Mini-Project 2: 3D Scanner

Ari Porad & Berwin Lan
Principles of Integrated Engineering, Section 5
Olin College of Engineering

September 30th, 2021

Abstract

In this mini-project, we built a 3D scanner using two servos in a pan-tilt configuration and a distance sensor. We then calibrated a model to map sensor readings to distance values, and measured the accuracy of that model. Finally, we used our mechanism to 3D scan a cardboard letter. By evaluating the scan data, we conclude that the 3D scanner is capable of scanning an object of known, well-defined geometry positioned within the sensor's operating range.

1 Introduction

This mini-project uses two servos in a pan-tilt configuration to sweep a sensor across a cardboard cutout of known, well-defined geometry (specifically, a letter A). Code was written in Arduino and Python to collect data, and MATLAB for data visualization (see Appendices). The sensor was calibrated developing a mathematical model to convert its readings to distances (see Section 2.3), and its accuracy was evaluated using a similar process (see Section 3.1). A single vertical sweep using the tilt servo was taken to verify the physical system and software (see Section 3.2). Finally, a full scan of the letter in two dimensions (both pan and tilt) was performed and a 3D point cloud was generated. In this final scan, the cardboard letter A can be clearly distinguished as a 2D plane (see Section 3.2).

2 Methods

2.1 Sensor Mounting

We mounted the sensor[1] on a pan-tilt mechanism, which was capable of rotating the sensor around two axes. Each axis was powered by a 180-degree hobby servo.[2]

We attached (using screws) the sensor to an L-bracket, which we glued onto the horn of the y-axis (tilt) servo. We attached that servo to the x-axis (pan) servo output through another, larger, L-bracket. The x-axis servo formed the base of our 3D scanner, and we mounted it inside a dedicated housing for support. We also taped this housing to the table for extra stability.

The axes of rotation, determined by the positions of the pan and tilt servos relative to the sensor, intersect the sensor. Thus, the sensor is located at $(0, 0)$, and rotation occurs about the x and y axes, which prevents translation during rotation. All physical components were 3D printed.

2.2 Wiring

We used an Arduino Uno R3 to control the servos and measure the sensor's value. The distance sensor was attached to the A0 pin on the Arduino, the tilt servo was attached to D9, and the pan servo was attached to D10 as pictured in Figure 1. Pins D9 and D10 were chosen because of their pulse width modulation capability, which allows precise control of servo motors. Data is sent from the Arduino to a Python program running on a laptop (and commands in the opposite direction) over the USB Serial connection.

Due to an unknown (but consistent) issue with this model of distance sensor, there is a regular pulse that slightly increases the output voltage of the sensor. If not accounted for, this introduces semi-random noise into the sensor readings. To avoid this issue, any time we read from the sensor, we always take the lowest of three $\mathrm{analogRead}$ calls. Because analo-

[1] A Sharp GP2Y0A02YK0F

[2] A Hobbyking HK15138

gRead takes a non-zero amount of time to run, at most two consecutive calls can fit within the duration of the pulse. This means that three consecutive analogRead calls will always have at least one reading not taken during the pulse, and that reading will always be the lowest (since the pulse increases the voltage). Consequently, taking the lowest of three consecutive readings will always result in a consistent, non-pulse-affected value.[3]

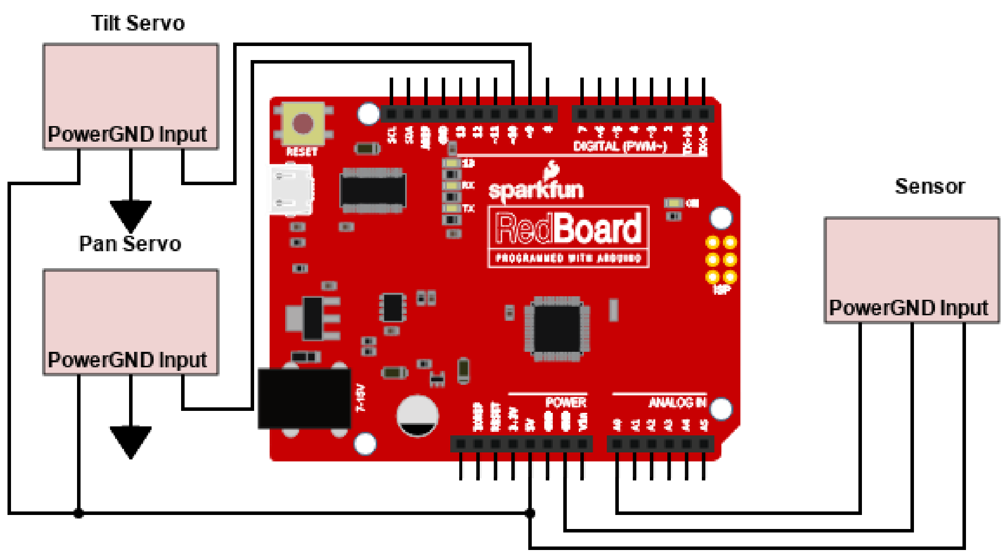


Figure 1: A wiring schematic of our system. The Arduino Uno is connected to the pan servo (D10), tilt servo (D9), and distance sensor (A0). The A0, D9, and D10 pins on the Arduino are input/output pins to write and collect data.

