3D Scanner - Time of Flight Edition

Anusha Datar
Quinn Kelley

September 25, 2018

1 Overview

The objectives of this lab were to develop a simple 3D scanner that utilized a pan/tilt mechanism and used it to visualize the results of scanning a known object. We decided to trial the experimental version of this lab, which replaced the infrared distance sensor with a time-of-flight sensor and required that we scan a convex object instead of a known surface.

2 CAD

We designed the housing for this model in SolidWorks, and we took care to tolerance the model by several millimeters to ensure a proper fit. Then, we 3D printed it. We experienced our first problem at this stage: because we lacked experience with print settings, we selected twenty percent infill, and our printed parts were quite weak. Another issue was that the arm that held the "tilt" servo was printed sideways with no support material, so it experienced slight deformation. Nevertheless, the pieces fit fairly well and the assembly held up through many scans.

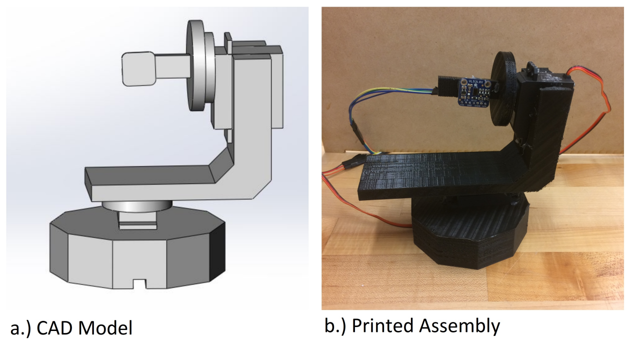

Figure 1: Pan/Tilt System

3 Circuit

A standard Arduino Uno was used as the central processing and power distribution unit for this scanner. The actuators for the pan and tilt functionality were both Vigor Precision VS-2MB servo motors driven by digital pins. The sensing unit, an Adafruit Time of Flight LIDAR sensor, used I2C communication to relay information to the Arduino. See Appendix A.1 for a full circuit diagram and Appendix A.2 for an image of the circuit.

4 Calibration

We determined the relationship between a target object's distance and the sensor's voltage by configuring the sensor to print values averaged over a known time to the Serial console, holding a board at various known distances away from the sensor and recording the output, and then fitting the curve in MATLAB (see Figure 2). We then took several distances, used the calibration curve to predict the output voltage, and then measured the actual voltage to validate our model. Plotting this (Figure 3) showed us that our calibration curve was fairly accurate, as the test data points did not stray very far from the line, which had the equation $Voltage = 8.8 * (distance\ in\ mm) + 80$ We used this equation in our python script to find the radial distance measured by the sensor.

