



grand_challenge_submission.main

[▼ View Source](#)

```
1  import sys
2  custom_module_path = '/home/pi/ENPM701-assignments/grand_challenge_submission/'
3  sys.path.append(custom_module_path)
4  from racer.robot import Robot
5
6
7  if __name__ == "__main__":
8
9      """
10     Grand Challenge Competition
11
12     Defines the baron robot and then initiates start
13     of competition
14     """
15
16     # Initialize robot
17     baron = Robot("Speedy Baron")
18
19     # Start imu sensor reading in parallel thread
20     # baron.imu_reader.daemon = True
21     # baron.imu_reader.start()
22
23     # Order list of blocks to retrieve
24     order = ['red', 'green', 'blue', 'red', 'green', 'blue', 'red', 'green', 'blue']
25
26     # Start engine and wait for que
27     baron.start(order)
28
29     # Plot the robot trajectory during the challenge
30     baron.plotPath()
```



grand_challenge_submission.racer.robot

[▼ View Source](#)

```
1  # General Python packages
2  import math
3  import cv2
4  import os
5  import numpy as np
6  import matplotlib.pyplot as plt
7  import time
8  import RPi.GPIO as gpio
9  import serial
10 import logging
11 import threading
12 import queue
13
14 from typing import List
15 from random import randint
16
17 import sys
18 custom_module_path = '/home/pi/ENPM701-assignments/grand_challenge_submission/'
19 sys.path.append(custom_module_path)
20
21 # This is the main interface script that will control the robot
22 # as it traverses in the challenge arena.
23
24 # Below are the necessary imports for its different features
25
26 # For visual perception tasks
27 from Perception import cntr_bbox
28
29 # Imports for sending pictures via email
30 import smtplib
31 from smtplib import SMTP
32 from smtplib import SMTPException
33 import email
34 from email.mime.image import MIMEImage
35 from email.mime.text import MIMEText
36 from email.mime.multipart import MIMEMultipart
37 from datetime import datetime
38 import imaplib
39
40 class Robot(object):
41     """
42     A class defining basic attributes of Baron robot that
43     will be autonomously achieving objective
44
45     Attributes:
46         name (str): Name of robot
47         _x (float): horizontal position of robot from start position
48         _y (float): vertical position of robot from start position
49         _yaw (float): yaw orientation from +x as horizontal [0,180] or (-180,0)
50         _travel(float): estimating the distance to drive and reach block
```

```

51     _turn (float): keeps the next turn needed by block
52     _depth (float): measures distance of obstacle in front of robot from image
53     _path (list): stores the path traveled by robot as List of (x,y) points and
54     _pwms (list): stores a list of pwms for the robot motors
55     _dutyicycle (dict): holds the baseline pwm dutyicycle for forward,reverse,
56                        pivotleft and pivotright
57     _success (bool): keeps track whether a given task is successful or not
58     _gripper_state (str): stores gripper duty cycle state as opened or closed
59     _order(list): List of blocks to pickup
60     _end_game (bool): tells competition is over by victory or failure
61     _order_no (int): keeps track of number of orders completed
62     _block_now (str): the block being pursued now
63     _task_now (str): keeps track of the task being completed at the moment
64     _actions (dict): holding list of all service actions robot performs with block
65                     on which process is being done at a given time
66     _status (dict): a dictionary keeping track of completed actions along with
67     _img_frame (img): saves current image frame
68     _out (cv2 video obj): gather all frames into a video
69     _logger (log): Log statements for debugging and status over the terminal
70     _kp (float): proportional gain
71     _ki (float): integral gain
