Mini Project 2: 3D Scanner

By: Michael Ku and Liam Carlin

Introduction:

For this mini project, we were challenged to create a 3d scanner using an IR sensor and two servos to create a Pan-Tilt Mechanism. All of this was taken together with an Arduino Uno to help take in all the information and passed to a MATLAB script to create a 2d representation of the scanned image. Throughout all this, all of our design choices and code will have a reason to back up why they were made.

Circuit development:

In general, the circuit development for this project was straightforward. According to the documentation for the servos and IR sensors, these are complete modules that are already designed to run directly from a power source at about 5 V. This worked well for us because the Arduino provides a 5 V supply pin that can power all of these components. There were initial concerns about whether the Arduino could supply enough current for all the devices, but in practice, we did not encounter any issues. With this, we could just do some simple connections to power our system. Below is our circuit diagram:

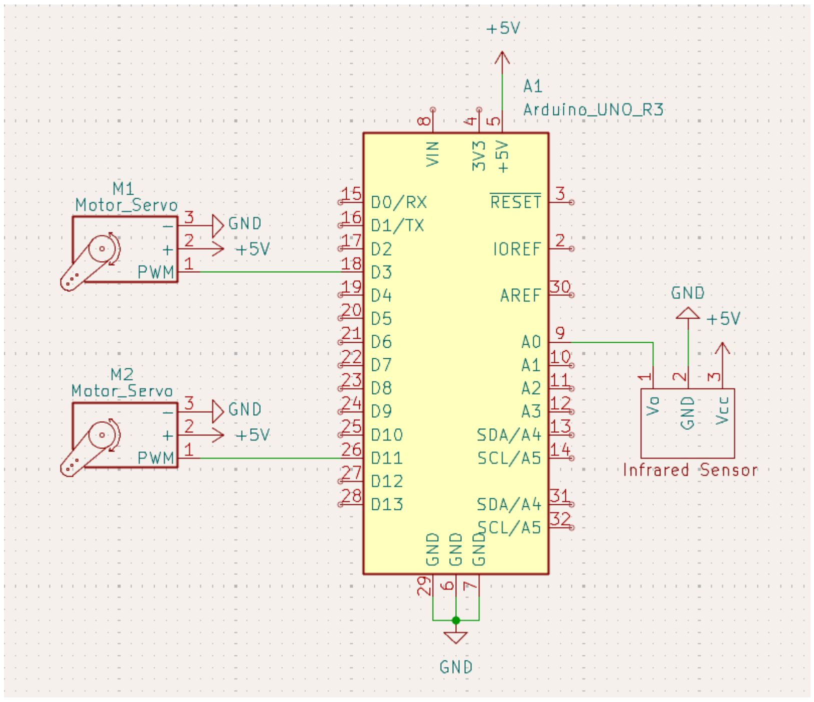


Fig 1: Image of the circut that we used to operate the scanner.

Calibrating the Sensor:

After our circuit is assembled, it's time to test our sensor to get some values and figure out how the analog voltage returned by the circuit can be turned into distances, which can be used to create our image. To do this, we must calibrate our sensor and create a calibration curve. Below is an image of how this experiment was completed.

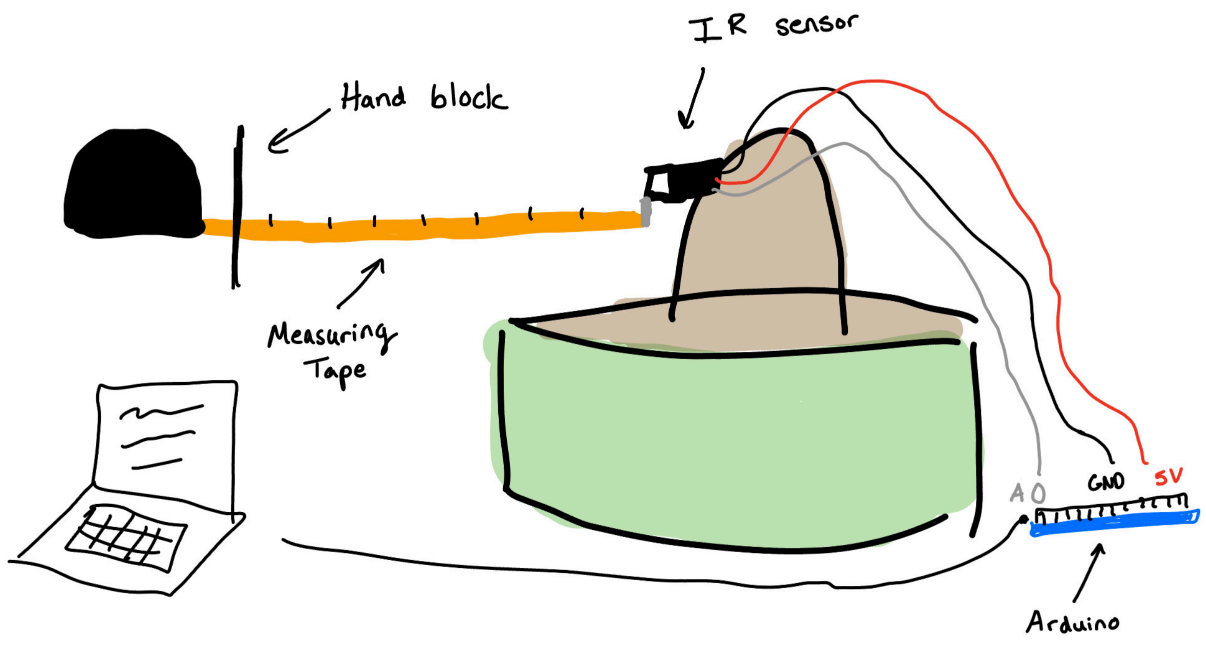
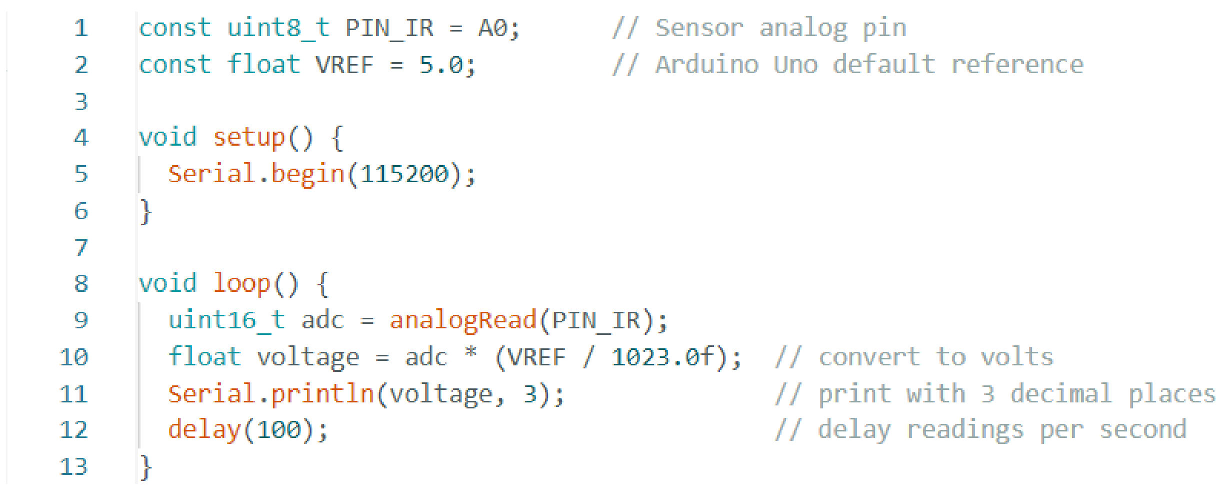