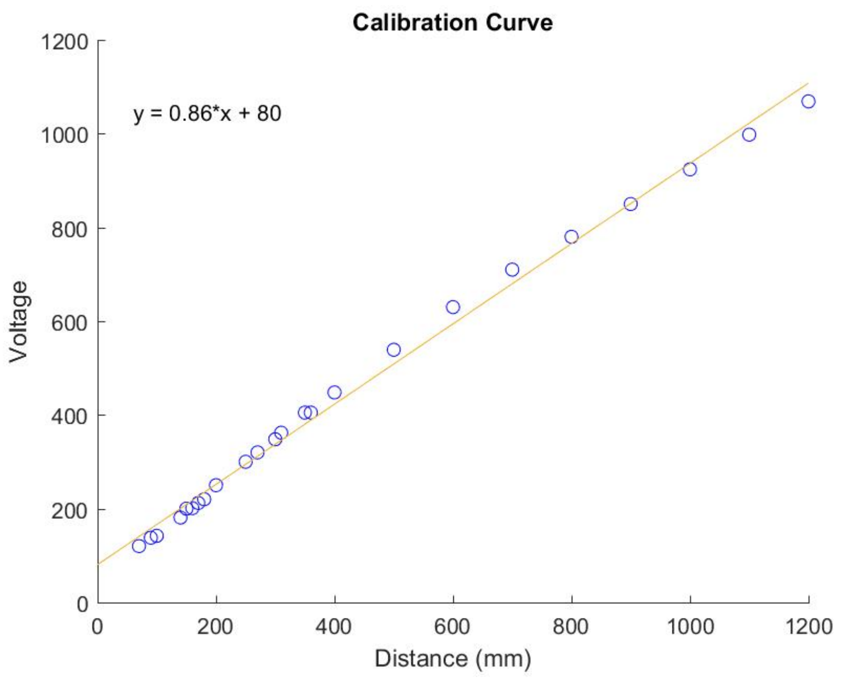

Figure 2: Sensor Calibration Curve

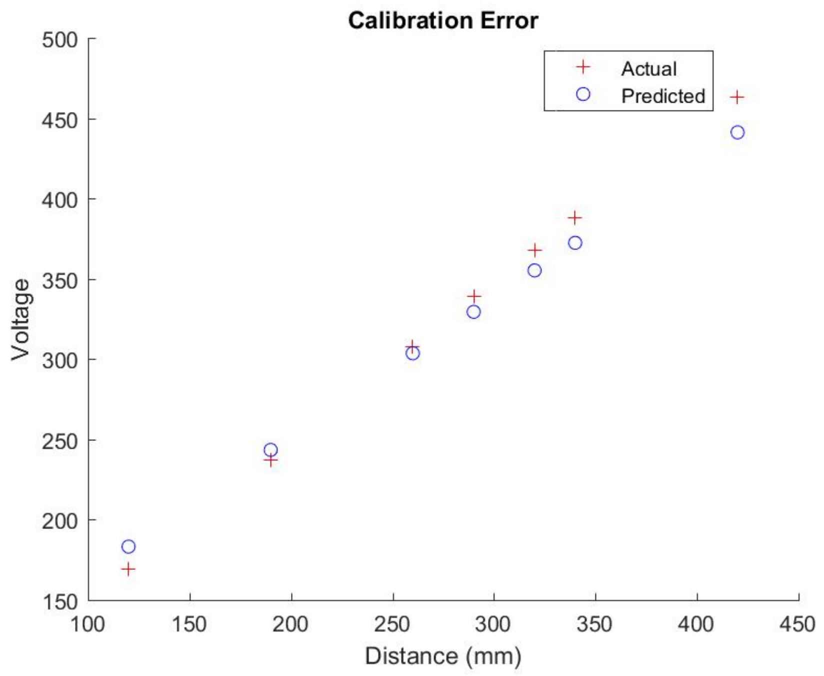

Figure 3: Calibration Error Curve

5 Software

The software associated with this lab contains two distinct sections: the Arduino code for the scanner itself and the python code to process the data. On the Arduino, the user specifies the ranges for both of the motors, and the Arduino directs the servo to begin at the minimum point for each motor, sweep all possible points within that range, and take the angle of each servo and the distance from the time-of-flight sensor at each step and transmit it over serial so it can be processed on a laptop by the python script. See Appendix B.1 for Arduino source code.

The goal of the python script was to take the Arduino data from the serial port, process it, and generate a three-dimensional visualization of the scanned object. We made a main python script to receive and process a data and a utilities module with conversion and plotting functions. Overall, the script processes the incoming serial data, which is in spherical coordinates, adjusts the angles to place the center of the scanning cone at the origin, and it converts the spherical coordinates to cartesian coordinates using the equations:

$$X = \rho * sin\phi * cos\theta \tag{1}$$

$$Y = \rho * sin\phi * sin\theta \tag{2}$$

$$Z = \rho * cos\phi \tag{3}$$

where the radial distance of the sensor $\rho = (8.6*\text{Measured Distance in mm}+80)$, which was the empirical calibration curve. In order to account for the reversal of the azimuth due to the initial sensor placement, $\phi = \text{Sensor longitude angle} - 120\,^{\circ}$. To shift to the correct range in latitude, $Theta = \text{latitude angle} - 120\,^{\circ}$. The coordinate points were finally visualized using the matplotlib 3D scatter plot feature. See Appendix B.2 for Python source code.

6 Results

We chose to scan this mug, expecting to see a cylindrical shape for the base, a vague outline of the handle, and some general distortion due to the mug's reflectiveness. Our results matched our predictions : we could recognize the handle and the general shape of the mug, but were not able to create the cleanest possible image. We also scanned the mug in front of a wall and were able to see the difference between the points on the wall and the points on the mug, suggesting that all possible points were selected and properly mathematically processed.