72     _kd (float): derivative gain
73     _direction (str): tells weather robot is moving forward or reverse
74     _event (event): event object to end the thread call
75     _pause_event (event): event object to pause and resume thread
76     imu_reader (thread): threading object to read from imu during forward run
77     _ser (serial): object reading serial input streams from imu sensor
78     _img_saved (img): keeping track of saved image to send in an email
79     _survey_loc (List): contains x,y locations for the robot to navigate and :
80                        so that to extract toy blocks for mission
81
82     """
83
84     # define the codec and create VideoWriter object
85     fourcc = cv2.VideoWriter_fourcc(*'XVID')
86     # Gripper States
87     grip_state = dict([('closed',2.5),('opened',7.5)])
88     # Matrix for Image frame
89     image = np.empty((480*640*3,),dtype=np.uint8)
90     # define the codec and create VideoWriter object
91     fourcc = cv2.VideoWriter_fourcc(*'XVID')
92     # Open .txt file to save data
93     f = open('robot_path_final.txt','a')
94
95     dutyset = [('f', [70,70]), ('rev', [30,30]),('l', 100),('r',100)]
96
97     duty = dict(dutyset)
98     font = cv2.FONT_HERSHEY_COMPLEX_SMALL
99
100    # turn angel
101    turn = 15
102
103    # detect radius
104    bbox_radius = 0
105
106    # Initialize FL and BR button count

```

```
107 counterBR = np.uint64(0)
108 counterFL = np.uint64(0)
109 buttonBR = int(0)
110 buttonFL = int(0)
111
112 # Proportional Control Gain
113 Kp = 1
114 Ki = 0.05
115 Kd = 0.08
116
117 # Important Locations in map in units of feet
118 origin = [0,0]
119 start_pt = [1,1]
120 clutter_env = [5,5]
121 const_zone = [2.25,6.5]
122
123 # create object to read camera
124 video = cv2.VideoCapture(0)
125
126 frame_center = (320,240)
127 (cx,cy) = (frame_center[0]/4.5, frame_center[1]/8)
128
129 queue = queue.Queue()
130
131 img_folder = 'challenge_pics'
132
133 counter = 0
134
135 # Environment assessment points
136 scan_spots = [(7.5,3),(7.5,8)]
137
138 # Global x, y, positions and yaw from fixed reference
139 # frame (0,0)
140 global_x = 1
141 global_y = 1
142 global_yaw = 0
143
144 def __init__(self, name: str):
145     """
146     Initialize Robot attributes
147
148     Args:
149         name: str - takes name of robot
150
151     Returns:
152         None
153     """
154
155     self.name = name
156     self._x = self.start_pt[0]
157     self._y = self.start_pt[1]
158     self._yaw = 0.0
159     self._travel = 0
160     self._turn = 0
161     self._depth = 0
162     self._path = [(1,1,0)]
```

```

163     self._pwms = []
164     self._duty_cycle = self.duty
165     self._success = False
166     self._gripper_state = self.grip_state['closed']
167     self._order = []
168     self._end_game = False
169     self._order_no = 0
170     self._block_now = ''
171     self._task_now = ''
172     self._actions = dict([('driving', False), ('rotating', False), ('picking', False), ('found', False)])
173     self._status = dict([('started', False), ('picked_up', False), ('found', False)])
174     self._img_frame = self.image
175     self._out = cv2.VideoWriter('grand_chall_vid.avi', self.fourcc, 3, (640, 480))
176     self._logger = logging.getLogger(name)
177     self._kp = self.Kp
178     self._ki = self.Ki
179     self._kd = self.Kd
180     self._direction = 'f'
181     self._event = threading.Event()
182     self._pause_event = threading.Event()
183     self._imu_reader = threading.Thread(target = self._imu_serial_straight, args = ())
184     self._ser = serial.Serial('/dev/ttyUSB0', 19200)
185     self._img_saved = self.image
186     self._survey_loc = self.scan_spots
187
188     # Set up logger with Debug initially
189     self._logger.setLevel(logging.DEBUG)
190     # Create console handler and set level to debug
191     console_handler = logging.StreamHandler()