2.3 Calibration

To accurately measure distances, we first needed to calibrate our sensor to calculate a relationship between sensor value that the Arduino measures and physical distance.[4] To do this, we placed a flat vision target at various known distances from the sensor and measured the reading from the Arduino at each distance (see Figure 2a). We focused on calibrating the sensor up to 60 inches, as the sensor's data sheet[5] suggested it would be the most accurate in this range.

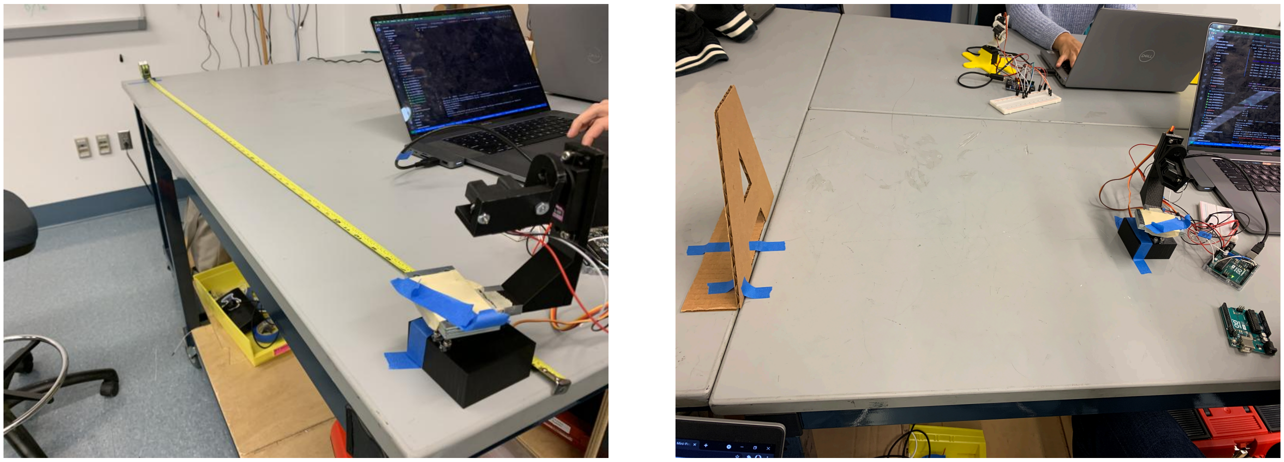
To simplify the process of collecting many distance/sensor reading pairs, we wrote a Python script which prompts the user to place the vision target at a series of distances,[6] and auto-

[3] Thank you to Professor Brad Minch for his tutorials which helped us debug this problem and guided us to the solution.

[4] We used an Arduino analog pin to measure the output voltage of the sensor, which returns a value between 0 (0V) and 1023 (5V).

[5] Provided by the PIE teaching team via Canvas.

[6] We collected a sample every 1.5 inches between 3 and 60 inches from the sensor, inclusive.



(a) Our calibration setup. We used a tape measure to place a flat cardboard vision target precise distances from the sensor, measuring the sensor's value each time.

(b) Our scanning setup. The 3D scanner is affixed to the tabletop and the cardboard letter vision target is placed at a distance of 20 inches in front of the sensor.

Figure 2: The physical setups for our 3D scanner, both for calibration and scanning.

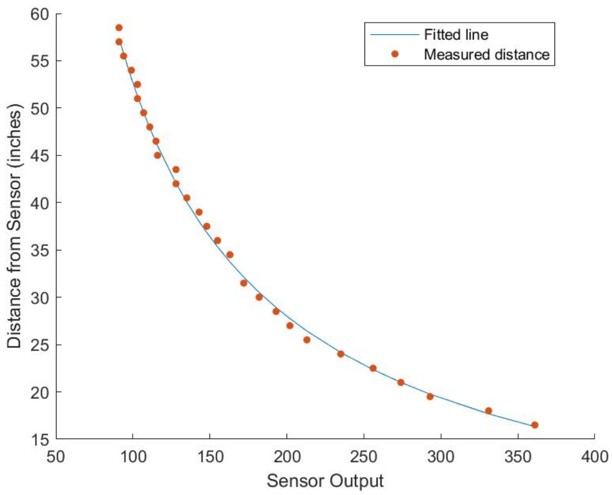
matically collects and tracks the sensor reading for each one. Source code for this script can be found in Appendix B).

