Mini-Project 2 Write-Up

Ian Eykamp
Sam Kaplan
Section 6

October 1st 2021

1 Overview

For this project, we built a 3D scanning device using a infrared (IR) distance sensor mounted on a pan/tilt mechanism actuated by two servos and controlled by an Arduino. The pan, tilt, and distance data are transmitted to a computer via a handshaking serial connection and visualized in real time using pyqtgraph.

1.1 Hardware

We 3D printed plastic cases to hold the servos. The bottom case holds the panning servo and is attached to a wide cardboard base that can be taped to any horizontal surface. The second case holds the tilting servo and is mounted on the pivot arm of the first servo; the IR sensor is mounted to the pivot arm of the second servo. This allows the sensor to be panned and tilted independently. One key design decision was to position the assembly such that the center of the IR sensor lies directly over the pivot point of the bottom servo and in line with that of the top servo. This eliminates any translational movement (in the xy plane) of the sensor during panning and tilting and simplifies the pan and tilt to spherical coordinates.

1.2 Circuit

The circuit we used is shown in Figure 1. The servos each have one wire connected to a digital PWM pin on the Arduino, and the IR sensor output is connected to an analog pin. The other wires supply 5V and GND. Additionally, we added two potentiometers, which we used for debugging the pan/tilt mechanism. We found most of the difficult work on this project to be more software related.

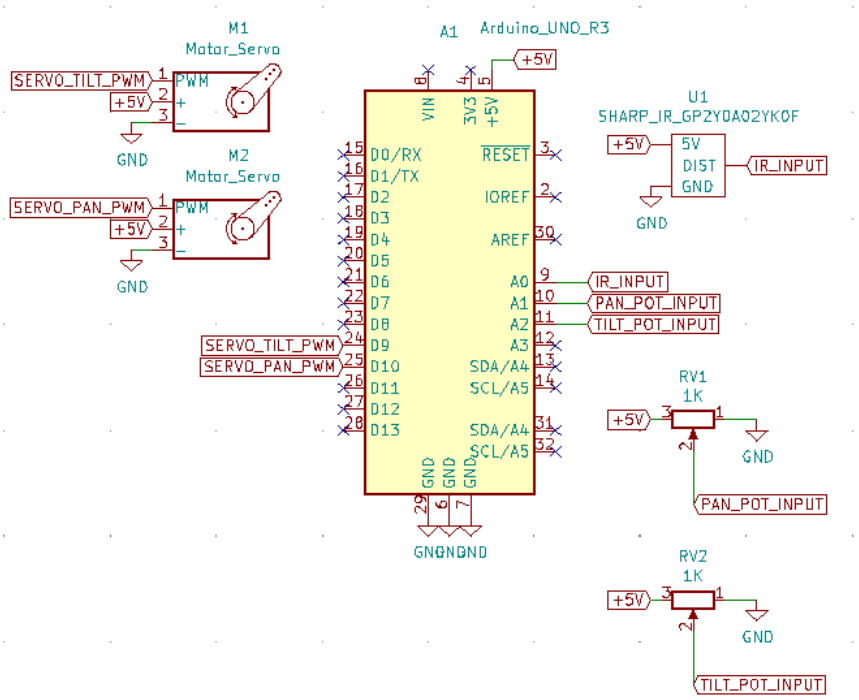

Figure 1: Electrical Schematic

1.3 Calibration

We tried two different calibrations methods. First, taped a large sheet of dark black plastic against the wall. We recorded the sensor's average analog reading at 15 different distances from the box, between 5 and 75 inches (12.7-190.5 cm). This encompasses the entire useful range of the sensor of 20-150 cm according to the datasheet. We thought that a black surface would give optimal readings as it would absorb more of the infrared beam than other surfaces. Our assumption was correct, but it actually produced poorer readings because less IR light was reflected from the surface.

