.temperature
    #temp = thermistor.value #if you want analog
    print((temp,))
    time.sleep(0.5)
```

For this lab in order to estimate the time constant of a thermistor you need to change the temperature somehow.

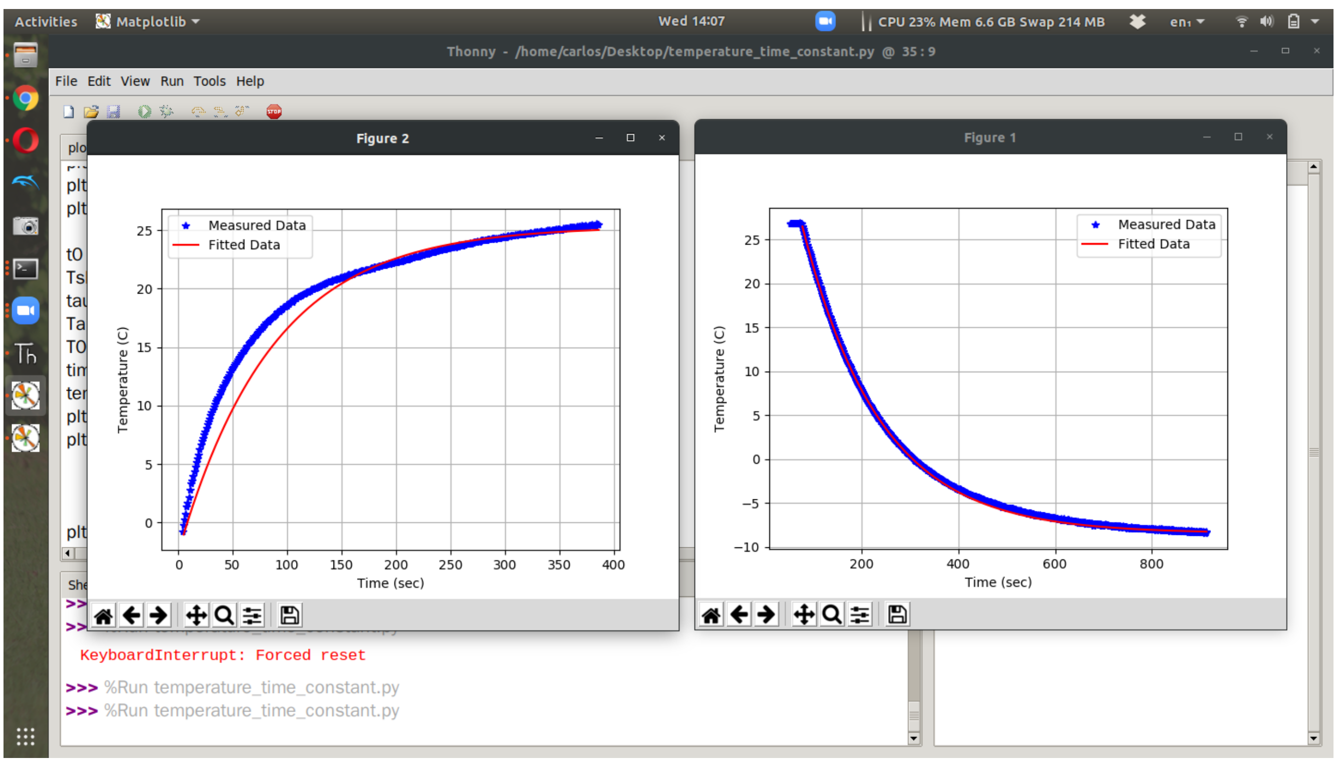
1. Start logging data and then place the CPX into the fridge
2. Put the CPX into the fridge and then pull the CPX out of the fridge and watch the temperature return to ambient
3. Walk outside on a hot (or cold) day and watch the CPX change temperature due to your A/C
4. Walk inside and watch the CPX get warmer (or colder) as your HVAC changes the CPX temperature.

18.5 Estimating Time Constant

I did the first two examples and plotted both data sets in the same script as shown below. I opted to use method 1 (see chapter 8) from the datalogging project and just have the data print to Serial and then unplug the CPX when I'm done taking data and copy and paste the data into a text file.



At this point it's possible to get the time constant by remember that the settling time (time it takes the temperature to settle out) is equal to 4 times the time constant (τ) and thus the time constant is the settling time divided by 4. After computing the settling time for both data sets and overlaying the equations on the measured data I get these two plots.



What was interesting was that the time constant for heating up was 62.5 seconds and for cooling down it was

155 seconds. The time to get cold was way slower than heating up. I'm not a heat transfer expert so I won't comment as to why this happened. One other import note I'd like to mention is the cool down phase was much more accurate than the heat up phase. This is most likely because when I pulled the thermistor out of the fridge I touched it with my hands and then moved it to a table. There was also alot more airflow outside the fridge which would change the overall dynamics. Still, the fitted data matches up pretty well and I hope yours does too.

18.6 Assignment

For this assignment, you are to change the ambient temperature of the CPX/CPB through heating or cooling and then record time and voltage while waiting for the CPX/CPB to return to ambient temperature. You are to then plot voltage, resistance and temperature as a function of time and then using the graph of temperature (not using a curve fit), estimate T_0, T_f,σ,T_s and τ. Finally, using equation 26, plot the estimate of $T(t)$.

