PoE Lab 2: 3D Scanning

Jared Briskman, Matthew Brucker

September 2017

1 Introduction

In this lab[3], we were tasked with building a 3D scanner utilizing two servos and an infrared distance sensor.

2 Mechanical

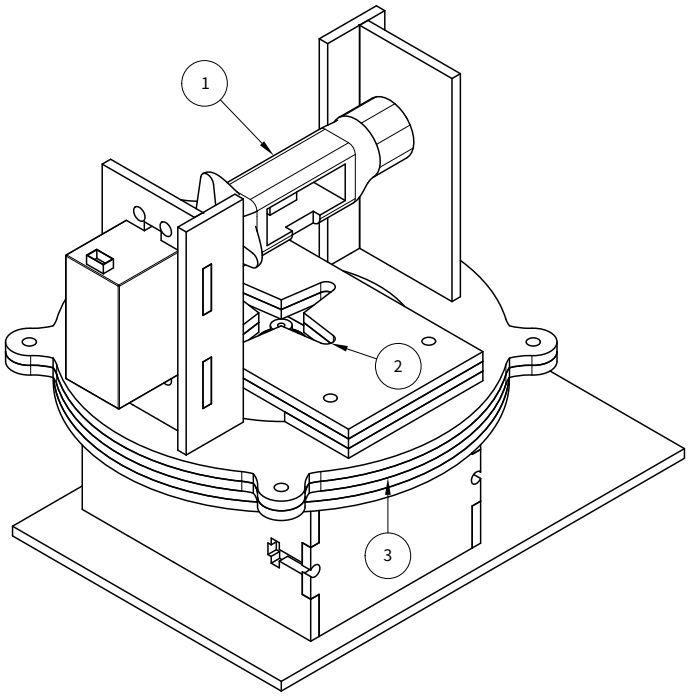


Figure 1: Wireframe render of pan-tilt mechanism. ① - IR sensor carrier, ② - Pan servo horn interface, ③ - Delrin slip rings

For mechanical design, we chose to exceed the necessary complexity required for a simple pan tilt, mostly for the satisfaction of "overkill". A render of the mechanism is shown in Figure 1. For ease of fabrication, it was primarily lasercut $1/8^{\text{th}}$ inch hardboard, with lasercut delrin bushing rings [③] and a 3D printed carrier for the IR distance sensor [①]. The pan servo horn [②] bears no axial load, and is only constrained rotationally. Additionally, the bending moment on the tilt servo is mitigated with a delrin pin and steel bushings on the opposite side. This design also keeps the IR sensor in line with both axes of rotation, minimizing the need for corrections from an offset projection. More specifically, when translating from spherical to cartesian coordinates, the origin of both coordinate systems are the same, simplifying the transformation.

3 Electrical

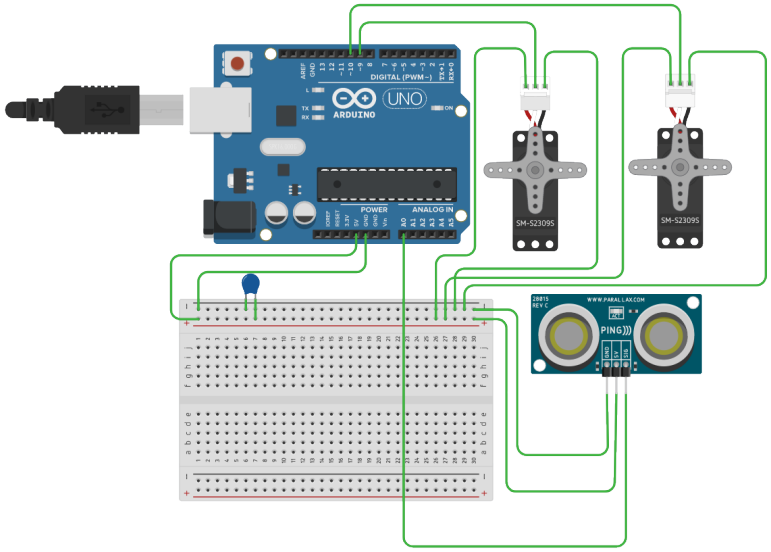


Figure 2: Basic electrical schematic, with a $10\mu F$ bypass capacitor between power and ground.

