PoE Lab 2: 3D Scanner

Braden Oh, Ashley Swanson

1 Introduction

In this lab we used an Arduino, two servos, and an infrared point-range finder to build a 3D scanner. The scanner had two dimensions of freedom (pitch and yaw) and logs a pitch angle, yaw angle, and distance measurement which is parsed by a Python script to be converted into a 3D plot of Cartesian coordinates.

2 Hardware Design

2.1 Servo Motors

The scanner is driven by two Futaba S3003 servo motors with an operating voltage of 5V and a pulse width control of 1.52 ms at a 3-5V square wave. The servos have three leads: positive, ground, and signal. Positive and ground were connected to the Arduino 5V and GND pins and the signal cables were connected to digital pins 6 and 7 for tilt and pan, respectively. These digital pins transmitted the square wave which

2.2 Infrared Range Sensor

The IR sensor is a Sharp GP2Y0A02YK0F Distance measuring sensor with an operating voltage of 5V. The sensor has three leads: positive, ground, and output. While operating, the sensor typically returns an analog measurement between 0V and 3V. Accordingly, we connected the output wire to the Arduino at analog pin 0, which returns an integer value between 0 and 1023, corresponding to 0V and 5V, respectively. The scanner firmware converts this integer value back into a voltage by multiplying the integer by a unit conversion factor of $\frac{5}{1023}$. The firmware then exports this voltage to Python for post processing.

It is also important to note that the measured voltage corresponds non-linearly to distance. In other words, if the distance measured between sensor readings of 0.5V and 1V is 60cm, the distance measured between readings of 1V and 1.5V will not be 60cm (and is in fact much closer to 20cm). More on this non-linear relationship later, in the Calibration section.

2.3 Coordinate Frames and Scanner Housing

The primary design constraint of the sensor housing is to ensure that the axes line up in a way that minimizes post processing of the data. To do this, we designed a housing geometry in which the axes of rotation of the two servos intersect perpendicularly (resulting in two normal basis vectors), and the center of the sensor receiver aligns with this intersection point when viewed from the front (resulting in a third basis vector that is normal to the first two), as shown in Figure 1. This geometry means that the sensor's line-of-sight vector always passes through the origin, so to account for offset from the origin we simply add the thickness of the sensor to each measured range to obtain the full radius at which each point exists in spherical space.

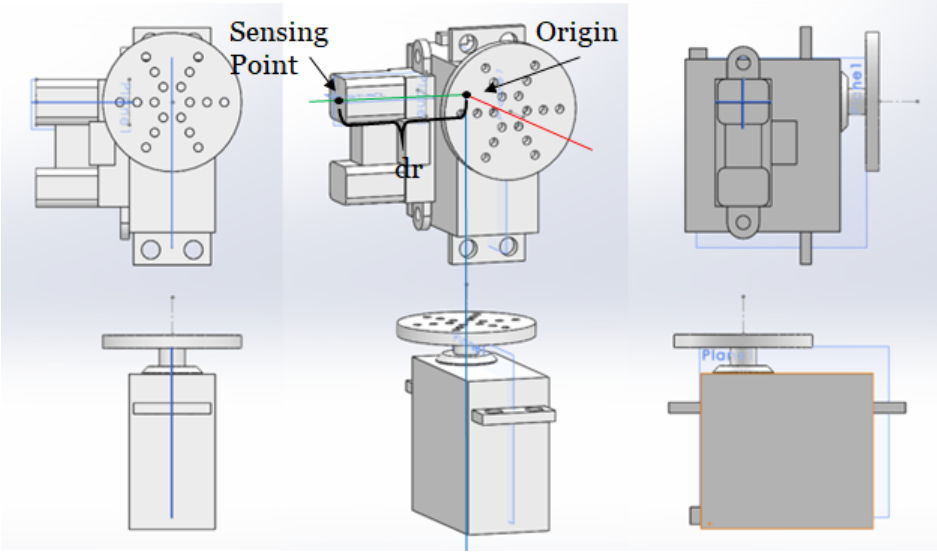

Figure 1: CAD diagrams of motor geometry. The axes of rotation for each motor are aligned and intersect at the global origin. The sensor line-of-sight vector intersects this origin, simplifying the post-processing process.