Once we had collected a set of measured distance and sensor reading pairs, we filtered any outliers and used the MATLAB Curve Fitting Toolbox (Figure 3a) to generate a calibration curve which could map an arbitrary sensor reading to a predicted distance value. Since the distance is the sensor's independent variable, we generated this curve as a mapping from distance to sensor reading (Equation 1), and used algebra to get our sensor reading to distance mapping (Equation 2). We chose to use a power model[7] because it fit our data best. Our resulting calibration curve has an R^2 value of $R^2 = 0.9981$, indicating that it accurately represents the measured behavior of the sensor. For more discussion of the accuracy of our system, see Section 3.1.

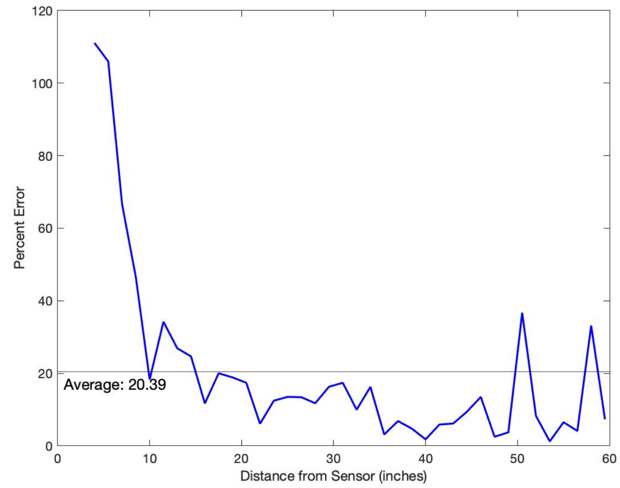
$$\text{SensorReading} = 7709 \times \text{Distance}^{-1.096} \tag{1}$$

$$\text{Distance} = \left(\frac{\text{SensorReading}}{7709}\right)^{\frac{1}{-1.096}} \tag{2}$$

[7] A model in the form of $f(x) = \alpha x^{\beta}$.



(a) Measured data and fitted calibration curve, showing the high correlation between our calibrated model and the sensor's measured behavior, leading to improved system accuracy ($R^2 = 0.9981$).



(b) Accuracy of our calibrated sensor, calculated by comparing computed to known distances. We calibrated the sensor for up to 60in, as suggested by the data sheet (Average error: 20.39%).

Figure 3: Analysis of our calibrated model, showing it as compared to measured data and the average error of the calibration curve.

3 Results

3.1 Accuracy

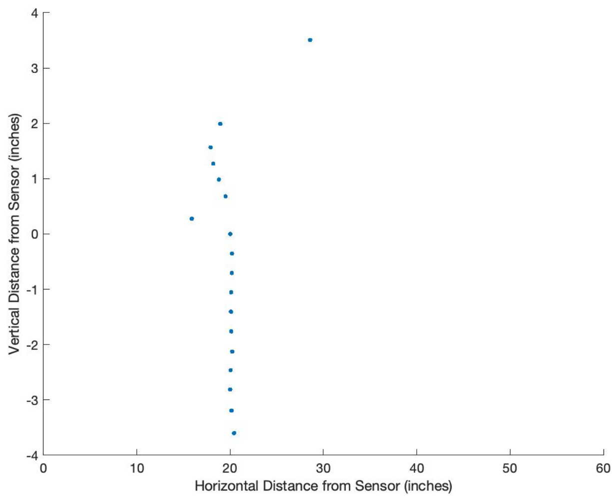
After calibrating our sensor, we tested the accuracy by placing a flat vision target at a series of known distances from the sensor, none of which were used during the calibration process.[8] We used the same data collection script we used for calibration for this process, but did not re-fit the model afterwards (see Section 2.3). We then compared the computed distance values from our calibrated model to the known distances, showing that our sensor had a mean accuracy of 20.39%, with drastically reduced accuracy below 10 inches[9] and fluctuating but fairly consistent accuracy between 10-60 inches, which is consistent with the sensor's data sheet (Figure 3b). We placed our cardboard letter at 20 inches for the final scans (Section 3.2) because of the relatively low error around that point.

