PIE MP2

Richard Li and Karina Lamoreux

September 2022

1 Introduction

Our mission for PIE MP2 was to create a pan/tilt mechanism using servos and an infrared sensor that is controlled by an Arduino. Then, we wanted to transmit servo angle and distance information from the sensor to our laptop for storage and visualization. We then wanted to create a 3D visual representation of an object of known, well-defined geometry (in our case, the letter K).

2 Bill of Materials

- Arduino
- USB cable
- 1X Sharp GP2Y0A02YK0F IR distance sensor (datasheet)
- 1X 3-pin Molex connector with red, black, and white wire tails
- 2X Tiankongrc MG996R servo motors

3 Process

3.1 Getting Sensor Output

Our code base uses Serial, a communications protocol, to read sensor outputs that are passed into the Arduino and process them in a Python script. A Serial connection can be established by creating a Serial instance in Arduino:

Arduino:

```
1 void setup() {
2     // initialize serial communications at 9600 bps:
3     Serial.begin(9600);
4 }
```

And then creating a Serial port in Python that's ready to receive data.
Python:

```
IR_SENSOR_PORT = "/dev/ttyACM0"
BAUD_RATE = 9600
serial_port = serial.Serial(IR_SENSOR_PORT, BAUD_RATE, timeout=1)
```

Then, to collect readings from the IR sensor, we can create an infinite loop that will print out the value that the sensor is reading:

Arduino:

```
void loop() {
    sensorValue = analogRead(analogInPin);
    Serial.println(sensorValue)
}
```

Python:

```
while True:
    data = serial_port.readline().decode()
    if len(data) > 0:
        print(data)
```

This will repeatedly give us outputs in Python that we can use to calibrate the sensor by measuring it against a known length.

3.2 Calibration

To calibrate our sensor, we measured the voltage output from the sensor at distances between 20cm and 50cm. Using Google Sheets, we compared them in a graph (shown in Figure 1) and found the calibration equation, relating distance (cm) Voltave (mV) using the google sheets trendline function:

Calibration Equation:

$$Voltage = 780 * e^{-0.0253x} \tag{1}$$

This can gives us distance(x) from any voltage that IR sensor reads. From there, we measured the voltage readings of four other distances within the range and added them to the graph (shown in red). We calculated the percent error for each point and found that the range of error was -2.39 % to 1.27% with and average of -0.41% error. As visible in Figure 1, the test points display a reliable calibration curve

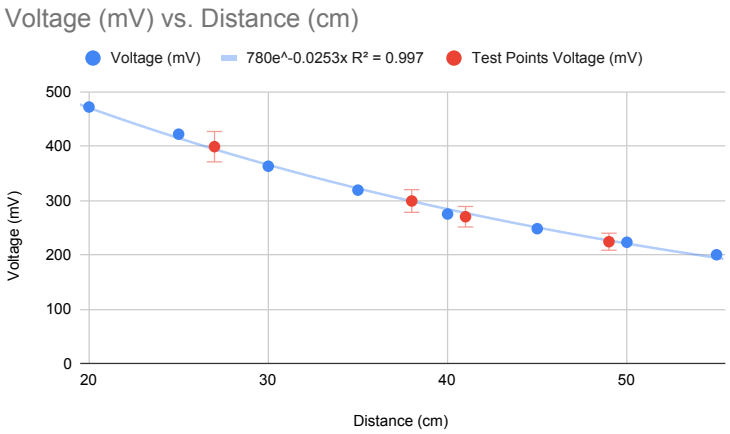

Figure 1: Calibration curve graph

3.3 Mechanical Design

To create the pan-tilt mechanism, we first looked at examples of other people's designs on the internet. We looked for simple designs that we could easily fabricate for inspiration. One image we found helpful is shown in Figure 2. They have a motor, mounted to a base plate, that faces up and spins a disk. On the side of this disk, a second motor is mounted horizontally that spins a plate.

