PIE Project 2: DIY 3D Scanner

Kate McCurley, Maya Cranor, and Oliver Buchwald
September 29th, 2022

Introduction

Our assignment for this project was to make a 3D scanner using a pan-tilt mechanism to move an infrared distance sensor with the goal of being able to scan a letter and use the data to create a visualization in which the scanned letter is clearly recognizable. We programmed an Arduino to collect data from the IR sensor and move two servos controlling the pan-tilt mechanism, which we laser cut. After calibrating our sensor with measured distances, we used Python to visualize our recorded data and create a plot of the letters we scanned.

Bill of Materials

- Arduino
- USB cable
- 1X Sharp GP2Y0A02YK0F IR distance sensor
- 1X 3-pin Molex connector with red, black, and white wire tails
- 2X Hobbyking HK15138 servo motors
- Hardboard
- 4X ½” 8-32 bolts
- 4X 8-32 nuts
- Wood glue

Circuit Schematic

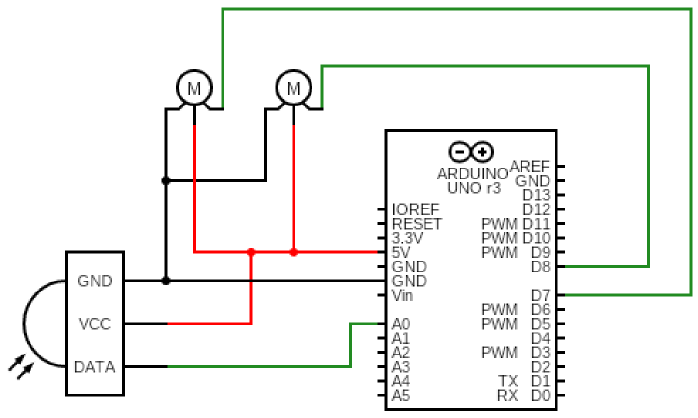


Figure 1: Schematic of our Arduino and Wiring

Figure 1 shows the electrical wiring of our project. The circles labeled M represent the servo motors, and the sensor is the infrared sensor we used to measure distance. Both servos and the sensor are connected to 5V and ground. The servos are connected to digital pins as they are controlled via PWM signals, while the sensor is connected to an analog pin so we can read data from it. We decided it was not necessary to add any resistors to that circuit as the spec sheets for both the IR sensor and the servos say that they can run at the current provided by our circuit. One change we would consider for the future is adding a capacitor, which would keep the input voltage more stable and allow for better consistency in the readings from the IR sensor.

Calibration Plot and Information

The process of testing and calibration involved setting up our sensor pointing straight forward (0 degrees pan and tilt) at a flat object. First, we verified that our sensor was generally working correctly by moving the object closer to and further from the sensor and seeing that this changed the values in an intuitive way. From there, we placed the object at measured distances and recorded the reading from the sensor.

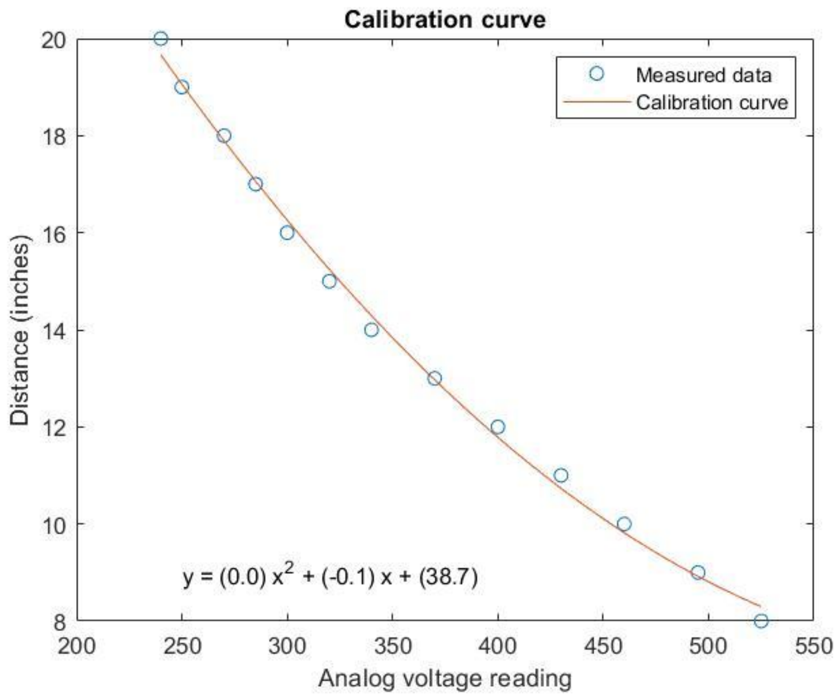


Figure 2: Calibration Curve for our IR sensor

This process generated our calibration curve (Figure 2). We generated an equation for a line of best fit using MATLAB polynomial curve fitting, which is shown in the corner of the graph. We tried using several different types of equations, and found that this polynomial gave the best fit. We verified this on our error plot (Figure 3), where we tested the equation against unique data points in the sensor range, which were different than the ones we used for calibration. We later used this same equation to generate our transfer function

