grand_challenge_submission.main

▼ View Source

```python
import sys
custom_module_path = '/home/pi/ENPM701-assignments/grand_challenge_submission/'
sys.path.append(custom_module_path)
from racer.robot import Robot


if __name__ == "__main__":

    """
    Grand Challenge Competition

    Defines the baron robot and then initiates start
    of competition
    """

    # Initialize robot
    baron = Robot("Speedy Baron")

#     # Start imu sensor reading in parallel thread
#     baron.imu_reader.daemon = True
#     baron.imu_reader.start()

    # Order list of blocks to retrieve
    order = ['red','green','blue','red','green','blue','red','green','blue']

    # Start engine and wait for que
    baron.start(order)

    # Plot the robot trajectory during the challenge
    baron.plotPath()
```

grand_challenge_submission.racer.robot

▼ View Source

```python
# General Python packages
import math
import cv2
import os
import numpy as np
import matplotlib.pyplot as plt
import time
import RPi.GPIO as gpio
import serial
import logging
import threading
import queue

from typing import List
from random import randint

import sys
custom_module_path = '/home/pi/ENPM701-assignments/grand_challenge_submission/'
sys.path.append(custom_module_path)

# This is the main interface script that will control the robot
# as it traverses in the challenge arena.

# Below are the necessary imports for its different features

# For visual perception tasks
from Perception import cntr_bbox

# Imports for sending pictures via email
import smtplib
from smtplib import SMTP
from smtplib import SMTPException
import email
from email.mime.image import MIMEImage
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from datetime import datetime
import imaplib

class Robot(object):
    """
    A class defining basic attributes of Baron robot that
    will be autonomously achieving objective

    Attributes:
        name (str): Name of robot
        _x (float): horizontal position of robot from start postion
        _y (float): vertical position of robot from start position
        _yaw (float): yaw orientation from +x as horizontal [0,180] or (-180,0)
        _travel(float): estimating the distance to drive and reach block
```

```
        _turn (float): keeps the next turn needed by block
        _depth (float): measures distance of obstacle in front of robot from image
        _path (list): stores the path traveled by robot as list of (x,y) points and
        _pwms (list): stores a list of pwms for the robot motors
        _dutycycle (dict): holds the baseline pwm dutycycle for forward,reverse,
                           pivotleft and pivotright
        _success (bool): keeps track whether a given task is successful or not
        _gripper_state (str): stores gripper duty cycle state as opened or closed
        _order(list): List of blocks to pickup
        _end_game (bool): tells competition is over by victory or failure
        _order_no (int): keeps track of number of orders completed
        _block_now (str): the block being pursued now
        _task_now (str): keeps track of the task being completed at the moment
        _actions (dict): holding list of all service actions robot performs with bo
                         on which process is being done at a given time
        _status (dict): a dictionary keeping track of completed actions along with
        _img_frame (img): saves current image frame
        _out (cv2 video obj): gather all frames into a video
        _logger (log): log statements for debuging and status over the terminal
        _kp (float): proportional gain
        _ki (float): integral gain
        _kd (float): derivative gain
        _direction (str): tells weather robot is moving forward or reverse
        _event (event): event object to end the thread call
        _pause_event (event): event object to pause and resume thread
        imu_reader (thread): threading object to read from imu during forward run
        _ser (serial): object reading serial input streams from imu sensor
        _img_saved (img): keeping track of saved image to send in an email
        _survey_loc (List): containes x,y locations for the robot to navigate and s
                            so that to extract toy blocks for mission

    """

    # define the codec and create VideoWriter object
    fourcc = cv2.VideoWriter_fourcc(*'XVID')
    # Gripper States
    grip_state = dict([('closed',2.5),('opened',7.5)])
    # Matrix for Image frame
    image = np.empty((480*640*3,),dtype=np.uint8)
    # define the codec and create VideoWriter object
    fourcc = cv2.VideoWriter_fourcc(*'XVID')
    # Open .txt file to save data
    f = open('robot_path_final.txt','a')

    dutyset = [('f', [70,70]), ('rev', [30,30]),('l', 100),('r',100)]

    duty = dict(dutyset)
    font = cv2.FONT_HERSHEY_COMPLEX_SMALL

    # turn angel
    turn = 15

    # detect radius
    bbox_radius = 0

    # Initialize FL and BR button count
```

```
    counterBR = np.uint64(0)
    counterFL = np.uint64(0)
    buttonBR = int(0)
    buttonFL = int(0)

    # Proportional Control Gain
    Kp = 1
    Ki = 0.05
    Kd = 0.08

    # Important locations in map in units of feet
    origin = [0,0]
    start_pt = [1,1]
    clutter_env = [5,5]
    const_zone = [2.25,6.5]

    # create object to read camera
    video = cv2.VideoCapture(0)

    frame_center = (320,240)
    (cx,cy) = (frame_center[0]/4.5, frame_center[1]/8)

    queue = queue.Queue()

    img_folder = 'challenge_pics'

    counter = 0

    # Environment assessment points
    scan_spots = [(7.5,3),(7.5,8)]

    # Global x, y, positions and yaw from fixed reference
    # frame (0,0)
    global_x = 1
    global_y = 1
    global_yaw = 0

    def __init__(self, name: str):
        """
        Initialize Robot attributes

        Args:
            name: str - takes name of robot

        Returns:
            None
        """

        self.name = name
        self._x = self.start_pt[0]
        self._y = self.start_pt[1]
        self._yaw = 0.0
        self._travel = 0
        self._turn = 0
        self._depth = 0
        self._path = [(1,1,0)]
```

```
        self._pwms = []
        self._dutycycle = self.duty
        self._success = False
        self._gripper_state = self.grip_state['closed']
        self._order = []
        self._end_game = False
        self._order_no = 0
        self._block_now = ''
        self._task_now = ''
        self._actions = dict([('driving',False),('rotating',False),('picking',Fals
        self._status = dict([('started',False),('picked_up',False),('found',False)
        self._img_frame = self.image
        self._out = cv2.VideoWriter('grand_chall_vid.avi', self.fourcc, 3, (640, 4
        self._logger = logging.getLogger(name)
        self._kp = self.Kp
        self._ki = self.Ki
        self._kd = self.Kd
        self._direction = 'f'
        self._event = threading.Event()
        self._pause_event = threading.Event()
        self.imu_reader = threading.Thread(target = self._imu_serial_straight, arg
        self._ser = serial.Serial('/dev/ttyUSB0', 19200)
        self._img_saved = self.image
        self._survey_loc = self.scan_spots

        # Set up logger with Debug initially
        self._logger.setLevel(logging.DEBUG)
        # Create console handler and set level to debug
        console_handler = logging.StreamHandler()
        console_handler.setLevel(logging.DEBUG)
        # log terminal messages to a file
        file_handler = logging.FileHandler('gc_log_3.log')
        file_handler.setLevel(logging.INFO)
        # Create formatter
        formatter = logging.Formatter('%(name)s - %(levelname)s - %(message)s')
        # Add formatter to handler
        console_handler.setFormatter(formatter)
        file_handler.setFormatter(formatter)
        # Add console handler to logger
        self._logger.addHandler(console_handler)
        self._logger.addHandler(file_handler)
        # Initialize gpios and pwms
        self._init()


    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        self._name = name

    def _init(self):
        """
        Initialize gpio pins and set pwms to command DC motors, encoders and
```

```
        servo

        Args:
            None

        Returns:
            None
        """

        gpio.cleanup()
        gpio.setmode(gpio.BOARD)

        # Setup GPIO pin(s)
        gpio.setup(36, gpio.OUT) # Servo

        gpio.setup(31, gpio.OUT) # IN1
        gpio.setup(33, gpio.OUT) # IN2
        gpio.setup(35, gpio.OUT) # IN3
        gpio.setup(37, gpio.OUT) # IN4

        gpio.setup(7, gpio.IN, pull_up_down = gpio.PUD_UP)
        gpio.setup(12, gpio.IN, pull_up_down = gpio.PUD_UP)

        self._pwms.clear()

        # initialize pwm signal to control motor
        pwm01 = gpio.PWM(31, 50)  # BackLeft motor
        pwm11 = gpio.PWM(33, 50) # FrontLeft motor
        pwm22 = gpio.PWM(35, 50) # FrontRight motor
        pwm02 = gpio.PWM(37, 50)  # BackRight motor
        pwmS = gpio.PWM(36, 50) # Servo

        self._pwms = [pwm01,pwm11,pwm22,pwm02,pwmS]

        for pwm in self._pwms:
            pwm.start(0)

    def _pwmZero(self):
        """
        Stops motors by zeroing pwm values in all pins
        controlling DC motors

        Args:
            None

        Returns:
            None
        """


        self._pwms[0].ChangeDutyCycle(0)
        self._pwms[1].ChangeDutyCycle(0)
        self._pwms[2].ChangeDutyCycle(0)
        self._pwms[3].ChangeDutyCycle(0)

    def _gameover(self):
```

```
        """
        Terminates run by stopping pwms and cleaning up
        gpio pins

        Args:
            None

        Returns:
            None
        """

        self._pwmZero()
        self._pwms[-1].ChangeDutyCycle(self.grip_state['closed'])
        time.sleep(1)
        for pwm in self._pwms:
            pwm.stop()
        gpio.cleanup()

    def _forward(self,vals):
        """
        Commands DC motors to drive robot forward by sending
        Dutycycles to respective pwm pins

        Args:
            None

        Returns:
            None
        """
        # Left wheels
        self._pwms[0].ChangeDutyCycle(vals[0])
        self._pwms[1].ChangeDutyCycle(0)
        # Right wheels
        self._pwms[2].ChangeDutyCycle(0)
        self._pwms[3].ChangeDutyCycle(vals[1])

    def _reverse(self,vals):
        """
        Commands DC motors to drive robot in reverses by sending
        Dutycycles to respective pwm pins

        Args:
            None

        Returns:
            None
        """
        # Left wheels
        self._pwms[0].ChangeDutyCycle(0)
        self._pwms[1].ChangeDutyCycle(vals[0])
        # Right wheels
        self._pwms[2].ChangeDutyCycle(vals[1])
        self._pwms[3].ChangeDutyCycle(0)

    def _pivotleft(self,vals):
        """
```

```
        Commands DC motors to pivot robot left by sending
        Dutycycles to respective pwm pins

        Args:
            None

        Returns:
            None
        """
        # Left wheels
        self._pwms[0].ChangeDutyCycle(0)
        self._pwms[1].ChangeDutyCycle(vals)
        # Right wheels
        self._pwms[2].ChangeDutyCycle(0)
        self._pwms[3].ChangeDutyCycle(vals)

    def _pivotright(self,vals):
        """
        Commands DC motors to pivot robot right by sending
        Dutycycles to respective pwm pins

        Args:
            None

        Returns:
            None
        """

        # Left wheels
        self._pwms[0].ChangeDutyCycle(vals)
        self._pwms[1].ChangeDutyCycle(0)
        # Right wheels
        self._pwms[2].ChangeDutyCycle(vals)
        self._pwms[3].ChangeDutyCycle(0)

    def _servo_cntrl(self,duty_cycle):
        """
        Commands servo motors to open and close gripper by sending
        Dutycycles to respective pwm pins

        Args:
            None

        Returns:
            None
        """

        self._pwms[-1].ChangeDutyCycle(duty_cycle)
        time.sleep(1)

        img = cntr_bbox.servo_img(duty_cycle)
        self._img_frame = cntr_bbox.dist_img(img, self._distance())

        #cntr_bbox.img_show("Servo status",self._img_frame)
        self._out.write(img)

```

```
    def _picam_frame(self):
        """
        Calls picamera object to take image frame from the pi camera
        and applies corrections by flipping to reflect reality

        Args:
            None

        Returns:
            None
        """
        # Take picture with camera
        cntr_bbox.camera.capture(self.image, format="bgr")
        image = self.image.reshape((480,640,3))

        self._img_frame = cv2.flip(image,1)

    def _mask_color(self, imageHSV):

        if self._block_now == 'green':
            # Trail Green Bock - LAB
            minHSV = np.array([46,88,121])
            maxHSV = np.array([65,141,255])
        elif self._block_now == 'red':
            # Trail Red block - LAB
            minHSV = np.array([151,107,147])
            maxHSV = np.array([255,255,255])
        elif self._block_now == 'blue':
            # Trail Blue block - LAB
            minHSV = np.array([72,94,97])
            maxHSV = np.array([126,188,212])

        maskHSV = cv2.inRange(imageHSV, minHSV, maxHSV)

        return maskHSV

    def _prep_image(self):
        """
        Processes images by applying HSV masking and gaussian blur
        for use by latter object detection functions

        Args:
            None

        Returns:
            None
        """

         # Convert image from BGR to HSV space
        imageHSV = cv2.cvtColor(self._img_frame,cv2.COLOR_BGR2HSV)

        # mask the green light from HSV and convert to grayscale
        mask = self._mask_color(imageHSV)

        # Mask HSV masked image of arrow
        blurred = cv2.GaussianBlur(mask,(11,11), 0)
```

```python

        return blurred

    def _grip_checker(self):
        """
        Processes images by applying a rectangular mask over desired
        region of interest to verify whether gripper has held toy block

        Args:
            None

        Returns:
            self._image_frame (img): Masked image
        """
        self._logger.info("Inside Grip Checker Function")
        # Create a mask of gripped block in image
        # Save the shape of the image array
        (height, width, c)  = self._img_frame.shape

        # Create mask of image just like picture
        mask = np.zeros_like(self._img_frame[:,:,0])

        # Create a white rectangle
        x,y,w,h = round(width/2)-130, round(height/2)-30, 250, 300
        cv2.rectangle(mask,  (x,y), (x+w, y+h), 255, -1)

        # Extract the image that aligns in the rectangle
        self._img_frame = cv2.bitwise_and(self._img_frame, self._img_frame, mask=m

        return self._img_frame

    def _wait2start(self):
        """
        Waits for the competition start QR code to be visible,
        and reads QR code in, which if corresponds to "ENPM701"
        returns a True boolean to begin competition

        Args:
            None

        Returns:
            start (bool): True/False based on detection of expected QR
        """
        # Define detector
        detector = cv2.QRCodeDetector()

        # Check if program start initiated
        start = False

        cnt = 0

        self._logger.info("Waiting for start cue . . . ")

        try:

            for frame in cntr_bbox.camera.capture_continuous(cntr_bbox.rawCapture,
```

```python

                # grab the current frame
                img = frame.array

                img = cv2.flip(img,1)

                data, bbox, _ = detector.detectAndDecode(img)

                if(bbox is not None):
                    for i in range(len(bbox)):
                        cv2.line(img, tuple(bbox[i][0]),tuple(bbox[(i+1)%len(bbox)
                        cv2.putText(img, data, (int(bbox[0][0][0]),int(bbox[0][0][

                if data:
                    self._logger.info(f"Data: {data}")
                    pass

                # Show resutls to the screen
                cv2.imshow("QR Code detector",img)
                key = cv2.waitKey(1) & 0xFF

                # write frame into file
                self._out.write(img)

                # clear the stream in preparation for the next frame
                cntr_bbox.rawCapture.truncate(0)

                # Break out of loop by pressing the q key
                if(key == ord("q")):
                    self._logger.warn("Program Terminated")
                    self.video.release()
                    cv2.destroyAllWindows()
                    break

                if data == "ENPM701":
                    self._logger.info("Cue Received \n")
                    self._logger.info('@' * 45)
                    self._logger.info("Starting Grand Challenge! \n")
                    self._logger.info('#' * 45)
                    start = True
                    self.video.release()
                    cv2.destroyAllWindows()
                    time.sleep(3)
                    break

        except Exception as error:
            self._logger.error(error)
            self.video.release()
            cv2.destroyAllWindows()

        return start

    def _email_media(self):

        # Take a picture of gripped block to email
        self._picam_frame()
```

```
        # Append status of servo on image
        self._img_frame = cntr_bbox.servo_img(self._gripper_state)
        self._out.write(self._img_frame)
        # Save to designated directory
        cv2.imwrite(f'{self.img_folder}/{self._block_now}_{self.counter}.png', sel
        #time.sleep(1)
        # Read saved image
        img_name = f'{self.img_folder}/{self._block_now}_{self.counter}.png'

        # send email to user with images
        smtpUser = 'ykebede2@terpmail.umd.edu'
        smtpPass = 'QwE@$d1219'

        toAdd = ['ENPM809TS19@gmail.com','yosephcollege@gmail.com']
        fromAdd = smtpUser

        f_time = datetime.now().strftime('%a %d %b @ %H:%M')
        subject = f'ENPM701-GrandChallenge-{f_time}-yosephK-{self._block_now}-{sel

        self.counter += 1

        msg = MIMEMultipart()
        msg['Subject'] = subject
        msg['From'] = fromAdd
        msg['To'] = ",".join(toAdd)

        msg.preamble = "Image @ " + f_time

        body = email.mime.text.MIMEText("Baron Robot image: " + f_time)
        msg.attach(body)

        fp = open(img_name, 'rb')
        img = MIMEImage(fp.read())
        fp.close()
        msg.attach(img)

        s = smtplib.SMTP('smtp.gmail.com', 587)
        s.ehlo()
        s.starttls()
        s.ehlo()
        s.login(smtpUser, smtpPass)
        s.sendmail(fromAdd, toAdd, msg.as_string())
        s.quit()

        self._logger.info("Email delivered!")

    def _meter2encoder(self,x_dist):
        """
        Reads in distance values in meters and converts to
        encoder counts

        Args:
            None

        Returns:
            encod (int): converted encoder counts from distance in m
```

```
        """
        encod = round(float(x_dist / (2*math.pi*0.0325))*960)

        return encod

    def _encoder2meter(self,encd):
        """
        Reads in encoder values and converts them to meters

        Args:
            None

        Returns:
            cms (float): converted encoder counts to distance in m
        """
        dist = round(float((encd / 960) * 2*math.pi*0.0325),4)

        return dist

    def _cm2encoder(self,x_dist):
        """
        Reads in distance values in centimeters and converts to
        encoder counts

        Args:
            None

        Returns:
            encod (int): converted encoder counts from distance in cms
        """
        dist = x_dist/100
        encod = self._meter2encoder(dist)
        return encod

    def _encoder2cm(self,encd):
        """
        Reads in encoder values and converts them to centimeters

        Args:
            None

        Returns:
            cms (float): converted encoder counts to distance in cms
        """
        dist = self._encoder2meter(encd)
        cms = round(dist*100,4)

        return cms

    def _feet2encoder(self,x_dist):
        """
        Reads in distance values in feet and converts to
        encoder counts

        Args:
            None
```

```python

        Returns:
            encod (int): converted encoder counts from distance in feet
        """
        dist = round(float(x_dist * 0.3048),4)
        encod = self._meter2encoder(dist)
        return encod

    def _encoder2feet(self,x_encod):
        """
        Reads in encoder values and converts to
        feet

        Args:
            None

        Returns:
            feet (float): converted encoder counts from distance in feet
        """
        meter = self._encoder2meter(x_encod)
        feet = round(meter * 3.28084,4)

        return feet

    def _feet2cm(self,x_dist):
        """
        Reads in distance values in feet and converts to
        encoder counts

        Args:
            None

        Returns:
            encod (int): converted encoder counts from distance in feet
        """
        dist = round(float(x_dist * 12 * 2.54),3)

        return dist

    def _cm2feet(self,x_dist):
        """
        Reads in distance values in feet and converts to
        encoder counts

        Args:
            None

        Returns:
            encod (int): converted encoder counts from distance in feet
        """
        dist = round(float(x_dist / (12 * 2.54)),2)

        return dist

    def _imu_serial(self):
        """
```

```
        Reads in serial robot's yaw angle value from imu sensor, cleans buffer,
        and converts it into a float data type. Convers reading from 0-360 to
        -180 < yaw < 180.

        Args:
            None

        Returns:
            self._yaw (float): robots yaw in degrees
        """
        self._ser.reset_input_buffer()
        self._ser.flush()

        while True:

            try:
                # Read for imu from serial
                if(self._ser.in_waiting > 0):
        #                 if cnt >5:
                    # Strip serial stream of extra characters
                    line = self._ser.readline()

                    line = line.rstrip().lstrip()

                    line = str(line)
                    line = line.strip("'")
                    line = line.strip("b'")

                    # Return float
                    line = float(line)

                    if (line > 180 and line <=360):
                        line = line - 360

                    self._yaw = line
                    return self._yaw

            except Exception as error:
                self._logger.error(f"Imu Error: {error}")

    def _imu_serial_straight(self,queue):
        """
        Reads in serial robot's yaw angle value from imu sensor, cleans buffer,
        and converts it into a float data type. Convers reading from 0-360 to
        -180 < yaw < 180.

        Only difference from self._imu_serial() is that this is run in a separate
        thread which is controlled by events to start, pause, and stop function ca

        Args:
            None

        Returns:
            self._yaw (float): robots yaw in degrees
        """