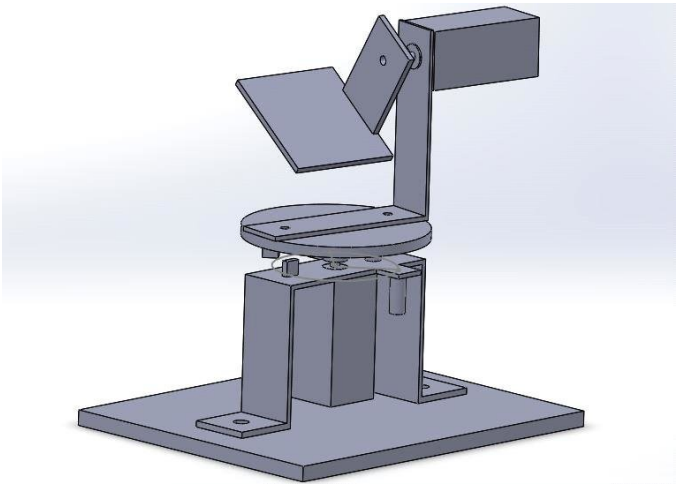

Figure 2: Inspiration image found from google image search: "Pan Tilt Mechanisms" ** We did not design this

Using these concepts, we created some initial sketches to make something similar. From the materials that came with the servo, we wanted to use the servo horns to connect the servo to the plates. Our initial sketch of the different sections of the pan-tilt mechanism are shown in Figure 3.

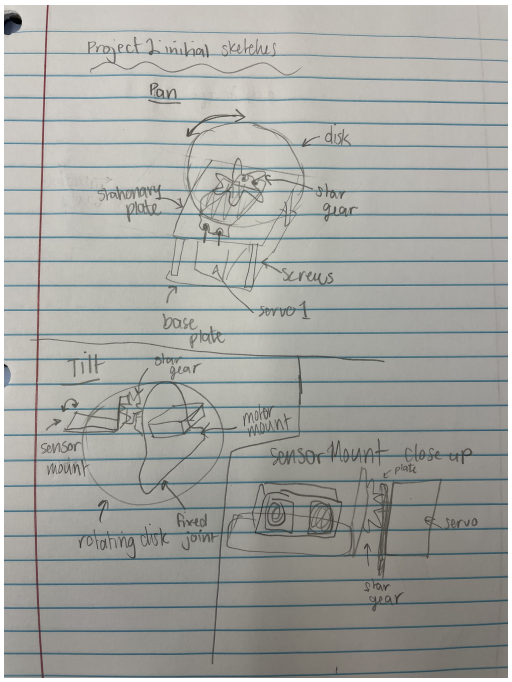

Figure 3: Initial sketches of our pan-tilt mechanism

Next, we created an initial design very similar to the sketches in SolidWorks (Figure 4).

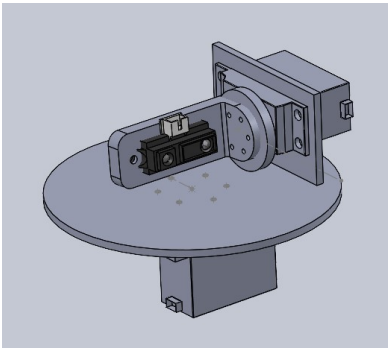

Figure 4: Initial CAD of our pan-tilt mechanism

Using the initial CAD, we adjusted the shape of each piece so the sensor would rotate directly around its center. We wanted to make sure there was no translation of the sensor during our sweeps that would create extra math for ourselves to do later. After we were happy with the relationship between each piece and the axes of rotation, we adapted the design of our two bracket pieces to make them more robust and take up less space as seen in Figure 5. The two brackets are shown in purple. We also added a base to mount the motor, Arduino, and breadboard.