After re-thinking, we then calibrated on a white surface (paper taped to a flat garbage can), using the same 15 distances as before. However, after comparing the accuracy of the different methods (using the same benchmarked distances), we found that the equations we derived using the black surface performed very well when used with white surfaces. While this is a completely coincidental discovery, our rationalization is that we calibrated the system for the worst possible expected conditions, and then "surprised" it with really good beam reflections.

We recorded our data in excel, which allowed us to try lots of different curve fits. After some trial and error, power regression appeared to most closely match the data, especially for distances greater than 50cm. This corresponds closely to the curves given on the datasheet, which made us feel very warm and fuzzy inside. The calibration equation we obtained was

$$\text{Distance} = 15954 \cdot \text{IR}^{-1.087} \tag{1}$$

where IR is the infrared sensor value, which ranges from 0 to 1023. The calibration data and regression curve are shown in Figure 2.

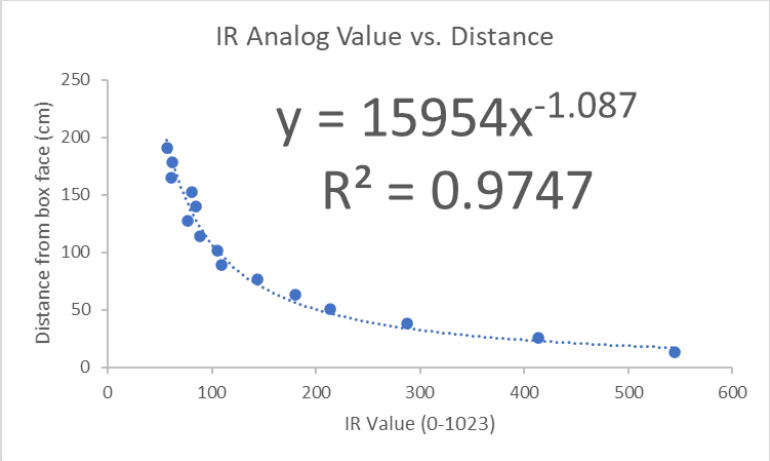

Figure 2: Calibration data set for the black wall face and best-fit power curve

We validated our calibration curve based on our data set for the white box face. The calculated distances match very closely with the actual measured values, as shown in Figure 2. As stated earlier, the data for the white box face follow a much smoother trend, but the calibration equation obtained from the black wall face still works very well, especially at small and large IR values.

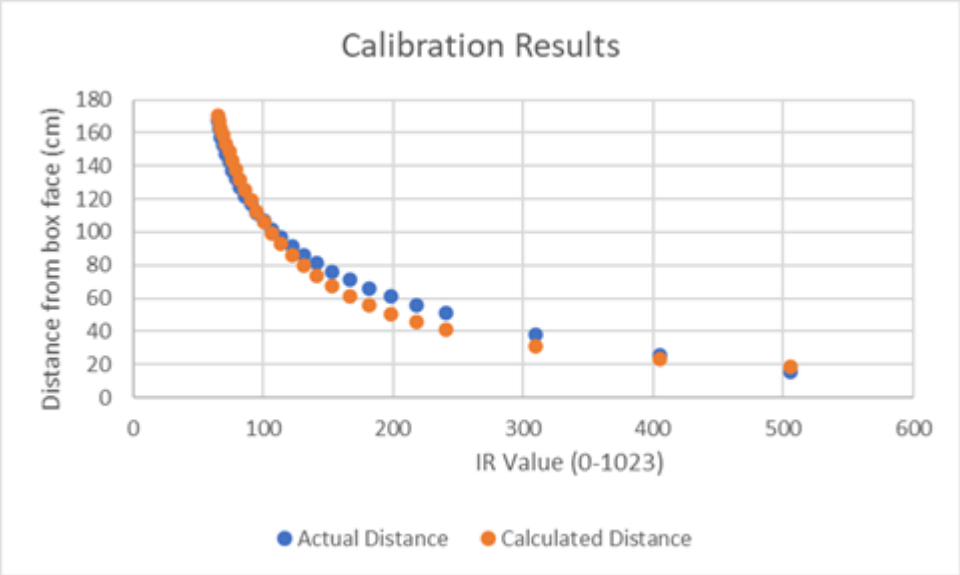

Figure 3: Measured distance vs. calculated distance based on the calibration equation

We later observed that the sensor works best with little ambient light, especially sunlight and overhead lights. When we blocked out these light sources, by putting the apparatus under a desk, we significantly reduced outliers in the analog readings.