Fig 2: Sketch of the full scanning setup.

As we can see, to calibrate our sensor, we took a bunch of distance measurements and their associated voltage reading from the Arduino, and used that to create a curve and equation that we used to output distance values. Below is the Arduino code that we used to output voltage values into a serial monitor:



```
const uint8_t PIN_IR = A0;       // Sensor analog pin
const float VREF = 5.0;          // Arduino Uno default reference

void setup() {
  Serial.begin(115200);
}

void loop() {
  uint16_t adc = analogRead(PIN_IR);
  float voltage = adc * (VREF / 1023.0f);  // convert to volts
  Serial.println(voltage, 3);              // print with 3 decimal places
  delay(100);                              // delay readings per second
}
```

Fig 3: Calibration code to find the correct voltage.

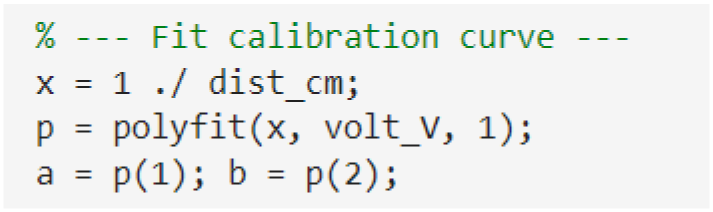
It is a very simple code where it sets up which pin the analog reading will come out of, starts up the serial monitor, which is where we will be reading our data, and then reads the data through analogRead() and converts that into an integer value from 0-1023. From that value, we have to divide the value by 1023 and multiply by 5 to get volts because our Arduino takes in 5V, and to get a true proportion, we have to do those two operations. Now that we have that equation, the output will be in volts and will print out on the monitor to see. With this complete, it was time to take distances and voltage readings and turn them into a calibration curve. Note that one of the things we took into account was the graph in the documentation for the IR sensor, which shows the voltage vs distance for the sensor. There was a huge peak at 20 cm, and after that, a clear curve formed. Because of this, we wanted to model our sensor on that curve after 20 cm, which is why all of our calibration data is after that point.

Here is a table of our calibration data.

Distance (cm)	Voltage (Volts)
20	2.65
25	2.23
35	1.61
40	1.59
45	1.37

50	1.24
60	0.97
70	0.89

From these data points, we can create a calibration curve that represents the voltage over distance of this sensor. We can do this by plotting the different points on MATLAB and fitting the equation to some constant values that would give us the approximate line of fit for our points and thus our sensor. Here is the code that we used to generate our equation:



```
% --- Fit calibration curve ---
x = 1 ./ dist_cm;
p = polyfit(x, volt_V, 1);
a = p(1); b = p(2);
```

Fig 4: Fitted variables for the calibration curve.

With this calculation, we can create a plot with our gathered points and line of best fit. However, before we do this, we should collect a couple of extra test points to see how accurate our line of best fit is and see if we need to make any changes. Below is the table of test points we collected:

Distance (cm)	Voltage (Volts)
22	2.502
32	1.89
42	1.51
52	1.19
62	.96

Now we can plot everything. Below is the MATLAB plot of all of our information that we have just gathered.

