PIE Mini Project 2: 3D Scanner

September 26, 2025

Ben Ricket, Nathaniel Banse

Introduction:

This project implements a 3D scanner using two servos and an infrared distance sensor, allowing for the construction of a three dimensional point cloud depicting the shape of a scanned surface. The main challenges associated with this project include converting between sensor output voltage and distance, moving the scanner to different angles using the servos, communicating the sensor output and servo angle over serial, converting the sensor data from spherical coordinates to cartesian coordinates, and visualizing the point cloud.

Our completed scanner consists of two servos, a distance sensor, PLA mounts to support the sensor and servos on the table and hold them together, an Arduino Uno running a script controlling the servos and writing the analog sensor data over a serial connection, and a Python script running on a connected computer to read, process, and visualize the serial data.

Sensor Calibration:

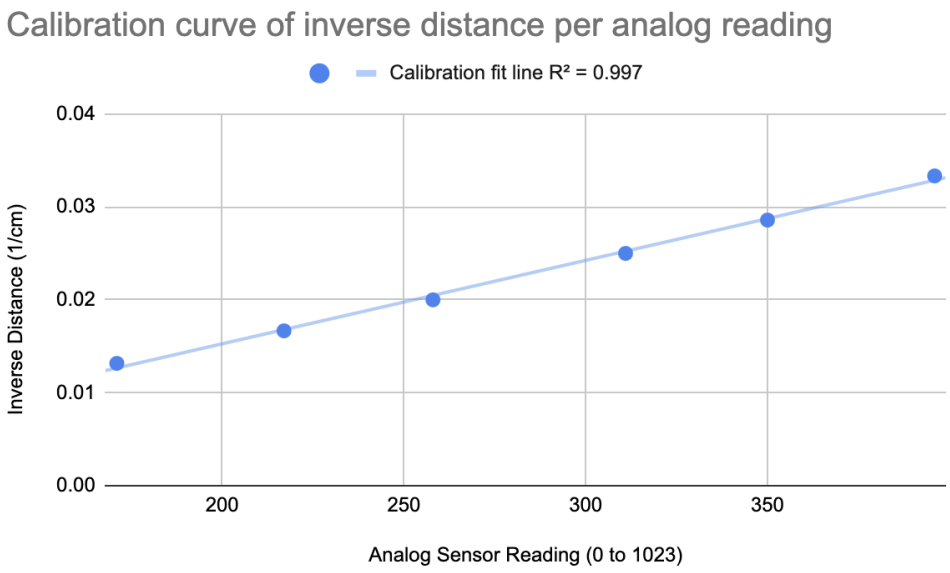

Figure 1: Graph displaying the data points we used to fit our calibration line and the resulting calibration line.

We calibrated our sensor by disconnecting the servos, and holding it a fixed distance from a sheet of cardboard. We used a measuring tape to get the perpendicular distance from the sensor to the sheet of cardboard. The data sheet shows that the sensor voltage is proportional to the inverse of the distance when between around 30 and 150 centimeters. The arduino reads the analog input as an int between 0 and 1023, but is proportional to voltage. As such, we don't need to convert to voltage before fitting our calibration line, we can fit a line directly to the relationship between inverse distance and the arduino analog input. Each data point was calculated by taking the average of ten sensor readings close in time, to help correct for noise from the sensor or arduino.

We fit a line to the inverse of the distance as a function of the sensor reading and recorded an R^2 value of 0.997. Given this high coefficient of determination, we concluded that our linear fit was a reasonable choice for our given range of distances, and validated it by testing on more distances within that range.