3.2 Scanning

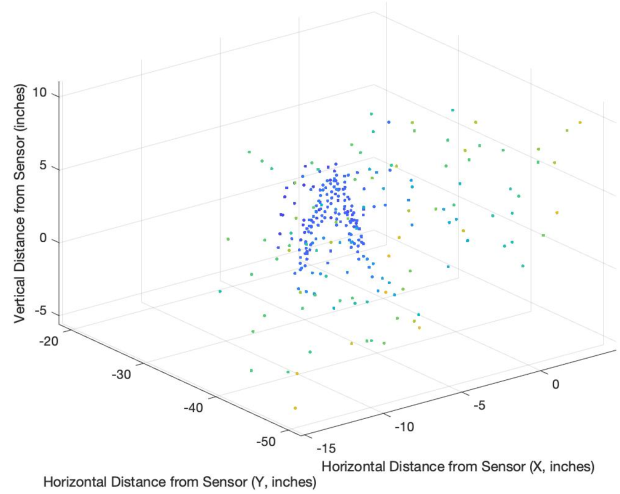
Figure 4a shows the plot of a vertical sweep along the vertical axis with the tilt servo. The clustering of points at $y = 20\text{in}$ is the cardboard letter as expected, since the pan-tilt mechanism is located at $(0, 0)$. This validates our software, physical configuration, and curve fitting equation, as the expected outcome was a vertical line representing a plane at a distance of 20 inches from the sensor.

[8]We collected a sample every 1.5 inches between 4 and 60 inches from the sensor, inclusive.

[9]If you considered only data points $> 10\text{in}$, our calibration model would have had an accuracy of 12.92%.



(a) Our 1D test scan, using only the tilt axis. As expected, a clear vertical line is visible 20in from the sensor, representing the side profile of the cardboard A.



(b) Our 2D test scan, using both the pan and tilt axes. The full cardboard A is detected and clearly visible, demonstrating the success of our 3D scanner.

Figure 4: Visualizations of our test scans, one using a single axis and one using both axes.

We then ran a full scan of our letter using both axes. The cardboard letter was again placed approximately 20 inches from the base of the pan-tilt mechanism (Figure 2b). Sensor measurements were taken at 1° increments on both the pan and tilt axes. Using Equation 2, the data was translated to physical distances from the sensor in inches. Any measurement more than 60in from the sensor (ie. outside the sensor's operating range) was discarded and the $\mathrm{sph2cart}$ command in MATLAB converted the data from spherical coordinates to Cartesian coordinates, as visualized in Figure 4b, where the letter A is clearly visible.

4 Conclusion

For this mini-project, we developed a 3D scanner using an infra-red distance sensor and two hobby servos. We used one servo each for the X and Y axes, and controlled the system with an Arduino microcontroller connected to a Python script running on a laptop. After calibration, our system was reliably accurate, especially in the 10-60in range. In the future, we'd extend our system to use more advanced and precise calibration techniques (possibly including controlling the ambient light), and to use motors with more fine-grained control than our hobby servos (which are limited to 1 degree increments).

4.1 Reflections

4.1.1 Berwin

This was the first time I've worked on a project that sent data between different programming languages (specifically, Arduino and Python). It was also an opportunity to consider the impact of the physical setup on data collection, since the position of the sensor relative to the axes of rotation impacted the accuracy of the final 3D point cloud. We also got to think about test design for sensor calibration, and coming up with a controlled experimental setup. I think that the area where we could have improved upon was cleaning and filtering the data, or collecting the data in a way that reduced noise. Overall, I am satisfied with our final results from this mini-project.

I fully agree with what Ari wrote about our team dynamic: I think we worked effectively with a positive outcome. Going into mini-project 3, I am definitely seeking a team with greater balance in majors/interests to have a more well-rounded team.

4.1.2 Ari

I've worked with Python a fair amount in the past, so a key goal of mine for this project was to write Python code that was robust and capable–I wanted to end up with clean, well-written code. I feel like we accomplished this goal fairly well. Our ArduinoLib wrapper for interfacing with the Arduino allows for clean, clear, and concise code without being too over-complicated. Aside from a faulty USB cable, our system was generally robust. We did hit a limitation when attempting to do our data analysis and graphing with Python: Jupyter notebooks don't allow interactive figures, and the ability to pan, zoom, and rotate a point cloud was essential for debugging our scanner. We ended up writing to a CSV from Python and importing that into MATLAB, which wasn't ideal but was a good cost-benefit trade-off. Calibrating the sensor was a fairly new activity for me, and I found it challenging but interesting. I'm pleased with where we were able to get our system, and I'm proud of this mini-project.

I think we worked fairly well as a team, especially considering that I was sick for a substantial portion of the project. My delta for next time would be that Berwin and I are both software-focused engineers, and having a balance of skill sets on the team might have been valuable.