Once the geometry was set, we designed a three-piece pan-tilt mechanism to house the servos and the sensors. The pan servo connects to the base, which extends up to a ring with a pivot point for the tilt portion. The tilt servo attaches to a protrusion in the ring and is encapsulated by a front and back shell. These shells come to a point on the left side and insert into a pivot point directly across the ring from the protrusion. The design of this housing aims to pair functionality, principally prioritizing the initial geometry constraint, with aesthetics, a component achieved by the round rings and curves throughout the design (Figure 2). The entire case was CAD'ed in SolidWorks and 3D printed.

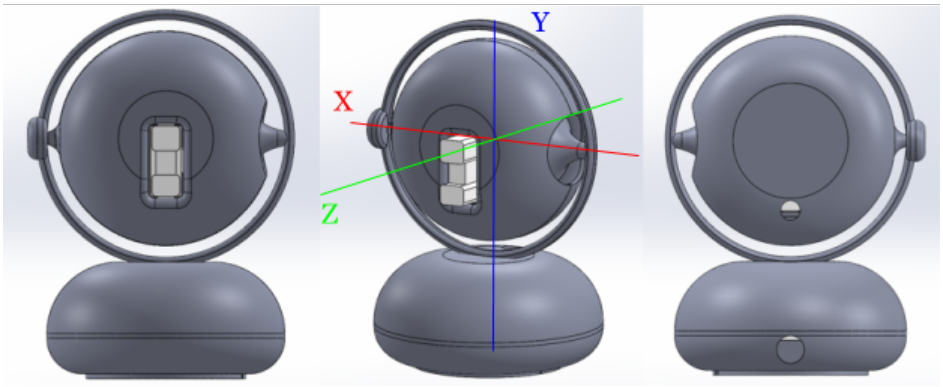

Figure 2: CAD diagrams of assembled case with overlaid basis vectors showing the origin and orientation of the global Cartesian coordinate frame.

3 Circuit Design

The circuit is fairly simple, consisting of only three components: the IR distance sensor and two servos. Each of these components requires a connection to power and ground, so we attached each piece of hardware to the Arduino 5V output and to Arduino ground. To control the motors, we attached the servo signal connections to digital pins 6 and 7 for tilt and pan, respectively. These pins output a pulse-width-modulation square wave to set the various servo angles. To interface with the IR sensor we attached the output wire to analog pin 0. This allows the Arduino to take a measurement of the raw voltage output by the sensor. A circuit diagram is given in Figure 3.

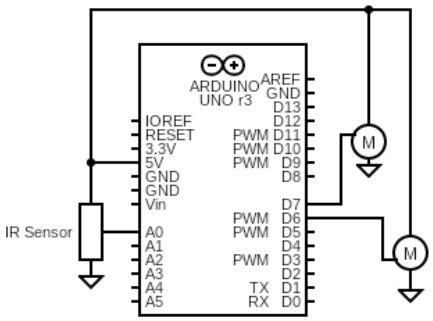

Figure 3: Wiring diagram showing the connections between the Arduino, two servo motors, and an IR distance sensor.

3.1 Protoshield

This circuit, while simple on paper, is easy to build incorrectly, as it quickly becomes difficult to discern the difference between the various signal cables on a breadboard. To help with this problem, Professor Brad Minch provided us an Arduino "Protoshield" of his own design. This printed circuit board fits perfectly over an Arduino and provides a blank set of rails and holes for pins that allow for easy circuit assembly and cable management. A photo of our assembled Protoshield is given in Figure 4.

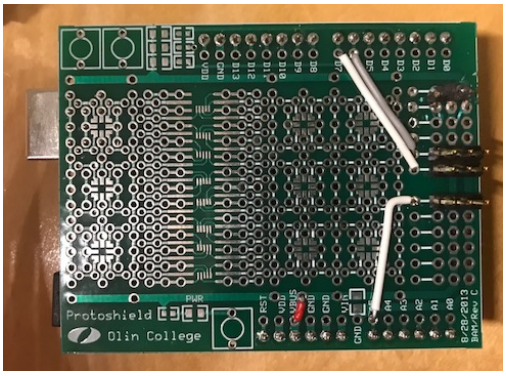

Figure 4: Photo of assembled Protoshield. The sets of three pins on the right side of the board allow easy plug-and-play connection of motors and sensors.

