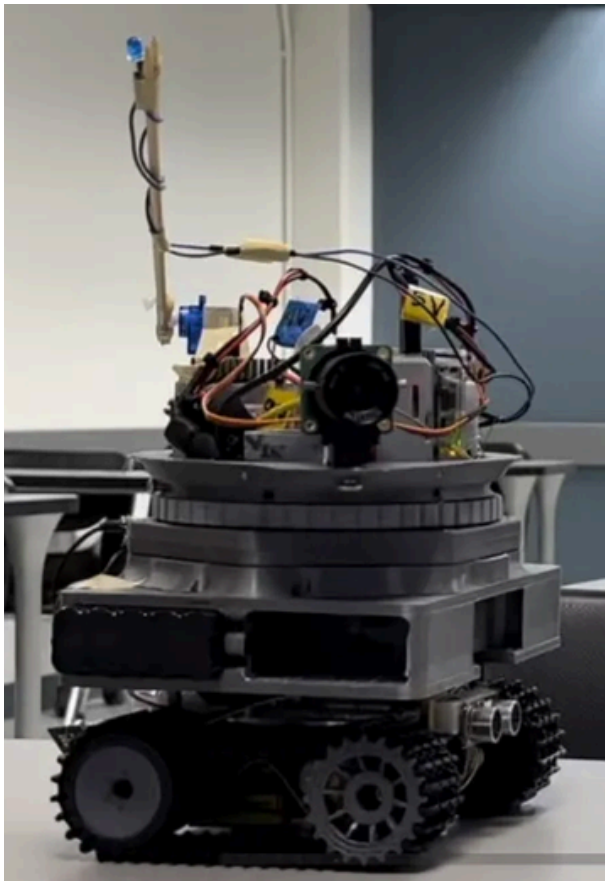
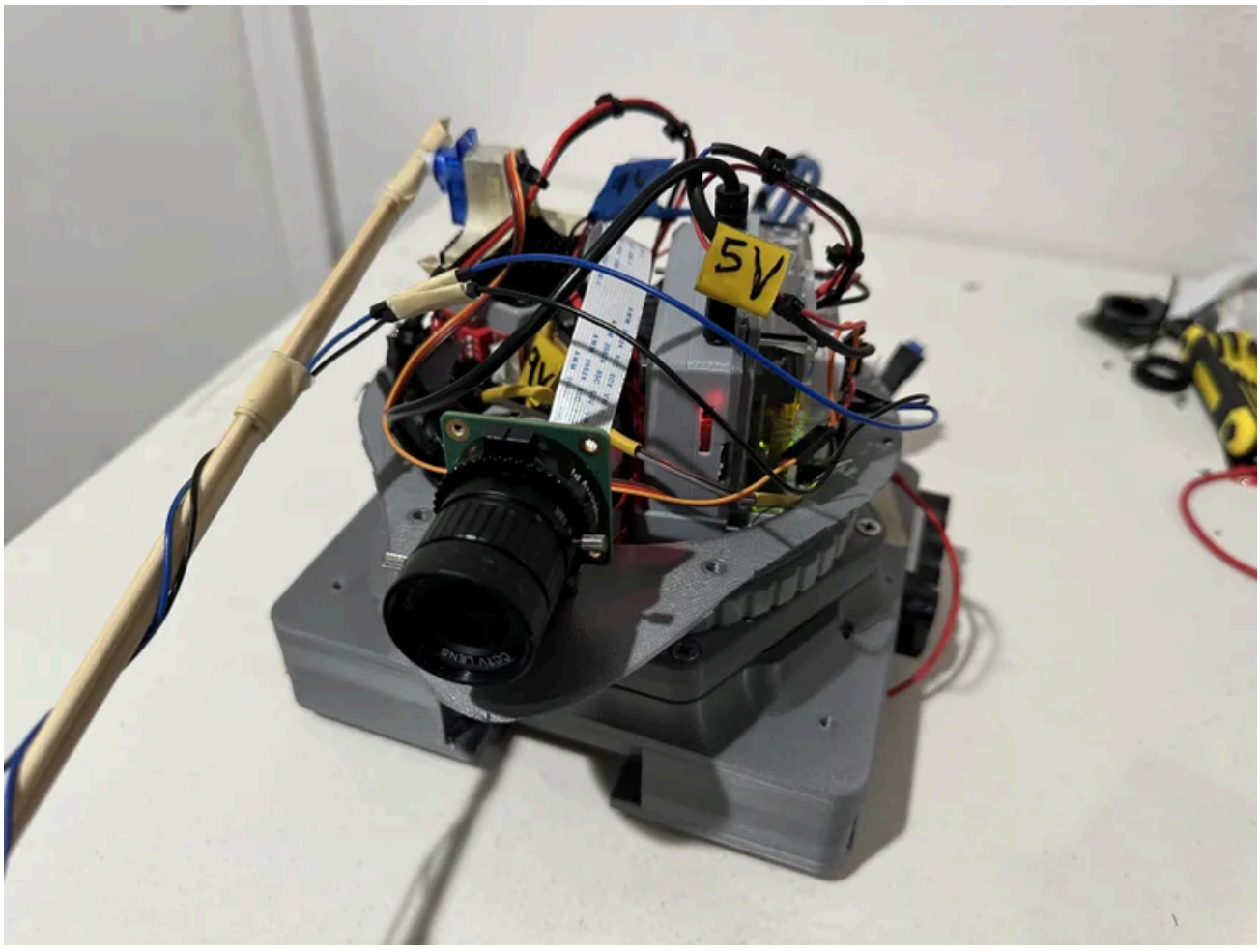
AUTO-AIMING TURRET — JANUARY - DECEMBER 2023



This was my first time experimenting with a Raspberry Pi, serial communication, Haarcascades and later LBPcascades. It was my first real experience in combining my programming experience with my hardware experience, and interfacing the two. While the CAD and mechanical innovations were easy, the low level computer science concepts offered a much steeper learning curve.