```

```
        while not self._event.is_set():

            self._ser.reset_input_buffer()
            self._ser.flush()

            try:
                while self._pause_event.is_set():
                    # Read for imu from serial
                    if(self._ser.in_waiting > 0):
                            line = self._ser.readline()

                            line = line.rstrip().lstrip()

                            line = str(line)
                            line = line.strip("'")
                            line = line.strip("b'")

                            # Return float
                            line = float(line)

                            if (line > 180 and line <=360):
                                line = line - 360

                            queue.put(line)

            except Exception as error:
                self._logger.warning(f"Imu Error inside thread: {error}")

    def _distance(self):
        """
        Reads in serial robot's distance from an obstacle using ultrasonic sensor l
        sending a trigger pulse and listening the returning echo to compuse the se
        distance from nearby object using triangulation technique. Then it saves c
        distance in self._depth

        Args:
            None

        Returns:
            None
        """
        pulse_start = 0
        pulse_end = 0

        # Define pin allocations
        trig = 16
        echo = 18
        # Setup GPIO board & pins
        gpio.setup(trig, gpio.OUT)
        gpio.setup(echo, gpio.IN)

        # Ensure output has no value
        gpio.output(trig, False)
        time.sleep(0.01)

        # Generate trigger pulse
```

```
        gpio.output(trig, True)
        time.sleep(0.00001)
        gpio.output(trig, False)

        # Generate echo time signal
        while gpio.input(echo) == 0:
            pulse_start = time.time()

        while gpio.input(echo) == 1:
            pulse_end = time.time()

        pulse_duration = pulse_end - pulse_start

        # Convert time to distance
        distance = pulse_duration * 17150
        self._depth = round(distance, 2)

        return self._depth

    def _rotate(self):
        """
        Reads in direction of rotation requested, and starts
        rotating the motors accordingly to achieve desired orientation.


        Args:
            None

        Returns:
            None
        """

        if self._direction == 'r':
            self._pivotright(60)
        elif self._direction == 'l':
            self._pivotleft(60)

    def _dist_estimate(self):
        """
        Reads in radius of bounding box from bbox detector function, and
        plugs radius into precomputed correlation function to estimate the
        depth of toy block from robot.

        Assigns results to self._travel and saves image showing value of distance
        appended to it in self._img_frame and writes frame to video object self._ou

        Args:
            None

        Returns:
            None
        """
        # Take image with picamera
        self._picam_frame()
        self._img_frame, __, bbox_radius, self._success = cntr_bbox.bbox_detect(se
```

```
        if self._success not in [None, False]:
            # Plug in pixel size into depth equation
            distance = (0.0099 * bbox_radius**2) - (1.8846*bbox_radius) + 103.47
            self._travel = round(distance,2)
            # Append image of estimated distance on image frame
            self._img_frame = cntr_bbox.dist_img(self._img_frame,self._travel)
            # save image into video object
            self._out.write(self._img_frame)
    def _corner_detect(self,img,origImg):

        # Create a list to store the x,y location of points
        pts_loc = []

        # Detect corners from image
        corners = cv2.goodFeaturesToTrack(img,5,0.01,10)

        if corners is not None:

            corners = np.int0(corners)

            # identify location of corners in image
            for i in corners:
                # Extract x,y coordinate of points
                x,y = i.ravel()

                pts_loc.append([x,y])

                # Draw circle ontop of original image
                cv2.circle(origImg, (x,y), 3, (0,0,255),-1)

            # Create a column vector from pts list
            pts_loc = np.array(pts_loc)

            return img, pts_loc, origImg, True
        else:
            return img, pts_loc, origImg, None

    def _center_det(self,pt_list, center):

        # Extract x,y points from pt_list
        x = pt_list[:,0]
        y = pt_list[:,1]

        # Determine the min and max width & height values
        # of the points, as if to drow rectangle around arrow
        x_min = x.min()
        y_min = y.min()

        x_max = x.max()
        y_max = y.max()

        # Store height of bounding box
        vert_dst = y_max - y_min

        # Store width of bounding box
        horz_dst = x_max - x_min
```

```
        # Compute and store half dimensions of
        # box, will come later when determining
        # arrow direction
        y_half = vert_dst/2 + y_min
        x_half = horz_dst/2 + x_min

        # Store center of the block
        #center = [int(round(x_half)), int(round(y_half))]
        center.append(int(round(x_half)))
        center.append(int(round(y_half)))
        # Estimate radius
        self.bbox_radius = int(round(math.sqrt((x_max - x_half)**2 + (y_max - y_ha

        return center, self.bbox_radius

    def _bbox_detect(self):
        """
        Reads in an image, and then applies preprocessing functions that mask and
        blur image. Then, attempts to detect corners of an object in frame, and if
        successful it will run further computation to determine the center and rad
        of detected object. Finally adds the draws the bounding circle and center p
        as well as write computed values on original image

        Assigns results to self._travel and saves image showing value of distance
        appended to it in self._img_frame and writes frame to video object self._ou

        Args:
            image (image): Original colored image taken in from camera

        Returns:
            bbox_radius (float): Radius of bounding circle detecting object
            len(pts_loc) (int): The number of corner points identified on object fr
        """
        # Initialize list to save corner points and bbox radius
        bbox_radius = 0
        pts_loc = []
        center = []

        # Take image using camera frame
        self._picam_frame()

        # Prepare image by masking and blurring
        image_blurred = self._prep_image()

        # Extract bbox radius from image
        self._img_frame, center, bbox_radius, self._success = cntr_bbox.bbox_detec

        return self._img_frame, center, bbox_radius, self._success

#         # Detect corners from image
#         img_crnr, pts_loc, image, var = self._corner_detect(image_blurred, self._
#
#         if var is not None:
#
#             # Check if corners are detected
```

```
#             if len(pts_loc) > 3:
#                 ## Draw contours over an image, if available
#                 center, bbox_radius = self._center_det(pts_loc, center)
#
#                 block_coordinate = "(" + str(center[0]) + "," + str(center[1]) +
#
#                 # Draw circle ontop of original image
#                 cv2.circle(image, tuple(center), bbox_radius, (0,255,255),2)
#                 cv2.circle(image, tuple(center), 0,(0,0,255),5)
#                 cv2.putText(image,block_coordinate,(0,int(self.cy)),self.font,2,
#
#                 # Draw a cross at center of frame
#                 cv2.line(image,(self.frame_center[0]-100, self.frame_center[1]),
#                 cv2.line(image,(self.frame_center[0], self.frame_center[1]-100),
#
#                 success = True
#
#                 self._logger.info(f"{self._block_now} block is detected")
#                 time.sleep(1)
#
#                 return image, center, bbox_radius, success
#
#             else:
#                 self._logger.info("Not clearly detected")
#                 success = False
#
#                 return image, center, bbox_radius, success
#
#         else:
#             print("No object of interest in scene")
#             return image, 0, 0, success

        self._logger.info(f"Radius {bbox_radius}")
        self._logger.info(f"Number of corners/edges detected are {len(pts_loc)}")

        #return bbox_radius, center #, len(pts_loc),

    def _pivot(self,deg_diff,just_turn=False,steady = False):
        """
        This function pivots robot to deseired angle

        Args:
            deg_diff (float): Amount of angular rotation to turn.
                      Default value is self._turn parameter

        Returns:
            None

        """
        self._logger.info("Inside pivot function")
        self._task_now = 'rotating'
        self._actions[self._task_now] = True

        # To know the direction of turn. Notation based on camera object detection
        # notation
#         thresh = 0.8
```

```
        if deg_diff >= 0:
            self._direction = 'r'
        else:
            self._direction = 'l'
#             thresh = -0.8

        # Initialize change in time for derivative control
        zones = [0.4, 0.8, 1]

        self._yaw = self._imu_serial()
        yaw_start = self._yaw

        # Angle diff initialized
        yaw_diff0 = deg_diff + 0.001

        yaw_final = yaw_start + deg_diff
        yaw_diff = yaw_final - yaw_start
        yaw_diff_old = yaw_diff

        self._logger.info(f"Needed final yaw is {yaw_final} degrees")
        yaw_current = yaw_start
        # Lower and upper bounds of dutycycle initialized
        lower_dc = 70
        upper_dc = 90
        # Divide the distance traveled into three zones
        trip = [x * yaw_diff for x in zones]
        # Variables for PID control
        derivative = 0
        integral = 0

        start = time.time()
        yaw_diff_list = [0,0]
        yaw_diff_list.append(yaw_diff)
        window_size = 5
        while True:
            try:

                #prcnt = abs(self._imu_serial() - yaw_start)/deg_diff
                prcnt = 1 - yaw_diff/yaw_diff0

                # Adopting moving average
#                 cumsum = np.cumsum(yaw_diff_list)
#                 cumsum[window_size:] = cumsum[window_size:] - cumsum[:-window_si
#                 moving_avg = cumsum[window_size - 1:] / window_size

                if prcnt >= zones[0] and prcnt <= zones[1]:
                    self._dutycycle[self._direction] = self.duty[self._direction]
                elif prcnt > zones[1]:
                    self._dutycycle[self._direction] = self.duty[self._direction]

                # Save robots pose by saving its x, yaw position
                self._path.append((round(self.global_x,2),round(self.global_y,2),r
                self._dutycycle[self._direction] = np.clip(self._dutycycle[self._d

                if abs(yaw_diff) <= 2:
                    self._dutycycle[self._direction] = 65
```

```

                elif abs(yaw_diff) >= 100:
                    self._dutycycle[self._direction] = 100



                if abs(yaw_diff) >= 0.8:

                    if ((np.sign(yaw_diff) == np.sign(yaw_diff_old)) or (abs(yaw_d
#                     if (abs(yaw_diff - np.mean(yaw_diff_list[-3:-1])) < 5):
#                     if abs(yaw_diff) >= 0.5:
                        self._logger.info(f"Current yaw {self._imu_serial()}; Angl

                        if not steady:
                            if self._direction == 'r':
                                self._pivotright(self._dutycycle[self._direction])
                            elif self._direction == 'l':
                                self._pivotleft(self._dutycycle[self._direction])
                        else:
                            if self._direction == 'r':
                                self._pivotright(70)
                            elif self._direction == 'l':
                                self._pivotleft(70)

                else:
                    self._pwmZero()
                    total_turn = yaw_start - self._imu_serial()#self.queue.get()
                    self._logger.info(f"Current yaw {self._imu_serial()}, Total tu
                    self._logger.info("Robot turn done")

                    yaw_diff = round(yaw_diff,2)

                    if not just_turn:
                        if steady:
                        #if (abs(yaw_diff) >= 0.):
                        # Check for overshooting

                            deg_diff = self._scan4object()

                            if deg_diff is not None:

                                if abs(deg_diff) > 0.5:

        #                             time.sleep(1)
                                    # Return pwm dutycycle back to base
                                    self._dutycycle[self._direction] = 70

                                    self._pivot(deg_diff,steady=True)

#                         if abs(yaw_diff) >= 0.6:
#                             self._dutycycle[self._direction] = 70
#
#                             self._pivot(yaw_diff,steady=True)


#                     elif abs(yaw_diff) > 0.6:
```

```
#
#                         # Return pwm dutcycle back to base
#                         self._dutycycle[self._direction] = 70
#
#                         self._pivot(yaw_diff)

                    # Return pwm dutcycle back to base
                    self._dutycycle[self._direction] = self.duty[self._direction]

                    # Save robots pose by saving its x, yaw position
                    self._path.append((round(self.global_x,2),round(self.global_y,

                    self._logger.info("Final orientation reached")
                    break

                yaw_diff_old = yaw_diff
                self._yaw = self._imu_serial()
                yaw_diff = yaw_final - self._yaw
                yaw_diff_list.append(yaw_diff)

#                 if ((np.sign(yaw_diff) != np.sign(yaw_diff_old)) and (abs(yaw_di
#                     yaw_diff_list.pop()
#                     yaw_diff = yaw_diff_old
                # Filtering data via median thresholding
#                 if len(yaw_diff_list) > 6:
#                     median = np.median(yaw_diff_list[-5:-1])
#                 else:
#                     median = np.median(yaw_diff_list[-3:-1])

#                 if (abs(yaw_diff - median) < 5):
#                     yaw_diff_list.append(yaw_diff)
#                 else:
#                     yaw_diff_list.pop()
#                     yaw_diff = yaw_diff_old
#                     yaw_diff_list.append(yaw_diff_old)

            except KeyboardInterrupt:
                self._logger.error("Pivot Interrupted")
                self._gameover()
            except Exception as error:
                self._logger.error(f"Error {error}")

        self._pwmZero()
        self._actions[self._task_now] = False
        self._logger.info("Pivot Finished")

    def _go2block(self, direction='f', eucl_disp=0, just_drive=False, steady=False
        """
        This function drives the robot in a straight path, either forward
        or reverse

        Args:
            direction: Direction of drive, forward or reverse

        Returns:
            None
```

```
        """
        self._task_now = 'driving'
        self._actions[self._task_now] = True
        self._logger.info("Going toward block (go2block)")

        # Divide the separation distance into three zones
        # First zone is 70% into drive
        # Second zone is 20%
        # Third zone is the last 10%
        # Each of the zones have their own dutycycle. Robot will
        # drive and stop at each zone, scan, align, and resume drive
        zones = [0.7, 0.85, 0.9]
        thresh = 20
        self._success = False
        # Check if straight drive mode is requested andd steady drive mode
        if not just_drive and not steady:
            # Use image from camera to estimate distance
            self._dist_estimate()
            # convert depth to encoder count
            orig_dist = self._cm2encoder(self._travel-35)

            self._logger.info(f"Distance estimated {self._travel}")
            self._logger.info(f"Distance to drive {self._travel - 35}")
        elif steady:
            # Use image from camera to estimate distance
            self._dist_estimate()
            # convert depth to encoder count
            orig_dist = self._cm2encoder(self._travel-5)
            self._logger.info(f"Distance estimated {self._travel}")
            self._logger.info(f"Driving slowly {self._travel-5} cms")
        elif just_drive:
            # Drive straight command
            orig_dist = self._feet2encoder(eucl_disp)
            self._logger.info(f"Just Driving Distance estimated {orig_dist}")
        elif just_drive and steady:
            # Drive straight command
            orig_dist = self._cm2encoder(eucl_disp)
            self._logger.info(f"Just Driving Distance estimated {orig_dist}")

        self._logger.info(f"Distance to drive {orig_dist} encoder")
        pos_encoder_orig = 0
        pos_encoder = 0
        orig_error = orig_dist

        self.counterFL = np.uint64(0)
        self.counterBR = np.uint64(0)
        self.buttonBR = int(0)
        self.buttonFL = int(0)

        # Defining the limit of dutycycle
        lower_dc = 70
        upper_dc = 100

        z1 = 80
        z2 = z1 - 20
```

```python
        z3 = z2 - 10

        # Divide up the path in different zones
        trip = [x * orig_dist for x in zones]

        # Determine if function completed successfully
        self._success = False

        #time.sleep(1)
        yaw0 = self._imu_serial()

        # For encoder level control
        integral_error = 0
        derivative_error = 0
        error_encoder0 = 0
        error_encoder = 0

        # For achieving overall distance
        integral = 0
        derivative = 0
        error0 = 0
        error = 0
        startD = 0

        cnt = 0
        # Assign direciton of drive from argument
        self._direction = direction
        # Compute the difference in distance between start state  and goal state
        self._logger.info("Sending command toward thrusters")
        self._logger.info(f"Checking encoder to cm {self._encoder2cm(orig_dist)}")

        self._picam_frame()
        self._img_frame = cntr_bbox.dist_img(self._img_frame, self._encoder2cm(ori
        #cntr_bbox.img_show("Detecting Block",self._img_frame)

        start = time.time()

        #To avoid division by zero
        if abs(orig_dist) < 0.001:
            orig_dist = 0.001

        x = 0
        y = 0

        # Save robots pose by saving its x, yaw position
        self._yaw = self._imu_serial()
        # Convert imu notation to general +ccw from positive x
        theta = -self._yaw

        while not self._success:

            # by considering the minimum of the two encoders
            pos_encoder = int(min(self.counterFL,self.counterBR))

            # Global error between start and goal state will be
            error = orig_dist - pos_encoder
```

```
            prcnt = round(1 - error/orig_dist,3)

            if (pos_encoder % 150) == 0:

                # Save robots pose by saving its x, yaw position
                x = self._encoder2feet(pos_encoder) * math.cos(math.radians(theta)
                y = self._encoder2feet(pos_encoder) * math.sin(math.radians(theta)

                self._path.append((round(self.global_x + x,2),round(self.global_y

#                 if (self._distance() < 30):
                if ((10 - self.global_x < 1 or self.global_x < 0.9) or ((10 - self
                    self._logger.info("Too close to border. Stop and Exit straight
                    self._pwmZero()
                    break

                #Record encoder states to txt file
                outstring = str(round(self.global_x + x,2)) + ' ' + str(round(self
                self.f.write(outstring)


            if prcnt < zones[0] and self._direction == 'f' and not steady:
                if cnt == 0:
                    lower_dc = 50
                    upper_dc = 60

                    error = orig_dist - pos_encoder
                    pos_encoder_orig = pos_encoder

                    self.duty[self._direction] = [55,55]
                    self._dutycycle[self._direction] = self.duty[self._direction]
                    cnt += 1

                    self._picam_frame()
                    self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
                    self._out.write(self._img_frame)

            elif (prcnt >= zones[0] and prcnt < zones[1]) and self._direction == '
                if cnt == 1:
                    lower_dc = 35
                    upper_dc = 50

                    pos_encoder_orig += pos_encoder

                    self.duty[self._direction] = [40,40]
                    self._dutycycle[self._direction] = self.duty[self._direction]
                    cnt += 1

                    self._picam_frame()
                    self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
                    self._out.write(self._img_frame)

            elif (prcnt >= zones[1] and prcnt < zones[2]) and self._direction == '
                if cnt == 1:
                    lower_dc = 28
                    upper_dc = 32
```

```
                    self._pwmZero()
                    pos_encoder_orig += pos_encoder

                    self.duty[self._direction] = [30,30]
                    self._dutycycle[self._direction] = self.duty[self._direction]
                    cnt += 1

                    self._picam_frame()
                    self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
                    self._out.write(self._img_frame)
            elif prcnt >= zones[2]:
                if cnt == 2:
                    lower_dc = 25
                    upper_dc = 28

                    pos_encoder_orig += pos_encoder

                    self.duty[self._direction] = [22,22]
                    self._dutycycle[self._direction] = self.duty[self._direction]
                    cnt += 1

                    self._picam_frame()
                    self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
                    self._out.write(self._img_frame)


            self._dutycycle[self._direction] = [np.clip(self._dutycycle[self._dire
                                                np.clip(self._dutycycle[self._dire

            if steady:
                self._dutycycle[self._direction] = [30,30]

            if self._direction == 'f':
                # Drive robot towards direction
                self._forward(tuple(self._dutycycle[self._direction]))
            elif self._direction == 'rev':
                # Drive robot towards direction
                self._reverse(tuple(self._dutycycle[self._direction]))

            # Count encoders for left and right motors
            if int(gpio.input(12)) != int(self.buttonBR):
                self.buttonBR = int(gpio.input(12))
                self.counterBR += 1

            if int(gpio.input(7)) != int(self.buttonFL):
                self.buttonFL  = int(gpio.input(7))
                self.counterFL += 1

            # PD Controller for encoder balance
            error_encoder = self.counterFL - self.counterBR

            if error_encoder > 25:
                # Give power to corresponding motors
                self._dutycycle[self._direction][0] -=  0.1 * self.duty[self._dire
#                 self._dutycycle[self._direction][1] += 0.1 * self.duty[self._dire
            elif error_encoder < -25:
```

```
                # Give power to corresponding motors
#                 self._dutycycle[self._direction][0] += 0.1 * self.duty[self._dire
                self._dutycycle[self._direction][1] -=  0.1 * self.duty[self._dire
            else:
                # Give power to corresponding motors
                self._dutycycle[self._direction][0] = self.duty[self._direction][0
                self._dutycycle[self._direction][1] = self.duty[self._direction][1


            if (error <= thresh) and (direction in ['f','rev']):
                self._yaw = self._imu_serial()
                angle_diff = yaw0 - self._yaw
                self._logger.info(f"counterBR: {self.counterBR} counterFL: {self.c
                self._logger.info(f"Angle shifted: {angle_diff} deg ")
                self._logger.info(f"Estimated distance traveled: {self._encoder2cm
                self.counterFL = np.uint64(0)
                self.counterBR = np.uint64(0)
                self.buttonBR = int(0)
                self.buttonFL = int(0)
                self._logger.info("Arrived in front of block")
                self._pwmZero()
                self._success = True
                self._actions[self._task_now] = False

                # Convert imu notation to general +ccw from positive x
                theta = -self._yaw

                # Save robots pose by saving its x, yaw position
                x = self._encoder2feet(pos_encoder) * math.cos(math.radians(theta)
                y = self._encoder2feet(pos_encoder) * math.sin(math.radians(theta)
                self.global_x += x
                self.global_y += y
                #self.global_yaw += self._yaw

                self._path.append((round(self.global_x,2),round(self.global_y,2),r

                self._logger.info(f"x value {round(self.global_x,2)} and y value {

                # Take image and save to video object
                self._picam_frame()
                self._img_frame = cntr_bbox.dist_img(self._img_frame, self._encode
                #cntr_bbox.img_show("Detecting Block",self._img_frame)
                self._out.write(self._img_frame)

                #Record encoder states to txt file
                outstring = '#' * 30 + '\n'
                self.f.write(outstring)
                outstring = 'End of Driving {self._direction}' + '\n'
                self.f.write(outstring)
                outstring = '#' * 30 + '\n'
                self.f.write(outstring)
                break

        # Return pwm dutcycle back to base
        self._dutycycle[self._direction] = self.duty[self._direction]