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%
5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. Take a picture of you heating or cooling the CPX/CPB - 20%
2. Plot the raw voltage and resistance in the thermistor vs time - 20%
3. Plot your temperature data with time on the x-axis and temperature on the y-axis and overlay your fitted data on top of your measured data. Points will be given based on how well your fit is - 20%
4. Include a table of your estimated parameters - 20%

19 Natural Frequency and Damping of a Second Order System and Issues with Aliasing

19.1 Parts List

1. Laptop
2. CPX/CPB
3. USB Cable
4. Some sort of oscillating system like a swinging pendulum or vibrating ruler

19.2 Learning Objectives

1. Learn the basic form of a second order system
2. Understand the difference between underdamped, critically damped, and overdamped
3. Understand natural frequency and damping ratio
4. Applied Estimation of a Second Order system
5. Understand the pitfalls of aliasing

19.3 Getting Started

A second order system undergoing free motion will have dynamics that look like this

$$\ddot{\theta} + 2\zeta\omega_n\dot{\theta} + {\omega_n}^2 \tag{27}$$

In this case, the solution to the above equation can be solved using standard second order differential equation techniques to obtain the solution below.

$$\theta(t) = \theta_0 e^{-\sigma t} cos(\omega_d t) \tag{28}$$

$$\omega_n = \sqrt{\sigma^2 + {\omega_d}^2} \tag{29}$$

$$\zeta = \frac{\sigma}{\omega_n} \tag{30}$$

Looking at the equations you can see that if the time series of the oscillations are known, the damping constant and damped natural frequency can be obtained. These two values can be combined to obtain the natural frequency and the damping ratio. So let's get some data.

19.4 Oscillatory Ideas

What I'd like you to do for this lab is to find some sort of parameter that varies in some sort of sinusoidal way. I've come up with a few ideas below. You may pick anyone you want although some are easier than others.

1. Drive over a speed bump - If you drive over a speed bump slowly and place your CPX on the dashboard your car will hopefully vibrate for a few seconds after you drive over it. Your acceleration will look somewhat like a sine wave.
2. Vibrate a ruler - If you place a ruler on the edge of a desk and deflect it, it will vibrate. If you place the CPX on the ruler you'll be able to measure the vibration of the ruler.
3. Attach CPX to a spring or a weight to the end of rubber bands - Take the CPX and attach it to a weight of some kind and attach the weight to a spring or a set of rubber bands. This is a mass spring damper simple and will vibrate at the natural frequency of square root of stiffness divided by mass. I chose to do this example.
4. Build a Pendulum: If you decide to build a pendulum, you need to hang a weight on the CPX or attach the CPX to something heavy. This will make the ratio between cable to end mass much smaller and thus better for data and fitting. An example includes duct taping your CPX to a water bottle or something. It would also be better to use a string and mount the CPX to the string with an external battery pack and have the CPX log data internally. This way most of the mass would be concentrated at the tip of the pendulum. Another idea is to take a paper towel cardboard tube and tape the cpx inside it with the cable running through it. Then duct a large weight on the end of the paper towel tube and then hinge the top of the tube by skewering a screwdriver through it. This will allow the tube to swing like a pendulum rather than the string.

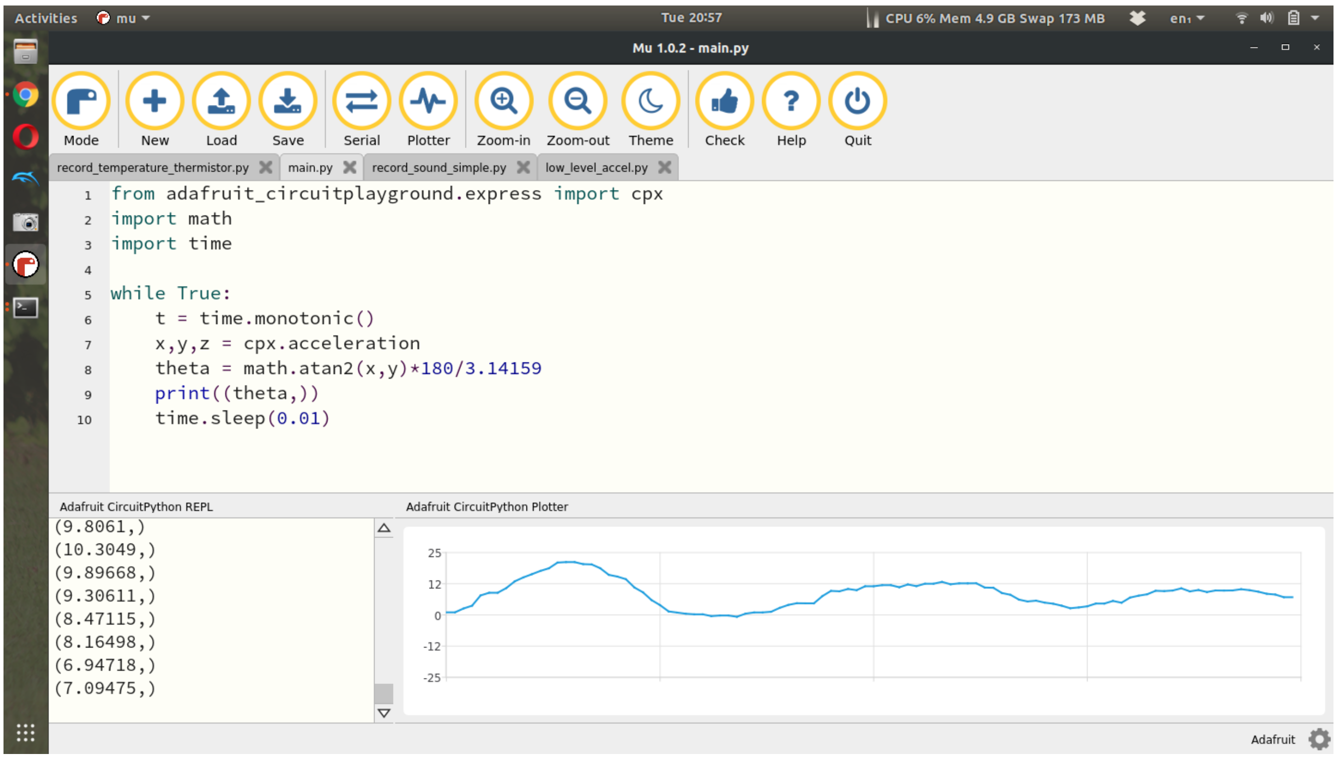
There are most likely many other options so try and get creative and find something oscillatory or dynamic in some way. Try to find something that changes relatively quickly. The temperature outside changes in a sinusoidal fashion but it's so slow it would take you days to do this experiment.