Appendices

Appendix A: ArduinoLib.py: Python $\Longleftrightarrow$ Arduino Library

```python
from serial import Serial

class Arduino:
  """
  A wrapper class around a serial port with some helpers for easy communication to/from an Arduino.

  Example:

  ```python
  with Arduino('/dev/com.whatever', baudRate=115200) as arduino:
    for line in arduino.lines():
      print("Got line from Arduino: " + line)
      arduino.write(55) # Writes 0x55 as a byte. Also accepts raw bytes or strings.
  ```
  """

  def __init__(self, port, baudRate=115200, timeout=1, logging=False):
    """
    Connect to an Arduino
    """
    self.logging = logging
    self.serial_port = Serial(port, baudRate, timeout=timeout)

  def _log(self, *args, **kwargs):
    """
    Helper method that acts like `print`, when logging=True but does nothing otherwise.
    """
    if self.logging:
      print("[Arduino]", *args, **kwargs)

  # We want to just pass through to the serial port's context manager
  def __enter__(self):
    """
    When using an Arduino as a context manager, the Arduino will intelligently open/close the serial
    port upon entering/exiting the context manager, including doing so multiple times.
    """
    self._log("Entering Arduino context manager, connecting serial port...")
    self.serial_port.__enter__()
```

```python

        # But return self so you can do `with Arduino(...) as arduino:`
        return self

    def __exit__(self, __exc_type, __exc_value, __traceback):
        """
        When using an Arduino as a context manager, the Arduino will intelligently open/close the serial
        port upon entering/exiting the context manager, including doing so multiple times.
        """
        self._log("Exiting Arduino context manager, disconnecting serial port...")
        return self.serial_port.__exit__(__exc_type, __exc_value, __traceback)

    # NB: Calling lines() or packets() more than once is undefined behavior
    def lines(self, drain_first=True):
        """
        Return an iterator that yields each line the Arduino sends over the Serial connection.

        If drain_first is True, any serial data already received and buffered but not yet processed will
        be erased.

        NOTE: This iterator will block while waiting for a line
        NOTE: Calling this method more than once, or calling it after packets() has been called, is
              undefined behavior.
        """
        if drain_first:
            self.serial_port.reset_input_buffer()

        while True:
            # NOTE: technically this would get rid of leading spaces too if that was something you cared about
            line = self.serial_port.readline().decode().strip()
            if len(line) > 0:
                self._log(f"Received Line: {line}")
                yield line

    def packets(self, drain_first=True):
        """
        Return an iterator that yields each packet the Arduino sends over the Serial connection.

        A packet is defined as a newline-terminated, comma-separated list of integers. In other words,
        this method expects that your Arduino writes data over serial that looks like this: `1,2,3\n`.

        If drain_first is True, any serial data already received and buffered but not yet processed will
        be erased.
```

```

        NOTE: This iterator will block while waiting for a line
        NOTE: Calling this method more than once, or calling it after lines() has been called, is undefined behavior.
        """
        for line in self.lines(drain_first=drain_first):
            packet = tuple(int(data) for data in line.split(','))
            self._log(f"Received Packet: {packet}")
            yield packet

    def write(self, data):
        """
        Write data to the Arduino over Serial. If data is bytes, it will be sent as-is. If data is an
        int, it will be converted to an unsigned 8-bit integer and sent that way (attempting to write an
        integer outside of the range 0-255 is an error). If data is a string it will be utf-8 encoded.
        If data is a list each element will be individually written as per the above rules.
        """
        self._log(f"Writing data (may need conversion to bytes): {data}")

        if not isinstance(data, bytes):
            if isinstance(data, str):
                self.write(data.encode('utf-8'))
            elif isinstance(data, int):
                self.write(data.to_bytes(1, 'big', signed=True))
            elif isinstance(data, list):
                for data_item in data:
                    self.write(data_item)
            else:
                raise Exception("Cannot write data of unknown type!")
        else:
            self.serial_port.write(data)

    def writeln(self, data):
        """
        Write a string to the Arduino over Serial, and add a newline at the end.
        """
        return self.write(f"{data}\n")
```