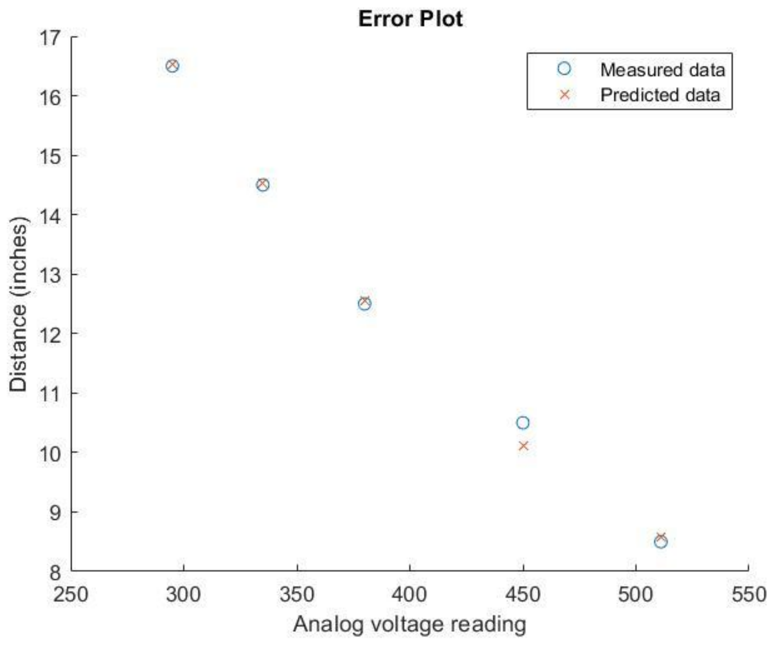


Figure 3: Error plot comparing newly measured readings to our calculated calibration values

Mechanical Design Process

We started the mechanical design of our pan-tilt scanner with the primary goal of centering both of our servos on the sensor. This goal meant that our sensor remained in one place and only tilted side to side and up and down, so our code did not have to account for a changing reference frame. In order to do this, we broke the mechanical elements into three main components: the sensor mount, the tilt servo holder, and the pan servo box.

Our fabrication was done almost entirely using laser cut hardboard and wood glue. We chose this method because it enabled us to quickly assemble an MVP of our mechanism. We made liberal use of finger joints in our design, which meant that we could slot our pieces together without needing to permanently glue them down, which was helpful for prototyping and iteration.

As a first attempt at designing our sensor mount we laser cut an L-shaped bracket. One side of the L screwed into the servo, and the other side screwed onto the sensor. This was a pretty bad way of mounting things, since it was very hard to square the L shape, and the sensor was not quite centered on the servo. Additionally, we found that screwing into the servo with the

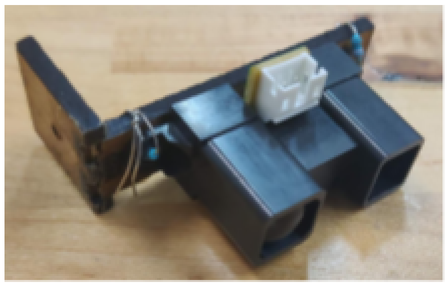


Figure 4: First iteration of sensor mount

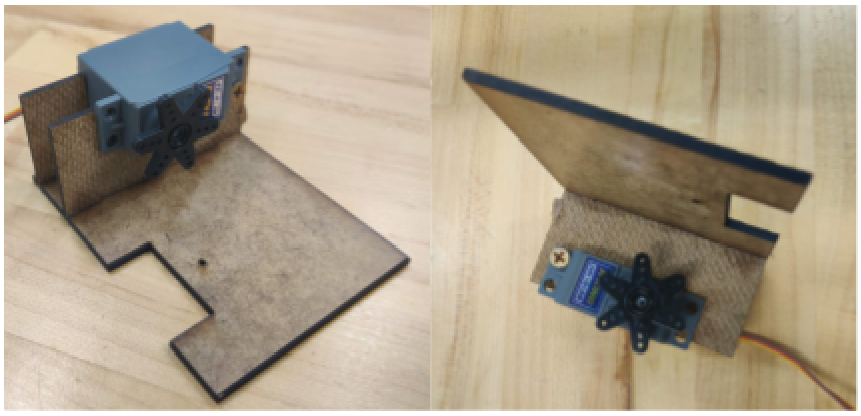


Figure 6: Initial tilt mechanism showing misaligned servo mount

provided screw resulted in a very loose connection, such that the sensor could easily spin independent of the servo.

