POE Lab 2 - DIY 3D Scanner

Carl Moser, Kevin Zhang

September 30, 2016

Abstract

In this lab we created a 3D scanner that could use an IR sensor to measure distances to its surroundings in virtually all directions and then plot them to generate a 3D visualization of the scanned area. The IR sensor was mounted onto a pan-tilt mechanism with two servos, and data flow passed from Arduino bootloader for collection to Python for processing to MATLAB for plotting. The scanner's success was demonstrated through a visualization of a letter made out of cardboard cutouts.

Narrative

We built the 3D scanner through an iterative process of solving smaller problems and then continually building up towards the end goal of generating a 3D visualization of the cardboard letter. Enumerated, our process looks like this:

Procedure

1. Test and experiment with basic functionality of sensor and servos
2. Create a calibration curve of voltage readings against distance for the IR sensor
3. Use Microsoft Excel to generate a calibration function for the sensor
4. Measure distances not on the calibration curve to create an error plot based on actual vs calculated distances
5. Design and 3D print a pan/tilt assembly, build a cardboard letter cutout
6. Write Arduino code to pan across a space and send the collected 2D data over Serial
7. Write Python script to receive 2D data from Arduino through Serial and process it into XY Cartesian coordinates
8. Write a MATLAB script to interpret and plot the processed 2D data
9. Expand Arduino code to scan an entire area using pan and tilt
10. Add functionality to Python script to receive and process 3D data into XYZ Cartesian coordinates
11. Add additional code to MATLAB script to visualize 3D data

Calibration

Our first big step was to calibrate the sensor such that it could output accurate distances that corresponded to voltage readings. We first created a calibration curve and then generated a calibration function, which we tested through an error plot to determine its accuracy.

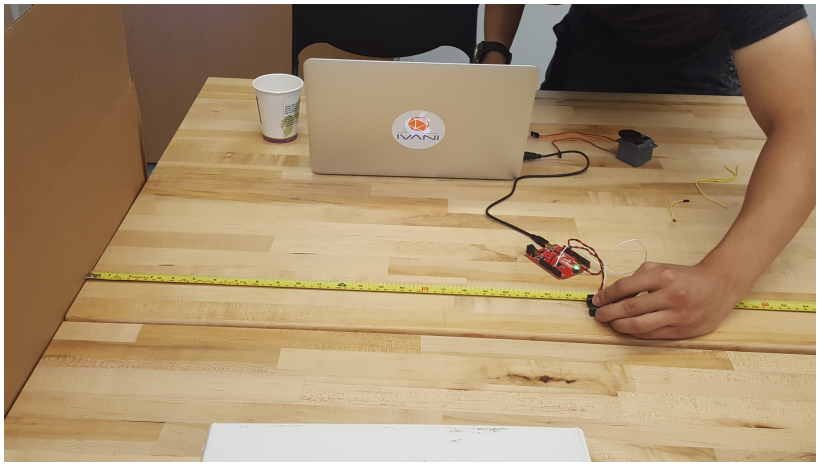


Figure 1: Collecting data points

Calibration Plot

We calibrated the IR sensor by pointing the sensor straight towards a cardboard wall and then collecting analog values from the sensor every 10 cm between 20 cm and 100 cm, as shown in Figure 1. The points were plotted as shown in Figure 2.

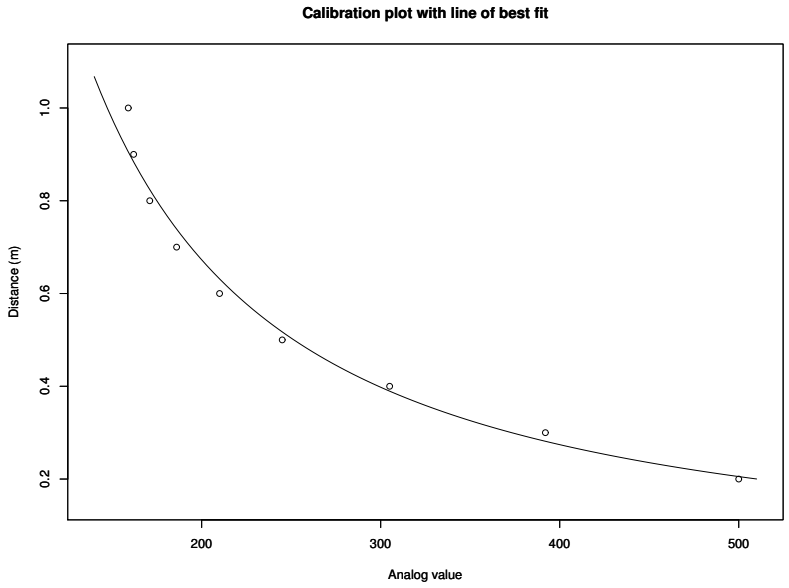


Figure 2: Plot showing the calibration curve
$R^2 = 0.9899$

The Calibration Function

To calculate the calibration function, we plotted the data in Microsoft Excel and went through the different options for line of best fit. We found the power function to have the highest R^2 value. Our calibration function is:

$$D = 642.17x^{-1.295} \tag{1}$$

where D is the distance calculated and x is the voltage reading. The function takes the reciprocal analog value and then scales it with an exponent and a scalar. Given that the R value of the function is $\sqrt{(.9899)} = -.995$ which is very close to -1 and suggests a strong relationship between the variables with this function, we believe that our calibration is accurate.

