Autonomous Drone Control System

Thomas Sheehan, Owen Kephart, and Emily Cai
Advised by Allan Moser
Swarthmore College Engineering Department
E90 2018 Report

Abstract

A convenient system to programmatically control an inexpensive drone was created with the intention to explore the concept of Simultaneous Localization and Mapping (SLAM). In addition to modifying the drone controller to receive input from a simple Python program running on a remote laptop, an ultrasonic one-dimensional rangefinder and 360° lidar were added for odometry purposes. Along with the physical modification of the drone, two python application programming interfaces (APIs) were created to simplify sensor data collection and drone control and allow for easier continuation of this project for future students interested in drones.

Table of Contents

Introduction

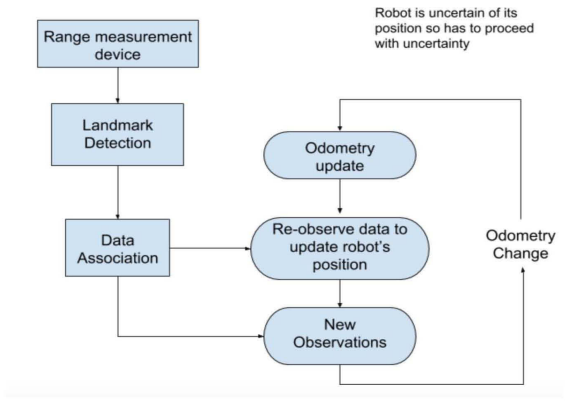
As each of us had significant experience in electrical engineering and computer science, we decided to modify a drone such that it could be of use for future E90 projects. While keeping future users in mind, we created easy-to-use APIs, thorough documentation, and a solid infrastructure for our code. This report will describe our closed-loop control system, our research and reasoning for which sensors and SLAM library we decided on using, how we modified the controller, and how we tested each component of our system. Our measure of success was that we provided a basic drone model for the Swarthmore Engineering Department to use for further experimentation with drones.

Research and Terminology

SLAM

Simultaneous Localization and Mapping or SLAM is the approach to the common robotics problem of mapping an unknown environment while traversing that environment. SLAM is a concept not an algorithm, which is why it's defined as an approach to the problem.

SLAM Concept



The general idea of SLAM is to have a sensor attached to a robot which starts to traverse a space. The sensor will pick up information of the outside world and create landmarks within the map it's building in the landmark detection phase. After this, the robot enters the data association phase which is the loop where the robot moves around the area and builds the map by tracking new landmarks and relates them back to the landmarks it already knows.

For this project, we researched many different SLAM libraries and eventually decided to use BreezySLAM. BreezySLAM is a simple, open-source package that allows us to run SLAM in Python on Linux/Mac OS. The API for BreezySLAM is also very simple and can work for essentially any Lidar. The one issue we ran into while working with BreezySLAM was that even though the Lidar parameters BreezySLAM expects are regularly spaced scans, the Lidar we decided to takes scans at variable rates. We overcame this problem by interpolating the data from our Lidar to give the illusion that our lidar was taking data at a consistent rate.