3.1 Circuit Design:

The electrical design is very minimal, as shown in Figure 2. The pan and tilt servos are connected to digital PWM pins 9 and 10 respectively, and the IR distance sensor is connected to analog pin A0. The only notable element is a $10\mu F$ bypass capacitor between V_{cc} and ground, as reccomended by the IR

sensor datasheet [2]. This filters AC noise in the 5V rail, theoretically leading to cleaner IR sensor readings.

4 Software

4.1 Testing and Calibration:

In order to test and calibrate the sensor, we chose to measure the voltage from the sensor at two different distances and calculate a linear approximation of the voltage-distance conversion function based on these two points. While somewhat crude, this method is well suited to scanning a flat letter on an open table. Our setup for calibration is shown in Figure 3.

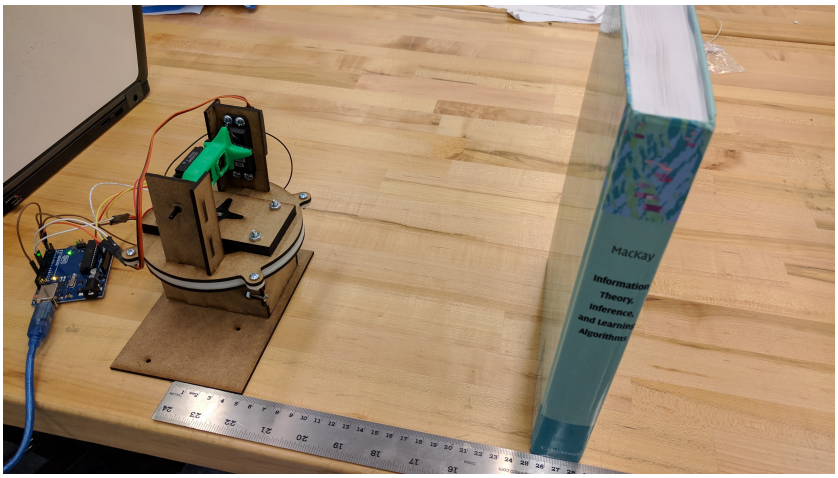


Figure 3: Our setup for calibrating the sensor.

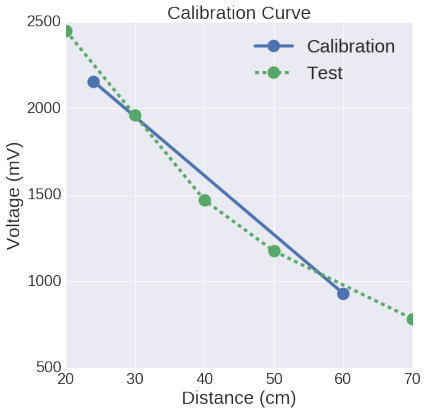
We linearly map the voltage to distance using a calibration function, as follows:

$$d = \frac{(V_{in} - V_{calibmin})(d_{calibmax} - d_{calibmin})}{V_{calibmax} - V_{calibmin}} + d_{calibmin} \tag{1}$$

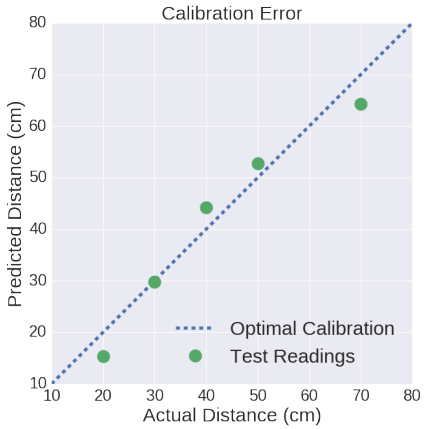
where d represents distances, and V represents voltages, and min and max represent the near and far calibration points, respectively. In this case, our constants are:

$$V_{calibmin} = 2156\,mV,\ V_{calibmax} = 931\,mV,\ d_{calibmin} = 24\,cm,\ d_{calibmax} = 60\,cm$$

This linear approximation allowed us to do the conversion from voltage to distance on the arduino with the built-in `map()`, which is much simpler than curve fitting. It is important to note that `AnalogRead()` returns units from 0 to



(a) Visualization of our calibration dataset and test dataset.



(b) Vizualization of predicted distances compared to actual distances.

Figure 4: Two views into our calibration. 4a shows the two measurements we used for our calibration function in blue, superimposed with the set of test measurements in green. 4b illustrates the error in our calibration. The further any measurement is from the X=Y line, the worse the calibration.

