Final Report

By Samantha Butler

Introduction

The goal of this project was to implement a self-balancing robot that would move in a way so that it would constantly be tracking human faces. This project was completed in two phases. The first phase was to get the robot to self-balance on its own, the second phase included building a face recognition system and lastly the third phase involved calculating what direction to move the robot and relay that information back to the Arduino nano on the robot.

Phase One

The following diagram in figure 1 can be referenced to capture an overview of how I went about getting the robot to self-balance. It involves a continuous closed loop of guess and checks. The control system should compare the robot's tilt angle to the desired target angle and calculate the error difference. The PID controller then calculates the motor control signal needed to correct the error.

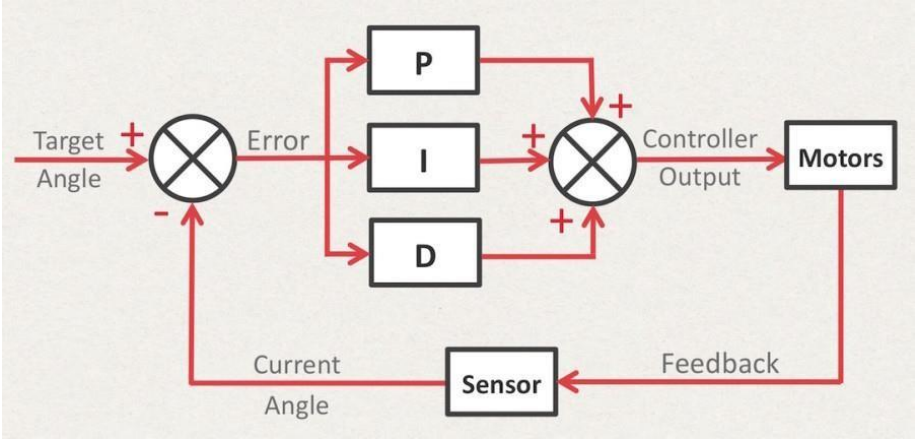

Figure 1. PID self-balance filter

The previous work on my robot involved solely calculations, measurements, and initial setup but nothing fully functioning yet. The steps I took summarizing my findings are as follows:

Step 1. Calibrating IMU

Initially, I calibrated my MPU sensor and tried gathering consistent offset values, both for the gyroscope and the accelerometer. For each iteration of calibrating my MPU sensor, I took 1000 values. My results when running three consecutive calibration iterations were as follows:

```
ready to calibrate
Accel calibration complete
offset values, x: -231, y: 845, z: -798
Gyro calibration complete
offset values, x: 80, y: -60, z: 204
```

```
ready to calibrate
Accel calibration complete
offset values, x: -242, y: 780, z: -782
Gyro calibration complete
offset values, x: 84, y: -60, z: 202
```

```
ready to calibrate
Accel calibration complete
offset values, x: -241, y: 867, z: -767
Gyro calibration complete
offset values, x: 73, y: -62, z: 201
```

Figure 2. Calibration Offset Values

As seen in figure 2 above my offset values were consistently similar to each iteration run. Using these calculated values, my average offset values are calculated below:

Final Report

By Samantha Butler

	x	y	z
Acceleration Offset Value	-238	831	-782
Gyroscope Offset Values	79	-61	202

Table 1. Average Offset Values

Moving forward I plan to hard code these values into my IMU sensor upon running my Arduino sketch so that my robot does not have to run the calibration every time upon start-up.

Step 2. Determining Tilt Angle

Things to keep in mind when determining my positive and negative roll tilt angles are that I have attached the acrylic guard on the front of my robot which will limit my positive roll angle significantly. Because I have nothing attached on the back of my robot, I should expect my negative pitch angle to be close to 90 degrees. Because of the structure of my robot and the way the wheels are mounted I included code to calculate the pitch angle however, do not expect my robot to flip over in that direction unless acted upon by blunt force. Because of this, I have only included results regarding to the roll angle. My first iteration prior to filtering the roll angle output was as follows:

```
roll: -78.46    pitch: 0.98                    roll: 31.49     pitch: 0.49
roll error: 78.46       pitch error: -0.98     roll error: -31.49      pitch error: -0.49

roll: -77.40    pitch: 0.49                    roll: 31.42     pitch: 0.19
roll error: 77.40       pitch error: -0.49     roll error: -31.42      pitch error: -0.19

roll: -77.70    pitch: 0.84                    roll: 30.72     pitch: -0.14
roll error: 77.70       pitch error: -0.84     roll error: -30.72      pitch error: 0.14

roll: -78.08    pitch: 0.99                    roll: 31.80     pitch: 0.44
roll error: 78.08       pitch error: -0.99     roll error: -31.80      pitch error: -0.44
```

