Mini-Project 2: DIY 3D Scanner

Elisa Dhanger, Jennifer Lee, Luke Nonas-Hunter

Section 3+4: Group 58

9/16/21

Abstract

In this project, we were tasked to create a mechanism that could scan a 3D object (in this case a cardboard cutout of a letter) and create a visualization for it. We used 2 servo sensors, an infrared distance sensor, and an Arduino for the electrical/mechanical component of this project. The mechanical support structures were 3D printed, and everything was coded in Arduino and python through VScode.

1 Procedure

1.1 Circuit

To begin, we started by connecting the necessary components to the Arduino. We were required to use two servo motors and an infrared distance sensor. We connected the servos to pin 9 and 10, and we connected the infrared distance sensor to analog pin 0 since we needed to receive a set of continuous data (Figure 1).

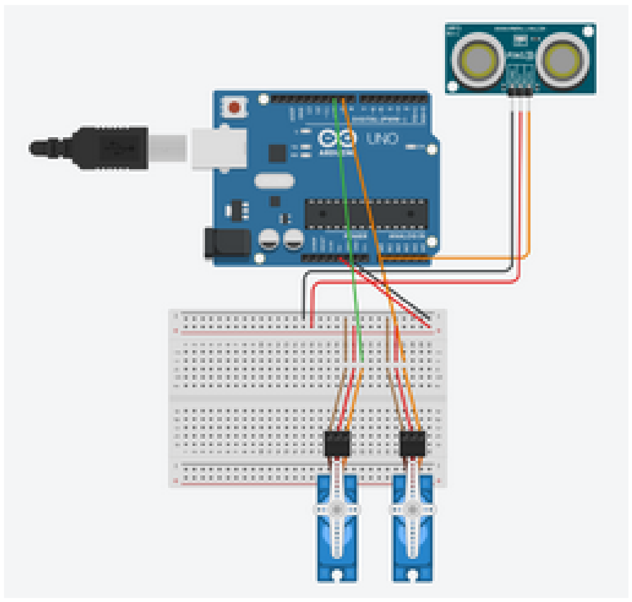

Figure 1: Circuit Diagram of setup.

After setting up the circuit, we started to run some calibration data for the sensor. The data that the Arduino was reading does not directly translate to distance, and in addition, we knew that the relationship between the analogRead() data and actual distance was not a direct linear relation. Therefore, we set up the sensor against a tape measure and recorded the analogRead() data at several distances across 0-28 inches into a spreadsheet and graphed out the data. Based on the results of these findings, we were able to calculate a line of best fit which would serve as the calibration curve (Figure 2).

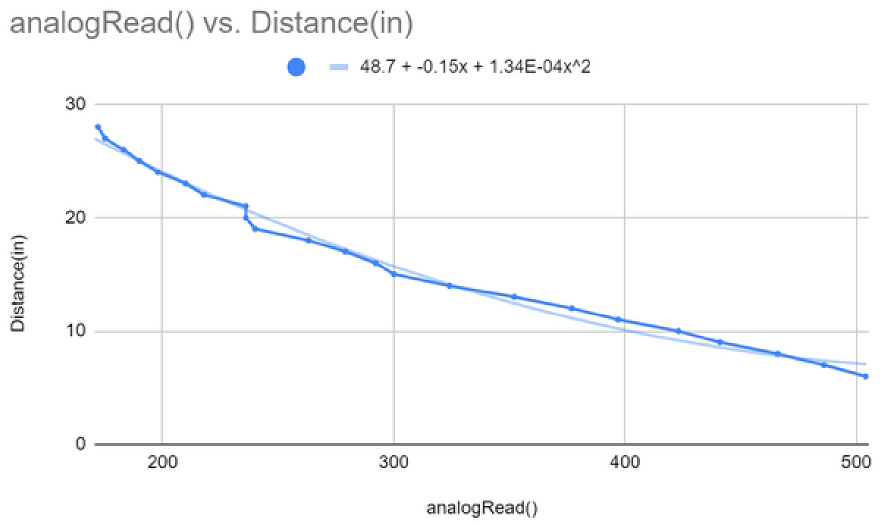

Figure 2: Calibration plot depicting relationship between the analogRead() data and the actual distance.

After setting up our calibration curve, we also were tasked to create an error plot to test how accurate the calibration curve is by taking measurements at certain distances and comparing the analogRead value for the actual distance vs. the predicted distance through the line of best fit (Figure 3). We had come to the conclusion that the distance was pretty accurate based on how low the error is, and that we could continue onto the next step.

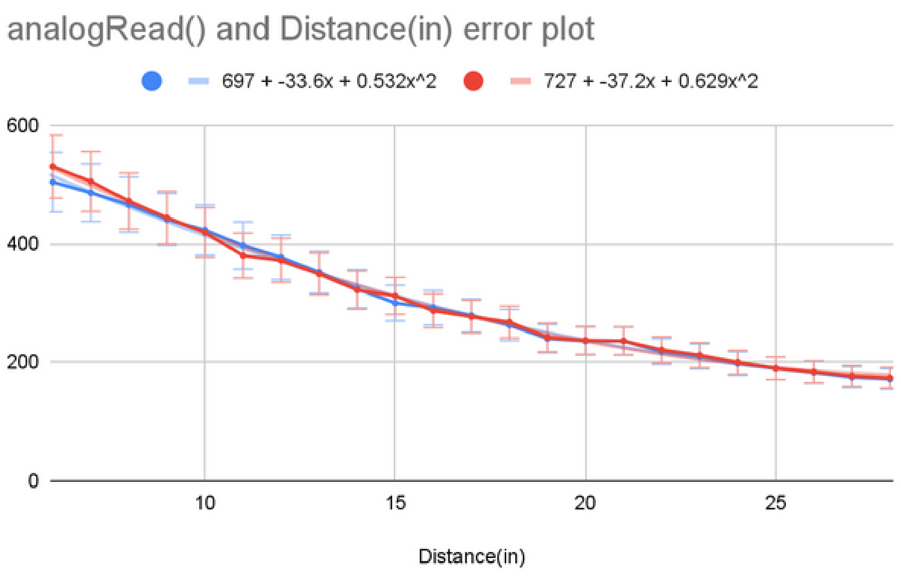

Figure 3: Error plot of calibration data.

1.2 Mechanism

For the 3D scanner, we had to build a pan-tilt mechanism to allow the infrared distance sensor to record distances from different yaws and pitches. This allows us to have enough data points to build a visualization of the object scanned.

To create the pan-tilt mechanism we started by creating sketches (Figure 4) of different pan-tilt mechanisms. We then used those sketches to start creating the CAD in SolidWorks. We started by creating the base of the pan-tilt (part 1) and our first iteration had the servo motor body centered in a cylinder (Figure 5).