4 Software Design

The system's software is split into three parts: firmware, post-processing, and rendering. The firmware consists of motor/sensor control code onboard the Arduino and is written in Arduino C. The post-processing script is executed on an attached computer and converts raw measurements from the sensor into Cartesian coordinates. It is written in Python. Our data visualization ended up being done in MATLAB, as Python's *matplotlib* library proved unreliable.

4.1 Firmware (Arduino C)

The firmware was divided into three functions: *setup()*, *loop()*, and *capture()*. The former two are special functions required by the Arduino: *setup()* handles code thttps://www.overleaf.com/project/5ba6db555e1a1 needs to be run only once - upon initialization - while *loop()* contains the body of code that the Arduino runs on repeat during its duration of operation. The latter function, *capture()*, is called throughout the main loop and takes a single range measurement.

4.1.1 *setup()*

Upon initialization, the *setup()* function is run once. This function instantiates two Servo objects, *panServo* and *tiltServo*, attaching them to digital pins 7 and 6, respectively. These Servo objects belong to the Arduino Servo class and provides for easy control of the motors without manually having to perform pulse-width-modulation. This function also opens a serial connection.

4.1.2 *loop()*

The main body of program is looped indefinitely in the *loop()* function. The main capability of this function is to collect range measurements while moving through a pre-programmed path; this path traverses the spherical space being scanned. We implemented this as a series of tilt and pan commands sent to the corresponding servo. Before scanning, the system adjusts to point at the bottom right-hand corner of the scan space and then writes a string reading *START* to the serial port. This string tells Python to begin parsing scan data. To execute a scan, the system first tilts upwards (about the X axis), taking measurements at every one degree of tilt. When it reaches the top edge of the scan space, it pans one degree to the right (about the Z axis), and then tilts back downwards, again taking measurements at every one degree of tilt. When it reaches the bottom edge of the scan space, it pans another degree to the right and repeats. This tilt up-pan-tilt down-pan sequence continues until the far left edge of the scan space is reached.

We chose to scan through tilts because it causes our pan servo, which bears the load of the entire mechanism, to turn minimally. Upon further investigation into the data sheet, we find that this transversal is opposite from from the ideal use of the sensor, given its orientation (i.e. the data sheet recommends parsing back-and-forth horizontally, rather than vertically). However, upon testing the inverse configuration, no clear differences appear in the scan output, so we chose to stick with our vertical implementation for its mechanical advantages. The similarities between the two scans most likely results from lengthy pauses between scans, as the sensor has time to take a clear reading (more on pausing below in the capture() section).

After taking a measurement, this function prints a string to the serial connection of the form *panAngle,tiltAngle,voltage*, where *panAngle* and *tiltAngle* are integer angle values and *voltage* is a float voltage measurement. This string is easily parse-able by Python. At the conclusion of a

scan, the function prints the string *STOP* three times. This string tells Python that the scan has completed and to stop parsing scan data.

4.1.3 *capture()*

The final function in our firmware is *capture()*. Measurements are taken by calling this function, which takes a set of measurements from the sensor and averages them together, reducing the amount of noise that could potentially arise from sensor latency. The latency of the sensor (the time it needs to point in a given direction before it can return an accurate range measurement) is known by the teaching team to be around 50 ms. To give ourselves margin, this function waits 70 ms between measurements. It takes a set of five measurements, each with a latency of 70 ms between it and the last measurement, and returns the mean of the set of voltages.

4.2 Post-Processing (Python)

To convert the Arduino's readouts into useful coordinates, we wrote a Python script that listens on the port which Arduino is writing to, parses the raw output, and converts pan, tilt, and voltage measurements into Cartesian coordinates. When the script is called, it listens on the Arduino serial port until it sees the string *START*. At that point, the script enters a loop and continuously checks its buffer for new data as Arduino prints measurements to the serial port. When reading a valid string, the Python script splits it into an array of numbers (int, int, float, as explained in the last section) and converts from spherical coordinates to Cartesian coordinates by the following expressions:

$$x = Rcos(\theta)cos(\phi)$$
$$y = Rcos(\theta)sin(\phi)$$
$$z = Rsin(\theta)$$

where R is the range measured by the sensor (not the raw voltage, more on this in the Calibration section), θ is the tilt angle (about the X axis) and ϕ is the pan angle (about the Z axis). These (x, y, z) coordinates are calculated and stored in arrays.