19.5 Pendulum Example

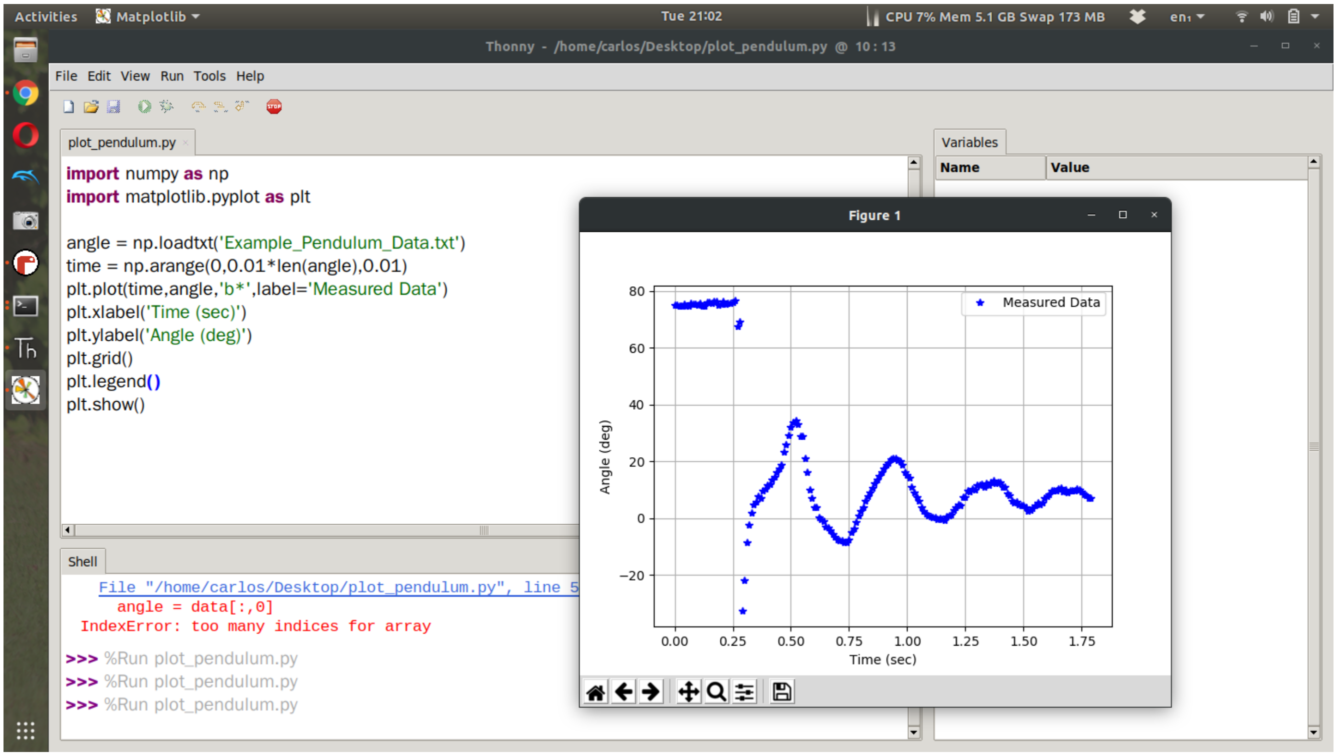
In this example I'm going to swing a pendulum in the X/Y plane of the accelerometer sensor so that I can ignore the Z axis data. I'm going to get the actual angle of the pendulum but if you're building something else you can ignore this part. You'll notice that when you point the CPX directly at you with the cable pointing up the x axis reads around 0 and the y axis reads around gravity. If you then rotate the sensor so that the cable is coming out of the left side of the CPX, the x axis is reading about gravity while the y axis is zero. This means we can form a triangle and get the angle using these two axes using the equation below which gives angle in degrees.

$$\theta = tan^{-1}(x/y)\frac{180}{\pi} \tag{31}$$

On the CPX specifically we want to import the math module and use the atan2 function. When I swing the pendulum then this is the result I get from Plotter in Mu.



If I then bring this into Python I get the following plot below. In my data set I only logged the angle. Since the time step between each point was 0.01 seconds I was able to just create a time series. It's pretty clear that there is some nonlinearity in the data so I chose to start the data at 0.5 seconds. Another thing I noticed was that the angle settled out to around 8 degrees so I chose to subtract off that bias from the angle data.