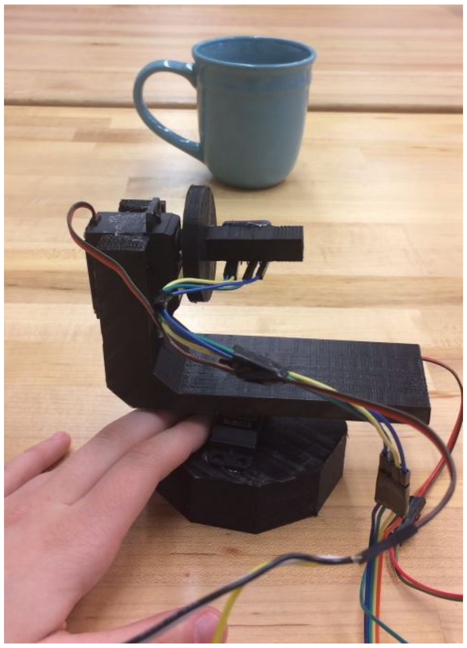

Figure 4: Scanning setup

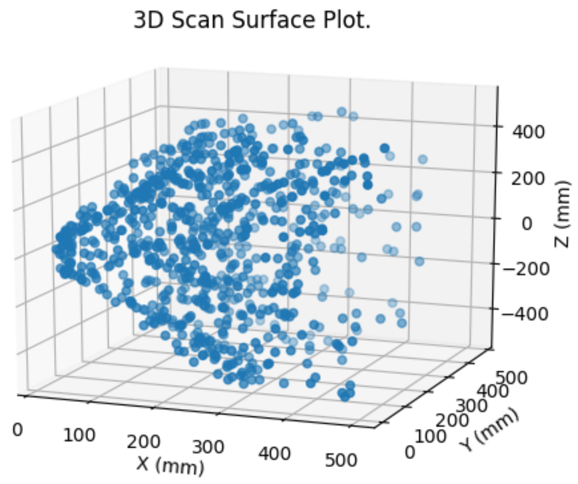

Figure 5: 3D scan of the Mug

7 Reflection

There were a few major lessons that we learned while completing this lab. We worked with serial and python, which was an interesting departure from exclusively Arduino-based processing. We especially gained experience with serial because we had a mysterious problem with the serial output where the output would stop sending data and only continue to transmit if we cleared the buffer: finding and debugging this issue was a frustrating and informative learning experience. We also learned to be more conscious about our data collection and processing methods: because of timing issues, we had a few jarring outliers in some of our test cases, which became confusing when we tried to interpret our data. Once we discovered them, found their causes (i.e. timing issues with the I2C input from the sensor), and eliminated them, our project went much more smoothly. A last problem we faced was with 3D printer settings - we used twenty percent infill and did not add as many supports as were necessary, so our parts were weak and we had to exercise caution with them - at the same time, we gained experience, and will not make the same mistakes in the future.

As for our thoughts on the experimental portion of this lab, we do not feel like our learning experience was divergent from the infrared sensor lab, but we do feel as though we had to deal with more (relatively) arbitrary roadblocks: the I2C protocol created a few issues, even with the Adafruit library, the results were hard to interpret and the measurements were not very clean, and the difficulty of finding an adequate convex object (such that we could confirm that the orientation was correct and that was not too reflective).