1.4 2D Test Scan

See our full setup in Figure 4:

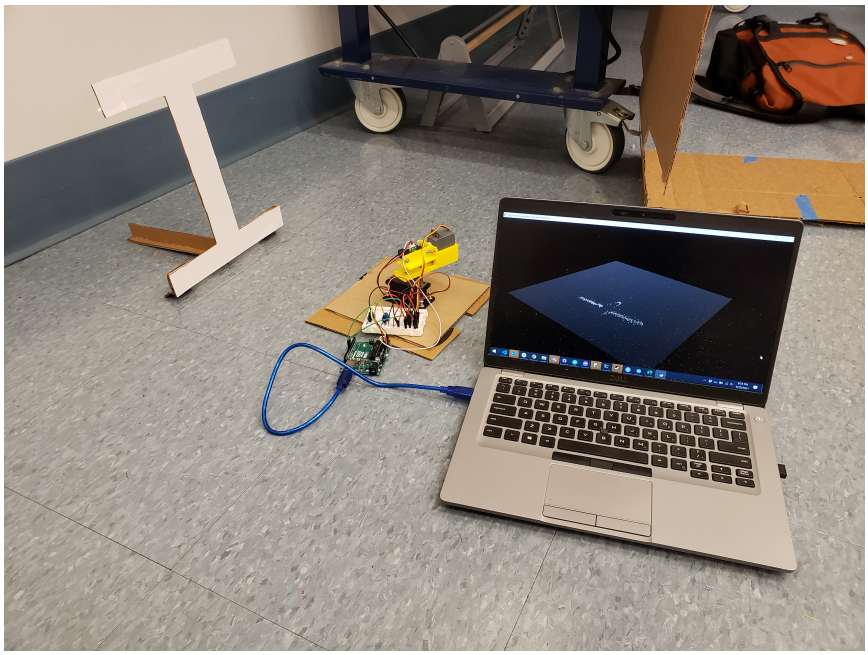

Figure 4: Setup for linear letter scan

We used our scanner to scan a 25x30cm letter 'I' cut out of cardboard, with a white paper front. We started with a linear scan across the midsection of the I, with the setup shown in Figure 4, and with the results in Figure 5. The scan shows the wall in the background and the midsection of the I about 50cm inches in front. The figure shows several passes, and the variation in the sensor reading between passes can be observed, which is especially apparent for the wall.