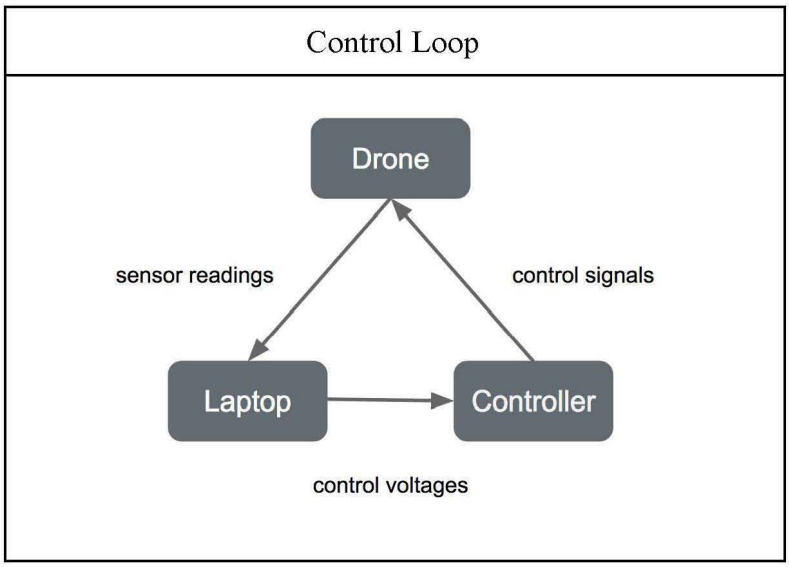
Sensors

Sensors are devices that detect or measure physical properties. In our case, we utilized two sensors: a 1D rangefinder and a LiDAR. The 1D rangefinder is a small device that emits sonar waves that bounce off objects in a singular direction of interest. It measures how far that object is from the sensor which it represents as string value in millimeters. We used this rangefinder as a way to stabilize the altitude of the drone in a room. The LiDAR is a similar detection system which uses light from a laser to observe objects around it. The light bounces off these objects as the sensor continuously rotates 360° and is able to determine the distance, angle, and power of each beam of light it sees as it comes back to the sensor (which relates to the object it bounced off). As described earlier, the scans from the Lidar will be the input for the SLAM program.

APIs

An API is an Application Programming Interface. The purpose of an API is a set of instructions or standards for using an application or tool. The idea is to abstract the underlying complexity of the operations performed under-the-hood of the application with the software and hardware and primarily focus on the job you wish to complete. API's are very convenient for users since they don't need to bother themselves with how exactly and operation works within an application, they just know that if they perform a certain action they should get a certain result.

Closed-Loop Control Design



The basic model of our project is centered around this closed loop between the drone, the laptop and the controller. Each object in our closed loop is responsible for a specific job in order to avoid confusion and create a more clear system. The drone acted as our sensor which only takes in information of the outside world to send to the laptop. The drone is not responsible for understanding the details and implications of the data, its only responsibility is to read and send the data to the laptop. The laptop acts as the brain of the system. The brain's job is to take in data from the outside world and come up with the proper actions in response to the data seen. Similarly, the laptop's job is to take in the information read from the sensors and interpret them in order to come up with the proper commands to send to the controller. Lastly, the controller acts like impulses leaving the brain, causing the body to physically move. In this way, the controller updates the power controls on the drone to correspond to the instructions sent from the laptop. This will actually cause our drone to move in response to the data taken in from its surroundings. This loop runs continuously and is constantly updating thus giving us a fully autonomous system.

Our overarching idea, and the concept we ultimately want to show, is that we can form a 3-way communication relay where the drone sends messages to the laptop which then interprets the messages and relays it to the controller which then ultimately sends the new control signals to the drone in order to have our drone autonomously control itself as it navigates a room. First, we will discuss each of the major system components in a bit more detail.

1. Drone

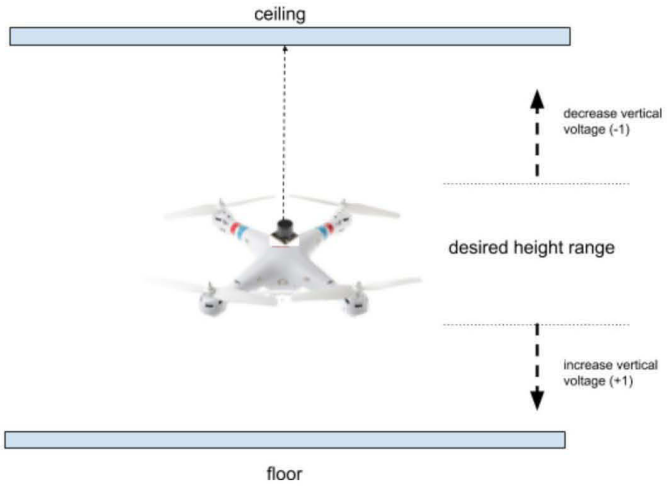
Syma X86 drone with sensors

Height Control

One problem we encountered early in our project was height stabilization. Since the drone we bought was an off-the-shelf drone it was designed to have the user control its height and did not have anything to stabilize it while in flight. This posed a big problem for us since we wanted to mount a LiDAR on top of it in order to get readings from around the room. Thus, in order to stabilize the height of the drone we ordered a one-dimensional sonar rangefinder. The rangefinder measures the distance to objects in the direction its facing, and our idea was to aim the sonar at the ceiling use the distances it measures to update the vertical power of the drone. We chose to aim our sonar at the ceiling since we knew that the ceiling would be a more consistent flat surface than the floor (if there is a box or a bench that the drone accidentally flies over, the height will not be jeopardized).