1024, where each unit represents 4.9 mV. As seen in Figure 4, this approximation gives quite reasonable results when dealing with large differences in distance. If more accuracy was desired, a third order calibration curve should suffice.

4.2 Device Side:

The Arduino code serves to actuate the servos and send the scaled sensor data over the serial port so it can be visualized. Rather than use a loop to control the movement of the servo, we use two variables that keep track of the pan and tilt angles and are continuously updated (7.1, lines 31 - 41). When the pan angle reaches one of its limits, it changes direction and increments the tilt angle (lines 36 - 37); once the tilt angle reaches 120°, the servos stop. The distance is recorded from the distance sensor and scaled using the `map()` function (line 30), and sent via serial, along with the current pan and tilt angles. We chose to send the numbers as strings of ASCII characters, as we found that it was simpler than dealing with sending them as bytes, since Python has functions for reading and decoding the characters via serial. We also used 'x' as a delimiting character between the values, so each datum sent via serial is in the form $a\mathrm{x}\phi\mathrm{x}r$.

4.3 Visualization:

We chose to use Python's NumPy, and matplotlib libraries to do the visualization of our sensor data, since they all have easy-to-use functions to do the math of plotting the points. The function `get_euler()` (7.2, lines 20-29) handles the

processing of the serial data, converting it into usable values and discarding any malformed data. After reading in the sensor data and saving it to a CSV file using points_to_csv() (lines 60-71), we convert the points from spherical coordinates using transform_points() (lines 45-50) and its helper functions rotz() (lines 32-35) and roty() (lines 38-41); these functions input a distance and the pan/tilt angles and convert points into Cartesian space using the following rotation matrices:

$$P_{cart} = R_y R_z \tag{2}$$

$$R_z = \begin{bmatrix} cos(\theta) & -sin(\theta) & 0 \\ sin(\theta) & cos(\theta) & 0 \\ 0 & 0 & 0 \end{bmatrix} \tag{3}$$

$$R_y = \begin{bmatrix} 1 & 0 & 0 \\ 0 & cos(\phi) & sin(\phi) \\ 0 & -sin(\phi) & cos(\phi) \end{bmatrix} \tag{4}$$

Then, once all the points have been converted, we plot them on a 3D plot, using get_pyplot() (lines 75 - 81) to set up the matplotlib settings, and with a colormap to help show the depth of the points (line 99).

5 Results

The setup for both our 2D and 3D scans was the same; it can be seen in Appendix 7.3.

5.1 2D

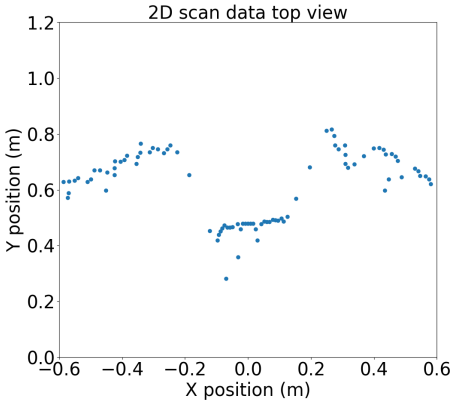
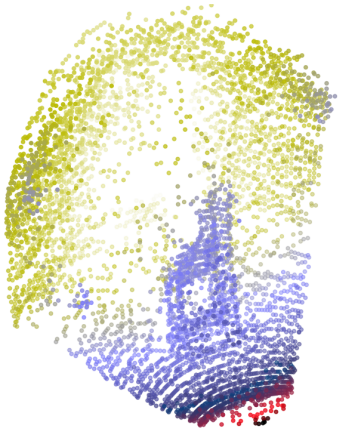
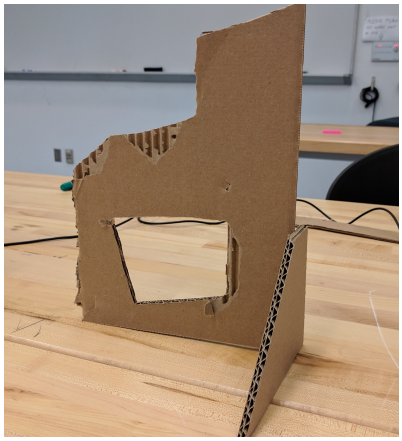


Figure 5: The results of our 2-dimensional scan.