Python continues checking the buffer for new measurement data until it sees the string STOP, at which point it breaks from its parsing loop and proceeds to process the Cartesian coordinates it has been storing in arrays. The first processing step it performs is to write the data to a CSV file: the script asks the user to enter a filename and then saves the x, y, and z coordinate arrays it has as columns in a CSV file (this is done by writing the arrays to a Pandas DataFrame object which has a built-in function called *to_csv()*). This both saves the data for future use and exports it to a format easily parsable by other languages, such as MATLAB. The second processing step this script performs is generating a 3D scatter plot with the Python library *matplotlib*. We found these scatter plots to have uneven axes and to be generally difficult to read, however, and so our final plots were made in MATLAB.

5 Calibration

In order to convert the voltage readout of the IR sensor to a meaningful distance (given by R, in the equations above), we created a calibration curve by measuring the voltage at known distances and

plotting the resultant voltage readouts. Unfortunately this is not a linear relationship, and we ended up fitting the following cubic curve to our calibration points using Python's *scipy.optimize.curve_fit* function:

$$V = -6.08552875 \times 10^{-5}x^3 + 6.68673717 \times 10^{-3}x^2 + -2.59185447 \times 10^{-1}x + 4.20527041$$

where V is a voltage corresponding to a given distance x. This function allows us solve for x given any measured voltage V. A plot of this curve against our calibration dataset is given in Figure 5.

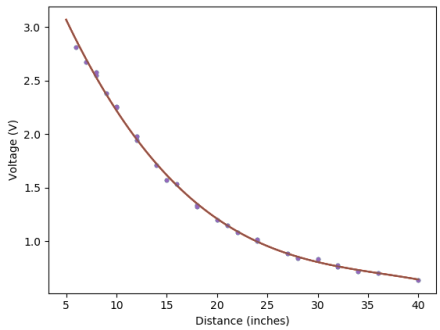

Figure 5: Graph of our calibration curve superimposed on our calibration dataset

5.1 Calibration Error

In order to test our calibration, we measured the voltage readout for 5 points that were not included in our original data set. As the plot given in Figure 6 illustrates, our voltage measurement is consistently below the estimated output. In this set of test cases, our average percent error is -8.46%, meaning our scans will output values that read, on average, 8.46% closer to the scanner than they actually are.

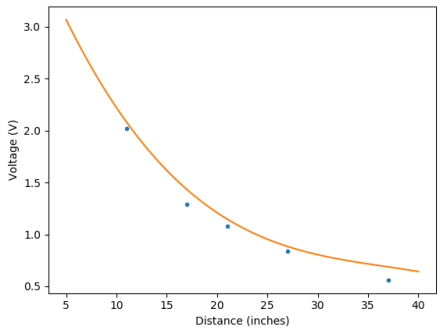

Figure 6: Graph of our calibration curve superimposed on our calibration dataset

6 Scanning

6.1 Single Line Scan

Figure 7 shows the output of a single vertical slice of the scanning area (a cardboard letter E). Because we scan in vertical slices rather than horizontal slices, a full 3D plot is a stack of vertical slices like this one. The points are not evenly spaced due to a previously mentioned depth filter, which eliminates ranges found beyond a certain distance.

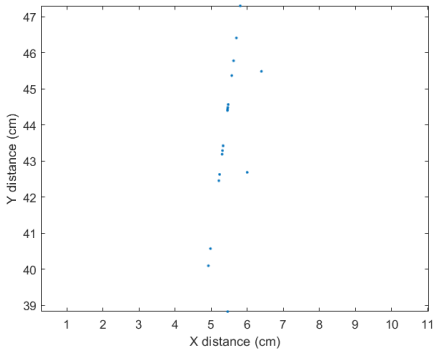

Figure 7: Graph of a single (vertical) cross-section of our letter E

6.2 Full 3D Scan

Figure 8 shows the output of a full scan of our area. The resulting point cloud clearly shows the surface of our cardboard letter E, crisply given the depth filter that eliminates noise from objects near the edge of the scanner's range. The surface appears as a relatively flat plane, largely an effect of averaging multiple range measurements for each point. Points in the plane ranged by about 3 cm in the y direction, an error of around ±1.5cm from an even plane.

