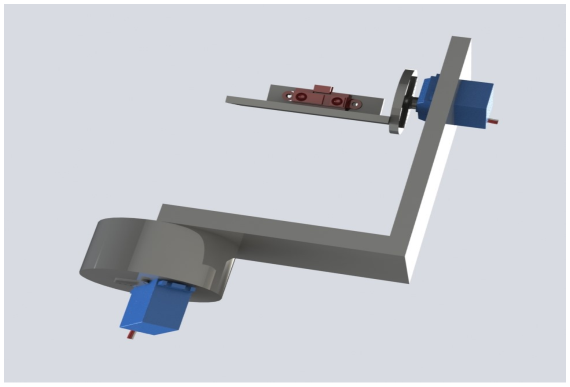
POE Lab 2: DIY 3D Scanner

Kristen Behrakis, Victoria McDermott

September 22, 2017

1 Description

The goal of this lab was to build a 3D scanner using a simple pan/tilt mechanism. We used our 3D scanner to scan the letter "V" and reproduce a computer-model for visualization. This is made possible by an infrared distance sensor, which uses reflected light off an object to find distance, and two hobby servo motors, which physically pan and tilt the distance sensor. To connect the two servos and the distance sensor, we designed and laser cut a mechanical connection device using SolidWorks (see Figure 1).


(a) SolidWorks model (blue portions are servos, red portion is distance sensor).

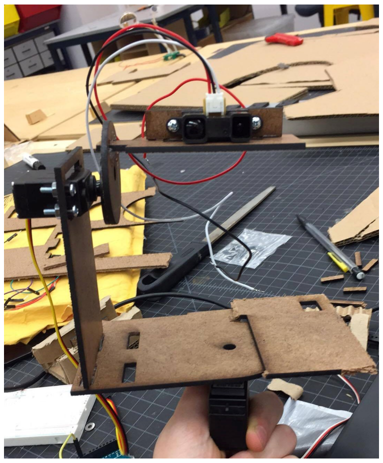

(b) Final device.

Figure 1: Device used to mechanically mount servos and distance sensor.

2 The Procedure

2.1 Understanding Component Functionality with Arduino IDE Examples

2.1.1 AnalogInput: Connecting a sensor to the Arduino

First, we needed to ensure that we could properly receive data from an analog input sensor. The example code provided used a potentiometer to turn an LED on and off. When we set up the circuit (as described in the example documentation), we noticed that adjusting the potentiometer affected the amount of time the light stayed on or off. When we examined the code more closely, we noticed that the delay was directly connected to the sensor value. This indicated that we were correctly reading input from the analog device.

```
// read the value from the sensor(aka potentiometer):
sensorValue = analogRead(sensorPin);
// turn the ledPin on
digitalWrite(ledPin, HIGH);
// stop the program for <sensorValue> milliseconds:
delay(sensorValue);
// turn the ledPin off:
digitalWrite(ledPin, LOW);
// stop the program for for <sensorValue> milliseconds:
delay(sensorValue);
```

In last week's lab, we used a distance sensor in a similar manner. We compared our circuit from last week to the example circuit, and this further solidified that we had accepted the input correctly.

2.1.2 AnalogInOutSerial: Relaying and interpreting sensor data

Next, we wanted to see if we could properly relay and use the information received from the sensor. The AnalogInOutSerial example reads an input pin and uses the result to set the pulse width modulation for an LED. By looking at this code, we learned that the "map" function is used to take the input from the analog pin and limit it to a given range. In particular, the LED was given an output of 0 to 255 while the potentiometer was held between 0 and 1023. This was key information for us, as it indicated how we can control the range of the data outputted.

```
// read the analog in value:
sensorValue = analogRead(analogInPin);
// map it to the range of the analog out:
outputValue = map(sensorValue, 0, 1023, 0, 255);
// change the analog out value:
analogWrite(analogOutPin, outputValue);

// print the results to the serial monitor:
Serial.print("sensor␣=␣");
Serial.print(sensorValue);
Serial.print("\t␣output␣=␣");
Serial.println(outputValue);

// wait 2 milliseconds before the next loop
```

```
  // for the analog-to-digital converter to settle
  // after the last reading:
  delay(2);
```

Additionally, sample sketch and Python code were provided in the assignment. The Python code prompts the user to choose whether to read a potentiometer or a button. This is also an important example, as it shows how to collect information in Python and send it to Arduino code for processing. When we tried to use the Python code, it did not work immediately. One thing that we did not realize was that we needed to set the "Serial" to our port on our computers (as the Arduino wouldn't be recognized otherwise). Although we solved this problem, we still weren't able to run the code properly. After some research on our error messages, we found that we were not running the correct version of Python. Once the correct installment was finished, we were able to run the python code, indicate which device we wanted to be read, and verify that the output on the Arduino was going to the correct device.