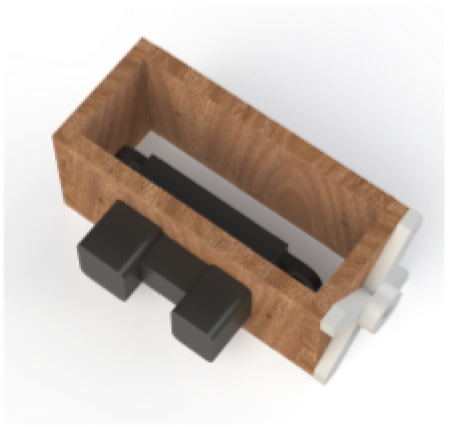


Figure 5: Final design for sensor mount

To fix this issue, we iterated on our design. Instead of an L shaped bracket, we designed a little box to hold the sensor. Since it was a 4 sided box rather than a freestanding L, we could use the 4 sides to square off of each other. It also had a cutout for the sensor, which is slotted in from inside rather than screwed onto the outside. This let us align the back of the sensor with the rotation axis of the servo, so that we could assume the position of the sensor is stationary. Lastly, the box was designed to interface with a servo horn rather than mounting directly to the servo. We laser cut small holes into one side for centering on a servo horn, and used resistors as placeholder twisty ties. Then we glued our hardboard together with wood glue, and super-glued the servo horn in place.

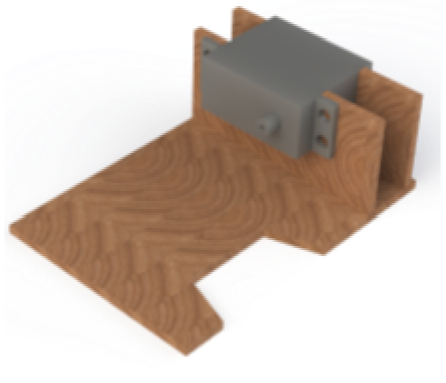


Figure 7: CAD of tilt mechanism

Next, we designed a holder for the “tilt” servo, which the sensor attached to. Our design for this part was pretty straightforward. We laser cut a base plate, as well as two pieces to support the servo, which slotted together with finger joints (yay!). The biggest issue we encountered at this stage was that the dimensions on the spec sheet for the servo were woefully incorrect. This resulted in an initial version of the servo mount that was terrible and wrong. However, with use of a pair of calipers, we were able to redimension our

CAD, and it worked flawlessly. This portion of our mechanical design was designed to be able to stand alone, for our initial 2D test scan. It had a single hole centered directly underneath the sensor, in order to interface with the pan mechanism.

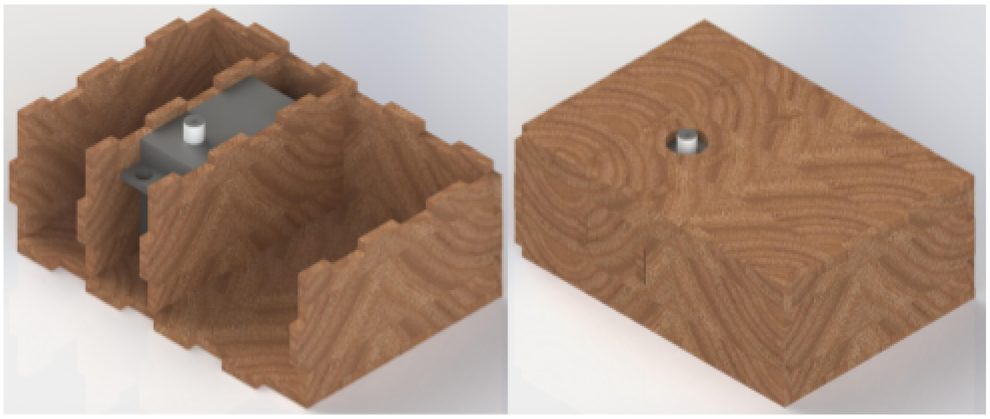


Figure 8: Pan servo box

Lastly, we designed a box to hold the bottom “pan” servo. This was quite literally a box, which had two dividers in its middle which we used to position the servo in place. When fully assembled, only the moving part of the servo protrudes out, allowing for a servo horn to slip on and attach to the table of the tilt mechanism.