The algorithm of the height stabilization program was similar to that of a PID controller in that the power is constantly adjusted based on the previous reading. However, a PID controller seemed too complicated for what we needed to do so we simplified the algorithm to increase or decrease the power by one based on the distance read from the rangefinder. Specifically, we wanted to keep our drone within a targeted range between the ground and the ceiling. We would increase the voltage if it fell below the range and decrease the voltage if it went above the specified range. Our plan can be depicted as follows:

Height Stabilization Idea



Following our idea, we created some python code using the Controller API and DataServer API, which we will discuss later. In addition to the logic in this code, we can see how the program immediately changes the vertical on the power as the sonar information was coming in.

Height Stabilization Code

```
import data_server
import controller
import json
import time

c = controller.Controller(json.load(open("../config.json")))
ds = data_server.DataServer(json.load(open("../config.json")))

c.set_vertical(169)

target = 1475
slack = 200

go = raw_input("hit enter to start")
while True:
    time.sleep(.5)
    sonar_reading = ds.get_sonar_data()

    if (sonar_reading < target-slack):
        c.set_vertical(c.get_vertical()-1)
        print "above"

    elif (sonar_reading > target+slack):
        c.set_vertical(c.get_vertical()+1)
        print "below"
```

Our algorithm is simple since we are able to abstract way the idea of updating the controller and read in the sonar messages in live time by creating instances of a Controller object and the DataServer object. Once we have these objects we can just follow the logic we had before by increasing and decreasing the analog value of vertical power by one depending on if the sonar reading was outside of our target range. Specifically in our program our target range was 1475 mm ± 200 mm. Therefore if the sonar reading was below or above this range we would get the current vertical power and increase or decrease it by one then set it. It's important to notice the momentary sleep function call we added to which ensured that we gave the drone time to react so we do not over adjust our power to stay in the specified range.

Mounting:

In order to mount both the sensors and the raspberry pi on the drone, we first needed to get an estimate on the weight at which the drone could bear and still ascend without going full power. We did this by by hanging a weight of batteries that weighed slightly more than the sensors we wished to add to the drone to see if it could still rise. Although it took more power, we found that the drone was able to ascend with the power roughly around ¾ of it's full power. Once we established that the drone was able to bear, we could now move on to mounting our sensors.

Before mounting the sensors we decided to place the raspberry pi at the base of the drone so it could avoid the propellers and allow the wires connected to it from the sensors to fall below the propellers as well. From observing the drone, we knew that the placement of the sensors would be most optimal on the top of the drone. Specifically, The LiDAR would be stationed directly on top of the drone in order to maintain equilibrium in addition to avoiding the propellers on the drone. The sonar would be slightly offset from the LiDAR in order to allow us to still see the ceiling. We stationed the sonar closest to the pins of the raspberry pi underneath in order to ensure not only a solid connection between the two but also to have less complicated wires fraying out of our drone.

We secured the sensors and the raspberry pi to the drone using duct tape in order to get an idea of what it would look like but in the future, we would want to 3D print a mount that would fit our drone and improve the sensors & Raspberry Pi's ease-of-access and secureness.

- format for data, sweepy module to read from lidar, sending it over (potentially different) port to laptop, same basic shape

The drone has two data links associated with it: the control signals which are sent to it from the controller and the sensor readings that it sends to the laptop. Of course, the process of sending control signals from the controller to the drone was already handled for us out of the box, but we did need to create a method for sending data from the drone to the laptop. We used a

Raspberry Pi to interpret the data from the sensors and send it over to the laptop via WiFi, which we will elaborate in the next section.

2. Laptop

As discussed above, we have finished all of the critical connections in the idealized loop of our project. The two links associated with the laptop (getting readings from the drone and sending commands to the controller) are accomplished by two different scripts: `data_server.py` and `controller.py`.