Figure 3. MPU roll angle range

From the data above my roll angle range was from approximately -77.91 to +31.36 degrees. An example of my results can be seen on the serial plotter below:

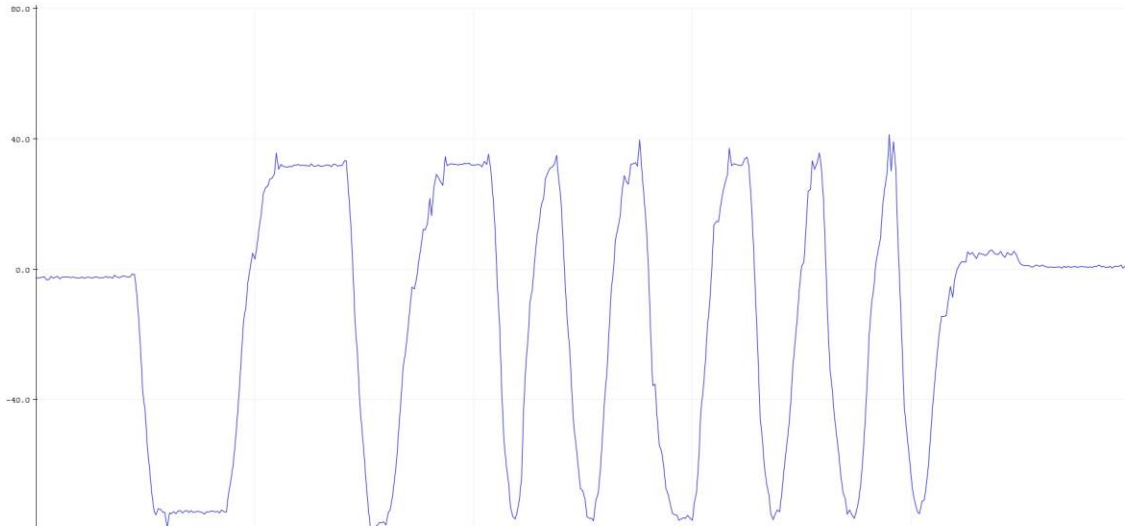

Figure 4. Pre-filtered exemplified roll angle range

Final Report

By Samantha Butler

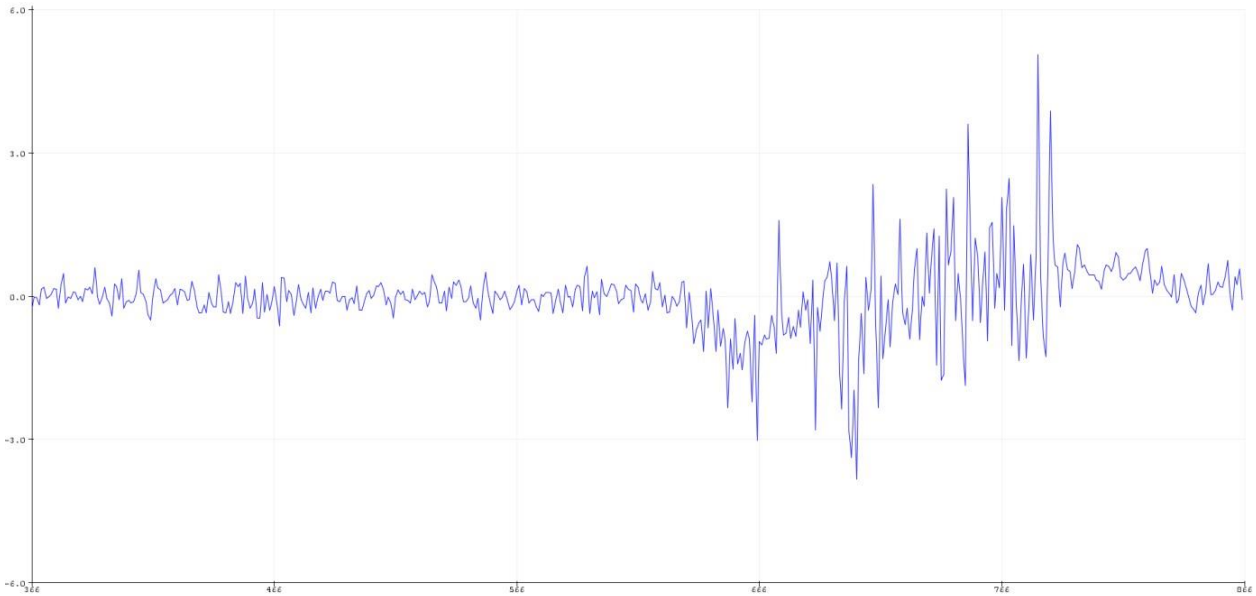

Figure 5. Pre-filtered exemplified roll with vibration

As you can see in figure 5, the raw roll data is very scattered and does not do well when any vibration occurs to the robot. Because of this I decided the roll angle should be put