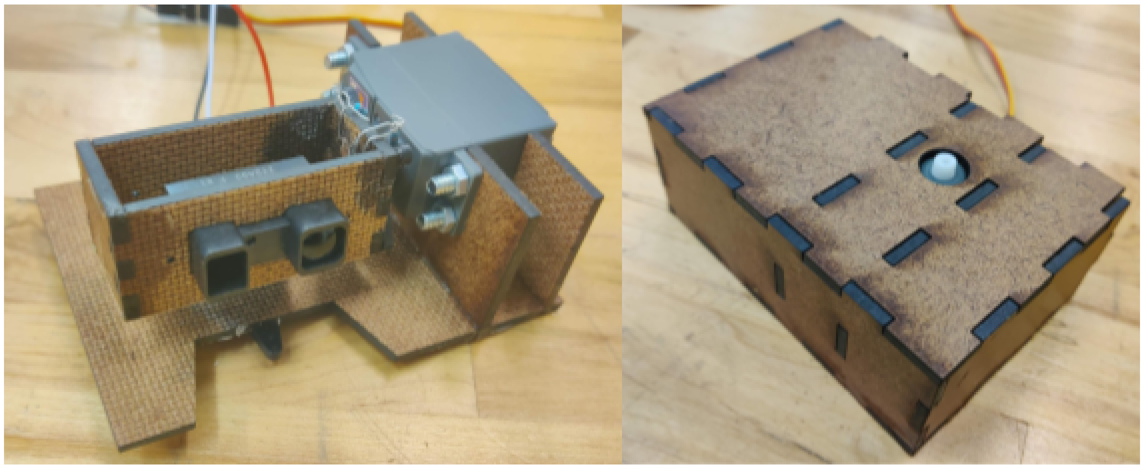
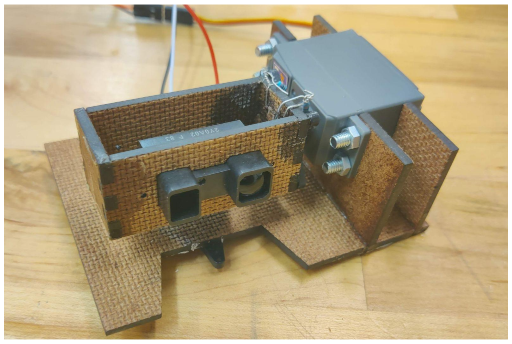


Figure 9: Final pan and tilt mechanisms assembled

Figure 9 shows both of our final mechanisms as assembled. In an ideal world, they would be slightly neater, and have less super glue drips. It would also be nice if we found small screws so that we could more securely attach the servo horns, rather than using a combination of superglue and resistor twisty ties. However, we made it work as shown.

One Servo Setup and Scan



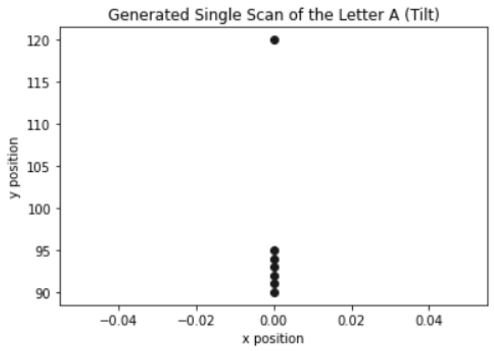


Figure 10: Setup for One Servo scan (tilt)
Figure 11: One Servo Scan of Letter A

We tested our scanner with only the tilt mechanism in order to first verify it worked. Figure 10 shows our test setup, which consisted of one servo and the sensor. We plotted the readings from this scan in Figure 11, showing one vertical cross section of the letter A.

Two Servo Setup and Scans

Next, we tested our scanner in 2D, combining both the pan and tilt mechanisms, as shown in figure 12. In order to produce a cleaner scan, we opened up the box holding our pan mechanism, and added a weight. We also propped up the scanner and letter so that our data would be cleaner. Before doing this, we noticed that the lower portion of our scan was very inaccurate due to the IR sensor reading the surface of the table instead of the air.

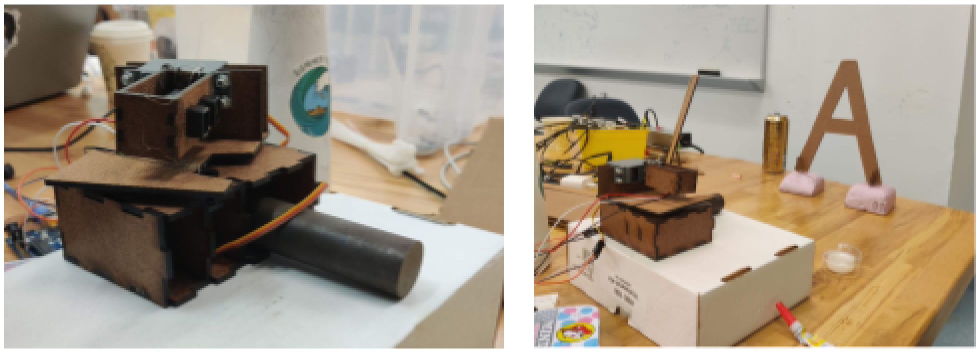
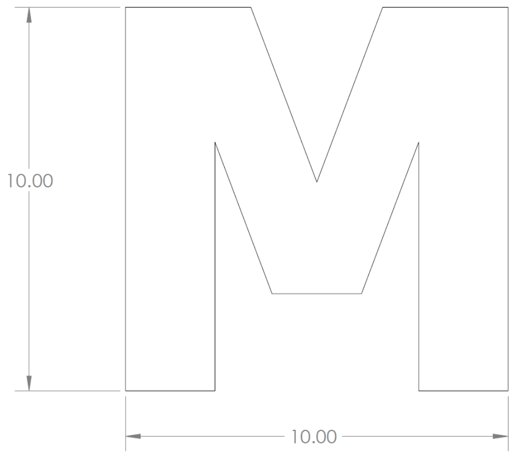


Figure 12: Setup for 2 Servo Scanning

We applied some data processing to the results of these scans, which we describe later in our Python Code explanation. This resulted in the plots shown in Figures 13 and 14.