```

```
    def _scan4object(self):
        """
        This function scans for block in environment by turning in place
        until it finds the block. If it doesn't find the block after full 360
        degree turn, it returns false to say not found

        Args:
            None

        Returns:
            None
        """

        self._task_now = 'scanning'
        self._actions[self._task_now] = True
        self._logger.info(f"Scanning for {self._block_now} block started")

        scan_turn = 0

        while True:
            # Detect object in frame

            self._picam_frame()
            #self._img_frame, ave_center, bbox_radius, success = self._bbox_detect
            self._img_frame, ave_center, bbox_radius, success = cntr_bbox.bbox_det

            if success not in [None,False]:
                # Object found
                diff = ave_center[0]-self.frame_center[0]
                # Convert the pixel difference to degrees
                self._turn = diff * cntr_bbox.pixel2deg

                if (self._turn > 180 and self._turn <=360):
                    self._turn = self._turn - 360

                self._yaw = -self._turn
                self._path.append((round(self.global_x,2),round(self.global_y,2),r

                self._success = True
                self._status['found'] = True
                self._task_now = 'scanning'
                self._actions[self._task_now] = False
                return self._turn

            else:
                # Pivot the robot to a predefined turn angle
                self._pivot(self.turn)
                # Read the distance from obstacle for reference
                self._distance()
                # Keep track of turn
                #self._yaw += self.turn
                scan_turn += self.turn
                self._yaw += -self._turn
                self._path.append((round(self.global_x,2),round(self.global_y,2),r

#               if scan_turn >= 90:
```

```
#                 # Scan the other side
#                 self._pivot(-self.turn)
#                 # Add scan angle
#                 scan_turn += -self.turn
#                 self._yaw += self._turn
#                 self._path.append((round(self.global_x,2),round(self.global_y,2),
#
#             if scan_turn <= -90:
#                 self._status['found'] = False
#                 self._success = False
#                 self._task_now = 'scanning'
#                 self._actions[self._task_now] = False
#                 self._logger.warning("No objects in nearby vicinity")
#                 return None

            if abs(scan_turn) >= 360:
                self._status['found'] = False
                self._success = False
                self._task_now = 'scanning'
                self._actions[self._task_now] = False
                self._logger.warning("No objects in nearby vicinity")
                return None

    def _pick_up(self):
        """
        This function drives the robot to steadily approach and picks up toy block

        Args:
            None

        Returns:
            None
        """
        self._success = False
        self._task_now = 'picking'
        self._actions[self._task_now] = True

        # Take image and save to video object
        self._picam_frame()
        #cntr_bbox.img_show("Detecting Block",self._img_frame)
        self._out.write(self._img_frame)

        while not self._success:
            # Open gripper
            self._servo_cntrl(self.grip_state['opened'])
            self._gripper_state = self.grip_state['opened']
            self._logger.info("Servo Opened")
            # Take image and save to video object
            self._picam_frame()
            #cntr_bbox.img_show("Detecting Block",self._img_frame)
            self._out.write(self._img_frame)
            # Align robot fine turne rotation
            degree = self._scan4object()

#             while True:
#                 if degree is not None:
```

```
#                     if abs(degree) > 0.6:
#                         self._pivot(degree, steady=True)
#                     else:
#                         break
#                 else:
#                     break
            # Scan block and drive slowly towards it
            self._logger.info("Driving slowly toward block")
            self._go2block(steady=True)
            self._logger.info("Slow drive completed")

            # Take image and save to video object
            self._picam_frame()
            #cntr_bbox.img_show("Detecting Block",self._img_frame)
            self._out.write(self._img_frame)

            # Close gripper
            self._servo_cntrl(self.grip_state['closed'])
            self._gripper_state = self.grip_state['closed']
            self._logger.info("Servo Closed")

            # Take image and save to video object
            self._picam_frame()
            #cntr_bbox.img_show("Detecting Block",self._img_frame)
            self._out.write(self._img_frame)

            #if (abs(radius2 - radius1) < 5) or (abs(center2[0] - center1[0]) < 3)
            self._logger.info(f"The {self._block_now} block has been gripped succe
            self._success = True
            self._status['picked_up'] = True
            self._actions[self._task_now] = False

        # Read saved image and send via email
        self._email_media()

    def _go2(self,destination=(0,0),area=''):
        """
        This function computes the difference in pose between the robot and the de
        then estimates the displacement vector along with the angle of rotation ne
        to align the robot head  on. It then calls the pivot and go2block function
        and command the robot to enter the construction zone.

        Args:
            destination (tuple): contans x,y location of destination
            area (str): Describes where that destination is

        Returns:
            None
        """
        self._logger.info(f"Initiating travel to {area}")
        self._success = False
        self._task_now = 'returning'
        self._actions[self._task_now] = False

        # Read last position from the appended path
        # self.global_x = self._path[-1][0]
```

```
        # self.global_y = self._path[-1][1]
        #self.global_yaw = self._path[-1][2]
        delta_x = destination[0]-self.global_x
        delta_y = destination[1]-self.global_y
        self._yaw = self._imu_serial()

        self._logger.info(f"Pose Now: X- {self.global_x} and Y- {self.global_y} an
        # Compute the euclidean distance between current robot location and constru
        disp = math.sqrt((delta_x)**2 + (delta_y)**2)
        # Calculate the yaw angle difference between these two points using slope
        rad_diff = math.atan2(delta_y,delta_x)
        # Convert from radians to angle
        target_angle = math.degrees(rad_diff)
        # Ensure angle is between [0, 360]
        target_angle %= 360

        self._logger.info(f"Computed target angle is {target_angle}")

        # # Place angle from [0,360] to [-180,0) U [0,180]
        if target_angle >=-180 and target_angle <=180:
            target_angle = -target_angle
        elif (target_angle >= 180 and target_angle <360):
            target_angle = 360 - target_angle
        elif (target_angle > -360 and target_angle <=-180):
            target_angle = -(target_angle + 360)

        self._logger.info(f"Normalized target angle is {target_angle}")
#         # Normalize angle difference to range [-180, 180) degrees
#         target_angle = (target_angle + 180) % 360 - 180
        # # Angle difference from current and target
        # deg_diff = self._yaw - target_angle
        # deg_diff %= 360

        # Based on the quadrant of the point heading towards normalize
        # the target angle
#         if delta_x < 0:
#             if delta_y > 0:
#                 #target_angle = 180 - target_angle
#                 pass
#             else:
# #                 target_angle = target_angle - 180
#                 pass
#         else:
#             if delta_y < 0:
# #                 target_angle = -target_angle
#                 pass

        # convert target angle with respect to imu configuration
        # target_angle = -target_angle

        # Angle difference from current and target
        deg_diff = target_angle - self._yaw
        #deg_diff %= 360

#         if (deg_diff > 180 and deg_diff <360):
#             deg_diff = (deg_diff - 360) # Turning to the right
```

```
#         elif (deg_diff > -360 and deg_diff <=-180):
#             deg_diff = (deg_diff + 360) # Turning to the left

#         # Place angle from [0,360] to [-180,0) U [0,180]
#         if (deg_diff > 180 and deg_diff <=360):
#             deg_diff = deg_diff - 360
#         # Normalize angle difference to range [-180, 180) degrees
#         deg_diff = (deg_diff + 180) % 360 - 180

        self._logger.info(f"Yaw that vehicle will pivot to {deg_diff}")
        # Pivot robot to face construction zone
        self._pivot(deg_diff, just_turn=True)
        self._logger.info(f"Started driving straight to {area}")
        # Drive robot foward to construction zone
        self._go2block(eucl_disp = disp,just_drive = True)
        self._logger.info(f"Arrived at {area}")

        # New global yaw
        self.global_yaw = self._imu_serial()

        self._success = True
        self._status['arrived'] = True
        self._actions[self._task_now] = False

    def _place(self):
        """
        This function places toy block

        Args:
            None

        Returns:
            None
        """
        self._success = False
        self._task_now = 'placing'
        self._actions[self._task_now] = True

        # Open gripper
        self._servo_cntrl(self.grip_state['opened'])
        self._gripper_state = self.grip_state['opened']
        self._logger.info(f"Gripper Opened to place {self._block_now} block")

        # Drive in reverse
        self._go2block(direction='rev',eucl_disp=7.5, just_drive=True, steady=True
        time.sleep(1)
        self._logger.info("Backed away from toy block")

        # Close gripper
        self._servo_cntrl(self.grip_state['closed'])
        self._gripper_state = self.grip_state['closed']
        self._logger.info("Gripper Closed to prepare for next order")

        self._success = True
        self._status['placed'] = True
        self._actions[self._task_now] = False
```

```

        self._logger.info(f"{self._block_now} block is now placed inside construct

        # Read saved image and send via email
        self._email_media()

    def _localize(self):
        """
        This function uses distance sensor to read from borderwall
        so that the robot's exact position in arena can easily be
        identified

        Args:
            None

        Returns:
            None
        """
        self._success = False
        self._task_now = 'localizing'
        self._actions[self._task_now] = True

        # pivot the robot to positive y first and read from distance sensor
        yaw_final = -90
        yaw_current = self._imu_serial()
        deg_diff = yaw_final - yaw_current

        #deg_diff %= 360
        # Normalize angle difference to range [-180, 180) degrees
        if (deg_diff > 180 and deg_diff <360):
            deg_diff = 360 - deg_diff
        elif (deg_diff > -360 and deg_diff <= -180):
            deg_diff = -(deg_diff + 360)

        # Turn towards borderwall
        self._pivot(deg_diff, just_turn=True)

        # Now read distance sensor value
        self._distance()
        delta_ft = self._cm2feet(self._depth)
        self._yaw = self._imu_serial()
        theta = -self._yaw
        x1 = delta_ft * math.cos(math.radians(theta))
        y1 = delta_ft * math.sin(math.radians(theta))

        # Extract y pose by normalizing from border wall
        if self.global_y > 5:
            self.global_y = 10 - y1
        else:
            self.global_y = y1

        # pivot the robot to negative x first and read from distance sensor
        yaw_final = -179
        yaw_current = self._imu_serial()
        deg_diff = yaw_final - yaw_current
        #deg_diff %= 360
```

```
        # Normalize angle difference to range [-180, 180) degrees
        if (deg_diff > 180 and deg_diff <360):
            deg_diff = 360 - deg_diff
        elif (deg_diff > -360 and deg_diff <= -180):
            deg_diff = -(deg_diff + 360)

        # Turn towards borderwall
        self._pivot(deg_diff, just_turn=True)

        # Now read distance sensor value
        self._distance()
        delta_ft = self._cm2feet(self._depth)
        self._yaw = self._imu_serial()
        theta = -self._yaw
        x2 = delta_ft * math.cos(math.radians(theta))
        y2 = delta_ft * math.sin(math.radians(theta))

        # Extract x pose by normalizing from border wall
        if self._x > 5:
            self.global_x = 10 - x2
        else:
            self.global_x = x2

        # Add detected pose of robot into path list
        self._path.append((round(self.global_x,2),round(self.global_y,2),round(sel

        self._success = True
        self._actions[self._task_now] = False
        self._logger.info(f"Localization complete")

    def _go2site(self):
        """
        This function commands the robot to drive back to the center
        of the arena or near the cluttered enviorment to resume delivering
        order

        Args:
            None

        Returns:
            None
        """

        self._logger.info("Initiating travel to cluttered environment")
        # Compute the euclidean distance between current robot location and constru
        disp = math.sqrt((self._x - self.clutter_env[0])**2 + (self._y - self.clut
        # Calculate the yaw angle difference between these two points using slope
        rad_diff = math.atan2(self.clutter_env[0] - self._x, self.clutter_env[1] -
        # Convert from radians to angle
        deg_diff = math.degrees(rad_diff)
        # Ensure angle is between [0, 360]
        deg_diff %= 360
        # Place angle from [0,360] to [-180,0) U [0,180]
        if (deg_diff > 180 and deg_diff <=360):
            deg_diff = deg_diff - 360
```

```

        self._logger.info("Orienting to cluttered environment")
        # Pivot robot to face construction zone
        self._pivot(deg_diff)
        self._logger.info("Started driving straight to cluttered environment")
        # Drive robot foward to construction zone
        self._go2block(eucl_disp = disp,just_drive = True)
        self._logger.info("Arrived at cluttered environment")

    def _go_home(self):
        """
        This function commands the robot to drive back to its home location
        marking the end of the challenge.

        Args:
            None

        Returns:
            None
        """
        self._logger.info("Initiating travel to home base")
        # Compute the euclidean distance between current robot location and constru
        disp = math.sqrt((self._x - self.start_pt[0])**2 + (self._y - self.start_p
        # Calculate the yaw angle difference between these two points using slope e
        rad_diff = math.atan2(self.start_pt[0] - self._x, self.start_pt[1] - self._
        # Convert from radians to angle
        deg_diff = math.degrees(rad_diff)
        # Ensure angle is between [0, 360]
        deg_diff %= 360
        # Place angle from [0,360] to [-180,0) U [0,180]
        if (deg_diff > 180 and deg_diff <=360):
            deg_diff = deg_diff - 360

        self._logger.info("Orienting to home base")
        # Pivot robot to face construction zone
        self._pivot(deg_diff)
        self._logger.info("Started driving straight to home base")
        # Drive robot foward to construction zone
        self._go2block(eucl_disp = disp,just_drive = True)
        self._logger.info("Arrived at home")
        self._logger.info("Thank you for playing game.")
        self._logger.info("Powering off.")
        self._end_game = True

    def plot_path(self):
        """
        Plot the path of the robot during the grand challenge.

        Args:
            None
        Return:
            None
        """
        # Take in the path list as an array 2D NumPy array
        path = np.array(self._path)
        x = path[:,0]
```

```
        y = path[:,1]

        # Create figure and plot
        fig, ax = plt.subplots(1,1)

        fig.suptitle('Motor Encoder Analysis')

        ax.plot(x,y,ls='solid', color='blue',linewidth=2, label=f'{self._name} Pat

        # 3) Label the axes, title, legend
        ax.set_ylabel('Y')
        ax.set_xlabel('X')
        ax.set_xlim(0,10)
        ax.set_ylim(0,10)
        plt.savefig('grandChallenge-robotPath.png')
        plt.show()
        plt.close()

    def start(self, order: List):
        """
        The sequence of actions occuring while baron robot works to pick and place
        all toy blocks to the construction zone. This function lists the steps nee
        for the robot to complete the orders passed to it at the beginning of comp

        Args:
            order: List[Blocks] list of all toy blocks needed to complete challeng

        Returns:
            None
        """

        try:
            # Assign order from object initialization
            self._order = order

            # First wait until startup QR Code has been read
            while True:
                if self._wait2start():
                    break

            while len(self._order) > 0:
                """While Robot is still active, resume challenge"""
                if (not self._end_game):

                    self._block_now = self._order.pop(0)
                    self._order_no +=1

                    # Begin challenge with choesn block
                    self._logger.info("-" * 20)
                    self._logger.info(f"*** Picking Up {self._block_now} block")
                    self._logger.info("-" * 20)

                    # Track the number of times robot drives to search area
                    search = 0

                    # Drive robot to center of arena, get view of toy blocks and b
```

```
                    # extracting blocks
                    self._go2(destination=self.clutter_env,area='cluttered environ

                    while True:

                        # Begin by scanning for toy block object
                        degree = self._scan4object()

                        # Take image and save to video object
                        #cntr_bbox.img_show("Detecting Block",self._img_frame)
                        self._out.write(self._img_frame)

                        # Drive toward block if successful, else pivot to
                        # a free space and drive
                        if self._success:
                            #self._logger(f"Pivoting Robot towards {self._block_no
                            self._pivot(degree,steady=True)

                            # Take image and save to video object
                            self._picam_frame()
                            __, __, __,__= cntr_bbox.bbox_detect(self._img_frame, 
                            #cntr_bbox.img_show("Detecting Block",self._img_frame)
                            self._out.write(self._img_frame)

                            self._logger.info(f"Drive Robot forward to {self._bloc
                            self._go2block()

                            # # Update global x and y
                            # self.global_x += self._x
                            # self.global_y += self._y

                            ##self._logger(f"{self._name} is now at x- {self.globa

                            # Take image and save to video object
                            self._picam_frame()
                            #cntr_bbox.img_show("Arrived at Block",self._img_frame,
                            self._out.write(self._img_frame)

                            ##self._logger(f"Pick up {self._block_now} block with 
                            self._pick_up()

                            # Drive robot to construction zone to place block
                            ##self._logger(f"Take {self._block_now} block to const
                            self._go2(destination=self.const_zone,area='constructi

                            #self._logger(f"Place {self._block_now} block inside c
                            self._place()

#                             #self._logger("Localizing to get accurate location o
                            self._localize()

                            ##self._logger(f"Pick up and place the next order in t
                            self._success = True
                            break
                        else:
                            #self._logger("No toy blocks found in current pose. Ch
```

```
                        #self._logger("try again.")

                        # Drive robot to the predetermined survey locations
                        if search > 1:
                            self._go2(destination=self._survey_loc[search],are
                        else:
                            #self._logger(f"Robot can't find {self._block_now}
                            #self._logger("In pursuit of time, moving on to ne
                            break
                        search += 1

                else:
                    #self._logger(f"Mission complete. Take robot back to home base")
                    self._go2(destination=self.start_pt,area='home base')
                    return

            except KeyboardInterrupt as error:
                self._logger.error("Error occurred: %s",error)
                self._gameover()
                time.sleep(1)
                return None
            except ValueError as error:
                self._logger.error("Error %s", error, exc_info=True)
                self._gameover()
                time.sleep(1)
                return None
```

class Robot:

▼ View Source

```
class Robot(object):
    """
    A class defining basic attributes of Baron robot that
    will be autonomously achieving objective

    Attributes:
        name (str): Name of robot
        _x (float): horizontal position of robot from start postion
        _y (float): vertical position of robot from start position
        _yaw (float): yaw orientation from +x as horizontal [0,180] or (-180,0)
        _travel(float): estimating the distance to drive and reach block
        _turn (float): keeps the next turn needed by block
        _depth (float): measures distance of obstacle in front of robot from image
        _path (list): stores the path traveled by robot as list of (x,y) points an
        _pwms (list): stores a list of pwms for the robot motors
        _dutycycle (dict): holds the baseline pwm dutycycle for forward,reverse,
                           pivotleft and pivotright
        _success (bool): keeps track whether a given task is successful or not
        _gripper_state (str): stores gripper duty cycle state as opened or closed
        _order(list): List of blocks to pickup
        _end_game (bool): tells competition is over by victory or failure
        _order_no (int): keeps track of number of orders completed
        _block_now (str): the block being pursued now
```

```
        _task_now (str): keeps track of the task being completed at the moment
        _actions (dict): holding list of all service actions robot performs with b
                         on which process is being done at a given time
        _status (dict): a dictionary keeping track of completed actions along with
        _img_frame (img): saves current image frame
        _out (cv2 video obj): gather all frames into a video
        _logger (log): log statements for debuging and status over the terminal
        _kp (float): proportional gain
        _ki (float): integral gain
        _kd (float): derivative gain
        _direction (str): tells weather robot is moving forward or reverse
        _event (event): event object to end the thread call
        _pause_event (event): event object to pause and resume thread
        imu_reader (thread): threading object to read from imu during forward run
        _ser (serial): object reading serial input streams from imu sensor
        _img_saved (img): keeping track of saved image to send in an email
        _survey_loc (List): containes x,y locations for the robot to navigate and
                            so that to extract toy blocks for mission

    """

    # define the codec and create VideoWriter object
    fourcc = cv2.VideoWriter_fourcc(*'XVID')
    # Gripper States
    grip_state = dict([('closed',2.5),('opened',7.5)])
    # Matrix for Image frame
    image = np.empty((480*640*3,),dtype=np.uint8)
    # define the codec and create VideoWriter object
    fourcc = cv2.VideoWriter_fourcc(*'XVID')
    # Open .txt file to save data
    f = open('robot_path_final.txt','a')

    dutyset = [('f', [70,70]), ('rev', [30,30]),('l', 100),('r',100)]

    duty = dict(dutyset)
    font = cv2.FONT_HERSHEY_COMPLEX_SMALL

    # turn angel
    turn = 15

    # detect radius
    bbox_radius = 0

    # Initialize FL and BR button count
    counterBR = np.uint64(0)
    counterFL = np.uint64(0)
    buttonBR = int(0)
    buttonFL = int(0)

    # Proportional Control Gain
    Kp = 1
    Ki = 0.05
    Kd = 0.08

    # Important locations in map in units of feet
    origin = [0,0]
```

```
    start_pt = [1,1]
    clutter_env = [5,5]
    const_zone = [2.25,6.5]

    # create object to read camera
    video = cv2.VideoCapture(0)

    frame_center = (320,240)
    (cx,cy) = (frame_center[0]/4.5, frame_center[1]/8)

    queue = queue.Queue()

    img_folder = 'challenge_pics'

    counter = 0

    # Environment assessment points
    scan_spots = [(7.5,3),(7.5,8)]

    # Global x, y, positions and yaw from fixed reference
    # frame (0,0)
    global_x = 1
    global_y = 1
    global_yaw = 0

    def __init__(self, name: str):
        """
        Initialize Robot attributes

        Args:
            name: str - takes name of robot

        Returns:
            None
        """

        self.name = name
        self._x = self.start_pt[0]
        self._y = self.start_pt[1]
        self._yaw = 0.0
        self._travel = 0
        self._turn = 0
        self._depth = 0
        self._path = [(1,1,0)]
        self._pwms = []
        self._dutycycle = self.duty
        self._success = False
        self._gripper_state = self.grip_state['closed']
        self._order = []
        self._end_game = False
        self._order_no = 0
        self._block_now = ''
        self._task_now = ''
        self._actions = dict([('driving',False),('rotating',False),('picking',Fals
        self._status = dict([('started',False),('picked_up',False),('found',False)
        self._img_frame = self.image
```

```
        self._out = cv2.VideoWriter('grand_chall_vid.avi', self.fourcc, 3, (640, 48
        self._logger = logging.getLogger(name)
        self._kp = self.Kp
        self._ki = self.Ki
        self._kd = self.Kd
        self._direction = 'f'
        self._event = threading.Event()
        self._pause_event = threading.Event()
        self.imu_reader = threading.Thread(target = self._imu_serial_straight, args
        self._ser = serial.Serial('/dev/ttyUSB0', 19200)
        self._img_saved = self.image
        self._survey_loc = self.scan_spots

        # Set up logger with Debug initially
        self._logger.setLevel(logging.DEBUG)
        # Create console handler and set level to debug
        console_handler = logging.StreamHandler()
        console_handler.setLevel(logging.DEBUG)
        # log terminal messages to a file
        file_handler = logging.FileHandler('gc_log_3.log')
        file_handler.setLevel(logging.INFO)
        # Create formatter
        formatter = logging.Formatter('%(name)s - %(levelname)s - %(message)s')
        # Add formatter to handler
        console_handler.setFormatter(formatter)
        file_handler.setFormatter(formatter)
        # Add console handler to logger
        self._logger.addHandler(console_handler)
        self._logger.addHandler(file_handler)
        # Initialize gpios and pwms
        self._init()


    @property
    def name(self):
        return self._name

    @name.setter
    def name(self, name):
        self._name = name

    def _init(self):
        """
        Initialize gpio pins and set pwms to command DC motors, encoders and
        servo

        Args:
            None

        Returns:
            None
        """

        gpio.cleanup()
        gpio.setmode(gpio.BOARD)

```

```
        # Setup GPIO pin(s)
        gpio.setup(36, gpio.OUT) # Servo

        gpio.setup(31, gpio.OUT) # IN1
        gpio.setup(33, gpio.OUT) # IN2
        gpio.setup(35, gpio.OUT) # IN3
        gpio.setup(37, gpio.OUT) # IN4

        gpio.setup(7, gpio.IN, pull_up_down = gpio.PUD_UP)
        gpio.setup(12, gpio.IN, pull_up_down = gpio.PUD_UP)

        self._pwms.clear()

        # initialize pwm signal to control motor
        pwm01 = gpio.PWM(31, 50)  # BackLeft motor
        pwm11 = gpio.PWM(33, 50) # FrontLeft motor
        pwm22 = gpio.PWM(35, 50) # FrontRight motor
        pwm02 = gpio.PWM(37, 50)  # BackRight motor
        pwmS = gpio.PWM(36, 50) # Servo

        self._pwms = [pwm01,pwm11,pwm22,pwm02,pwmS]

        for pwm in self._pwms:
            pwm.start(0)

    def _pwmZero(self):
        """
        Stops motors by zeroing pwm values in all pins
        controlling DC motors

        Args:
            None

        Returns:
            None
        """


        self._pwms[0].ChangeDutyCycle(0)
        self._pwms[1].ChangeDutyCycle(0)
        self._pwms[2].ChangeDutyCycle(0)
        self._pwms[3].ChangeDutyCycle(0)

    def _gameover(self):
        """
        Terminates run by stopping pwms and cleaning up
        gpio pins

        Args:
            None

        Returns:
            None
        """

        self._pwmZero()
```

```
        self._pwms[-1].ChangeDutyCycle(self.grip_state['closed'])
        time.sleep(1)
        for pwm in self._pwms:
            pwm.stop()
        gpio.cleanup()

    def _forward(self,vals):
        """
        Commands DC motors to drive robot forward by sending
        Dutycycles to respective pwm pins

        Args:
            None

        Returns:
            None
        """
        # Left wheels
        self._pwms[0].ChangeDutyCycle(vals[0])
        self._pwms[1].ChangeDutyCycle(0)
        # Right wheels
        self._pwms[2].ChangeDutyCycle(0)
        self._pwms[3].ChangeDutyCycle(vals[1])

    def _reverse(self,vals):
        """
        Commands DC motors to drive robot in reverses by sending
        Dutycycles to respective pwm pins

        Args:
            None

        Returns:
            None
        """
        # Left wheels
        self._pwms[0].ChangeDutyCycle(0)
        self._pwms[1].ChangeDutyCycle(vals[0])
        # Right wheels
        self._pwms[2].ChangeDutyCycle(vals[1])
        self._pwms[3].ChangeDutyCycle(0)

    def _pivotleft(self,vals):
        """
        Commands DC motors to pivot robot left by sending
        Dutycycles to respective pwm pins

        Args:
            None

        Returns:
            None
        """
        # Left wheels
        self._pwms[0].ChangeDutyCycle(0)
        self._pwms[1].ChangeDutyCycle(vals)
```

```
        # Right wheels
        self._pwms[2].ChangeDutyCycle(0)
        self._pwms[3].ChangeDutyCycle(vals)

    def _pivotright(self,vals):
        """
        Commands DC motors to pivot robot right by sending
        Dutycycles to respective pwm pins

        Args:
            None

        Returns:
            None
        """

        # Left wheels
        self._pwms[0].ChangeDutyCycle(vals)
        self._pwms[1].ChangeDutyCycle(0)
        # Right wheels
        self._pwms[2].ChangeDutyCycle(vals)
        self._pwms[3].ChangeDutyCycle(0)

    def _servo_cntrl(self,duty_cycle):
        """
        Commands servo motors to open and close gripper by sending
        Dutycycles to respective pwm pins

        Args:
            None

        Returns:
            None
        """

        self._pwms[-1].ChangeDutyCycle(duty_cycle)
        time.sleep(1)

        img = cntr_bbox.servo_img(duty_cycle)
        self._img_frame = cntr_bbox.dist_img(img, self._distance())

        #cntr_bbox.img_show("Servo status",self._img_frame)
        self._out.write(img)

    def _picam_frame(self):
        """
        Calls picamera object to take image frame from the pi camera
        and applies corrections by flipping to reflect reality

        Args:
            None

        Returns:
            None
        """
        # Take picture with camera
```

```
        cntr_bbox.camera.capture(self.image, format="bgr")
        image = self.image.reshape((480,640,3))

        self._img_frame = cv2.flip(image,1)

    def _mask_color(self, imageHSV):

        if self._block_now == 'green':
            # Trail Green Bock - LAB
            minHSV = np.array([46,88,121])
            maxHSV = np.array([65,141,255])
        elif self._block_now == 'red':
            # Trail Red block - LAB
            minHSV = np.array([151,107,147])
            maxHSV = np.array([255,255,255])
        elif self._block_now == 'blue':
            # Trail Blue block - LAB
            minHSV = np.array([72,94,97])
            maxHSV = np.array([126,188,212])

        maskHSV = cv2.inRange(imageHSV, minHSV, maxHSV)

        return maskHSV

    def _prep_image(self):
        """
        Processes images by applying HSV masking and gaussian blur
        for use by latter object detection functions

        Args:
            None

        Returns:
            None
        """

         # Convert image from BGR to HSV space
        imageHSV = cv2.cvtColor(self._img_frame,cv2.COLOR_BGR2HSV)

        # mask the green light from HSV and convert to grayscale
        mask = self._mask_color(imageHSV)

        # Mask HSV masked image of arrow
        blurred = cv2.GaussianBlur(mask,(11,11), 0)

        return blurred

    def _grip_checker(self):
        """
        Processes images by applying a rectangular mask over desired
        region of interest to verify whether gripper has held toy block

        Args:
            None

        Returns:
```

```
            self._image_frame (img): Masked image
        """
        self._logger.info("Inside Grip Checker Function")
        # Create a mask of gripped block in image
        # Save the shape of the image array
        (height, width, c)  = self._img_frame.shape

        # Create mask of image just like picture
        mask = np.zeros_like(self._img_frame[:,:,0])

        # Create a white rectangle
        x,y,w,h = round(width/2)-130, round(height/2)-30, 250, 300
        cv2.rectangle(mask,  (x,y), (x+w, y+h), 255, -1)

        # Extract the image that aligns in the rectangle
        self._img_frame = cv2.bitwise_and(self._img_frame, self._img_frame, mask=m

        return self._img_frame

    def _wait2start(self):
        """
        Waits for the competition start QR code to be visible,
        and reads QR code in, which if corresponds to "ENPM701"
        returns a True boolean to begin competition

        Args:
            None

        Returns:
            start (bool): True/False based on detection of expected QR
        """
        # Define detector
        detector = cv2.QRCodeDetector()

        # Check if program start initiated
        start = False

        cnt = 0

        self._logger.info("Waiting for start cue . . . ")

        try:

            for frame in cntr_bbox.camera.capture_continuous(cntr_bbox.rawCapture,

                # grab the current frame
                img = frame.array

                img = cv2.flip(img,1)

                data, bbox, _ = detector.detectAndDecode(img)

                if(bbox is not None):
                    for i in range(len(bbox)):
                        cv2.line(img, tuple(bbox[i][0]),tuple(bbox[(i+1)%len(bbox)
                        cv2.putText(img, data, (int(bbox[0][0][0]),int(bbox[0][0][
```

```

                if data:
                    self._logger.info(f"Data: {data}")
                    pass

                # Show resutls to the screen
                cv2.imshow("QR Code detector",img)
                key = cv2.waitKey(1) & 0xFF

                # write frame into file
                self._out.write(img)

                # clear the stream in preparation for the next frame
                cntr_bbox.rawCapture.truncate(0)

                # Break out of loop by pressing the q key
                if(key == ord("q")):
                    self._logger.warn("Program Terminated")
                    self.video.release()
                    cv2.destroyAllWindows()
                    break

                if data == "ENPM701":
                    self._logger.info("Cue Received \n")
                    self._logger.info('@' * 45)
                    self._logger.info("Starting Grand Challenge! \n")
                    self._logger.info('#' * 45)
                    start = True
                    self.video.release()
                    cv2.destroyAllWindows()
                    time.sleep(3)
                    break

        except Exception as error:
            self._logger.error(error)
            self.video.release()
            cv2.destroyAllWindows()

        return start

    def _email_media(self):

        # Take a picture of gripped block to email
        self._picam_frame()
        # Append status of servo on image
        self._img_frame = cntr_bbox.servo_img(self._gripper_state)
        self._out.write(self._img_frame)
        # Save to designated directory
        cv2.imwrite(f'{self.img_folder}/{self._block_now}_{self.counter}.png', sel
        #time.sleep(1)
        # Read saved image
        img_name = f'{self.img_folder}/{self._block_now}_{self.counter}.png'

        # send email to user with images
        smtpUser = 'ykebede2@terpmail.umd.edu'
        smtpPass = 'QwE@$d1219'
```

```python

        toAdd = ['ENPM809TS19@gmail.com','yosephcollege@gmail.com']
        fromAdd = smtpUser

        f_time = datetime.now().strftime('%a %d %b @ %H:%M')
        subject = f'ENPM701-GrandChallenge-{f_time}-yosephK-{self._block_now}-{sel

        self.counter += 1

        msg = MIMEMultipart()
        msg['Subject'] = subject
        msg['From'] = fromAdd
        msg['To'] = ",".join(toAdd)

        msg.preamble = "Image @ " + f_time

        body = email.mime.text.MIMEText("Baron Robot image: " + f_time)
        msg.attach(body)

        fp = open(img_name, 'rb')
        img = MIMEImage(fp.read())
        fp.close()
        msg.attach(img)

        s = smtplib.SMTP('smtp.gmail.com', 587)
        s.ehlo()
        s.starttls()
        s.ehlo()
        s.login(smtpUser, smtpPass)
        s.sendmail(fromAdd, toAdd, msg.as_string())
        s.quit()

        self._logger.info("Email delivered!")

    def _meter2encoder(self,x_dist):
        """
        Reads in distance values in meters and converts to
        encoder counts

        Args:
            None

        Returns:
            encod (int): converted encoder counts from distance in m
        """
        encod = round(float(x_dist / (2*math.pi*0.0325))*960)

        return encod

    def _encoder2meter(self,encd):
        """
        Reads in encoder values and converts them to meters

        Args:
            None