2.1.3 Sweep: Functionality of Servos

As we had never programmed with servos before, we decided to test our servos with the Sweep example. By looking at the code, we noticed that we would need to keep track of the servo positions (which, in the example, is set between 0 and 180 degrees). This would be helpful for us later, when we specified how far to tilt and rotate our servos attached to our distance sensor. When we ran the provided script, we noticed that our servo activated and rotated quite slowly (exactly as expected given the delays and small movements of 1 degree).

```
// goes from 0 degrees to 180 degrees
  for (pos = 0; pos <= 180; pos += 1) {
    // in steps of 1 degree
    // tell servo to go to position in variable 'pos'
    myservo.write(pos);
    // waits 15ms for the servo to reach the position
    delay(15);
  }
  // goes from 180 degrees to 0 degrees
  for (pos = 180; pos >= 0; pos -= 1) {
  // tell servo to go to position in variable 'pos'
    myservo.write(pos);
    // waits 15ms for the servo to reach the position
    delay(15);
  }
```

2.2 Setting Up the Distance Sensor

2.2.1 Calibration

Now that we had a solid understanding of how the components work, we needed to calibrate our distance sensor. When we used the distance sensor during the last lab, we did not need to calibrate the sensor, as the distances were all relative. For this lab, however, it is critical for our final visualization graphic that the sensor is calibrated to measure distances as accurately as possible. To calibrate the sensor, we recorded sensor readings for several known distances. To do this, we used a black board and set it a certain distance away from the distance sensor. Then, we would print out the corresponding sensor values in the serial monitor. For each distance, we observed the values coming out, took note of the range of values, and recorded the average sensor value. We also did this process twice to make sure that our sensor values were consistent.

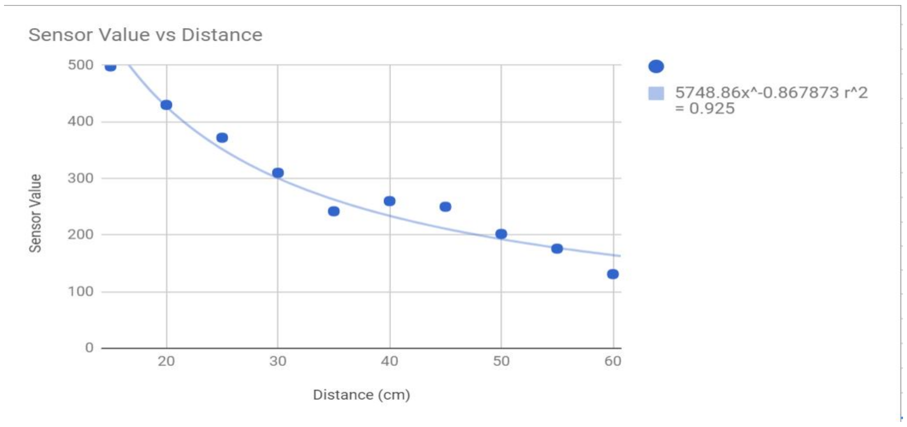
We then converted the sensor values to Voltage, as was specified in the assignment. To do this, we utilized the AnalogInOutSerial file in the Arduino IDE examples (but instead of the analog input being a potentiometer, we used the distance sensor). To find the output voltage, we started with our original input voltage (5 Volts) and divided it by 1024. The divisor was chosen because when we analyzed the code, we saw that the sensorValue would fall between 0 and 1023 (leading to 1024 options). Multiplying the provided sensor value by the 5/1024 (our conversion factor or the value per step), results in the voltage output. We also referenced an article with a similar goal to help in this conclusion (http://luckylarry.co.uk/arduino-projects/arduino-using-a-sharp-ir-sensor-for-distance-calculation/).

The results are shown in Table 1.

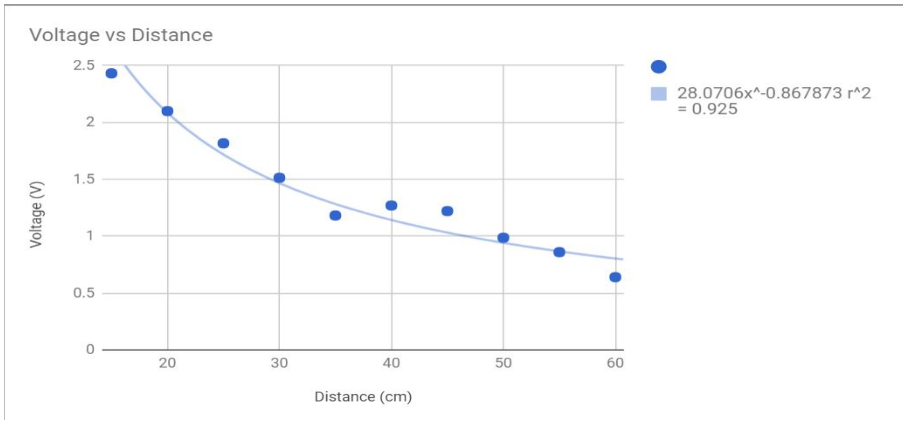
Distance Sensor Value for Varying Distances		
Distance (cm)	Sensor Output	Sensor Voltage (V)
15 cm	498	2.43
20 cm	430	2.10
25 cm	372	1.82
30 cm	310	1.51
35 cm	242	1.18
40 cm	260	1.27
45 cm	250	1.22
50 cm	202	0.99
55 cm	176	0.86
60 cm	131	0.64

Table 1: Sensor Values for varying distances.

Based on the sensor output and corresponding distances, we created a scatterplot and found a line of best fit. After reasding some similar papers regarding these distance sensors (as well as qualitatively finding the fit), we decided to choose a power series fit. This has an R-squared value of over 0.9, which also indicated that this would be a good fit. Our equation and plot are shown below in Figure 2.