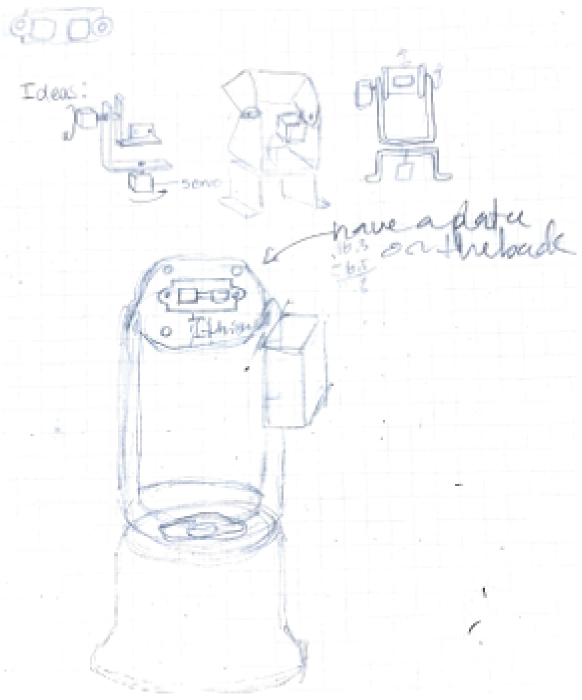

Figure 4: Sketches of different possible pan-tilt ideas to build.

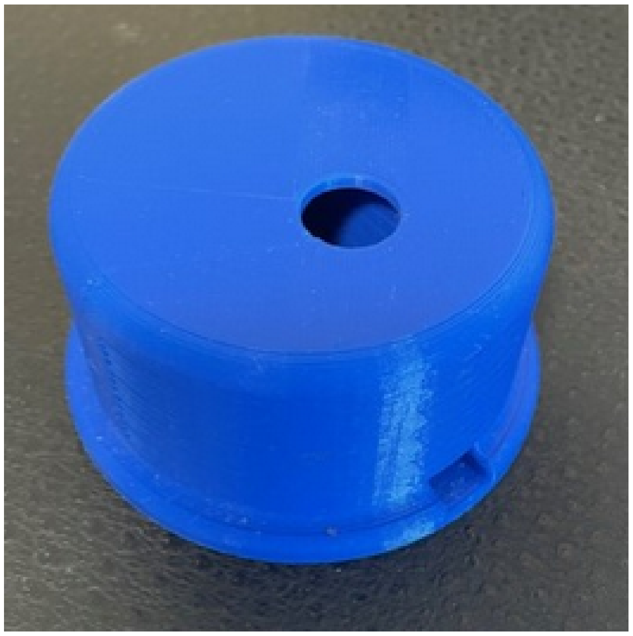

Figure 5: Part 1 with the servo motor centered and the shaft off-center (3D printed).

We realized after printing the piece that having the servo motor centered caused the shaft to be off-center meaning the next part of the pan-tilt that attached to the shaft would spin off-center. To fix this we centered the shaft of the servo motor (Figure 6).

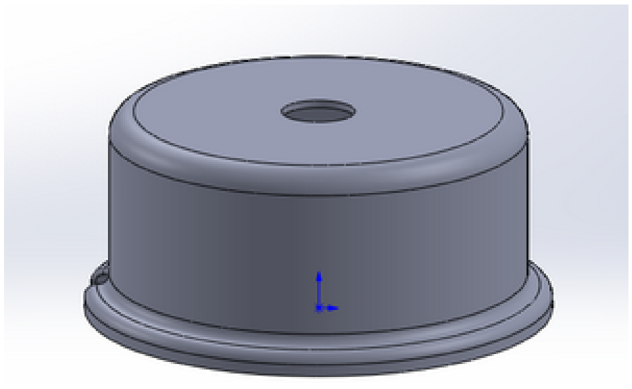

Figure 6: SolidWorks CAD model of part 1 with a centered shaft.

After creating the base we began designing the part that attaches to the shaft of the first servo motor (part 2). This part was made to have two arms that come up from the sides that have a hole on one side for a metal pin and a hole on the other side for the shaft of the servo motor. These would make it able to be able to hold and have the sensor-holder (part 3) rotate (pitch). We also added a motor case on one side of the part to better hold the second servo motor and have a place to screw it into the device. The first iteration of this part was too bulky (Figure 7) and so we redesigned this part to be thinner and not have a fully circular base but only a section of a circle (Figure 8).

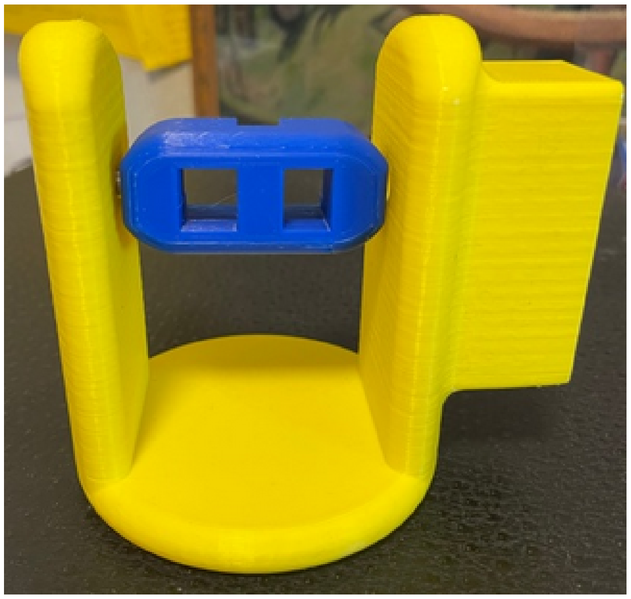

Figure 7: Part 2 edition 1 with thicker arms and a round base.

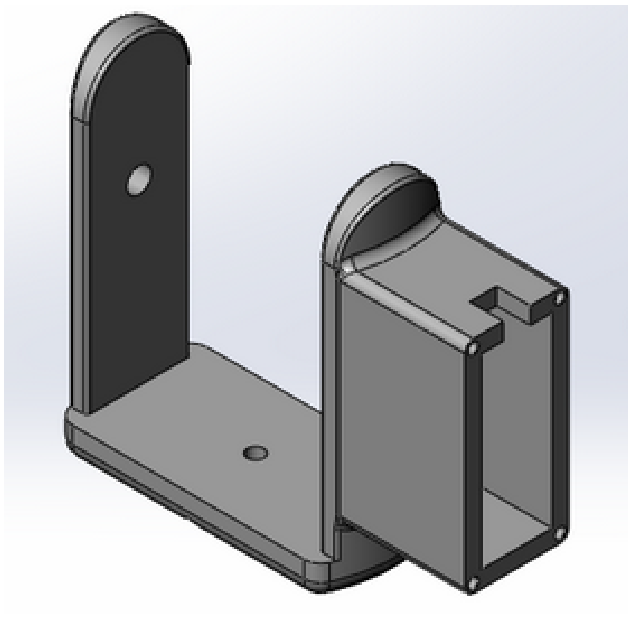

Figure 8: SolidWorks CAD model of part 2 with thinner arms and a cut-down base.

To then hold and rotate the sensor we created a sensor-holder that has a spot to screw the servo motor's horn to the sensor holder. It also has a spot to press-fit the metal pin through part 2 to part 3 (Figure 9).

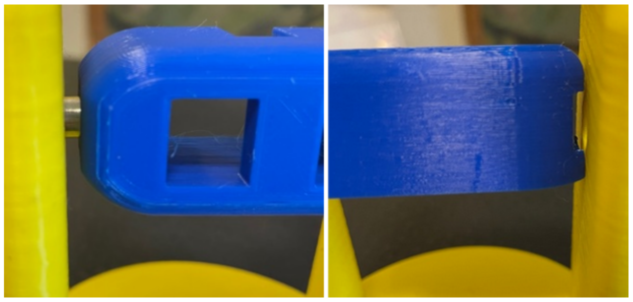

Figure 9: On left shows where the metal pin connects part 2 and part 3 together. On right shows where the motor horn would meet with part 3.