```

```
        Returns:
            cms (float): converted encoder counts to distance in m
        """
        dist = round(float((encd / 960) * 2*math.pi*0.0325),4)

        return dist

    def _cm2encoder(self,x_dist):
        """
        Reads in distance values in centimeters and converts to
        encoder counts

        Args:
            None

        Returns:
            encod (int): converted encoder counts from distance in cms
        """
        dist = x_dist/100
        encod = self._meter2encoder(dist)
        return encod

    def _encoder2cm(self,encd):
        """
        Reads in encoder values and converts them to centimeters

        Args:
            None

        Returns:
            cms (float): converted encoder counts to distance in cms
        """
        dist = self._encoder2meter(encd)
        cms = round(dist*100,4)

        return cms

    def _feet2encoder(self,x_dist):
        """
        Reads in distance values in feet and converts to
        encoder counts

        Args:
            None

        Returns:
            encod (int): converted encoder counts from distance in feet
        """
        dist = round(float(x_dist * 0.3048),4)
        encod = self._meter2encoder(dist)
        return encod

    def _encoder2feet(self,x_encod):
        """
        Reads in encoder values and converts to
        feet
```

```

        Args:
            None

        Returns:
            feet (float): converted encoder counts from distance in feet
        """
        meter = self._encoder2meter(x_encod)
        feet = round(meter * 3.28084,4)

        return feet

    def _feet2cm(self,x_dist):
        """
        Reads in distance values in feet and converts to
        encoder counts

        Args:
            None

        Returns:
            encod (int): converted encoder counts from distance in feet
        """
        dist = round(float(x_dist * 12 * 2.54),3)

        return dist

    def _cm2feet(self,x_dist):
        """
        Reads in distance values in feet and converts to
        encoder counts

        Args:
            None

        Returns:
            encod (int): converted encoder counts from distance in feet
        """
        dist = round(float(x_dist / (12 * 2.54)),2)

        return dist

    def _imu_serial(self):
        """
        Reads in serial robot's yaw angle value from imu sensor, cleans buffer,
        and converts it into a float data type. Convers reading from 0-360 to
        -180 < yaw < 180.

        Args:
            None

        Returns:
            self._yaw (float): robots yaw in degrees
        """
        self._ser.reset_input_buffer()
        self._ser.flush()
```

```python

        while True:

            try:
                # Read for imu from serial
                if(self._ser.in_waiting > 0):
        #                 if cnt >5:
                    # Strip serial stream of extra characters
                    line = self._ser.readline()

                    line = line.rstrip().lstrip()

                    line = str(line)
                    line = line.strip("'")
                    line = line.strip("b'")

                    # Return float
                    line = float(line)

                    if (line > 180 and line <=360):
                        line = line - 360

                    self._yaw = line
                    return self._yaw

            except Exception as error:
                self._logger.error(f"Imu Error: {error}")

    def _imu_serial_straight(self,queue):
        """
        Reads in serial robot's yaw angle value from imu sensor, cleans buffer,
        and converts it into a float data type. Convers reading from 0-360 to
        -180 < yaw < 180.

        Only difference from self._imu_serial() is that this is run in a separate
        thread which is controlled by events to start, pause, and stop function ca

        Args:
            None

        Returns:
            self._yaw (float): robots yaw in degrees
        """

        while not self._event.is_set():

            self._ser.reset_input_buffer()
            self._ser.flush()

            try:
                while self._pause_event.is_set():
                    # Read for imu from serial
                    if(self._ser.in_waiting > 0):
                            line = self._ser.readline()

                            line = line.rstrip().lstrip()
```

```

                            line = str(line)
                            line = line.strip("'")
                            line = line.strip("b'")

                            # Return float
                            line = float(line)

                            if (line > 180 and line <=360):
                                line = line - 360

                            queue.put(line)

            except Exception as error:
                self._logger.warning(f"Imu Error inside thread: {error}")

    def _distance(self):
        """
        Reads in serial robot's distance from an obstacle using ultrasonic sensor l
        sending a trigger pulse and listening the returning echo to compuse the sep
        distance from nearby object using triangulation technique. Then it saves co
        distance in self._depth

        Args:
            None

        Returns:
            None
        """
        pulse_start = 0
        pulse_end = 0

        # Define pin allocations
        trig = 16
        echo = 18
        # Setup GPIO board & pins
        gpio.setup(trig, gpio.OUT)
        gpio.setup(echo, gpio.IN)

        # Ensure output has no value
        gpio.output(trig, False)
        time.sleep(0.01)

        # Generate trigger pulse
        gpio.output(trig, True)
        time.sleep(0.00001)
        gpio.output(trig, False)

        # Generate echo time signal
        while gpio.input(echo) == 0:
            pulse_start = time.time()

        while gpio.input(echo) == 1:
            pulse_end = time.time()

        pulse_duration = pulse_end - pulse_start
```

```python

        # Convert time to distance
        distance = pulse_duration * 17150
        self._depth = round(distance, 2)

        return self._depth

    def _rotate(self):
        """
        Reads in direction of rotation requested, and starts
        rotating the motors accordingly to achieve desired orientation.


        Args:
            None

        Returns:
            None
        """

        if self._direction == 'r':
            self._pivotright(60)
        elif self._direction == 'l':
            self._pivotleft(60)

    def _dist_estimate(self):
        """
        Reads in radius of bounding box from bbox detector function, and
        plugs radius into precomputed correlation function to estimate the
        depth of toy block from robot.

        Assigns results to self._travel and saves image showing value of distance
        appended to it in self._img_frame and writes frame to video object self._ou

        Args:
            None

        Returns:
            None
        """
        # Take image with picamera
        self._picam_frame()
        self._img_frame, __, bbox_radius, self._success = cntr_bbox.bbox_detect(se

        if self._success not in [None, False]:
            # Plug in pixel size into depth equation
            distance = (0.0099 * bbox_radius**2) - (1.8846*bbox_radius) + 103.47
            self._travel = round(distance,2)
            # Append image of estimated distance on image frame
            self._img_frame = cntr_bbox.dist_img(self._img_frame,self._travel)
            # save image into video object
            self._out.write(self._img_frame)
    def _corner_detect(self,img,origImg):

        # Create a list to store the x,y location of points
        pts_loc = []
```

```
        # Detect corners from image
        corners = cv2.goodFeaturesToTrack(img,5,0.01,10)

        if corners is not None:

            corners = np.int0(corners)

            # identify location of corners in image
            for i in corners:
                # Extract x,y coordinate of points
                x,y = i.ravel()

                pts_loc.append([x,y])

                # Draw circle ontop of original image
                cv2.circle(origImg, (x,y), 3, (0,0,255),-1)

            # Create a column vector from pts list
            pts_loc = np.array(pts_loc)

            return img, pts_loc, origImg, True
        else:
            return img, pts_loc, origImg, None

    def _center_det(self,pt_list, center):

        # Extract x,y points from pt_list
        x = pt_list[:,0]
        y = pt_list[:,1]

        # Determine the min and max width & height values
        # of the points, as if to drow rectangle around arrow
        x_min = x.min()
        y_min = y.min()

        x_max = x.max()
        y_max = y.max()

        # Store height of bounding box
        vert_dst = y_max - y_min

        # Store width of bounding box
        horz_dst = x_max - x_min

        # Compute and store half dimensions of
        # box, will come later when determining
        # arrow direction
        y_half = vert_dst/2 + y_min
        x_half = horz_dst/2 + x_min

        # Store center of the block
        #center = [int(round(x_half)), int(round(y_half))]
        center.append(int(round(x_half)))
        center.append(int(round(y_half)))
        # Estimate radius
```

```
        self.bbox_radius = int(round(math.sqrt((x_max - x_half)**2 + (y_max - y_ha

        return center, self.bbox_radius

    def _bbox_detect(self):
        """
        Reads in an image, and then applies preprocessing functions that mask and
        blur image. Then, attempts to detect corners of an object in frame, and if
        successful it will run further computation to determine the center and rad
        of detected object. Finally adds the draws the bounding circle and center p
        as well as write computed values on original image

        Assigns results to self._travel and saves image showing value of distance
        appended to it in self._img_frame and writes frame to video object self._ou

        Args:
            image (image): Original colored image taken in from camera

        Returns:
            bbox_radius (float): Radius of bounding circle detecting object
            len(pts_loc) (int): The number of corner points identified on object fr
        """
        # Initialize list to save corner points and bbox radius
        bbox_radius = 0
        pts_loc = []
        center = []

        # Take image using camera frame
        self._picam_frame()

        # Prepare image by masking and blurring
        image_blurred = self._prep_image()

        # Extract bbox radius from image
        self._img_frame, center, bbox_radius, self._success = cntr_bbox.bbox_detec

        return self._img_frame, center, bbox_radius, self._success

#         # Detect corners from image
#         img_crnr, pts_loc, image, var = self._corner_detect(image_blurred, self._
#
#         if var is not None:
#
#             # Check if corners are detected
#             if len(pts_loc) > 3:
#                 ## Draw contours over an image, if available
#                 center, bbox_radius = self._center_det(pts_loc, center)
#
#                 block_coordinate = "(" + str(center[0]) + "," + str(center[1]) +
#
#                 # Draw circle ontop of original image
#                 cv2.circle(image, tuple(center), bbox_radius, (0,255,255),2)
#                 cv2.circle(image, tuple(center), 0,(0,0,255),5)
#                 cv2.putText(image,block_coordinate,(0,int(self.cy)),self.font,2,
#
#                 # Draw a cross at center of frame
```

```
#                   cv2.line(image,(self.frame_center[0]-100, self.frame_center[1]),
#                   cv2.line(image,(self.frame_center[0], self.frame_center[1]-100),
#
#                   success = True
#
#                   self._logger.info(f"{self._block_now} block is detected")
#                   time.sleep(1)
#
#                   return image, center, bbox_radius, success
#
#               else:
#                   self._logger.info("Not clearly detected")
#                   success = False
#
#                   return image, center, bbox_radius, success
#
#           else:
#               print("No object of interest in scene")
#               return image, 0, 0, success

        self._logger.info(f"Radius {bbox_radius}")
        self._logger.info(f"Number of corners/edges detected are {len(pts_loc)}")

        #return bbox_radius, center #, len(pts_loc),

    def _pivot(self,deg_diff,just_turn=False,steady = False):
        """
        This function pivots robot to deseired angle

        Args:
            deg_diff (float): Amount of angular rotation to turn.
                      Default value is self._turn parameter

        Returns:
            None

        """
        self._logger.info("Inside pivot function")
        self._task_now = 'rotating'
        self._actions[self._task_now] = True

        # To know the direction of turn. Notation based on camera object detection
        # notation
#         thresh = 0.8
        if deg_diff >= 0:
            self._direction = 'r'
        else:
            self._direction = 'l'
#           thresh = -0.8

        # Initialize change in time for derivative control
        zones = [0.4, 0.8, 1]

        self._yaw = self._imu_serial()
        yaw_start = self._yaw