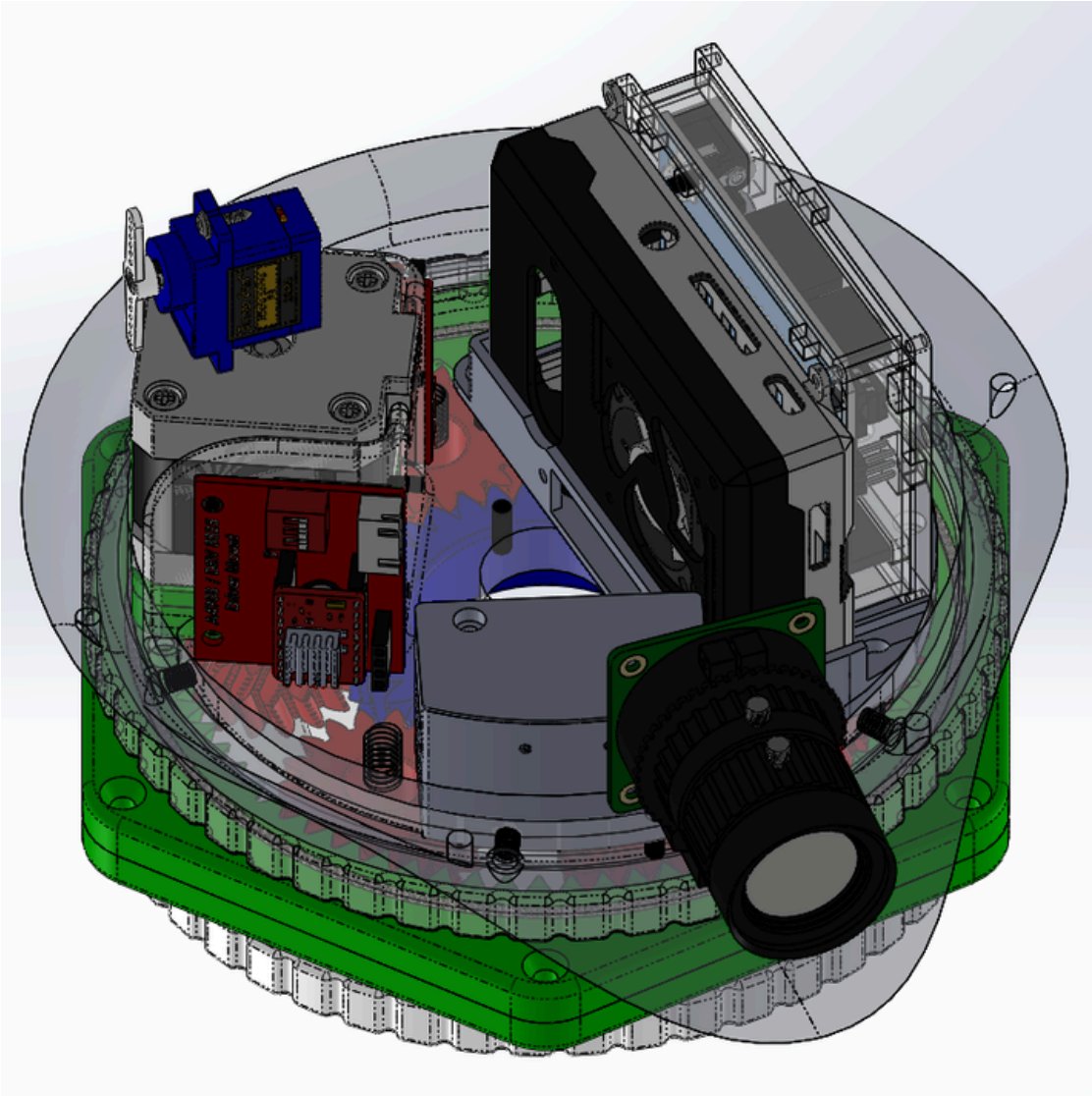
The Auto-Aiming turret uses a Pi-Cam, to gain visuals, a Raspberry-Pi, to do image processing and coordinate fining, and an Arduino, which receives instructions from the Pi, and then commands a stepper motor, moving the turret. The turret will aim for the nearest face, and will try to center its view with the target. The farther the face is from the “centerline” of the camera view, the faster the turret moves towards it.

This project took nearly an entire year, and most of the time was really spent learning concepts such as serial communication, different communication protocols, threading, and image buffering. From about January to March, I was still experimenting with serial communication. Then, I started getting hardware and 3D prints done from about June to August. October to December was spent assembling and then finally getting both code and mechanicals to run together.

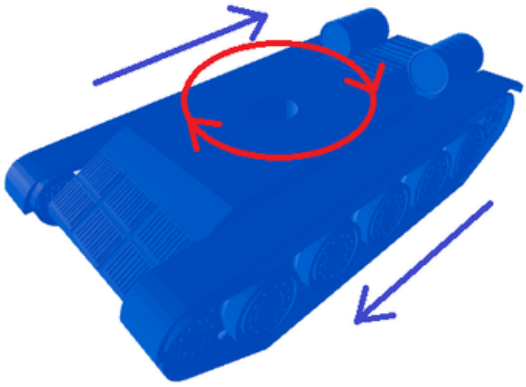


Although I was experienced in Solidworks already, the shear number of parts and redesigns made the CAD take a total of about a month's work. (I had to spread my focus to other things during June - August) The turret has three general subassemblies:

- Movement
  - Turret
    - Ring gear and Planet Gear Base
      - What allows the base to move, and has a 1:3 ratio
    - Turret Base (Planet Carrier in Planetary Gear terminology)
      - Where everything is mounted onto
      - There is a space dedicated to mounting the Arduino and Pi together
      - There is a dedicated "box" for the stepper motor to fit into. Its walls allow an A4988 stepper motor driver to be mounted onto it, ensuring close connections between the stepper motor, and its driver
      - There is a dedicated area onto which to mount the power electronics
  - Motors
    - Stepper Motor, which actuates the planet gear that drives the base
      - The stepper motor is geared down with a 1:4 Planetary Gearbox
- Computers and Microcontrollers
  - Pi-Cam and Raspberry-Pi to get and process images, and then return coordinates of faces onto each frame that the camera got. This information is then sent over to the Arduino
  - Arduino then sends commands to the stepper motor drivers
  - These are housed together, ensuring easy wiring between both components of the "brain" of the turret
- Power Electronics and Motor Drivers
  - Motor Drivers, as mentioned above, are mounted onto the wall of the stepper motor "box"
  - Power electronics are all housed together in a small box (the box has a round wall, and is square everywhere else)
  - If a battery cannot be mounted onto the turret base, there are screw holes that allow a slip ring to be mounted in the center. The slip ring only needs two wires: power, and ground, as all the other signal and electrical connections are on top of the base, and move along with it.



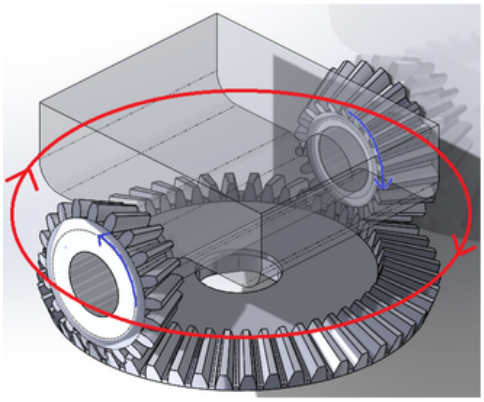
What makes the turret especially unique is that it was designed to never have wires tangle as the turret spins, and can even move past 360 degrees. This is achieved by, as mentioned before, having everything mounted onto the turret base.

