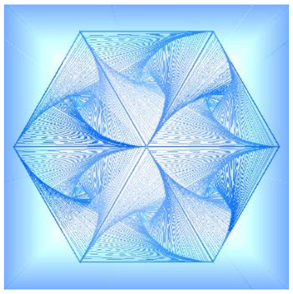
Adam Patni

Technical Documentation



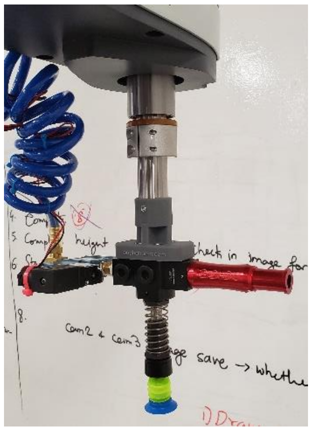
Contents

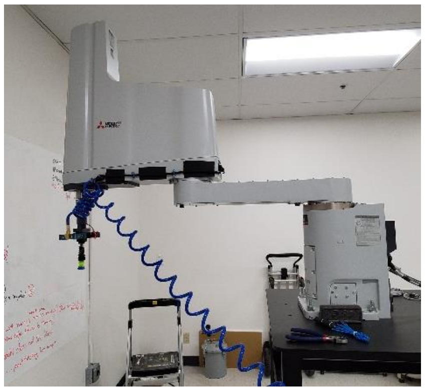
EverestLabs.AI (Summer 2019)

Objective

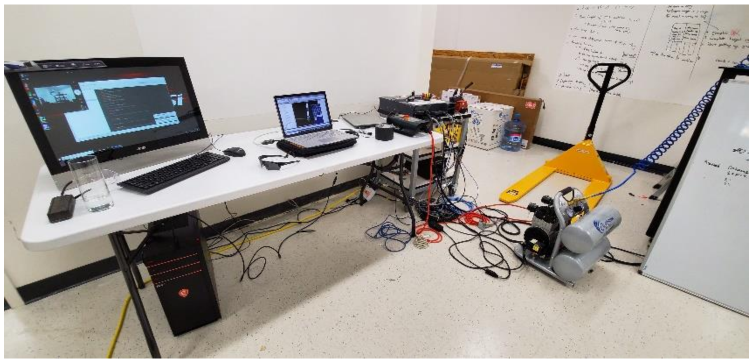
To deploy sorting and detection algorithms for recyclables in Waste Management facilities on a Mitsubishi RH-20FH Industrial 4-Axis Scara Robot.

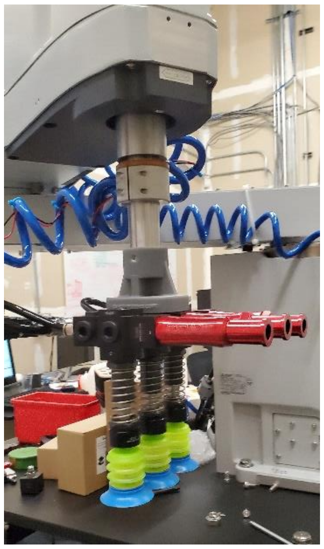
Process



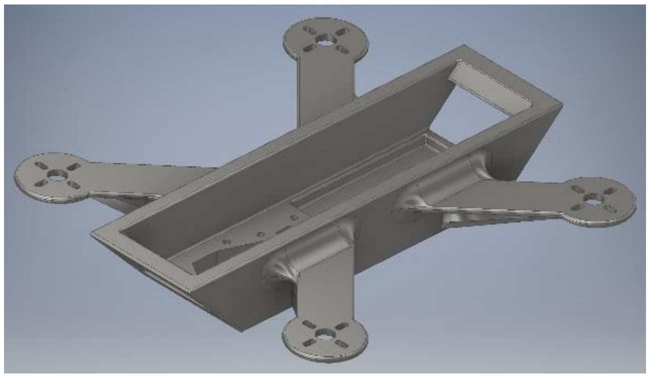


1. Assembly
   a. Design and mount suction interface
   b. Wire internal/external pneumatics and electronics
   c. Assemble main CPU along with PLC
2. Software Implementation
   a. Develop software for the Mitsubishi using MELFA BASIC programming language along with Python to integrate computer vision detection.
3. Redesign Phase
   a. Redesigned the suction mount interface to utilize three suction cups to increase the surface area long which collection could occur. This increased the time range with which the robot could successfully pick up objects.
4. Check out some cool testing video here: https://photos.app.goo.gl/FoKr7ftMGWLu





3D Printed Drone Designs (Fall 2018 – Spring 2019)



Objective

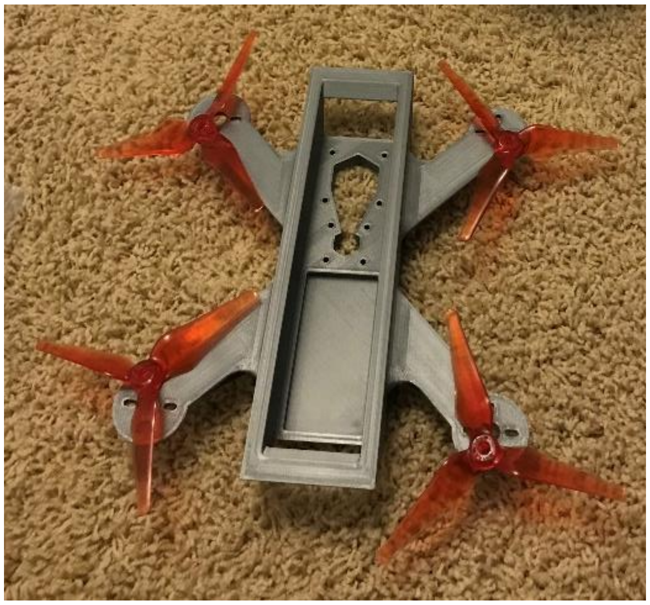
To design cost-effective, durable, and lightweight 3D printed drone frames.

Requirements

1. Must be cost-effective (<$25)
2. Must hold up under rigorous flight routines
3. Must be lightweight in order to conserve battery