`data_server.py` implements the `DataServer()` class, which contains a multi-threaded server, with one thread for each sensor reading (for now, this is just two threads, one for sonar readings, and the other for LiDAR readings). Each thread essentially acts as its own server, listening on a seperate port, and implementing its own protocol for data acquisition. It is relatively easy to receive data from the sonar, as it is a simple reading that is a maximum of 4 digits long, so we can just tell the server to listen for 4 byte messages, and send an extra space at the end of shorter messages to standardize the length. However, the LiDAR readings contain many different scans which may be of variable length, and so it is necessary to first send the size of the scan from the drone to the laptop in a predictable (4 byte) length, and then have the laptop listen for a message of that size. Because the threads are isolated from each other, it is very easy to implement these different protocols without any interference. This isolation is also valuable in that it avoids issues where the faster sonar range readings would be bottlenecked and unreadable while the slower 2D LiDAR readings were taken. Each can go at its own maximum rate. The multi-threaded architecture does come with some extra challenges, however. In order for us to expose methods that can reliably give the most recent sensor readings, we need to make sure that we don't simultaneously update the data and read it at the same time, as this could corrupt some of the readings. We avoid this issue by locking the threads whenever they need to update or read a specific bit of data, ensuring that those two actions can never occur at the same time.

`controller.py` implements the `Controller()` class, which is much simpler than the above class. It implements get/set methods for each of the motor values. The set functions all implicitly call a private method within the class that standardizes the sending protocol. Namely, after the laptop connects to the arduino over the USB port, it converts the function call into a message of the form "`<pin index> <power value>`". The pin index is a value from 0 to 3 representing the pin controlling up/down, rotation, forward/back, and left/right motion, respectively. The power value is a number from 0 to 255 representing the relative voltage which we wish to supply to this pin. The discretization into 256 potential value is not our construction, this is a limitation of the specific Adafruit Circuit Playground device that we are using, as the `analogWrite()` function that it uses only take values in this range. These values are then automatically translated to a 0-3.3V range which is used by the controller. After it sends this message, it updates its local value of the given motor to reflect this change, which allows it to

remember what the motor value is set to, and effectively enables the get functions for each of the motors.

With these two APIs, a user has all they need to use drone sensor readings to generate drone movement patterns.

3. Controller

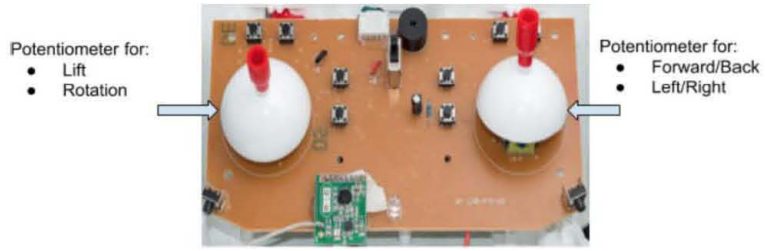
As mentioned above, in order to proceed in reaching our goal of having our drone move autonomously we had to first hack the controller. By hack the controller, we mean to disconnect the controller signals from coming from the joysticks and use signals from a computer instead.

To actually hack the controller, we first had to analyze the controller and understand how it worked fundamentally. We had to make careful observations about how things worked because once we started messing with the connections the effects would be irreversible. Thus our first step was to first test the controls on the outside and observe how the drone acted and then open up the controller and observe how the controls worked under the hood on the PCB board.

The Controller for Syma x8c

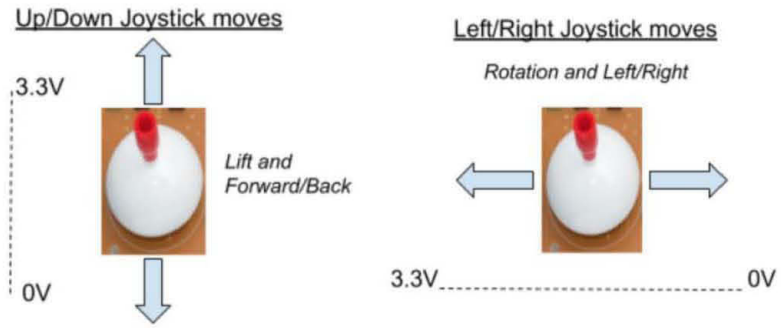
In observing the controller we found that the controller essentially controlled four directions of movement using two joysticks: up/down, left/right, forward/backward, and rotateLeft/rotateRight. To control these movements it used two joysticks that would relay signals based on the direction we pushed the stick. We found that the left joystick was responsible for both the up/down control signals and the rotate-left/rotate-right signals while the right joystick was responsible for the lateral movements with forward/back and left/right. In addition to the joystick, the other essential piece to the controller is the antennae which actually relays the signals to the drone.