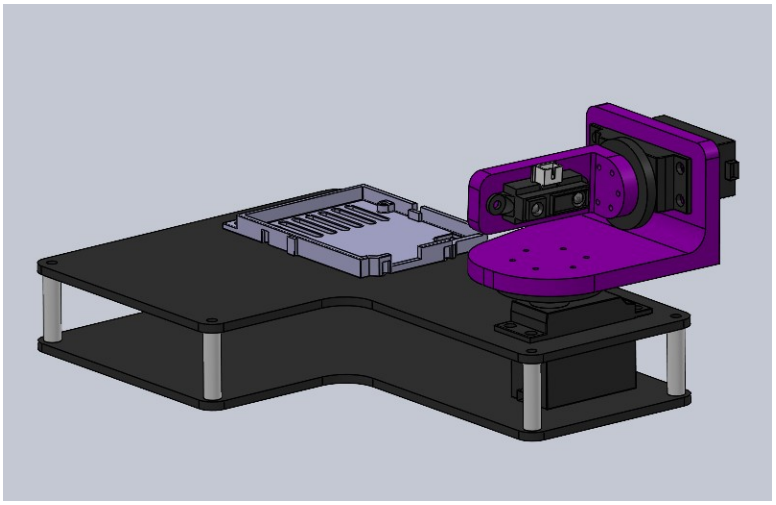

Figure 5: Final CAD

To realize our CAD model, we used the 3D printers to print out our two brackets and the laser cutter to cut out the two base sheets from black acrylic. We used wire to attach the purple brackets to the servo horns, and attached everything else with nuts and bolts – including 1" spacers between the two base sheets. Figure 6 displays an image of our final mechanism.

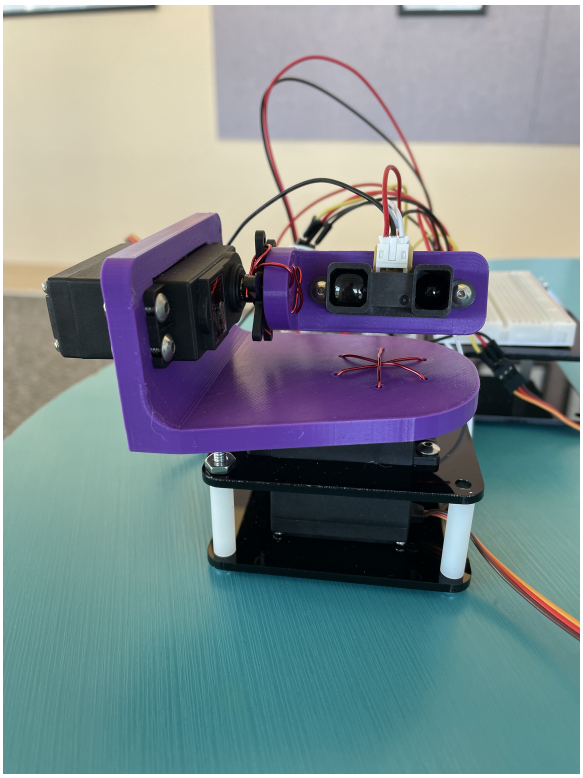

Figure 6: Final mechanism

3.4 Electrical Design

To control our pan-tilt mechanism, we first looked at the specification sheets for the servo motors and sensor to learn how their pin-outs corresponds to ground, power and signal. After learning this, we powered each of our electrical components with 5V and plugged and connected them all to ground. We plugged our panning motor into digital pin 9 and our tilting motor into digital pin 8. We connected our sensor to A4 to give data back to the computer, with a bypass capacitor of 10 µF as specified in the data sheet. Figure 7 displays our electrical schematic.