Models

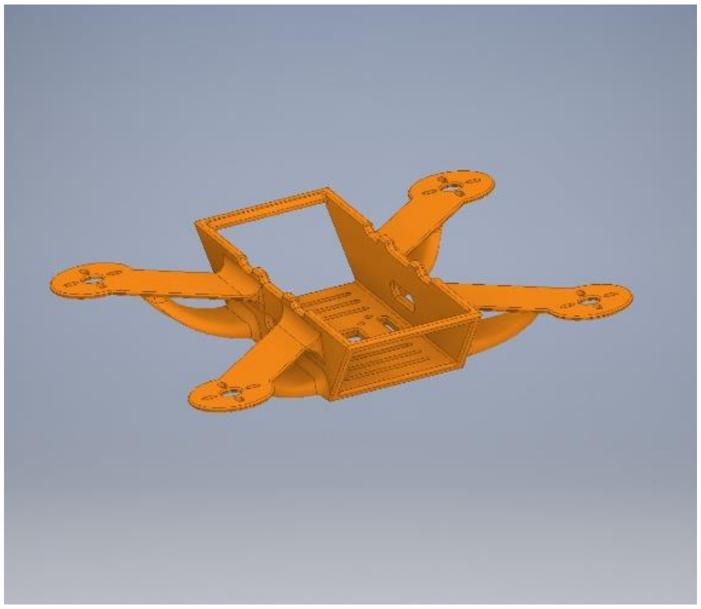
1. CAD Models



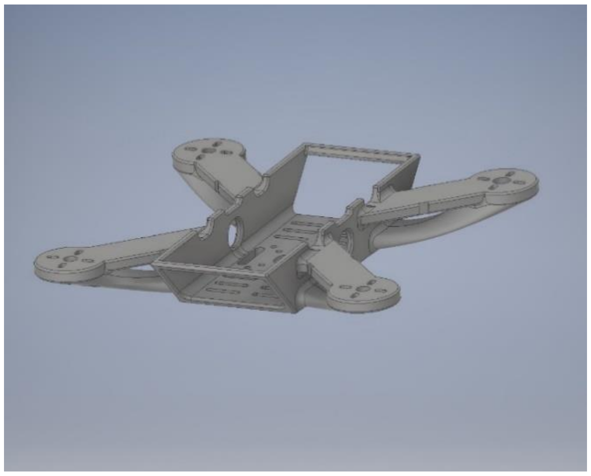
Process

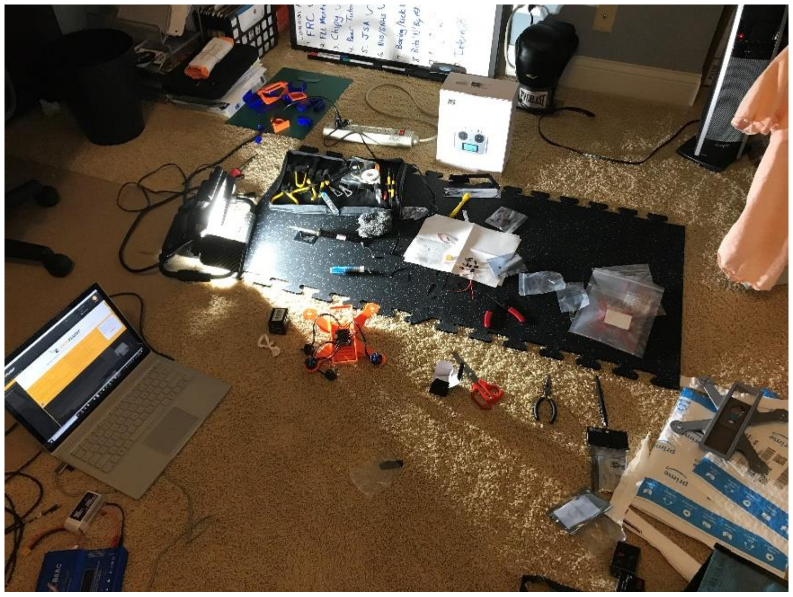
- **Iteration 1 (Gray)**
  - This iteration did not work out because my design was too heavy (.651 lbs) and the electronics board did not fit inside the frame when wires were soldered on.

- **Iteration 2 (Orange)**
  - This iteration worked a lot better than the previous one. I reduced the weight to .339 lbs and added support structures for the wings. However, after mounting the motors and turning the drone on, I could see and feel oscillations throughout the frame. After watching a slo-mo video of the wings, I realized I needed to beef up the wings in addition to their support structures.



- **Iteration 3**
  - This iteration flew! After redesigning the wing support structure and increasing the thickness, I could not feel any discernible oscillations in the frame. After addressing some issues with the electronics, I mounted the propellers and began to fly. Unfortunately, I learned that I'm not a very good flyer, and the drone got destroyed in a tree within the first few minutes…
  - I am planning on increasing the base strength for the next design. Iteration 4 is currently a work in progress.



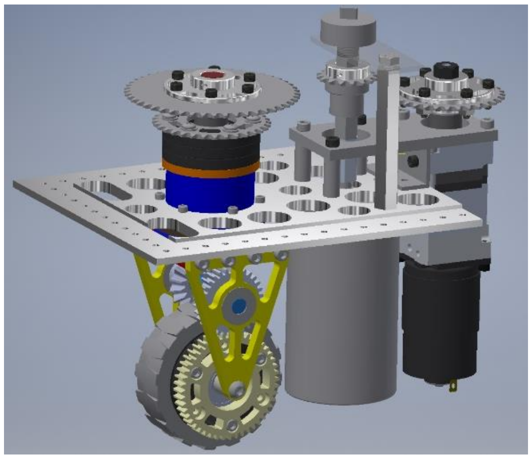


Swerve Drive Designs (Summer 2017 - Spring 2018)

Objective

To improve Team 3061 (Huskie Robotics) Swerve Drive technology by designing new modules before the season begins. Swerve Drive is a holonomic drivetrain where each of the four wheels is independently driven and steered.

Requirements



1. Must be durable and reliable
2. Must minimize backlash for control systems
3. Must utilize sensors to measure position and distance
4. Must be lightweight and compact

Models

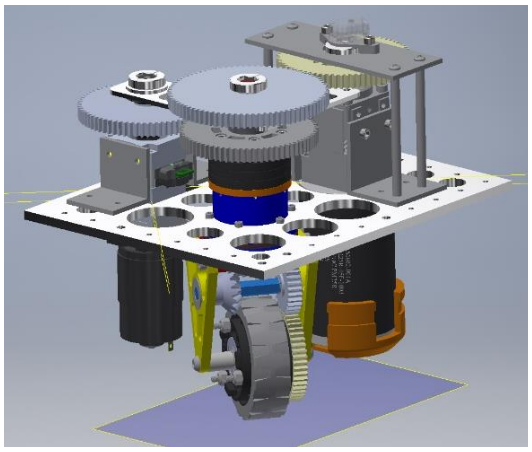
1. CAD Models
2. Location: Swerve Module 2018 FINAL.stp

Process

Huskie Robotics' Old Design:

Problems:

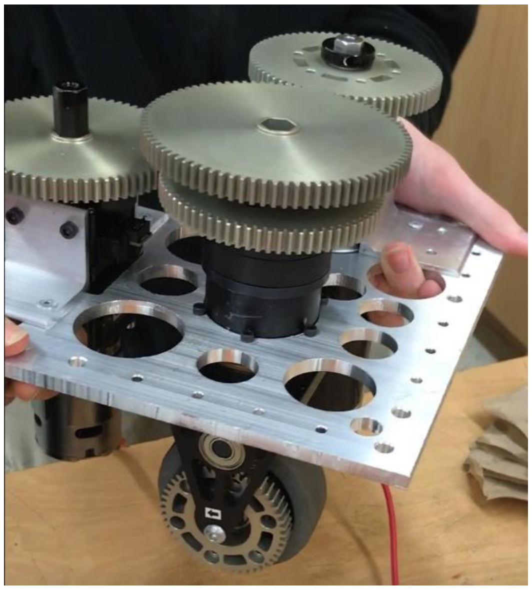
- Weight: 15 lbs * 4 modules = 60 lbs
- Chain Driven (inaccurate/backlash)
- Unreliable (broke many times)



My Design (Iteration 5):

Improvements:

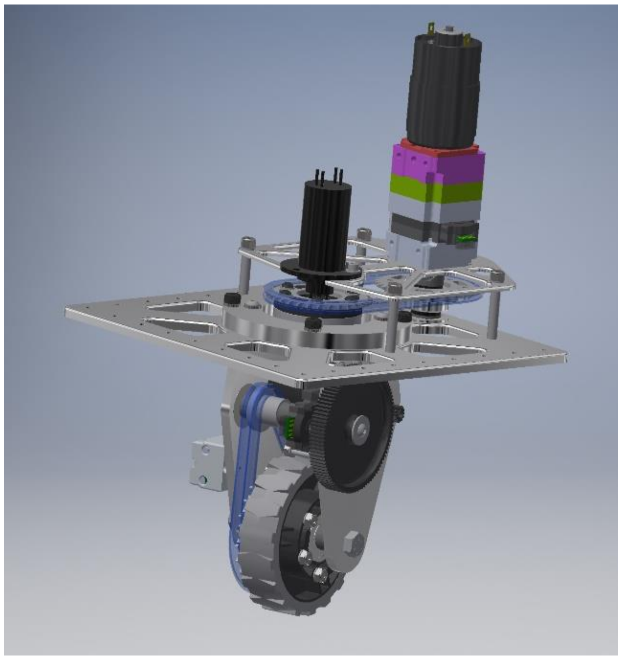
- Weight: 12 lbs * 4 modules = 48 lbs
- Gear Driven (minimize backlash)
- Durable (never needed replacement)



Drivetrain Designs (Summer 2018 – Spring 2019)

Objective

To experiment with new types of drivetrains and different ways of mounting sensors, motors, and wiring. Below are examples of my swerve and tank drivetrains.



Models

1. CAD Models
2. Location: Swerve Module 2019 v4.stp
3. Location: Gearbox 775.stp

Designs

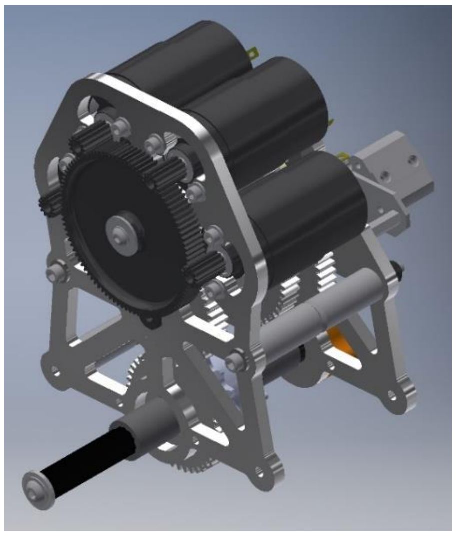
Swerve Design (Iteration 7):

Pros:

- Weight: 7 lbs * 4 modules = 28 lbs
- Chain drive (tensioned, reduced backlash)
- Compact
- Looks super cool :)

Cons:

- Lots of fabrication (16 * 4 modules = 64 parts!)
- Reduced power (smaller motors)



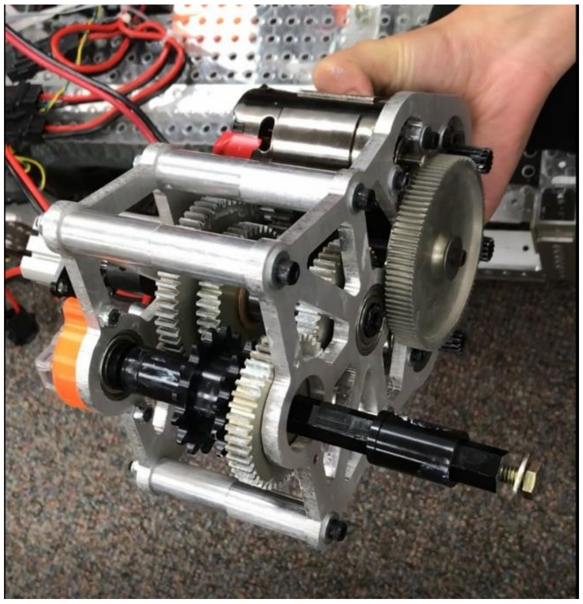
Tank Drive Gearbox (Iteration 1):

Pros:

- Compact, lightweight
- Ball shifter (high torque / low torque shifting)

Cons:

- Untested for strength
- Reduced power



Computer Vision Application (Summer 2017 – Spring 2019)

Objective

To improve the autonomous capabilities of Team 3061 (Huskie Robotics) by creating a framework that can be utilized each year to build a new vision processing system. Additionally, improve the team's path-planning algorithm by integrating machine learning.

Code

1. https://github.com/aapatni/chipy2017
2. https://github.com/HuskieRobotics/Huskie-Vision

Requirements

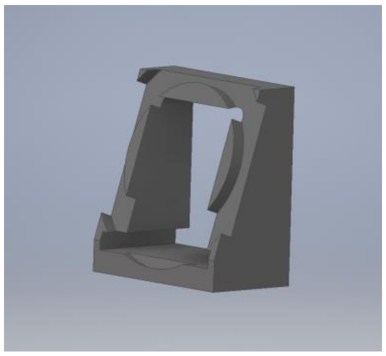
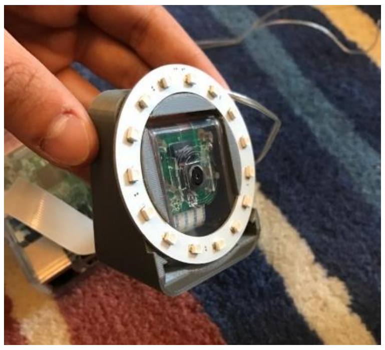
1. Must be maintainable, easy to understand, and open-sourced
2. Must consistently and accurately detect targets
3. Must control the robot's trajectory and direct it toward targets
4. Must utilize a machine-learning model to develop relationships between image and robot position
5. Must interface between Raspberry Pi and RoboRIO (onboard processor)

Materials

1. Raspberry Pi + Camera v2
2. 3D Printed Mounting Mechanisms
3. Jupyter Notebook, Python, OpenCV, OpenGL, Pandas, Scikit-Learn