```

```
        # Angle diff initialized
        yaw_diff0 = deg_diff + 0.001

        yaw_final = yaw_start + deg_diff
        yaw_diff = yaw_final - yaw_start
        yaw_diff_old = yaw_diff

        self._logger.info(f"Needed final yaw is {yaw_final} degrees")
        yaw_current = yaw_start
        # Lower and upper bounds of dutycycle initialized
        lower_dc = 70
        upper_dc = 90
        # Divide the distance traveled into three zones
        trip = [x * yaw_diff for x in zones]
        # Variables for PID control
        derivative = 0
        integral = 0

        start = time.time()
        yaw_diff_list = [0,0]
        yaw_diff_list.append(yaw_diff)
        window_size = 5
        while True:
            try:

                #prcnt = abs(self._imu_serial() - yaw_start)/deg_diff
                prcnt = 1 - yaw_diff/yaw_diff0

                # Adopting moving average
#                 cumsum = np.cumsum(yaw_diff_list)
#                 cumsum[window_size:] = cumsum[window_size:] - cumsum[:-window_si
#                 moving_avg = cumsum[window_size - 1:] / window_size

                if prcnt >= zones[0] and prcnt <= zones[1]:
                    self._dutycycle[self._direction] = self.duty[self._direction]
                elif prcnt > zones[1]:
                    self._dutycycle[self._direction] = self.duty[self._direction]

                # Save robots pose by saving its x, yaw position
                self._path.append((round(self.global_x,2),round(self.global_y,2),r
                self._dutycycle[self._direction] = np.clip(self._dutycycle[self._d

                if abs(yaw_diff) <= 2:
                    self._dutycycle[self._direction] = 65

                elif abs(yaw_diff) >= 100:
                    self._dutycycle[self._direction] = 100



                if abs(yaw_diff) >= 0.8:

                    if ((np.sign(yaw_diff) == np.sign(yaw_diff_old)) or (abs(yaw_d
#                     if (abs(yaw_diff - np.mean(yaw_diff_list[-3:-1])) < 5):
#                     if abs(yaw_diff) >= 0.5:
                        self._logger.info(f"Current yaw {self._imu_serial()}; Angl
```

```
                        if not steady:
                            if self._direction == 'r':
                                self._pivotright(self._dutycycle[self._direction])
                            elif self._direction == 'l':
                                self._pivotleft(self._dutycycle[self._direction])
                        else:
                            if self._direction == 'r':
                                self._pivotright(70)
                            elif self._direction == 'l':
                                self._pivotleft(70)

                else:
                    self._pwmZero()
                    total_turn = yaw_start - self._imu_serial()#self.queue.get()
                    self._logger.info(f"Current yaw {self._imu_serial()}, Total tu
                    self._logger.info("Robot turn done")

                    yaw_diff = round(yaw_diff,2)

                    if not just_turn:
                        if steady:
                        #if (abs(yaw_diff) >= 0.):
                        # Check for overshooting

                            deg_diff = self._scan4object()

                            if deg_diff is not None:

                                if abs(deg_diff) > 0.5:

#                                     time.sleep(1)
                                    # Return pwm dutycycle back to base
                                    self._dutycycle[self._direction] = 70

                                    self._pivot(deg_diff,steady=True)

#                         if abs(yaw_diff) >= 0.6:
#                             self._dutycycle[self._direction] = 70
#
#                             self._pivot(yaw_diff,steady=True)


#                     elif abs(yaw_diff) > 0.6:
#
#                         # Return pwm dutycycle back to base
#                         self._dutycycle[self._direction] = 70
#
#                         self._pivot(yaw_diff)

                    # Return pwm dutycycle back to base
                    self._dutycycle[self._direction] = self.duty[self._direction]

                    # Save robots pose by saving its x, yaw position
                    self._path.append((round(self.global_x,2),round(self.global_y,
```

```
                    self._logger.info("Final orientation reached")
                    break

                yaw_diff_old = yaw_diff
                self._yaw = self._imu_serial()
                yaw_diff = yaw_final - self._yaw
                yaw_diff_list.append(yaw_diff)

#                 if ((np.sign(yaw_diff) != np.sign(yaw_diff_old)) and (abs(yaw_di
#                     yaw_diff_list.pop()
#                     yaw_diff = yaw_diff_old
                # Filtering data via median thresholding
#                 if len(yaw_diff_list) > 6:
#                     median = np.median(yaw_diff_list[-5:-1])
#                 else:
#                     median = np.median(yaw_diff_list[-3:-1])

#                 if (abs(yaw_diff - median) < 5):
#                     yaw_diff_list.append(yaw_diff)
#                 else:
#                     yaw_diff_list.pop()
#                     yaw_diff = yaw_diff_old
#                     yaw_diff_list.append(yaw_diff_old)

            except KeyboardInterrupt:
                self._logger.error("Pivot Interrupted")
                self._gameover()
            except Exception as error:
                self._logger.error(f"Error {error}")

        self._pwmZero()
        self._actions[self._task_now] = False
        self._logger.info("Pivot Finished")

    def _go2block(self, direction='f', eucl_disp=0, just_drive=False, steady=False
        """
        This function drives the robot in a straight path, either forward
        or reverse

        Args:
            direction: Direction of drive, forward or reverse

        Returns:
            None

        """
        self._task_now = 'driving'
        self._actions[self._task_now] = True
        self._logger.info("Going toward block (go2block)")

        # Divide the separation distance into three zones
        # First zone is 70% into drive
        # Second zone is 20%
        # Third zone is the last 10%
        # Each of the zones have their own dutycycle. Robot will
        # drive and stop at each zone, scan, align, and resume drive
```

```
        zones = [0.7, 0.85, 0.9]
        thresh = 20
        self._success = False
        # Check if straight drive mode is requested andd steady drive mode
        if not just_drive and not steady:
            # Use image from camera to estimate distance
            self._dist_estimate()
            # convert depth to encoder count
            orig_dist = self._cm2encoder(self._travel-35)

            self._logger.info(f"Distance estimated {self._travel}")
            self._logger.info(f"Distance to drive {self._travel - 35}")
        elif steady:
            # Use image from camera to estimate distance
            self._dist_estimate()
            # convert depth to encoder count
            orig_dist = self._cm2encoder(self._travel-5)
            self._logger.info(f"Distance estimated {self._travel}")
            self._logger.info(f"Driving slowly {self._travel-5} cms")
        elif just_drive:
            # Drive straight command
            orig_dist = self._feet2encoder(eucl_disp)
            self._logger.info(f"Just Driving Distance estimated {orig_dist}")
        elif just_drive and steady:
            # Drive straight command
            orig_dist = self._cm2encoder(eucl_disp)
            self._logger.info(f"Just Driving Distance estimated {orig_dist}")

        self._logger.info(f"Distance to drive {orig_dist} encoder")
        pos_encoder_orig = 0
        pos_encoder = 0
        orig_error = orig_dist

        self.counterFL = np.uint64(0)
        self.counterBR = np.uint64(0)
        self.buttonBR = int(0)
        self.buttonFL = int(0)

        # Defining the limit of dutycycle
        lower_dc = 70
        upper_dc = 100

        z1 = 80
        z2 = z1 - 20
        z3 = z2 - 10

        # Divide up the path in different zones
        trip = [x * orig_dist for x in zones]

        # Determine if function completed successfully
        self._success = False

        #time.sleep(1)
        yaw0 = self._imu_serial()

        # For encoder level control
```

```
        integral_error = 0
        derivative_error = 0
        error_encoder0 = 0
        error_encoder = 0

        # For achieving overall distance
        integral = 0
        derivative = 0
        error0 = 0
        error = 0
        startD = 0

        cnt = 0
        # Assign direciton of drive from argument
        self._direction = direction
        # Compute the difference in distance between start state  and goal state
        self._logger.info("Sending command toward thrusters")
        self._logger.info(f"Checking encoder to cm {self._encoder2cm(orig_dist)}")

        self._picam_frame()
        self._img_frame = cntr_bbox.dist_img(self._img_frame, self._encoder2cm(ori
        #cntr_bbox.img_show("Detecting Block",self._img_frame)

        start = time.time()

        #To avoid division by zero
        if abs(orig_dist) < 0.001:
            orig_dist = 0.001

        x = 0
        y = 0

        # Save robots pose by saving its x, yaw position
        self._yaw = self._imu_serial()
        # Convert imu notation to general +ccw from positive x
        theta = -self._yaw

        while not self._success:

            # by considering the minimum of the two encoders
            pos_encoder = int(min(self.counterFL,self.counterBR))

            # Global error between start and goal state will be
            error = orig_dist - pos_encoder
            prcnt = round(1 - error/orig_dist,3)

            if (pos_encoder % 150) == 0:

                # Save robots pose by saving its x, yaw position
                x = self._encoder2feet(pos_encoder) * math.cos(math.radians(theta)
                y = self._encoder2feet(pos_encoder) * math.sin(math.radians(theta)

                self._path.append((round(self.global_x + x,2),round(self.global_y

#                 if (self._distance() < 30):
                if ((10 - self.global_x < 1 or self.global_x < 0.9) or ((10 - self
```

```
                self._logger.info("Too close to border. Stop and Exit straight
                self._pwmZero()
                break

            #Record encoder states to txt file
            outstring = str(round(self.global_x + x,2)) + ' ' + str(round(self
            self.f.write(outstring)


        if prcnt < zones[0] and self._direction == 'f' and not steady:
            if cnt == 0:
                lower_dc = 50
                upper_dc = 60

                error = orig_dist - pos_encoder
                pos_encoder_orig = pos_encoder

                self.duty[self._direction] = [55,55]
                self._dutycycle[self._direction] = self.duty[self._direction]
                cnt += 1

                self._picam_frame()
                self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
                self._out.write(self._img_frame)

        elif (prcnt >= zones[0] and prcnt < zones[1]) and self._direction == '
            if cnt == 1:
                lower_dc = 35
                upper_dc = 50

                pos_encoder_orig += pos_encoder

                self.duty[self._direction] = [40,40]
                self._dutycycle[self._direction] = self.duty[self._direction]
                cnt += 1

                self._picam_frame()
                self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
                self._out.write(self._img_frame)

        elif (prcnt >= zones[1] and prcnt < zones[2]) and self._direction == '
            if cnt == 1:
                lower_dc = 28
                upper_dc = 32
                self._pwmZero()
                pos_encoder_orig += pos_encoder

                self.duty[self._direction] = [30,30]
                self._dutycycle[self._direction] = self.duty[self._direction]
                cnt += 1

                self._picam_frame()
                self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
                self._out.write(self._img_frame)
        elif prcnt >= zones[2]:
            if cnt == 2:
```

```
                    lower_dc = 25
                    upper_dc = 28

                    pos_encoder_orig += pos_encoder

                    self.duty[self._direction] = [22,22]
                    self._dutycycle[self._direction] = self.duty[self._direction]
                    cnt += 1

                    self._picam_frame()
                    self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
                    self._out.write(self._img_frame)


            self._dutycycle[self._direction] = [np.clip(self._dutycycle[self._dire
                                                np.clip(self._dutycycle[self._dire

            if steady:
                self._dutycycle[self._direction] = [30,30]

            if self._direction == 'f':
                # Drive robot towards direction
                self._forward(tuple(self._dutycycle[self._direction]))
            elif self._direction == 'rev':
                # Drive robot towards direction
                self._reverse(tuple(self._dutycycle[self._direction]))

            # Count encoders for left and right motors
            if int(gpio.input(12)) != int(self.buttonBR):
                self.buttonBR = int(gpio.input(12))
                self.counterBR += 1

            if int(gpio.input(7)) != int(self.buttonFL):
                self.buttonFL  = int(gpio.input(7))
                self.counterFL += 1

            # PD Controller for encoder balance
            error_encoder = self.counterFL - self.counterBR

            if error_encoder > 25:
                # Give power to corresponding motors
                self._dutycycle[self._direction][0] -=  0.1 * self.duty[self._dire
#                 self._dutycycle[self._direction][1] += 0.1 * self.duty[self._dire
            elif error_encoder < -25:
                # Give power to corresponding motors
#                 self._dutycycle[self._direction][0] += 0.1 * self.duty[self._dire
                self._dutycycle[self._direction][1] -=  0.1 * self.duty[self._dire
            else:
                # Give power to corresponding motors
                self._dutycycle[self._direction][0] = self.duty[self._direction][0
                self._dutycycle[self._direction][1] = self.duty[self._direction][1


            if (error <= thresh) and (direction in ['f','rev']):
                self._yaw = self._imu_serial()
                angle_diff = yaw0 - self._yaw
```

```
                self._logger.info(f"counterBR: {self.counterBR} counterFL: {self.c
                self._logger.info(f"Angle shifted: {angle_diff} deg ")
                self._logger.info(f"Estimated distance traveled: {self._encoder2cm
                self.counterFL = np.uint64(0)
                self.counterBR = np.uint64(0)
                self.buttonBR = int(0)
                self.buttonFL = int(0)
                self._logger.info("Arrived in front of block")
                self._pwmZero()
                self._success = True
                self._actions[self._task_now] = False

                # Convert imu notation to general +ccw from positive x
                theta = -self._yaw

                # Save robots pose by saving its x, yaw position
                x = self._encoder2feet(pos_encoder) * math.cos(math.radians(theta)
                y = self._encoder2feet(pos_encoder) * math.sin(math.radians(theta)
                self.global_x += x
                self.global_y += y
                #self.global_yaw += self._yaw

                self._path.append((round(self.global_x,2),round(self.global_y,2),r

                self._logger.info(f"x value {round(self.global_x,2)} and y value {

                # Take image and save to video object
                self._picam_frame()
                self._img_frame = cntr_bbox.dist_img(self._img_frame, self._encode
                #cntr_bbox.img_show("Detecting Block",self._img_frame)
                self._out.write(self._img_frame)

                #Record encoder states to txt file
                outstring = '#' * 30 + '\n'
                self.f.write(outstring)
                outstring = 'End of Driving {self._direction}' + '\n'
                self.f.write(outstring)
                outstring = '#' * 30 + '\n'
                self.f.write(outstring)
                break

        # Return pwm dutcycle back to base
        self._dutycycle[self._direction] = self.duty[self._direction]

    def _scan4object(self):
        """
        This function scans for block in environment by turning in place
        until it finds the block. If it doesn't find the block after full 360
        degree turn, it returns false to say not found

        Args:
            None

        Returns:
            None
        """
```

```
        self._task_now = 'scanning'
        self._actions[self._task_now] = True
        self._logger.info(f"Scanning for {self._block_now} block started")

        scan_turn = 0

        while True:
            # Detect object in frame

            self._picam_frame()
            #self._img_frame, ave_center, bbox_radius, success = self._bbox_detect
            self._img_frame, ave_center, bbox_radius, success = cntr_bbox.bbox_det

            if success not in [None,False]:
                # Object found
                diff = ave_center[0]-self.frame_center[0]
                # Convert the pixel difference to degrees
                self._turn = diff * cntr_bbox.pixel2deg

                if (self._turn > 180 and self._turn <=360):
                    self._turn = self._turn - 360

                self._yaw = -self._turn
                self._path.append((round(self.global_x,2),round(self.global_y,2),r

                self._success = True
                self._status['found'] = True
                self._task_now = 'scanning'
                self._actions[self._task_now] = False
                return self._turn

            else:
                # Pivot the robot to a predefined turn angle
                self._pivot(self.turn)
                # Read the distance from obstacle for reference
                self._distance()
                # Keep track of turn
                #self._yaw += self.turn
                scan_turn += self.turn
                self._yaw += -self._turn
                self._path.append((round(self.global_x,2),round(self.global_y,2),r

#             if scan_turn >= 90:
#                 # Scan the other side
#                 self._pivot(-self.turn)
#                 # Add scan angle
#                 scan_turn += -self.turn
#                 self._yaw += self._turn
#                 self._path.append((round(self.global_x,2),round(self.global_y,2),
#
#             if scan_turn <= -90:
#                 self._status['found'] = False
#                 self._success = False
#                 self._task_now = 'scanning'
#                 self._actions[self._task_now] = False
```

```
#                 self._logger.warning("No objects in nearby vicinity")
#                 return None

            if abs(scan_turn) >= 360:
                self._status['found'] = False
                self._success = False
                self._task_now = 'scanning'
                self._actions[self._task_now] = False
                self._logger.warning("No objects in nearby vicinity")
                return None

    def _pick_up(self):
        """
        This function drives the robot to steadily approach and picks up toy block

        Args:
            None

        Returns:
            None
        """
        self._success = False
        self._task_now = 'picking'
        self._actions[self._task_now] = True

        # Take image and save to video object
        self._picam_frame()
        #cntr_bbox.img_show("Detecting Block",self._img_frame)
        self._out.write(self._img_frame)

        while not self._success:
            # Open gripper
            self._servo_cntrl(self.grip_state['opened'])
            self._gripper_state = self.grip_state['opened']
            self._logger.info("Servo Opened")
            # Take image and save to video object
            self._picam_frame()
            #cntr_bbox.img_show("Detecting Block",self._img_frame)
            self._out.write(self._img_frame)
            # Align robot fine turne rotation
            degree = self._scan4object()

#             while True:
#                 if degree is not None:
#                     if abs(degree) > 0.6:
#                         self._pivot(degree, steady=True)
#                     else:
#                         break
#                 else:
#                     break
            # Scan block and drive slowly towards it
            self._logger.info("Driving slowly toward block")
            self._go2block(steady=True)
            self._logger.info("Slow drive completed")

            # Take image and save to video object
```

```
            self._picam_frame()
            #cntr_bbox.img_show("Detecting Block",self._img_frame)
            self._out.write(self._img_frame)

            # Close gripper
            self._servo_cntrl(self.grip_state['closed'])
            self._gripper_state = self.grip_state['closed']
            self._logger.info("Servo Closed")

            # Take image and save to video object
            self._picam_frame()
            #cntr_bbox.img_show("Detecting Block",self._img_frame)
            self._out.write(self._img_frame)

            #if (abs(radius2 - radius1) < 5) or (abs(center2[0] - center1[0]) < 3)
            self._logger.info(f"The {self._block_now} block has been gripped succe
            self._success = True
            self._status['picked_up'] = True
            self._actions[self._task_now] = False

        # Read saved image and send via email
        self._email_media()

    def _go2(self,destination=(0,0),area=''):
        """
        This function computes the difference in pose between the robot and the de
        then estimates the displacement vector along with the angle of rotation ne
        to align the robot head  on. It then calls the pivot and go2block function
        and command the robot to enter the construction zone.

        Args:
            destination (tuple): contans x,y location of destination
            area (str): Describes where that destination is

        Returns:
            None
        """
        self._logger.info(f"Initiating travel to {area}")
        self._success = False
        self._task_now = 'returning'
        self._actions[self._task_now] = False

        # Read last position from the appended path
        # self.global_x = self._path[-1][0]
        # self.global_y = self._path[-1][1]
        #self.global_yaw = self._path[-1][2]
        delta_x = destination[0]-self.global_x
        delta_y = destination[1]-self.global_y
        self._yaw = self._imu_serial()

        self._logger.info(f"Pose Now: X- {self.global_x} and Y- {self.global_y} an
        # Compute the euclidean distance between current robot location and constr
        disp = math.sqrt((delta_x)**2 + (delta_y)**2)
        # Calculate the yaw angle difference between these two points using slope
        rad_diff = math.atan2(delta_y,delta_x)
        # Convert from radians to angle
```

```
        target_angle = math.degrees(rad_diff)
        # Ensure angle is between [0, 360]
        target_angle %= 360

        self._logger.info(f"Computed target angle is {target_angle}")

        # # Place angle from [0,360] to [-180,0) U [0,180]
        if target_angle >=-180 and target_angle <=180:
            target_angle = -target_angle
        elif (target_angle >= 180 and target_angle <360):
            target_angle = 360 - target_angle
        elif (target_angle > -360 and target_angle <=-180):
            target_angle = -(target_angle + 360)

        self._logger.info(f"Normalized target angle is {target_angle}")
#         # Normalize angle difference to range [-180, 180) degrees
#         target_angle = (target_angle + 180) % 360 - 180
        # # Angle difference from current and target
        # deg_diff = self._yaw - target_angle
        # deg_diff %= 360

        # Based on the quadrant of the point heading towards normalize
        # the target angle
#         if delta_x < 0:
#             if delta_y > 0:
#                 #target_angle = 180 - target_angle
#                 pass
#             else:
# #                 target_angle = target_angle - 180
#                 pass
#         else:
#             if delta_y < 0:
# #                 target_angle = -target_angle
#                 pass

        # convert target angle with respect to imu configuration
        # target_angle = -target_angle

        # Angle difference from current and target
        deg_diff = target_angle - self._yaw
        #deg_diff %= 360

#         if (deg_diff > 180 and deg_diff <360):
#             deg_diff = (deg_diff - 360) # Turning to the right
#         elif (deg_diff > -360 and deg_diff <=-180):
#             deg_diff = (deg_diff + 360) # Turning to the left

#         # Place angle from [0,360] to [-180,0) U [0,180]
#         if (deg_diff > 180 and deg_diff <=360):
#             deg_diff = deg_diff - 360
#         # Normalize angle difference to range [-180, 180) degrees
#         deg_diff = (deg_diff + 180) % 360 - 180

        self._logger.info(f"Yaw that vehicle will pivot to {deg_diff}")
        # Pivot robot to face construction zone
        self._pivot(deg_diff, just_turn=True)
```

```
        self._logger.info(f"Started driving straight to {area}")
        # Drive robot foward to construction zone
        self._go2block(eucl_disp = disp,just_drive = True)
        self._logger.info(f"Arrived at {area}")

        # New global yaw
        self.global_yaw = self._imu_serial()

        self._success = True
        self._status['arrived'] = True
        self._actions[self._task_now] = False

    def _place(self):
        """
        This function places toy block

        Args:
            None

        Returns:
            None
        """
        self._success = False
        self._task_now = 'placing'
        self._actions[self._task_now] = True

        # Open gripper
        self._servo_cntrl(self.grip_state['opened'])
        self._gripper_state = self.grip_state['opened']
        self._logger.info(f"Gripper Opened to place {self._block_now} block")

        # Drive in reverse
        self._go2block(direction='rev',eucl_disp=7.5, just_drive=True, steady=True
        time.sleep(1)
        self._logger.info("Backed away from toy block")

        # Close gripper
        self._servo_cntrl(self.grip_state['closed'])
        self._gripper_state = self.grip_state['closed']
        self._logger.info("Gripper Closed to prepare for next order")

        self._success = True
        self._status['placed'] = True
        self._actions[self._task_now] = False

        self._logger.info(f"{self._block_now} block is now placed inside construct

        # Read saved image and send via email
        self._email_media()

    def _localize(self):
        """
        This function uses distance sensor to read from borderwall
        so that the robot's exact position in arena can easily be
        identified

```

```
        Args:
            None

        Returns:
            None
        """
        self._success = False
        self._task_now = 'localizing'
        self._actions[self._task_now] = True

        # pivot the robot to positive y first and read from distance sensor
        yaw_final = -90
        yaw_current = self._imu_serial()
        deg_diff = yaw_final - yaw_current

        #deg_diff %= 360
        # Normalize angle difference to range [-180, 180) degrees
        if (deg_diff > 180 and deg_diff <360):
            deg_diff = 360 - deg_diff
        elif (deg_diff > -360 and deg_diff <= -180):
            deg_diff = -(deg_diff + 360)

        # Turn towards borderwall
        self._pivot(deg_diff, just_turn=True)

        # Now read distance sensor value
        self._distance()
        delta_ft = self._cm2feet(self._depth)
        self._yaw = self._imu_serial()
        theta = -self._yaw
        x1 = delta_ft * math.cos(math.radians(theta))
        y1 = delta_ft * math.sin(math.radians(theta))

        # Extract y pose by normalizing from border wall
        if self.global_y > 5:
            self.global_y = 10 - y1
        else:
            self.global_y = y1

        # pivot the robot to negative x first and read from distance sensor
        yaw_final = -179
        yaw_current = self._imu_serial()
        deg_diff = yaw_final - yaw_current
        #deg_diff %= 360

        # Normalize angle difference to range [-180, 180) degrees
        if (deg_diff > 180 and deg_diff <360):
            deg_diff = 360 - deg_diff
        elif (deg_diff > -360 and deg_diff <= -180):
            deg_diff = -(deg_diff + 360)

        # Turn towards borderwall
        self._pivot(deg_diff, just_turn=True)

        # Now read distance sensor value
        self._distance()
```

```
        delta_ft = self._cm2feet(self._depth)
        self._yaw = self._imu_serial()
        theta = -self._yaw
        x2 = delta_ft * math.cos(math.radians(theta))
        y2 = delta_ft * math.sin(math.radians(theta))

        # Extract x pose by normalizing from border wall
        if self._x > 5:
            self.global_x = 10 - x2
        else:
            self.global_x = x2

        # Add detected pose of robot into path list
        self._path.append((round(self.global_x,2),round(self.global_y,2),round(sel

        self._success = True
        self._actions[self._task_now] = False
        self._logger.info(f"Localization complete")

    def _go2site(self):
        """
        This function commands the robot to drive back to the center
        of the arena or near the cluttered enviorment to resume delivering
        order

        Args:
            None

        Returns:
            None
        """

        self._logger.info("Initiating travel to cluttered environment")
        # Compute the euclidean distance between current robot location and constru
        disp = math.sqrt((self._x - self.clutter_env[0])**2 + (self._y - self.clut
        # Calculate the yaw angle difference between these two points using slope
        rad_diff = math.atan2(self.clutter_env[0] - self._x, self.clutter_env[1] -
        # Convert from radians to angle
        deg_diff = math.degrees(rad_diff)
        # Ensure angle is between [0, 360]
        deg_diff %= 360
        # Place angle from [0,360] to [-180,0) U [0,180]
        if (deg_diff > 180 and deg_diff <=360):
            deg_diff = deg_diff - 360

        self._logger.info("Orienting to cluttered environment")
        # Pivot robot to face construction zone
        self._pivot(deg_diff)
        self._logger.info("Started driving straight to cluttered environment")
        # Drive robot foward to construction zone
        self._go2block(eucl_disp = disp,just_drive = True)
        self._logger.info("Arrived at cluttered environment")

    def _go_home(self):
        """
        This function commands the robot to drive back to its home location
```

```
        marking the end of the challenge.

        Args:
            None

        Returns:
            None
        """
        self._logger.info("Initiating travel to home base")
        # Compute the euclidean distance between current robot location and constru
        disp = math.sqrt((self._x - self.start_pt[0])**2 + (self._y - self.start_p
        # Calculate the yaw angle difference between these two points using slope e
        rad_diff = math.atan2(self.start_pt[0] - self._x, self.start_pt[1] - self._
        # Convert from radians to angle
        deg_diff = math.degrees(rad_diff)
        # Ensure angle is between [0, 360]
        deg_diff %= 360
        # Place angle from [0,360] to [-180,0) U [0,180]
        if (deg_diff > 180 and deg_diff <=360):
            deg_diff = deg_diff - 360

        self._logger.info("Orienting to home base")
        # Pivot robot to face construction zone
        self._pivot(deg_diff)
        self._logger.info("Started driving straight to home base")
        # Drive robot foward to construction zone
        self._go2block(eucl_disp = disp,just_drive = True)
        self._logger.info("Arrived at home")
        self._logger.info("Thank you for playing game.")
        self._logger.info("Powering off.")
        self._end_game = True

    def plot_path(self):
        """
        Plot the path of the robot during the grand challenge.

        Args:
            None
        Return:
            None
        """
        # Take in the path list as an array 2D NumPy array
        path = np.array(self._path)
        x = path[:,0]
        y = path[:,1]

        # Create figure and plot
        fig, ax = plt.subplots(1,1)

        fig.suptitle('Motor Encoder Analysis')

        ax.plot(x,y,ls='solid', color='blue',linewidth=2, label=f'{self._name} Pat

        # 3) Label the axes, title, legend
        ax.set_ylabel('Y')
        ax.set_xlabel('X')
```

```
        ax.set_xlim(0,10)
        ax.set_ylim(0,10)
        plt.savefig('grandChallenge-robotPath.png')
        plt.show()
        plt.close()

    def start(self, order: List):
        """
        The sequence of actions occuring while baron robot works to pick and place
        all toy blocks to the construction zone. This function lists the steps need
        for the robot to complete the orders passed to it at the beginning of compe

        Args:
            order: List[Blocks] list of all toy blocks needed to complete challenge

        Returns:
            None
        """

        try:
            # Assign order from object initialization
            self._order = order

            # First wait until startup QR Code has been read
            while True:
                if self._wait2start():
                    break

            while len(self._order) > 0:
                """While Robot is still active, resume challenge"""
                if (not self._end_game):

                    self._block_now = self._order.pop(0)
                    self._order_no +=1

                    # Begin challenge with choesn block
                    self._logger.info("-" * 20)
                    self._logger.info(f"*** Picking Up {self._block_now} block")
                    self._logger.info("-" * 20)

                    # Track the number of times robot drives to search area
                    search = 0

                    # Drive robot to center of arena, get view of toy blocks and be
                    # extracting blocks
                    self._go2(destination=self.clutter_env,area='cluttered environ

                    while True:

                        # Begin by scanning for toy block object
                        degree = self._scan4object()

                        # Take image and save to video object
                        #cntr_bbox.img_show("Detecting Block",self._img_frame)
                        self._out.write(self._img_frame)