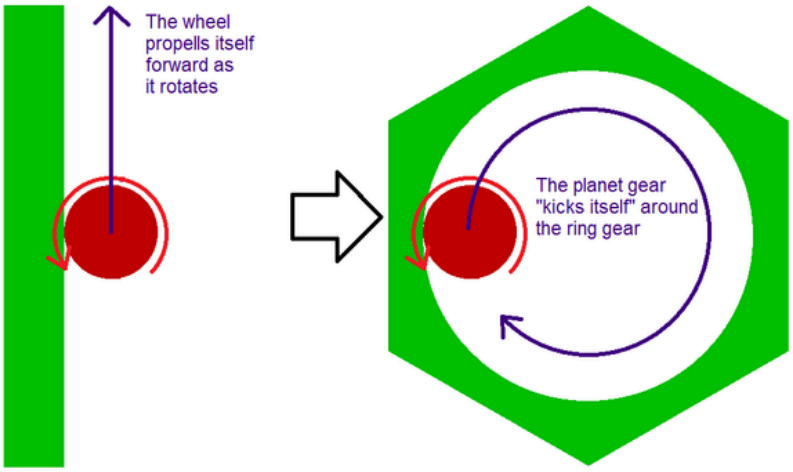


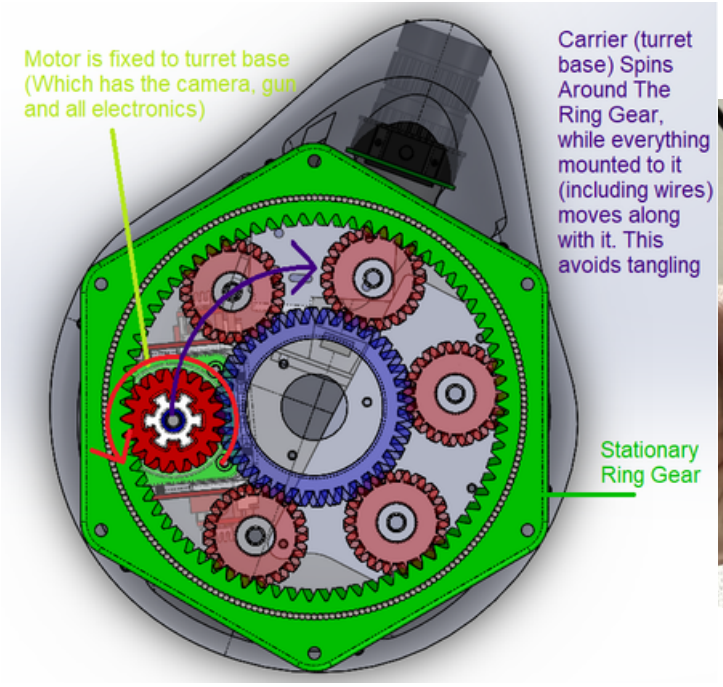
The original concept of this came from imagining a tank trying to turn - in order to turn in place, both tracks must move in opposite directions. I then imagined that instead of wheels and tracks, what if the tank was driven by gears? So I then imagined the ground being a giant bevel gear, and the tracks also as bevel gears. I then realized this could be flattened to a planetary gearset, and then I later realized that because the gears already constrained motion, I only needed to drive one gear.

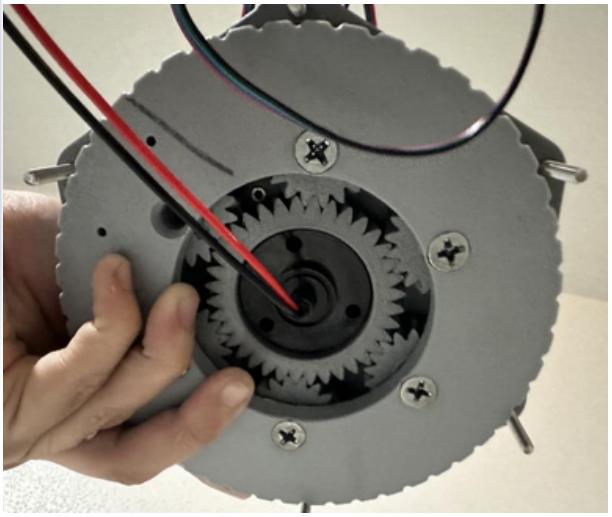
In the end, the turret moves by having its “wheel” (planet gear) kick itself against the “ground” (ring gear) to spin around. And because everything is wired on top of the “chassis” (turret base), the wires move along with it, meaning no relative motion between the ends of the wires (both ends of the wires are on the “chassis”), meaning no tangling.

This gearing also creates a reduction in speed, and an increase in torque. The final ratio is 1:3, in which the planet gear must rotate 3 times in order to make a full revolution around the ring.