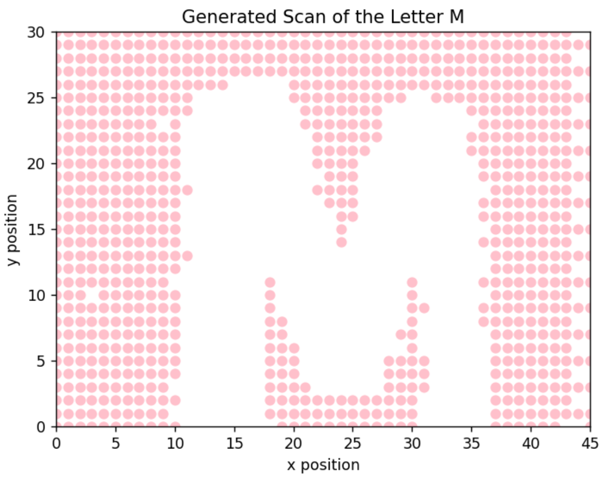


Figure 13: 2D scan of the letter M compared to a Solidworks drawing of it

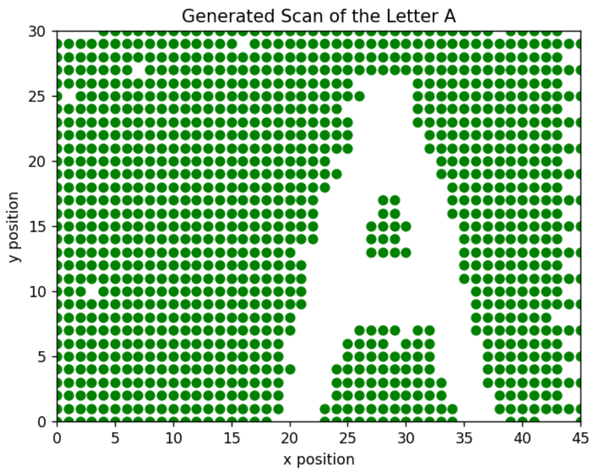


Figure 14: 2D scan of the letter A

Fun Code Diagram + Code Explanation/Design Decisions

The general idea of our code is that it uses the Arduino IDE to move the servos and read data from the IR sensor, then sends this data via Serial to Python, which converts the sensor data into a scanned image.

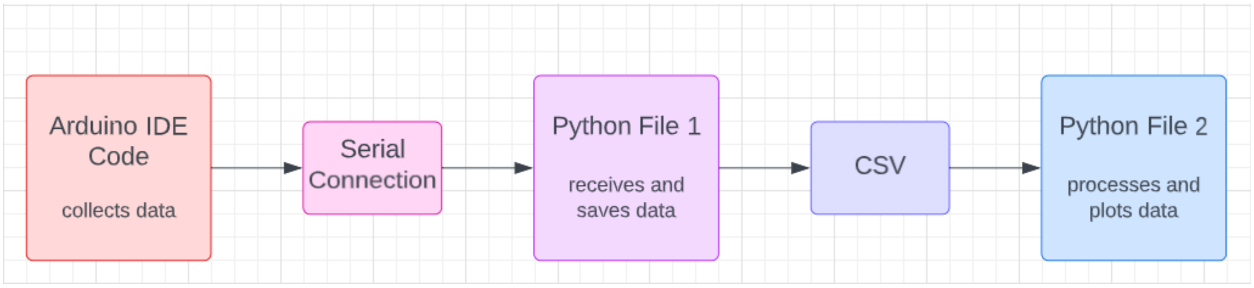


Figure 15: Diagram showing overall flow of scanned data throughout our code

Below is the text of the three files of our source code broken up with explanations of key features and decisions we made throughout the process.

Arduino Code and Documentation: Moving Servos and Reading Sensor Values

```
#include <Servo.h>

Servo panX;
Servo tiltY;
const uint8_t IR_SENS = A0;

const uint8_t PAN_INCREMENT = 1; // increment of pan angle in degrees
const uint8_t TILT_INCREMENT = 1; // increment of tilt angle in degrees
const uint8_t NUM_READINGS = 3; // number of readings taken at each point
const uint32_t DELAY = 50; // delay in ms between servo movements

int sum = 0; // for averaging sensor values

bool scanning = 1;

```

The first section of the code includes the servo library, for working with our two servos, and defines some variables for use throughout the code. The variables we define include Servo objects, the pin of the IR sensor, and constants for our scan - we defined these at the top of our code as constants so they would be easy to change as we tested. This allowed us to change how many degrees the servos moved between points, how many readings we took at each point, and how much time we waited at each point.

```
void setup() {
  panX.attach(7);
  tiltY.attach(8);

  pinMode(IR_SENS, INPUT);

  Serial.begin(9600);

  delay(5000); // time to start python script
}