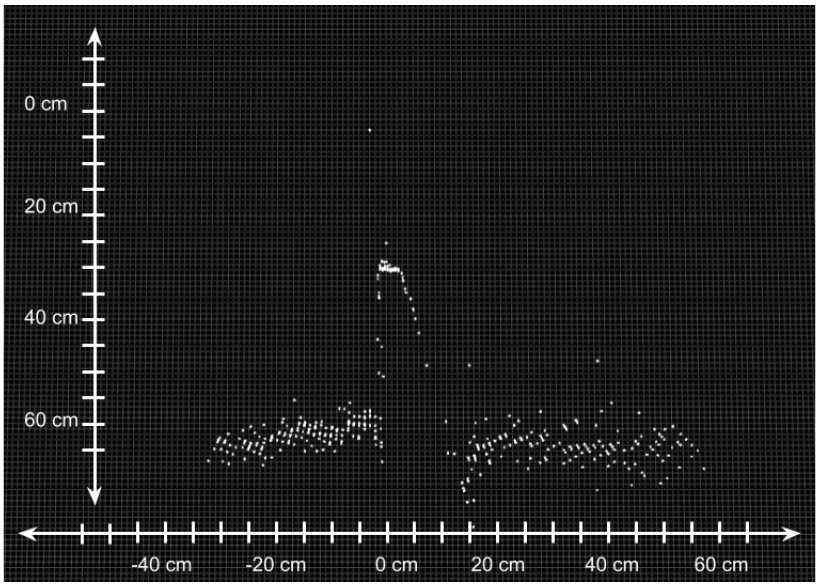

Figure 5: Results of linear letter scan. The horizontal and vertical axes show intervals of 5cm. The peak is the midsection of the letter I, which is about 5cm wide. The wall is shown along the bottom of the graph, and the origin is at the top.

The peak of the scan represents the middle part of our letter I, which makes sense. Sanity checked!

1.5 3D Plot

Next, we went for the full 3D plot, encountering two problems on the way. First, our python live-plotter was crashing about 30 seconds into a scan. We quickly realized that this was due to a buffer overflow, and resolved the error by slowing our scanning speed (on the embedded side of the system) to reduce the rate of Serial transmission.

Next, our trigonometry seemed to be wrong. We quickly realized that the servo angles were not zeroed. We used the debugging potentiometers to adjust the servos until the tilt servo was level and the pan servo was at 90 degrees. We recorded the angles that the servos were reading and added offsets in the code. After those two corrections, we got pretty nice looking 3D plots of our letter!

Out equations for converting from polar coordinates (servo angles and distance) to the Cartesian plane is as follows:

$$x = IR_{corrected} \cdot cos(\theta) \tag{2}$$

$$y = IR_{corrected} \cdot sin(\theta) \tag{3}$$

$$z = IR_{corrected} \cdot sin(\phi) \tag{4}$$

where θ is the angle of the pan servo and ϕ is the angle of the tilt servo.

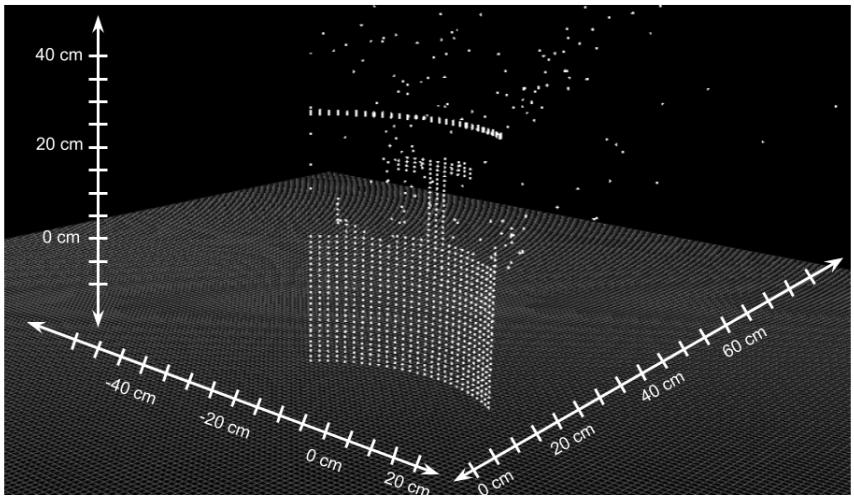

Figure 6: Results of 3D scan. The lower half of the letter in the scan is confused with the floor surface due to the large tilt range and over-correction in the plotting algorithm. The axes show intervals of 5cm.

There are three small issues in our final rendering. Despite transforming from spherical to Cartesian coordinates, our plot still has significant curvature, such that it appears to be projected onto a cylinder. We believe this is a by-product of the second issue, which is our aggressive filtering of the IR sensor data. After applying our calibration factor, we implemented a minimum and maximum threshold within which to plot the distance data. This was intended to improve the distinction between the letter face and the open space behind it, but it had some unintended consequences. For example, it leads to the appearance that all of the points lie in an (albeit curved) plane. Our tilt servo also traversed a large range, meaning it picked up a portion of the floor. Because of our thresholding, it appears in the graph that the bottom of the I is very wide, when in reality this just represents that the distance from the sensor to the floor is similar to the distance from the sensor to the face of the I. Finally, the horizontal line at the top of the scan is an artifact of starting each new tilting scan too quickly, and attempting to read sensor values for a high tilt angle before the servo had reached the specified angle.