```

```
            # Drive toward block if successful, else pivot to
            # a free space and drive
            if self._success:
              #self._logger(f"Pivoting Robot towards {self._block_no
              self._pivot(degree,steady=True)

              # Take image and save to video object
              self._picam_frame()
              __, __, __,__= cntr_bbox.bbox_detect(self._img_frame, 
              #cntr_bbox.img_show("Detecting Block",self._img_frame)
              self._out.write(self._img_frame)

              self._logger.info(f"Drive Robot forward to {self._bloc
              self._go2block()

              # # Update global x and y
              # self.global_x += self._x
              # self.global_y += self._y

              ##self._logger(f"{self._name} is now at x- {self.globa

              # Take image and save to video object
              self._picam_frame()
              #cntr_bbox.img_show("Arrived at Block",self._img_frame,
              self._out.write(self._img_frame)

              ##self._logger(f"Pick up {self._block_now} block with 
              self._pick_up()

              # Drive robot to construction zone to place block
              ##self._logger(f"Take {self._block_now} block to const
              self._go2(destination=self.const_zone,area='constructi

              #self._logger(f"Place {self._block_now} block inside c
              self._place()

#              #self._logger("Localizing to get accurate location o
              self._localize()

              ##self._logger(f"Pick up and place the next order in t
              self._success = True
              break
            else:
              #self._logger("No toy blocks found in current pose. Ch
              #self._logger("try again.")

              # Drive robot to the predetermined survey locations
              if search > 1:
                self._go2(destination=self._survey_loc[search],area
              else:
                #self._logger(f"Robot can't find {self._block_now}
                #self._logger("In pursuit of time, moving on to ne
                break
              search += 1

      else:
```

```
                #self._logger(f"Mission complete. Take robot back to home base")
                self._go2(destination=self.start_pt,area='home base')
                return

        except KeyboardInterrupt as error:
            self._logger.error("Error occurred: %s",error)
            self._gameover()
            time.sleep(1)
            return None
        except ValueError as error:
            self._logger.error("Error %s", error, exc_info=True)
            self._gameover()
            time.sleep(1)
            return None
```

A class defining basic attributes of Baron robot that will be autonomously achieving objective

Attributes:

- **name (str):** Name of robot
- **_x (float):** horizontal position of robot from start postion
- **_y (float):** vertical position of robot from start position
- **_yaw (float):** yaw orientation from +x as horizontal [0,180] or (-180,0)
- **_travel(float):** estimating the distance to drive and reach block
- **_turn (float):** keeps the next turn needed by block
- **_depth (float):** measures distance of obstacle in front of robot from image
- **_path (list):** stores the path traveled by robot as list of (x,y) points and yaw
- **_pwms (list):** stores a list of pwms for the robot motors
- **_dutycycle (dict):** holds the baseline pwm dutycycle for forward,reverse, pivotleft and pivotright
- **_success (bool):** keeps track whether a given task is successful or not
- **_gripper_state (str):** stores gripper duty cycle state as opened or closed
- **_order(list):** List of blocks to pickup
- **_end_game (bool):** tells competition is over by victory or failure
- **_order_no (int):** keeps track of number of orders completed
- **_block_now (str):** the block being pursued now
- **_task_now (str):** keeps track of the task being completed at the moment
- **_actions (dict):** holding list of all service actions robot performs with bool on which process is being done at a given time
- **_status (dict):** a dictionary keeping track of completed actions along with
- **_img_frame (img):** saves current image frame
- **_out (cv2 video obj):** gather all frames into a video
- **_logger (log):** log statements for debuging and status over the terminal
- **_kp (float):** proportional gain
- **_ki (float):** integral gain

- **_kd (float):** derivative gain
- **_direction (str):** tells weather robot is moving forward or reverse
- **_event (event):** event object to end the thread call
- **_pause_event (event):** event object to pause and resume thread
- **imu_reader (thread):** threading object to read from imu during forward run
- **_ser (serial):** object reading serial input streams from imu sensor
- **_img_saved (img):** keeping track of saved image to send in an email
- **_survey_loc (List):** containes x,y locations for the robot to navigate and scan so that to extract toy blocks for mission

Robot(name: str) ▼ View Source

```
    def __init__(self, name: str):
        """
        Initialize Robot attributes

        Args:
            name: str - takes name of robot

        Returns:
            None
        """

        self.name = name
        self._x = self.start_pt[0]
        self._y = self.start_pt[1]
        self._yaw = 0.0
        self._travel = 0
        self._turn = 0
        self._depth = 0
        self._path = [(1,1,0)]
        self._pwms = []
        self._dutycycle = self.duty
        self._success = False
        self._gripper_state = self.grip_state['closed']
        self._order = []
        self._end_game = False
        self._order_no = 0
        self._block_now = ''
        self._task_now = ''
        self._actions = dict([('driving',False),('rotating',False),('picking',F
        self._status = dict([('started',False),('picked_up',False),('found',Fal
        self._img_frame = self.image
        self._out = cv2.VideoWriter('grand_chall_vid.avi', self.fourcc, 3, (640
        self._logger = logging.getLogger(name)
        self._kp = self.Kp
        self._ki = self.Ki
        self._kd = self.Kd
        self._direction = 'f'
        self._event = threading.Event()
        self._pause_event = threading.Event()
```

```
        self.imu_reader = threading.Thread(target = self._imu_serial_straight,
        self._ser = serial.Serial('/dev/ttyUSB0', 19200)
        self._img_saved = self.image
        self._survey_loc = self.scan_spots

        # Set up logger with Debug initially
        self._logger.setLevel(logging.DEBUG)
        # Create console handler and set level to debug
        console_handler = logging.StreamHandler()
        console_handler.setLevel(logging.DEBUG)
        # log terminal messages to a file
        file_handler = logging.FileHandler('gc_log_3.log')
        file_handler.setLevel(logging.INFO)
        # Create formatter
        formatter = logging.Formatter('%(name)s - %(levelname)s - %(message)s')
        # Add formatter to handler
        console_handler.setFormatter(formatter)
        file_handler.setFormatter(formatter)
        # Add console handler to logger
        self._logger.addHandler(console_handler)
        self._logger.addHandler(file_handler)
        # Initialize gpios and pwms
        self._init()
```

Initialize Robot attributes

Arguments:

- **name:** str - takes name of robot

Returns:

None

def plot_path(self):

▼ View Source

```
    def plot_path(self):
        """
        Plot the path of the robot during the grand challenge.

        Args:
            None
        Return:
            None
        """
        # Take in the path list as an array 2D NumPy array
        path = np.array(self._path)
        x = path[:,0]
        y = path[:,1]

        # Create figure and plot
        fig, ax = plt.subplots(1,1)

        fig.suptitle('Motor Encoder Analysis')
```

```
        ax.plot(x,y,ls='solid', color='blue',linewidth=2, label=f'{self._name}

        # 3) Label the axes, title, legend
        ax.set_ylabel('Y')
        ax.set_xlabel('X')
        ax.set_xlim(0,10)
        ax.set_ylim(0,10)
        plt.savefig('grandChallenge-robotPath.png')
        plt.show()
```

Plot the path of the robot during the grand challenge.

Arguments:

- None

Return:

None

def start(self, order: List):

▼ View Source

```
    def start(self, order: List):
        """
        The sequence of actions occuring while baron robot works to pick and p
        all toy blocks to the construction zone. This function lists the steps
        for the robot to complete the orders passed to it at the beginning of

        Args:
            order: List[Blocks] list of all toy blocks needed to complete chal

        Returns:
            None
        """

        try:
            # Assign order from object initialization
            self._order = order

            # First wait until startup QR Code has been read
            while True:
                if self._wait2start():
                    break

            while len(self._order) > 0:
                """While Robot is still active, resume challenge"""
                if (not self._end_game):

                    self._block_now = self._order.pop(0)
                    self._order_no +=1

                    # Begin challenge with choesn block
                    self._logger.info("-" * 20)
```

```
                    self._logger.info(f"*** Picking Up {self._block_now} block
                    self._logger.info("-" * 20)

                    # Track the number of times robot drives to search area
                    search = 0

                    # Drive robot to center of arena, get view of toy blocks a
                    # extracting blocks
                    self._go2(destination=self.clutter_env,area='cluttered env

                    while True:

                        # Begin by scanning for toy block object
                        degree = self._scan4object()

                        # Take image and save to video object
                        #cntr_bbox.img_show("Detecting Block",self._img_frame)
                        self._out.write(self._img_frame)

                        # Drive toward block if successful, else pivot to
                        # a free space and drive
                        if self._success:
                            #self._logger(f"Pivoting Robot towards {self._bloc
                            self._pivot(degree,steady=True)

                            # Take image and save to video object
                            self._picam_frame()
                            __, __, __,__= cntr_bbox.bbox_detect(self._img_fra
                            #cntr_bbox.img_show("Detecting Block",self._img_fr
                            self._out.write(self._img_frame)

                            self._logger.info(f"Drive Robot forward to {self._
                            self._go2block()

                            # # Update global x and y
                            # self.global_x += self._x
                            # self.global_y += self._y

                            ##self._logger(f"{self._name} is now at x- {self.g

                            # Take image and save to video object
                            self._picam_frame()
                            #cntr_bbox.img_show("Arrived at Block",self._img_f
                            self._out.write(self._img_frame)

                            ##self._logger(f"Pick up {self._block_now} block w
                            self._pick_up()

                            # Drive robot to construction zone to place block
                            ##self._logger(f"Take {self._block_now} block to c
                            self._go2(destination=self.const_zone,area='constr

                            #self._logger(f"Place {self._block_now} block insi
                            self._place()

#                             #self._logger("Localizing to get accurate locati
```

```
                            self._localize()

                            ##self._logger(f"Pick up and place the next order
                            self._success = True
                            break
                        else:
                            #self._logger("No toy blocks found in current pose
                            #self._logger("try again.")

                            # Drive robot to the predetermined survey location
                            if search > 1:
                                self._go2(destination=self._survey_loc[search]
                            else:
                                #self._logger(f"Robot can't find {self._block_
                                #self._logger("In pursuit of time, moving on t
                                break
                            search += 1

            else:
                #self._logger(f"Mission complete. Take robot back to home base
                self._go2(destination=self.start_pt,area='home base')
                return

        except KeyboardInterrupt as error:
            self._logger.error("Error occurred: %s",error)
            self._gameover()
            time.sleep(1)
            return None
        except ValueError as error:
            self._logger.error("Error %s", error, exc_info=True)
            self._gameover()
            time.sleep(1)
            return None
```

The sequence of actions occuring while baron robot works to pick and place all toy blocks to the construction zone. This function lists the steps needed for the robot to complete the orders passed to it at the beginning of competition

Arguments:

- **order:** List[Blocks] list of all toy blocks needed to complete challenge

Returns:

None

grand_challenge_submission.Perception .cntr_bbox

▼ View Source

```
import cv2
import os
import imutils
import numpy as np
import matplotlib.pyplot as plot
import math
import picamera
from picamera.array import PiRGBArray
from picamera import PiCamera
import time
#from Locomotion.drive_robot import pivot, imu_serial,pwmZero,pwms

import smtplib
from smtplib import SMTP
from smtplib import SMTPException
import email
from email.mime.image import MIMEImage
from email.mime.text import MIMEText
from email.mime.multipart import MIMEMultipart
from datetime import datetime
import imaplib

# initialize the Raspberry Pi camera
camera = PiCamera()
camera.resolution = (640, 480)
camera.vflip = True
camera.framerate = 25
rawCapture = PiRGBArray(camera, size=(640,480))
time.sleep(2)

font = cv2.FONT_HERSHEY_COMPLEX_SMALL

# # Read original RGB Image from library
# block = cv2.imread("greenBlock.png")
pics = []

ave_center = [0,0]
bbox_radius = 0

frame_center = (320,240)

(cx,cy) = (frame_center[0]/4.5, frame_center[1]/8)

# define the codec and create VideoWriter object
fourcc = cv2.VideoWriter_fourcc(*'XVID')
out = cv2.VideoWriter('blockRetrieval.avi', fourcc, 3, (640, 480))

```

```python
# Open .txt file to save data
f = open('pxlradius_distance-02.txt','w')

image = np.empty((480*640*3,),dtype=np.uint8)

detected = False

obj = ''

pixel2deg = 0.061

# Initialize servo gripper states
closed = 2.5
half = 5
open_full = 7.5

# # create object to read camera
# video = cv2.VideoCapture(0)
#
# if (video.isOpened() == False):
#     print("Error reading video")

# def cv2_cam_frame():
#     # Pick image from video stream
#     success, image = video.read()
#
#     return image

def picam_frame():
    global image

    # Take picture with camera
    camera.capture (image, format="bgr")
    image = image.reshape((480,640,3))

    image = cv2.flip(image,1)


    return image

def img_show(name, img):
    cv2.imshow(name,img)
#     key = cv2.waitKey(1) & 0xFF
    cv2.waitKey(0)
    # Write frame to video
    # out.write
    cv2.destroyAllWindows()

def dist_img(img,dist):

    dist = str(dist) + "cm"
    data = f"{obj} Sodar: " + dist

    cv2.putText(img, data, (355,30),cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,0),2)

    return img
```

```

def servo_img(grip_state):

    img = picam_frame()
    data = str(grip_state)
    data = "Duty Cycle: " + data + "%"

    cv2.putText(img, data, (20,30),cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,0),2)

    if grip_state == half:
        cv2.putText(img, "Half-Opened", (20,55),cv2.FONT_HERSHEY_SIMPLEX,1,(0,125,1
    elif grip_state == open_full:
        cv2.putText(img, "Fully-Opened", (20,55),cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,25
    elif grip_state == closed:
        cv2.putText(img, "Fully-Closed", (20,55),cv2.FONT_HERSHEY_SIMPLEX,1,(255,0,

    return img

def mask_color(image, imageHSV):
    global obj
    # HSV bounds
    # minHSV = np.array([41,74,163])
    # maxHSV = np.array([63,159,255])
#     # Trail Green Bock - LAB
#     minHSV = np.array([47,56,172])
#     maxHSV = np.array([255,255,255])

    if obj == 'green':
        # Trail Green Bock - LAB
        minHSV = np.array([46,88,121])
        maxHSV = np.array([65,141,255])
    elif obj == 'red':
        # Trail Red block - LAB
        minHSV = np.array([151,107,147])
        maxHSV = np.array([255,255,255])
    else:
        # Trail Blue block - LAB
        minHSV = np.array([72,94,97])
        maxHSV = np.array([126,188,212])

#     # Mask the gripper
#     mask_gripper = np.zeros(image.shape[:2], np.uint8)
#     mask_gripper[0:0, 380:0] = 255
#
#     # Cut out gripper
#     mask_img = cv2.bitwise_and(image, image, mask = mask_gripper)
    maskHSV = cv2.inRange(imageHSV, minHSV, maxHSV)

    return maskHSV

def prep_image(image):

     # Convert image from BGR to HSV space
    imageHSV = cv2.cvtColor(image,cv2.COLOR_BGR2HSV)

    # mask the green light from HSV and convert to grayscale
```

```
    mask = mask_color(image, imageHSV)

    # Mask HSV masked image of arrow
    blurred = cv2.GaussianBlur(mask,(11,11), 0)

    return blurred

def corner_detect(img,origImg):

    # Create a list to store the x,y location of points
    pts_loc = []

    # Detect corners from image
    corners = cv2.goodFeaturesToTrack(img,5,0.01,10)

    if corners is not None:

        corners = np.int0(corners)

        # identify location of corners in image
        for i in corners:
            # Extract x,y coordinate of points
            x,y = i.ravel()

            pts_loc.append([x,y])

            # Draw circle ontop of original image
            cv2.circle(origImg, (x,y), 3, (0,0,255),-1)

        # Create a column vector from pts list
        pts_loc = np.array(pts_loc)

        return img, pts_loc, origImg, True
    else:
        return img, pts_loc, origImg, None

def center_det(pt_list, center):
    global bbox_radius
    # Extract x,y points from pt_list
    x = pt_list[:,0]
    y = pt_list[:,1]

    # Determine the min and max width & height values
    # of the points, as if to drow rectangle around arrow
    x_min = x.min()
    y_min = y.min()

    x_max = x.max()
    y_max = y.max()

    # Store height of bounding box
    vert_dst = y_max - y_min

    # Store width of bounding box
    horz_dst = x_max - x_min

```

```
    # Compute and store half dimensions of
    # box, will come later when determining
    # arrow direction
    y_half = vert_dst/2 + y_min
    x_half = horz_dst/2 + x_min

    # Store center of the block
    #center = [int(round(x_half)), int(round(y_half))]
    center.append(int(round(x_half)))
    center.append(int(round(y_half)))
    # Estimate radius
    bbox_radius = int(round(math.sqrt((x_max - x_half)**2 + (y_max - y_half)**2)))

    return center, bbox_radius


def bbox_detect(image, image_blurred):
    global bbox_radius
    global obj

    center = []
    radius = 0
    success = None

#     # Extract frame from camera
#     image = picam_frame()
#
#     # Prep image for object detection
#     img_blurred = prep_image(image)

    # Detect corners from image
    img_crnr, pts_loc, image, var = corner_detect(image_blurred, image)

    if var is not None:

        # Check if corners are detected
        if len(pts_loc) > 3:
            ## Draw contours over an image, if available
            center, bbox_radius = center_det(pts_loc, center)

            block_coordinate = "(" + str(center[0]) + "," + str(center[1]) + ")"

            # Draw circle ontop of original image
            cv2.circle(image, tuple(center), bbox_radius, (0,255,255),2)
            cv2.circle(image, tuple(center), 0,(0,0,255),5)
            cv2.putText(image,block_coordinate,(0,int(cy)),font,2,(0,0,0),2)

            # Draw a cross at center of frame
            cv2.line(image,(frame_center[0]-100, frame_center[1]),(frame_center[0]+
            cv2.line(image,(frame_center[0], frame_center[1]-100),(frame_center[0],

            success = True

            print(f"{obj} block is detected")
            time.sleep(1)

```

```
            return image, center, bbox_radius, success

        else:
            print("Not clearly detected")
            success = False

            return image, center, bbox_radius, success

    else:
        print("No object of interest in scene")
        return image, 0, 0, success

def scanObject(color):
    global ave_center
    global obj
    global bbox_radius

    detected = False
    obj = color
    # Rotate every small angle, take a picture to detect existence of toy block
    stat = 0
    turn_angle = 15
    deg_diff = 0
    yaw = imu_serial()
    yaw_orig = yaw

    while not detected:

        image, ave_center, bbox_radius, success = bbox_detect()

        # write frame into file
#         out.write(image)
        if success:
            diff = ave_center[0]-frame_center[0]
            # Convert the pixel difference to degrees
            deg_diff = diff * pixel2deg

            detected = True

            return detected, deg_diff, image

        else:
            # Turn the robot by 15 degrees
            #stat += turn_angle

            return detected, turn_angle, image

            yaw = pivot(deg_diff= turn_angle)
            time.sleep(1)


        if stat >=360:
            print("Could not detect block from area")
            print("Drive to a different location and try again")
            break