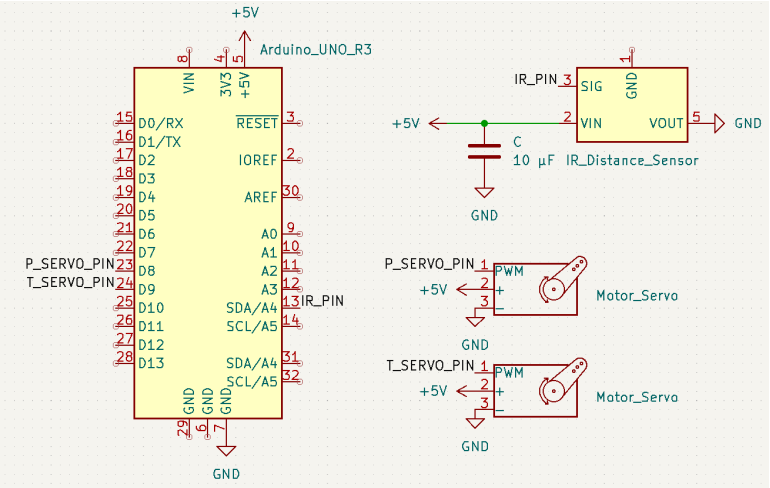

Figure 7: Our electrical schematic

We did face a problem where plugging in the IR sensor seemed to "break" our Arduino. After a few minutes of panicking and wondering if we had fried an Arduino, we realized that the exposed pins of the IR sensor were touching each other on our breadboard. This resulted in a short that was preventing us from sending any signals to the Arduino. After electrical-taping the two pins so they weren't touching, everything worked smoothly.

3.5 Software Design

As previously mentioned, our code base for this project is split between Arduino and Python. Because we knew that the motors would move more mass when panning (left to right) than tilting (top to bottom) simply by nature of our design, we decided to program our 3-D scanner to scan top to bottom, then pan, scan top to bottom, then pan, etc. This helped stabilize our machine and give us cleaner readings.

In Arduino, we command the servo motors to move (lines 3,4 12, and 17) and command the IR sensor to read data (line 10). We additionally, ask the Arduino to tell us – over Serial – the positions of the motors (lines 8-9 and 15-16) and the reading of the IR sensor (line 11). This will be transmitted the same way our calibration data was transmitted.

Arduino:

```
1  void loop() {
2    // set servos to initial position
3    pan_servo.write(pan_pos);
4    tilt_servo.write(tilt_pos);
5    Serial.println("starting");
```

```
  for(pan_pos = 0; pan_pos <= 45; pan_pos += 1){
    for(tilt_pos = 30; tilt_pos < 100; tilt_pos +=1){
       Serial.print("tilt ");
       Serial.println(tilt_pos);
       sensorValue = analogRead(analogInPin);
       Serial.println(sensorValue);
       tilt_servo.write(tilt_pos);
       delay(40);
    }
    Serial.print("pan ");
    Serial.println(pan_pos);
    pan_servo.write(pan_pos);
    delay(40);
  }
}
```

In Python, we receive and organize this data for visualization using Pandas (a data manipulation package)...

Python:

```
# X for pan degree, Y for tilt degree, S for raw sensor reading
display = pd.DataFrame(index=["X", "Y", "S"],
                       columns=range(0, ((tilt_range-1)*pan_range)))

# Initialize steppers that will iterate through dataframe
tilt_step = 0
pan_step = 0
step = 0
```

...then slowly fill it up with the data that the Arduino transmitted. Notice that we search for "tilt" and "pan" in the string that the Arduino transmits to dictate where and how data is logged. This prevents any buggy inputs from the Arduino (an extra tick of tilt that shouldn't exist, for example) from sneaking into our data, as it should break when it receives the incorrect input. Every

```
while pan_step < pan_range and starting:
    data = serial_port.readline().decode()
    if len(data) > 0:
        print(data)
        if "tilt" in data:
            tilt_step += 1
            # Print out tilt angle
            print(data)
            continue

```

```
11          if "pan" in data:
12              pan_step += 1
13              tilt_step = 0
14              # Print out pan angle
15              print(data)
16              continue
17
18          # Log to database
19          display.at["X", step] = int(pan_step)
20          display.at["Y", step] = int(tilt_step)
21          display.at["S", step] = int(data)
22
23      step += 1
```

Afterwards, we can save "display" to a csv.

```
1 path = "display_panning"
2 display.to_csv(f"{path}.csv")
```

"path" now references our file where our data lives.

The way that we chose to visualize depth was using a color map, where darker colors corresponded to parts of the image that were closer to the camera. Matplotlib, the visualization package we used, has existing color mapping tools that we used to make this happen (line 2 and 9). Additionally, since we scanned right to left, we had to negate our X values (line 9) in order to display an image in the correct orientation.