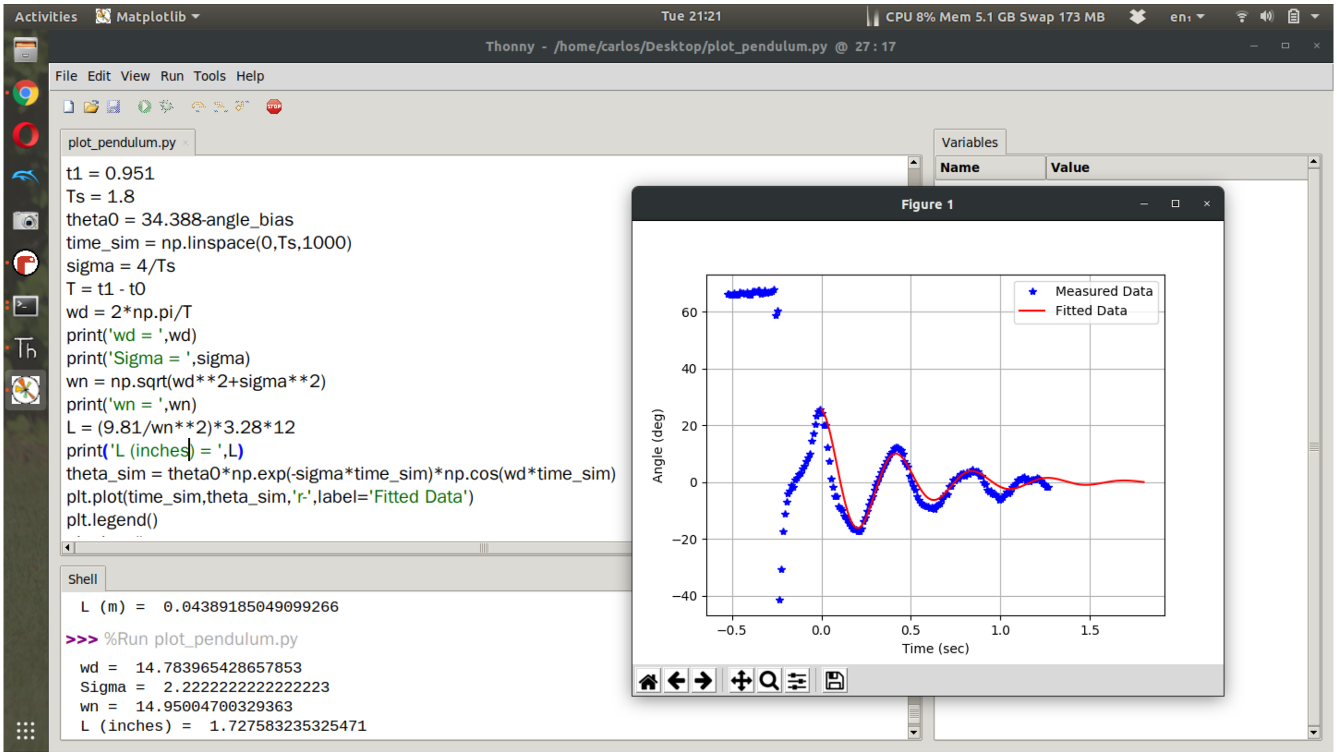
After trimming the data and removing some bias it was time to get my damping constant and damped natural frequency. There are a few equations that can help you obtain these parameters. First, the settling time is the length of time it takes for the oscillations to settle. The settling time can be used to find the damping constant. This is equal to:

$$\sigma = \frac{4}{T_s} \tag{32}$$

For my data set the settling time was about 1.25 seconds which gave a damping constant of 3.2. Once I had the damping constant I could obtain the damped natural frequency. This was done by measuring the distance between two peaks in the data set. There is a peak at around 0.5 seconds and another at around 0.95 seconds. I can use this to compute a period T. Period can be computed to angular frequency using the equation below.

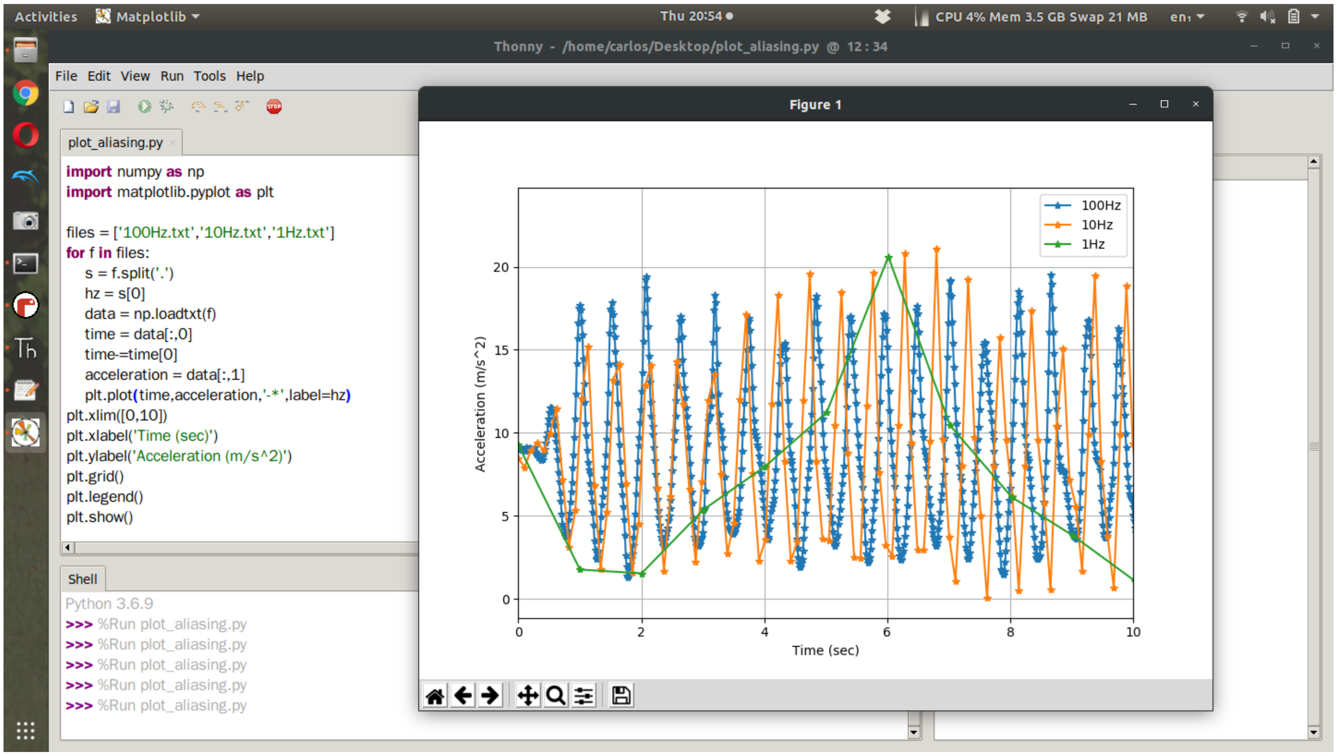
$$\omega_d = \frac{2\pi}{T} \tag{33}$$

Using the period in my wave form I obtained a damped natural frequency of about 14.8 rad/s. Using these values I can plot the simulated data on top of the measured data noting that my initial angle was about 35 degrees minus the bias of 8 degrees. When I first plotted the data I noticed that my fit wasn't entirely perfect. My period was correct but my damping rate was too high. I realized it was because my settling time was too big. I increased the settling time to 1.8 seconds and got this plot here. You can see that my fitted data lined up almost perfectly with my measured data.