Appendix B: Calibration

Python

```python
import os

START_DISTANCE = 7   # in
END_DISTANCE = 67    # in
STEP_DISTANCE = 1.5  # in

# 0 on the tape measure is 3in behind the sensor. This enables us to display distances to the user
# as they would read them from the tape measure, but store the data accurately.
OFFSET = -3 # in,

def floatrange(start, stop, step):
  """
  Just like the builtin range(), but works with floats
  """
  value = start
  while value <= stop:
    yield value
    value += step

# Get the path to the folder that this script is in: https://stackoverflow.com/a/4060259
__location__ = os.path.realpath(os.path.join(os.getcwd(), os.path.dirname(__file__)))

arduinoComPort = "/dev/cu.usbmodem143201"
data = []

with Arduino(arduinoComPort, baudRate=115200, logging=True) as arduino:
  print(f"Connected to Arduino! Calibrating from {START_DISTANCE}in to {END_DISTANCE}in ({STEP_DISTANCE}in steps)...")
  packets = arduino.packets()

  for distance in floatrange(START_DISTANCE, END_DISTANCE + 1, STEP_DISTANCE):
    input(f"Move the target to {distance}in from the sensor, then press ENTER: ")
    arduino.write(1)
    reading = next(packets)[0]
    print(f"Got: {reading} @ {distance}in")
    data.append((distance + OFFSET, reading))

  print("Done!")

  name = input("Name this calibration: ").strip()
  path = os.path.join(__location__, "calibrations", name + '.csv')
```

```python

  with open(path, 'w') as f:
    f.write(”distance,sensorReading\n”)
    for distance, reading in data:
      f.write(f”{distance},{reading}\n”)

  print(”Calibration written to: ” + path)
```

Arduino

```cpp
#include <Servo.h>

#define BAUD_RATE 115200

#define SENSOR_PIN A0
#define Y_SERVO_PIN 9
#define X_SERVO_PIN 10

// See setup()
#define X_SERVO_POS 90
#define Y_SERVO_POS 100

Servo xServo;
Servo yServo;

void setup()
{
  // Connect to Python
  Serial.begin(BAUD_RATE);

  // Configure the pins
  pinMode(SENSOR_PIN, INPUT);

  // Calibration doesn't move the servos, but we want to set them to a fixed position.
  // If the servos aren't physically attached, this does nothing which is fine.
  xServo.attach(X_SERVO_PIN);
  yServo.attach(Y_SERVO_PIN);
  xServo.write(X_SERVO_POS);
  yServo.write(Y_SERVO_POS);
}

void loop()
{
  // Python writes one byte for every sensor reading it wants to receive
  if (Serial.available() > 0)
```

```
  {
    Serial.read(); // Consume one byte

    delay(250); // Go slow to avoid overwhelming things

    // Getting the lowest of three sensor readings, as per Brad's instructions
    int sensorValue = min(min(analogRead(SENSOR_PIN), analogRead(SENSOR_PIN)),
    ↪  analogRead(SENSOR_PIN));

    // Send the sensor value back to Python
    Serial.println(sensorValue);
  }
}
```

MATLAB: Curve Fitting

```
function [fitresult, gof] = createFit(measuredFiltered, actualFiltered)
%CREATEFIT(MEASUREDFILTERED,ACTUALFILTERED)
%  Create a fit.
%
%  Data for 'distance_from_data' fit:
%      X Input : measuredFiltered
%      Y Output: actualFiltered
%  Output:
%      fitresult : a fit object representing the fit.
%      gof : structure with goodness-of fit info.
%
%  See also FIT, CFIT, SFIT.

%  Auto-generated by MATLAB on 17-Sep-2021 10:26:27


%% Fit: 'distance_from_data'.
[xData, yData] = prepareCurveData( measuredFiltered, actualFiltered );

% Set up fittype and options.
ft = fittype( 'exp1' );
opts = fitoptions( 'Method', 'NonlinearLeastSquares' );
opts.Display = 'Off';
opts.StartPoint = [160.905831540222 -0.00439662441852814];

% Fit model to data.
[fitresult, gof] = fit( xData, yData, ft, opts );

% Plot fit with data.
```

```
figure( 'Name', 'distance_from_data' );
title("Analog output vs. Distance")
h = plot( fitresult, xData, yData );
legend( h, 'actualFiltered vs. measuredFiltered', 'distance_from_data', 'Location', 'NorthEast',
  ↪ 'Interpreter', 'none' );
% Label axes
xlabel( 'measuredFiltered', 'Interpreter', 'none' );
ylabel( 'actualFiltered', 'Interpreter', 'none' );
grid on
```