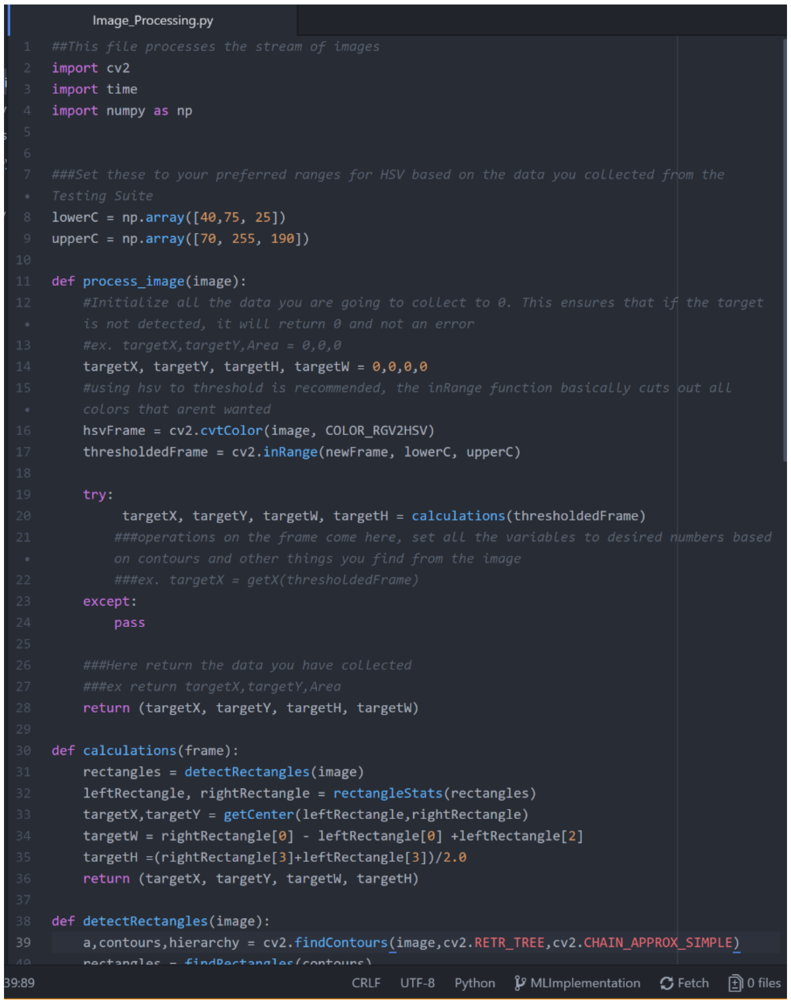
Process

Step 1: Designing the Camera Mount

The camera mount needed to both house the Raspberry Pi Camera and hold the Green LED ring that shines at the retroreflective tape targets on the field.

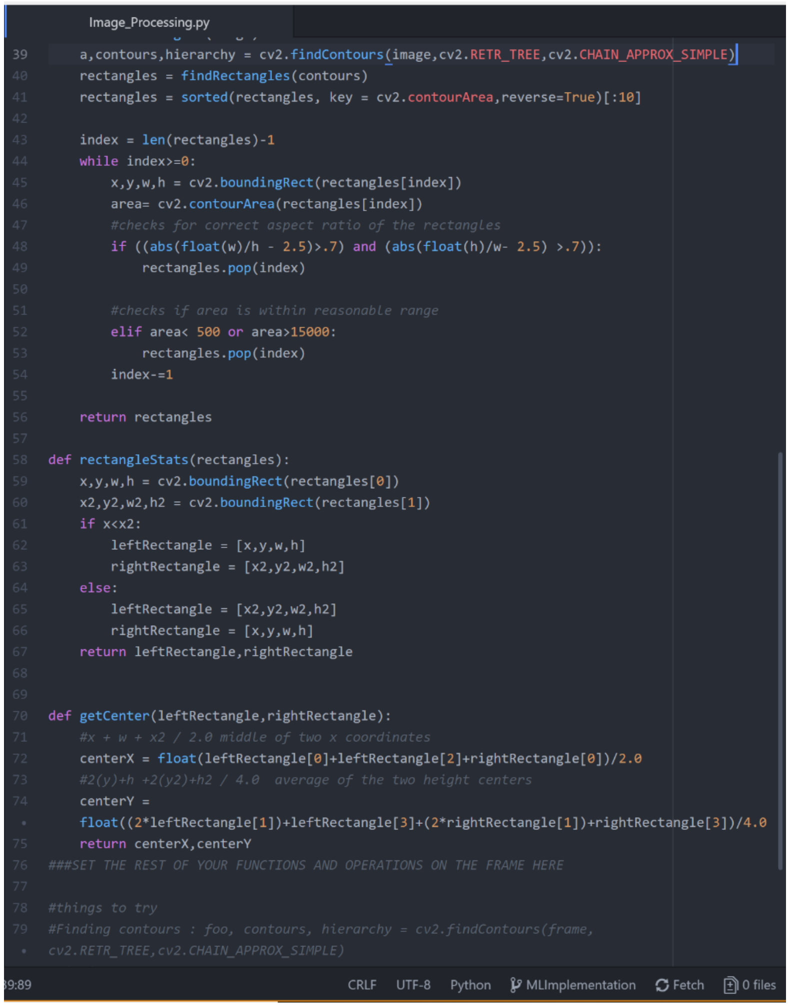
Step 2: Rectangle Detection Code



Image_Processing.py

```python
##This file processes the stream of images
import cv2
import time
import numpy as np


###Set these to your preferred ranges for HSV based on the data you collected from the Testing Suite
lowerC = np.array([40,75, 25])
upperC = np.array([70, 255, 190])

def process_image(image):
    #Initialize all the data you are going to collect to 0. This ensures that if the target is not detected, it will return 0 and not an error
    #ex. targetX,targetY,Area = 0,0,0
    targetX, targetY, targetH, targetW = 0,0,0,0
    #using hsv to threshold is recommended, the inRange function basically cuts out all colors that arent wanted
    hsvFrame = cv2.cvtColor(image, COLOR_RGV2HSV)
    thresholdedFrame = cv2.inRange(newFrame, lowerC, upperC)

    try:
         targetX, targetY, targetW, targetH = calculations(thresholdedFrame)
        ###operations on the frame come here, set all the variables to desired numbers based on contours and other things you find from the image
        ###ex. targetX = getX(thresholdedFrame)
    except:
        pass

    ###Here return the data you have collected
    ###ex return targetX,targetY,Area
    return (targetX, targetY, targetH, targetW)

def calculations(frame):
    rectangles = detectRectangles(image)
    leftRectangle, rightRectangle = rectangleStats(rectangles)
    targetX,targetY = getCenter(leftRectangle,rightRectangle)
    targetW = rightRectangle[0] - leftRectangle[0] +leftRectangle[2]
    targetH =(rightRectangle[3]+leftRectangle[3])/2.0
    return (targetX, targetY, targetW, targetH)

def detectRectangles(image):
    a,contours,hierarchy = cv2.findContours(image,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
```