Error

We then updated our Arduino code to print the distance based on the analog value. Once we uploaded the code, we set the sensor at distances between our original data (25cm, 35cm, etc...) such that they were not utilized within the original calibration, and then we plotted the calculated values vs the actual values to find the error function. Our error function, as shown in Figure 3, was:

$$y = x - 0.01 \tag{2}$$

where x is the actual distance and y is the calculated distance. The fact that the equation is almost $y = x$ means that the calculated values were basically the same as the actual values measured. The R value being $\sqrt{(.9981)} = .999$ confirms that the variables are highly correlated with this linear relationship, and thus the calibration function is accurate.

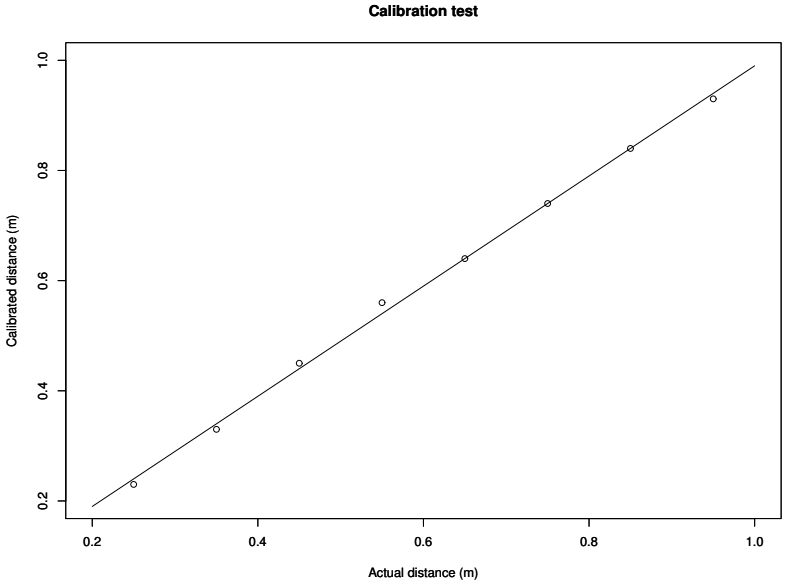


Figure 3: Plot showing the error
$R^2 = 0.9981$

Circuit Diagram

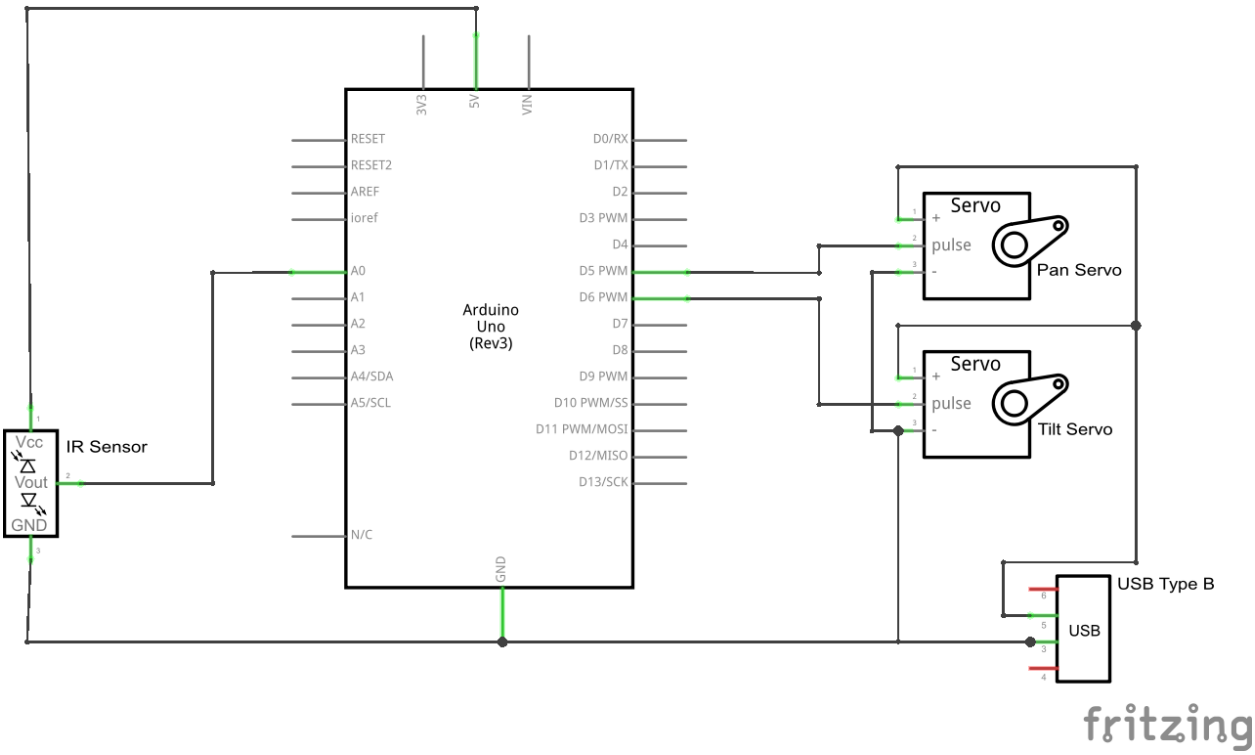
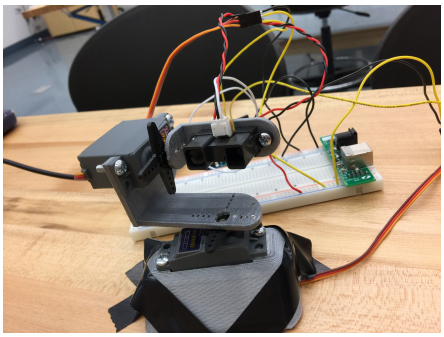