(a) Line of best fit found for our measured distances for servo value outputs.



(b) Line of best fit found for our measured distances for voltage outputs.

Figure 2: Lines of best fit.

2.2.2 Testing Accuracy

Next, we needed to test the accuracy of our calibration function. To do this, we used our function to predict known distances (see Table 2). Based off of our results, the average error was about 8.1%. The values appear to have significantly less error as the distances increases, which could be due to the sensor minimum distance being around 20cm. Our greatest error occurred at 18cm, and without this outlier, the average error would be 5.6 %. A plot of this data (with corresponding error bars) can be seen in Figure 3.

We also compared our calculated voltage to the expected voltage from the datasheet. This comparison is shown in Figure 4. We found that the average voltage difference is 0.36 V, with the most significant discrepancy occurring at 35 cm (about a 0.52 V difference). It appears that our measured values fit best between 40 and 50 cm.

Actual and Predicted Distances		
Actual Distance (cm)	Predicted Distance (cm)	Error (%)
18	23	27.8
23	25	8.7
27	31	14.8
32	34	6.3
38	40	5.3
42	42	0
48	51	6.3
53	52	1.9
58	59	1.7

Table 2: Actual distances and predicted distances found from our calibration function. The average error 8.1 %.

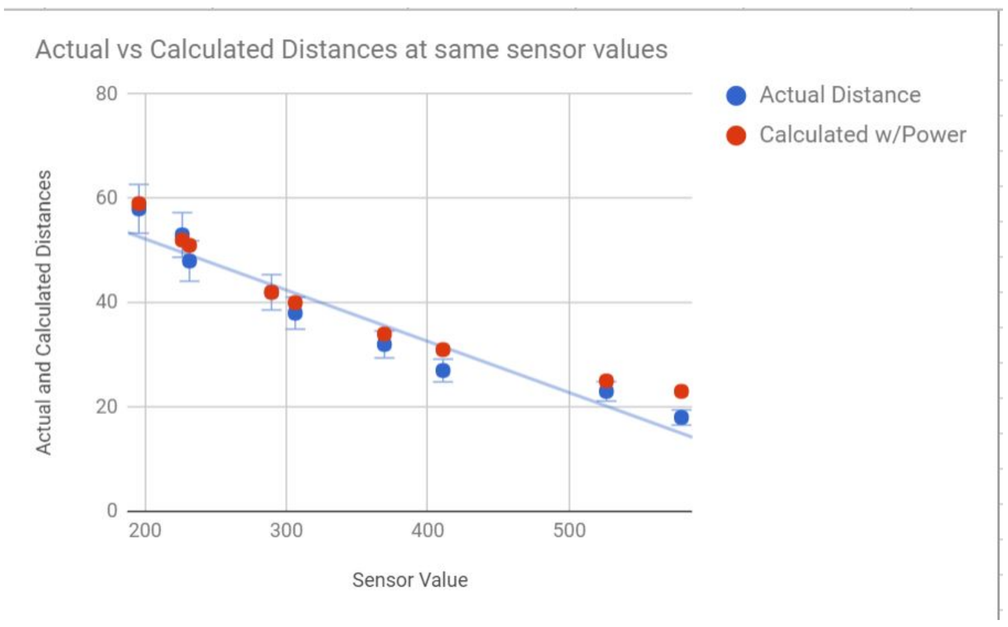


Figure 3: Actual vs Calculated Distances. The error bars on the graph indicate the average error of 8.1%.

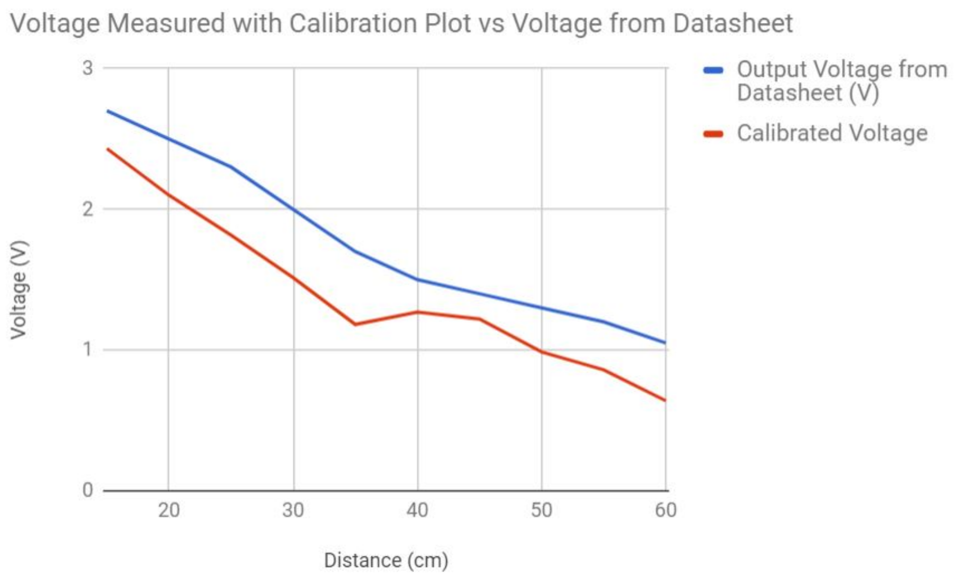
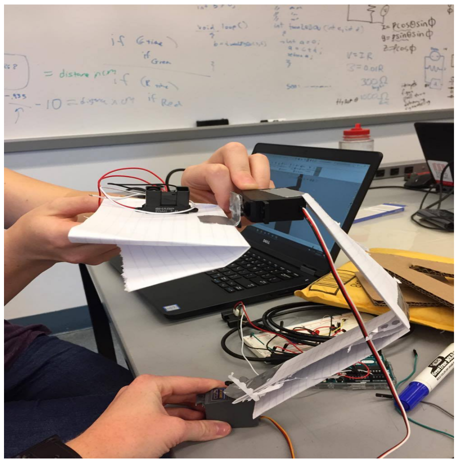