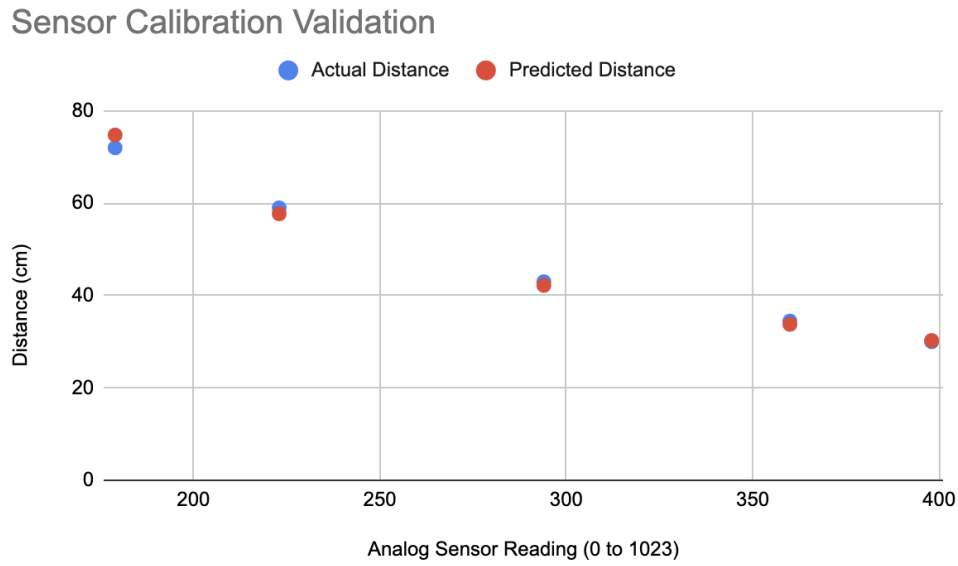

Figure 2: Graph validating our sensor calibration by comparing the measured (with a tape measure) and calculated (with our distance sensor) distance.

After validating our calibration curve on five more data points, we calculated an average percent error (absolute value of predicted distance - actual distance) of 2.39%. Given this result, we concluded that our calculated transfer function would be accurate for the distance ranges we

tested, and went on to use our calculated slope and intercept for our data processing. The calibration function is implemented in figure 3.

```
for index,reading in enumerate(readings):
    if reading < 420 and reading > 150:
        dist_list.append(1/(INVERSE_CM_PER_READING*reading + INVERSE_CM_AT_ZERO))
        pitches.append(pitch_list[index])
        yaws.append(yaw_list[index])
```

Figure 3: Code snippet depicting the implementation of the transfer function

We implement the transfer function of the distance sensor by defining two constants, INVERSE_CM_PER_READING and INVERSE_CM_AT_ZERO, which correspond to the slope and intercept of the line expressing inverse distance as a function of the sensor reading. We obtain distance by multiplying the sensor reading by INVERSE_CM_PER_READING, adding INVERSE_CM_AT_ZERO, and taking the reciprocal of the resulting quantity. To prevent noise and ignore points from ranges outside of our calibration curve, we ignore points with a sensor reading of greater than or equal to 420 and less than or equal to 150 which correspond to 28.5cm and 92.8cm respectively.

Scanning Setup:

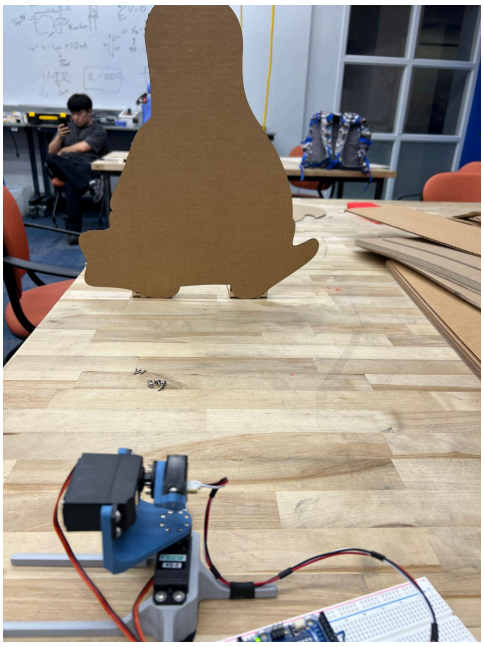

Figure 4: Image of our scanner, Arduino, and the Linux penguin (the object we are scanning).