MATLAB: Error Analysis

```
%% Calibration Curve/Data Graph

% Load data
realValues = readtable("C:\Users\blan\Desktop\PIE-
  ↪ MiniProject2\mp2\Calibrator\calibrations\calibration3.csv");
distances = table2array(realValues(:,1));
readings = table2array(realValues(:,2));

% Remove the clear outlier at distances = 33
distancesFiltered = distances(distances > 15 & distances ~= 33 & distances < 60);
readingsFiltered = readings(distances > 15 & distances ~= 33 & distances < 60);

%   y = sensorValue
%   a =        7709
%   b =      -1.096
%
%
%   return (y / a) ** (1 / b)

a = 7709; b = -1.096;
calculated_distance = (readingsFiltered / a) .^ (1/b);

% Plot actual and fitted data
figure(); clf; hold on;
plot(readingsFiltered, calculated_distance); label1 = "Fitted line";
plot(readingsFiltered, distancesFiltered, '.', 'MarkerSize', 15); label2 = "Measured distance";
xlabel("Sensor Output"); ylabel("Distance from Sensor (inches)");
legend({label1, label2}, 'Location', 'best');

%% Error Calculation
figure; clf;

% Read the data
```

```
data = readtable('error.csv');
mask = data.distance < 60; % Mask to the supported range
knownDistance = data.distance(mask);
sensorReading = data.sensorReading(mask);

a = 7709; b = -1.096; % Calibration parameters

% Do the calculations
calculatedDistance = (sensorReading / a) .^ (1/b);
percentError = abs((calculatedDistance - knownDistance) ./ knownDistance) .* 100;

plot(knownDistance, percentError, 'b-', 'LineWidth', 1.5);
xlabel("Distance from Sensor (inches)"); ylabel("Percent Error");
yline(mean(percentError), "k-", "Average: " + round(mean(percentError), 2),
 ↪  "LabelHorizontalAlignment","left", "LabelVerticalAlignment", "bottom", "FontSize", 12);
```

Appendix C: Scanning

Python

```
from serial import SerialException
from time import time

arduinoComPort = "/dev/cu.usbmodem143201"

# This function is mostly generated by MATLAB Curve Fitting
def sensor_to_distance(sensorValue):
    """
    NOTE: The curve is fit with distance on the x axis and the sensor reading on the y axis, which we
    need to reverse to get a reading -> distance mapping.

    OUTPUT OF MATLAB CURVE FITTING:

    General model Power1:
      f(x) = a*x^b
Coefficients (with 95% confidence bounds):
       a =        7709  (7265, 8153)
       b =      -1.096  (-1.113, -1.078)

Goodness of fit:
    SSE: 345.2
    R-square: 0.9982
    Adjusted R-square: 0.9981
    RMSE: 3.337
    """

    y = sensorValue

    # Copy and paste the coefficients from MATLAB above to here (hence the weird formatting)
    a =        7709
    b =      -1.096

    # This is the equation given by MATLAB but solved for y
    return (y / a) ** (1 / b)

# We had some issues with the Arduino disconnecting (turns out it was the cable!), so we build a
# mechanism by which it can resume scanning where it left off if gets interrupted. That's why
# do_scan accepts three parameters--but the default values are correct if starting a scan from scratch.
def do_scan(data=[], last_xPos=0, last_yPos=0):
```

```python
  data = list(data) # duplicate the list to avoid mutating things
  try:
    with Arduino(arduinoComPort, baudRate=115200) as arduino:
      print("Connected to Arduino!")

      # Enables scanning, and sets the position
      # NB: the default values (0 for both axes) are outside the intended scanning range, and
      # so the Arduino will automatically clip them to the default starting values.
      # See Scanner.ino for more details.
      arduino.write([1, last_xPos, last_yPos])

      for status, xPos, yPos, sensorValue in arduino.packets():
        if status == 1: # If status is 1, we've completed the scan
          print("Done!")
          return data
        if sensorValue == 0: # If sensorValue is 0, that's noise so ignore this packet
          continue
        distance = sensor_to_distance(sensorValue)
        print(f"Got Packet: xPos = {xPos}, yPos = {yPos}, sensorValue = {sensorValue}, distance = {distance}in")
        data.append((xPos, yPos, distance))
        last_xPos = xPos
        last_yPos = yPos
  except SerialException: # If we disconnected
    print("Arduino crashed! Trying to restart!")
    return do_scan(data, last_xPos, last_yPos)

# Run a scan
data = do_scan()

# Save the data as a CSV
with open(f"/Users/ariporad/work/PIE/02-miniproject2/mp2/scanner/scan_{int(time())}.csv", 'w') as f:
    for line in [('xPos', 'yPos', 'distance') ] + data:
        f.write(','.join((str(x) for x in line)) + '\n')
```