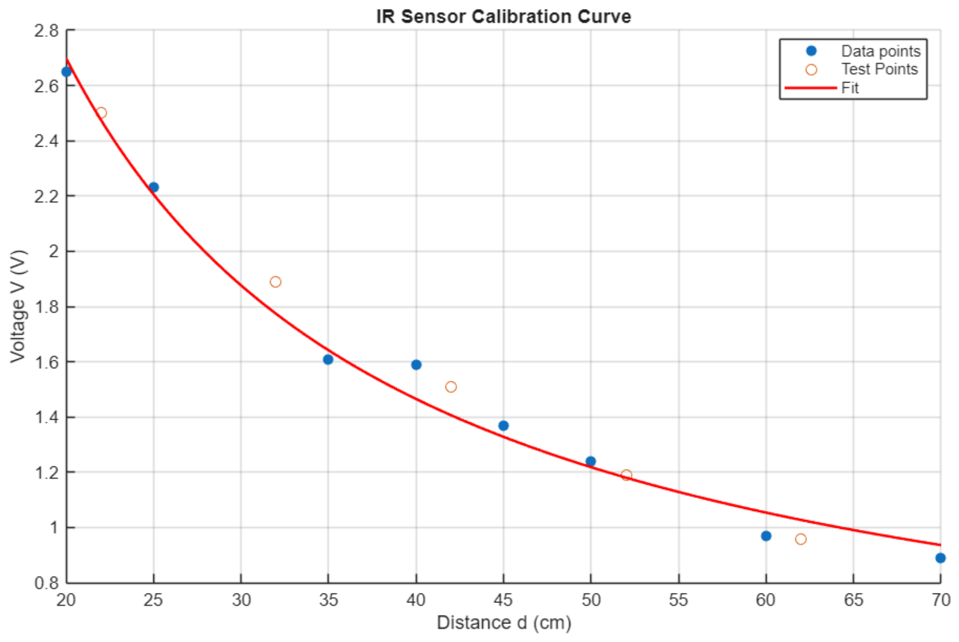


Fig 5: Calibration curve for IR sensor.

As we can see, our line of best fit creates a nice calibration curve for us to use with the test points coming close to or around where they should be. We can compare it to the graph from the IR sensors documentation, which is shown in Figure 6 (bottom of page), which is the actual distance graph. From here, we can also double-check our curve by creating an error table:

test_dist_cm	test_volt_V	d_est	error_cm
22	2.502	21.737	-0.2626
32	1.89	29.759	-2.2405
42	1.51	38.606	-3.3941
52	1.19	51.497	-0.50314
62	0.96	67.759	5.759

Fig 6: Test distances and voltages for the calibration curve.

We have a little bit of error, but nothing too big that should cause us any issues when it comes to scanning. In addition to this, the largest error came from a large distance of 62, which our module will not have to deal with at all. In the end, we were left with this calibration equation:

$$Voltage = 49.3511 * \frac{1}{d} + 0.2317$$

However, since we are inputting voltage and wanting distance out, this is the equation we will be placing into our Arduino code:

$$Distance = \frac{49.3511}{Voltage - 0.2317}$$

Now, with our equation, we can begin to plan how we are going to scan objects.

Servo Apparatus

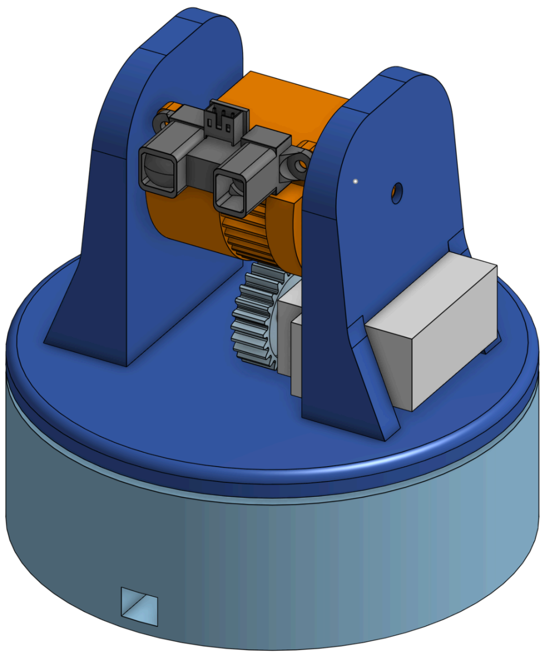


Fig 7: Image of the 3d printed apparatus which holds, pans, and tilts the IR sensor.

We mounted the two servos together using a combination of 3D-printed parts and bolts. The guiding principle behind our design was to keep the center of gravity as low as possible to minimize sway and improve stability during scanning. To achieve this, we placed one servo in the base to handle panning, while the second servo was mounted on the underside of the panning plate to provide tilt. Instead of directly mounting the tilt servo, we used a rack-and-pinion mechanism, where a gear on the servo horn meshed with teeth on the tilt plate. This allowed for smooth tilting of the sensor while keeping the system compact and stable.

The model was fabricated using an FDM 3D printer and consists of five parts that fit together to form the apparatus. The base houses the pan servo, while the top plate secures one of the towers. Tower 2 holds the tilt servo; this tower is designed to be separable so the servo can slide in and then be bolted to the top plate. The servo horn attaches to the circular gear, and finally, the rack is mounted on a semicircle where the IR sensor is positioned. All parts are fastened together with screws that thread directly into the plastic components.

Object Selection:

Below is the object we selected to scan. This will come in handy as a reference for this next section.