Single Servo Scans:

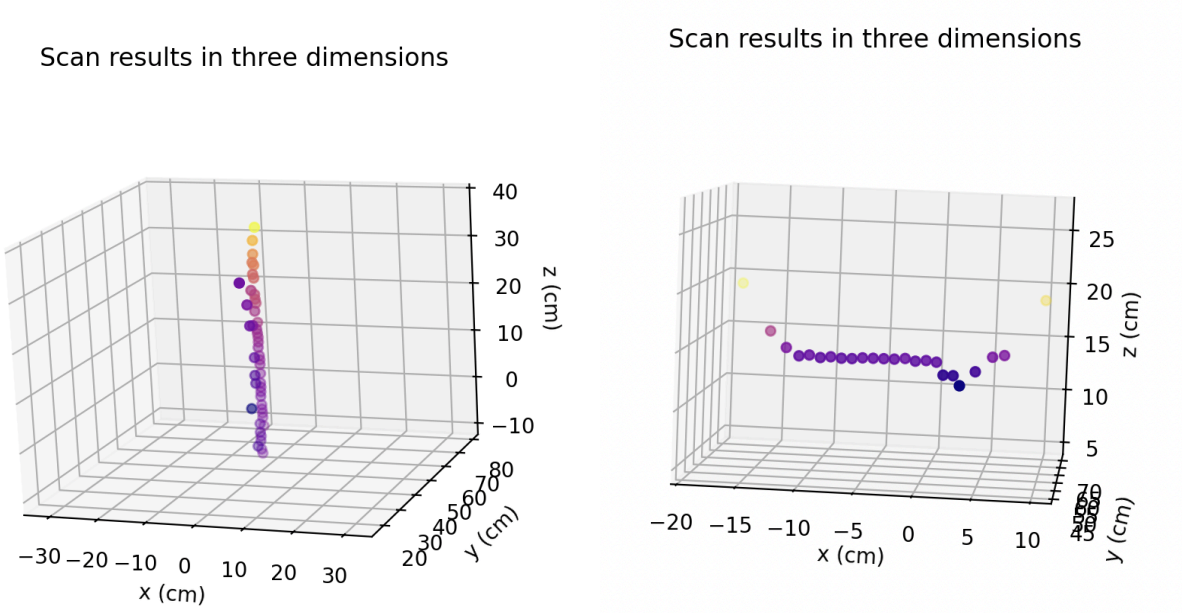

Figure 5: Resulting scan data from sweeping across only the pitch (tilt) servo (left), and only the yaw (pan) servo (right).

We implemented control to allow each of the servos to sweep over their range while holding the other servo constant, ensuring the sensor could be independently rotated about either axis. The above plots are plotted in three dimensions, with additional information (distance from the origin) described through the color of the data points.

Total Scan (Both Servos):

For our identifiable figure outline to scan, we chose a cutout of the shape of Tux, the Linux mascot. We positioned it roughly 51cm away from the sensor, well within the tested range of our sensor and calibration function, and swept both servos across a range of angles in 1 degree increments, recording the distance readings from the sensor. Note that our setup for the two servo scan is identical to the setup for the one servo scan — we disabled one servo in the code, but the physical mount, electrical connections, and physical setup of our scannable object relative to the scanner was identical, depicted in Figure 4.