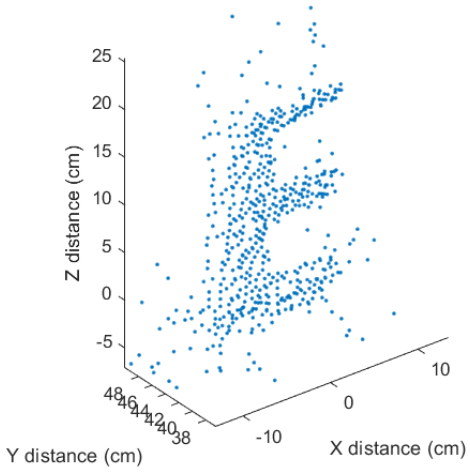

Figure 8: Graph of our complete 3D scan of the letter E

Figure 9 shows a photo of our testing setup; Arduino controlling the scanner, Python reading the serial output of the Arduino, and MATLAB rendering the final plot. The sunlight-blocking backboard was beyond the filter depth of the scanner and so does not appear in the final plot.

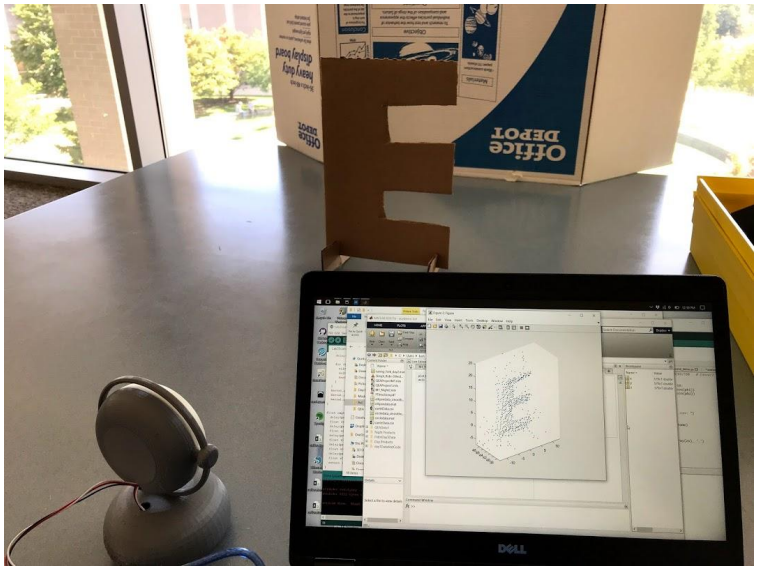

Figure 9: Photo of our test setup

7 Appendices

7.1 Appendix 1: Firmware Code (Arduino C)

```
#include <Servo.h>

//-------------------------------------------------------------------------
// GLOBAL VARIABLES
int pan = 7;        // digital pin 7
int tilt = 6;       // digital pin 6
int sensor = 0;     // analog pin 0
float voltage = 0.0;
int pause = 70;     // Latency between reading sensor voltages (ms)
Servo panServo;     // Servo object for the pan motor
Servo tiltServo;    // Servo object for the tilt motor
int panAngle = 0;   // Pan angle in degrees
int tiltAngle = 0; // Tilt angle in degrees

//-------------------------------------------------------------------------
// Function: setup()
//
// Function required by Arduino.  Executes one-time setup to be
// performed upon initialization.
```

```
//-------------------------------------------------------------------
void setup() {
  panServo.attach(pan);
  tiltServo.attach(tilt);
  Serial.begin(9600);
}

//-------------------------------------------------------------------
// Function: loop()
//
// Function required by Arduino.  "Main" function that loops for the
// duration of the Arduino's operation.
//-------------------------------------------------------------------
void loop() {

  // Point to the "center" of the scan area (directly in line with y-axis)
  panServo.write(30);
  delay(500);
  tiltServo.write(63);
  delay(4000);

  // Move to bottom right corner of scan area
  panServo.write(0);
  delay(500);
  tiltServo.write(45);
  delay(500);

  // Transmit a START signal to indicate Python should begin parsing
  Serial.println("START");

  // Loop through every pan angle...
  for (panAngle = 15; panAngle <= 45; panAngle += 1) {

    panServo.write(panAngle);
    delay(pause);

    for (tiltAngle = 53; tiltAngle <= 93; tiltAngle += 1) {
      tiltServo.write(tiltAngle);
      voltage = capture();
      Serial.println(String(panAngle) + "," + String(tiltAngle) + "," + String(volt
    }

    panAngle += 1;
    panServo.write(panAngle);
    delay(pause);

    for (tiltAngle = 93; tiltAngle >= 53; tiltAngle -= 1) {
      tiltServo.write(tiltAngle);
```

```
      voltage = capture();
      Serial.println(String(panAngle) + "," + String(tiltAngle) + "," + String(volt
    }
  }
  Serial.println("STOP");
  Serial.println("STOP");
  Serial.println("STOP");
}

//-----------------------------------------------------------------------
// Function: capture()
//
// This function captures a single voltage measurement.  It reads
// the analog pin voltage 5 times at a given latency and then
// returns the average of the 5 measurements
//-----------------------------------------------------------------------
float capture() {
  delay(pause);
  float v1=analogRead(sensor) * (5.0 / 1023.0);
  delay(pause);
  float v2=analogRead(sensor) * (5.0 / 1023.0);
  delay(pause);
  float v3=analogRead(sensor) * (5.0 / 1023.0);
  delay(pause);
  float v4=analogRead(sensor) * (5.0 / 1023.0);
  delay(pause);
  float v5=analogRead(sensor) * (5.0 / 1023.0);
  return (v1 + v2 + v3 + v4 + v5) / 5;
}
//-----------------------------------------------------------------------
```