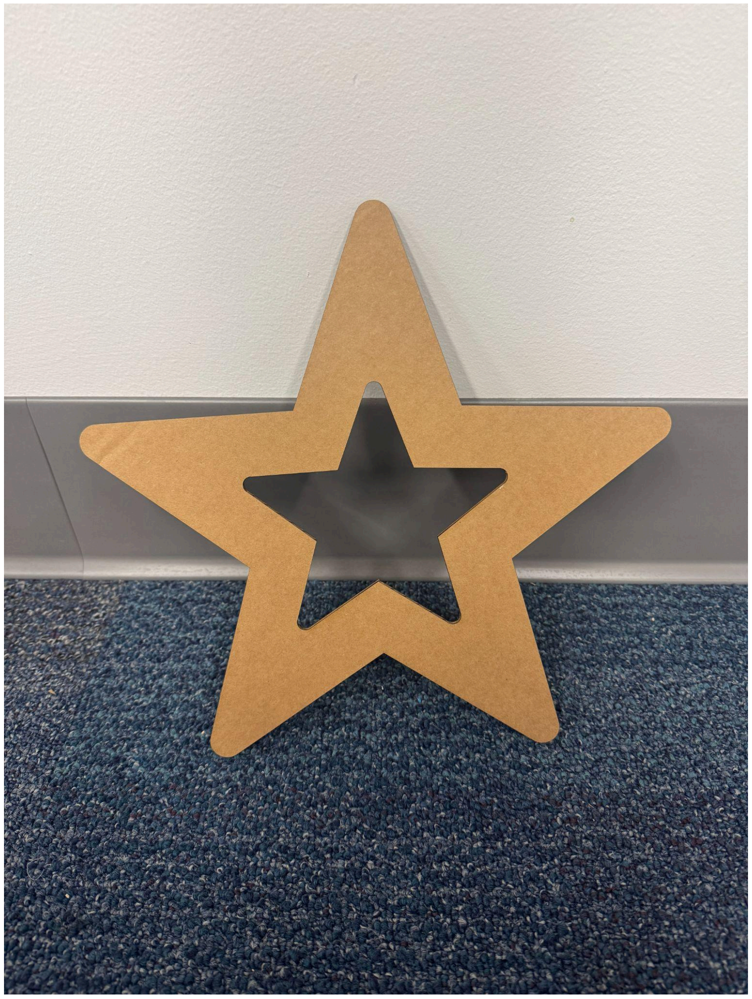


Fig 8: Image of the scanned object.

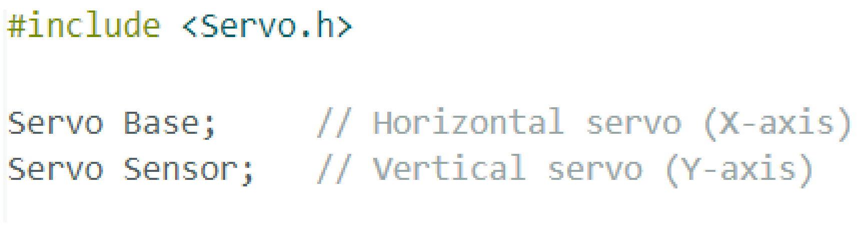
Scanning and modeling:

Now that we have our mechanical portion figured out, we can actually do some scanning and hopefully replicate an object. When looking at how we wanted to design our code, we wanted to break it down into a couple of main ideas:

1. We will have the vertical servo go from up to down to scan the vertical image of the object
2. We will have the IR sensor capture data throughout that vertical image and create a new line when done
3. Once a vertical scan is complete, we will rotate the base servo slightly to move to the next horizontal point and repeat the vertical scan.

With these four steps, we should be able to have a 2d image representation with distances of an object we are trying to scan.

Let's begin with step one. To make a servo move, it is fairly simple. All you have to do on the Arduino IDE is to include the servo.h library, initialize the servo, and then write positions. The photo below shows the initialization for both servos for this project.



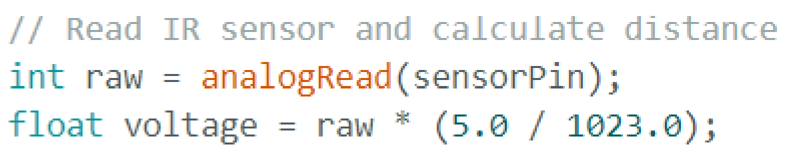
```
#include <Servo.h>

Servo Base;     // Horizontal servo (X-axis)
Servo Sensor;   // Vertical servo (Y-axis)
```

Fig 9: Image of the 3d printed apparatus which holds, pans, and tilts the IR sensor.

From here we can now use the in-built function .attach() to assign a digital pin to the servo and .write() to assign an angle to the servo. With this, we have the fundamentals to start programming the servos with a loop and aligning that with the IR sensor.

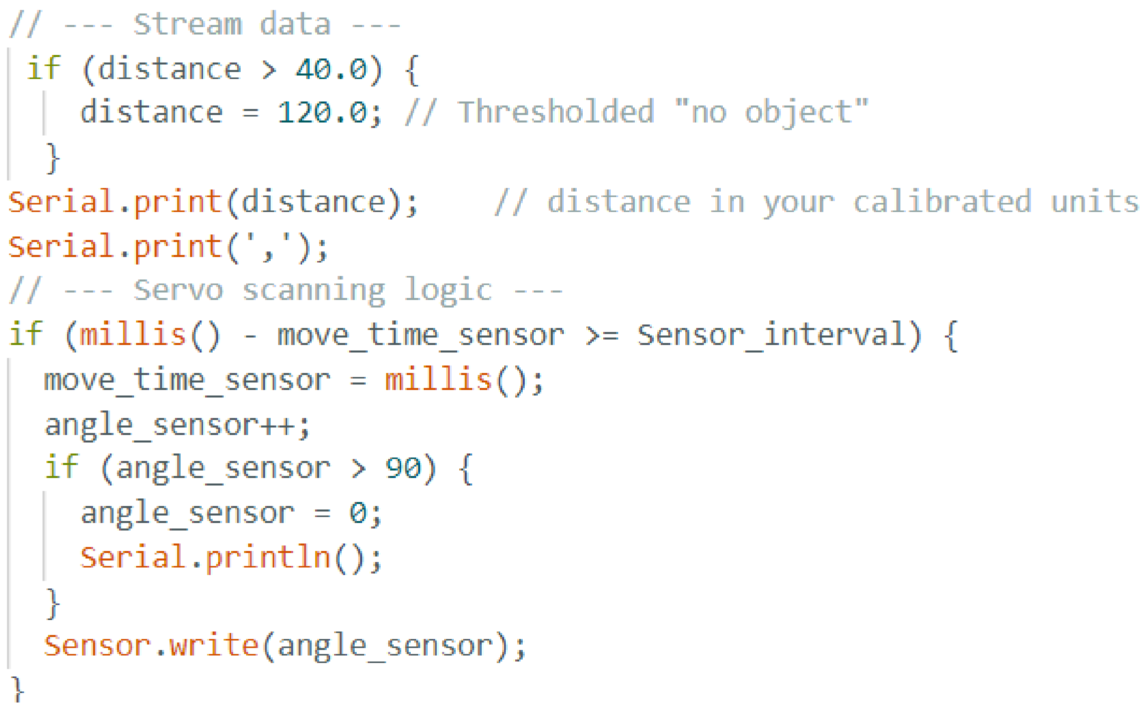
Let's move on to the IR sensor now. To receive data from the IR sensor, we can use analogRead() to read the analog values from a certain pin. Since we are using A0 (from the circuit diagram) we can use analogRead(A0) to get the voltage values using the same multiplication and division explained in the calibration (take input and multiply by 5/1023). With this voltage, we can take that reading and plug it into our calibration equation to get distance. It will look something like this:



```
// Read IR sensor and calculate distance
int raw = analogRead(sensorPin);
float voltage = raw * (5.0 / 1023.0);
```