Figure 4: Schematic of the 3d scanner

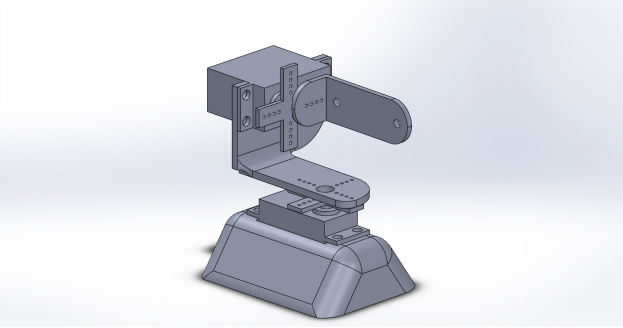
Figure 4 shows our circuit that we used for the 3D scanner. The set up is pretty typical. The IR sensor is put on an analog pin because it outputs analog values. The servos are put on PWM digital pins both for clarity and because apparently there was a hardware problem with reading inputs when all components were utilizing the same pin types. Since an Arduino can only provide roughly 400-500 mA of current and the our two servos apparently need more than that to be powered, the servos are attached to an external usb power supply to ensure all components are sufficiently powered.

Assembly

When designing the pan/tilt assembly, one of our main objectives was to have the IR sensor head rotate about its center. This would allow us to easily process the data and convert between coordinate systems when plotting because the sensor would always be at the origin, preventing any additional calculations due to the sensor moving while scanning. Figure 5a shows what the assembly looks like. Figure 5b is our CAD model. Notice that the pan servo is centered right underneath the IR sensor's position, and the IR sensor is pushed back a bit such that the front of the sensor's head is aligned with the tilt servo. This ensures that the point of measurement would always be at the origin when spinning around on the two axes.



(a) Our pan/tilt assembly



(b) CAD of our pan/tilt assembly

Figure 5: Our Assembly

How our code works

Arduino setup

Our Arduino code uses serial communication to interface with the Python script that we wrote. It allows for 2D and 3D scanning. The code begins by starting Serial and setting up the servos, then proceeds to scan an area. For each point in the area scanned, the code will send a data package containing a distance, a pan angle, and a tilt angle to Serial.

Defining variables

Our Arduino code is where the parameters of the scan are defined such as the upper and lower bounds for the pan and tilt. It also defines the change in the angles, the servo pins, and the sensor pin. We chose to use $\#define$ instead of declaring them as variables because of the increased efficiency, shown in the code below. The pre-compiler will substitute the value of the variables later in the code before the sketch is compiled. This means that there is an easy way to change any of these values without having an impact on performance.

```
8  // The upper and lower bounds for pan
9  #define thetaLower 40
10 #define thetaUpper 100
11
12 // The upper and lower bounds for tilt
13 #define phiLower 40
14 #define phiUpper 100
15
16 // The change in phi/theta
17 #define delta 1
18
19 // The pins for the sensor and servos
20 #define analogInPin A0
21 #define panPin 5
22 #define tiltPin 6
```

Synchronizing start times with Python

To receive input from our Python code, we decided to use a $serialEvent()$ that updates two booleans based on the string input, as shown in the code below. If the input is *start*, it will begin a 3D scan which can take some time to complete. This input is what allows Python to synchronize with Arduino such that both scripts begin at the same time and all data is properly collected. When we were planning on having a real time plot of the scan, we decided to implement functionality to

pause or reset the scan. The 2D scan was relatively fast so we decided to define it in a function vs. have it be a part of *loop* This helped shape the current version of the loop function.

```
void serialEvent(){
  /*
    This function queues up serial events while the loop is running
    and before the loop resets, it runs through the queue. It reads
    the input as a string and runs through some if statements
  */
  input = Serial.readString();
  if(input == "start" && !start){
    /*
      If the input is start and the scan has not started,
      it sets start and started to true
    */
    started = true;
    start = true;
  }
  else if (input == "stop"){
    // If the input is stop, the arduino resets
    resetFunc();
  }
  else if(input == "pause" && started){
    // If the input is pause and it has started scanning, it pauses
    start = !start;
  }
  else if(input == "twodimensions" && !started){
    /*
      If the input is for a 2d scan and a 3d scan is not in process,
      it will start a 2d scan
    */
    capture2d();
  }
}
```