Due to the fact we redesigned part 2 we then had to redesign part 3 to be longer so it would fit better with the thinner part 2 (Figure 10).

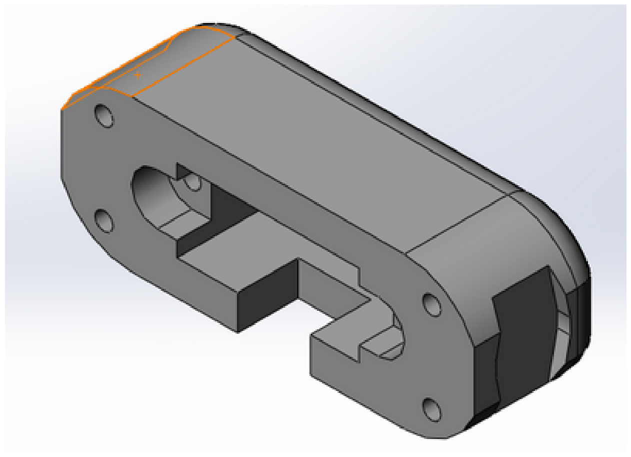

Figure 10: Final SolidWorks CAD model of part 3: the sensor holder.

For final touches, we created two covers, one for the sensor holder and one for part 2's servo motor holder (Figure 11).

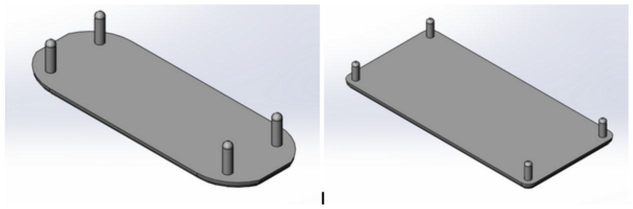

Figure 11: On the left is the sensor cover for part 3. On the right is the servo motor cover for part 2.

Throughout this process, we created an assembly that allowed us to make sure that parts we lining up correctly and fit together. We have Figure 12 to show the final assembly in SolidWorks of all of the parts mated together.

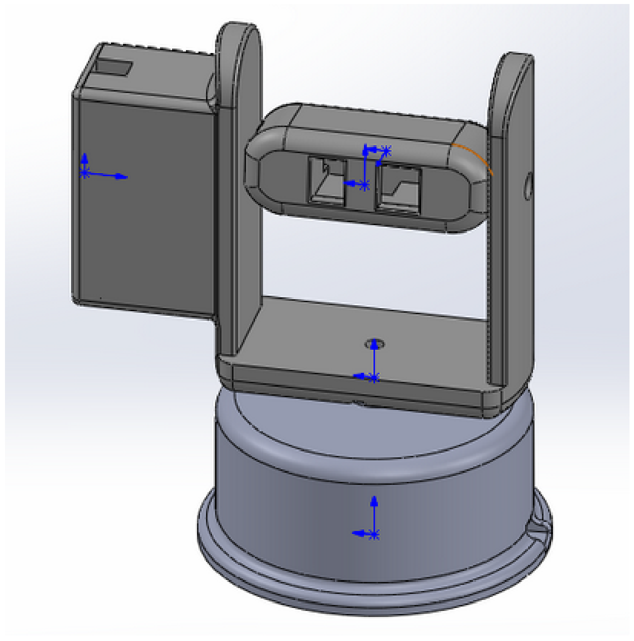

Figure 12: Shows the full assembly of parts 1,2,3 and the sensor and servo motor covers .

To fabricate the parts we used 3D printing because each of the parts are strangely shaped and hard to manufacture in traditional means. 3D printing also allowed us to quickly test out parts and find our mistakes faster.

1.3 Code

1.3.1 Software Architecture

Before we even opened our computers to start coding, we all sat in front of a whiteboard and created a system diagram for the software architecture as shown in Figure 13.

During the discussion of our software architecture, we decided we wanted to do as much processing in python as possible and have the Arduino act only as a bridge between Python and the physical components of the scanner. If we compare our scanner to a human body, then the python code would be like the brain, the Arduino would be the spinal cord/sensor neurons, the infrared distance sensor would be the eyes, and the servos would be the muscles. The python code would gather information from the user (like image resolution) and generate all the servo positions for each point in the final image. Then it would individually send those servo positions to the Arduino, wait for the Arduino to update the servos, and finally ask the Arduino to collect data from the IR distance sensor and send it back to Python using serial communication. Python would repeat these steps for every servo position. Once all the data has been collected, python would convert the raw data into meaningful measurements using a calibration curve we created and graph all the scanned points on a 3 axis plot which it presents to the user. To make it easier to code all the different tasks Python needed to be responsible for, we divided the tasks into three main groups: setup, communication, and visualization. If we expand on our human body analogy, the different parts of the Python code could be thought of as the different parts of the brain where each part specializes in a different function and all need to work together for the system to function.

In reflection, we found it incredibly helpful to have defined the system architecture so early in the project. Aside from it allowing us to divide the coding work up between the three of us

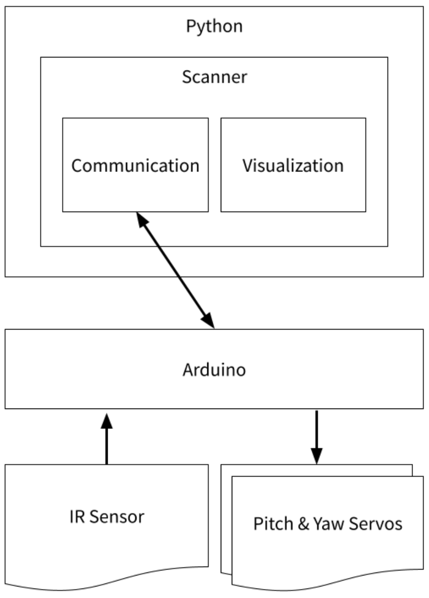

Figure 13: Software architecture diagram.

and work on it in parallel, it made the integration process much less painful because the inputs, outputs, and responsibilities of each code section were clearly defined.

1.3.2 Implementing Arduino-Python Communication

Once we started coding, we decided to start by tackling the Arduino-Python communication. We started implementing this first because it felt like the most mission-critical piece to us.