```

The next section of our code is the `setup()` section, which runs once. This assigns our Servo objects to specific pins, sets the pinmode of the sensor to input, begins serial communication, then waits 5 seconds to allow us time to also set up and start running the Python code before scanning.

```
void loop() {
  while(scanning) {
    // go through all tilt angles
    for(int i = 135; i >= 105; i -= TILT_INCREMENT) {
      tiltY.write(i);
      // go through all pan angles
        for(int j = 45; j <= 80; j += PAN_INCREMENT) {
          // check if moving left or right (for scanning back and forth)
          if(i % 2 == 0) {
            panX.write(j); // moving from right to left
          }
          else {
            panX.write(125 - j); // moving from left to right
          }

          delay(DELAY); // wait briefly before taking data

          // take NUM_READINGS readings from the sensor and average them
          for(int n = 0; n < NUM_READINGS; n++) {
            sum += analogRead(IR_SENS);
          }

          // send angles and sensor data to Python
          Serial.println(String(i) + ", " + String(j) + ", " + String(sum / NUM_READINGS));

          sum = 0;
      }
    }
    Serial.println("end"); // let Python know the scan is complete
    scanning = 0; // stop moving Servos / taking data once scan is complete
  }
}
```

The final section of the code is our `loop()` function, which runs repeatedly. Inside this function, we have a while loop checking whether or not we're actively scanning - once we've taken our scan, the servos stop moving and we stop reading and sending data from the IR sensor. Within this, we have two for loops: the outside one looping through tilt angles and the inside one looping through pan angles. At each point we set the servo position to, we take the average of a number of readings based on `NUM_READINGS`, the constant we defined earlier in our code. We then send the tilt angle, pan angle, and distance sensor reading to the Serial port which communicates it to Python to be processed and plotted.

Python Code and Documentation: Processing and Visualizing Scanned Data

Python File 1: Read Serial Data and Write CSV

```python
import serial
import csv

arduino_com_port = "COM6"
baud_rate = 9600

serial_port = serial.Serial(arduino_com_port, baud_rate, timeout = 1)


reading_data = True
data = [[], [], []]

while reading_data:
  line_of_data = serial_port.readline().decode()
  if len(line_of_data) > 0:
    # print values for debugging
    # print("sensor value: " + line_of_data)

    # check if our scan is complete
    if "end" in line_of_data:
      reading_data = False
    else:
      # if not complete, add to data matrix
      data[0].append(int(line_of_data.split(", ")[0]))
      data[1].append(int(line_of_data.split(", ")[1]))
      data[2].append(int(line_of_data.split(", ")[2]))

```

Most of our first Python file is dedicated to reading the data sent over Serial from the Arduino. First we set up the connection to the serial port, then we loop through reading lines of data and adding them to the data matrix until the Arduino sends the line “end,” which indicates that the scan is complete and signals us to move on to the next section of the code.

```python
# write data to CSV
with open("letterA.csv", "w", newline = "") as f:
  w = csv.writer(f)
  for i in range(len(data[0])):
    w.writerow([data[0][i], data[1][i]], data[2][i])
```

The end of our first Python file writes the data to a CSV. We did this for two reasons: the first being that it allowed us to collaborate on code more easily, as one person could be working with the physical scanner while the other was working from CSVs of already scanned values, and the second being that it meant we didn’t have to re-scan (a time consuming process) every time we wanted to generate a new visualization of our data.

Python File 2: Read CSV, Process Data, and Create Graph

```python
import csv
import matplotlib.pyplot as plt

# calibration function into inches
def convert_to_inches(data):
  return 7.49078357932937710**-5 * data**2 + (-0.0972 * data) + 38.6755

```

Our code begins by defining our transfer function from the sensor voltage readings to distances in inches based on our calibration function described above. We use this function later on to check the distance of each point in our data.

```python
with open("letterA.csv") as f:
  r = csv.reader(f)

  points = [[], []]
  for row in r:
    if int(convert_to_inches(int(row[2]))) > 18:
      points[0].append(int(row[0]) - 90) # subtract start point in degrees
      points[1].append(int(row[1]) - 50) # subtract start point in degrees

```

Our next Python file begins by reading the CSV and turning it into a matrix of points (pairs of x, y values). The points are added to the grid if they are further than 18” from the sensor, as calculated by our transfer function. We placed our letter 18” from the sensor, which means that the points shown represent where the letter *was not present.* We decided on this visualization (shown previously in Figures 13 & 14) because we found it to look clearer with the letter as negative space than plotting points where the letter was scanned.

```python
# re-center data (due to scanner movement in opposite directions)
for i in range(len(points[0]) - 1):
    if int(points[1][i]) > int(points[1][i + 1]):
      points[1][i] += 2
    else:
      points[1][i] -= 2
```