A Appendix A

A.1 Circuit Diagram

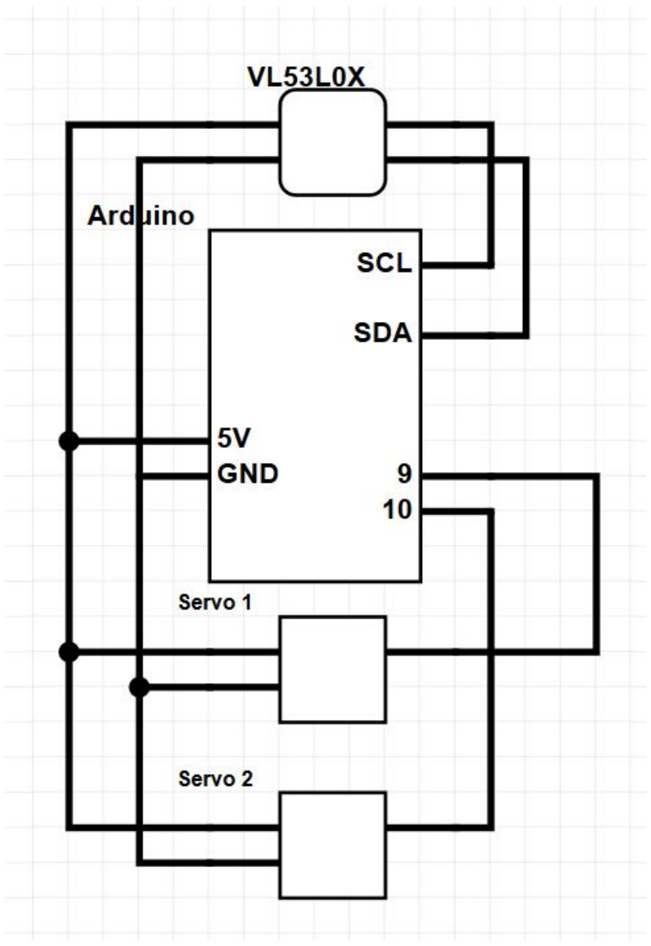

Figure 6: Circuit diagram

A.2 Circuit Image

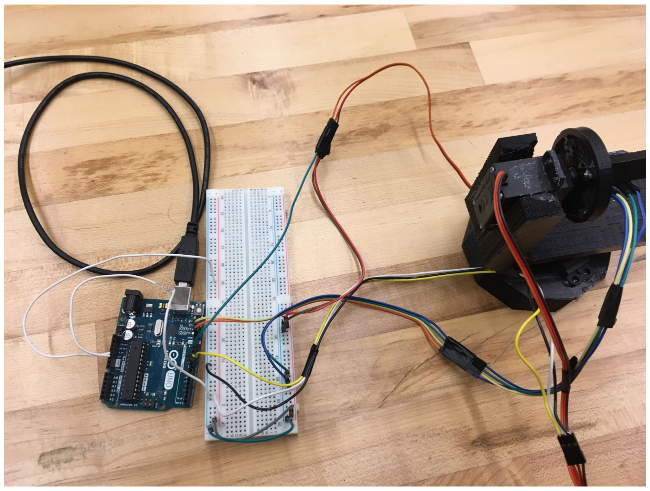

Figure 7: Circuit Image

B Appendix B

B.1 Arduino Source Code

```
 * Transmits servo angle and distance sensor data
 * for each servo step over serial for further
 * analysis. This code is intentionally generic and lightweight
 * as all processing occurs onboard a laptop with python instead
 * of on the microcontroller.
 */

// Grab the libraries for the time-of-flight sensor and for the servos.
#include "Adafruit_VL53L0X.h"
#include "Servo.h"

// The sensor works over I2C, so no need to worry about pin mappings for that.
Adafruit_VL53L0X lox = Adafruit_VL53L0X();
// Servos need pin mappings, declarations, bounds, and position counters.
// Keeping naming conventions generic to emphasize post-processing.
Servo servoA;
Servo servoB;
int servoApin = 9;
int servoBpin = 10;
int servoApos = 0;
int servoBpos = 0;

int servoAmin = 45;
int servoAmax = 135;
int servoBmin = 45;
int servoBmax = 100;
```

```

void setup() {
  // Kick off Serial. Be sure to match baudrate on python script.
  Serial.begin(9600);
  while (! Serial) {
    delay(1);
  }
  // Make sure that the sensor boots before moving any motors.
  if (!lox.begin()) {
    // DEBUG
    // Serial.println(F("Failed␣to␣boot␣VL53L0X"));
    while(1);
  }
  // DEBUG
  // Serial.println(F("Booted␣sensor."));
  // Declare servos.
  servoA.attach(servoApin);
  servoB.attach(servoBpin);
  // Confirm that the servos go home.
  zero_servos();
}

void loop() {
  /*
   * Print a consistent data structure of distance measurement and each servo position
   * over serial so it can be processed by the python script. The format should be
   * [DISTANCE_READ (mm), SERVOA_ANGLE, SERVOB_ANGLE]
   */
  VL53L0X_RangingMeasurementData_t measure;

  for (servoApos = servoAmin; servoApos <= servoAmax; servoApos += 1) {
    servoA.write(servoApos);
    delay(15);
    for (servoBpos = servoBmin; servoBpos <= servoBmax; servoBpos += 1) {
      servoB.write(servoBpos);
      delay(15);
      // Send data from the sensor if there is no phase failure. Otherwise, just
      // move on to the next angle because we can try to interpolate in post.
      lox.rangingTest(&measure, false); // pass in 'true' to get debug data printout!
      if (measure.RangeStatus != 4) {
        // Sends appropriate values over serial.
        Serial.print("Distance:␣");
        Serial.print(measure.RangeMilliMeter);
        Serial.print("␣");
        Serial.print(servoApos);
        Serial.print("␣");
        Serial.print(servoBpos);
        Serial.print("\n");
      }
    }
  }
  // Continue for multiple scans if appropriate.
  delay(500);
  zero_servos();
  delay(100);
}

```

```
void zero_servos() {
  /*
   * Bring servos back to their original positions at the start of processing.
   */
  servoA.write(servoAmin);
  delay(15);
  servoB.write(servoBmin);
  delay(15);
  delay(200);
}
```

B.2 Python Source Code

We also created two Python programs with which to process the data - a quick program to process data from the serial port and a utilities file with function implementations.