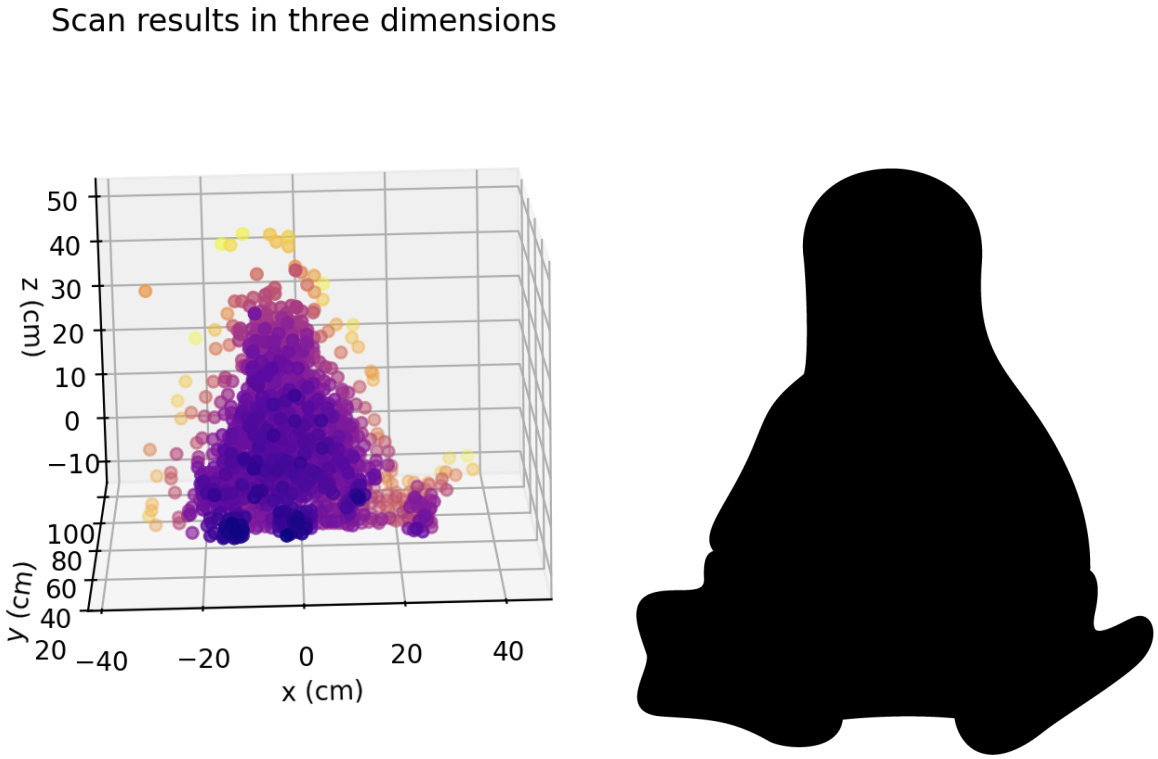

Figure 6: (Left) The recorded shape of the scanned cardboard cutout mascot, displayed as a scatterplot in three dimensions with the color of points corresponding to their distance from the origin. (Right) Simplified Linux penguin mascot which we used as our object to scan.

The shape of the mascot cutout was recognizable in the plotted scan of the data, though the sensor did pick up more noise around the edge of the shape, leading to plotted points past the edge of the shape appearing farther than they were in reality.

To produce the X, Y, and Z coordinates of the points, we implemented a function converting an array of points in spherical coordinates (polar angle, azimuthal angle, and radius) into Cartesian coordinates (X, Y, and Z.) Unlike many conventional spherical coordinate systems, our polar angle, pitch, was measured relative to the horizontal — i.e., a pitch of 0 roughly corresponded to the distance sensor pointing straight horizontally. Given this, the Z coordinate, vertical, is equal to the radius multiplied by the sine of the pitch angle, whereas X and Y were proportional to the cosine of the pitch angle. Given that a yaw of 0 corresponded to the sensor pointing in the X direction, X was therefore also proportional to the cosine of the yaw, whereas Y was proportional to the sine of the yaw. Thus, letting phi denote our yaw and theta denote our pitch relative to the horizontal, our coordinate conversions were:

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

We also converted pitch and yaw into degrees when doing this conversion, before passing them through the trigonometric functions. One issue we initially ran into was not accounting for the zero pitch of our servo being different from the pitch we assumed in our calculations, which was horizontal. Our servo pitch was 90 degrees offset, requiring us to decrease the pitches transferred over serial by 90 degrees in order to accurately represent our shape.