Our Main Loop

Instead of using a nested for loop to do the scan, we decided to use *if* statements to decide when to increment angles, reset angles, and determine when the scan is over. This is shown in the code below. We made this design decision because *serialEvent* only runs through the serial queue at the end of *loop*. The loop takes no more than 500ms to run through which allows us to pause or reset at any point.

```
void loop() {
  /*
    The 3d pan/tilt scanner code was written in the loop
    instead of a nested for loop because serial events are
    not registerd within the loop

    This allows for the scanning to be reset or paused
  */
  if(start){
    // If start is true, it the scanning begins
    if(phi > phiUpper){
      /*
        If the tilt angle is greater than the maximum angle, it resets
        to lower bound and increases the pan angle by delta
      */
      phi = phiLower;
      theta = theta + delta;
      tilt.write(phi);
      pan.write(theta);
      // Delays to allow for the tilt servo to move to the initial position
      delay(250);
    }
    if(theta > thetaUpper){
      /*
        If the pan angle is greater than the maximum angle, the scan
        is over so it resets theta and phi to their lower bounds and
        sets start/started to false
      */
      phi = phiLower;
      theta = thetaLower;
      pan.write(theta);
      start = false;
      started = false;
    }

    // Delays to allow the tilt servo to move to the next position
    delay(100);

    // This is the calibration function
    dist = 642.17*pow(analogRead(analogInPin), -1.295);

```

```
    // Returns the calculated distance, the tilt angle, and the pan angle
    Serial.println(String(dist) + ", " + phi + ", " + theta);

    // Tilts to the next position and adds the change in angle to phi
    tilt.write(phi);
    phi = phi + delta;
  }
}
```

Python Script

Our Python script sets up a connection with the serial port in use, receives the data coming from the Arduino, and processes it. Processing consists of formatting the raw data into a usable form, converting the data from Spherical coordinates to Cartesian, and compiling all the values into three arrays, one for X, Y, and Z. The data is then written to a csv file where it can be picked up by MATLAB.

Receiving Data

From previous experience, we knew that serial communication can be delicate, especially since the receiver is constantly reading data and assumes the data is always valid. There was the possibility that Python could be trying to read data from the serial port but the Arduino wasn't outputting data, so it would receive nothing. There is also a glitch where Arduino can send a package of data across serial but it would be incomplete and not all the components of the data make it through. Finally, even if Arduino sends through a complete package of data, the contents of the components could be blank. We wrote a robust method that caught virtually all issues a data package from Arduino could have, as shown below. The first conditional checks if the data is real and contains something. The second conditional checks that the data has all the parts, which consist of at least radius and theta separated by commas. Lastly, the final conditional ensures that all the parts contain usable information. If all these conditions are passed, then the data is formatted into a float array and returned. With this method, we never had an error regarding serial data communication, and data collection was smooth.

```
def ReadArduino(ser):
        """
        Reads Arduino inputs and converts them to a usable form
        """

        s = ser.readline()

        #ensures the the data is real and isn't corrupted.
        if s is not None and len(s) > 0:

                data = s.split(', ')

                #checks if the data has enough points to be considered valid
                if len(data) >= 2:

                        #formatting
                        data[-1] = data[-1][:-2]
                        print data

                        #checks that infrared sensor works properly (broken will return nothing or empty strings)
                        if all(len(item) > 0 for item in data):

                                for i in range(len(data)):
                                        data[i] = float(data[i])

                                return data

        #will return None if data is bad, which will prompt the while loop to close
        return None
```

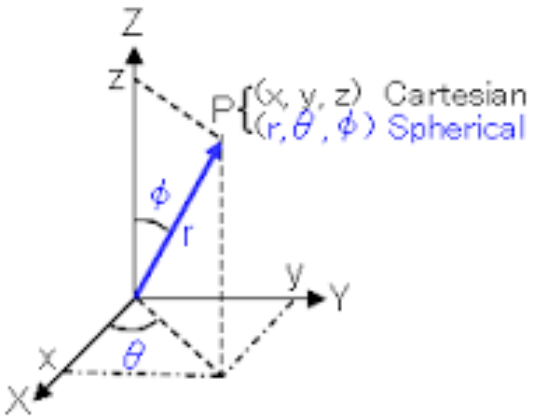


Figure 6: Visual of how Spherical relates to Cartesian

Converting from Spherical Coordinates to Cartesian

After the data was received, we converted the data point from Spherical Coordinates to Cartesian. As shown in Figure 6, the IR scanner will give data in the form of a radius, a pan angle, and a tilt angle, which are essentially Spherical coordinates. Converting to Cartesian requires using the formulas shown in the code below, which consist mainly of trigonometry with right triangles to go from angles to sides. After the data was converted it could be more easily used for visualization.

```
def Spherical2Cartesian(r, phi, theta):
        """
        Converts from Spherical Coordinates to Cartesian Coordinates
        Assumes phi and theta are in degrees
        """

        x = r * np.sin(np.radians(phi)) * np.cos(np.radians(theta))
        y = r * np.sin(np.radians(phi)) * np.sin(np.radians(theta))
        z = r * np.cos(np.radians(phi))

        return x, y, z


def Polar2Cartesian(r, theta):
        """
        Converts from Polar Coordinates to Cartesian Coordinates
        Assumes phi and theta are in degrees
        """

        x = r * np.cos(np.radians(theta))
        y = r * np.sin(np.radians(theta))

        return x, y
```