Joystick responsibilities



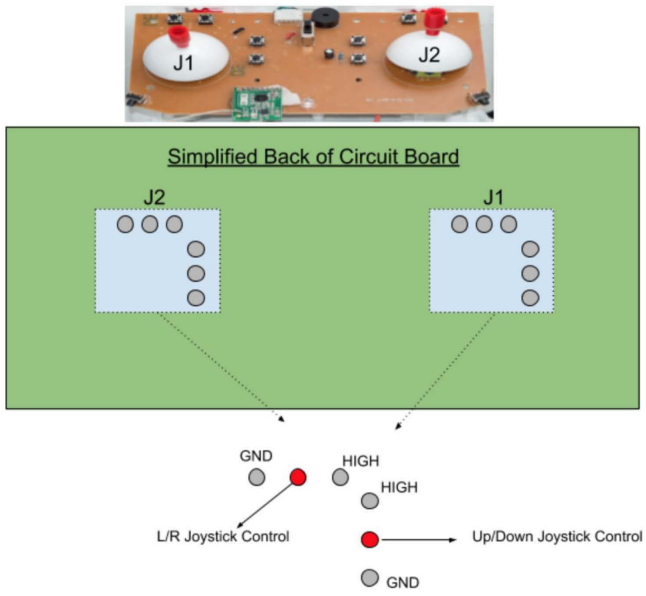
Once we had a basic understanding of the controls on the surface, we had to open up the controller and observe what physically was going underneath. We found that the joysticks were essentially 2-way potentiometers that had a range from 0.0V to 3.3V depending on the way it is oriented. By testing each possible orientation of the joysticks and using a multimeter we found the voltages to change according to this diagram:

Joystick Controller Voltages



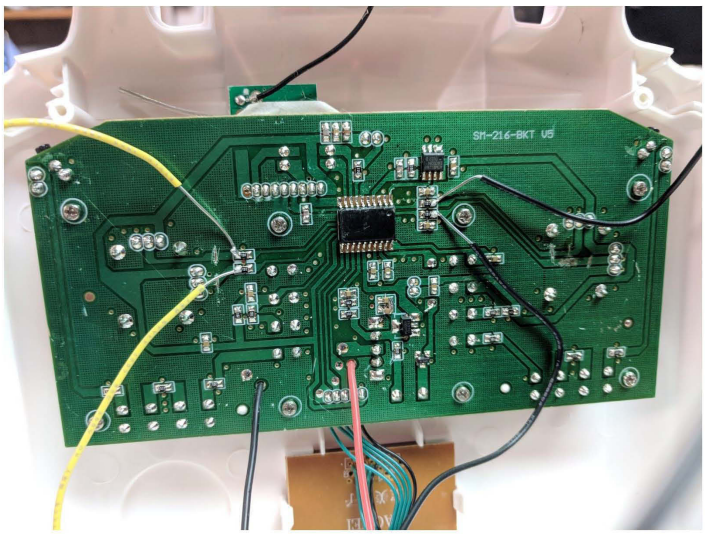
Therefore, we knew that if we could send voltages ranging from 0V to 3.3V from the computer we could replicate the signals coming from the joysticks. Our goal was to eliminate the joystick signals so in order to completely bypass these signals we needed to cut the connection. To do this, we need to observe the back of the PCB board which contained all the internal circuitry of the controller. Opening up the controller we found that among all the circuitry and solder points that we were only concerned with four specific points. The simplified view of the PCB board points we wished to change looked like:

Important Points on Circuit Board



By observing these 4 points, we found out that if we could sever the copper connection coming from each of these points and render the joysticks useless. In doing so, we could solder on our own wires to the joystick signal receivers and be able to send voltages to them via a microcontroller guided by a computer program.