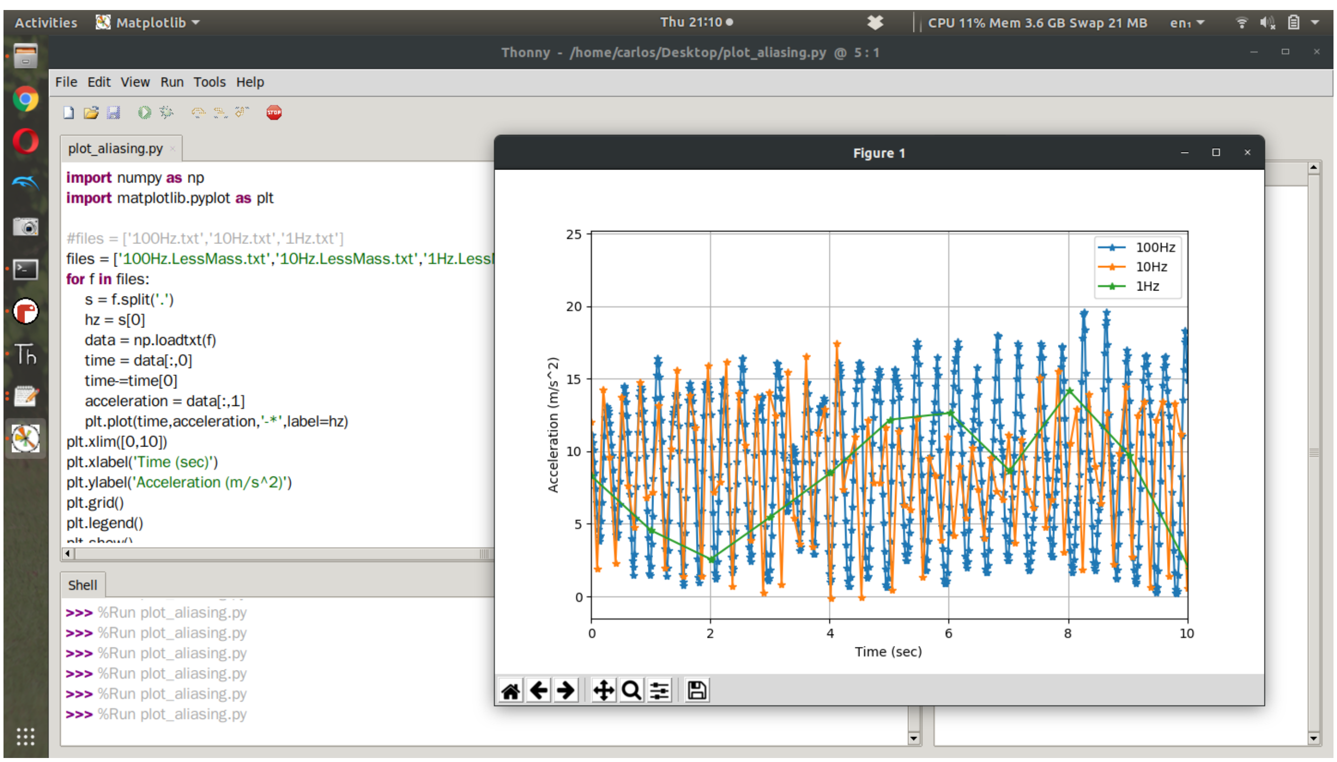


19.6 Aliasing

Aliasing occurs when you don't take data fast enough. The speed of taking data to avoid aliasing depends on the natural frequency of the system itself. To show this I changed the length of the pendulum which changed the natural frequency. I then proceeded to measure the angle as it oscillated at its natural frequency. I repeated this at sampling frequencies of 1, 10 and 100 Hz. The way I changed the sampling frequency was by changing the time.sleep value in the while True loop. The accelerometer code I used can be found on Github. After I finished the experiment I had 3 data files that I plotted on top of each other. This what I got with the longest string. It's easy to see in the photo that sampling at 1 Hz was way too slow to capture the natural oscillations of the water bottle. However, 100 Hz and even 10 Hz was plenty fast to sample the oscillations. According to the recorded data there was above 19 cycles in 9 seconds which is about 2 Hz. In this case, as long as we sample at 4 Hz the signal will be captured properly which is why 10 Hz and 100 Hz is able to capture the signal correctly.



Once I sampled my waveform at 3 different sampling rates I elected to shrink the length of the pendulum to change the natural frequency and see if that affected any aliasing I saw. The results are shown in the figure below.



In this case you can see that there were 33 cycles in 10 seconds which is 3.3 Hz. The Nyquist criteria states that I need to sample at 6.6 Hz. The Nyquist criteria is very specific though in that if you sample at twice the frequency, you will just obtain the correct frequency. This does not mean that you will capture every data point properly.