1.6 Code

The Arduino code uses the Servo library to control the pan/tilt mechanism. It loops through the functional range of each servo using nested for loops, processes the IR sensor value at each position, and prints a string to the serial port containing the pan and tilt angles and the sensor value. The string is plain text with a comma between each value, which is a format that is both easy to debug and easy to manipulate on the Python side using the split function. Once the Arduino transmits a data point, it waits until it receives a confirmation from the Python script. Python sends the entire string back to the Arduino over the serial port, and the Arduino checks it against the values it just sent; if the values don't match, the Arduino resends the packet. This handshake protocol ensures that, if the communication breaks down (for example, if the visualization freezes temporarily), no data points are lost.

The only processing that is done on the Arduino side is that it reads the IR sensor three times using analogRead, then transmits only the minimum value. This is based on the fact that the analog signal from the sensor contains regular voltage spikes every millisecond, with a width of about 120 μs. We know that the analogRead call takes about 100 μs to complete, so if we perform three analogReads back-to-back, at least one of them will lie outside of a voltage spike. Hence we compute the minimum of the three readings.

The python code is pretty simple: we have functions for our calibration correction and polar-Cartesian conversion. The serial communication loop in held inside an update function, which gets called continuously by pyqtgraph.

1.7 Reflection

Ian: We undertook several small challenges in different parts of our project. We implemented a basic handshaking protocol for transmitting data over Serial between the Arduino and the computer. This made our interface more robust and made it really easy to collect data. Because the scanner wouldn't start until a serial connection was established with the Python script, we could upload the program from the Arduino IDE, then switch to the plotting script and hit run – only then would things start moving. And if we wanted to restart or pause our script, the scanner would pause and then reset when we chose. Needless to say, this made debugging much easier.

Another challenge was implementing the live plotting using pyqt. Sam mostly worked on this, while I handled the scanning and transmission on the Arduino side. The two halves of the project were in constant communication (literally, via Serial), so we had to work closely even when we were asynchronous. I fell sick during the last week of the project, which hampered our ability to fully debug our 3D coordinate graphing; however, we identified areas for improvement, such as limiting the range of the sensor scan and making our sensor filtering more robust, that we think we could have implemented had we had more time.

Sam: I really enjoyed getting into the weeds with the Arduino-Python communication. Understanding how everything was being transmitted at the byte level (and how that is preferable to using strings [lots of memory]), and implmenting the handshake protocol was incredibly cool. Getting to flex my pyqt skills after working with it this summer was also nice - in my opinion nothing is more satisfying than a nice live plot. I also CAD'd the first iteration (which ended up being replaced due to axis alignment issues), but it was still nice to practice that skill and get a clean looking print.

I wish we had gotten a little more time to debug some of the weird artifacts in our plotting, but I am sure they are mostly little tweaks to our math that would just require a lot of debugging. Unfortunately with Ian being sick the last week of the project, we didn't get a chance to put our collective brains together on that one and were unable to fully resolve it, but I think our plot turned out great anyway!

2 Appendix

For your convenience, we have pasted our source code below. You can also see it on GitHub here.