(a) Results visualized in a point cloud.



(b) The actual letter we scanned.

Figure 6: Results from 3D scanning, with a letter clearly visible.

5.2 3D

Figure 6a Shows the final output of our 3D scanner, plotted as a point cloud in cartesian space. While the main subject looks slightly misshapen, this is in fact accurate to reality, as seen in 6b.

6 Reflections

If we had to choose a kaizen from this lab, it would be to focus on getting stable baseline functionality and only then building on top of it. This is by no means a new insight, but we were reminded of the utility of small, achievable steps when rapidly prototyping both the hardware and software for the scanner. In hardware, compensating for lasercutting kerf in order to try and get press fit parts took several attempts. It would have been much simpler to test several configurations in the beginning to find optimal settings, and then cut the entire design once. In software, we tried to do real time plotting from streamed data first, and encountered difficulties. Scaling back to recording data, then plotting afterward was much more successful, and should have been the first goal.

Otherwise, we were able to gain a deeper understanding of matplotlib as a data visualization tool, which will continue to be useful in the future. It was also quite enjoyable to build a slightly excessive mechanical system.

7 Appendix

[View code on Github] [View CAD on Onshape]

7.1 Embedded Code:

```
/*
 * Jared and Matt PoE lab 2
 */
#include <Servo.h>
const int sensorPin = A0;
const int panServoPin = 9;
const int tiltServoPin = 10;

int theta=55; // Starting value for theta
int phi=60; // Starting value for phi
int thetaMult = 1; // Keeps track of which direction the pan servo is sweeping
int phiMult = 1; // Allows for incrementing the tilt angle
bool isMoving = true; // Indicates whether the servos are currently moving or not.

Servo panServo;
Servo tiltServo;

void setup() {
  Serial.begin(9600);
  panServo.attach(panServoPin);
  tiltServo.attach(tiltServoPin);
}

void loop() {
  // Prints the pan angle, tilt angle, and scaled distance via the serial port.
  Serial.print(theta);
  Serial.print('x');
  Serial.print(phi);
  Serial.print('x');
  Serial.println(map(analogRead(sensorPin), 440, 190, 25, 60));
  if (isMoving) { // If the servo is still moving
    theta += thetaMult; // Update the pan angle
    panServo.write(theta);
    tiltServo.write(phi);
    if (theta == 55 || theta == 145) { // If we reach one end of the sweep, start sweeping the other direction
      thetaMult = -thetaMult;
      phi += phiMult;
      if (phi == 120) { // If we reach the end of the scan, stop moving and tell the visualization code that we're no longer moving.
        Serial.print('$');
        isMoving = false;
  }}}
  delay(50); // Delay so that we aren't overloading the serial port
}
```

7.2 Visualization Code:

```
"""
PoE Lab 2 Visualization Code
Jared Briskman and Matt Brucker
"""
#!/usr/local/bin/python

from serial import Serial
import numpy as np
import matplotlib.pyplot as plt
from math import pi, cos, sin
import matplotlib as mpl
from mpl_toolkits.mplot3d import Axes3D
import pandas as pd
import csv

cxn = Serial('/dev/ttyACM0', baudrate=9600)


# Reads a point in spherical form via serial data.
def get_euler(cxn):
    serial_val = cxn.readline()
    if len(serial_val) >= 8:
        try:
            serial_str = serial_val[:-2].decode("utf-8")
        except:
            pass
        str_split = serial_str.split('x')
        if '' not in str_split:
            return str_split


def rotz(theta):  # Generates a z-axis rotation matrix for a given angle
    return [[cos(theta), -sin(theta), 0],
            [sin(theta), cos(theta), 0],
            [0, 0, 1]]


def roty(phi):  # Generates a y-axis rotation matrix for a given angle
    return [[1, 0, 0],
            [0, cos(phi), sin(phi)],
            [0, -sin(phi), cos(phi)]]


# Transforms a list of points from spherical coordinates to Cartesian
def transform_points(thetas, phis, lens):
    points = []
    points_euler = list(zip(thetas, phis, lens))
    points_cart = [point_transform(*point) for point in points_euler]
    print(points)
    return points_cart