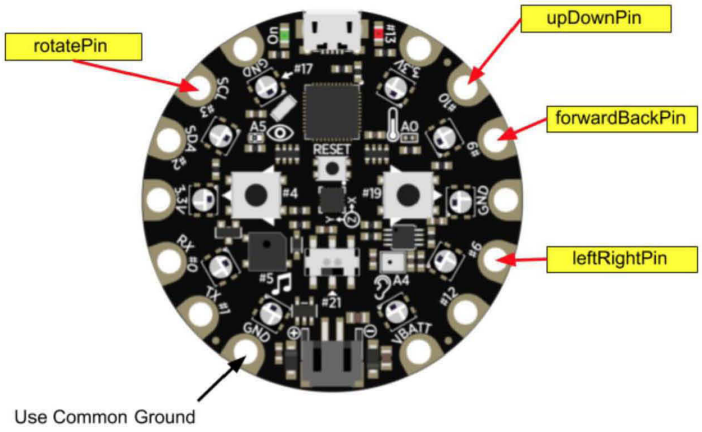
Bypassing the Joystick Controls



Once the connections were severed and the wires were soldered on the next step was to decide on the microcontroller to deliver the voltages to the controller. In making our decision, we knew we wanted a controller that could send voltages up to 3.3V and a controller that was able to be coded to change these voltages based on our input. The controller we decided to use was the Adafruit Circuit Playground, a microcontroller which can output a max of 3.3V and can be coded in Arduino C.

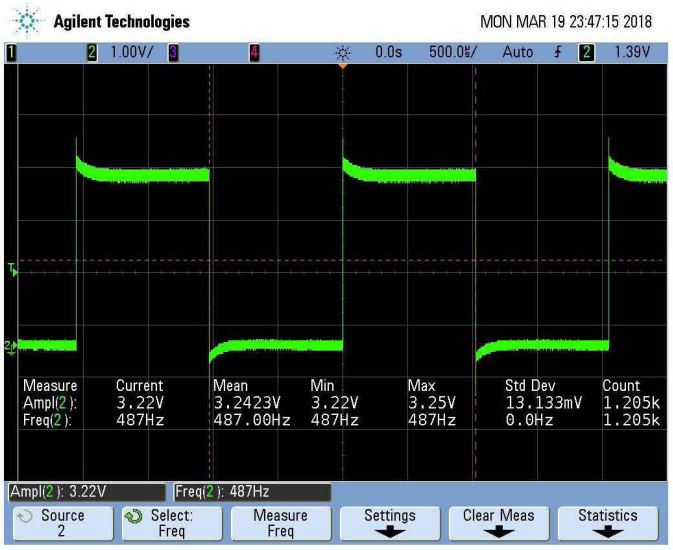
The Adafruit Circuit Playground has four GPIO output pins that we could write voltages to by sending analog values. Using the Arduino library we could analogWrite values to pinouts sending values ranging from 0-255 which equated to voltages ranging from 0-3.3V. We assigned D3, D6, D9, and D10 pins to be the output for the voltages being sent to the controller for all four actions. In addition to this we also used to the ground pin on the Adafruit Circuit Playground to ensure our circuit had a common ground.

Adafruit Circuit Playground Pin Assignments



An unforeseen dilemma that arose from using the Adafruit Circuit Playground GPIO pins though was that the voltages that were being sent were using Pulse Width Modulation.

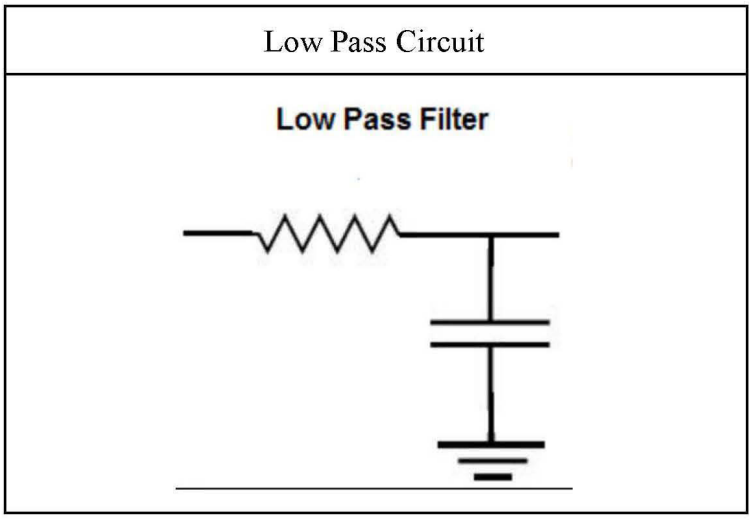
Signal to Controller PWM



Pulse Width Modulation is a signal that is sent through pulses instead of using a steady signal. The problem here is that the drone expected a solid steady voltage and the PWM signals were causing audible variance in the motors causing it's flight and control to vary significantly as we tried to change voltages.