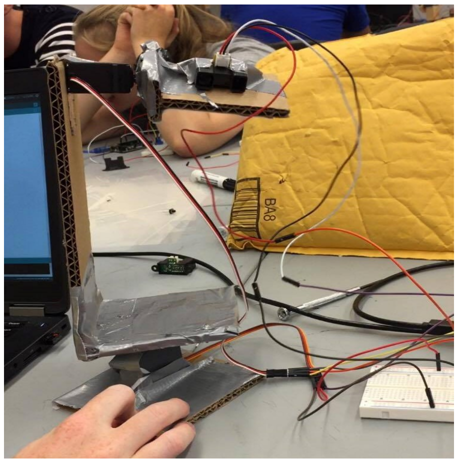
Figure 4: Calibrated Voltage vs Expected Datasheet Voltage. The average error in the voltage is approximately 22 %.

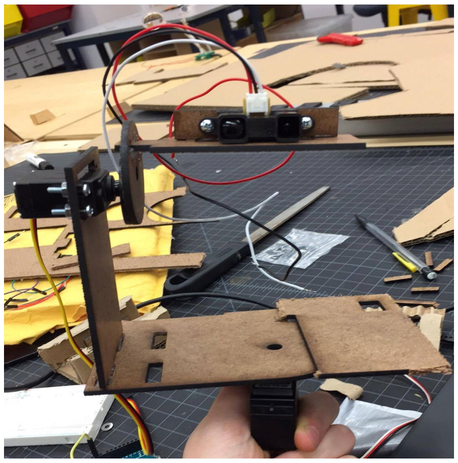
2.3 Setting up the Mechanical Mount

After our sensor was properly calibrated, we made a mount to connect our two servos to the distance sensor. After looking on the internet for some inspiration, we came across a pan/tilt mechanism that we believed we could edit and reconstruct to fit our needs (see source here). One important aspect in the construction of this mount is that the sensor needed to be centered above the movement points (i.e. centered above the servo). This increases stability and ensures our angle measurments are accurate.

We began by constructing a rudimentary paper-model to better visualize the final setup. Then, we made a fully functioning prototype out of cardboard (see a video of it here). Finally, we designed and laser-cut our final piece. See Figure 5 for images from each stage.


(a) Stage 1: A simple paper model.


(b) Stage 2: A functioning cardboard model


(c) Stage 3: Final laser-cut model.

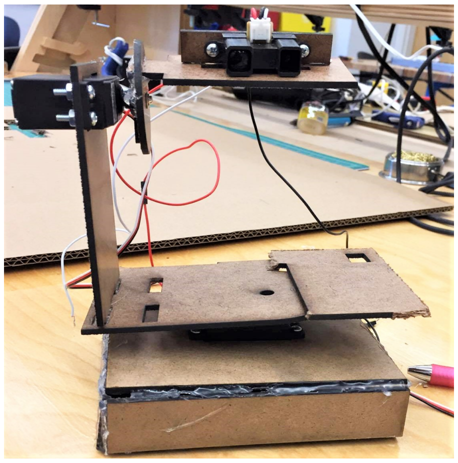
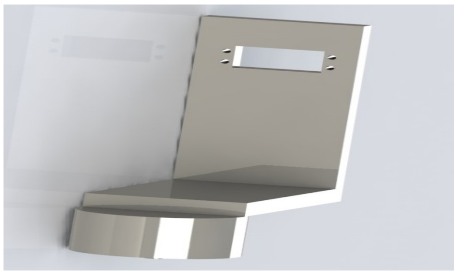

(d) Stage 4: Final laser-cut model with stability base.

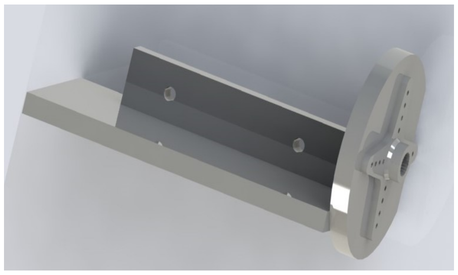
Figure 5: Stages of developing the mechanical mount.