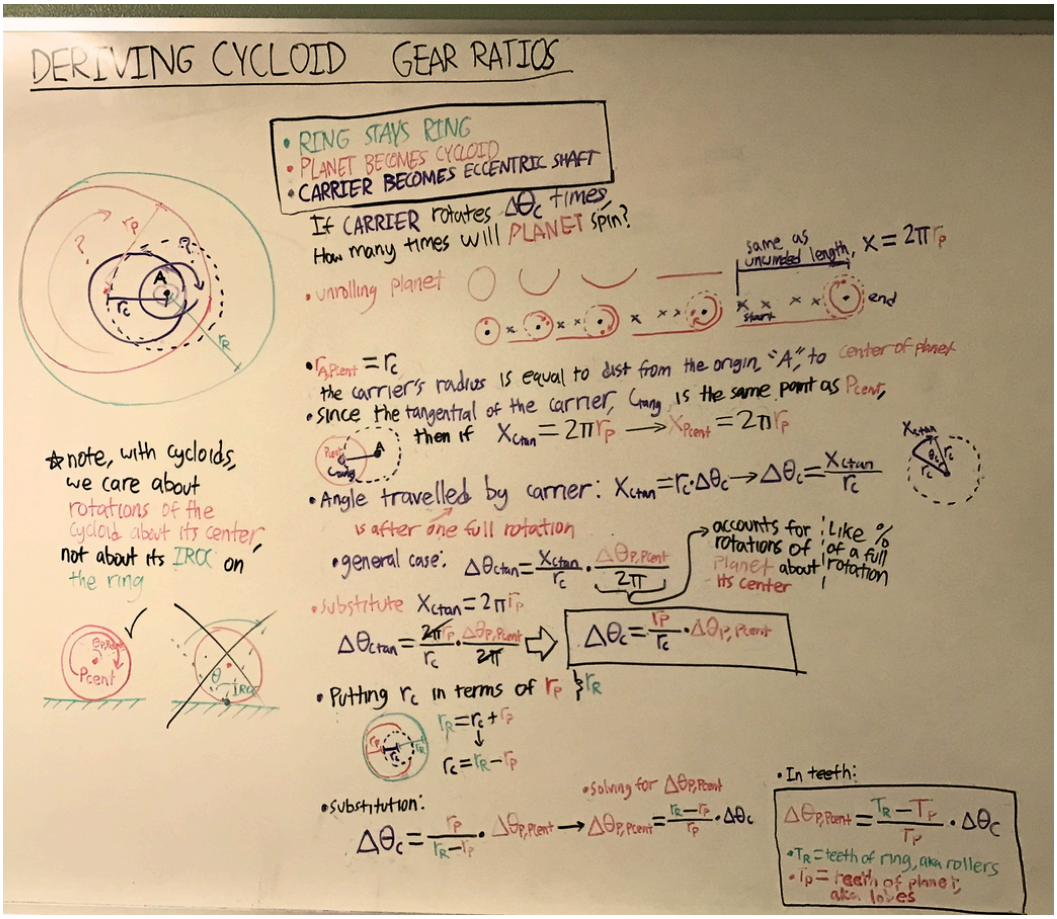








AUTO-AIMING TURRET — RING GEAR BASE CALCULATIONS



Keeping in mind that the Carrier is the turret base:

- $T_R = 72$
- $T_P = 18$
- So then...
- $R_{atio} = \frac{T_R - T_P}{T_P}$

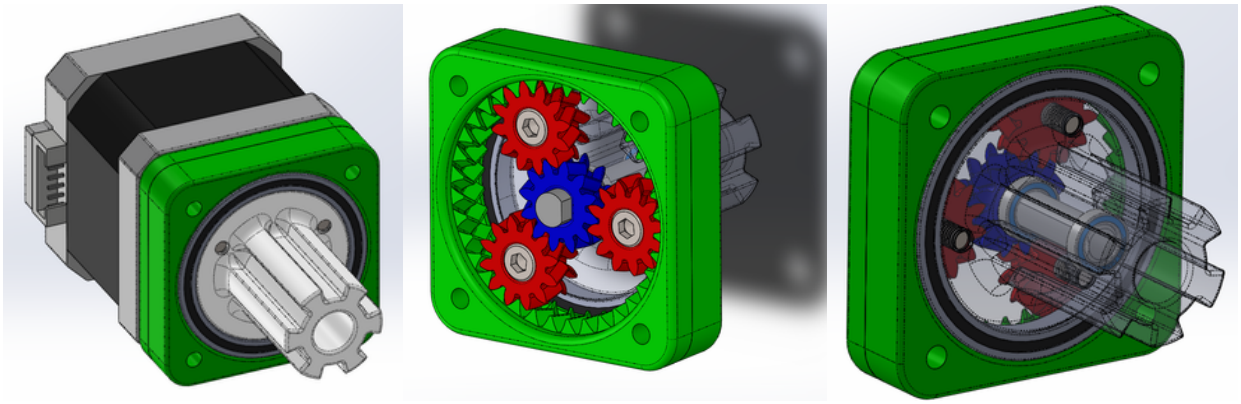
$$R_{atio} = \frac{72 - 18}{18} = 3$$

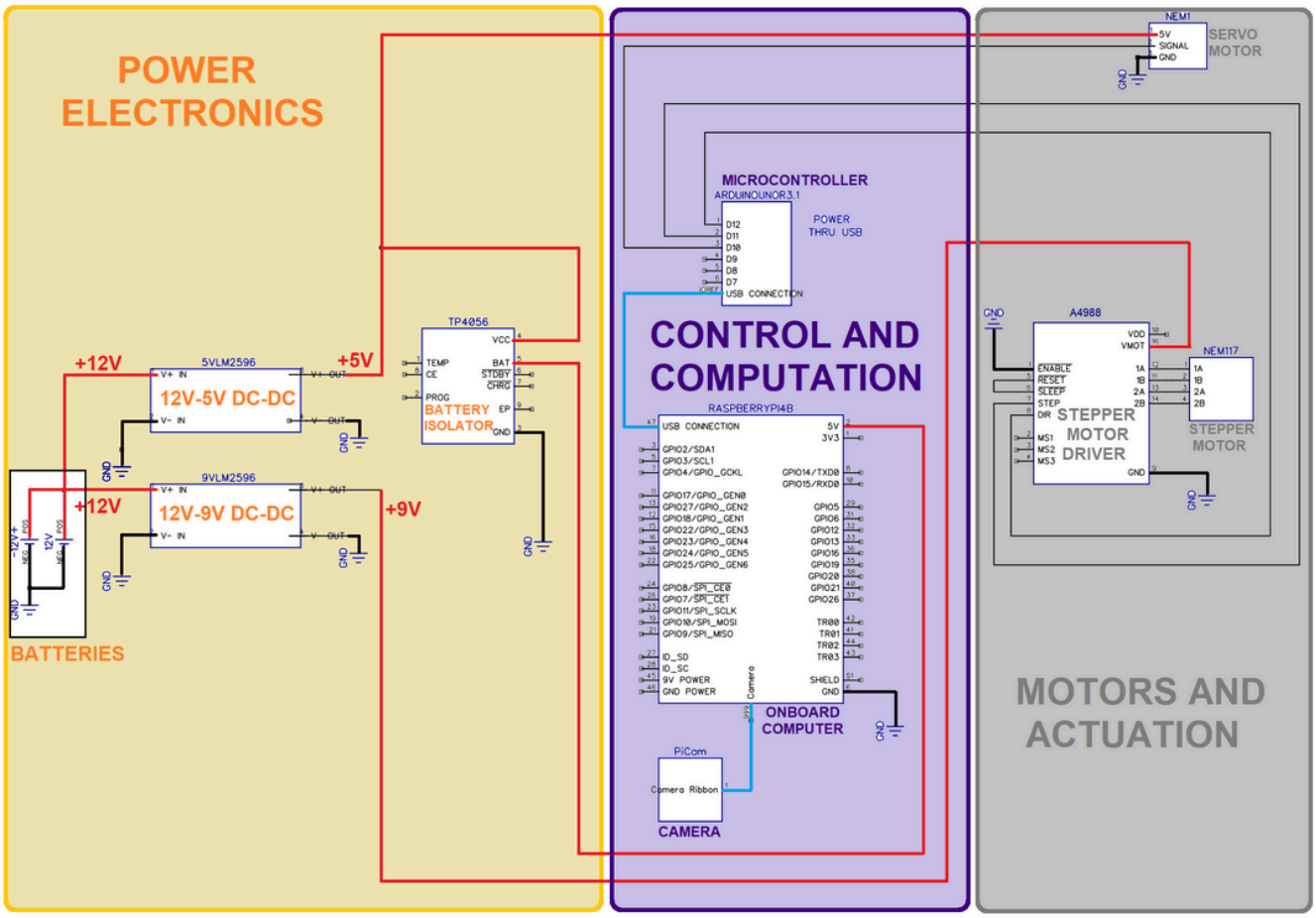
AUTO-AIMING TURRET — STEPPER MOTOR AND GEARBOX

Because I was testing this first without the turret, I needed to ensure that the motor I chose would move the a desired test angle, and I needed to be able to verify the angle it hit. DC motors need a proper PID feedback for this, so to simplify things, I went with a stepper motor, because assuming they don't skip steps, they always move to their desired location without any feedback needed.

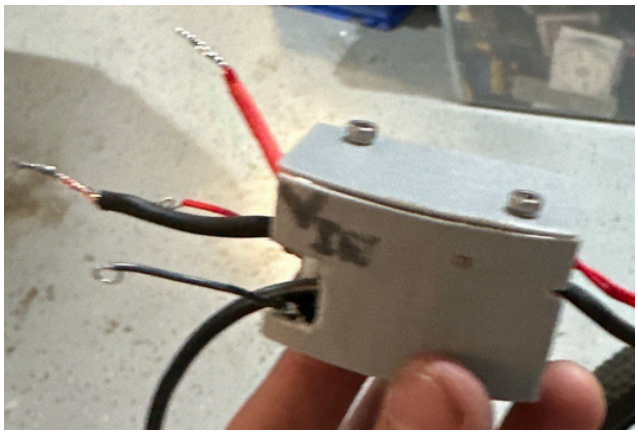
I first tested it by sending it manually typed coordinates through the Arduino serial. Then, I connected the camera and Pi to send coordinates to the Arduino. Before having the system where it would try to center itself with the target, initial testing had it so that the motor would point towards your face as you moved it in front of the camera, and stop there. Its coordinate (this will be elaborated upon later) would not be centered

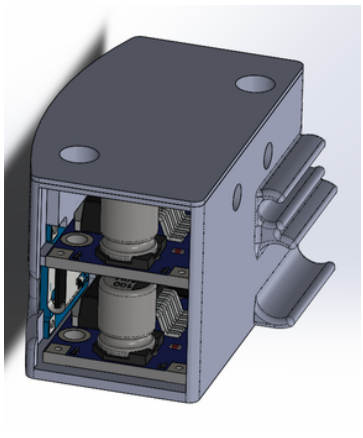
The problems arose when I assembled the motor and turret together. Despite the inherent reduction of the ring base, I found that it was still too difficult for the motor to actually move the base with everything else on top of it weighing it down. So I then decided to gear down the stepper motor with a planetary gearbox. The output shaft (connected to the planet carrier) was centered against both the stepper motor shaft (two blue bearings), and against the ring gear (black bearing), ensuring straight alignment and operation. The gearbox was a 1:4 reduction, meaning that coupled with the ring base's inherent reduction of 1:3, the final motor reduction was 1:12 - the motor would have to spin 12 full revolutions to rotate once around the ring. While this made the turret much slower to move and chase faces, it actually helped, since the pi cam would not have a fast enough framerate, making its visuals blurry when it was panning at high speeds.



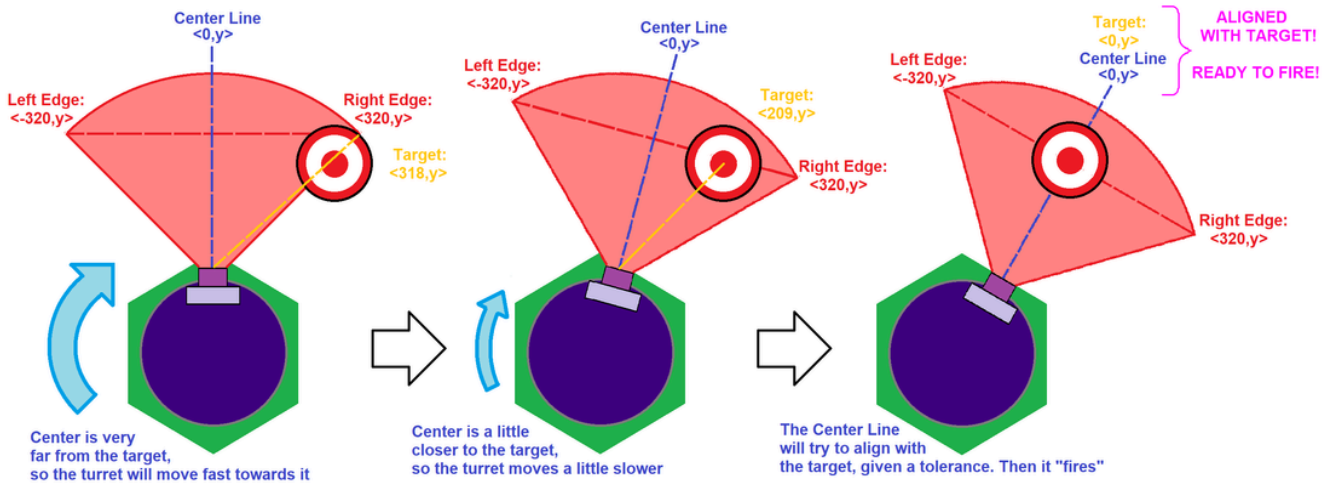


- List of electronics Used:
- Power Electronics
  - LM 2596 DC-DC Stepdown DC-DC Converter (x2)
  - TP4056 Battery Isolator (x1)
  - Slip Ring (x1) - Although the batteries are not on the turret, they only require power and ground wire through the slip ring
  - 12V NiMH 2000mAh Batteries (x2)
- Control and Calculation
  - Arduino Uno R3 (x1)
  - Raspberry Pi 4B,8G (x1)
  - Raspberry Pi HQ Camera (x1)
  - Arducam Lens (x1)
- Motors and Actuation
  - NEMA 17 42Ncm Stepper Motor (x1) - Controlled the horizontal rotation
  - A4988 Stepper Motor Driver (x1)
  - A4988 Shield (x1) - Controlled the vertical rotation
  - SG90 Servo Motor (x1)
- Etc
  - LED (x1) - The LED and buzzer would activate when the turret was properly aligned with a face, simulating “firing”
  - Buzzer (x1)