39:89 CRLF UTF-8 Python MLImplementation Fetch 0 files

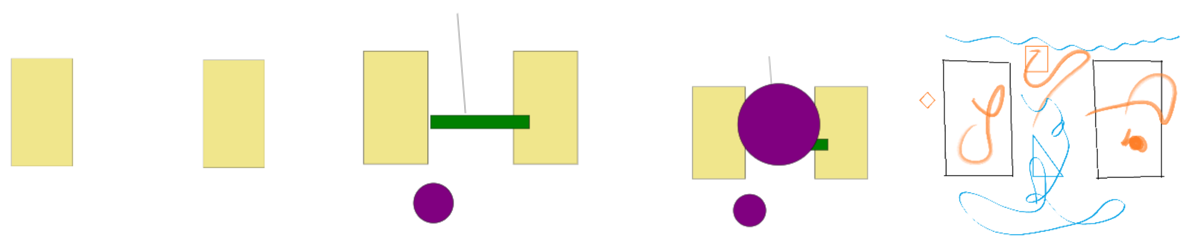


Image_Processing.py

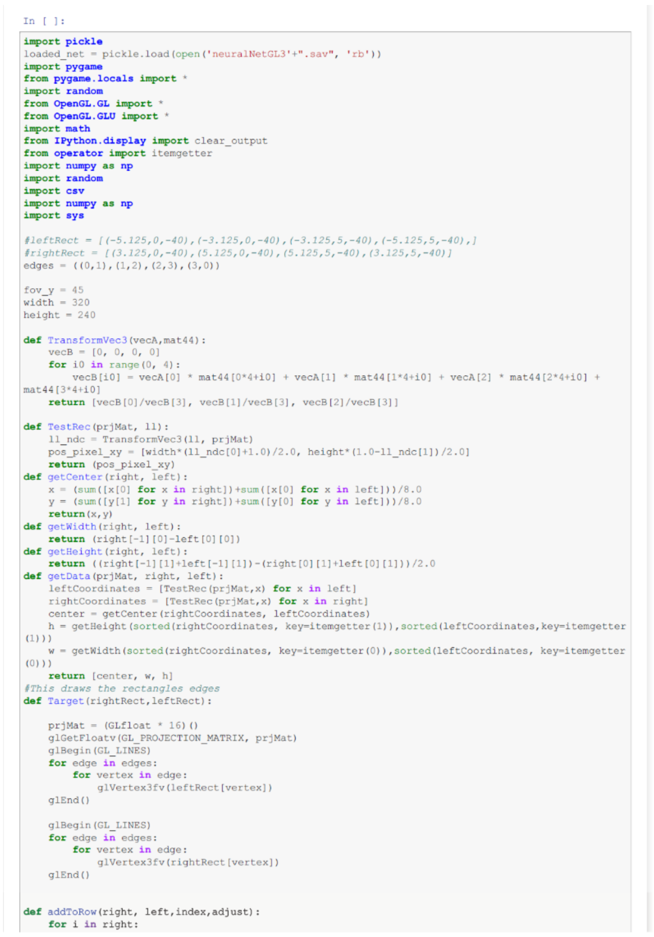
```python
    a,contours,hierarchy = cv2.findContours(image,cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
    rectangles = findRectangles(contours)
    rectangles = sorted(rectangles, key = cv2.contourArea,reverse=True)[:10]

    index = len(rectangles)-1
    while index>=0:
        x,y,w,h = cv2.boundingRect(rectangles[index])
        area= cv2.contourArea(rectangles[index])
        #checks for correct aspect ratio of the rectangles
        if ((abs(float(w)/h - 2.5)>.7) and (abs(float(h)/w- 2.5) >.7)):
            rectangles.pop(index)

        #checks if area is within reasonable range
        elif area< 500 or area>15000:
            rectangles.pop(index)
        index-=1

    return rectangles

def rectangleStats(rectangles):
    x,y,w,h = cv2.boundingRect(rectangles[0])
    x2,y2,w2,h2 = cv2.boundingRect(rectangles[1])
    if x<x2:
        leftRectangle = [x,y,w,h]
        rightRectangle = [x2,y2,w2,h2]
    else:
        leftRectangle = [x2,y2,w2,h2]
        rightRectangle = [x,y,w,h]
    return leftRectangle,rightRectangle


def getCenter(leftRectangle,rightRectangle):
    #x + w + x2 / 2.0 middle of two x coordinates
    centerX = float(leftRectangle[0]+leftRectangle[2]+rightRectangle[0])/2.0
    #2(y)+h +2(y2)+h2 / 4.0  average of the two height centers
    centerY = float((2*leftRectangle[1])+leftRectangle[3]+(2*rightRectangle[1])+rightRectangle[3])/4.0
    return centerX,centerY
###SET THE REST OF YOUR FUNCTIONS AND OPERATIONS ON THE FRAME HERE

#things to try
#Finding contours : foo, contours, hierarchy = cv2.findContours(frame, cv2.RETR_TREE,cv2.CHAIN_APPROX_SIMPLE)
```

39:89 CRLF UTF-8 Python MLImplementation Fetch 0 files

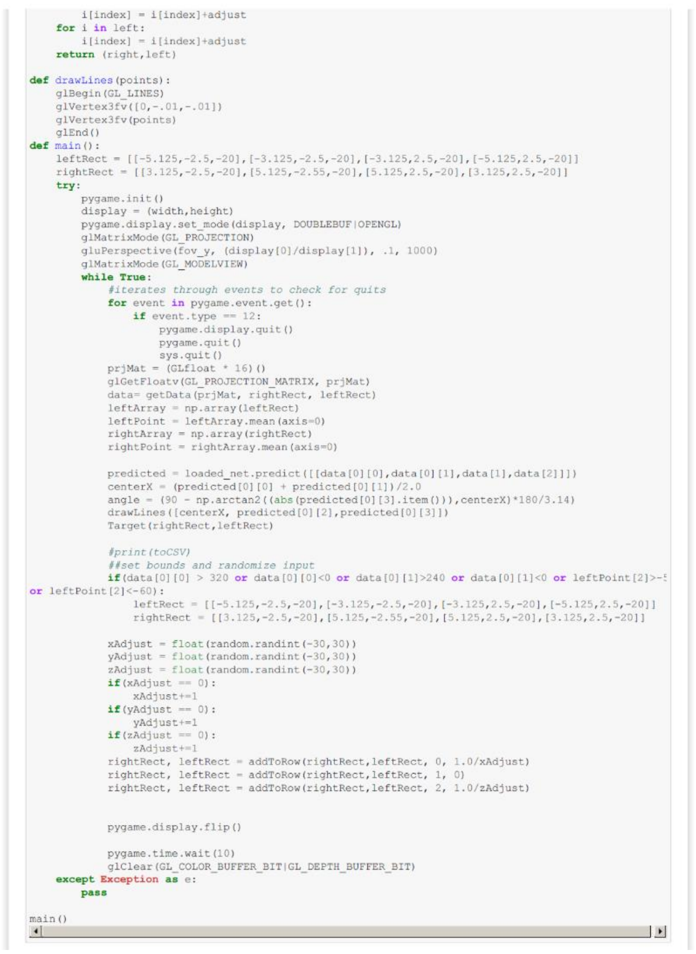
Using OpenCV and its contour detection modules, I found all the shapes that the camera could see. Then, using a series of filters based on shape, dimensions, area, and many other factors, I cut down to the two rectangles that I wanted to detect. To test out my detection and filtering, I drew some images in Microsoft Paint with added error to make sure I was finding the correct rectangles. From the correct rectangles, I found the center point along with size so the robot could figure out distance and direction to travel.