192     console_handler.setLevel(logging.DEBUG)
193     # Log terminal messages to a file
194     file_handler = logging.FileHandler('gc_log_3.log')
195     file_handler.setLevel(logging.INFO)
196     # Create formatter
197     formatter = logging.Formatter('%(name)s - %(levelname)s - %(message)s')
198     # Add formatter to handler
199     console_handler.setFormatter(formatter)
200     file_handler.setFormatter(formatter)
201     # Add console handler to logger
202     self._logger.addHandler(console_handler)
203     self._logger.addHandler(file_handler)
204     # Initialize gpios and pwms
205     self._init()
206
207
208     @property
209     def name(self):
210         return self._name
211
212     @name.setter
213     def name(self, name):
214         self._name = name
215
216     def _init(self):
217         """
218         Initialize gpio pins and set pwms to command DC motors, encoders and

```

```
219         servo
220
221     Args:
222         None
223
224     Returns:
225         None
226     """
227
228     gpio.cleanup()
229     gpio.setmode(gpio.BOARD)
230
231     # Setup GPIO pin(s)
232     gpio.setup(36, gpio.OUT) # Servo
233
234     gpio.setup(31, gpio.OUT) # IN1
235     gpio.setup(33, gpio.OUT) # IN2
236     gpio.setup(35, gpio.OUT) # IN3
237     gpio.setup(37, gpio.OUT) # IN4
238
239     gpio.setup(7, gpio.IN, pull_up_down = gpio.PUD_UP)
240     gpio.setup(12, gpio.IN, pull_up_down = gpio.PUD_UP)
241
242     self._pwms.clear()
243
244     # initialize pwm signal to control motor
245     pwm01 = gpio.PWM(31, 50) # BackLeft motor
246     pwm11 = gpio.PWM(33, 50) # FrontLeft motor
247     pwm22 = gpio.PWM(35, 50) # FrontRight motor
248     pwm02 = gpio.PWM(37, 50) # BackRight motor
249     pwmS = gpio.PWM(36, 50) # Servo
250
251     self._pwms = [pwm01, pwm11, pwm22, pwm02, pwmS]
252
253     for pwm in self._pwms:
254         pwm.start(0)
255
256     def _pwmZero(self):
257         """
258         Stops motors by zeroing pwm values in all pins
259         controlling DC motors
260
261         Args:
262             None
263
264         Returns:
265             None
266         """
267
268
269         self._pwms[0].ChangeDutyCycle(0)
270         self._pwms[1].ChangeDutyCycle(0)
271         self._pwms[2].ChangeDutyCycle(0)
272         self._pwms[3].ChangeDutyCycle(0)
273
274     def _gameover(self):
```

```
275     """
276     Terminates run by stopping pwms and cleaning up
277     gpio pins
278
279     Args:
280         None
281
282     Returns:
283         None
284     """
285
286     self._pwmZero()
287     self._pwms[-1].ChangeDutyCycle(self.grip_state['closed'])
288     time.sleep(1)
289     for pwm in self._pwms:
290         pwm.stop()
291     gpio.cleanup()
292
293     def _forward(self,vals):
294         """
295         Commands DC motors to drive robot forward by sending
296         Dutycycles to respective pwm pins
297
298         Args:
299             None
300
301         Returns:
302             None
303         """
304         # Left wheels
305         self._pwms[0].ChangeDutyCycle(vals[0])
306         self._pwms[1].ChangeDutyCycle(0)
307         # Right wheels
308         self._pwms[2].ChangeDutyCycle(0)
309         self._pwms[3].ChangeDutyCycle(vals[1])
310
311     def _reverse(self,vals):
312         """
313         Commands DC motors to drive robot in reverses by sending
314         Dutycycles to respective pwm pins
315
316         Args:
317             None
318
319         Returns:
320             None
321         """
322         # Left wheels
323         self._pwms[0].ChangeDutyCycle(0)
324         self._pwms[1].ChangeDutyCycle(vals[0])
325         # Right wheels
326         self._pwms[2].ChangeDutyCycle(vals[1])
327         self._pwms[3].ChangeDutyCycle(0)
328
329     def _pivotleft(self,vals):
330         """
```

```
331 Commands DC motors to pivot robot left by sending