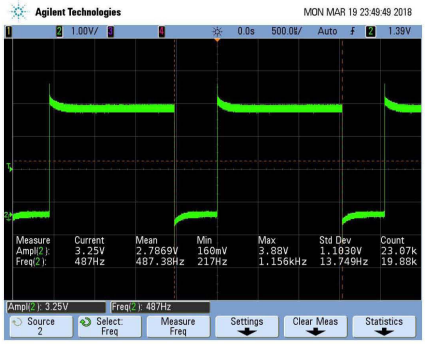
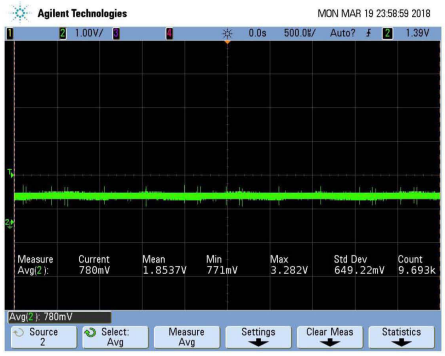
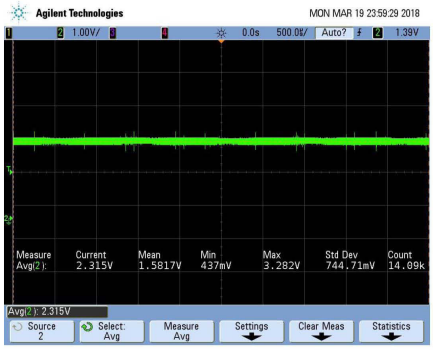
Therefore we needed to find a way to stabilize the signal and send a steady voltage to the controller. In order to do this, we decided to take the output signals through a low pass RC filters to smooth out the signal.

Low Pass Circuit



Using the oscilloscope, we found that the PWM signals coming from the circuit playground where approximately 490 Hz. Therefore, we needed to have a cutoff frequency of our circuit that would be less than that. We ultimately decided on using the resistor and capacitor values, 5.1 kΩ and 4.7 µF respectively. By using these values we developed 4 low pass circuits with a cutoff frequency of 6.6 Hz so we know it was guaranteed to work in our case.

After applying these low pass filters, we were able to get a smooth signal to go to the controller and thus rid ourselves of the the variance in the motors.

Before	After
 	 

Once we were able to successfully bypass the joystick signals and send steady voltages to the controller via the computer, it was time to develop C code to guide the drone from user input. We decided to take a two step approach to designing the code for the Adafruit Circuit Playground. First, we wished to create a code that was keyboard driven that would allow us to control how the drone would move. Then once we were confident in the controls, we created a code that would listen for measurements so that it would be guided from a sensor instead thus making it more autonomous.

As mentioned before, we started with the keyboard driven Arduino Code first. For the arduino code it's essentially split up into two sections: a setup section which is only called once on the startup and the loop section which is constantly running while the arduino is in use.

Arduino Code Void Setup - Input over Serial

```
// controller-specific constants
#define upDownPin 10
#define rotatePin 3
#define forwardBackPin 9
#define leftRightPin 6

const int pins[4] = {upDownPin, rotatePin, forwardBackPin, leftRightPin};
int powers[4];
int i;

void reset() {
  powers[0] = 0;
  powers[1] = 127;
  powers[2] = 127;
  powers[3] = 127;

  for (i = 0; i < 4; i++){
    analogWrite(pins[i], powers[i]);
  }
}

void setup() {
  // basic setup
  CircuitPlayground.begin();
  Serial.begin(9600);

  // make sure all pins are output pins
  for (i = 0; i < 4; i++){
    pinMode(pins[i], OUTPUT);
  }
  analogWrite(pins[0],0);
  delay(1000);
  analogWrite(pins[0],255);
  delay(1000);
  analogWrite(pins[0],0);
```

The set up of our Arduino Code is responsible for initializing the controller to communicate with the drone by both performing the startup sequence required to control the drone and setting up the proper pins and powers on the Adafruit Circuit Playground to be able to send voltages or control signals to the drone.

For our code, we start with defining two integer arrays both of size fur to correspond to the four possible control actions the drone can be moved in. We define a pin number array which is responsible for keeping track of which pin location we are changing and we define a power array of the same size which acts as holder for the powers of the pin in the corresponding index in the pin number array. By having both of these arrays, we can update the controller signals by only changing the powers array and writing out the analog voltages to the controller using the pin number in the corresponding index of the index in the power array.