Python:

```
1  display = pd.read_csv(f"{path}.csv")
2  display = display.iloc[:, 1:]
3  display = display.to_numpy()
4
5  # normalize colors to map them.
6  normalized_colors = display[2]/np.linalg.norm(display[2])
7
8  fig = plt.figure()
9  # to prevent stretching
10 plt.axis("equal")
11
12 # negate x values because we scanned right -> left
13 plt.scatter(x=-display[0], y=display[1], cmap="Greys", c=normalized_colors)
14 plt.savefig('K_panning.png', dpi='figure')
15 fig.tight_layout()
16 plt.show()
```

Finally, we need to calculate the actual distance that corresponds to the sensor reading. Utilizing our calibration curve, we can implement a simple calculation

function (lines 13-17) in Python that will replace our data file's raw readings with actual numbers in centimeters (lines 6-10).

Python:

```
1  def calculate(path):
2      new_distances = []
3      database = pd.read_csv(f"{path}.csv")
4      database = database.iloc[:, 1:]
5      database = database.to_numpy()
6      for distance in database[2]:
7          distance = calculate_real_distance(distance)
8          new_distances.append(distance)
9      database[2] = new_distances
10     pd.DataFrame(database).to_csv(f"{path}_calculations.csv")
11     return True
12
13 def calculate_real_distance(ir_reading):
14     if ir_reading == 0:
15         return 0.0
16     distance = (math.log(ir_reading/780))/-0.0253
17     return distance
```

3.6 One Sensor Sweep

We did our 1 sensor sweep of our K a little under the middle of the letter. We set the tilt range to 1, essentially preventing it from moving as we scanned horizontally.

As can be seen in figure 8, there are two very distinct clusters of data that represent the two legs of the letter "K". That cluster of data at the very left isn't relevant data (likely one of our bodies that got in the way).

To create our sweep, we only used the 'pan' motor (shown in blue in Figure 8) to rotate the entire upper mechanism left and right.

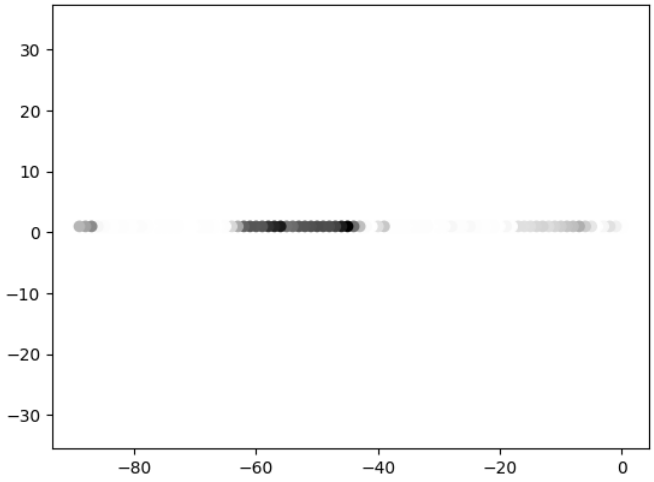

Figure 8: Using only the panning servo to sweep the letter K

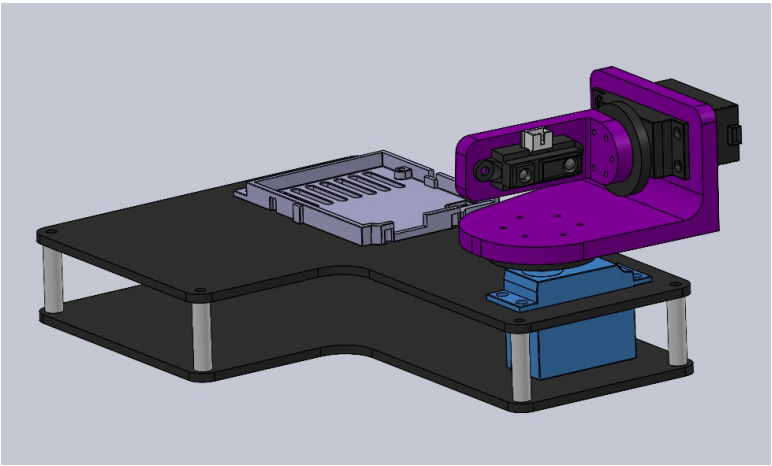

Figure 9: Components used to do a one sensor sweep of the letter highlighted

3.7 Two Sensor Sweep

After we've accomplished a one sensor sweep, we changed the tilt range until it was sweeping the whole letter, and then ran it again. Note that figures 5 and 9 do show the entire mechanism including the two servos that handle the two sensor sweep.