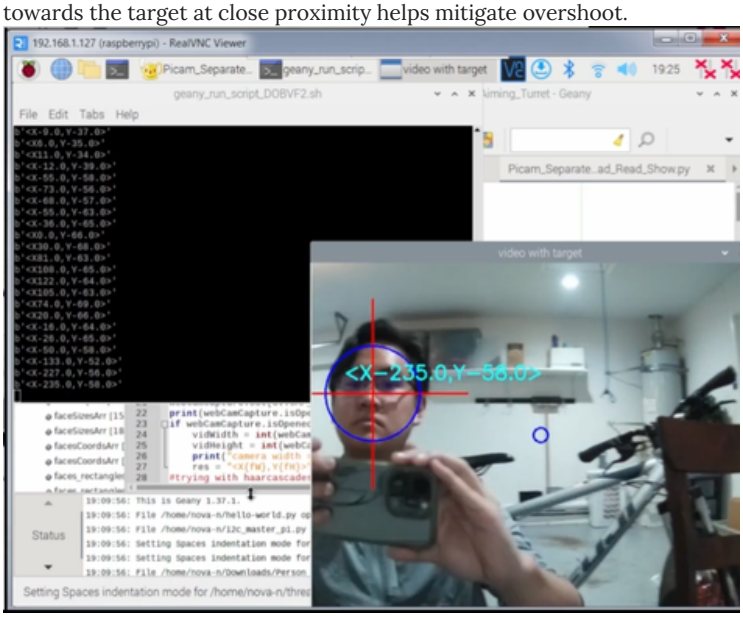




A CAD model and the actual housing for the power electronics

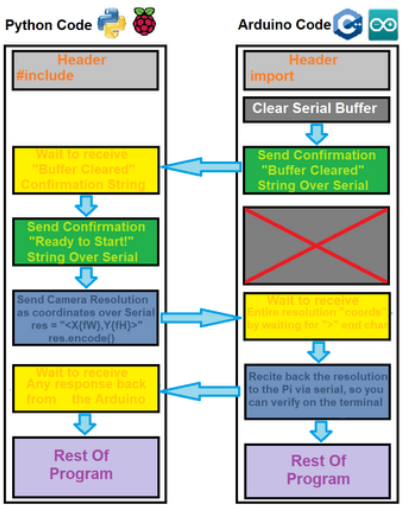


The aiming algorithm is explained in the diagram above. (It is only concerned with the horizontal, aka, the turret base turning, not aiming up, since vertical aim is done by a separate servo, and does not move the camera). As shown, the turret aims with something to proportional control. When a face, or target is within the camera's field of view, the camera will want to align itself with it. This is because the "gun" is fixed to the turret base, as is the camera, meaning if the camera is aligned with the target, then so is the gun, making it ready to fire. The farther the "Center Line" is away from the target, the faster the turret swings towards it. It is beneficial for the turret to be slow when close to the target for two reasons: Firstly, because the Raspberry Pi isn't very powerful, when the turret pans quickly, so does the camera. This causes a blurry image, and it cannot process frames that quickly, leading to a very skippy video that it reads. This means it can skip faces or skip over the target all together. Secondly, having the turret actuate slowly towards the target at close proximity helps mitigate overshoot.



The camera has a resolution of 640x480p. When the camera is on, these pixels can be turned into coordinates (note, depth is not taken into account here, but that will be addressed later). This means that the horizontal is 640 pixels wide, and 480 pixels tall. Setting the center coordinate to <0,0> (the blue circle), that means the left and right edges have coordinates <-320,y> and <320,y> respectively. The Pi then finds faces in each frame of the live video (will be expanded upon later), and translates those into coordinates. These are seen on the black terminal output window, and are constantly sent to the Arduino. As the turret moves closer to the face in question, the face's X coordinate gets closer and closer to the center line, meaning the X coordinate gets closer to <0,y>. When this occurs, and the servo motor points the "gun" (The wooden stick with lights), it "fires".

AUTO-AIMING TURRET — START AND SYNCHRONIZATION



The arduino needs to know when it is being sent coordinates by the raspberry pi, and the raspberry pi needs to know when it can even start sending coordinates over. This is what the synchronization on startup is for.

The Arduino will always start faster than the Pi's Python code, since the Arduino immediately starts once the power is turned on. It needs to wait for the Pi to send over the camera resolution, so that it can calculate how much to move the motor, mapping the coordinate's position on the Pi's screen, to motor movements from the Arduino.

Once it has received the resolution, the Arduino can start receiving coordinates, and will always be waiting for them in the serial buffer.

I used the "Serial Communication Basics" tutorial by Robin2 on the Arduino Forums to learn how to transmit data via the serial bus to and from thar Arduino. The serial buffer has to constantly anticipate for incoming characters, and as soon as one is read, it gets tossed out of the buffer. This is why these characters must be stored.

To know when a message starts, it picks up on the "<" char. This means that any letters and numbers after it can be parsed, and are for sure part of the same coordinate, and not some left over number from the previous sent coordinate.

To know when a message ends, it picks up on the ">" char, meaning that any other numbers or characters that come after it are irrelevant to that coordinate. The string "<X--,Y-->" then gets stored as a variable, and then is further processed.

The ParseInt() function is known as a blocking function, meaning that any code after the function call will not run until the entire string has been parsed. Parsing the string "<X--,Y-->" using the will cause the motor to run jagged, because the code is constantly starting and stopping, because it has to get through several characters before reading the numbers.

```
void recvStartEndMarkers(){
  //static is a variable specific to a function. When it is called repeatedly, it remembers its value between calls,
  //despite being defined initially in the function. This happens until the program ends. The variable isn't reinitialized.
  //this is similar to defining a variable globally so that it doesn't reset every time you call the function
  static bool recvInProgress = false;
  //static byte ndx = 0; //byte is an unsigned number from 0 to 255, so takes one byte of memory
  static int ndx = 0;//C++ doesn't recognize byte, so just using int instead. int is signed, so takes 2 bytes.
  char startMarker = '<';
  char endMarker = '>';
  static char rc;
  //is static bool recvInProgress, since entire coordinate is NOT sent all at once over serial

  while(Serial.available() >0){
    rc = Serial.read(); //picks up one at a time
    if(recvInProgress == true){
      if(rc != endMarker){ //while the chars aren't '>', keep going.
        //Serial.println(rc);
        recievedChars[ndx] = rc; // ndx increments with the loop, and is the char position of the new recieved char
        //Serial.println(recievedChars);
        ndx++;
        if(ndx >= numChars){ //if ndx is position 32 or greater, ndx becomes position 31
          ndx = numChars - 1;
          //How many characters can be received?
          //In the examples I have assumed that you will not need to receive more than 32 bytes. That can easily be altered b
          //Note that the 64 byte size of the Arduino serial input buffer does not limit the number of characters that you ca
        }
      }else{
        //Serial.println(recievedChars);
        recvInProgress = false;
        recievedChars[ndx] = '\0'; //terminates the string, since will increment one more times than the # of chars
        strcpy(tempChars,recievedChars);//copies receivedChars onto tempChars
        ndx = 0;
        coordsReady = true;
      }
    }else if(rc == startMarker){
      recvInProgress = true; //if rc finds '<', then starts the if statement above
    }
  }
  //Serial.println(recievedChars);
  return;
}
```