Once we have these two arrays setup, we then initialize the values according to the values we measured the to be at the start originally. It's important to initialize these values in the setup stage to avoid "garbage" values being set at the start which could harm the drone by giving values that it's not permitted to have. Once these values are set and sent to the controller via

analogWrite(), we have to go through the startup sequence to actually connect to the drone. The startup sequence physically is flickering up the vertical power joystick once. This translates to the code setting the up/down power pin to be 0, then 255, and finally 0 again with slight pauses in between in order to allow the controller to react. In doing all of this in the setup we have succeeded in initializing the controller to be ready to give signals to the drone.

Arduino Code Void Loop - Input over Serial

```
// commands are formatted as:
// [ pin ] [ power ]
// if the pin is not a number between 0 and 4, or the power is not between 0 and
// 256, then the reset() function will automatically be called
int cmdpin;
int cmdpower;
String cmd;
void loop() {
  if (Serial.available() > 0) {
    cmd = Serial.readStringUntil('\n');
    int ret = sscanf(cmd.c_str(), "%d %d", &cmdpin, &cmdpower);
    if (cmdpin < 0 || cmdpin > 3 || cmdpower < 0 || cmdpower > 255) {
        reset();
    } else {
        analogWrite(pins[cmdpin], cmdpower);
    }
  }
}
```

Once we initialized our controller and synced it to the drone we were able to pass in inputs via a python program that would tell us which pins to change and to what analog value by our void loop. As mentioned before, the void loop is constantly running for the entire time that the Adafruit Circuit Playground is powered thus we can treat it as a infinite loop. Therefore the job of our void loop is to constantly check if there were any messages passed to it via usb and if there were then it had to act on the command. In order to perform this check we use the Serial.readStringUntil() function which reads the string passed serially from the laptop and stores everything up to the space as a string. Once we have this string command saved, we want to turn the two integers passed to it to actually become integer values within the code. Therefore, we convert these two string numbers to integers using the sscanf() function.

We made the assumption that everything being read serially would be in the form "%d %d" which means that it would be two integers separated by a space. Specifically, the form will come in the form of "pin # and pin power" to signify exactly which pin we want to change and to what power. We could make this assumption because this is the form we decided on sending information from in the Controller API that we designed and we know that nothing else should get sent across. Thus, once we get these two integers out of the string format we need to actually set the controller values to be the new updated values. In order to do this, its just a simple update of the power array at the pin number index passed in and then a analogWrite() to actually send it

to the controller. However, we took a defensive programming approach to setting the controller voltages and wanted to check if the information passed in from the user was actually a valid action. We do this by checking to see if the pin number was valid (0-3) and if the power set was a valid power (0-255). If any of these is not true then we avoid setting the controller and reset the controller to its original values to avoid harming the drone. Once the controller completed one action it would then run indefinitely waiting for commands to come in and setting the controller accordingly. This loop allowed us to use our Controller API to pass messages from the laptop to the controller and utilize the controller-drone connection previously established to control the drone.

Conclusions & Future Work

While we were not able to get a complete implementation of an autonomous drone working, we still look at this project as a success. We were able to successfully transmit signals from the drone to the laptop and finally to the controller completing the basic loop we set out to make. Although some of the commands did not work out perfectly due to physical properties of the drone such as balance, we viewed this as an excellent proof of concept which reaffirmed the success of our project.

As we mentioned before, potential future work would include mounting the lidar with 3D printed mount and using the SLAM concept to allow the drone to map the room as it traveled around it. The APIs we created and the closed-loop control project we created for this project can be found on GitHub and can be shared with the engineering department for future use. Future groups could also modify the drone and create their own code to find new solutions to the height control problem and have the opportunity to use a different SLAM approach to attack the problem of traversing an unknown environment while also keeping track of the growing map of that environment.

Acknowledgements

We would like to acknowledge Professor Moser for meeting with us weekly and advising us. We would also like to acknowledge Ed Jaoudi for helping us acquire the necessary materials for our project. We would also like to acknowledge Cassy for helping us find a place to work on our project. We also want to acknowledge Professor Cheever and Doug Wilen for sharing their knowledge and resources with us.