```

```
    else:
        print("Desired object has been located")

    return detected, deg_diff, image

def dist_estimate():
    # Read in pixel radius from image
    global bbox_radius

    img, __, bbox_radius, __ = bbox_detect()

    # Plug in pixel size into depth equation
    distance = (0.0099 * bbox_radius**2) - (1.8846*bbox_radius) + 103.47

    # Append image of estimated distance on image frame
    image = dist_img(img,round(distance,2))

    return distance, image

def email_media(obj):

    image = picam_frame()

    # send email to user with images
    smtpUser = 'ykebede2@terpmail.umd.edu'
    smtpPass = 'QwE@$d1219'

    toAdd = ['ENPM809TS19@gmail.com','yosephcollege@gmail.com']
    fromAdd = smtpUser

    f_time = datetime.now().strftime('%a %d %b @ %H:%M')
    subject = f'{obj} block picked up for transportation to construction area: ' +

    msg = MIMEMultipart()
    msg['Subject'] = subject
    msg['From'] = fromAdd
    msg['To'] = ",".join(toAdd)

    msg.preamble = "Image @ " + f_time

    body = email.mime.text.MIMEText("Baron Robot image: " + f_time)
    msg.attach(body)

    fp = open(sodar_img, 'rb')
    img = MIMEImage(fp.read())
    fp.close()
    msg.attach(img)

    fp = open(grip_img, 'rb')
    img = MIMEImage(fp.read())
    fp.close()
    msg.attach(img)

    s = smtplib.SMTP('smtp.gmail.com', 587)
    s.ehlo()
    s.starttls()
```

```
    s.ehlo()
    s.login(smtpUser, smtpPass)
    s.sendmail(fromAdd, toAdd, msg.as_string())
    s.quit()

    print("Email delivered!")

# if __name__ == "__main__":
#
#     global obj
#     i = 20
#     block = ['g','b','r']
#
#     while True:
#
# #         obj = block.pop(0)
#         obj = 'g'
#
#         detected, deg_diff = detectObject()
#
# #         # Read image from picamera
# #         image = picam_frame()
# #
# #         # to accomodate for pi camera mount
# #         image = cv2.flip(image, 1)
# #
# #         # Draw bbox
# #         radius = bbox_detect(image)
#
#         outstring = str(i) + ',' + str(radius) + '\n'
#         f.write(outstring)
#
#         i += 10
#
#     f.close()
#
#     for i in range(40,200,30):
#
#         # Read an image from library
#         image = cv2.imread(f"block_pics/blocks_{i}.png")
#
#         # Convert image from BGR to HSV space
#         imageHSV = cv2.cvtColor(image,cv2.COLOR_BGR2HSV)
#
#     #     tmp = cv2.cvtColor(block, cv2.COLOR_BGR2GRAY)
#     #
#     #     _, alpha = cv2.threshold(tmp, 0, 200, cv2.THRESH_BINARY)
#     #
#     #     b, g, r = cv2.split(block)
#     #
#     #     rgba = [b, g, r, alpha]
#     #
#     #     dst = cv2.merge(rgba,4)
#
#         # mask the green light from HSV and convert to grayscale
#         mask = mask_color(image, imageHSV)
```

```
#
#           blurred = cv2.GaussianBlur(mask,(11,11), 0)
#
#           # Detect corners from image
#           img_crnr, pts_loc, image = corner_detect(mask, image)
#
#           ## Draw contours over an image, if available
#           center, radius = center_det(pts_loc)
#
#           block_coordinate = "(" + str(center[0]) + "," + str(center[1]) + ")"
#
#           # Draw circle ontop of original image
#           cv2.circle(image, center, radius, (0,255,255),2)
#           cv2.circle(image, center, 0,(0,0,255),5)
#           cv2.putText(image,block_coordinate,(0,int(cy/2)),font,2,(0,0,0),2)
#
#           # Draw a cross at center of frame
#           cv2.line(image,(frame_center[0]-100, frame_center[1]),(frame_center[0]+10
#           cv2.line(image,(frame_center[0], frame_center[1]-100),(frame_center[0], f
#
#           cv2.imshow('pixel map of block',image)
#           cv2.imshow('black mask',mask)
#           cv2.waitKey(0)
#           cv2.destroyAllWindows()
#           pxl_radius.append(radius)
#
#           outstring = str(i) + ' ' + str(radius) + '\n'
#           f.write(outstring)
```

def picam_frame():

▼ View Source

```
def picam_frame():
    global image

    # Take picture with camera
    camera.capture (image, format="bgr")
    image = image.reshape((480,640,3))

    image = cv2.flip(image,1)


    return image
```

def img_show(name, img):

▼ View Source

```
def img_show(name, img):
    cv2.imshow(name,img)
#     key = cv2.waitKey(1) & 0xFF
    cv2.waitKey(0)
    # Write frame to video
    # out.write
```

```
    cv2.destroyAllWindows()
```

def dist_img(img, dist):

▼ View Source

```
def dist_img(img,dist):

    dist = str(dist) + "cm"
    data = f"{obj} Sodar: " + dist

    cv2.putText(img, data, (355,30),cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,0),2)

    return img
```

def servo_img(grip_state):

▼ View Source

```
def servo_img(grip_state):

    img = picam_frame()
    data = str(grip_state)
    data = "Duty Cycle: " + data + "%"

    cv2.putText(img, data, (20,30),cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,0),2)

    if grip_state == half:
        cv2.putText(img, "Half-Opened", (20,55),cv2.FONT_HERSHEY_SIMPLEX,1,(0,125,1
    elif grip_state == open_full:
        cv2.putText(img, "Fully-Opened", (20,55),cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,25
    elif grip_state == closed:
        cv2.putText(img, "Fully-Closed", (20,55),cv2.FONT_HERSHEY_SIMPLEX,1,(255,0,

    return img
```

def mask_color(image, imageHSV):

▼ View Source

```
def mask_color(image, imageHSV):
    global obj
    # HSV bounds
    # minHSV = np.array([41,74,163])
    # maxHSV = np.array([63,159,255])
#     # Trail Green Bock - LAB
#     minHSV = np.array([47,56,172])
#     maxHSV = np.array([255,255,255])

    if obj == 'green':
        # Trail Green Bock - LAB
        minHSV = np.array([46,88,121])
        maxHSV = np.array([65,141,255])
    elif obj == 'red':
```

```
        # Trail Red block - LAB
        minHSV = np.array([151,107,147])
        maxHSV = np.array([255,255,255])
    else:
        # Trail Blue block - LAB
        minHSV = np.array([72,94,97])
        maxHSV = np.array([126,188,212])

#     # Mask the gripper
#     mask_gripper = np.zeros(image.shape[:2], np.uint8)
#     mask_gripper[0:0, 380:0] = 255
#
#     # Cut out gripper
#     mask_img = cv2.bitwise_and(image, image, mask = mask_gripper)
    maskHSV = cv2.inRange(imageHSV, minHSV, maxHSV)

    return maskHSV
```

def prep_image(image):

▼ View Source

```
def prep_image(image):

     # Convert image from BGR to HSV space
    imageHSV = cv2.cvtColor(image,cv2.COLOR_BGR2HSV)

    # mask the green light from HSV and convert to grayscale
    mask = mask_color(image, imageHSV)

    # Mask HSV masked image of arrow
    blurred = cv2.GaussianBlur(mask,(11,11), 0)

    return blurred
```

def corner_detect(img, origImg):

▼ View Source

```
def corner_detect(img,origImg):

    # Create a list to store the x,y location of points
    pts_loc = []

    # Detect corners from image
    corners = cv2.goodFeaturesToTrack(img,5,0.01,10)

    if corners is not None:

        corners = np.int0(corners)

        # identify location of corners in image
        for i in corners:
            # Extract x,y coordinate of points
            x,y = i.ravel()

```

```
            pts_loc.append([x,y])

            # Draw circle ontop of original image
            cv2.circle(origImg, (x,y), 3, (0,0,255),-1)

        # Create a column vector from pts list
        pts_loc = np.array(pts_loc)

        return img, pts_loc, origImg, True
    else:
        return img, pts_loc, origImg, None
```

def center_det(pt_list, center):

▼ View Source

```
def center_det(pt_list, center):
    global bbox_radius
    # Extract x,y points from pt_list
    x = pt_list[:,0]
    y = pt_list[:,1]

    # Determine the min and max width & height values
    # of the points, as if to drow rectangle around arrow
    x_min = x.min()
    y_min = y.min()

    x_max = x.max()
    y_max = y.max()

    # Store height of bounding box
    vert_dst = y_max - y_min

    # Store width of bounding box
    horz_dst = x_max - x_min

    # Compute and store half dimensions of
    # box, will come later when determining
    # arrow direction
    y_half = vert_dst/2 + y_min
    x_half = horz_dst/2 + x_min

    # Store center of the block
    #center = [int(round(x_half)), int(round(y_half))]
    center.append(int(round(x_half)))
    center.append(int(round(y_half)))
    # Estimate radius
    bbox_radius = int(round(math.sqrt((x_max - x_half)**2 + (y_max - y_half)**2)))

    return center, bbox_radius
```

def bbox_detect(image, image_blurred):

▼ View Source

```python
def bbox_detect(image, image_blurred):
    global bbox_radius
    global obj

    center = []
    radius = 0
    success = None

#     # Extract frame from camera
#     image = picam_frame()
#
#     # Prep image for object detection
#     img_blurred = prep_image(image)

    # Detect corners from image
    img_crnr, pts_loc, image, var = corner_detect(image_blurred, image)

    if var is not None:

        # Check if corners are detected
        if len(pts_loc) > 3:
            ## Draw contours over an image, if available
            center, bbox_radius = center_det(pts_loc, center)

            block_coordinate = "(" + str(center[0]) + "," + str(center[1]) + ")"

            # Draw circle ontop of original image
            cv2.circle(image, tuple(center), bbox_radius, (0,255,255),2)
            cv2.circle(image, tuple(center), 0,(0,0,255),5)
            cv2.putText(image,block_coordinate,(0,int(cy)),font,2,(0,0,0),2)

            # Draw a cross at center of frame
            cv2.line(image,(frame_center[0]-100, frame_center[1]),(frame_center[0]+
            cv2.line(image,(frame_center[0], frame_center[1]-100),(frame_center[0],

            success = True

            print(f"{obj} block is detected")
            time.sleep(1)

            return image, center, bbox_radius, success

        else:
            print("Not clearly detected")
            success = False

            return image, center, bbox_radius, success

    else:
        print("No object of interest in scene")
        return image, 0, 0, success
```

def scanObject(color):

▼ View Source

```
def scanObject(color):
    global ave_center
    global obj
    global bbox_radius

    detected = False
    obj = color
    # Rotate every small angle, take a picture to detect existence of toy block
    stat = 0
    turn_angle = 15
    deg_diff = 0
    yaw = imu_serial()
    yaw_orig = yaw

    while not detected:

        image, ave_center, bbox_radius, success = bbox_detect()

        # write frame into file
#         out.write(image)
        if success:
            diff = ave_center[0]-frame_center[0]
            # Convert the pixel difference to degrees
            deg_diff = diff * pixel2deg

            detected = True

            return detected, deg_diff, image

        else:
            # Turn the robot by 15 degrees
            #stat += turn_angle

            return detected, turn_angle, image

            yaw = pivot(deg_diff= turn_angle)
            time.sleep(1)


        if stat >=360:
            print("Could not detect block from area")
            print("Drive to a different location and try again")
            break

    else:
        print("Desired object has been located")

    return detected, deg_diff, image
```

def dist_estimate():

▼ View Source

```python
def dist_estimate():
    # Read in pixel radius from image
    global bbox_radius

    img, __, bbox_radius, __ = bbox_detect()

    # Plug in pixel size into depth equation
    distance = (0.0099 * bbox_radius**2) - (1.8846*bbox_radius) + 103.47

    # Append image of estimated distance on image frame
    image = dist_img(img,round(distance,2))

    return distance, image
```

def email_media(obj):

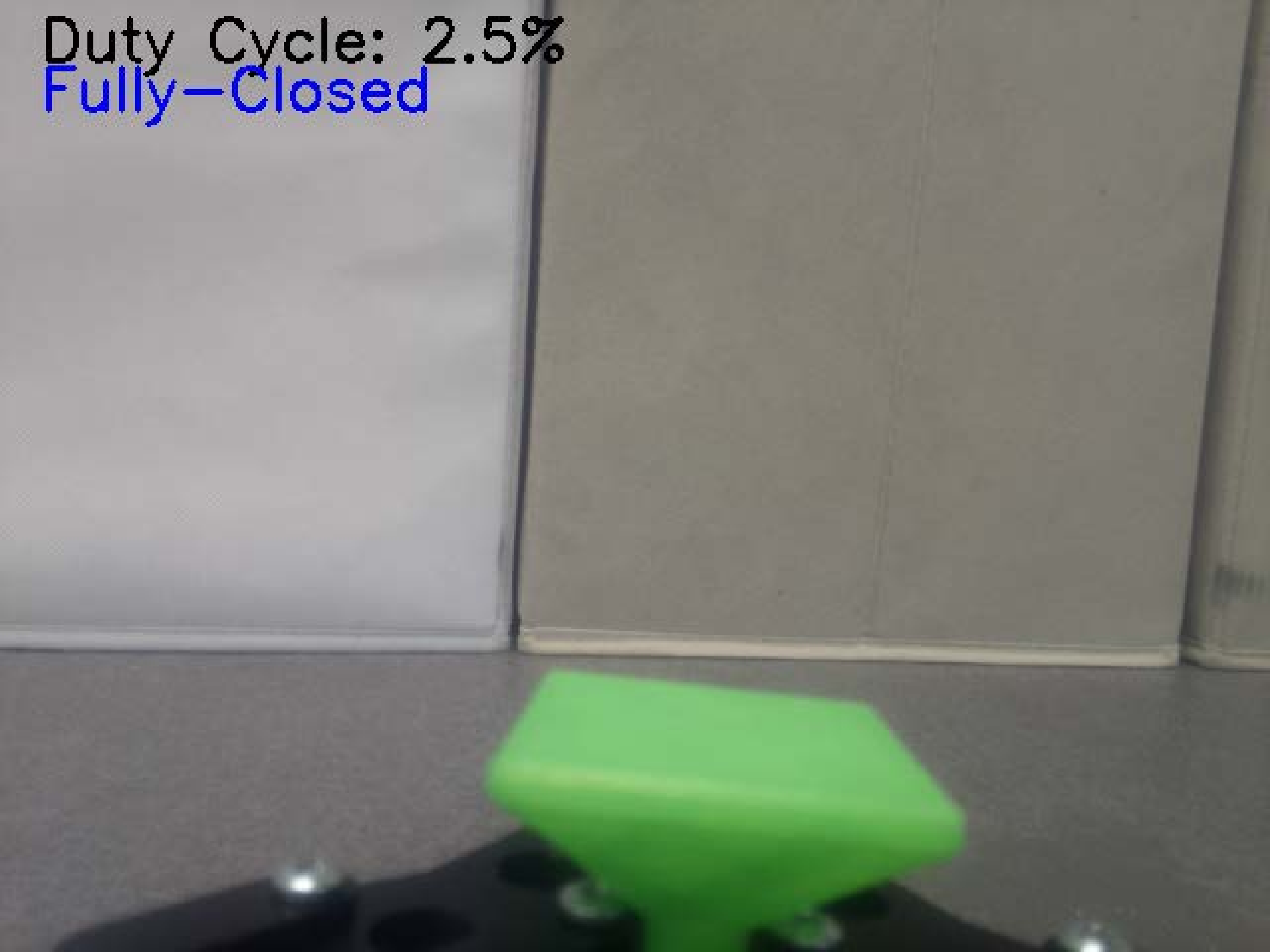
▼ View Source

```python
def email_media(obj):

    image = picam_frame()

    # send email to user with images
    smtpUser = 'ykebede2@terpmail.umd.edu'
    smtpPass = 'QwE@$d1219'

    toAdd = ['ENPM809TS19@gmail.com','yosephcollege@gmail.com']
    fromAdd = smtpUser

    f_time = datetime.now().strftime('%a %d %b @ %H:%M')
    subject = f'{obj} block picked up for transportation to construction area: ' +

    msg = MIMEMultipart()
    msg['Subject'] = subject
    msg['From'] = fromAdd
    msg['To'] = ",".join(toAdd)

    msg.preamble = "Image @ " + f_time

    body = email.mime.text.MIMEText("Baron Robot image: " + f_time)
    msg.attach(body)

    fp = open(sodar_img, 'rb')
    img = MIMEImage(fp.read())
    fp.close()
    msg.attach(img)

    fp = open(grip_img, 'rb')
    img = MIMEImage(fp.read())
    fp.close()
    msg.attach(img)

    s = smtplib.SMTP('smtp.gmail.com', 587)
    s.ehlo()
    s.starttls()
```

```
    s.ehlo()
    s.login(smtpUser, smtpPass)
    s.sendmail(fromAdd, toAdd, msg.as_string())
    s.quit()

```

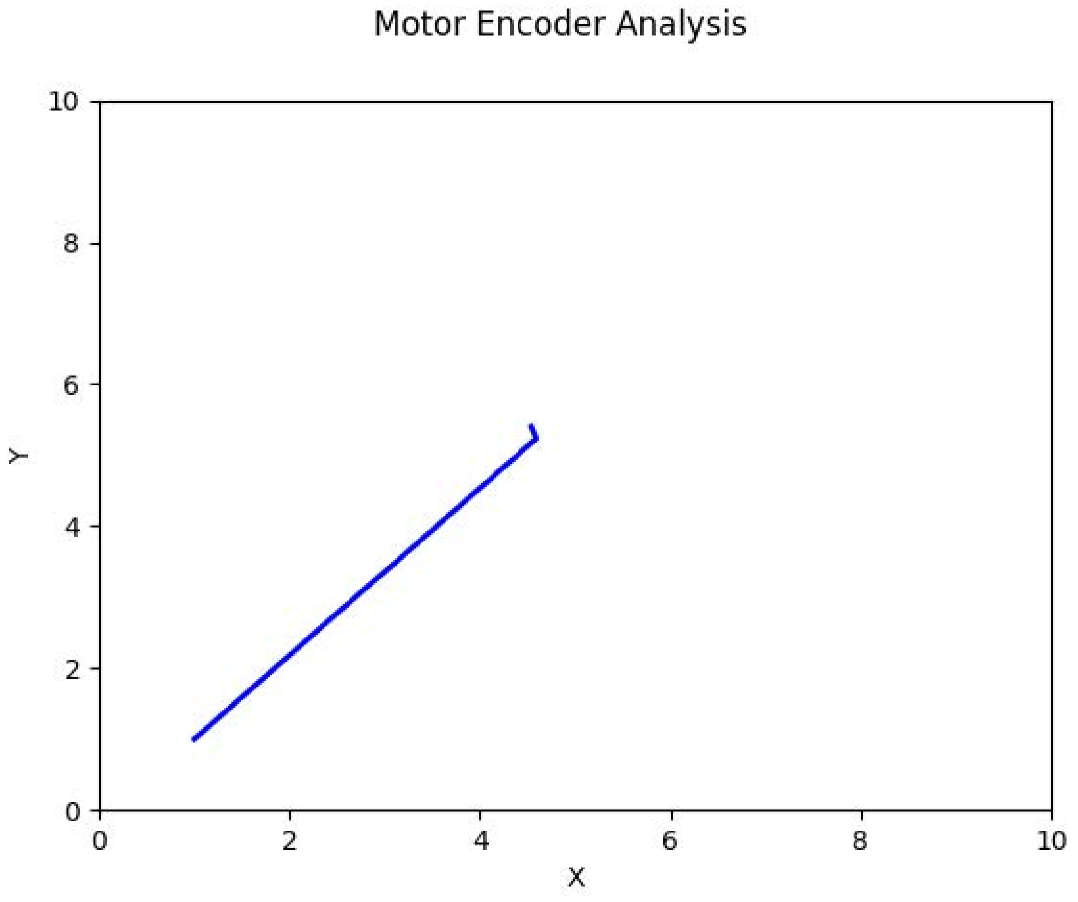

Duty Cycle: 2.5%
Fully-Closed


Duty Cycle: 2.5%
Fully-Closed


Duty Cycle: 2.5%
Fully-Closed


Duty Cycle: 2.5%
Fully-Closed


Duty Cycle: 2.5%
Fully-Closed



Motor Encoder Analysis
10
8
6
4
2
0
Y
0
2
4
6
8
10
X