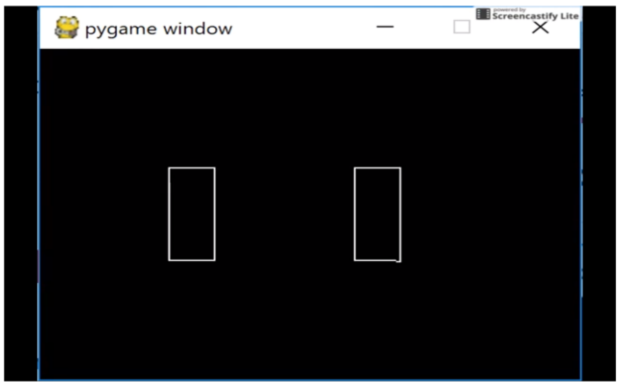
Step 3: 3D Space Simulations

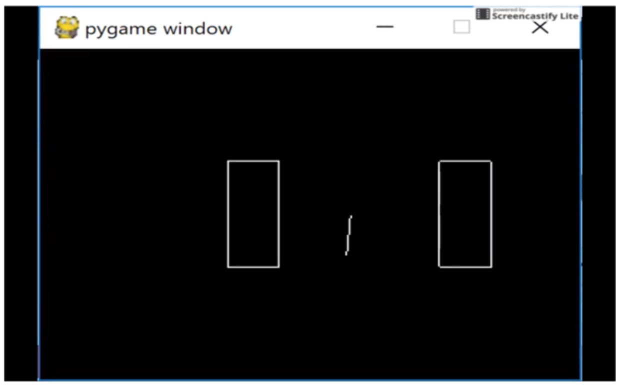


```
In [ ]:
import pickle
loaded_net = pickle.load(open('neuralNetGL3'+".sav", 'rb'))
import pygame
from pygame.locals import *
import random
from OpenGL.GL import *
from OpenGL.GLU import *
import math
from IPython.display import clear_output
from operator import itemgetter
import numpy as np
import random
import csv
import numpy as np
import sys

#leftRect = [(-5.125,0,-40),(-3.125,0,-40),(-3.125,5,-40),(-5.125,5,-40),]
#rightRect = [(3.125,0,-40),(5.125,0,-40),(5.125,5,-40),(3.125,5,-40)]
edges = ((0,1),(1,2),(2,3),(3,0))

fov_y = 45
width = 320
height = 240

def TransformVec3(vecA,mat44):
    vecB = [0, 0, 0, 0]
    for i0 in range(0, 4):
        vecB[i0] = vecA[0] * mat44[0*4+i0] + vecA[1] * mat44[1*4+i0] + vecA[2] * mat44[2*4+i0] +
mat44[3*4+i0]
    return [vecB[0]/vecB[3], vecB[1]/vecB[3], vecB[2]/vecB[3]]

def TestRec(prjMat, ll):
    ll_ndc = TransformVec3(ll, prjMat)
    pos_pixel_xy = [width*(ll_ndc[0]+1.0)/2.0, height*(1.0-ll_ndc[1])/2.0]
    return (pos_pixel_xy)
def getCenter(right, left):
    x = (sum([x[0] for x in right])+sum([x[0] for x in left]))/8.0
    y = (sum([y[1] for y in right])+sum([y[0] for y in left]))/8.0
    return(x,y)
def getWidth(right, left):
    return (right[-1][0]-left[0][0])
def getHeight(right, left):
    return ((right[-1][1]+left[-1][1])-(right[0][1]+left[0][1]))/2.0
def getData(prjMat, right, left):
    leftCoordinates = [TestRec(prjMat,x) for x in left]
    rightCoordinates = [TestRec(prjMat,x) for x in right]
    center = getCenter(rightCoordinates, leftCoordinates)
    h = getHeight(sorted(rightCoordinates, key=itemgetter(1)),sorted(leftCoordinates,key=itemgetter
(1)))
    w = getWidth(sorted(rightCoordinates, key=itemgetter(0)),sorted(leftCoordinates, key=itemgetter
(0)))
    return [center, w, h]
#This draws the rectangles edges
def Target(rightRect,leftRect):

    prjMat = (GLfloat * 16)()
    glGetFloatv(GL_PROJECTION_MATRIX, prjMat)
    glBegin(GL_LINES)
    for edge in edges:
        for vertex in edge:
            glVertex3fv(leftRect[vertex])
    glEnd()

    glBegin(GL_LINES)
    for edge in edges:
        for vertex in edge:
            glVertex3fv(rightRect[vertex])
    glEnd()


def addToRow(right, left,index,adjust):
    for i in right:
```



```
        i[index] = i[index]+adjust
    for i in left:
        i[index] = i[index]+adjust
    return (right,left)

def drawLines(points):
    glBegin(GL_LINES)
    glVertex3fv([0,-.01,-.01])
    glVertex3fv(points)
    glEnd()
def main():
    leftRect = [[-5.125,-2.5,-20],[-3.125,-2.5,-20],[-3.125,2.5,-20],[-5.125,2.5,-20]]
    rightRect = [[3.125,-2.5,-20],[5.125,-2.55,-20],[5.125,2.5,-20],[3.125,2.5,-20]]
    try:
        pygame.init()
        display = (width,height)
        pygame.display.set_mode(display, DOUBLEBUF|OPENGL)
        glMatrixMode(GL_PROJECTION)
        gluPerspective(fov_y, (display[0]/display[1]), .1, 1000)
        glMatrixMode(GL_MODELVIEW)
        while True:
            #iterates through events to check for quits
            for event in pygame.event.get():
                if event.type == 12:
                    pygame.display.quit()
                    pygame.quit()
                    sys.quit()
            prjMat = (GLfloat * 16)()
            glGetFloatv(GL_PROJECTION_MATRIX, prjMat)
            data= getData(prjMat, rightRect, leftRect)
            leftArray = np.array(leftRect)
            leftPoint = leftArray.mean(axis=0)
            rightArray = np.array(rightRect)
            rightPoint = rightArray.mean(axis=0)

            predicted = loaded_net.predict([[data[0][0],data[0][1],data[1],data[2]]])
            centerX = (predicted[0][0] + predicted[0][1])/2.0
            angle = (90 - np.arctan2((abs(predicted[0][3].item())),centerX)*180/3.14)
            drawLines([centerX, predicted[0][2],predicted[0][3]])
            Target(rightRect,leftRect)

            #print(toCSV)
            ##set bounds and randomize input
            if(data[0][0] > 320 or data[0][0]<0 or data[0][1]>240 or data[0][1]<0 or leftPoint[2]>-5
or leftPoint[2]<-60):
                leftRect = [[-5.125,-2.5,-20],[-3.125,-2.5,-20],[-3.125,2.5,-20],[-5.125,2.5,-20]]
                rightRect = [[3.125,-2.5,-20],[5.125,-2.55,-20],[5.125,2.5,-20],[3.125,2.5,-20]]

            xAdjust = float(random.randint(-30,30))
            yAdjust = float(random.randint(-30,30))
            zAdjust = float(random.randint(-30,30))
            if(xAdjust == 0):
                xAdjust+=1
            if(yAdjust == 0):
                yAdjust+=1
            if(zAdjust == 0):
                zAdjust+=1
            rightRect, leftRect = addToRow(rightRect,leftRect, 0, 1.0/xAdjust)
            rightRect, leftRect = addToRow(rightRect,leftRect, 1, 0)
            rightRect, leftRect = addToRow(rightRect,leftRect, 2, 1.0/zAdjust)


            pygame.display.flip()

            pygame.time.wait(10)
            glClear(GL_COLOR_BUFFER_BIT|GL_DEPTH_BUFFER_BIT)
    except Exception as e:
        pass

main()
```