The coordinates are "Parsed" by using token delimiters. When a coordinate is read through by the above function, what is saved is "X--,Y--". The start and end markers are removed. The only thing left to remove are the commas, and letters. This is what strtok() does. What remains is "X--Y--". The numbers after the letters must now be saved to their respective coordinates.

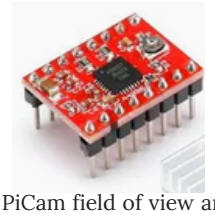
It would be cumbersome to have a variable for each coordinate. X for x, Y for y, Z for z, and so on. What if I had several more coordinates than X and Y? This is why I stored my coordinates in an array. [X,Y]. As seen, the first slot in the array goes the X coordinate. The Y coordinate goes into the second.

```
void coordAssigner(char fromSerial[],float (&coordsGiven)[sizeof(coordAxes)],const char *delim,const char coordAxesGiven[]){
  //don't need to redefine default argument in function definition, just need to do it once
  //need to specify size of coords array, or will get error "reference to array of unkown bound int"
  char *token;
  token = strtok(fromSerial, delim); //breaks out the tokens X100 Y200 and Z400 out
    //but if you check token, it only displays 1 at a time. Must call strtok again, but with NULL to get the next token
  int iteration = 0;
  while(token != NULL){
    if(token[0] == coordAxesGiven[iteration]){ //compares the first char from the token to the char of the coordAxes, so compares char to char
      token ++; //just moves it past the letter X, Y or Z to the number
      coordsGiven[iteration] = atoi(token);
    }
    iteration ++;
    token = strtok(NULL,delimiter); //gets to the next token
  }
  for(int j = 0; j<sizeof(coordAxes);j++){
    //Serial.print(coords[j]);
    //Serial.print(",");
  }
  //Serial.println();
  return;
}
```

```
desiredMoveHoriz = -1.00 * round( coords[0]/cameraResolution[0] * actualSteps);
```

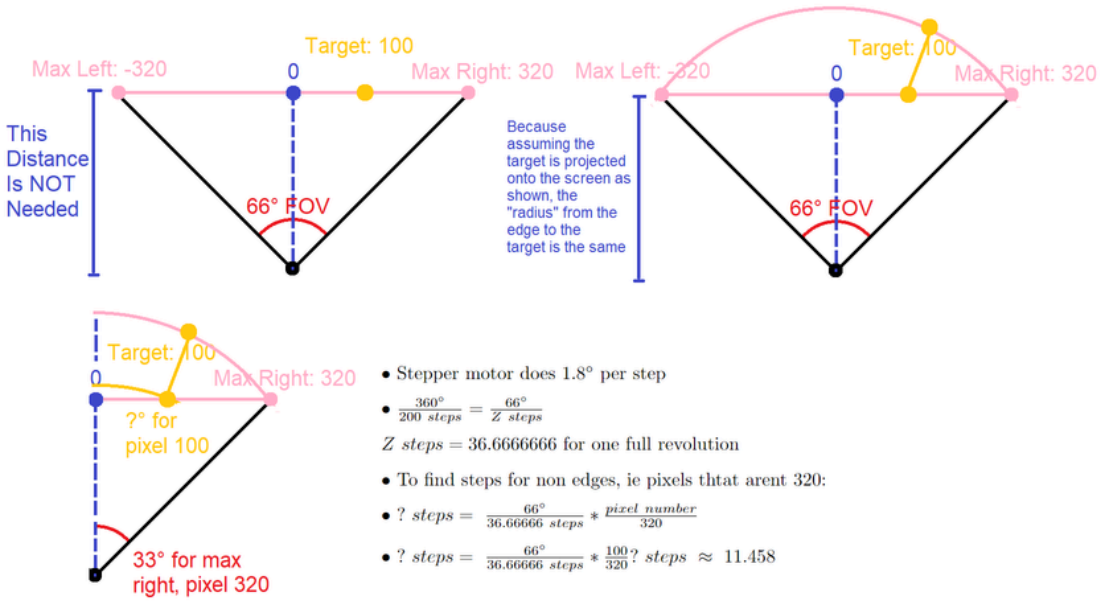
AUTO-AIMING TURRET — ARDUINO MOTOR CONTROL

The stepper motors on the Arduino were controlled by an A4988 stepper motor driver, and the library that came with it.



```
#include <AccelStepper.h>
```

I used a NEMA 17 stepper motor, which did 200 steps to do one complete revolution. The PiCam field of view angle was 66°, and the camera resolution was 640p horizontally. I needed these to solve for how many steps it would need to take to track a face. Drawn below is the mathematical and geometric explanation.



This simple math was easy to test. Before attaching the camera to the turret, I simply attached a piece of tape to the stepper motor shaft. When the Pi would send coordinates, the Arduino would move the motor, and I could see the shaft position easily. The tape would easily point in my direction, without overshoot.

AUTO-AIMING TURRET — FACE TRACKING AND CAMERA CODE

```
cascade_front_faces = cv.CascadeClassifier("/home/nova-n/git/Auto_Aiming_Turret/LBPcascades/lbpcascade_frontalface.xml")
cascade_side_faces = cv.CascadeClassifier("/home/nova-n/git/Auto_Aiming_Turret/LBPcascades/lbpcascade_profileface.xml")
```

Although I first used Haarcascades to detect faces, I found it was too slow for the responsiveness I needed. This is why I switeched to LBPcascades. Although it was less accurate, the turret would at least not lose the face if it moved too quickly out of view, because there was less lag to process and move with LBP.