We start by just trying to get Python to send a message to the Arduino and have the Arduino blink a light when it received a message.

```python
import serial
import time

arduino = serial.Serial(port="/dev/ttyACM4", baudrate=115200, timeout=5)

while True:
    for _ in range(25):
        arduino.write(bytes("TEST MESSAGE", 'utf-8'))
    time.sleep(3)
```

```c
void setup() {
    /* The setup function starts serial communcation, sets up a debug LED.
     */
    Serial.begin(115200);
    Serial.flush();
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LOW);
}

void loop() {
    /* The main loop reads incoming serial messages. If a message is detected in
   the
     * input buffer, an LED is blinked on.
     */
    if (Serial.available() > 0) {
        digitalWrite(LED_BUILTIN, HIGH);
        delay(100)
    } else {
        digitalWrite(LED_BUILTIN, LOW);
    }
}
```

Once we were successfully receiving a message from the Arduino, we wrote code so the Arduino could respond to these messages.

```
import serial
import time

arduino = serial.Serial(port="/dev/ttyACM4", baudrate=115200, timeout=5)

while True:
    for _ in range(25):
        arduino.write(bytes("TEST MESSAGE", 'utf-8'))
        print(arduino.read_until(bytes("\r\n", 'utf-8')).decode("utf-8"))
    time.sleep(3)
```

```
void sendMessage(String messageData) {
    /* Send a message to python using serial
     *
     * Parameters:
     * messageData (String): the message data to be sent.
     */
    Serial.println(messageData);
}

void setup() {
    /* The setup function starts serial communcation, sets up a debug LED.
     */
    Serial.begin(115200);
    Serial.flush();
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LOW);
}

void loop() {
    /* The main loop reads incoming serial messages. If a message is detected in
    the
     * input buffer, an LED is blinked on.
     */
    if (Serial.available() > 0) {
        digitalWrite(LED_BUILTIN, HIGH);
        String message = Serial.readStringUntil('\n');
        sendMessage(message);
    } else {
        digitalWrite(LED_BUILTIN, LOW);
    }
}
```

Getting to this point was super exciting for our team. This was the core mechanism behind our system and having it working felt great. At this point, we spent some time trying to implement some best practices when it comes to writing in Python. We grouped our code into reusable classes and functions, wrote comments, utilized the `logging` Python package to add some debug statements to our code, and added `raise` statements to catch if any unexpected values were trying to be sent or received over serial.

We also spent some time formalizing the message protocol we wanted to use. We decided that every message was going to start with a message type. The type would be represented by one character that indicated the kind of data that was going to follow. `'M'` meant that the message contained servo positions, `'S'` meant the message was trying to request IR distance sensor data, and `'T'` was a test message with arbitrary data to check serial communication was working properly. Each message would also end with a specific series of end-of-message characters, specifically, `'\r\n'`. We choose these characters because they are automatically printed to the serial bus by the Arduino when using the `Serial.println()` function.

At this point our code looked something like this:

```python
import time
import logging
import logging.config
import serial

logger_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'logging.conf')
logging.config.fileConfig(logger_path, disable_existing_loggers=False)
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

class Communication():
    """Infrastructure for serial communication with the Arduino."""

    EOM = "\r\n"
    SEND_MESSAGE_TYPES = ["M", "S", "T"]
    RECIEVE_MESSAGE_TYPES = ["M", "S","T"]

    def __init__(self, baudrate=115200, port="/dev/ttyACM4"):
        """Instantiate a Communication object.

        Parameters:
            baudrate (int): the baudrate of the serial port. This should match the
                baudrate set on the Arduino.
            port (str): the serial port where the Arduino is connected.
        """

        logger.info("Starting communication with Arduino...")
        self.arduino = serial.Serial(port=port, baudrate=baudrate, timeout=5)
        time.sleep(5)
        self.arduino.flush()
        response = self.send_recieve("T","12345")
        if response["data"] == "12345":
            logger.info("Serial communication ready!")
        else:
            logger.info(f"ERROR: {response}")


    def send(self, message_type, message_data):
        """Send data to the arduino.

        Parameters:
            message_type (str): the message type. Must be a type listed in the send
                message type list stored in the communications class.
            message_data (str): data to be sent over the serial bus to the Arduino.

        Raises:
            ValueError: when the message type does not match a message type listed
                in the SEND_MESSAGE_TYPES list.
        """
        if message_type not in self.SEND_MESSAGE_TYPES:
            raise ValueError(f"Incorrect message type: {message_type}")
        message = f"{message_type}{message_data}{self.EOM}"
        self.arduino.write(bytes(message, 'utf-8'))


    def receive(self):
        """Receive data from the arduino.

        Returns:
            (dict): contains message type, processed data, and error value.
        """
        raw_data = self.arduino.read_until(bytes(self.EOM, 'utf-8')).decode("utf-8")
        try:
            message_type = raw_data[0]
        except IndexError:
            return {
                "message_type": "ERROR",
                "data": "EMPTY",
                "error": 2,
            }
        if message_type in self.RECIEVE_MESSAGE_TYPES:
            data = raw_data.split("\n")[0].split("\r")[0][1:]
            return {
                "message_type": message_type,
                "data": data,
                "error": 0,
            }
        else:
            return {
                "message_type": "ERROR",
                "data": raw_data,
                "error": 1,
            }

    def send_recieve(self, message_type, message_data):
        """Send data and expect a response.

        Parameters:
            message_type (str): the message type. Must be a type listed in the send
                message type list stored in the communications class.
            message_data (str): data to be sent over the serial bus to the Arduino.
        Returns:
            (dict): contains message type, processed data, and error value.
        """
        self.send(message_type, message_data)
        response = self.receive()
        if response["message_type"] != message_type:
            raise ValueError(f"Unexpected message type. Message: {response}")
        return response
```

```
#define TYPE_INDEX 0
#define DATA_INDEX 1

void sendMessage(char messageType, String messageData) {
    /* Send a message to python using serial
     *
     * Parameters:
     *  messageType (char): the type of the message. Examples include, 'M' for
     *      motor response, 'S' for sensor response, or 'T' for test.
     * messageData (String): the message data to be sent.
     */
    Serial.print(messageType);
    Serial.println(messageData);
}

void setup() {
    /* The setup function starts serial communcation, sets up a debug LED.
     */
    Serial.begin(115200);
    Serial.flush();
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LOW);
}

void loop() {
    /* The main loop reads incoming serial messages. If a message is detected in
    the
     * input buffer, an LED is blinked on.
     */
    if (Serial.available() > 0) {
        digitalWrite(LED_BUILTIN, HIGH);
        String message = Serial.readStringUntil('\n');
        char messageType = message.charAt(TYPE_INDEX);
        String data = message.substring(DATA_INDEX);
        sendMessage(message);
    } else {
        digitalWrite(LED_BUILTIN, LOW);
    }
}
```

1.3.3 Controlling Servos and Getting IR Distance Data

Our next step was to write Arduino code that would either set servo positions or read the IR distance sensor. However, before we could start messing around with the sensor and motors, we needed to build the infrastructure that was going to decipher and execute the commands from Python. This required manipulating the raw message string being received by the Arduino from the serial bus. Specifically, we wanted to isolate the message type and the message data. Once the Arduino knows the message type, it can easily decide how to handle the rest of the message and generate a response to send back. We implemented this using the `charAt()` function and a `switch` statement. The code is shown below:

```
#define TYPE_INDEX 0
#define DATA_INDEX 1

void sendMessage(char messageType, String messageData) {
    /* Send a message to python using serial
     *
     * Parameters:
     *  messageType (char): the type of the message. Examples include, 'M' for
     *      motor response, 'S' for sensor response, or 'T' for test.
     * messageData (String): the message data to be sent.
     */
    Serial.print(messageType);
    Serial.println(messageData);
}

void analyzeMessage(String message) {
    /* Decode serial messages from python and execute corresponding
     * response function and returns the response.
     *
     * Parameters:
     *  message (String): the raw message sent over serial from python.
     */
    char messageType = message.charAt(TYPE_INDEX);
    String data = message.substring(DATA_INDEX);
    String response;
    switch (messageType) {
        case 'M':
            response = "MOTOR";
            sendMessage('M', response);
            break;
        case 'S':
            response = "SENSOR";
            sendMessage('S', response);
            break;
        case 'T':
            sendMessage('T', "12345");
            break;
    }
}

void setup() {
    /* The setup function starts serial communcation, sets up a debug LED.
     */
    Serial.begin(115200);
    Serial.flush();
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LOW);
}

void loop() {
    /* The main loop reads incoming serial messages and sends them
     * to the analyzeMessage function where they are decoded and the
     * proper actions are taken.
     */
    if (Serial.available() > 0) {
        digitalWrite(LED_BUILTIN, HIGH);
        String message = Serial.readStringUntil('\n');
        analyzeMessage(message);
    } else {
        digitalWrite(LED_BUILTIN, LOW);
    }
}
```