through the low pass filter and then passed through the complementary filter with the gyroscope angle. After doing so I got the following results.

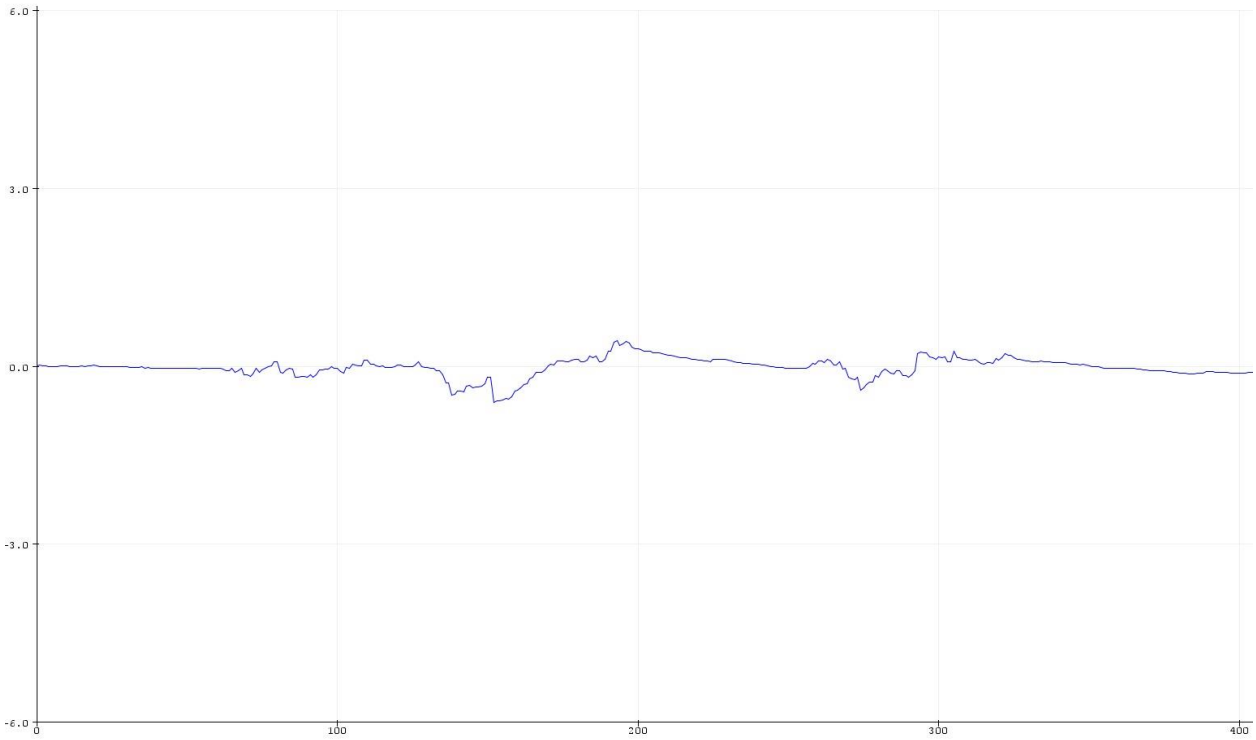

Figure 6. Complimentary filtered roll angle with vibration

Final Report

By Samantha Butler

After passing the raw data through the filter you can see how the data is significantly less sporadic and consistent, even with vibration. It was up until this point as to what had been completed in the project. The next phase was putting together all of the information I had and getting the robot to balance.