Fig 10: Reading and calculating IR distance

Now, with all of this combined, we can make our first vertical scan of our object. To limit this to only one servo, we will only define the sensor servo and create a simple code that will start the vertical at 0 degrees, let it print distances followed by commas into the serial monitor until the servo has hit 90 degrees, and then will start printing new values. The servo would pan downwards one degree every 150 ms to make sure we get a good collection of values to show a clear result. In addition to this, we wanted to make sure that our scans and hits really stood out. As a result, we added an extra line of code to help us filter values that were far and things that were not meant to be scanned. To do this, we created a threshold value that stated if the IR saw a value that was at least this big (40 cm in our case), we removed the value and labeled it all as 120 cm to make the colors all line up with one another. With this included, this is what the code looked like for our one vertical scan.



```
// --- Stream data ---
 if (distance > 40.0) {
    distance = 120.0; // Thresholded "no object"
  }
Serial.print(distance);    // distance in your calibrated units
Serial.print(',');
// --- Servo scanning logic ---
if (millis() - move_time_sensor >= Sensor_interval) {
  move_time_sensor = millis();
  angle_sensor++;
  if (angle_sensor > 90) {
    angle_sensor = 0;
    Serial.println();
  }
  Sensor.write(angle_sensor);
}
```

Fig 11: Image of the servo scanning and data stream

All of this code takes place in the Arduino loop and keeps on going. However, for our sake, we did not need more than one loop, as when the angle of the sensor gets passed 90 degrees, it begins a new line of data, where we can end it and copy the collected data. From this one line of data, we can use the MATLAB function imagesc(), which takes a matrix of values and creates an image out of it. Since we only took one line of data, it will only be a 1xn matrix; however, we can still put out a result. This was a scan of the middle of the star that sees both the top, the gap, and the bottom:

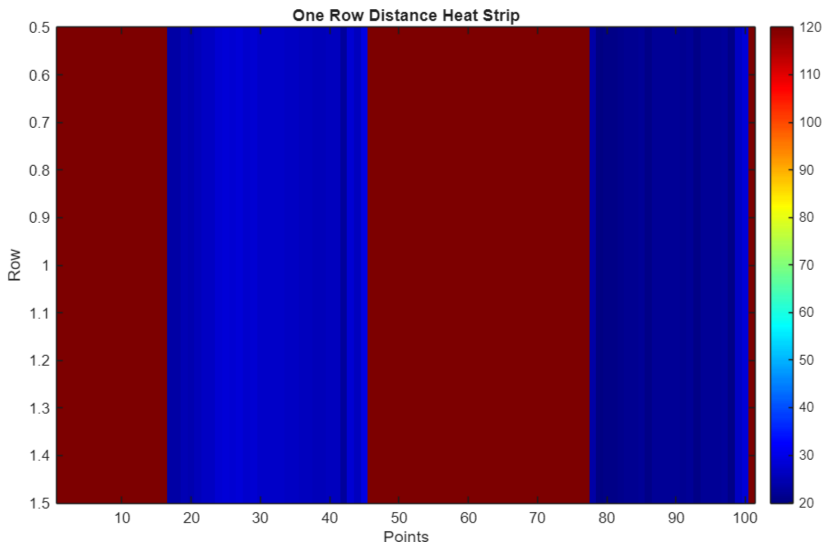


Fig 12: Image single servo scan.

This is good because we can see that our sensor is hitting the solid parts we want it to scan and discarding the values we don’t. However, one row of values doesn't do much for us as it does not represent an object. Now we can work on the full logic to create a full 3d scanner to scan a full object.