Now that the infrastructure was in place, we were ready to write functions to interact with servos and sensor. For the servo code, we relied heavily on the Servo C++ package. It includes functions like `servo.attach(pin)` which set up a PWM compatible pin so it can provide a PWM signal to the servos and `servo.write(angle)` which turns the shaft so it is `angle` degrees away from the starting/zero point. For the IR sensor, we just used the `analogRead()` function to get raw data from the sensor. At this point, we did not worry about applying the calibration curve equation for filtering the data because we felt those were all things that could be done in Python. The new Arduino code was as follows:

```
#define TYPE_INDEX 0
#define DATA_INDEX 1
#define PITCH_SERVO_PIN 9
#define YAW_SERVO_PIN 10
#define DISTANCE_SENSOR_PIN 0
#include <Servo.h>

Servo pitchServo;
Servo yawServo;

int current_pitch;
int current_yaw;

bool setServoPosition(int pitch, int yaw) {
    /* Set the servos to the given pitch and yaw.
     *
     * Parameters:
     *  pitch (int): PWM value for setting the angle of the pitch motor shaft.
     *  yaw (int): PWM value for setting the angle of the yaw motor shaft.
     *
     * Returns:
     *  (bool): success of motor update
     */
    yawServo.write(yaw);
    pitchServo.write(pitch);
    return true;
}

String respondSensorMessage(String message) {
    /* Decodes sensor command messages and generates response.
     *
     * Parameters:
     *  message (String): Data from sensor message sent over serial from python.
     *
     * Returns:
     *  (String): message to be sent as a response
     */
    String response = "";
    int distance;
    distance = analogRead(DISTANCE_SENSOR_PIN);
    return String(distance);
}

String respondServoMessage(String message) {
    /* Decodes motor command messages and generates response.
     *
     * Parameters:
     *  message (String): Data from motor message sent over serial from python.
     *
     * Returns:
     *  (String): message to be sent as a response
     */
    int pitch = message.substring(0, 3).toInt();
    int yaw = message.substring(4, 7).toInt();
    setServoPosition(pitch, yaw);
    return true;
}

void sendMessage(char messageType, String messageData) {
    /* Send a message to python using serial
     *
     * Parameters:
     *  messageType (char): the type of the message. Examples include, 'M' for
     *      motor response, 'S' for sensor response, or 'T' for test.
     * messageData (String): the message data to be sent.
     */
    Serial.print(messageType);
    Serial.println(messageData);
}

void analyzeMessage(String message) {
    /* Decode serial messages from python and execute corresponding
     * response function and returns the response.
     *
     * Parameters:
     *  message (String): the raw message sent over serial from python.
     */
    char messageType = message.charAt(TYPE_INDEX);
    String data = message.substring(DATA_INDEX);
    String response;
    switch (messageType) {
        case 'M':
            response = respondServoMessage(data);
            sendMessage('M', response);
            break;
        case 'S':
            response = respondSensorMessage(data);
            sendMessage('S', response);
            break;
        case 'T':
            sendMessage('T', "12345");
            break;
    }
}

void setup() {
    /* The setup function starts serial communcation, sets up a debug LED, and
     * assigns PWM pins to servos.
     */
    Serial.begin(115200);
    Serial.flush();
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LOW);

    pitchServo.attach(PITCH_SERVO_PIN);
    yawServo.attach(YAW_SERVO_PIN);
    current_pitch = 0;
    current_yaw = 0;
    setServoPosition(current_pitch, current_yaw);
}

void loop() {
    /* The main loop reads incoming serial messages and sends them
     * to the analyzeMessage function where they are decoded and the
     * proper actions are taken.
     */
    if (Serial.available() > 0) {
        digitalWrite(LED_BUILTIN, HIGH);
        String message = Serial.readStringUntil('\n');
        analyzeMessage(message);
    } else {
        digitalWrite(LED_BUILTIN, LOW);
    }
}
```

At this point, our Arduino code had reached its minimum viable product. It could read commands from Python, execute those commands by setting the angle of the servos or reading sensor data and then send a response with the relevant data. However, while the basic functionality was great, there was still more work that had to be done. We'll discuss some of these changes in the filtering section.

1.3.4 Scanning a Line

At this point, we felt like we should try running a more rigorous test to make sure everything we had done so far was working. We decided that we wanted to try sweeping a yaw value across one pitch value and observe the outcome. We set up the scanner in front of a wall so that we knew exactly what to expect: a straight line. But before we were able to perform the test, we needed to write the Python infrastructure that would allow us to sweep across points, get sensor data, and display visuals.

Our first step was to write the `Scanner` class which contains functions that allow it to send motor position requests and sensor data requests to the Arduino. These functions also use the calibration data we collected earlier to translate the raw data into meaningful data and vice-versa. For example, it's intuitive to want to sweep over values centered on zero however when the scanner was pointing perfectly forward, it was actually at `(150, 90)` which means that we will need to offset all of the servo positions so that the angles sent to the Arduino are actual angles rather than relative angles to the scanners `(0,0)` position. For the sensor data, we used the equation we calculated earlier in the report to convert the raw sensor data into inches.

```python
import time
import logging
import logging.config
import os

import numpy as np
from scanviz.communication import Communication

logger_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'logging.
    conf')
logging.config.fileConfig(logger_path, disable_existing_loggers=False)
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

class Scanner():
    """API for interfacing with the scanner"""

    def __init__(self):
        self.comms = Communication()

    def set_position(self, pitch, yaw):
        """Send a message to the Arduino to set the pitch and roll of the Scanner.

        Parameters:
            pitch (float): Representing the desired pitch angle.
            yaw (float): Representing the desired yaw angle.
        """
        adjusted_pitch = pitch + 150
        adjusted_yaw = -yaw + 90
        if adjusted_pitch > 180 or adjusted_yaw > 180:
            raise ValueError("Cannot send motor angles greater than 180 deg.")
        logger.debug(
            "Setting servo positions to (pitch)"
            f" {int(round(adjusted_pitch))}, (yaw) {int(round(adjusted_yaw))}."
        )
        message = f"{int(round(adjusted_pitch)):03d}+{int(round(adjusted_yaw)):03d}"
        response = self.comms.send_recieve("M", message)
        if int(response["data"]) == 0:
            raise ValueError("Servo did not respond. Stopping program.")
        else:
            time.sleep(int(response["data"])/10)
        logger.debug("Servo positions have been set.")

    def get_distance(self):
        """Send a message to Arduino to send three distance measurements over serial.

        Returns:
            (float): Calibrated output from distance sensor in inches.
        """
        logger.debug("Getting measured sensor distance.")
        response = self.comms.send_recieve("S", "GET")
        logger.debug(f"Information recieved: {response['data']}")
        raw_data = response["data"].split(",")
        output = []
        for data in raw_data:
            output += [48.7 - (0.15 * int(data)) + (0.000134 * (int(data)**2))]
        return output
```