2.1 Python Source Code

```

import serial
from pyqtgraph.Qt import QtCore, QtGui
import pyqtgraph.opengl as gl
import numpy as np
import time
import sys
import math


def correction(data):
    # this function applies the calibration curve for the IR sensor

    # print for debugging
    print(data)

    if data <= 0:
        # if < 0, make very small so things w/exponents don't break
        data = 0.0000001
    if data >= 150:
        #if > 150, make 150 as that is the max range of sensor
        data = 150

    # this is our calibration equation
    return 2906.5 * pow(data, -.899)


def xyx_transfrom(ir, pan, tilt):
```

```
    # this function takes polar coords (dist, pan+tilt angle) and converts to cartesian coords
    x = ir * math.cos(math.radians(pan))
    y = ir * math.sin(math.radians(pan))
    z = ir * math.sin(math.radians(tilt))

    return [x,y,z]

def update():
    # this function is called by pyqtgraph and updates the plot. All comms w/Arduino happen here

    # this need to be global to be used in the plot (which lives outside this function)
    global count
    global pos1
    global serialPort

    # list for Arduino serial data
    serialLine = []

    # this is the big Arduino comm loop
    while True:
        ser_data = serialPort.read(1) # read 1 byte from Arduino
        if (ser_data.decode('utf8') == '\\'): # check for header (sign to begin message)
            break
        print("WAITING")

    while True:
        ser_data = serialPort.read(1) # read 1 byte from Arduino
        ser_char = ser_data.decode('utf8') # convert bytes -> utf8
        if (ser_char == '\n'): # check for message end char
            break
        serialLine.append(ser_char)

    # create string from list in order to use split operator. Returns a list of strings
    serialStrings = "".join(serialLine).split(',')
    serialLine = [] # reset serialLine for the next message

    # must convert strings to nums to actual nums!
    # also applying servo offsets
    ir_raw = int(serialStrings[0])
    ir_corrected = correction(ir_raw)
    pan_angle = int(serialStrings[1]) - 84
    tilt_angle = 90 - int(serialStrings[2]) + 13

    # convert from polar to cartesian
    x, y, z = xyx_transfrom(ir_corrected, pan_angle, tilt_angle)

    # write data back to Arduino for error checking
    for string in serialStrings:
        serialPort.write(string.encode('utf8'))


```

```
    # graphing stuff!
    new = np.array([[x,y,z]]) # creates new 1x3 array
    pos1 = np.append(pos1, new, 0) # append to current scatter
    array (holds all data)
    scttrPlt.setData(pos=pos1, color=color) # set the updated data
    to the scatter plot object



if __name__ == '__main__':
    # creates QT gui app window
    app = QtGui.QApplication([])
    w = gl.GLViewWidget()
    w.opts['distance'] = 20
    w.show()
    w.setWindowTitle('3D IR Scanner')

    # setting grid options
    g = gl.GLGridItem()
    g.translate(0, -10, 0)
    g.setSize(200,200,200)
    w.addItem(g)

    # setting Arduino stuff
    arduinoComPort = "COM7"
    baudRate = 9600
    serialPort = serial.Serial(arduinoComPort, baudRate, timeout=1)

    # creates original scatter data array
    pos1 = np.array([[0,0,0]])

    # sets color to white
    color = (1,1,1,1)
    size = 0.5

    # needs to be global for update function to work
    global scttrPlt
    scttrPlt = gl.GLScatterPlotItem(pos=pos1, size=size, color=
    color, pxMode=False)
    w.addItem(scttrPlt)

    # connects the window to the update function (will crash if
    something in the serial comm hangs)
    t = QtCore.QTimer()
    t.timeout.connect(update)

    t.start(50)

    if (sys.flags.interactive != 1) or not hasattr(QtCore, '
    PYQT_VERSION'):
        QtGui.QApplication.instance().exec_()
```