Hence in the chart above, the blue line at 100 Hz is perfect, the green line at 1 Hz is too slow (less than 6.6 Hz) and the orange line at 10 Hz captures the frequency correctly but between 5 and 8 seconds does not adequately capture the waveform. In my opinion, in order to sample the data effectively and not just obtain the frequency of the waveform, you need to sample at 4 times the natural frequency. So for the waveform of 3.3 Hz, the Montalvo frequency would be 13.2 Hz which is higher than the 10 Hz. This explains why the 10 hz sample is not perfect while the 100 hz sample rate is much better.

19.7 Assignment

For this assignment you are to create a system that oscillates and capture data while oscillating. You are to sample the system at 1, 10 and 100 Hz and plot the shifted and cleaned data for all 3 sample rates on the same graph. For the data sampled at 100 Hz you are then to estimate the initial amplitude, the settling time and the period using the graph. Then compute the natural frequency and damping ratio. Finally, using equation 28, plot the amplitude as a function of time on top of your 100hz sampled data. **Note, if you decide to make a pendulum you must ensure that the CPX/CPB does not spin when oscillating otherwise you won't be able to use your acceleration data easily.**

Grading Rubric

For every project you must turn in a properly formatted engineering report submitted as a PDF. The grading rubric is shown below.

1. The first page will be a title page with your name, title of the project and date - Pass/Fail
2. The second page will contain an introduction which explains the project itself, learning objectives and expected outcomes - 10%
3. After the introduction you then include the project specific requirements (see below) - 80%
4. All figures must have appropriate figure captions, axis labels and a paragraph explaining the figure - 10%
5. The last few pages will be appendices which are pass/fail. Failure to include both Appendix "A" and "B" will result in a zero in the assignment. Appendix "A" will be a link (Youtube or Google Drive) to a professional video recording of you and your screen explaining the project and showing any of the systems operate as asked. **This includes any portion of the experiment such as heating up your thermistor or rotating a potentiometer and showing the response in the Serial monitor**. Appendix "B" will be any and all code used in the project - **Failure to provide code or an adequate video results in a failure of the assignment.**

Project Specific Requirements

1. Include a photo of your experiment and explain the state that you are measuring - 10%
2. Plot the results of your data (make sure to connect your data points) vs time for all 3 sampling frequencies (1,10, 100Hz) and include a legend - 20%
3. Include a table of your estimated parameters as well as the natural frequency and damping ratio - 20%
4. Plot your measured data sampled at 100hz with your fitted equation on top - 20%
5. Based on your results and your computed natural frequency, comment on the Nyquist frequency and the Montalvo frequency of your system and whether or not you encountered aliasing in your experiment based on the figures and the Nyquist/Montalvo frequency - 10%

20 Results and Discussion

The chapters above detailed about 20 projects with multiple subsections that can be used for an instrumentation course. These projects have iterated and been deployed in a real classroom environment since the Fall of 2019. The results and discussion below represent multiple years of analysis and "in the field" iteration of courses. That is, the authors and faculty have used feedback from students to make the course better and better every semester. This chapter here represents the results and discussion of feedback from the students as well as conclusions drawn from faculty and the authors of this text.

20.1 Data Collection and Analysis

Qualitative data collection for this study was done using end of the semester surveys. Course surveys are delivered by the University Office of Institutional Effectiveness. The instructor of the course only receives results of the survey. Eight questions in the survey are restricted to a 1-5 Likert scale while four questions are open ended text responses[28]. The surveys are sent to every student taking the course via email. The surveys are optional and as such not all students completed the survey. The data for the course was compiled for all questions and compared against the Departmental average. This provides a base of comparison for this course across the Department of Mechanical Engineering. It is worth noting that the authors do reflect on how their own positionality may affect interpretation of data. This reflectivity is a core characteristic in qualitative research and is important in establishing credibility and showing transparency. A more in depth analysis of the results are examined in the discussion section.

20.2 Results

The main goal of implementing this survey was to determine the answer to the following question: **What response does implementation of the kit yield from students?** To answer this question, the 4th question in the course survey will be examined. This question states: *"What did you like best about how the instructor taught the course?"* Table 1 includes examples of questionnaire responses which is in the appendix.

Semester	Quotes
Fall 2019	”The course was taught expertly with proper use of in class equipment available and hands on design problems with experiments relative to the material being taught.” ”I like how he introduced us to the Circuit Playground which allowed to be hands-on since none of my other classes are.”
Spring 2020 (Remote Mid-Way)	”Hands on experience” ”I especially enjoyed the hands-on Fridays portion of the course. The practical knowledge of creating something with the book-learning we gained during the lecture was priceless.” ”Fridays were my favorite days. I got to experience hands on projects where I learned the most from.”
Fall 2020 (Hybrid)	”I loved the hands on approach to the course. It is the first course, outside senior design, where we had the opportunity to exercise our engineering knowledge on an actual project.” ”The projects! this [taught] a lot of people hands-on skills”
Spring 2021 (Hybrid)	”I liked the hands on aspect of the class the most” ”I wish all other class were taught as well” ”I like that we actually had a hands on project, and several projects that were fun to do”
Fall 2021	”I love how he tries to engage his students and help them learn real world engineering, not just textbook stuff. He really wants to watch his students succeed inside and outside of the classroom. We need more classes and professors like this in engineering! It was a fantastic class and instructor”
Spring 2022	”I had no interest in coding or robotics like things but now I do. I grew an interest in coding after absolutely hating it. I love seeing the application of what I’ve learned so far in my degree path. I am a hands-on learner, so the labs and the project were my favorite.”