Arduino

```cpp
#include <Servo.h>

#define BAUD_RATE 115200

#define SENSOR_PIN A0
#define Y_SERVO_PIN 9
#define X_SERVO_PIN 10
```

```

// All positions are servo positions, so within [0, 180]
#define X_POS_MIN 90
#define X_POS_MAX 90
#define X_POS_STEP 1
#define Y_POS_MIN 75
#define Y_POS_MAX 100
#define Y_POS_STEP 1

int mode = 1; // 0 = disabled, 1 = scanning

Servo yServo;
int yPos = Y_POS_MIN;

Servo xServo;
int xPos = X_POS_MIN;

void setup()
{
  // Configure the Serial connection
  Serial.begin(BAUD_RATE);
  Serial.setTimeout(1);

  // Setup the sensor
  pinMode(SENSOR_PIN, INPUT);

  // Initialize all servos to their starting positions
  yServo.attach(Y_SERVO_PIN);
  yServo.write(yPos);
  xServo.attach(X_SERVO_PIN);
  xServo.write(xPos);
}

void loop()
{
  // First, check if we've received data from Python
  if (Serial.available() > 0)
  {
    mode = Serial.read(); // First byte is always the new mode

    // If we've just started scanning, Python can also provide exactly two numbers as the starting
    // positions of the servos (used to resume a failed scan).
    if (mode == 1 && Serial.available() >= 2)
    {
      // The provided positions must fit within the allowed position boundaries
```

```
    // Python uses this when starting a fresh scan, by sending 0, 0 to get the minimum positions
    xPos = constrain(Serial.read(), X_POS_MIN, X_POS_MAX);
    yPos = constrain(Serial.read(), Y_POS_MIN, Y_POS_MAX);
  }

  // If we've just switched modes, drain all bytes currently in the Serial buffer
  // It's unclear exactly how beneficial this is
  while (Serial.available()) Serial.read();
}

switch (mode)
{
  case 0: // Disabled
    // When disabled, always reset the servo positions
    xPos = X_POS_MIN;
    yPos = Y_POS_MIN;
    break;
  case 1: // Scan
  {
    // Read from the sensor, using Brad's suggestion to take the lowest of three readings
    int sensorValue = min(min(analogRead(SENSOR_PIN), analogRead(SENSOR_PIN)), analogRead(SENSOR_PIN));

    // Send data to Python
    Serial.print("0,"); // status = 0, because we are still scanning
    Serial.print(xPos);
    Serial.print(",");
    Serial.print(yPos);
    Serial.print(',');
    Serial.println(sensorValue);

    // Move the servos
    yPos += Y_POS_STEP;   // increment y
   if (yPos > Y_POS_MAX) // if we've finished a sweep along the y axis, reset and increment x
    {
      yPos = Y_POS_MIN;
      xPos += X_POS_STEP;
    }
    if (xPos > X_POS_MAX) // if we've just incremented x beyond the end of the x axis, we're done!
    {
      xPos = X_POS_MIN; // reset the x servo (y is already at the starting position if we're here)
      mode = 0; // Stop scanning
      Serial.println("1,0,0,0"); // Tell Python we finished
```

```
      }
    }
  }

  // Every loop, move the servos to whereever they should be
  yServo.write(yPos);
  xServo.write(xPos);

  // Delay to give servos time to move and to avoid overloading Serial
  delay(400);
}
```

MATLAB

```
%% 1D Scan Graph
figure; clf;

% Read the data
data = readtable('1d.csv');
xPos = deg2rad(data.xPos);
yPos = deg2rad(data.yPos + 90); % We need to rotate one axis by 90deg otherwise it's sideways
knownDistance = data.distance;

mask = data.distance <= 60; % Mask to supported range

% Convert to Cartesian
[x, y, z] = sph2cart(xPos(mask), yPos(mask), knownDistance(mask));

% Plot
scatter(-y, z, 70, "."); % Invert y-axis so the image is the right direction
xlabel("Horizontal Distance from Sensor (inches)");
ylabel("Vertical Distance from Sensor (inches)");
xlim([0, 60]); % Always show the full range

%% 2D Scan Graph
figure; clf;

% Read the data
data = readtable('1scan.csv');

xPos = deg2rad(data.xPos);
yPos = deg2rad(data.yPos + 90); % We need to rotate one axis by 90deg otherwise it's sideways
knownDistance = data.distance;

% Mask to supported range
```

```
mask = knownDistance <= 60;
xPos = xPos(mask);
yPos = yPos(mask);
knownDistance = knownDistance(mask);

% Convert to Cartesian (MATLAB can't do a spherical graph)
[x, y, z] = sph2cart(xPos, yPos, knownDistance);

% Plot
scatter3(x, y, z, 20, knownDistance, ".");
xlabel("Horizontal Distance from Sensor (X, inches)");
ylabel("Horizontal Distance from Sensor (Y, inches)");
zlabel("Vertical Distance from Sensor (inches)");
```