332 Dutycycles to respective pwm pins
333
334 Args:
335     None
336
337 Returns:
338     None
339     """
340 # Left wheels
341 self._pwms[0].ChangeDutyCycle(0)
342 self._pwms[1].ChangeDutyCycle(vals)
343 # Right wheels
344 self._pwms[2].ChangeDutyCycle(0)
345 self._pwms[3].ChangeDutyCycle(vals)
346
347 def _pivotright(self,vals):
348     """
349     Commands DC motors to pivot robot right by sending
350     Dutycycles to respective pwm pins
351
352     Args:
353         None
354
355     Returns:
356         None
357         """
358
359     # Left wheels
360 self._pwms[0].ChangeDutyCycle(vals)
361 self._pwms[1].ChangeDutyCycle(0)
362 # Right wheels
363 self._pwms[2].ChangeDutyCycle(vals)
364 self._pwms[3].ChangeDutyCycle(0)
365
366 def _servo_cntrl(self,duty_cycle):
367     """
368     Commands servo motors to open and close gripper by sending
369     Dutycycles to respective pwm pins
370
371     Args:
372         None
373
374     Returns:
375         None
376         """
377
378 self._pwms[-1].ChangeDutyCycle(duty_cycle)
379 time.sleep(1)
380
381 img = cntr_bbox.servo_img(duty_cycle)
382 self._img_frame = cntr_bbox.dist_img(img, self._distance())
383
384 #cntr_bbox.img_show("Servo status",self._img_frame)
385 self._out.write(img)
386
```



```
387 def _picam_frame(self):
388     """
389     Calls picamera object to take image frame from the pi camera
390     and applies corrections by flipping to reflect reality
391
392     Args:
393         None
394
395     Returns:
396         None
397     """
398     # Take picture with camera
399     cntr_bbox.camera.capture(self.image, format="bgr")
400     image = self.image.reshape((480,640,3))
401
402     self._img_frame = cv2.flip(image,1)
403
404 def _mask_color(self, imageHSV):
405
406     if self._block_now == 'green':
407         # Trail Green Block - LAB
408         minHSV = np.array([46,88,121])
409         maxHSV = np.array([65,141,255])
410     elif self._block_now == 'red':
411         # Trail Red block - LAB
412         minHSV = np.array([151,107,147])
413         maxHSV = np.array([255,255,255])
414     elif self._block_now == 'blue':
415         # Trail Blue block - LAB
416         minHSV = np.array([72,94,97])
417         maxHSV = np.array([126,188,212])
418
419     maskHSV = cv2.inRange(imageHSV, minHSV, maxHSV)
420
421     return maskHSV
422
423 def _prep_image(self):
424     """
425     Processes images by applying HSV masking and gaussian blur
426     for use by latter object detection functions
427
428     Args:
429         None
430
431     Returns:
432         None
433     """
434
435     # Convert image from BGR to HSV space
436     imageHSV = cv2.cvtColor(self._img_frame, cv2.COLOR_BGR2HSV)
437
438     # mask the green light from HSV and convert to grayscale
439     mask = self._mask_color(imageHSV)
440
441     # Mask HSV masked image of arrow
442     blurred = cv2.GaussianBlur(mask, (11,11), 0)
```

```
443
444     return blurred
445
446 def _grip_checker(self):
447     """
448     Processes images by applying a rectangular mask over desired
449     region of interest to verify whether gripper has held toy block
450
451     Args:
452         None
453
454     Returns:
455         self._image_frame (img): Masked image
456     """
457     self._logger.info("Inside Grip Checker Function")
458     # Create a mask of gripped block in image
459     # Save the shape of the image array
460     (height, width, c) = self._img_frame.shape
461
462     # Create mask of image just like picture
463     mask = np.zeros_like(self._img_frame[:, :, 0])
464
465     # Create a white rectangle
466     x, y, w, h = round(width/2)-130, round(height/2)-30, 250, 300
467     cv2.rectangle(mask, (x, y), (x+w, y+h), 255, -1)
468
469     # Extract the image that aligns in the rectangle
470     self._img_frame = cv2.bitwise_and(self._img_frame, self._img_frame, mask=mask)