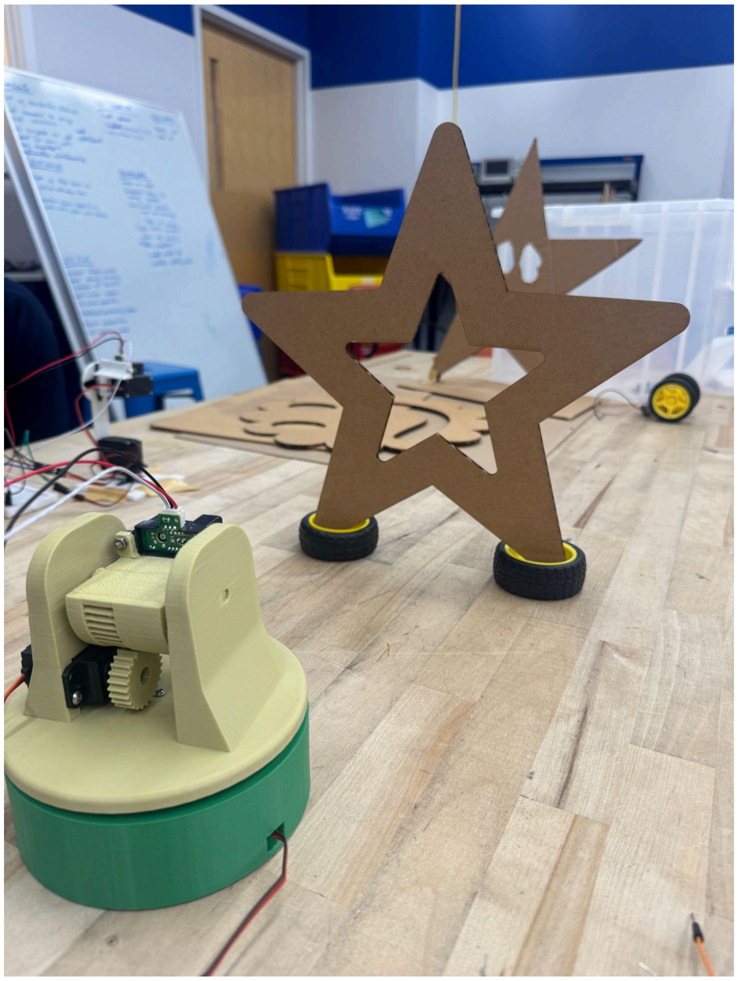
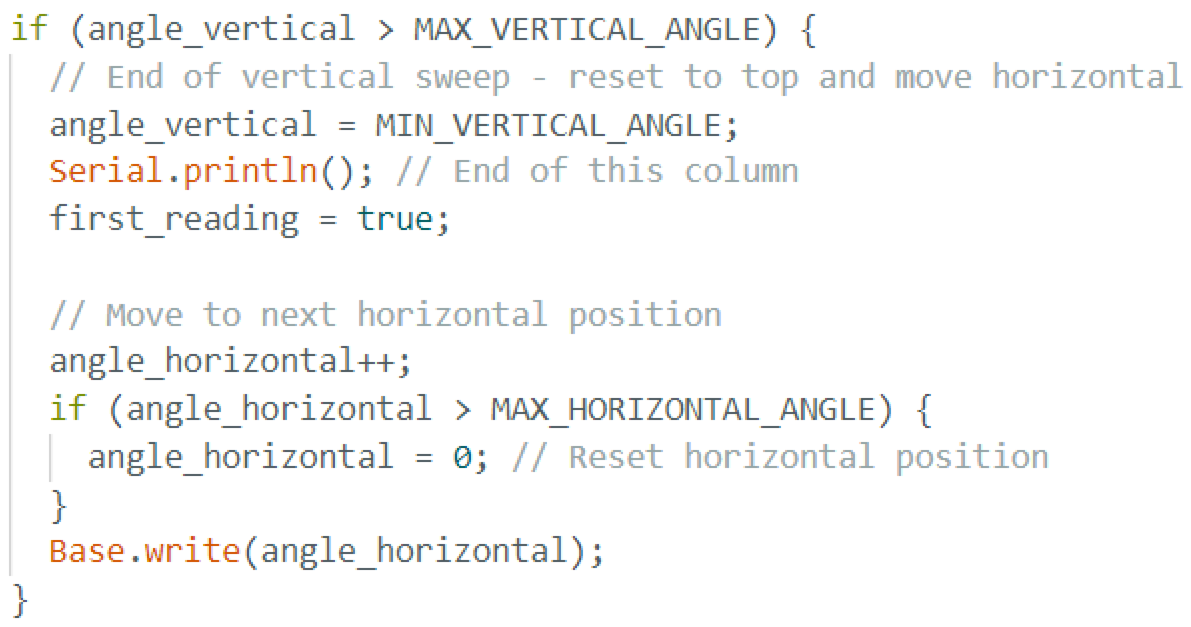


Fig 13: Image of the single servo scan setup.

The only thing we haven't completed is step three, which is to implement the base servo. To do this, all we have to do is make it so that whenever a down pan is complete, we move the top servo to its top starting position and then shift the base one degree to go to the next position. To do this, we create a couple of conditional if logic statements our way. Below is the code that completes that:



```
if (angle_vertical > MAX_VERTICAL_ANGLE) {
  // End of vertical sweep - reset to top and move horizontal
  angle_vertical = MIN_VERTICAL_ANGLE;
  Serial.println(); // End of this column
  first_reading = true;

  // Move to next horizontal position
  angle_horizontal++;
  if (angle_horizontal > MAX_HORIZONTAL_ANGLE) {
    angle_horizontal = 0; // Reset horizontal position
  }
  Base.write(angle_horizontal);
}
```

Fig 14: Code for the IR sweep.

As you can see, if the angle of the servo goes past the maximum angle that we want (90 degrees in this case), we then reset the vertical (panning) servo, print a new line, reset our first reading variable to make sure that our output stays perfectly in CSV format, and then shift the base servo one degree. There is also the condition that if the base servo hits its max angle, we reset, but that is not necessary in our case. With all of our code and logic in place, we can finally capture our scan and plot to see how it did. Below is a photo of how our scan was set up:

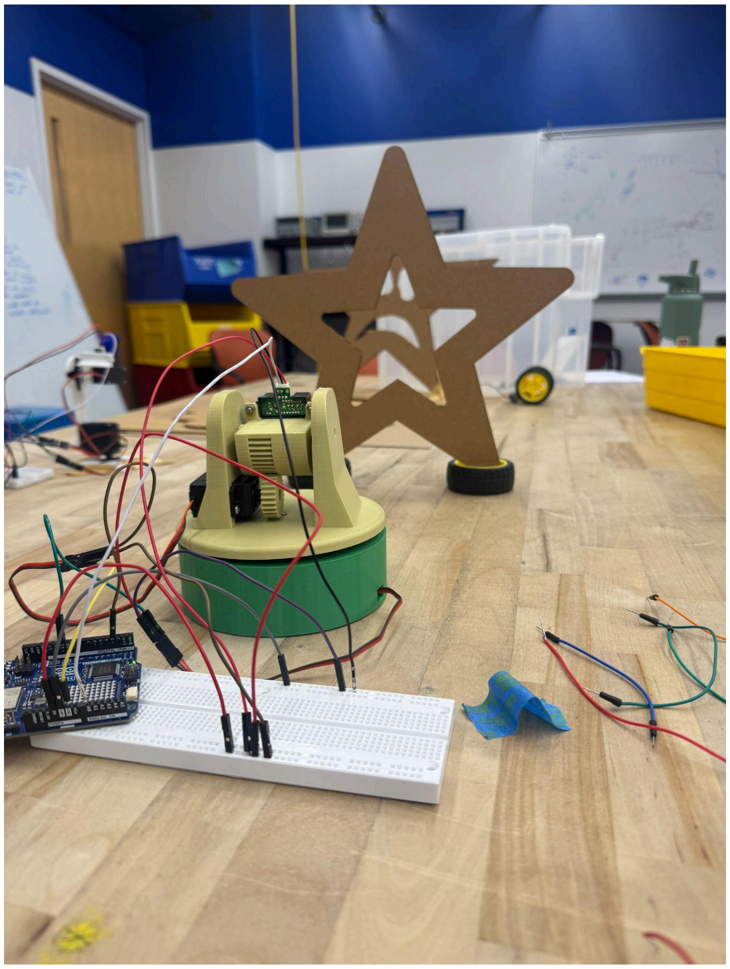


Fig 15: Apparatus set up for double servo sweep.

After collecting our data, we transferred it to MATLAB using the imagesc() function once again, and this was our output:

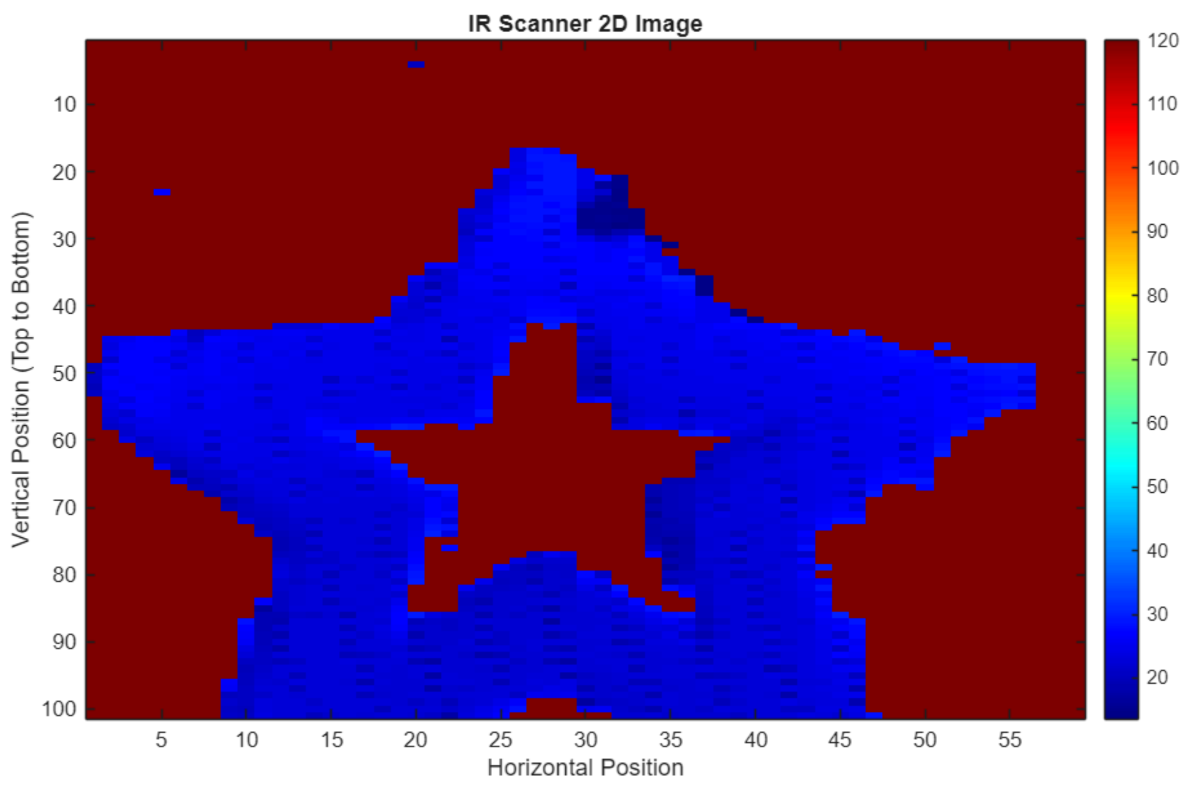


Fig 16: Final IR scan of object.

Taking a look at our scan, we think we did a pretty good job. The general shape of the scan is there, and the gap is clear and star-shaped. The next section will go over our thoughts and reflections.

Reflection

Mechanical Design

The overall result of our mechanical design was a sturdy and wiggle-free setup that could reliably sweep the sensor. One possible improvement would be to integrate bearing balls or similar supports into the plate, which would allow smoother rotation and further reduce the risk of wobble.

Electrical Design

We wired the IR distance sensor and servos to the Arduino using a breadboard and jumper wires. While this was quick to set up, it introduced a recurring issue: the wires often came loose during operation, especially as the gantry moved and tugged on them. This caused interruptions where either the servos stopped rotating or the sensor stopped recording mid-scan. We did not use crimp pins or housings, which could have provided more robust and reliable connections. A future version of this project would benefit from better cable management and more secure connectors.

Software and Programming
With our Arduino code, we encountered some minor issues with serial communication, mainly when trying to copy data from the Arduino serial monitor into MATLAB. While the data streamed correctly during scanning, copying values caused highlighting problems that interrupted logging. Saving directly to CSV would be a more robust approach in the future.

Visualization and Results

We used MATLAB for data visualization, as it provided an efficient workflow for processing the CSV files and generating plots. Our test object was reproduced clearly in the scan output. The scale of the scan matched the original geometry, with consistent step sizes in both the horizontal and vertical directions. While there was some slight angular distortion, it was minimal and did not significantly impact the recognition of the shape. Next time, if we wanted a possibly even clearer image, we would account for the angular distortion and possibly take more data points.