Fall 2022	”The hands on projects and labs” ”Very engaging with the students and cared for each student and wanted to help when needed. Gave good real world applications as well as fun, interesting assignments that helped us students learn.”

Table 1: Questionnaire responses to ”What did you like best about how the instructor taught the course?”

Note that certain semesters were during the Pandemic of COVID-19. In the Spring of 2020 the University transitioned to fully online lectures on March 23rd. That was mid-way through the semester. In Fall 2020 and Spring 2021 the students participated in a hybrid lecture format where students were both allowed to come to class as well as remain remote and log on via Zoom. In the Fall 2021 semester, the University returned to fully in person lectures without masks.

To further answer this question, results from the question “How effective was the instructor in helping students achieve course objectives” is shown in Table 2 also in the appendix. This table shows the response rate per semester as well as average score for the question The last column shows the average score across the department. Spring 2020 has an asterisk because the University of South Alabama created a different course survey due to the onset of COVID-19. Instead the question “Please tell us your perception of the quality of the class prior to the transition to [online] on March 23rd” was used during Spring 2020. As such, the departmental scores are not listed.

Semester	Response Rate	Average Score (1-5)	Department Avg Score (1-5)
Fall 2019	22%	4.20	3.63
Spring 2020 (Remote Mid-Way) *	30%	4.66*	N/A
Fall 2020 (Hybrid)	46%	4.64	3.57
Spring 2021 (Hybrid)	45%	4.75	3.73
Fall 2021	42%	4.70	4.14
Spring 2022	47%	4.94	3.91
Fall 2022	57%	5.0	4.0
Average	41.3%	4.70	3.83

Table 2: Average Score for "Instructor Effectiveness in Helping Students Achieve Course Objectives" (*Alternate question during COVID-19 Pandemic)

From analysis of the surveys, 4 themes emerged which capture student perceptions of the newly developed course. These include: Teaching Style, Course Design & Comparison, Balance between Theory & Application and Connected Learning. Table 3 shows these four themes with definitions of each. These themes overall represent the perspectives students had on the course based on the survey results.

Theme	**Definition**
Teaching Style	Instructor teaching style and personality has an impact on course which is sometimes hard to distinguish from the course design itself
Course Design & Comparison	Student evaluation of different aspects of the course, including the flipped-classroom design, structure of lecture and class time, as well as specifics of the book. Course is compared to others.
Balance between Theory & Application	Students enjoyed the application aspect of the PBL Instrumentation course, but suggested adding more theory to the GitBook provided for the course. Students wanted more harmony between the textbook and the kit.
Connected Learning	Students extend learning outside of the classroom and make connections to their personal life

Table 3: Resultant themes and definitions of themes Theme

20.3 Discussion

This lab at home kit has been implemented since the Fall of 2019. By the time of this writing, the kit has been implemented for 7 semesters, three of which were during COVID-19. The kit was very useful during the COVID-19 pandemic (Spring 2020, Fall 2020 and Spring 2021) because students could order the kit and complete the projects remotely. If students had trouble they could email issues or take a video and share it or even set up a quick video conferencing call. The students were able to see their course objectives hands on which provided a unique way of learning during the pandemic. When the students used the kit in the classroom once the University returned to in person lectures, the students still enjoyed the projects as shown in the survey results.

In regards to the second research question about student satisfaction of the course through the use of the kit, Table 1 clearly show a positive response in regards to the projects and the lab at home kit. For brevity only two to three quotes were shown but for some semesters at least 25 quotes were generated with many of them reflecting on the "hands on learning" component of the course.

Furthermore, Table 2 shows an on average course rating that is 22% higher than the departmental average. It can be said with confidence that the quotes in Table 1 are in direct response to the implementation of the kit and that the students would not have written these words had the kit not been an addition to the course. This proves the first research question that a kit can be implemented as well as answers the second research question.

When these quotes are taken into consideration with the higher course ratings and higher student responses, it is evident that the take-home lab kit promoted more active student engagement in the course and, by extension, more active student learning. This data shows, in conjunction with student commentary, that the projects associated with kits fostered a positive learning experience for students while also covering required content for the course. Projects like this one are a strong strategy professors can add to their repertoire as they teach content. Per Gardner's and Bloom's theories of learning, this would indicate the students likely mastered content and skills in this setting just as well as, if not better than they would in a more traditional lecture format[9, 1].

20.4 Limitations of Findings

Although survey analysis resulted in themes that gave perspectives on the course, there are clear limitations in the resultant findings. The authors are well aware that this data does not conclusively show that implementing a kit in a course increases achievement of course objectives. Many other variable are present in this study, such as the instructor, the course and even the classroom the course is taught in. This study is limited in generalizability based on the small response rate in some of the semesters. Findings are also not generalizable to all engineering Instrumentation courses across the world or even the United States given the influence of the instructor's teaching style and personality in the course. Future research should expand on the study by recruiting individual interview participants and survey respondents to gain a better understanding of the impact of the course on students.

However, findings are useful for instructors seeking to improve their own course by adopting the at-home kit and documentation developed. This theme appeared multiple times among students and it is very possible that students were confused about whether they liked the course or liking the professor. For students, the instructor had a big influence on making the course enjoyable; Preliminary results from this qualitative study can be used to inform course design and teaching practice in future courses.

21 Conclusions

A lab at home kit has been described here. This kit is part of the curriculum for Instrumentation & Experimental Methods and uses the CircuitPlayground Express (CPX) from Adafruit as well as other electronic components. The CPX uses CircuitPython, a derivative of Python. Python like all scripting languages allows students to focus on writing concise code to use the CPX as a tool rather than getting bogged down in syntax, compilation and runtime errors. All software is free, open source and has numerous examples on the Adafruit Learn page as well as the internet at large and Github [23].