471
472     return self._img_frame
473
474 def _wait2start(self):
475     """
476     Waits for the competition start QR code to be visible,
477     and reads QR code in, which if corresponds to "ENPM701"
478     returns a True boolean to begin competition
479
480     Args:
481         None
482
483     Returns:
484         start (bool): True/False based on detection of expected QR
485     """
486     # Define detector
487     detector = cv2.QRCodeDetector()
488
489     # Check if program start initiated
490     start = False
491
492     cnt = 0
493
494     self._logger.info("Waiting for start cue . . . ")
495
496     try:
497
498         for frame in cntr_bbox.camera.capture_continuous(cntr_bbox.rawCapture,
```

```
499
500     # grab the current frame
501     img = frame.array
502
503     img = cv2.flip(img,1)
504
505     data, bbox, _ = detector.detectAndDecode(img)
506
507     if(bbox is not None):
508         for i in range(len(bbox)):
509             cv2.line(img, tuple(bbox[i][0]),tuple(bbox[(i+1)%len(bbox)
510             cv2.putText(img, data, (int(bbox[0][0][0]),int(bbox[0][0][
511
512     if data:
513         self._logger.info(f"Data: {data}")
514         pass
515
516     # Show results to the screen
517     cv2.imshow("QR Code detector",img)
518     key = cv2.waitKey(1) & 0xFF
519
520     # write frame into file
521     self._out.write(img)
522
523     # clear the stream in preparation for the next frame
524     cntr_bbox.rawCapture.truncate(0)
525
526     # Break out of loop by pressing the q key
527     if(key == ord("q")):
528         self._logger.warn("Program Terminated")
529         self.video.release()
530         cv2.destroyAllWindows()
531         break
532
533     if data == "ENPM701":
534         self._logger.info("Cue Received \n")
535         self._logger.info('@' * 45)
536         self._logger.info("Starting Grand Challenge! \n")
537         self._logger.info('#' * 45)
538         start = True
539         self.video.release()
540         cv2.destroyAllWindows()
541         time.sleep(3)
542         break
543
544     except Exception as error:
545         self._logger.error(error)
546         self.video.release()
547         cv2.destroyAllWindows()
548
549     return start
550
551     def _email_media(self):
552
553         # Take a picture of gripped block to email
554         self._picam_frame()
```

```

555     # Append status of servo on image
556     self._img_frame = cntr_bbox.servo_img(self._gripper_state)
557     self._out.write(self._img_frame)
558     # Save to designated directory
559     cv2.imwrite(f'{self.img_folder}/{self._block_now}_{self.counter}.png', self._img_frame)
560     #time.sleep(1)
561     # Read saved image
562     img_name = f'{self.img_folder}/{self._block_now}_{self.counter}.png'
563
564     # send email to user with images
565     smtpUser = 'ykebede2@terpmail.umd.edu'
566     smtpPass = 'QwE@$d1219'
567
568     toAdd = ['ENPM809TS19@gmail.com', 'yosephcollege@gmail.com']
569     fromAdd = smtpUser
570
571     f_time = datetime.now().strftime('%a %d %b @ %H:%M')
572     subject = f'ENPM701-GrandChallenge-{f_time}-yosephK-{self._block_now}-{self.counter}'
573
574     self.counter += 1
575
576     msg = MIMEMultipart()
577     msg['Subject'] = subject
578     msg['From'] = fromAdd
579     msg['To'] = ",".join(toAdd)
580
581     msg.preamble = "Image @ " + f_time
582
583     body = email.mime.text.MIMEText("Baron Robot image: " + f_time)
584     msg.attach(body)
585
586     fp = open(img_name, 'rb')
587     img = MIMEImage(fp.read())
588     fp.close()
589     msg.attach(img)
590
591     s = smtplib.SMTP('smtp.gmail.com', 587)
592     s.ehlo()
593     s.starttls()
594     s.ehlo()
595     s.login(smtpUser, smtpPass)
596     s.sendmail(fromAdd, toAdd, msg.as_string())
597     s.quit()
598
599     self._logger.info("Email delivered!")