7.2 Appendix 2: Post-Processing Code (Python)

```
from mpl_toolkits.mplot3d import Axes3D
import scipy.optimize
import math
import numpy as np
import serial
import time
import matplotlib.pyplot as plt
import pandas as pd

# initialize useful variables
arduinoComPort = "COM4"
baudRate = 9600

# open the serial port
```

```
serialPort = serial.Serial(arduinoComPort, baudRate, timeout=1)


#----------------------------------------------------------------------------
# FUNCTION DEFINITIONS

# f
# Describes calibration curve given an x distance and y voltage
def f(x, y):
    return (-6.08552875e-05*x*x*x)+(6.68673717e-03*x*x)+(-2.59185447e-01*x)+(4.2052

# volt2dist
# Uses calibration curve function to convert a voltage to a distance
def volt2dist(voltage, func=f, x0=20):
    return scipy.optimize.fsolve(func, x0, voltage)[0]*2.54  # Convert in to cm

def get_wrekt():
    return (get_wrekt())

#----------------------------------------------------------------------------

# Infinite loop until we see a START signal from the scanner
start = False
while not start:
    lineOfData = serialPort.readline().strip()
    if lineOfData != "START":
        start = True
        print("Starting scan...")

# Once we begin, create empty arrays for the x, y, and z coordinates
xs = []
ys = []
zs = []

# Keep looping while data is coming from the scanner...
while True:

    # Grab the next line in the buffer
    lineOfData = serialPort.readline().strip()

    # When the scanner says STOP break out of the infinite loop
    if lineOfData == "STOP":
            break

    # Check that the string contains valid data
    if len(lineOfData.split(','))==3:

        data = lineOfData.split(',')
```

```
            print(data)
            # data[0] = pan angle = phi
            # data[1] = tilt angle = theta
            # data[2] = voltage

            r = volt2dist(float(data[2])) + 2.5   # Add sensor distance from origin
            phi = (float(data[0]) + 60)*3.14159265/180     # Convert to radians ):
            theta = (float(data[1]) - 61)*3.14159265/180   # Convert to radians

            # If the point is closer than 50 cm, include it, otherwise ignore it.
            # This serves as a filter that captures only points within a certain
            # maximum distance.
            if r*math.cos(theta)*math.sin(phi) < 50:
                xs.append(r*math.cos(theta)*math.cos(phi))
                ys.append(r*math.cos(theta)*math.sin(phi))
                zs.append(r*math.sin(theta))

# Once we've broken out of the loop we need to clean up
serialPort.close()
print("Scan finished, serial port closed")

# Allow the user to save the data to a .csv file
name = raw_input("Enter a filename ending in .csv: ")
d = {'x':xs, 'y':ys, 'z':zs}
data = pd.DataFrame(data=d)
data.to_csv('C:\Users\\boh\Documents\PoE\\'+name)

# Generate a matplotlib plot (WARNING - THIS GENERATES BAD PLOTS)
fig = plt.figure()
ax = fig.add_subplot(111, projection='3d')
ax.scatter(np.array(xs), np.array(ys), np.array(zs), '.')
ax.set_xlabel('X')
ax.set_ylabel('Y')
ax.set_zlabel('Z')
plt.show()
```