B.2.1 Serial Processing File

```python
#!/usr/bin/env python3
# Quinn Kelley and Anusha Datar
# Principles of Engineering Lab 2
# Serial-based processing test file.

"""
Listens to arduino over serial port to collect data to generate visualization.
Uses functions from utilities file.
"""
from utilities import *
import serial
import time

# Substitue in port number of the arduino. On most UNIX-based systems, the
# arduino will be at "/dev/ttyACM0" and on most Windows machines it will be
# at a COM port location.
PORT = "/dev/ttyACM0"
# Set the baudrate. This must match baudrate on the arduino program.
BAUD = 9600

# A list of all the cartesian points generated to plot later.
points = []
serialPort = serial.Serial(PORT, BAUD, timeout=10)
# Whether or not the scan has ended, based on servo angles from arduino code.
scanContinue = True

while scanContinue:
    try:
        data = serialPort.readline().decode().split()
        time.sleep(1);
        if len(data)==4:
            # Use the servo bounds from the arduino program. It could be cool
            # to have the arduino send a calibration message during setup
            # and grab those rather than harcoding them.
            if (int(cartesian_point[1]) == 120 and int(cartesian_point[2]) == 90):
                scanContinue = False
            cartesian_point = spherical_to_cartesian(cartesian_point)
            # Sometimes, because of timing difficulties, the sensor will send
            # out erroneous values, and these numbers will create huge outliers
            # that make the plots difficult to read. Eliminate these.
            valid_point = True
            for element in cartesian:
                # Threshold empirically determined; object under study was
                # much closer than this.
                if abs(element) > 600:
                    valid_point = False
            if valid_point:
                points.append(cartesian_point)
    except:
        # If something gets cut off on a measurement, move on to the next
        # received message, but wait to minimize residual errors.
        time.sleep(.5)
        continue

# Plot points following data collection.
create_three_dimensional_plot(points)
```

B.2.2 Utilities File

```python
# Utilities file.
"""
Because of shared functions with File IO and serial-based communication, this
file proved to be a valuable resource to maintain a consistent architecture
throughout test phases.
"""
import math
import matplotlib.pyplot as plt
from mpl_toolkits.mplot3d import Axes3D
import serial
import time


def spherical_to_cartesian(point):
    """
    Converts spherical point from pan/tilt mechanism into cartesian points.
    Note that it assumes a 0 to 120 servo-angle based cone for scanning and
    employs an empiricially determined calibration curve to find the radial
    distance measured by the sensor.

    point : A list as follows:
            [DISTANCE_MEASUREMENT, LATITUDE_ANGLE, LONGITUDINAL_ANGLE]
    returns : A list with the cartesian point equivalent of the input:
              [X, Y, Z]
    """
    distance = 8.6*int(point[0]) + 80
    # Our current angles are from 60 degrees to 120 degrees. Shift those
    # to be between -60 and 60 degrees.
    theta = int(point[1]) - 120
    # Subtract 120 + 90 from this one to account for reversal of azimuthal
    # direction because of existing sensor orientation.
    phi = int(point[2]) - 120
    # Spherical coordinate formula.
    x = distance*math.sin(math.radians(phi + 180))*math.cos(math.radians(theta))
    y = -distance*math.sin(math.radians(phi + 180))*math.sin(math.radians(theta))
    z = distance*math.cos(phi)
    cartesian_point = [x, y, z]
    return cartesian_point

def create_three_dimensional_plot(points):
    """
    Creates three dimensional visualization of data collected from scan.

    points : List of lists where each sub-list is a cartesian point
             [X, Y, Z]
    """
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    x_values = []
    y_values = []
    z_values = []
    for element in points:
        x_values.append(element[0])
        y_values.append(element[1])
        z_values.append(element[2])
    # Use the matplotlib 3D plot feature.
    ax.scatter(xs=x_values, ys=y_values, zs=z_values)
    ax.set_title("3D␣Scan␣Surface␣Plot.")
    ax.set_xlabel("X␣(mm)")
    ax.set_ylabel("Y␣(mm)")
    ax.set_zlabel("Z␣(mm)")
    plt.show()

def create_three_dimensional_surface(points):
    """
    Creates three dimensional visualization of data collected from scan.

    points : List of lists where each sub-list is a cartesian point
             [X, Y, Z]
    """
    fig = plt.figure()
    ax = fig.add_subplot(111, projection='3d')
    x_values = []
    y_values = []
    z_values = []
    for element in points:
        x_values.append(element[0])
        y_values.append(element[1])
        z_values.append(element[2])
    # Use the matplotlib 3D plot feature.
    ax.plot_trisurf(x_values, y_values, z_values)
    ax.set_title("3D␣Scan␣Surface␣Plot.")
    ax.set_xlabel("X␣(mm)")
    ax.set_ylabel("Y␣(mm)")
    ax.set_zlabel("Z␣(mm)")
    plt.show()
```