Step 3. Balancing States

I originally started off with all kp =10, kd=0, and ki=0. By starting with such a low kp I realized it did not turn the motors fast enough, nor have a quick enough response time to correct the error. I increased kp until it was equal to 50 and I felt I got a quick enough response time. However, with such a high kp my robot was reacting almost too quickly. As my output showed it increased the motor's speed from 0 to its max and minimum speed of -255 or 255 so quickly that it caused my robot to move forward and backward too quickly. Figures 7 through 9 exemplify a few of the various kp, ki, and kd values I demonstrated before I burnt my right server motor out and had to buy a new one.

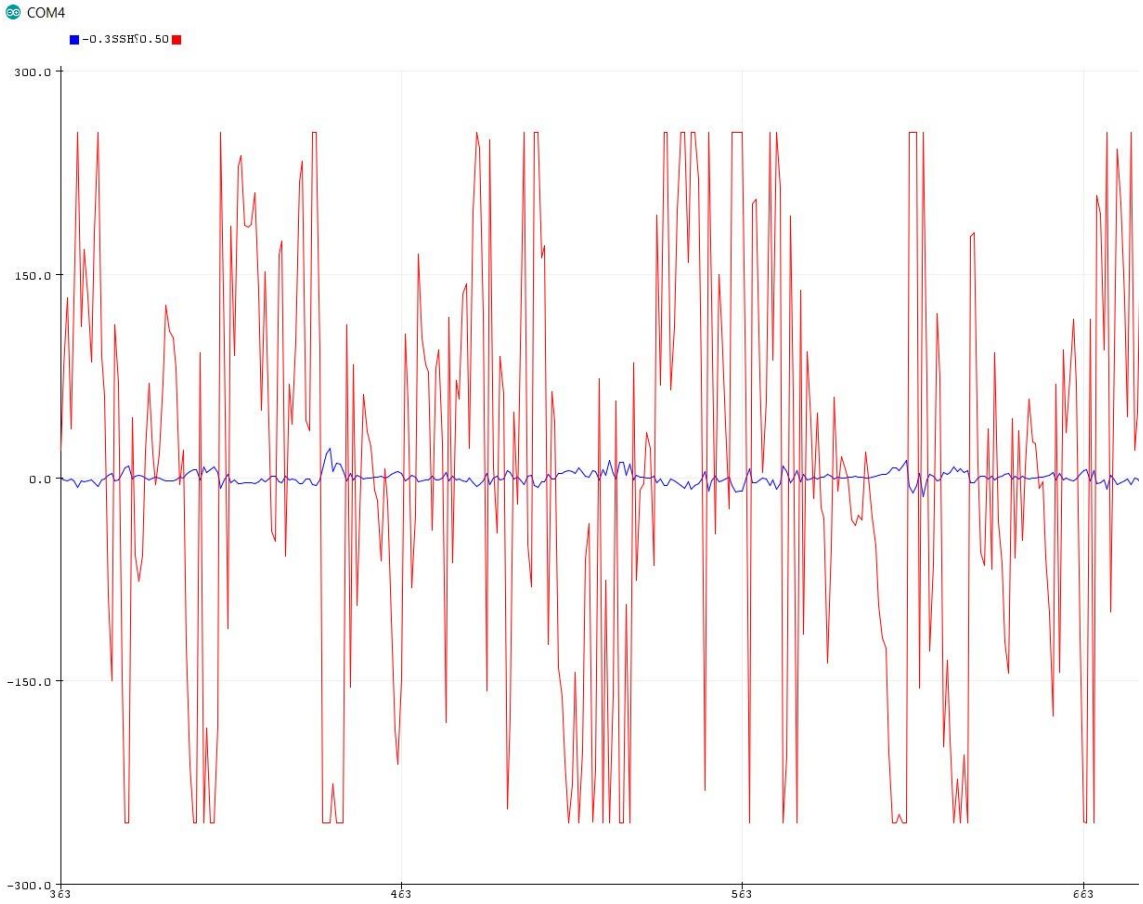

Figure 7. Motor PID speed with kp=50

Final Report

By Samantha Butler

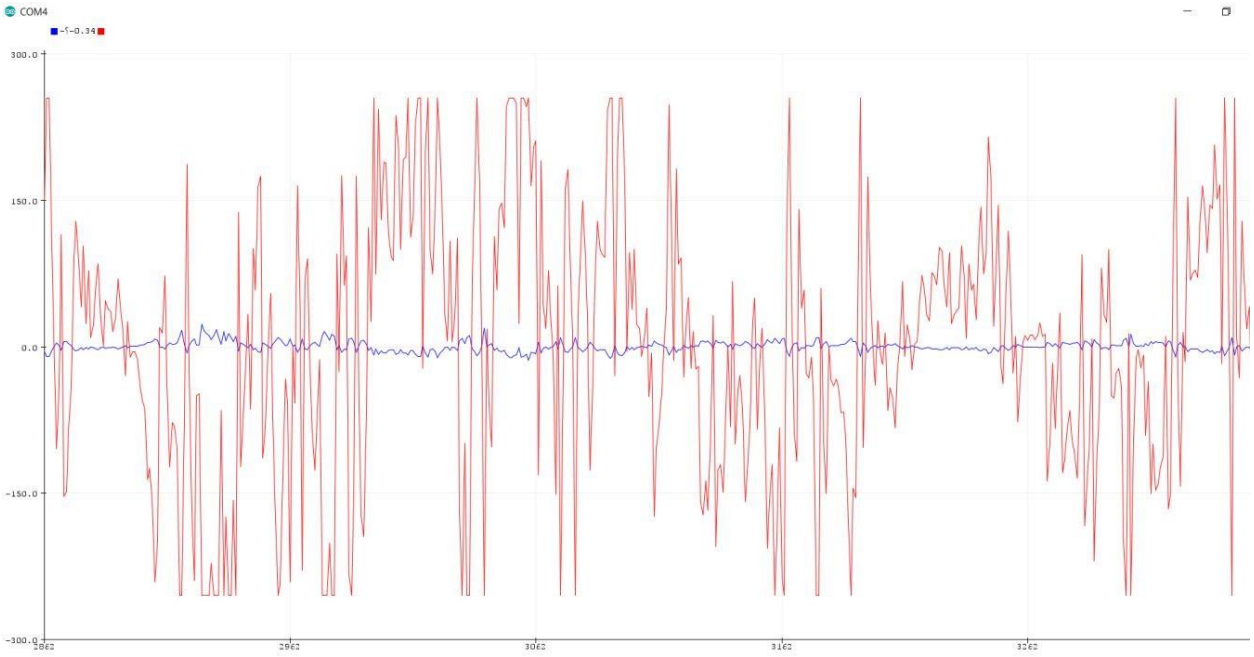

Figure 8. Motor PID speed with kp=30,kd=0,ki=0

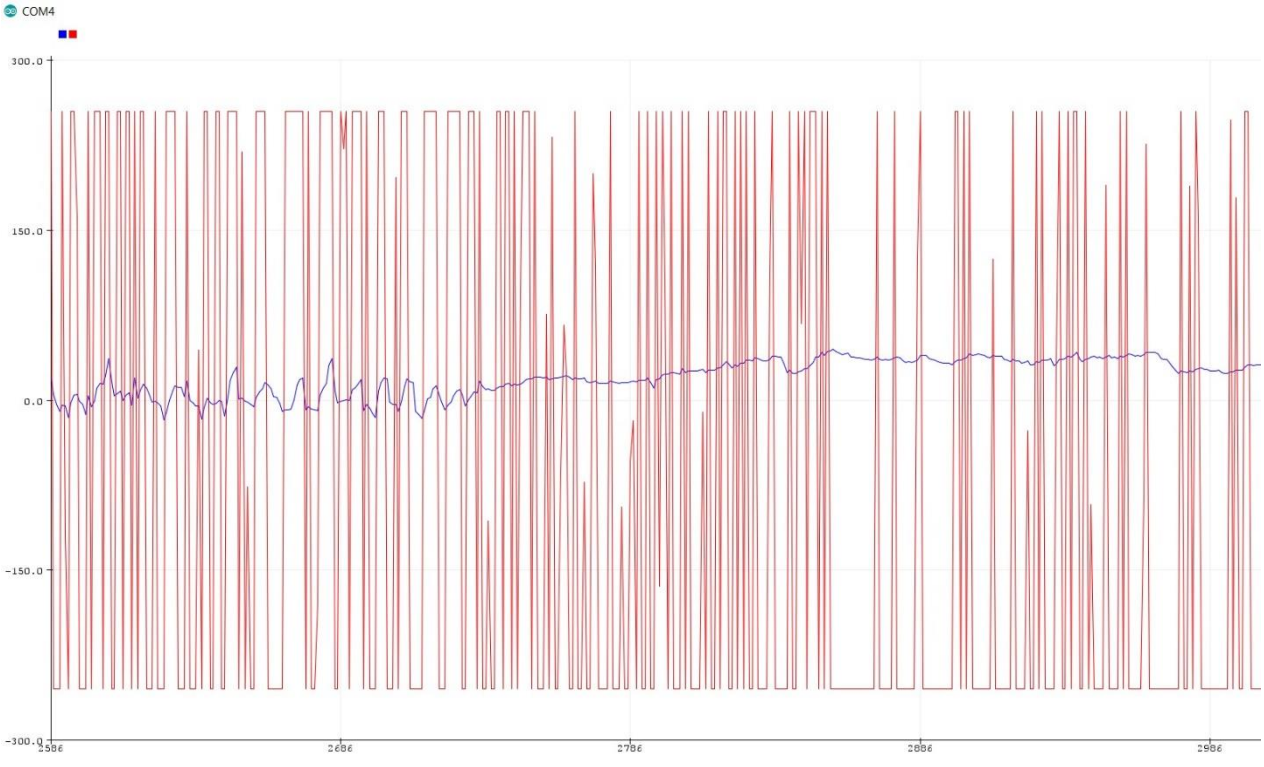

Figure 9. Motor PID speed with kp=30, kd=5,ki=0

By the time I FINALLY got my robot to balance I had to significantly reduce my kp and recalculate my calibration values to be more accurate. My final values were the following:

Final Report

By Samantha Butler

```
offset.xG = 206;
offset.yG = -185;
offset.zG = 113;
offset.x = -416;
offset.y = 737;
offset.z = 2235;
```