Code Function:

```
def init_connection():
    """
    Initialize serial connection with a port specified by the first argument
    passed when calling the python file.
    """
    port = "/dev/cu.usbmodem1051DB37DE2C2"
    if len(sys.argv) > 1:
        port = sys.argv[1]
    cxn = Serial(port, baudrate=9600)
    cxn.write([int(1)])
    return cxn
```

Figure 7: Code depicting establishment of serial connection

Our Python script initiates a serial connection on a port specified by the first command line argument passed to the Python script, defaulting to the port that we found the sensor automatically connected to on the laptop we tested and ran the script on. Additionally, if a 0 is passed as a second command line argument when calling the script, it instructs the code to spoof data from the sensor and plot a sphere rather than actually call the scan, which was useful in testing our coordinate transformations without the sensor connected.

```
if __name__ == "__main__":
    if len(sys.argv) > 2 and sys.argv[2] == "0":
        print("spoofing data")
        spoof_data()
    else:
        main()
```

Figure 8: Code depicting option to spoof sensor data.

If the option to spoof data is not used, the script prompts the user for the desired mode of functionality of the script: either running just the pitch servo, just the yaw servo, both servos at once, sampling a single data point, or loading previously scanned data from a file without using the servos again.

```
# Prompt user to begin scanning
user_in = input("Run script to gather + plot data? y/n\n")
if user_in.strip().lower() in {"y", "yes"}:
    while user_in not in {1, 2, 3, 4, 5}:
        user_in = int(
            input(
                "Enter mode: 1 (Yaw servo only), 2 (Pitch servo only), 3 "\
                "(Both servos), 4 (Single data point), 5 (Load data from "\
                "recorded_scan.csv): "
            )
        )
```

Figure 9: Code depicting input to specify desired scanner behavior.

For all options but the one loading previous data, the integer representing each command is passed over serial to the Arduino. These values are read in by the Arduino's serial connection and processed by a switch statement to properly tune the bounds of each servo and initiate the scan. The implementation of the scan itself simply consists of a nested for loop, where the outer loop controls the yaw and the inner loop controls the pitch. After each servo movement, the Arduino waits for a specified number of milliseconds, ensuring the servo is able to fully complete its movement before scanning. Sensor readings lower than a specified threshold are thrown out, corresponding to empty space, and if the sensor cannot output a valid reading in a given number of attempts, that data point is skipped. Sensor readings are sent over the serial connection accompanied by corresponding angles of the pitch and yaw servos. When the scan has completed, the Arduino writes the value "-1" to the serial connection, indicating it is finished.

Following this, the values recorded by the Python script are processed as described above, mapped from spherical to Cartesian coordinates, and plotted.

Circuit Diagram:

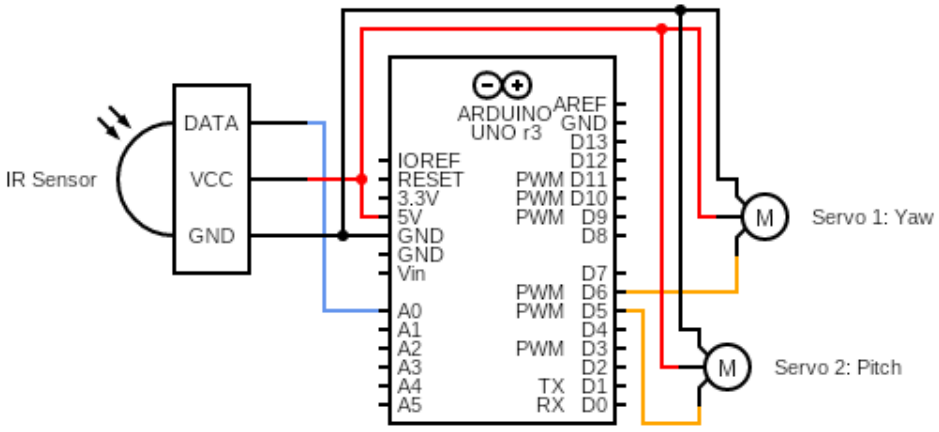

Figure 10: Circuit diagram for 3D scanner.

The data sheet for the IR sensor asks for 5v as VCC, and the servo motors work between 4.8v and 6v, so the same 5v header was used to power both. DIO 5 and 6 were chosen because of their PWM capability and convenient placement. Analog 0 was chosen because it allowed for a shorter wire between it and the breadboard, and, as an analog pin, can properly output the analog voltage reading.