Looking at our first visualizations of our data (shown below in Figure 16), it almost appeared as though there were two letters slightly off-set from one another.
We realized that because our scanner moves pretty quickly without stopping for a long time at each point, the direction in which the scanner is moving impacts the data values. To correct for this, we shifted the data slightly depending on the direction the scanner was moving. When the sensor was moving right to left we shifted the data right slightly, and did the opposite when it was moving left. We found that with all letters we graphed, this corrected for the scanning error and generated one clear image.

```python


plt.scatter(points[1], points[0], color = "green")

plt.xlim([0, 45])
plt.ylim([0, 30])
plt.xlabel("x position")
plt.ylabel("y position")
plt.title("Generated Scan of the Letter A")

plt.show()
```

The final section of our code takes the processed data and generates a scatterplot of it. We used matplotlib to generate the scatter plot, then set the x and y limits as well as axes labels and titles for a clearer and easier to understand graph.

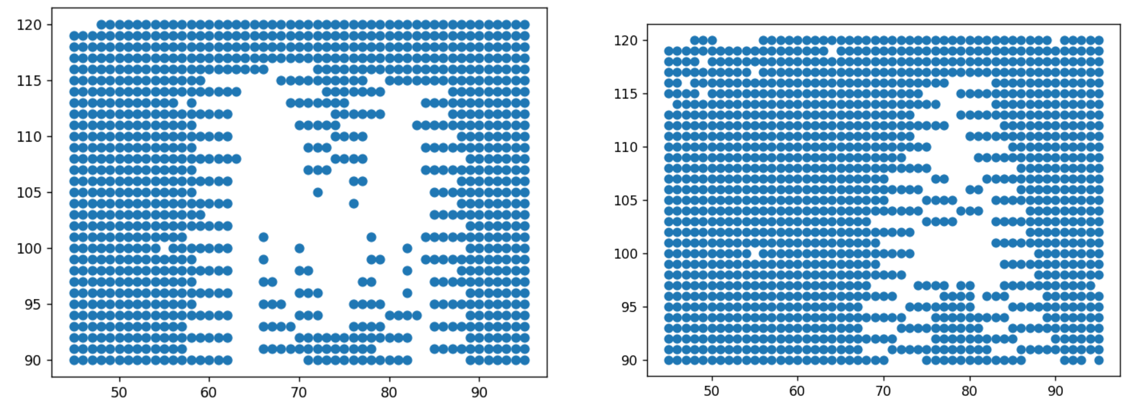


Figure 15: Initial Scans prior to improved data processing

Final Reflection / Challenges

In terms of our code, one of the biggest things we would work on given more time would be further tuning the scanning constants (the increments we moved the servos by, the amount of time we waited between points, the number of readings taken at each point). Not only would this allow us to be more accurate, it would also allow us to optimize for speed. Additionally, we initially decided it would make more sense to take a median of our readings at a certain point (disregarding outliers, which were our main concern) than a mean; however, implementing taking the mean took significantly less time.

We also learned a lot from our mechanical and electrical design processes. We went through several iterations on an MVP of our mechanical design in order to enable testing of our code while we worked on the final product. Given more time, we would have liked to make a slightly more finished mechanical portion of our scanner, but we were able to focus on function over form throughout.

Going into future PIE projects, we'd like to consider planning out our time further in advance. For this project, we didn't put much of a focus on hitting our deliverables. This meant that at the end of the project, we found ourselves rushing to finish things that were meant to be initial steps, such as the one servo scan. In the future, we'd hope to avoid this stressful situation by focusing first on the main deliverables so that we can dedicate our time and energy to the smaller improvements to our code and overall design we would've liked to make.

Code Index

This section contains code that was worked on, but not as part of the active code needed for scanning a letter. This includes ideas for future implementations if this project were to be continued.

Cartesian Code

This is code that we didn't have time to fully implement. It would be a good next step, but would have to be debugged. It converts the polar coordinates to cartesian coordinates. Additionally it would add some heatmapping to the plotting in order to show the depth of the scan.

```python
import math
import pandas as pd
import matplotlib.pyplot as plt

#input conditions of scan
x_start = 45
x_end = 95
y_start = 90
y_end = 120
x_zero = (-x_start+x_end+1)/2+x_start
y_zero = (-y_start+y_end+1)/2+y_start

```

Calculate where the origin of the scan is via the start end and end scanning angles.