Overall Reflection
Overall, the project demonstrated a successful integration of mechanical, electrical, and software elements. We are particularly proud of two aspects:

- The rack-and-pinion mechanism provided a unique and effective way to achieve tilting while keeping the system compact.
- The simplicity of the scanning code, which was easy to implement and reliable in execution.

There are several areas for improvement that we could implement in the future:

- Increasing the **resolution** of the scan to capture smaller, more detailed objects.
- Improving **wiring reliability** with secure connectors or custom harnesses.
- Exploring **additional mechanical supports** (e.g., bearings) for even smoother and more precise motion.

Appendix:

Vertical Scan Arduino Code:

```
#include <Servo.h>
Servo Sensor;

const int sensorPin = A0;          // analog input for your IR sensor
const uint16_t Sensor_interval = 150; // ms between servo updates

uint32_t move_time_sensor;
int angle_sensor = 0;

void setup() {
  Serial.begin(9600);
  Sensor.attach(3);

  // Move servos to their starting angles
  Sensor.write(angle_sensor);
  move_time_sensor = millis();
}

void loop() {
  // --- Read analog voltage from sensor ---
  int raw = analogRead(sensorPin);              // 0-1023 on a 5 V Arduino
  float voltage = raw * (5.0 / 1023.0);         // convert to volts

  // --- Convert voltage to distance using your calibration ---
  // d = 49.3511 / (V - 0.2317)
  float distance = 49.3511 / (voltage - 0.2317);

  // --- Stream data ---
   if (distance > 40.0) {
     distance = 120.0; // Thresholded "no object"
    }
  Serial.print(distance);    // distance in your calibrated units
  Serial.print(',');
  // --- Servo scanning logic ---
  if (millis() - move_time_sensor >= Sensor_interval) {
    move_time_sensor = millis();
    angle_sensor++;
    if (angle_sensor > 90) {
      angle_sensor = 0;
      Serial.println();
    }
    Sensor.write(angle_sensor);
  }
}
```

Full Scan Arduino Code:

```
#include <Servo.h>

Servo Base;     // Horizontal servo (X-axis)
Servo Sensor;   // Vertical servo (Y-axis)

const int sensorPin = A1;              // IR sensor analog input
const uint16_t Sensor_interval = 150;  // Time between movements (ms)

uint32_t move_time_sensor;

int angle_horizontal = 0;  // Left to right (Base)
int angle_vertical = 0;   // Up to down (Sensor)
bool first_reading = true; // Track first reading in a line

const int MAX_VERTICAL_ANGLE = 100;
const int MIN_VERTICAL_ANGLE = 0;
const int MAX_HORIZONTAL_ANGLE = 120;

void setup() {
  Serial.begin(115200);
  Sensor.attach(3);       // Vertical axis (up/down)
  Base.attach(11);        // Horizontal axis (left/right)

  // Initialize positions
  Base.write(angle_horizontal);
  Sensor.write(angle_vertical);
  delay(750); // Give servos time to reach top position

  move_time_sensor = millis();
  first_reading = true;
}

void loop() {
  if (millis() - move_time_sensor >= Sensor_interval) {
    move_time_sensor = millis();

    // Read IR sensor and calculate distance
    int raw = analogRead(sensorPin);
    float voltage = raw * (5.0 / 1023.0);
    float distance;
    if (voltage <= 0.2317) {
      distance = 120.0; // Out of range
    } else {
      distance = 49.3511 / (voltage - 0.2317); // Your calibration
    }
    if (distance > 30.0) {
      distance = 120.0; // Thresholded "no object"
    }

    // Output distance data in CSV format
    if (!first_reading) {
      Serial.print(",");
    }
    Serial.print(distance, 1); // 1 decimal place
    first_reading = false;

    // --- Vertical Scanning (Top to Bottom Only) ---
    angle_vertical++;

    if (angle_vertical > MAX_VERTICAL_ANGLE) {
      // End of vertical sweep - reset to top and move horizontal
      angle_vertical = MIN_VERTICAL_ANGLE;
      Serial.println(); // End of this column
      first_reading = true;

      // Move to next horizontal position
      angle_horizontal++;
      if (angle_horizontal > MAX_HORIZONTAL_ANGLE) {
        angle_horizontal = 0; // Reset horizontal position
      }
      Base.write(angle_horizontal);
    }

    // Update vertical servo position
    Sensor.write(angle_vertical);
  }
}
```

MATLAB Calibration Code

```
%Calibration Curve
```

```matlab
dist_cm = [20 25 35 40 45 50 60 70]';      % known distances
volt_V  = [2.65 2.23 1.61 1.59 1.37 1.24 0.97 0.89]';  % measured voltages

%Test Values
test_dist_cm = [22 32 42 52 62]';                % known distances
test_volt_V  = [2.502 1.89 1.51 1.19 .96]';       % measured voltages

% --- Fit calibration curve ---
x = 1 ./ dist_cm;
p = polyfit(x, volt_V, 1);
a = p(1); b = p(2);

fprintf('Calibration model: V = %.4f*(1/d) + %.4f\n', a, b);
% --- Plot calibration curve ---
figure;
scatter(dist_cm, volt_V, 'filled'); hold on;
scatter(test_dist_cm, test_volt_V);
d_plot = linspace(min(dist_cm), max(dist_cm), 300);
plot(d_plot, polyval(p, 1./d_plot), 'r', 'LineWidth', 1.5);
xlabel('Distance d (cm)'); ylabel('Voltage V (V)');
title('IR Sensor Calibration Curve');
legend('Data points','Test Points','Fit','Location','best'); grid on;
```

MATLAB Image Generation

```matlab
data = readmatrix('scanner - scan_data.csv')'; %Loads Data Set
figure; %Creates new figure
imagesc(data); %Generates Image
colorbar; %creates side bar

%Formatting
title('IR Scanner 2D Image');
ylabel('Vertical Position (Top to Bottom)');
xlabel('Horizontal Position');
colormap(jet);
```