Figure 10. Offset Calibration Values

Phase Two

This part of my project involved implementing a successful facial recognition system for all human faces. The hardware I used was a raspberry pi3 and a raspberry pi cam attached. As easy as this sounds , it was this particular phase that took nearly a week to debug. After all was said and done, the following was overcome to get to where the recognition is set up the way it is:

- My original raspberry pi4 I was using had a bad usb port
- Then the hdmi port would not display anything to my monitor
- Could interface with the pi but not the camera/video via ssh
- The raspberry pi3 I switched to had faulty behavior in regards to its wireless connection
- SD became faulty and would no longer store any made files on it
- A bad USB cable relaying incorrect serial values from the Arduino to the pi
- Lastly, my battery powering my pi died

Needless to say, it seemed like the odds were against me and I was not able to achieve the level of sophistication I wanted to in this project.

For recognizing faces I leveraged the assignment we did using Haar Cascades. By using Haar Cascades I was trying to invoke this object detection method using a boosted cascade of simple features. This is a machine learning based approach where a cascade function is trained from a lot of positive and negative images. It is then used to detect objects in other images. In my particular case I was detecting faces. Luckily using opencv's libraries I was able to get my raspberry pi can up and running in no time. My final function in order to do is summed up below:

faceDetection(classifiers,windowWidth, windowHeight, scaleFactor, minNeighbors, minSize, nameOfWindow')

It was then that I thought it would be awesome if my pi could differentiate between which person was who. But I soon came to the conclusion that I do not have a large enough data set of my partner and I to be able to properly train my machine learning algorithm. After doing research I came across the concept of generative adversial networks (GANs). They enable one to be able to generate fairly realistic synthetic images making it hard to tell the difference between the fake and real images. Because I did not have a lot of photos of myself I decided to try and implement this GAN with a few photos I do have and be able to have it generate more synthetic images of me. ‘

The GAN is made of two parts the generator and the discriminator. The generator takes in a random point in the latent space an generates a fake image. The discriminator take in both real and fake images and predicts whether the image came from the training set or is from the generated network.