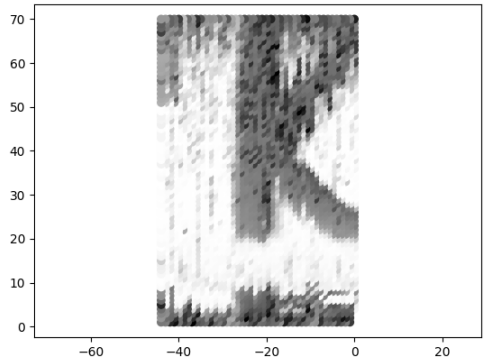

Figure 10: Sweeping the letter K with both the pan and tilt sensors

4 Reflection

Overall, the mechanical aspects of our design worked well for our purposes. It likely would have been beneficial to make the motor and sensor mounting brackets lighter, as the lower bracket was leaning sideways a little bit due to the weight of the motor. We also could have tried to put the motor closer to the center of rotation to solve this issue. Secondly, wire generally isn't a good choice to connect the brackets to the horns. If we were creating anything more robust, we would want to use something like bolts.

With regards to the code, we were kind of banking on the fact that we would collect so many sensor readings that the accuracy of any single reading didn't actually matter so much. Even in figure 10, this manifested itself as an array of differently-colored, "somewhat dark" spots that make up the K. If we wanted a smoother illustration, we should've paused at every point, taken multiple readings, and averaged those before actually storing the data.

Additionally, the code base was developed rather hastily because we wanted to get data as quickly as possible. This led to a lot of redesign as we actually tried to make the code modular and clean. The addition of "visualization" and "calculation" functions were huge quality-of-life improvements that could've come much earlier if we had planned out the code's workflow from the beginning.

However, despite all this, it was still a code base that did what we needed it to do, and the added improvements gave us huge amounts of freedom to customize our workflow and make it as specific as we needed.

The teaming experience was really awesome, since we both got to see the way our fields worked with each other to build something cool!

5 Source Code

5.1 Arduino

```
1
2 #include <Servo.h>
3 Servo pan_servo;
4 Servo tilt_servo;
5
6 const int analogInPin = A4;  // Analog input pin that the
      potentiometer is attached to
7 int pan_pos = 0;
8 int tilt_pos = 30;
9
10 int sensorValue = 0;        // value read from the pot
11
12 void setup() {
13   // initialize serial communications at 9600 bps:
14   pan_servo.attach(9);  // attaches the pan servo on pin
       9 to the servo object
15   tilt_servo.attach(8);  // attaches the tilt servo on
       pin 10 to the servo object
16   Serial.begin(9600);
17 }
18
19 void loop() {
20   // set servos to initial position
21   pan_servo.write(pan_pos);
22   tilt_servo.write(tilt_pos);
23   Serial.println("starting");
24   for(pan_pos = 0; pan_pos <= 45; pan_pos += 1){
25     for(tilt_pos = 30; tilt_pos < 100; tilt_pos +=1){
26        Serial.print("tilt ");
27        Serial.println(tilt_pos);
28        sensorValue = analogRead(analogInPin);
29        Serial.println(sensorValue);
30        tilt_servo.write(tilt_pos);
31        delay(40);
32     }
33     Serial.print("pan ");
34     Serial.println(pan_pos);
35     pan_servo.write(pan_pos);
36     delay(40);
37   }
38 }
```