Synchronizing with the end of data collection

In addition to synchronizing the start times between Python and Arduino, we also had to synchronize the end times. Our code block for the data collection loop, shown below, solves that problem with a simple variable. We had a user input determine what to do with incoming data, depending on if its 2D or 3D. Once the Arduino code finishes, a counter would check how many times the *ReadArduino*() method would return a *None* because Arduino was no longer outputting data. Once the counter passes a threshold to make sure that data transmission is actually done, the loop would exit. This then allows for a distinction between data collection and writing data to a csv file.

```
while start:

                #reads a datapoint
                datapoint = ReadArduino(ser)

                if datapoint is not None:
```

```
134
135                         #decides if 3D or 2D, and updates data arrays accordingly
136                         if dim == "3d":
137                                 coordinates = Spherical2Cartesian(datapoint[0], datapoint[1], datapoint[2])
138                                 xarray, yarray, zarray = Update3DData(coordinates[0], coordinates[1], coordinates[2], xarray,
                                        yarray, zarray)
139
140                         elif dim == '2d':
141                                 coordinates = Polar2Cartesian(datapoint[0], datapoint[1])
142                                 xarray, yarray = Update2DData(coordinates[0], coordinates[1], xarray, yarray)
143
144                 #stops the loop once data stops coming through
145                 else:
146                         stop_counter += 1
147                         if stop_counter > 5:
148                                 start = False
```

MATLAB Code

We used MATLAB primarily for visualization since it's so good at dealing with large vectors and plotting. Thus our MATlAB code is quite simple. We read the processed coordinates from an csv file and separate it into the three axes; X, Y, and Z. Then we use scatterplots to visualize the data points, using appropriate labeling as needed.The MATLAB code's sole purpose was to plot and visualize large data vectors through a user-friendly interface.

Challenges

While doing this lab, we encountered a couple of obstacles that took us some time to overcome.

Overloading the USB port

The first challenge that we had to overcome was the fact that having the sensor and servos be powered from the Arduino caused the Arduino to draw too much power from a single port and shut down to prevent overheating. To fix this, we decided to power both of the servos from a separate usb port using the usb power supply we made in ISIM.

Our definition of the origin

The next challenge that we faced was that we had been using arbitrary angles for our upper and lower bounds because we had not centered the sensor to be pointing straight forward when the servos were at 90 degrees. Since we had assumed that the sensor was at the origin when converting from Spherical coordinates, our visualizations were off-center and seemed distorted and curved. To remedy this, we changed our scan range to be centered around 90 degrees such that the calculated front in the code and the real front of the sensor were aligned.

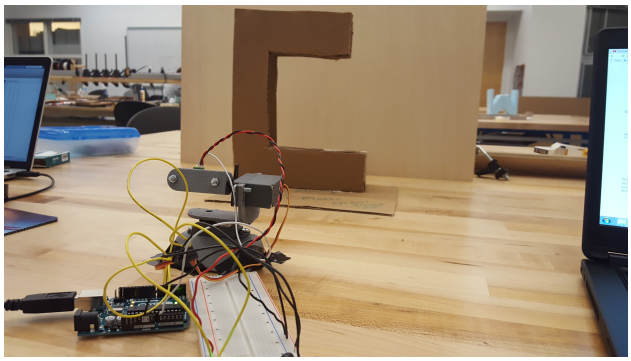
Visualization

Another challenge we faced was plotting in Python. We decided to go with Python at first because it was much easier to connect to serial than with MATLAB. But after collecting data we realized that plotting thousands of points in Python which isn't meant for heavy graphical computation resulted in extremely slow visualizations. Python also wasn't user-friendly in terms of plotting, so we decided to switch to MATLAB for visualizations. To make transitioning to MATLAB easier, we would still utilize our processing in Python, but instead of plotting we would save the data into a csv file, which could be read by MATLAB. MATLAB proved to be much easier for visualizing 3D plots.

Scanning

To test our scanner, we cut the letter C out of cardboard and stood it in front of a backdrop. We then set the scanner about 30 centimeters away from the letter. Our set up for scanning was the same for both the 2D and 3D scan, using the same assembly and components, except for differences in the code that we ran.

(a) Our cardboard letter



(b) Our setup for scanning

Figure 7: Our Setup

2D Scan

To begin 2D scanning, our scanner moves the tilt servo almost parallel to the horizontal. It then moves the pan servo across the area of interest and collects the data for each change in theta.