Once that was done, we wrote the `Visualization` class which contains functions that allow it to generate a mesh of all the points we want to scan and allow it to graph the collected data points using `matplotlib`. At this point, we were only trying to scan a line so we only needed to sweep across one variable while keeping the other one constant. However, rather than creating one variable for the dimension we were sweeping across, we still defined both dimensions as variables. For the pitch, we assigned it to be a constant while the other was a list of points. We pretended that the constant variable was a list of points and wrote the rest of the code to try and accommodate that. This included figuring out the math to translate the spherical coordinates being produced by the scanner from the servo positions and the measured distance into cartesian coordinates rather than just plotting the polar data we were generating. While this was more work at the moment, implementing the math at this point made it so much easier to jump from scanning just a line to a mesh of points. Below is the code we wrote:

```python
import logging
import logging.config
import os

import numpy as np
import scipy.signal as scipy
import matplotlib.pyplot as plt
from matplotlib import ticker, cm

logger_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'logging.
    conf')
logging.config.fileConfig(logger_path, disable_existing_loggers=False)
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

class Visualization():
    """Tools for visualizing scanner data."""
    def __init__(self):
        """Instantiate visualization object"""
        pass

    def generate_mesh(self, resolution):
        """Generate pitch & yaw angles for scanning

        Parameters:
            resolution (int): Number of points to scan
        Returns:
            (list): List of lists outlining all the yaw values to scan.
            (int): Integer representing the pre-set pitch for all measurements.
        """
        if resolution > 60:
            raise ValueError("Resolution can't be higher than 180")
        pitch = np.linspace(-30, 30, resolution)
        yaw = [150]
        pitch_mesh, yaw_mesh = np.meshgrid(pitch, yaw)
        return pitch_mesh, yaw_mesh

    def create_viz(self, pitch, yaw, radius):
        """Create a matplotlib graph to visualize the data."""
        pitch_rad = np.deg2rad(pitch)
        yaw_rad = np.deg2rad(yaw)
        x = radius * np.sin(yaw_rad) * np.cos(pitch_rad)
        y = radius * np.sin(pitch_rad)
        z = radius * np.cos(pitch_rad) * np.cos(yaw_rad)

        fig = plt.figure(figsize=plt.figaspect(2.))
        fig.suptitle('3D Scan')

        # 3D Render
        ax_3D = fig.add_subplot(projection='3d')

        surf = ax_3D.scatter(
            x,
            z,
            y,
        )

        fig.tight_layout()
        plt.show()
```

Finally, we wrote a `sweep()` function which lives in the `Scanner` class and sweeps through all the points generated by the `Visualization` class, collects data at each point using the rest of the `Scanner` class and `Communication` class, and then sends that data to the `create_viz()` function to produce a final output image. The code for the `sweep()` function is shown below.

```
def sweep(self, resolution):
    """Sweep over a set of pitch and yaw values and collect distance data.

    Parameters:
        resolution (int): Determines the number of measurements taken by the
            sensor. The number of measurements equals (2*(resolution**2)).
    """
    logger.info(f"Scan Beginning. Estimated Time: {((resolution**2)*1.5)/60}")
    pitch_mesh, yaw_mesh = self.viz.generate_mesh(resolution)
    radius_mesh = []
    for row in range(len(pitch_mesh)):
        radius_mesh_row = []
        for col in range(len(pitch_mesh[row])):
            self.set_position(pitch_mesh[row][col], yaw_mesh[row][col])
            radius_mesh_row += [self.get_distance()]
        radius_mesh += [radius_mesh_row]
    self.viz.create_viz(pitch_mesh, yaw_mesh, radius_mesh)
```

Once all the code had been written we ran the function and generated our first scan! The results are shown in Figure 14.

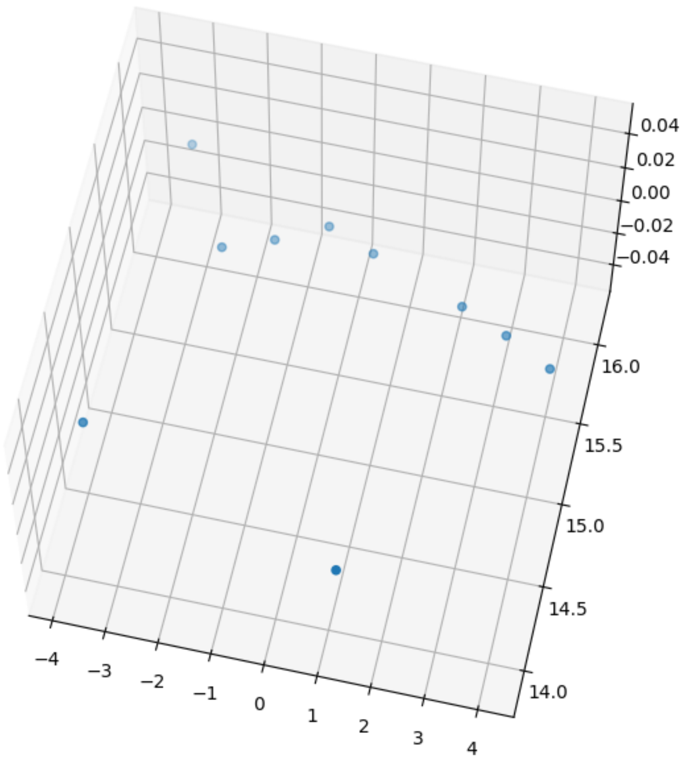

Figure 14: Single line from a scan of a wall.

While there was some noise and random outliers, for the most part, it was clear we had scanned a flat wall. This test made us realize that we were going to need to apply a filter to the data to help remove noise and reduce outliers.

1.3.5 Scanning a Mesh

Thanks to the extra work we did while taking a scan of just a line, we were able to easily make the jump to scanning a 2D grid of points very easy. All we had to do was change one line of code from defining the pitch as a single number to defining it as a list of points as shown below:

```python
pitch = np.linspace(-30, 30, resolution)
```

At this point, we had fully completed the minimum-viable product we were hoping to achieve. We set up our letter and took our first ten by ten image scan. The results are shown in Figure 15

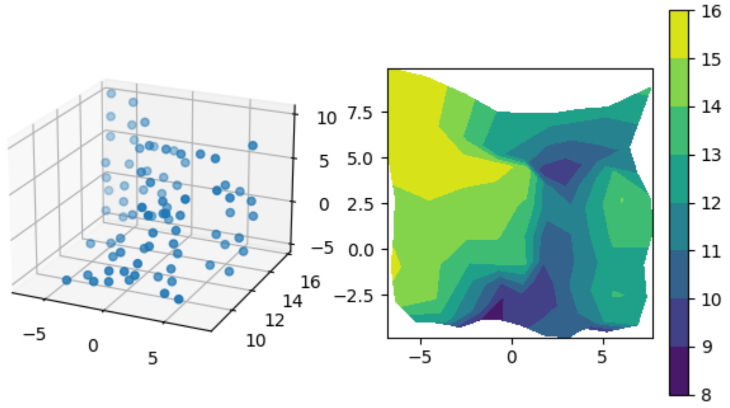

Figure 15: A low resolution (10 point by 10 point) scan of the letter J against a wall.