As with all other courses in engineering it is possible to teach this course without any hands on material. Instead, a project based learning (PBL) style is utilized where students use the CPX and other components of the lab at home kit to complete guided projects requiring data acquisition, analysis and post-processing all geared towards enhancing learning of course objectives. The final project in the class allows the students to be creative, take ownership, collaborate with others, and use critical thinking to develop something new.

The responses from the students is positive with a 22% increase over the department average over 7 semesters. The qualitative survey responses from students are also positive and show that the kit is effective in enhancing student satisfaction through interaction of tangible objects. Student satisfaction correlates to student performance as shown in the literature. Hopefully, similar kits can be created for other classes at USA as well as other universities in STEM across the world.

22 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.

References

[1] Howard gardner's theory of multiple intelligences. in instructional guide for university faculty and teaching assistants. https://www.niu.edu/citl/resources/guides/instructional-guide, 2020. Northern Illinois University Center for Innovative Teaching and Learning.

[2] R.A. Ralph. Post secondary project-based learning in science, technology, engineering and mathematics. *Journal of Technology and Science Education (JOTSE)*, 6(1):26–35., 2015.

[3] Linda K. Silverman Richard M. Felder. Learning and teaching styles in engineering education. *Engineering Education*, 78(7):674–681, 1988.

[4] Tang Yong. The mode of theoretical knowledge and practical knowledge combination: The significance of internship. *World Journal of Education*, 2(4):55–63., 2012.

[5] J. C. Perrenet, P. A. J. Bouhuijs, and J. G. M. M. Smits. The suitability of problem-based learning for engineering education: Theory and practice. *Teaching in Higher Education*, 5(3):345–358, 2000.

[6] Krajcik Joseph S. and Phyllis C. Blumenfeld. *Project-based learning*. Academia, 2006.

[7] Paul Marshall. Do tangible interfaces enhance learning? In *Proceedings of the 1st International Conference on Tangible and Embedded Interaction*, TEI '07, page 163–170, New York, NY, USA, 2007. Association for Computing Machinery.

[8] BARBARA M. OLDS and RONALD L. MILLER. The effect of a first-year integrated engineering curriculum on graduation rates and student satisfaction: A longitudinal study. *Journal of Engineering Education*, 93(1):23–35, 2004.

[9] P. Armstrong. Bloom's taxonomy. Accessed 2/2/22 https://cft.vanderbilt.edu/guides-sub-pages/blooms-taxonomy/, 2010. Vanderbilt University Center for Teaching.

[10] Lyle D. Feisel and Albert J. Rosa. The role of the laboratory in undergraduate engineering education. *Journal of Engineering Education*, 94(1):121–130, 2005.

[11] Juan P. Oliver and Fiorella Haim. Lab at home: Hardware kits for a digital design lab. *IEEE Transactions on Education*, 52(1):46–51, 2009.

[12] F. G. Martin. Integrating hardware experiences into a computer architecture core course. *Journal of Computer Science College*, 21:39–52, 6 2006.

[13] N. Manjikian and S. Simmons. Evolution and enhancements of a microprocessor systems course. *IEEE Transactions Education*, 42(4):360, 11 1999.

[14] P. Li W. Durfee and D. Waletzko. Take-home lab kits for system dynamics and controls courses. *Proceedings of the American Control Conference*, pages 1319–1322, 2004.

[15] D. Nicoletti D. Cyganski and J. A. Orr. A new introductory electrical engineering curriculum for the first-year student. *IEEE Transactions on Education*, 37(2):171–177, 5 1994.

[16] Sebastian Nanz and Carlo A. Furia. A comparative study of programming languages in rosetta code. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, volume 1, pages 778–788, 2015.

[17] Integrated development environments. https://wiki.python.org/moin/IntegratedDevelopmentEnvironments, 2021. Accessed: 2021-12-09.

[18] Tiobe index. https://www.tiobe.com/tiobe-index/, 2022. Accessed: 2021-3-25.

[19] Hans Fangohr. A comparison of c, matlab, and python as teaching languages in engineering. In Marian Bubak, Geert Dick van Albada, Peter M. A. Sloot, and Jack Dongarra, editors, *Computational Science - ICCS 2004*, pages 1210–1217, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[20] Circuitpython. https://circuitpython.org/, 2021. Accessed: 12-09-2021.

[21] Carlos Montalvo, Lisa Schibelius, and Marine Leabeater. *Project Based Engineering Instrumentation with CircuitPython*. Github, 2023. https://github.com/cmontalvo251/LaTeX/blob/master/PBL_CircuitPython_Instrumentation/main.pdf.

[22] Adafruit. Adafruit learn, 2021. Adafruit https://learn.adafruit.com/.

[23] Carlos Montalvo. Facility for aerial systems and technology github repository. https://github.com/cmontalvo251, 2022.

[24] Adafruit. Adafruit, 2023. Adafruit https://adafruit.com/.

[25] Arduino. Arduino, 2023. https://www.arduino.cc/.

[26] Adafruit. Adafruit ble connect, 2023. https://learn.adafruit.com/bluefruit-le-connect?gclid=CjwKCAiA3KefBhByEiwAi2LDHOVV9IbeAFTEsKBx285M-B7o3VSCeW7uoU0gbDqJ3D0KDX_xnKD-uxoCrrAQAvD_BwE.

[27] Kujtim Mustafa, Ragmi Mustafa, and Refik Ramadani. Measuring the voltage, current and resistance of the ldr sensor through the arduino uno. *Asian Journal of Research in Computer Science*, 16:211–222, 10 2023.

[28] Andrew T. Jebb, Vincent Ng, and Louis Tay. A review of key likert scale development advances: 1995–2019. *Frontiers in Psychology*, 12, 2021.