```

```
52
53 def point_transform(theta, phi, dist):
54     point_euler = [(int(theta)-100)*(pi/180), -(int(phi)-90)*(pi
       /180), int(dist)/100]
55     orig_vec = [0, point_euler[2], 0]  # The original vector to
       transform
56     point_transform = np.dot(roty(point_euler[1]), np.dot(rotz(
       point_euler[0]), orig_vec))
57     return point_transform
58
59
60 def points_to_csv(file_name='points.txt'):
61     all_points = []
62     while(len(all_points) < 60*90):  # While we've received less
       than the maximum number of points
63         if cxn.inWaiting:  # Wait to receive new serial data
64             euler = get_euler(cxn)
65             if euler:  # If we get valid serial data:
66                 if '$' in ''.join(euler):  # Stop if we reach the
       end
67                     break
68                 all_points.append(euler)
69     df = pd.DataFrame(list(all_points), columns=list('xyz'))
70     df.to_csv(file_name)
71     return all_points
72
73
74 # Sets up pyplot and returns the figure
75 def get_pyplot():
76     mpl.rcParams['toolbar'] = 'None'
77     plt.rcParams['image.cmap'] = 'gist_stern'
78     fig = plt.figure()
79     ax = fig.add_subplot(111, projection='3d')
80     ax.set_axis_off()
81     return ax
82
83
84 points_to_csv()
85 # Open the list of points and format them correctly
86 with open('points.txt', 'r') as points_file:
87     points_reader = csv.reader(points_file)
88     list_points = list(points_reader)[1:]
89     points_clean = [point[1:] for point in list_points]
90
91 # Converts the list of points from spherical to Cartesian
       coordinates
92 points_split = zip(*points_clean)
93 point_vals = transform_points(*points_split)
94 points_deconstruct = zip(*point_vals)
95 points_deconstruct2 = zip(*point_vals) # Needs a copy for the color
        map to work
96
97 # Plots the points and generates the heatmap
98 ax = get_pyplot()
99 ax.scatter(*points_deconstruct, c=list(points_deconstruct2)[1])
100 plt.show()
```

7.3 Setup:

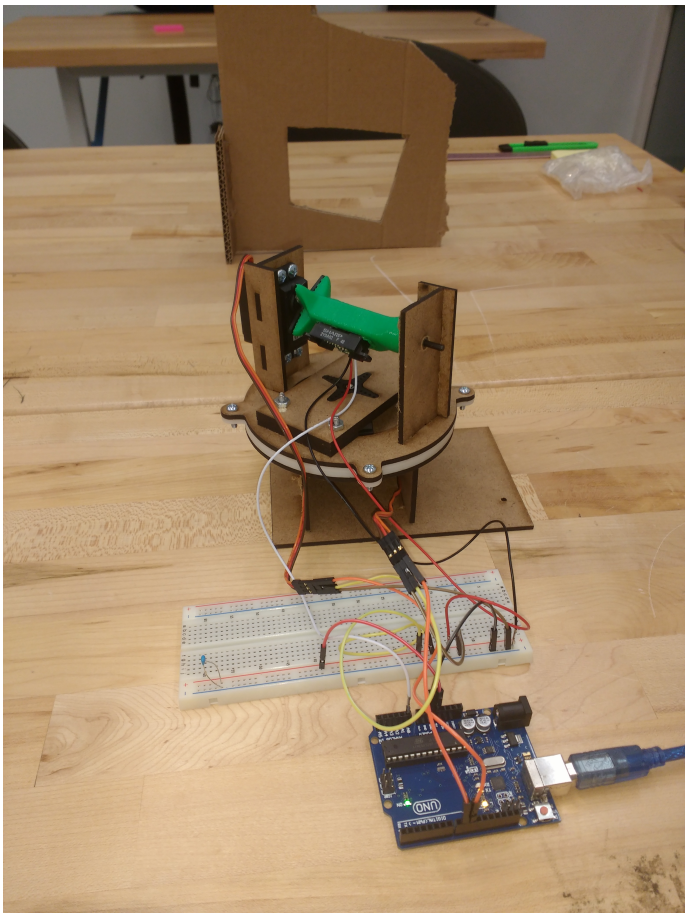


Figure 7: A picture of our final scanner.

References

[1] Servo Datasheet, http://www.vigorprecision.com.hk/uploadfile/20120530/20120530145808454.pdf

[2] Sensor Datasheet, https://www.pololu.com/file/0J156/gp2y0a02yk_e.pdf

[3] Lab Handout, http://poe.olin.edu/lab2.html