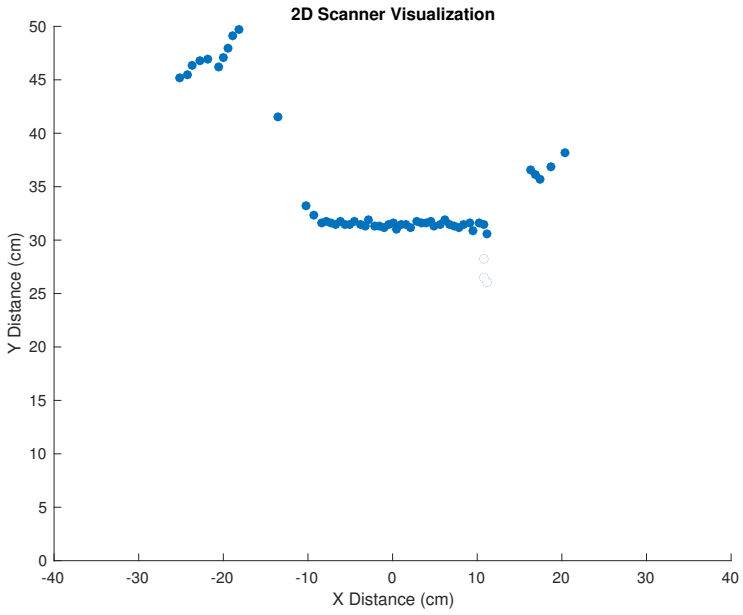
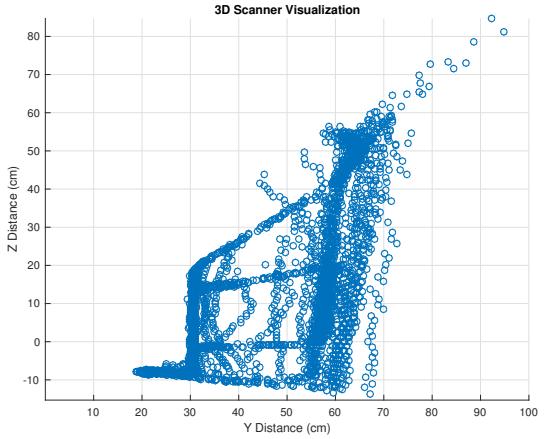


Figure 8: A plot of the 2D scan. The letter is scanned at about 32 cm. The letter was physically about 33 cm away.

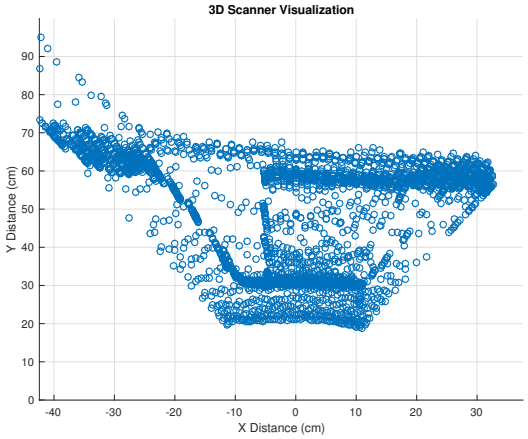
As shown in Figure 8, the straight horizontal line in the graph is the top down view of the letter, with extraneous readings on the edges to show that the sensor had scanned passed the letter. Upon measurement, we found that our scanner was able to scan within roughly 1 cm of the actual distance between the letter and the sensor.

3D Scan

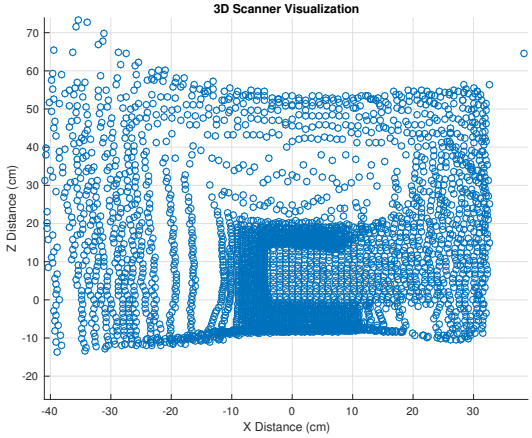
Finally, comes the 3D scan. To begin, our scanner started at the top-right most point of its scanning area. It then moves across the area of the interest by first scanning top-down and then moving over one degree. The process continues longitudinally, and it takes one data point at each angle, resulting in thousands of data points.



(a) Side view of the 3D plot



(b) Top view of the 3D plot



(c) Front view of the 3D plot

Figure 9: The 3D Scan of the C

As can be seen in Figure 9c, there is a very clear letter C in the visualization surrounded by the backdrop. The front view also shows that the backdrop was not picked up behind the letter, resulting in another C-like shape and confirming that the 3D cardboard C exists and was scanned.

Figure 9a and 9b display side and top views of the same scan, to show that the wall is indeed set behind the actual letter and there is dimension to the space. In Figure 9a, the first bold vertical line towards the left is the letter, and the larger bold vertical line on the right is the wall, with some noise in the middle. In Figure 9b, the first bold horizontal line is actually the ground, the second bold horizontal line right above the first is the letter, and the big bold horizontal line much higher than the first two is the wall, displaying depth in our scan and accurately depicting the 3D visualization.

Reflection

We both had a good experience with this lab. Our iterative process of systematically building up the system one component at a time by breaking the problem into smaller ones allowed for minimal error in our code and our assembly. When we were working together on the lab, our work was efficient and focused, so we made a lot of progress each time we met. Finally, our team synergy was very good. We both trusted each other to get stuff done, so we could quickly delegate tasks for each person to do. We also thought along the same lines on various topics, making decisions easy and effective.