Final Report

By Samantha Butler

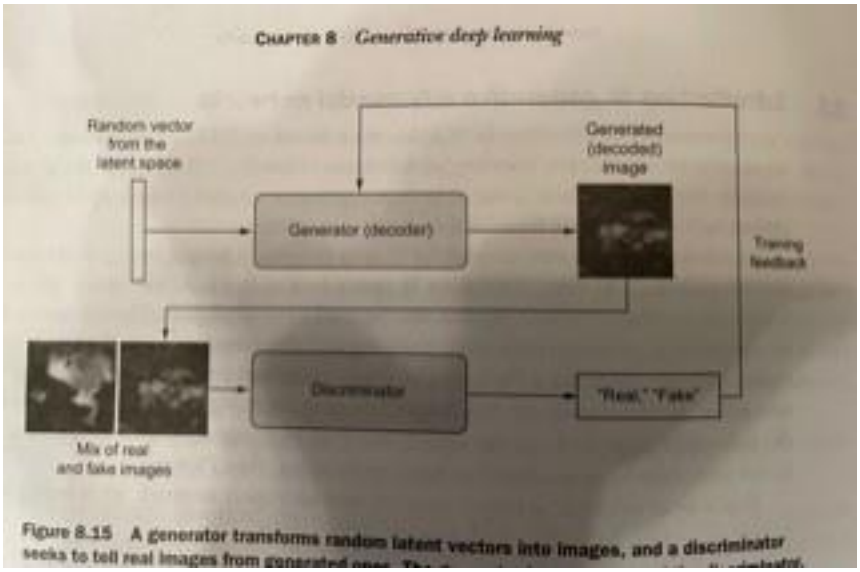

Figure 8.15 A generator transforms random latent vectors into images, and a discriminator seeks to tell real images from generated ones.

Phase Three

My goal for phase 3 was to take the location of the generated box around my face and calculate how far from the edges of the fixed camera window it has moves and then to send a character via serial communication to the Arduino nano which would move the robot according to where ones face was. I had generated the functions to do so on the Arduino but ran out of time to put it completely together. The generated functions to differentiate between which was to turn the robot can be seen in figure 11.

```
//**** DIRECTION COMMANDS ******//

float directionCommand(char command, unsigned long encoderStartTime){
  float rollTarget = readXYcoordinates();
  if (command == 'e') //Balance
  {
    rollTarget = 0.0;
    #ifdef DEBUG
      Serial.println(rollTarget);
    #endif
    return rollTarget;
  }
  if (command == 'f') //Forward
  {
    rollTarget = 0.28;
    #ifdef DEBUG
      Serial.println(rollTarget);
    #endif
    return rollTarget;
  }
  if (command == 'b') //Backward
  {
    rollTarget = readXYcoordinates();
    #ifdef DEBUG
      Serial.println(rollTarget);
    #endif
    return rollTarget;
  }
  if (command == 'l') //LeftPivot
  {
    actualLSpeed = leftMotorSpeed(encoderStartTime);
    actualRSpeed = rightMotorSpeed(encoderStartTime);
    encoderStartTime = millis();
    leftMotorBackward((char)leftMotorPID(100, actualLSpeed));
    rightMotorForward((char)rightMotorPID(100, actualRSpeed));
    delayMicroseconds(500);
    return 0.0;
  }
  if (command == 'r') //RightPivot
  {
    //actualLSpeed = leftMotorSpeed(encoderStartTime); //Not being used
    //actualRSpeed = rightMotorSpeed(encoderStartTime); //Not being used
    encoderStartTime = millis();
    leftMotorForward((char)255);   //leftMotorForward((char)leftMotorPID(100, actualLSpeed));
    rightMotorBackward((char)255); //rightMotorBackward((char)rightMotorPID(100, actualRSpeed));
    return 0.0;
  }
  return 0.0;
```

Figure 11. Direction Control functionality