1.3.6 Filtering

Looking at our first scan, it was mostly clear that we had scanned the letter J however there was still noise and random outliers. To fix these problems we did two things. The first one was to modify the Arduino data collection code to collect multiple scans of the same location rather than just one. This would allow the python code to take the average of the many scans and hopefully reduce the number of outliers in the final image. The second one was to apply a convolution filter to the final image using the `scipy` package. A convolution filter takes a matrix smaller than the original image and slides it across the image. For each grouping of points it slides over, it multiplies the group with the matrix and adds all the values which will represent one pixel in the filtered image. Ultimately, this filter would reduce the accuracy of the image, but also would help reduce any noise in the image. After all these changes were made, we had the final iteration of our code for the project.

```python
import time
import logging
import logging.config
import serial

import numpy as np
import scipy.signal as scipy
import matplotlib.pyplot as plt
from matplotlib import ticker, cm

logger_path = os.path.join(os.path.dirname(os.path.abspath(__file__)), 'logging.conf')
logging.config.fileConfig(logger_path, disable_existing_loggers=False)
logger = logging.getLogger(__name__)
logger.setLevel(logging.DEBUG)

class Communication():
    """Infrastructure for serial communication with the Arduino."""

    EOM = "\r\n"
    SEND_MESSAGE_TYPES = ["M", "S", "T"]
    RECIEVE_MESSAGE_TYPES = ["M", "S","T"]

    def __init__(self, baudrate=115200, port="/dev/ttyACM4"):
        """Instantiate a Communication object.

        Parameters:
            baudrate (int): the baudrate of the serial port. This should match the
                baudrate set on the Arduino.
            port (str): the serial port where the Arduino is connected.
        """

        logger.info("Starting communication with Arduino...")
        self.arduino = serial.Serial(port=port, baudrate=baudrate, timeout=5)
        time.sleep(5)
        self.arduino.flush()
        response = self.send_recieve("T","12345")
        if response["data"] == "12345":
            logger.info("Serial communication ready!")
        else:
            logger.info(f"ERROR: {response}")


    def send(self, message_type, message_data):
        """Send data to the arduino.

        Parameters:
            message_type (str): the message type. Must be a type listed in the send
                message type list stored in the communications class.
            message_data (str): data to be sent over the serial bus to the Arduino.

        Raises:
            ValueError: when the message type does not match a message type listed
                in the SEND_MESSAGE_TYPES list.
        """
        if message_type not in self.SEND_MESSAGE_TYPES:
            raise ValueError(f"Incorrect message type: {message_type}")
        message = f"{message_type}{message_data}{self.EOM}"
        self.arduino.write(bytes(message, 'utf-8'))


    def receive(self):
        """Receive data from the arduino.

        Returns:
            (dict): contains message type, processed data, and error value.
        """
        raw_data = self.arduino.read_until(bytes(self.EOM, 'utf-8')).decode("utf-8
    ")
        try:
            message_type = raw_data[0]
        except IndexError:
            return {
                "message_type": "ERROR",
                "data": "EMPTY",
                "error": 2,
            }
        if message_type in self.RECIEVE_MESSAGE_TYPES:
            data = raw_data.split("\n")[0].split("\r")[0][1:]
            return {
                "message_type": message_type,
                "data": data,
                "error": 0,
            }
        else:
            return {
                "message_type": "ERROR",
                "data": raw_data,
                "error": 1,
            }

    def send_recieve(self, message_type, message_data):
        """Send data and expect a response.

        Parameters:
            message_type (str): the message type. Must be a type listed in the
    send
                message type list stored in the communications class.
            message_data (str): data to be sent over the serial bus to the Arduino
    .
        Returns:
            (dict): contains message type, processed data, and error value.
        """
        self.send(message_type, message_data)
        response = self.receive()
        if response["message_type"] != message_type:
            raise ValueError(f"Unexpected message type. Message: {response}")
        return response

class Scanner():
    """API for interfacing with the scanner"""

    def __init__(self):
        self.comms = Communication()
        self.viz = Visualization()

    def set_position(self, pitch, yaw):
        """Send a message to the Arduino to set the pitch and roll of the Scanner.

        Parameters:
            pitch (float): Representing the desired pitch angle.
            yaw (float): Representing the desired yaw angle.
```

```python
        """
        adjusted_pitch = pitch + 150
        adjusted_yaw = -yaw + 90
        if adjusted_pitch > 180 or adjusted_yaw > 180:
            raise ValueError("Cannot send motor angles greater than 180 deg.")
        logger.debug(
            "Setting servo positions to (pitch)"
            f" {int(round(adjusted_pitch))}, (yaw) {int(round(adjusted_yaw))}."
        )
        message = f"{int(round(adjusted_pitch)):03d}+{int(round(adjusted_yaw)):03d}"
        response = self.comms.send_recieve("M", message)
        if int(response["data"]) == 0:
            raise ValueError("Servo did not respond. Stopping program.")
        else:
            time.sleep(int(response["data"])/10)
        logger.debug("Servo positions have been set.")

    def get_distance(self):
        """Send a message to Arduino to send three distance measurements over serial.

        Returns:
            (float): Calibrated output from distance sensor in inches.
        """
        logger.debug("Getting measured sensor distance.")
        response = self.comms.send_recieve("S", "GET")
        logger.debug(f"Information recieved: {response['data']}")
        raw_data = response["data"].split(",")
        output = []
        for data in raw_data:
            output += [48.7 - (0.15 * int(data)) + (0.000134 * (int(data)**2))]
        logger.debug(f"Measured Value: {sum(output)/len(output)}")
        return sum(output)/len(output)

    def sweep(self, resolution):
        """Sweep over a set of pitch and yaw values and collect distance data.

        Parameters:
            resolution (int): Determines the number of measurements taken by the
                sensor. The number of measurements equals (2*(resolution**2)).
        """
        logger.info(f"Scan Beginning. Estimated Time: {((resolution**2)*1.5)/60}")
        pitch_mesh, yaw_mesh = self.viz.generate_mesh(resolution)
        radius_mesh = []
        for row in range(len(pitch_mesh)):
            radius_mesh_row = []
            for col in range(len(pitch_mesh[row])):
                self.set_position(pitch_mesh[row][col], yaw_mesh[row][col])
                radius_mesh_row += [self.get_distance()]
            radius_mesh += [radius_mesh_row]
        self.viz.create_viz(pitch_mesh, yaw_mesh, radius_mesh)

class Visualization():
    """Tools for visualizing scanner data."""
    def __init__(self):
        """Instantiate visualization object"""
        pass

    def generate_mesh(self, resolution):
        """Generate pitch & yaw angles for scanning

        Parameters:
            resolution (int): Number of points to scan
        Returns:
            (list): List of lists outlining all the pitch values to scan.
            (list): List of lists outlining all the yaw values to scan.
        """
        if resolution > 60:
            raise ValueError("Resolution can't be higher than 180")
        pitch = np.linspace(-30, 30, resolution)
        yaw = np.linspace(-30, 30, resolution)
        pitch_mesh, yaw_mesh = np.meshgrid(pitch, yaw)
        return pitch_mesh, yaw_mesh

    def _moving_avg(self, data):
        window = np.ones((2, 2)) / 4
        return scipy.convolve2d(data, window, 'valid')

    def create_viz(self, pitch, yaw, radius):
        """Create a matplotlib graph to visualize the data."""
        smooth_radius = self._moving_avg(radius)
        pitch_rad = [row[1:] for row in np.deg2rad(pitch)[1:,:]]
        yaw_rad = [row[1:] for row in np.deg2rad(yaw)[1:,:]]
        # Unfiltered Output
        # smooth_radius = radius
        # pitch_rad = np.deg2rad(pitch)
        # yaw_rad = np.deg2rad(yaw)
        x = smooth_radius * np.sin(yaw_rad) * np.cos(pitch_rad)
        y = smooth_radius * np.sin(pitch_rad)
        z = smooth_radius * np.cos(pitch_rad) * np.cos(yaw_rad)

        fig = plt.figure(figsize=plt.figaspect(2.))
        fig.suptitle('3D Scan')

        # 2D Contour
        ax_2D = fig.add_subplot(1, 2, 2)

        contour = ax_2D.contourf(x, y, z)
        ax_2D.set_aspect('equal', 'box')
        fig.colorbar(contour, ax=ax_2D, shrink=0.5)

        # 3D Render
        ax_3D = fig.add_subplot(1, 2, 1, projection='3d')

        surf = ax_3D.scatter(
            x,
            z,
            y,
        )

        fig.tight_layout()
        plt.show()
```

```c
#define TYPE_INDEX 0
#define DATA_INDEX 1
#define PITCH_SERVO_PIN 9
#define YAW_SERVO_PIN 10
```

```
#define DISTANCE_SENSOR_PIN 0
#include <Servo.h>

Servo pitchServo;
Servo yawServo;

int current_pitch;
int current_yaw;

bool setServoPosition(int pitch, int yaw) {
    /* Set the servos to the given pitch and yaw.
     *
     * Parameters:
     *  pitch (int): PWM value for setting the angle of the pitch motor shaft.
     *  yaw (int): PWM value for setting the angle of the yaw motor shaft.
     *
     * Returns:
     *  (bool): success of motor update
     */
    yawServo.write(yaw);
    pitchServo.write(pitch);
    return true;
}

String respondSensorMessage(String message) {
    /* Decodes sensor command messages and generates response.
     *
     * Parameters:
     *  message (String): Data from sensor message sent over serial from python.
     *
     * Returns:
     *  (String): message to be sent as a response
     */
    String response = "";
    int distance;
    for (int i = 0; i <= 15; i++) {
        distance = analogRead(DISTANCE_SENSOR_PIN);
        response += String(distance, DEC) + ",";
    }
    return response.substring(0, response.length() - 1);
}

String respondServoMessage(String message) {
    /* Decodes motor command messages and generates response.
     *
     * Parameters:
     *  message (String): Data from motor message sent over serial from python.
     *
     * Returns:
     *  (String): message to be sent as a response
     */
    int pitch = message.substring(0, 3).toInt();
    int yaw = message.substring(4, 7).toInt();
    int waitTime = 15;
    setServoPosition(pitch, yaw);
    String response = String(waitTime, DEC);
    return response;
}