Our team could also definitely improve. One issue was that neither of us was well-versed in mechanical engineering, and a major component of this was the mechanical design of the assembly. We might have taken a bit too much time on the assembly, and had we been a more balanced team with one software-oriented person and one mechanically-oriented person, the time spent on various parts of the project might have been more even. Another issue was scheduling meeting times. Our schedules were slightly off, so it was hard sometimes to find adequate times to meet up. Lastly, sometimes we could've had more communication between the team, which might have been partly due to the fact that we never set up a strong communication medium beforehand. It was often hard to contact each other in a timely manner, which sometimes resulted in delegating tasks and working separately.

Appendix

Source Code

Arduino Code

```
/*
  Lab 2
  Carl Moser and Kevin Zhang
 */
#include <math.h>
#include <Servo.h>

// The upper and lower bounds for pan
#define thetaLower 60
#define thetaUpper 120

// The upper and lower bounds for tilt
#define phiLower 50
#define phiUpper 110

// The change in phi/theta
#define delta 10

// The pins for the sensor and servos
#define analogInPin A0
#define panPin 5
#define tiltPin 6


```

```
// Declaring the pan and tilt servos
Servo pan;
Servo tilt;

// Declaring and initializing the angles to their lowest values
int theta = thetaLower; // pan
int phi = phiLower; // tilt

// The value to hold the calibrated distance
double dist = 0;

// Initializing a string to hold serial input
String input = "";

// Volatile booleans to start the scan and to tell if a 3d scan is in the process
volatile bool start = false;
volatile bool started = false;

// Declaring a reset function to stop a scan
void(* resetFunc)(void) = 0;


void setup() {
  // Attaching the servos to their respective pins
  pan.attach(panPin);
  tilt.attach(tiltPin);

  // Setting the servos to their lowest values
  pan.write(theta);
  tilt.write(phi);

  // Initializing serial communication
  Serial.begin(9600);
}


void loop() {
  /*
    The 3d pan/tilt scanner code was written in the loop
    instead of a nested for loop because serial events are
    not registerd within the loop

    This allows for the scanning to be reset or paused
  */
  if(start){
    // If start is true, it the scanning begins
    if(phi > phiUpper){
      /*
        If the tilt angle is greater than the maximum angle, it resets
        to lower bound and increases the pan angle by delta
       */
      phi = phiLower;
      theta = theta + delta;
      tilt.write(phi);
      pan.write(theta);
      // Delays to allow for the tilt servo to move to the initial position
      delay(250);
    }
    if(theta > thetaUpper){
      /*
        If the pan angle is greater than the maximum angle, the scan
        is over so it resets theta and phi to their lower bounds and
        sets start/started to false
       */
      phi = phiLower;
      theta = thetaLower;
      pan.write(theta);
      start = false;
      started = false;
    }

    // Delays to allow the tilt servo to move to the next position
    delay(100);

    // This is the calibration function
    dist = 64217*pow(analogRead(analogInPin), -1.295);

    // Returns the calculated distance, the tilt angle, and the pan angle
    Serial.println(String(dist) + ", " + phi + ", " + theta);

    // Tilts to the next position and adds the change in angle to phi
    tilt.write(phi);
    phi = phi + delta;
  }
}


void capture2d(){
  /*
    This function captures a 2d scan along the pan axis, with phi
    being halfway between the maximum and minimum. This function is
    outside of the main loop because it is relatively fast compared
```

```
    to a 3d scan and does not warrant a need to pause or reset
  */

  // Sets the tilt to the midpoint
  tilt.write(100);

  for(theta = thetaLower; theta < thetaUpper; theta = theta + delta){
    /*
      This for loop increments theta by delta and moves the
      pan servo. At each point, the distance is calculated
      and sent to the computer over serial
    */
    pan.write(theta);
    delay(100);
    dist = 64217*pow(analogRead(analogInPin), -1.295);
    Serial.println(String(dist) + ", " + theta);
  }
  // After the 2d scan, the pan and tilt servos are reset to their lower bounds
  theta = thetaLower;
  tilt.write(phiLower);
  pan.write(theta);
}


void serialEvent(){
  /*
    This function queues up serial events while the loop is running
    and before the loop resets, it runs through the queue. It reads
    the input as a string and runs through some if statements
  */
  input = Serial.readString();
  if(input == "start" && !start){
    /*
      If the input is start and the scan has not started,
      it sets start and started to true
    */
    started = true;
    start = true;
  }
  else if (input == "stop"){
    // If the input is stop, the arduino resets
    resetFunc();
  }
  else if(input == "pause" && started){
    // If the input is pause and it has started scanning, it pauses
    start = !start;
  }
  else if(input == "twodimensions" && !started){
    /*
      If the input is for a 2d scan and a 3d scan is not in process,
      it will start a 2d scan
    */
    capture2d();
  }
}
```