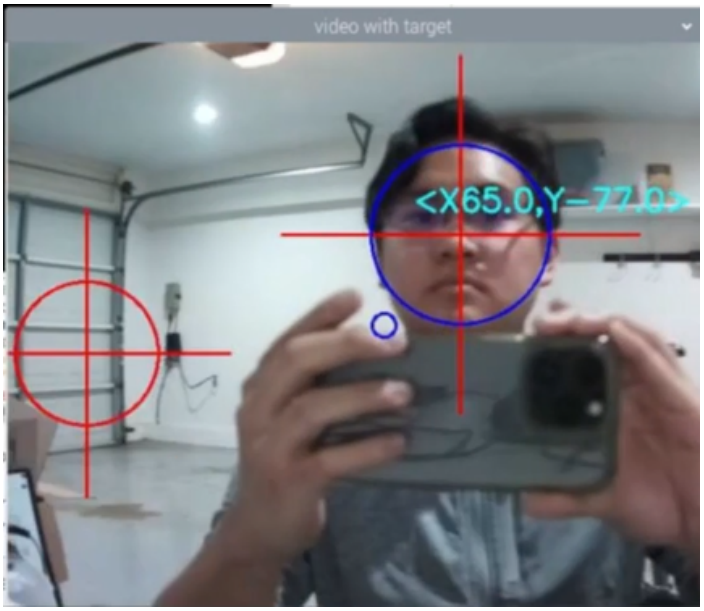
Each frame of the camera then went through processing, so that faces could be identified, and marked on screen.

```
faceSizesArr = [] #reset face sizes array
facesCoordsArr = [] #reset the coordinates array
for (x,y,w,h) in faces_rectanglesVid:
    #cv.rectangle(frame,(x,y),(x+w,y+h), (0,0,255),2)
    #cv2.circle(image, center_coordinates, radius, color, thickness)
    cv.circle(frame,(x+w//2,y+h//2),(w+h)//4,(0,0,255),2)
    cv.line(frame,(x+w//2,y-h//2),(x+w//2,y+3*h//2),(0,0,255),2)
    cv.line(frame,(x-w//2,y+h//2),(x+3*w//2,y+h//2),(0,0,255),2)
    faceSizesArr.append((w+h)//4)
    facesCoordsArr.append([x+w//2,y+h//2])

#If no faces detected, will only send <X0.00,Y.00> ONCE before not sending anymore
if len(faceSizesArr) == 0 and wasEmpty == False:
    coords = "<X{fX},Y{fY}>STOP-NOTHING-HERE".format(fX = "NULL", fY = "NULL")
    arduino.write(coords.encode())
    print(coords.encode())
#sending the largest target, aka largest face, aka, closest face to the arduino
if len(faceSizesArr) != 0:
    wasEmpty = False
    largestFaceIndex = faceSizesArr.index( max(faceSizesArr) )
    cv.circle(frame,(facesCoordsArr[largestFaceIndex][0], \
    facesCoordsArr[largestFaceIndex][1]), max(faceSizesArr),(255,0,0),2)
    #Note, I want the center of the screen to be (0,0), but it defines the top left as 0,0, so I must modify the code
    coords = "<X{fX},Y{fY}>".format(fX = facesCoordsArr[largestFaceIndex][0] - centerCoord[0] \
    , fY = facesCoordsArr[largestFaceIndex][1] - centerCoord[1])
    arduino.write(coords.encode())
    print(coords.encode())
    cv.putText(frame , coords,(facesCoordsArr[largestFaceIndex][0]-40,facesCoordsArr[largestFaceIndex][1]-20), \
    cv.FONT_HERSHEY_SIMPLEX,0.9,(255,2550),2)
else:
    wasEmpty = True; #notice executes AFTER sending <X0.00,Y0.00>
#cv.circle(frame,(int(2*centerCoord[0]),int(2*centerCoord[1])),1,(255,0,0),2)
cv.circle(frame,(320,240),10,(255,0,0),2)
```

LBP had more false positives than Haarcascades. But even if the face detection software detecteed multiple faces, the turret would not get confused. The Pi would only send the closest target's coordinates to the Arduino. As a face/target gets closer to the camera, it will appear larger than faces further away. This data is taken into account, via the radius of the circle that is marked on each face.

This was also when I first learned to use threads, multi-processing, and what I/O bound operations were. I made classes, that created objects which had methods to read and show video. The reading and showing are I/O bound operations, so they were run in separate threads. There was no need to lock threads, because the reader just gets from the buffer, and outputs a frame. The shower takes the frame from the reader. It was best to do these in objects, because while functions can return the frames of the webcam, when starting a thread, there is no way to get the output of a function. The object can hold these frames instead, and these will be accessed




```
frameFinishedReading = threading.Event()
frameQueue = queue.Queue()
gotFrameTrueFalseQueue = queue.Queue()
#communication between threads, and also, a way for the main thread
#(the main program) to know if vidReader is finished reading, and has outputted a frame
class videoReader:
    def __init__(self,cam,lock): #initial state of an object's attributes
        #capture.read() returns two outputs: a boolean that says if a frame is succesfully returned, and each frame
        self.camera = cam
        (self.gotFrame , self.frame) = self.camera.read()
        self.stop = False
        self.timeAtBeginLoop = None
        self.timeAtEndLoop = None
    #can't use self.camera as a default value for the function
    def getAndReadFrame(self,camToRead = None): #the function that needs to be continuously called
        #the self.frame is just the initial value, and __init__ is not a
        print("called getAndReadFrame")
        camToRead = self.camera
        global latencyTime #must declare global in the actual function that accesses it
        while self.stop == False:
            self.timeAtBeginLoop = time.time()
            if self.gotFrame == False or (cv.waitKey(1) & 0xFF == ord("d")):
                self.stop == True
                gotFrameTrueFalseQueue.put(False)
                break
                return
            else:
                #lock.acquire()
                (self.gotFrame , self.frame) = camToRead.read()
                #lock.release()
                #WAITING FOR EVENT CAUSES DELAY AGAIN
                #frameFinishedReading.set() #Event Set
                #print("event set")
                #frameFinishedReading.clear()
            self.timeAtEndLoop = time.time()
            latencyTime = self.timeAtEndLoop - self.timeAtBeginLoop
        return

#Because the video displayer takes form a global queue, its best to leave this as a function, instead of an object
def videoDisplayer(displayerKeepGoing = True):
    while displayerKeepGoing:
        displayerKeepGoing = gotFrameTrueFalseQueue.get()
        if displayerKeepGoing == False:
            break
            return
        elif frameQueue.qsize() > 0:
            cv.imshow("video with target",frameQueue.get())
        else:
            pass
    if displayerKeepGoing == False:
            return
    return
```