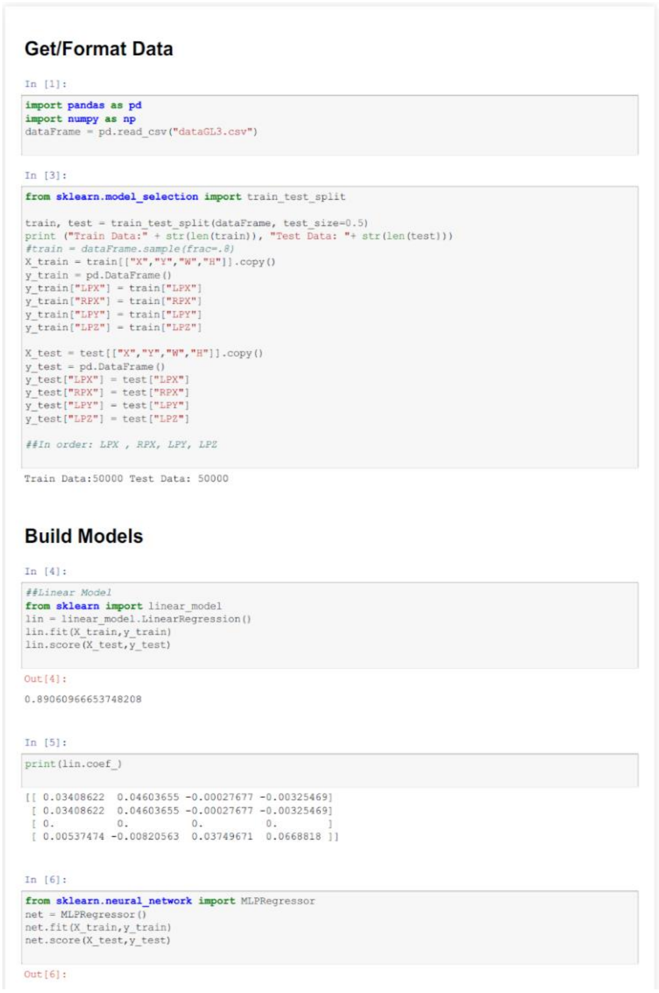




Data Collection (Left Side) .... Projected Path (Right Side)

Using OpenGL, I created a simulation of the rectangles in 3D space. Then, I projected these 3D rectangles onto a 2D screen, simulating the view the robot would have. Using trigonometry, I created a best line of fit between the origin (robot) and the rectangles in the simulation.

Step 4: Training the Neural Network



Get/Format Data

In [1]:

```
import pandas as pd
import numpy as np
dataFrame = pd.read_csv("dataGL3.csv")
```

In [3]:

```
from sklearn.model_selection import train_test_split

train, test = train_test_split(dataFrame, test_size=0.5)
print ("Train Data:" + str(len(train)), "Test Data: "+ str(len(test)))
#train = dataFrame.sample(frac=.8)
X_train = train[["X","Y","W","H"]].copy()
y_train = pd.DataFrame()
y_train["LPX"] = train["LPX"]
y_train["RPX"] = train["RPX"]
y_train["LPY"] = train["LPY"]
y_train["LPZ"] = train["LPZ"]

X_test = test[["X","Y","W","H"]].copy()
y_test = pd.DataFrame()
y_test["LPX"] = test["LPX"]
y_test["RPX"] = test["RPX"]
y_test["LPY"] = test["LPY"]
y_test["LPZ"] = test["LPZ"]

##In order: LPX , RPX, LPY, LPZ
```

```
Train Data:50000 Test Data: 50000
```

Build Models

In [4]:

```
##Linear Model
from sklearn import linear_model
lin = linear_model.LinearRegression()
lin.fit(X_train,y_train)
lin.score(X_test,y_test)
```

Out[4]:

```
0.89060966653748208
```

In [5]:

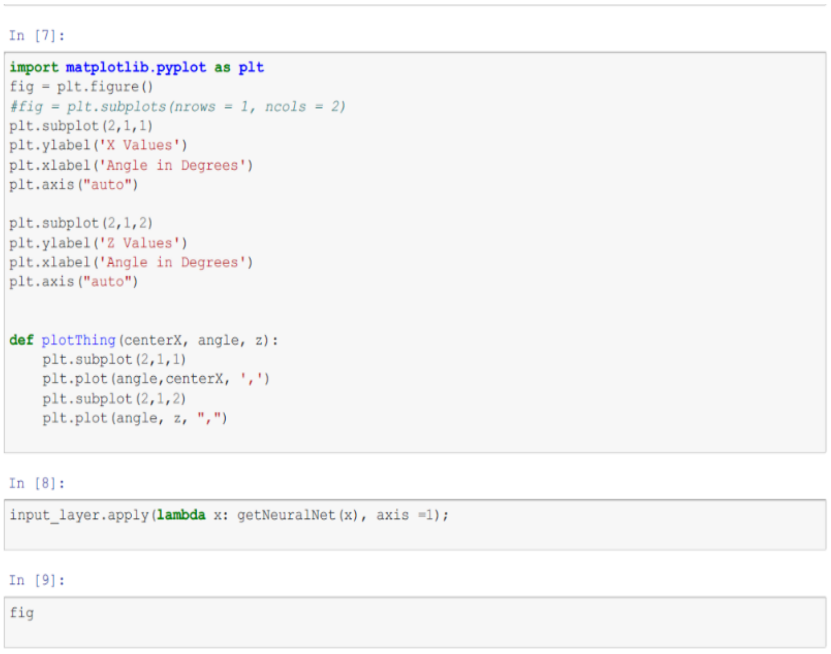
```
print(lin.coef_)
```

```
[[ 0.03408622  0.04603655 -0.00027677 -0.00325469]
 [ 0.03408622  0.04603655 -0.00027677 -0.00325469]
 [ 0.          0.          0.          0.        ]
 [ 0.00537474 -0.00820563  0.03749671  0.0668818 ]]
```

In [6]:

```
from sklearn.neural_network import MLPRegressor
net = MLPRegressor()
net.fit(X_train,y_train)
net.score(X_test,y_test)
```

Out[6]:



In [7]:

```
import matplotlib.pyplot as plt
fig = plt.figure()
#fig = plt.subplots(nrows = 1, ncols = 2)
plt.subplot(2,1,1)
plt.ylabel('X Values')
plt.xlabel('Angle in Degrees')
plt.axis("auto")

plt.subplot(2,1,2)
plt.ylabel('Z Values')
plt.xlabel('Angle in Degrees')
plt.axis("auto")


def plotThing(centerX, angle, z):
    plt.subplot(2,1,1)
    plt.plot(angle,centerX, ',')
    plt.subplot(2,1,2)
    plt.plot(angle, z, ",")
```

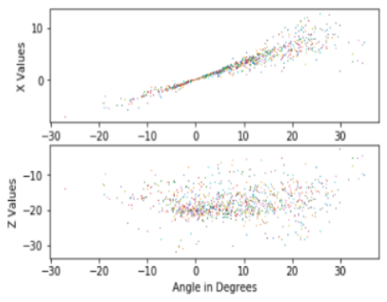
In [8]:

```
input_layer.apply(lambda x: getNeuralNet(x), axis =1);
```

In [9]:

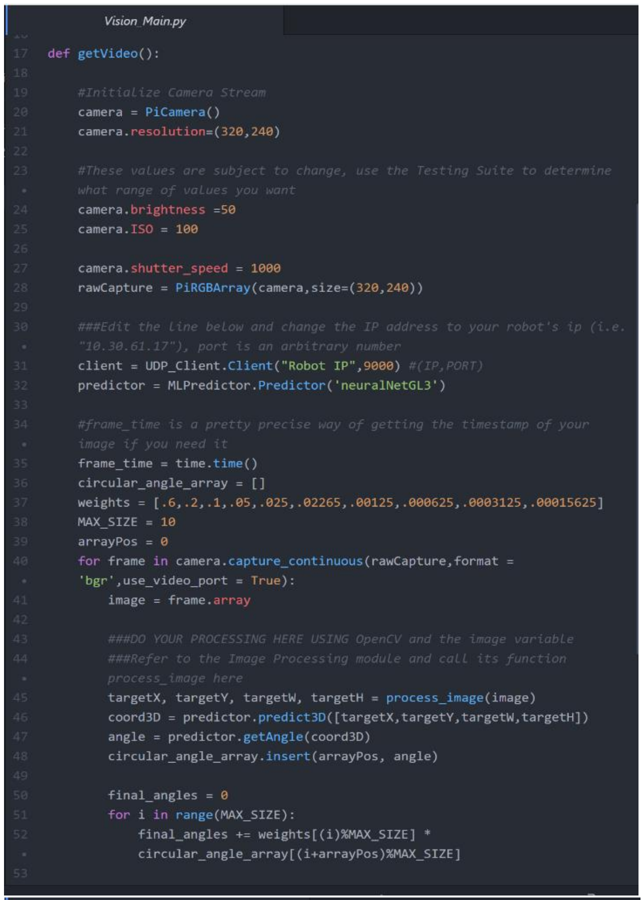
```
fig
```

Out[9]:



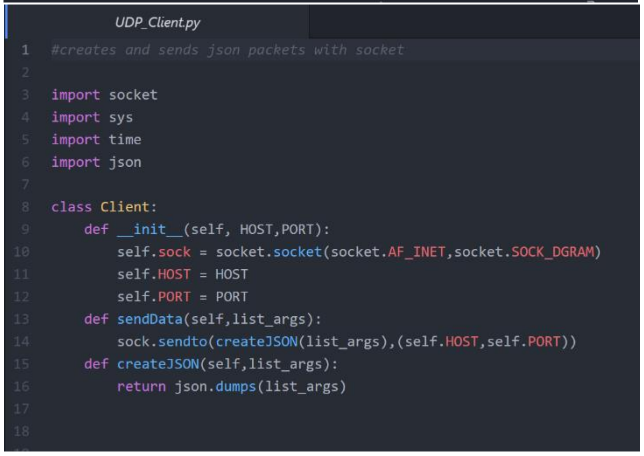
After collecting millions of data samples from the simulation above, I trained a neural network to find the relationships between input (camera view) and output (path). I also tried using a linear regressor, but it yielded a much lower success rate than the neural network. On the right side are graphs that help visual the relationships between the angle (path), x position (left/right) and z position (depth).

Step 5: Integration onto Victoria (Robot)



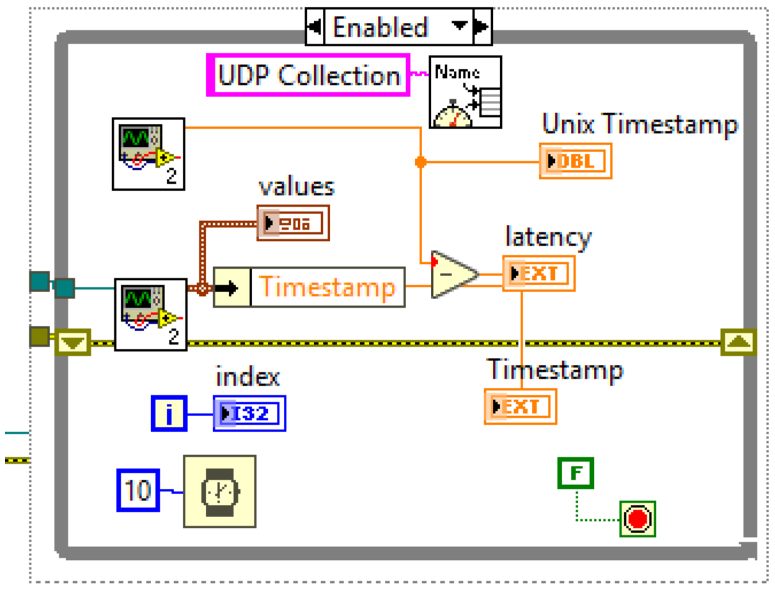
Vision_Main.py

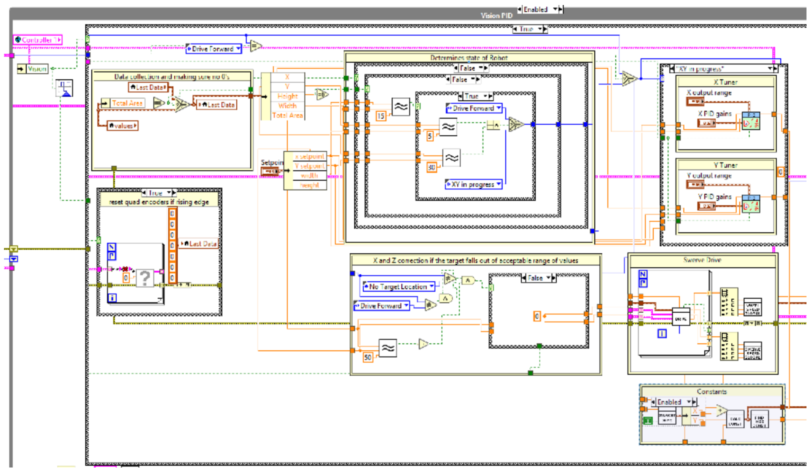
```python
def getVideo():

    #Initialize Camera Stream
    camera = PiCamera()
    camera.resolution=(320,240)

    #These values are subject to change, use the Testing Suite to determine what range of values you want
    camera.brightness =50
    camera.ISO = 100

    camera.shutter_speed = 1000
    rawCapture = PiRGBArray(camera,size=(320,240))

    ###Edit the line below and change the IP address to your robot's ip (i.e. "10.30.61.17"), port is an arbitrary number
    client = UDP_Client.Client("Robot IP",9000) #(IP,PORT)
    predictor = MLPredictor.Predictor('neuralNetGL3')

    #frame_time is a pretty precise way of getting the timestamp of your image if you need it
    frame_time = time.time()
    circular_angle_array = []
    weights = [.6,.2,.1,.05,.025,.02265,.00125,.000625,.0003125,.00015625]
    MAX_SIZE = 10
    arrayPos = 0
    for frame in camera.capture_continuous(rawCapture,format = 'bgr',use_video_port = True):
        image = frame.array

        ###DO YOUR PROCESSING HERE USING OpenCV and the image variable
        ###Refer to the Image Processing module and call its function process_image here
        targetX, targetY, targetW, targetH = process_image(image)
        coord3D = predictor.predict3D([targetX,targetY,targetW,targetH])
        angle = predictor.getAngle(coord3D)
        circular_angle_array.insert(arrayPos, angle)

        final_angles = 0
        for i in range(MAX_SIZE):
            final_angles += weights[(i)%MAX_SIZE] * circular_angle_array[(i+arrayPos)%MAX_SIZE]

```



UDP_Client.py

```python
#creates and sends json packets with socket

import socket
import sys
import time
import json

class Client:
    def __init__(self, HOST,PORT):
        self.sock = socket.socket(socket.AF_INET,socket.SOCK_DGRAM)
        self.HOST = HOST
        self.PORT = PORT
    def sendData(self,list_args):
        sock.sendto(createJSON(list_args),(self.HOST,self.PORT))
    def createJSON(self,list_args):
        return json.dumps(list_args)


```





Collecting images + Sending UDP Packets ……………… Collecting Packets + Controlling Robot Motion

After training my neural network, the next step became integrating my work onto our robot. This involved networking (UDP) between the Raspberry Pi and roboRIO (robot computer). Additionally, I had to build a control system on the roboRIO which could accept angle and distance as parameters and move the robot accordingly.

Check out the successful videos here: https://youtu.be/8ka5wbTj-P8