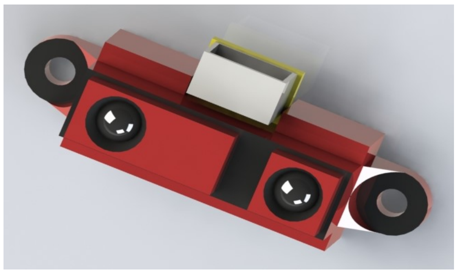
2.3.1 Problems Encountered

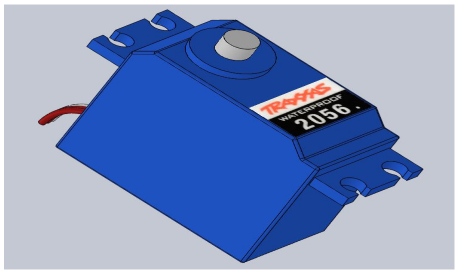
Unfortunately, we ran into quite a few issues before reaching our final model. One issue we had arose in our cardboard model. One of the servos, which was taped into our model (and therefore was important for the whole structure) was causing our Arduino to short. When the power supply and ground were both plugged in at the same time, the LED on the Arduino would not turn on. After talking with some ninjas, we deduced that there was an internal issue with the servo (possibly fried). We then had to redo our cardboard model with a new servo, which ended up working perfectly.

The biggest problem we encountered involved 3D printing our final model. Originally, we had planned on 3D printing our model (see Figure 6), but after 3 failed print-jobs, we decided that this was no longer a viable option. It tended to fail within 30 minutes, which did not bode well for us because the entire print was expected to take 5 and a half hours to complete (see Figure 7). Eventually, we decided to edit our SolidWorks pieces so that we could laser cut our final model.


(a) SolidWorks model of the main body of the mount. This connects one servo to the second servo.

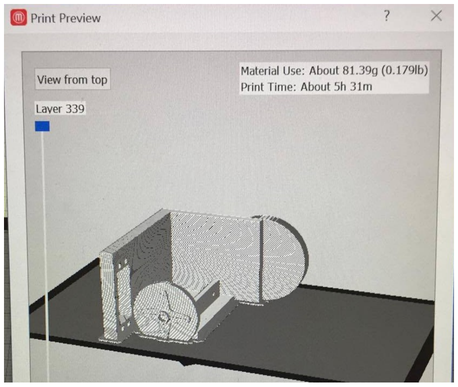

(b) SolidWorks model of the top potion of the mount. This connects the distance sensor to the second servo.


(c) SolidWorks model of the distance sensor. Obtained from GrabCad.



(d) SolidWorks model of the servo Obtained from GrabCad.

Figure 6: Individual components of the CAD from Figure 1.

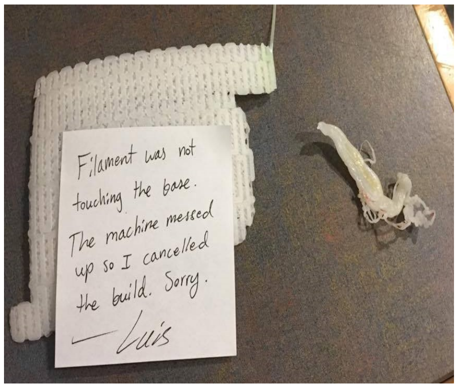

Print Preview

View from top

Layer 339

Material Use: About 81.39g (0.179lb)
Print Time: About 5h 31m

(a) Print was meant to take 5.5 hours, which is a very long time considering the prints failed each time after 30 minutes.


Filament was not touching the base. The machine messed up so I cancelled the build. Sorry.
— Luis

(b) A note left on our print after a failed attempt.

Figure 7: Problems encountered when trying to 3D print.

2.4 Top-down Representation Using 1 Servo

For the top-down representation, we moved our bottom servo to scan a straight horizontal line across our letter. Because our letter is a "V," we thought we would get the most information from scanning horizontally (so we could see the gap in the middle of the V). The basic setup is shown in Figure 8. In our setup, we placed our letter about 40 cm away from the setup, as we found that this was one of the most accurate distances during our calibration (see Table 2).

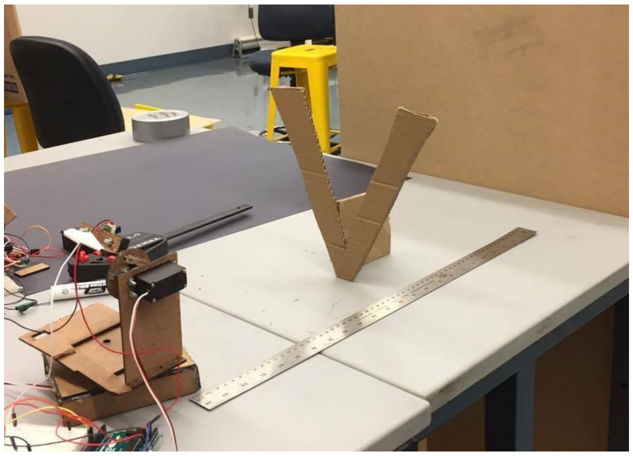
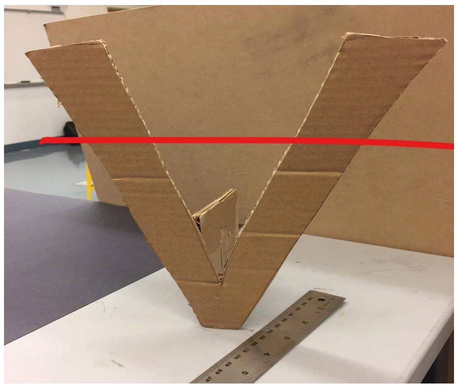


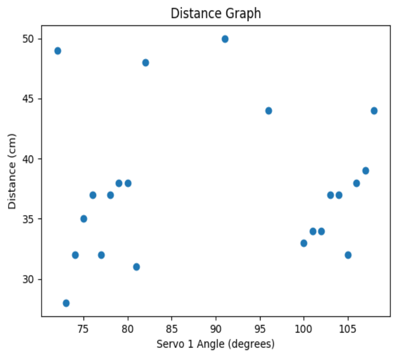
Figure 8: Whole setup for scanning the letter. The same setup was used for both the top-down representation and the full 3d representation.