Python Code

```
#Interfaces with an Arduino board to receive data about the Pan-Tilt Scanner
#we built and gives XYZ coordinates for 3D Imaging. Loads to a csv file for better visualization
#in MATLAB or some other program.

import serial
import numpy as np
from mpl_toolkits.mplot3d import Axes3D
from matplotlib import pyplot as plt
import time
import csv


def StartUp():
        """
        Sets up Serial to communicate with Arduino
        """

        ser = serial.Serial('/dev/cu.usbmodem1421', 9600, timeout=1)

        #resets the serial port and prints a statement to ensure it's working
        ser.close()
        time.sleep(1)
        ser.open()
        print ser.isOpen()

        return ser


def ReadArduino(ser):
        """
        Reads Arduino inputs and converts them to a usable form
        """
```

```
33
34          s = ser.readline()
35
36          #ensures the the data is real and isn't corrupted.
37          if s is not None and len(s) > 0:
38
39                  data = s.split(', ')
40
41                  #checks if the data has enough points to be considered valid
42                  if len(data) >= 2:
43
44                          #formatting
45                          data[-1] = data[-1][:-2]
46                          print data
47
48                          #checks that infrared sensor works properly (broken will return nothing or empty strings)
49                          if all(len(item) > 0 for item in data):
50
51                                  for i in range(len(data)):
52                                          data[i] = float(data[i])
53
54                                  return data
55
56          #will return None if data is bad, which will prompt the while loop to close
57          return None
58
59
60  def Spherical2Cartesian(r, phi, theta):
61          """
62          Converts from Spherical Coordinates to Cartesian Coordinates
63          Assumes phi and theta are in degrees
64          """
65
66          x = r * np.sin(np.radians(phi)) * np.cos(np.radians(theta))
67          y = r * np.sin(np.radians(phi)) * np.sin(np.radians(theta))
68          z = r * np.cos(np.radians(phi))
69
70          return x, y, z
71
72
73  def Polar2Cartesian(r, theta):
74          """
75          Converts from Polar Coordinates to Cartesian Coordinates
76          Assumes phi and theta are in degrees
77          """
78
79          x = r * np.cos(np.radians(theta))
80          y = r * np.sin(np.radians(theta))
81
82          return x, y
83
84
85  def Update3DData(x,y,z, xarray, yarray, zarray):
86          """
87          Updates the 3D data based on the continual readings from Arduino.
88          """
89
90          xarray.append(x)
91          yarray.append(y)
92          zarray.append(z)
93
94          return xarray, yarray, zarray
95
96
97  def Update2DData(x, y, xarray, yarray):
98          """
99          Updates the 2D plot based on the continual readings from Arduino.
100         """
101
102         xarray.append(x)
103         yarray.append(y)
104
105         return xarray, yarray
106
107
108 if __name__ == "__main__":
109
110         dim = raw_input('We doing 2D or 3D?\n\n')
111
112         #data arrays
113         xarray = []
114         yarray = []
115         zarray = []
116
117         #counter which checks to see if sufficient time has passed before exiting while loop
118         stop_counter = 0
119
120         #keeps the loop looping
121         start = True
122
123         #starts up serial port, and sends a command to begin Arduino processes
124         ser = StartUp()
```

```
        time.sleep(2)
        ser.write('start')

        while start:

                #reads a datapoint
                datapoint = ReadArduino(ser)

                if datapoint is not None:

                        #decides if 3D or 2D, and updates data arrays accordingly
                        if dim == "3d":
                                coordinates = Spherical2Cartesian(datapoint[0], datapoint[1], datapoint[2])
                                xarray, yarray, zarray = Update3DData(coordinates[0], coordinates[1], coordinates[2], xarray,
                                     yarray, zarray)

                        elif dim == '2d':
                                coordinates = Polar2Cartesian(datapoint[0], datapoint[1])
                                xarray, yarray = Update2DData(coordinates[0], coordinates[1], xarray, yarray)

                #stops the loop once data stops coming through
                else:
                        stop_counter += 1
                        if stop_counter > 5:
                                start = False


        #decides if 3D or 2D, and writes to a csv file accordingly
        if dim == "3d":

                file = open('test3d.csv','wb')

                writer = csv.writer(file, delimiter=',')

                writer.writerow(xarray)
                writer.writerow(yarray)
                writer.writerow(zarray)

                file.close()

        if dim == "2d":
                file = open('test2d.csv','wb')

                writer = csv.writer(file, delimiter=',')

                writer.writerow(xarray)
                writer.writerow(yarray)

                file.close()

        #ensures closure of the serial port
        ser.close()
```

Matlab Code

```
%Visualizer for Python processed data.


%reads from 3D data file
data = csvread('test3d.csv');

%splits into 3 arrays, one for each axis.
xlist = data(1,:);
ylist = data(2,:);
zlist = data(3,:);

%plots a scatter plot to represent data.
scatter3(xlist, ylist, zlist);
title('3D Scanner Visualization')
xlabel('X Distance (cm)');
ylabel('Y Distance (cm)');
zlabel('Z Distance (cm)');
axis([-60 60 -50 100 -50 100])

%reads from 2D data file
data2 = csvread('test2d.csv');

%splits into 2arrays, one for each axis.
xlist = data2(1,:);
ylist = data2(2,:);

%plots a scatter plot to represent data.
figure
scatter(xlist, ylist, 'filled');
title('2D Scanner Visualization')
xlabel('X Distance (cm)');
ylabel('Y Distance (cm)');
axis([-40 40 0 50])
```