2.2 Arduino Source Code

```
#include <Servo.h>

#define DEBUG 0 // set this to true to enable serial.println of
```

```
    debugging code segments.

Servo pan_servo;
Servo tilt_servo;

// pan and tilt angles to scan through
const int pan_range[2] = {50, 130};
const int tilt_range[2] = {0, 135};

// rotation in degrees between sensor readings
const int pan_step = 1;
const int tilt_step = 1;

#define IR_PIN A0
#define PACKET_SIZE 3

void setup()
{
    Serial.begin(9600);
    pinMode(LED_BUILTIN, OUTPUT);
}

void loop()
{
    scanAndTransmit();
}

int readIR()
{
    return analogRead(IR_PIN);
}

int filterMin()
{
    // reads three samples and returns the minimum.
    // this is necessary to avoid spikes in the analog signal.
    int N_SAMPLES = 3;
    int output = 1025;
    for (int i = 0; i < N_SAMPLES; i++)
    {
        output = min(output, readIR());
    }
    return output;
}

void scanAndTransmit()
{
    int packet[PACKET_SIZE];
    bool response = false;

    // iterate over the entire scanning range
    for (int pan = pan_range[0]; pan < pan_range[1]; pan +=
    pan_step)
    {
        for (int tilt = tilt_range[0]; tilt < tilt_range[1]; tilt
    += tilt_step)
        {
```

```
            packet[1] = pan;
            packet[2] = tilt;
            pan_servo.write(pan);
            tilt_servo.write(tilt);

            packet[0] = filterMin();

            response = sendPacket(packet); // returns true if the handshake is successful
            digitalWrite(LED_BUILTIN, response);

            // if the response was bad, re-send the packet until true
            while (response == false)
            {
                delay(100);
                response = sendPacket(packet);
            }

            delay(100); // helps ensure that the serial buffer doesn't overflow too quickly
        }
    }
}

void packetHeader() // send at the beginning of a packet
{
    Serial.print('\\');
}

void packetFooter() // send at the end of a packet
{
    Serial.print('\n');
}

bool sendPacket(int packet[PACKET_SIZE])
{
    // prints each value in packet separated by a comma
    // This format is used for parsing on the Python side.
    packetHeader();
    for (int i = 0; i < PACKET_SIZE; i++)
    {
        Serial.print(packet[i]);

        if (i < PACKET_SIZE - 1)
        {
            Serial.print(',');
        }
    }
    packetFooter();
    return checkReceived(packet); // Returns true if the handshake is completed and the values match.
}

bool checkReceived(int sentPacket[PACKET_SIZE])
{
    // waits for a handshake from the Python code and checks if the
```

```
     received packet
    // matches the sent packet
    int receivedPacket[PACKET_SIZE];

    //delay until any packet is received
    while (Serial.available() <= 0)
    {
        if (DEBUG)
        {
            Serial.print("."); // wait for a packet response
            delay(50);
        }
    }
    // check if the received packet matches the sent one
    for (int i = 0; i < PACKET_SIZE; i++)
    {
        if (Serial.available() <= 0)
        {
            // if the Serial buffer ends before we reach the
            // packet size, then the handshake has failed.
            if (DEBUG)
            {
                Serial.println("Message too short");
            }
            return false;
        }
        receivedPacket[i] = Serial.read() - '0'; // the - '0'
    converts from an ascii character to an equivalent integer
    }

    // debugging statements
    if (DEBUG)
    {
        if (!equalArrays(receivedPacket, sentPacket))
        {
            Serial.println("Message didn't match");
        }
        else
        {
            Serial.println("Message matches!");
        }

        Serial.print("receivedPacket = ");
        printArray(receivedPacket);
        Serial.print(", sentPacket = ");
        printArray(sentPacket);
        Serial.println("");
    }

    // after reading the first characters in PACKET_SIZE, ignore
    the rest
    flushBuffer();
    return equalArrays(receivedPacket, sentPacket);
}

void printArray(int array[PACKET_SIZE]) // prints each element an
    array; used for debugging
```

```
{
    for (int i = 0; i < PACKET_SIZE; i++)
    {
        Serial.print(array[i]);
    }
}

bool equalArrays(int array1[PACKET_SIZE], int array2[PACKET_SIZE])
    // checks if each element in two arrays are equal
{
    for (int i = 0; i < PACKET_SIZE; i++)
    {
        if (array1[i] != array2[i])
        {
            return false;
        }
    }
    return true;
}

void flushBuffer() // flush the serial buffer
{
    while (Serial.available() > 0)
    {
        Serial.read();
    }
}
```