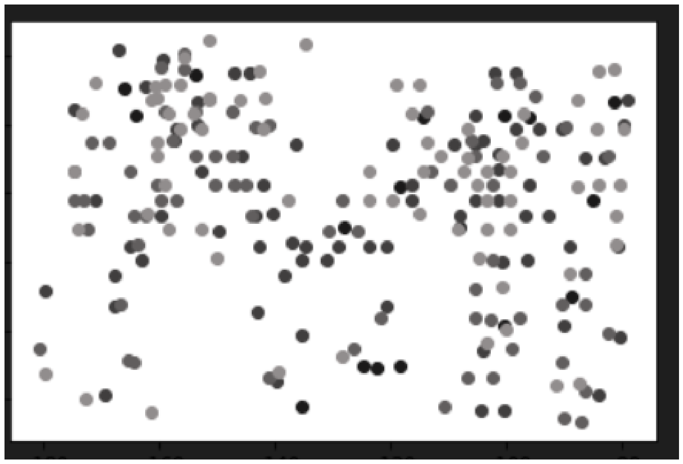
```python
#import scan
fullscan = pd.read_csv("A_M.csv")

#calibration function into inches
def convert_to_inches(data):
    converted_data = []
    for i in range(len(data)):

converted_data.append(7.490783579329377*10**-5*(int(data[i])**2) +
(-0.0972*int(data[i])) + 38.6755)
    return converted_data

#convert into cartesian coordinate system
def convert_to_x(x,data):
    converted_data = []
    for i in range(len(data)):
        x_angle = x[i] - x_zero
        x_coord = math.sin(math.radians(x_angle))*data[i]
        converted_data.append(x_coord)
    return converted_data

#convert into cartesian coordinate system
def convert_to_y(y,data):
    converted_data = []
    for i in range(len(data)):
        y_angle = y[i] - y_zero
        y_coord = math.sin(math.radians(y_angle))*data[i]
        converted_data.append(y_coord)
    return converted_data
```

Using basic trig with right triangles the code calculates the angle from the origin and uses the reading from the sensor as the hypotenuse. It then calculates the opposite side which should be the offset from the origin and the cartesian coordinate. It works the same for both x and y. However during implementation it appeared to skew the data, and we didn’t have the time to fully debug why that was happening. One possible reason is that we abstracted the sensor as being a fixed point when in reality it shifted over a tiny bit (we designed the mechanical side to try to prevent this as much as possible, but it’s impossible to get perfect). Another reason is that the sensor has a margin of error, and since the sensor reading is used to calculate the “side of the triangle” it could make the results more fuzzy.

```
#run functions for conversions
distance = convert_to_inches(fullscan['depth'])
x = convert_to_y(fullscan['pan'],fullscan['depth'])
y = convert_to_y(fullscan['tilt'],fullscan['depth'])

#transform functions to be graphable by a scatter plot
def plot_points(x,y,distance,start,end):
    points = [[], []]
    for i in range(len(x)):
        if start <distance[i] ⩽ end:
            points[0].append(x[i])
            points[1].append(y[i])
    return points

#create heatmap buckets of different colors for the graph
points1 = plot_points(x,y,distance,19.4,19.6)
points2 = plot_points(x,y,distance,19.6,19.8)
points3 = plot_points(x,y,distance,19.2,19.4)
points4 = plot_points(x,y,distance,19.0,19.2)
points5 = plot_points(x,y,distance,19.8,20.0)
points6 = plot_points(x,y,distance,18.8,19.0)
points7 = plot_points(x,y,distance,20.0,20.2)

plt.scatter(points7[0], points7[1], c= '#1a1919')
plt.scatter(points6[0], points6[1], c= '#403e3e')
plt.scatter(points5[0], points5[1], c= '#403e3e')
plt.scatter(points4[0], points4[1], c= '#615e5e')
plt.scatter(points3[0], points3[1], c= '#615e5e')
plt.scatter(points2[0], points2[1], c= '#918d8d')
plt.scatter(points1[0], points1[1], c= '#918d8d')
plt.show()

```

To give the plot a heatmap effect, we hard-coded buckets to sort the data into. This was based on trial and error. Ideally there would be a mathematical formula to calculate it. Potentially a different plotting tool might have had built in capabilities.

This is what the plot looked like:



The M is definitely visible, but not fully defined. With more time to debug, we think that our code would work.

Matlab converison/transfer function code

```matlab
%calibration code
clf
y = [8,9,10,11,12,13,14,15,16,17,18,19,20];
x = [525,495,460,430,400,370,340,320,300,285,270,250,240];

p = polyfit(x,y,2)

x2 = 240:.1:525;
y2 = polyval(p,x2);
plot(x,y,'o',x2,y2)
s = sprintf('y = (%.1f) x^2 + (%.1f) x +
(%.1f)',p(1),p(2),p(3))
text(250, 9 ,s)

xlabel('Analog voltage reading')
ylabel('Distance (inches)')
title('Calibration curve')
legend('Measured data', 'Calibration curve', 'location',
'northeast')

%error plot code
clf
figure
hold on
error_y = [8.5,10.5,12.5,14.5,16.5]
error_x = [511,450,380,335,295]
error_predicted_y = p(1)*error_x.^2+p(2)*error_x+p(3);
plot(error_x,error_y,'o')
plot(error_x,error_predicted_y,'x')

xlabel('Analog voltage reading')
ylabel('Distance (inches)')
title('Error Plot')
legend('Measured data', 'Predicted data', 'location',
'northeast')
```