Mechanical design:

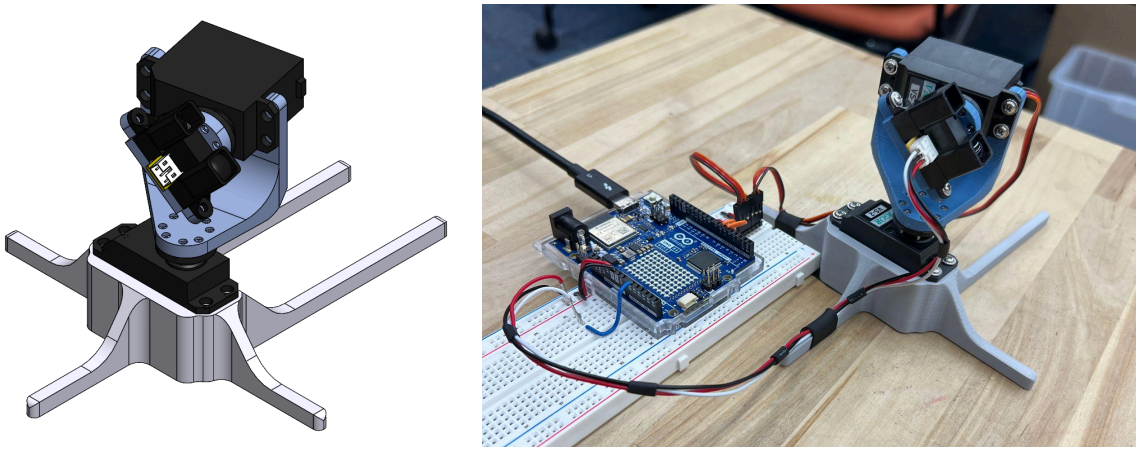

Figure 11: Main assembly in CAD with components color matched to their real world counterparts (left). Image of completed scanner (right).

The main constraint we were working around for the mechanical design of the scanner was ensuring that the distance sensor was centered on the rotational axis of both of the servo

motors. This allows us to measure distance irrespective of the servos' angles, allowing us to simplify our code. Within that constraint, we focussed on trying to use as little material as possible. Using less material is beneficial for if we were to mass manufacture our scanner because it reduces weight, waste, and manufacturing time. In order to do this, we mounted the sensor perpendicular to the pitch servo so that the offset of the pitch servo with respect to the yaw servo could be minimized. This ensured that the pitch servo mount could be made as small as possible. We also used long but thin legs on the scanner base, allowing us to use less material and letting us heat shrink the sensor and servo wires to them for better wire management throughout the full range of motion of the scanner. We also exclusively made use of self tapping screws (shown in figure 11 right, but not modeled in figure 11 left) for the assembly of the scanner. This made assembly easier by eliminating accessibility issues caused by nuts buried inside of printed parts, and improved the theoretical longevity of our sensor's design by making all the bolted joints more vibration resistant.

Reflection:

The majority of the project went well, and we were satisfied with the performance of our sensor in scanning our chosen symbol, the Linux penguin. One strategy we found useful in developing our scanner was testing it without any servo movement, passing false servo positions to the visualization code as if it were actually moving, and graphing the result. When this depicted a sphere (constant radius from origin) we knew our coordinate conversion was correct, and when we didn't see this, we knew our conversion needed to be fixed.

Were we to do this project again, we would have probably chosen a more recognizable shape, or at least integrated cutouts into the one we used. Though the outline of it is somewhat recognizable, it still feels akin to a somewhat lumpy triangle, making evaluating our reconstruction of it more difficult. We would also have liked to integrate the calibration and curve-fitting into our code, rather than having to do it in a separate spreadsheet and manually pass the values into the code. Formally measuring the physical orientation of the servos relative to their internally set angle would also help us get the plotting even better — we knew that zero degrees of pitch was within a small number of degrees of straight down, but specifically measuring this would have improved the accuracy of our visualization. Additionally, it would have been nice had we added functionality to tune the angle bounds and speed of the scan from

the Python script, rather than hard-coding those values in the Arduino-side code and relying on Python only to send the correct start command.

Project Repository:

https://github.com/benricket/pitch-yaw-scanner