5.2 Python

```python
"""
Runs the main pipeline of our 3D visualizer system
"""

import math
import serial
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd

# Define a few constants to streamline workflow
USING_SENSOR = False  # So there aren't errors if we just want to simulate data
IR_SENSOR_PORT = "/dev/ttyACM1" # the port that the arduino is plugged into
BAUD_RATE = 9600

DISPLAY = True  # Show the plot with the visualization of the data
CALCULATE = True  # Calculate the actual distance associated with the sensor readings
LOGGING = True  # Save the data to some file for future visualization


def visualize(path):
    """
    To simplify repeating the visualization of the data without needing to run with the
    arduino every time, we saved the data from a previous arduino run into some .csv,
    and now we can simply visualize that instead with this code.

    Args:
        path (str): the path to the csv file containing the IR sensor data.

    Returns:
        Nothing, but visualizes the plot
    """
    display = pd.read_csv(f"{path}.csv")
    display = display.iloc[:, 1:]
    display = display.to_numpy()

    # normalize colors to map them.
    normalized_colors = display[2]/np.linalg.norm(display[2])

    fig = plt.figure()
    # to prevent stretching
    plt.axis("equal")

    # negate x values because we scanned right -> left
    plt.scatter(x=-display[0], y=display[1], cmap="Greys", c=normalized_colors)
    plt.savefig('K_panning.png', dpi='figure')
    fig.tight_layout()
    plt.show()
```

```
49
50 def calculate(path):
51     """
52     Calculate real distances given the sensor readings
53
54     Args:
55       PATH (str): The path to the datafile containing our IR data.
56
57     Returns:
58       True once completed
59
60     """
61     new_distances = []
62     database = pd.read_csv(f"{path}.csv")
63     database = database.iloc[:, 1:]
64     database = database.to_numpy()
65     for distance in database[2]:
66         distance = calculate_real_distance(distance)
67         new_distances.append(distance)
68     database[2] = new_distances
69     pd.DataFrame(database).to_csv(f"{path}_calculations.csv")
70     return True
71
72 def calculate_real_distance(ir_reading):
73     """
74     Given some reading from the IR sensor, plug it into the calibration
75     curve to translate that to a real distance.
76
77     Args:
78         ir_reading (int): the raw number that the ir sensor spits out.
79     """
80     # We want to account for this case because ln(0) returns an error
81     if ir_reading == 0:
82         return 0.0
83     distance = (math.log(ir_reading/780))/-0.0253
84     return distance
85
86 def main():
87     """
88     Run the main pipeline of the sensor visualization.
89     """
90     if USING_SENSOR:
91         serial_port = serial.Serial(IR_SENSOR_PORT, BAUD_RATE, timeout=1)
92
93     # Corresponding with degrees. Set based upon arduino settings
94     tilt_range = 1
95     pan_range = 45
96
97     # X for pan degree, Y for tilt degree, S for raw sensor reading
98     display = pd.DataFrame(index=["X", "Y", "S"],
```

```
                            columns=range(0, ((tilt_range-1)*pan_range)))

    # Initialize steppers that will iterate through dataframe
    tilt_step = 0
    pan_step = 0
    step = 0

    # Wait for the arduino to say you're starting before starting
    starting = False
    while starting is False:
        data = serial_port.readline().decode()
        if len(data) > 0:
            if data == "starting\r\n":
                starting = True

    # Main workflow: step through pan and tilt and record sensor reading
    while pan_step < pan_range and starting:
        data = serial_port.readline().decode()
        if len(data) > 0:
            print(data)
            if "tilt" in data:
                tilt_step += 1
                # Print out tilt angle
                print(data)
                continue

            if "pan" in data:
                pan_step += 1
                tilt_step = 0
                # Print out pan angle
                print(data)
                continue

            # Log to database
            display.at["X", step] = int(pan_step)
            display.at["Y", step] = int(tilt_step)
            display.at["S", step] = int(data)

        step += 1

    # Define a filename to log to
    path = "display_panning"

    if LOGGING:
        display.to_csv(f"{path}.csv")

    if DISPLAY:
        visualize(path)

    if CALCULATE:
```

```
149         calculate(path)
150
151
152 if __name__ == "__main__":
153     main()
```