For the top down representation we only moved one servo to sweep horizontally across the letter 'V'. We filtered distances greater than 50 cm out of our results list completely and put values less than 50 cm into our y list with their corresponding degree position of the servo at that distance. We then used this list of x and y values to generate a graph which shows the servo 1 angle versus the distance the sensor is reading in centimeters.

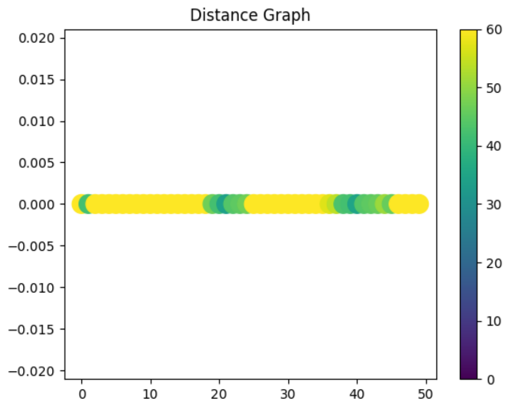
```
for i, my_result in enumerate(results_list):
    if my_result[2] <= 50:
        '''2d version'''
        xval = my_result[0]
        yval = my_result[2]

        xs.append(xval)
        ys.append(yval)

fig = plt.figure()

plt.scatter(xs, ys)
plt.xlabel('Servo 1 Angle (degrees)')
plt.ylabel('Distance (cm)')

plt.title('Distance Graph')
plt.savefig("graph.png")
plt.show()
}
```

A shown in Figure 9, our Python visualization shows a cluster of points on the left and the right with a visible gap in the middle. This is as expected for the letter 'V,' and the cluster of points even appear to slant consistent with the letter V (especially the cluster on the right). There are, however, a few points that are inconsistent with our letter. These point are found between the two clusters and could be due to inconsistency with our sensor or movements/distractions that could have swayed the results.


(a) Cardboard letter 'V' that was scanned. The red is the portion of the letter that was scanned.



(b) Visual representation of the 'V' using Python.



(c) Visual representation of the 'V' using a heat map. The yellow values indicate the letter is closer (and there is a visible gap between yellow portions, indicating a V-shape)

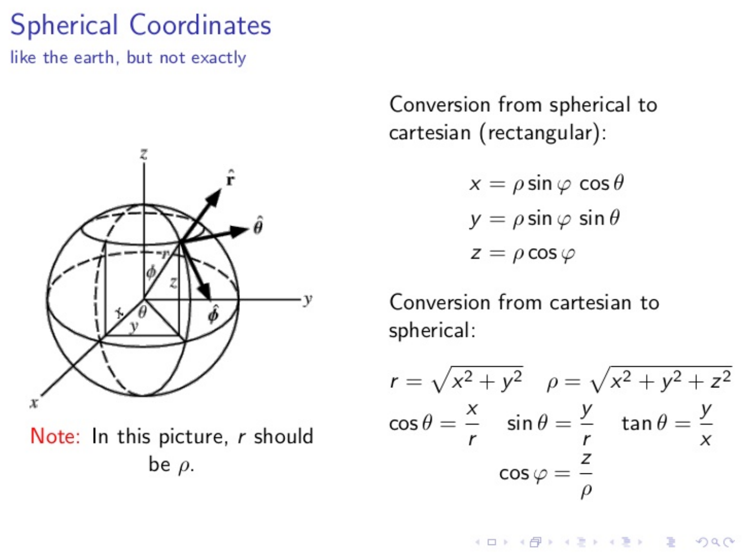
Figure 9: The top-down representation of the actual vs scanned letter 'V.'

2.5 Full Pan/Tilt Mechanism with 2 Servos

For the full pan/tilt mechanism, we used the identical setup as in Figure 8 except that both servos were being utilized. The scanning mechanism worked by moving our sensor horizontally by a small amount, completely scanning the vertical component for that section, and moving horizontally again. We tried our pan/tilt mechanism on two different letters, as shown in Figure 11. We used the following equations to convert from create our visualization (where θ is our bottom servo moving left and right and ϕ is our top servo panning up and down):

$$x = distance * cos(\theta)sin(\phi) \tag{1}$$
$$y = distance * sin(\theta)sin(\phi) \tag{2}$$
$$z = distance * cos(\phi) \tag{3}$$
$$\tag{4}$$

Figure 10 below shows a solid representation of how the spherical coordinates correspond to our measured distances and angles.



Spherical Coordinates

like the earth, but not exactly

Note: In this picture, r should be ρ.