```

```
void sendMessage(char messageType, String messageData) {
    /* Send a message to python using serial
     *
     * Parameters:
     *  messageType (char): the type of the message. Examples include, 'M' for
     *      motor response, 'S' for sensor response, or 'T' for test.
     * messageData (String): the message data to be sent.
     */
    Serial.print(messageType);
    Serial.println(messageData);
}

void analyzeMessage(String message) {
    /* Decode serial messages from python and execute corresponding
     * response function and returns the response.
     *
     * Parameters:
     *  message (String): the raw message sent over serial from python.
     */
    char messageType = message.charAt(TYPE_INDEX);
    String data = message.substring(DATA_INDEX);
    String response;
    switch (messageType) {
        case 'M':
            response = respondServoMessage(data);
            sendMessage('M', response);
            break;
        case 'S':
            response = respondSensorMessage(data);
            sendMessage('S', response);
            break;
        case 'T':
            sendMessage('T', "12345");
            break;
    }
}

void setup() {
    /* The setup function starts serial communcation, sets up a debug LED, and
     * assigns PWM pins to servos.
     */
    Serial.begin(115200);
    Serial.flush();
    pinMode(LED_BUILTIN, OUTPUT);
    digitalWrite(LED_BUILTIN, LOW);

    pitchServo.attach(PITCH_SERVO_PIN);
    yawServo.attach(YAW_SERVO_PIN);
    current_pitch = 0;
    current_yaw = 0;
    setServoPosition(current_pitch, current_yaw);
}

void loop() {
    /* The main loop reads incoming serial messages and sends them
     * to the analyzeMessage function where they are decoded and the
     * proper actions are taken.
     */
    if (Serial.available() > 0) {
        digitalWrite(LED_BUILTIN, HIGH);
        String message = Serial.readStringUntil('\n');
        analyzeMessage(message);
    } else {
        digitalWrite(LED_BUILTIN, LOW);
    }
}
```

2 Results

Figure 16 shows the final scan we took which included filtering and averaging the data to create a clean image. We performed this scan at a higher resolution (roughly twice what we were using before). While the measurements are not very accurate due to small inaccuracies in the calibration, bad data read from the sensor, and the convolution filter; the shape of the letter is very clear which is ultimately what we wanted to achieve.

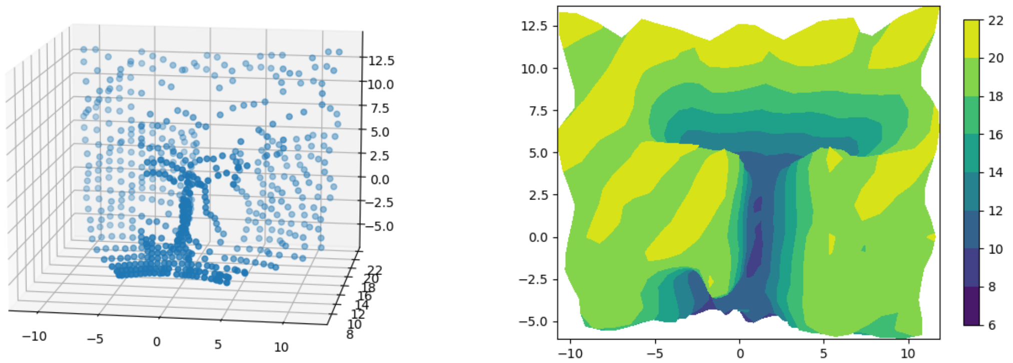

Figure 16: A high resolution (25 point by 25 point) scan of the letter J against a wall with filtering.

3 Analysis

We can now see the letter J displayed in the python visualization software that we made. There are some inconsistencies, and there are slopping artifacts caused by the convolution filter, but otherwise, the letter is very clear and well depicted. The mechanism itself was pretty smooth, at times it would wiggle but it was reliable for the most part. One way we could try to improve the image quality is to change the type of filtering we are performing on the data which alters the data less than the convolution filter while still removing the noise.

4 Conclusion and Reflection

During this project, we believe that we did a very good job of focusing on the integration aspect of the project rather than going our separate ways. We assisted each other to help learn things that we had wanted to learn and support each other. Everyone was fully aware of what was going on, and there was clear communication. In our first meeting, we made sure to lay down our expectations/goals/strengths/weaknesses for this project so that we were all on the same page. We prioritized the Python and Arduino communication and used VScode so that all team members were contributing/understanding the code. We had a very good scaffolding, which made it extremely easy to execute the actual scan once the framework was integrated. Overall we're super proud of the results!