Conversion from spherical to cartesian (rectangular):

$$x = \rho \sin\varphi \cos\theta$$
$$y = \rho \sin\varphi \sin\theta$$
$$z = \rho \cos\varphi$$

Conversion from cartesian to spherical:

$$r = \sqrt{x^2 + y^2} \qquad \rho = \sqrt{x^2 + y^2 + z^2}$$
$$\cos\theta = \frac{x}{r} \qquad \sin\theta = \frac{y}{r} \qquad \tan\theta = \frac{y}{x}$$
$$\cos\varphi = \frac{z}{\rho}$$

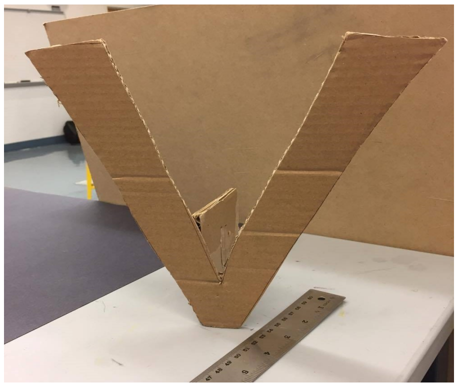
Figure 10: Image we used to better understand conversions between spherical and Cartesian coordinates.

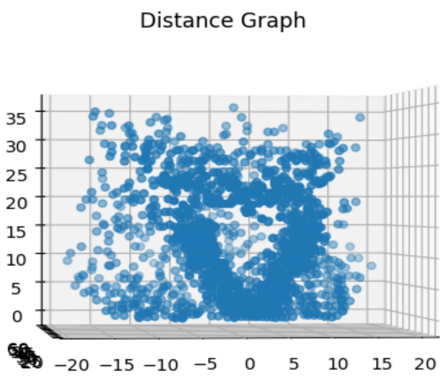
For the Cartesian coordinates for our full pan tilt mechanism we decided to convert from the spherical coordinates we had to Cartesian (see above equations). We used one of the servo angles as our theta, one as our phi, and the distance being read in from our distance sensor as the radius of the sphere. With these three values in mind we were able to calculate an x, y, and z value. We added these values to x, y, and z lists then created a 3d scatter plot including points where our distance values are less than 50 cm because our sensor is not accurate for values greater than about 60 cm and our letter was only about 40 cm away from our sensor so it is fine to filter out these values.

```
for i, my_result in enumerate(results_list):
    if my_result[2] <= 50:
        '''3d stuff'''
        xval = my_result[2]*math.cos(math.radians(my_result[0]))*
        math.sin(math.radians(my_result[1]))
        yval = my_result[2]*math.sin(math.radians(my_result[0]))*
        math.sin(math.radians(my_result[1]))
        zval = my_result[2]*math.cos(math.radians(my_result[1]))

        xs.append(xval)
        ys.append(yval)
        zs.append(zval)

fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')

ax.scatter(xs, ys, zs)
```

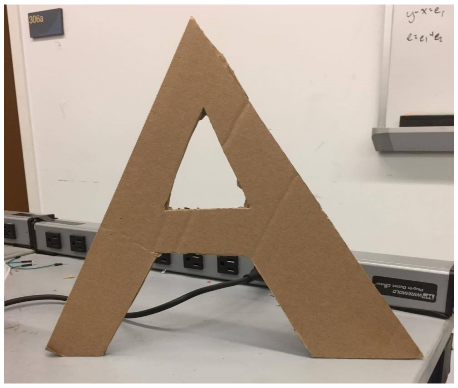
```
plt.title('Distance␣Graph')
plt.savefig("graph.png")
plt.show()
}
```

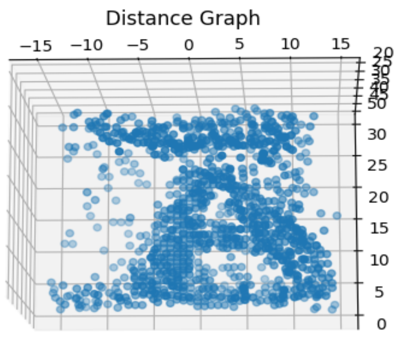
When we graphed the results from running our program (see Figure 11) we were clearly able to see the letter we were scanning in our graph. We saw some noisy points above and below our letter. We think that the noisy points above and below the letter are mainly due to the fact that our sensor was scanning too far above and below and was sensing the table and backdrop in the bottom and top of our graphs. There might also be some error with our servo when it is switching direction. Generally, however we are happy with the way our graphs turned out since we are able to clearly see the letters we were scanning in the 3d scatter plot and 3d color plots we created.


(a) Cardboard letter 'V' that was scanned.

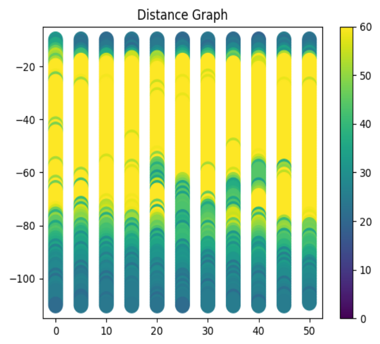


(b) Final scanned letter 'V.'

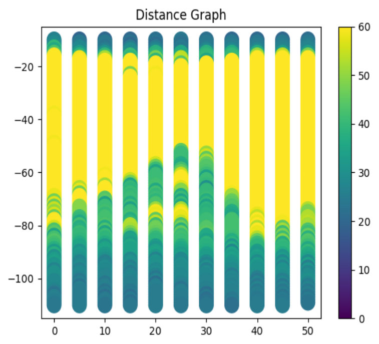

(c) Cardboard letter 'A' that was scanned.



(d) Final scanned letter 'A.'



(e) Heatmap representation of letter 'V.'



(f) Heatmap representation of letter 'A.'

Figure 11: The actual vs scanned letters.

3 Appendix

Listing 1: 2D Arduino Sweep Code

```
#include <Servo.h>

Servo myservo;
Servo myservo2;
const byte sensor_pin = A0;
const byte CMD_START = 1;

long prev_t = 0;
int sensor_value = 0;
int pos = 0;
byte cmd_id = 0;

String distance="";
String message = "";

void setup() {
  //Setup input and outputs
  myservo.attach(10);  // attaches the servo on pin 10 to the servo object
  myservo2.attach(9);
  myservo2.write(60);

  Serial.begin(9600);
}

void loop() {

  if(Serial.available() >= 0) {
    cmd_id = Serial.read();
  }
  else{
    cmd_id = 0;
  }

  switch(cmd_id){
    case 0:
        myservo.write(50);
      break;
    case CMD_START:
    // start sweeping the servos over the letter
    // goes from 70 degrees to 110 degrees
      for (pos = 70; pos <= 110; pos += 1) {
        // in steps of 1 degree
          myservo.write(pos); // tell servo to go to position in variable 'pos'
          delay(15);
          sensor_value = analogRead(sensor_pin);
          distance = int(5748.858*pow(sensor_value, -.868));
          message = String(pos) + "," + String(110) + "," + distance;
          Serial.println(message);
          delay(500);
      }
      message = "";
      break;
    break;
  }
}
```

Listing 2: Pan Tilt Arduino Sweep Code

```
#include <Servo.h>

Servo myservo;
Servo myservo2;
const byte sensor_pin = A0;
const byte CMD_START = 1;

long prev_t = 0;
int sensor_value = 0;
int pos = 0;
int pos2 = 0;
byte cmd_id = 0;

String distance="";
String message = "";

void setup() {
  //Setup input and outputs
  myservo.attach(10);  // attaches the servo on pin 10 to the servo object
  myservo2.attach(9);
  myservo.write(90);
  Serial.begin(9600);
}

void loop() {

  if(Serial.available() >= 0) {
    cmd_id = Serial.read();
  }
  else{
    cmd_id = 0;
  }

  switch(cmd_id){
    case 0:
        myservo.write(0);
        myservo2.write(10);
      break;
    case CMD_START:
    // start sweeping the servos over the letter
    // goes from 70 degrees to 110 degrees
      for (pos = 70; pos <= 110; pos += 1) {
        // in steps of 5 degrees
          myservo.write(pos); // tell servo to go to position in variable 'pos'
          delay(15);
        for (pos2=50; pos2<=90; pos2+=1){
            myservo2.write(pos2);
            delay(15);
            sensor_value = analogRead(sensor_pin);
            distance = int(5748.858*pow(sensor_value, -.868));
            message = String(pos) + "," + String(pos2) + "," + distance;
            Serial.println(message);
            delay(15);
        }
      }
      message = "";
      break;
    break;
  }
}
```

Listing 3: Python Graphing Code

```
#!/usr/local/bin/python

from serial import Serial, SerialException

import json
import math
import numpy as np
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D


cxn = Serial('/dev/ttyACM0', baudrate=9600)
running = False
results_list = []
xs = []
ys = []
zs = []

while(True):
    try:
        if not running:
            cmd_id = int(input("Please enter a command ID (1 -
             start the sweep, 0 - cancel sweep/stop: "))
        if int(cmd_id) > 1 or int(cmd_id) < 0:
            print ("Values other than 0 or 1 are ignored.")
        else:
            cxn.write([int(cmd_id)])
            if cmd_id == 1:
                running = True
            while cxn.inWaiting() < 1:
                pass
            result = cxn.readline();
            if '110,110,' in str(result):
                running = False
                cxn.write([0])
                cmd_id = 0
                with open('sweep_file.csv', 'w') as f:
                    json.dump(results_list, f)
                for i, my_result in enumerate(results_list):
                    if my_result[2] <= 50:
                        '''3d stuff'''
                        xval = my_result[2]*math.cos(math.radians
                        (my_result[0]))*math.sin(math.radians(my_result[1]))
                        yval = my_result[2]*math.sin(math.radians
                        (my_result[0]))*math.sin(math.radians(my_result[1]))
                        zval = my_result[2]*math.cos(math.radians(my_result[1]))

                        '''2d version'''
                        '''xval = my_result[0]
                        yval = my_result[2]'''

                        xs.append(xval)
                        ys.append(yval)
                        zs.append(zval)

                fig = plt.figure()
                ax = fig.add_subplot(111, projection='3d')

                ax.scatter(xs, ys, zs)

                '''
                plt.scatter(xs, ys)
                plt.xlabel('Servo 1 Angle (degrees)')
                plt.ylabel('Distance (cm)')
                '''

                plt.title('Distance_Graph')
                plt.savefig("graph.png")
                plt.show()
            result = str(result)
            result = result.strip("\\r\\n'")
            result = result[2:]
            servo1angle = result[:result.index(',')]
            result = result[result.index(',')+1:]
            servo2angle = result[:result.index(',')]
            result = result[result.index(',')+1:]
            distance = result
            results_list.append((int(servo1angle),
            int(servo2angle), int(distance)))
            print (str(result))
except ValueError:
    print ("You must enter an integer value between 1 and 3.")
```