600
601     def _meter2encoder(self, x_dist):
602         """
603         Reads in distance values in meters and converts to
604         encoder counts
605
606         Args:
607             None
608
609         Returns:
610             encod (int): converted encoder counts from distance in m

```

```
611         """
612         encod = round(float(x_dist / (2*math.pi*0.0325))*960)
613
614         return encod
615
616     def _encoder2meter(self, encd):
617         """
618         Reads in encoder values and converts them to meters
619
620         Args:
621             None
622
623         Returns:
624             cms (float): converted encoder counts to distance in m
625         """
626         dist = round(float((encd / 960) * 2*math.pi*0.0325),4)
627
628         return dist
629
630     def _cm2encoder(self, x_dist):
631         """
632         Reads in distance values in centimeters and converts to
633         encoder counts
634
635         Args:
636             None
637
638         Returns:
639             encod (int): converted encoder counts from distance in cms
640         """
641         dist = x_dist/100
642         encod = self._meter2encoder(dist)
643         return encod
644
645     def _encoder2cm(self, encd):
646         """
647         Reads in encoder values and converts them to centimeters
648
649         Args:
650             None
651
652         Returns:
653             cms (float): converted encoder counts to distance in cms
654         """
655         dist = self._encoder2meter(encd)
656         cms = round(dist*100,4)
657
658         return cms
659
660     def _feet2encoder(self, x_dist):
661         """
662         Reads in distance values in feet and converts to
663         encoder counts
664
665         Args:
666             None
```

```
667
668     Returns:
669         encod (int): converted encoder counts from distance in feet
670         """
671     dist = round(float(x_dist * 0.3048),4)
672     encod = self._meter2encoder(dist)
673     return encod
674
675 def _encoder2feet(self,x_encod):
676     """
677     Reads in encoder values and converts to
678     feet
679
680     Args:
681         None
682
683     Returns:
684         feet (float): converted encoder counts from distance in feet
685         """
686     meter = self._encoder2meter(x_encod)
687     feet = round(meter * 3.28084,4)
688
689     return feet
690
691 def _feet2cm(self,x_dist):
692     """
693     Reads in distance values in feet and converts to
694     encoder counts
695
696     Args:
697         None
698
699     Returns:
700         encod (int): converted encoder counts from distance in feet
701         """
702     dist = round(float(x_dist * 12 * 2.54),3)
703
704     return dist
705
706 def _cm2feet(self,x_dist):
707     """
708     Reads in distance values in feet and converts to
709     encoder counts
710
711     Args:
712         None
713
714     Returns:
715         encod (int): converted encoder counts from distance in feet
716         """
717     dist = round(float(x_dist / (12 * 2.54)),2)
718
719     return dist
720
721 def _imu_serial(self):
722     """
```

```

723     Reads in serial robot's yaw angle value from imu sensor, cleans buffer,
724     and converts it into a float data type. Convers reading from 0-360 to
725     -180 < yaw < 180.
726
727     Args:
728         None
729
730     Returns:
731         self._yaw (float): robots yaw in degrees
732     """
733     self._ser.reset_input_buffer()
734     self._ser.flush()
735
736     while True:
737
738         try:
739             # Read for imu from serial
740             if(self._ser.in_waiting > 0):
741                 # if cnt >5:
742                 # Strip serial stream of extra characters
743                 line = self._ser.readline()
744
745                 line = line.rstrip().lstrip()
746
747                 line = str(line)
748                 line = line.strip('"')
749                 line = line.strip("b'")
750
751                 # Return float
752                 line = float(line)
753
754                 if (line > 180 and line <=360):
755                     line = line - 360
756
757                 self._yaw = line
758                 return self._yaw
759
760         except Exception as error:
761             self._logger.error(f"Imu Error: {error}")
762
763     def _imu_serial_straight(self,queue):
764         """
765         Reads in serial robot's yaw angle value from imu sensor, cleans buffer,
766         and converts it into a float data type. Convers reading from 0-360 to
767         -180 < yaw < 180.
768
769         Only difference from self._imu_serial() is that this is run in a separate
770         thread which is controlled by events to start, pause, and stop function call
771
772         Args:
773             None
774
775         Returns:
776             self._yaw (float): robots yaw in degrees
777         """
778

```

```
779 while not self._event.is_set():
780
781     self._ser.reset_input_buffer()
782     self._ser.flush()
783
784     try:
785         while self._pause_event.is_set():
786             # Read for imu from serial
787             if(self._ser.in_waiting > 0):
788                 line = self._ser.readline()
789
790                 line = line.rstrip().lstrip()
791
792                 line = str(line)
793                 line = line.strip("'")
794                 line = line.strip("b'")
795
796                 # Return float
797                 line = float(line)
798
799                 if (line > 180 and line <=360):
800                     line = line - 360
801
802                 queue.put(line)
803
804     except Exception as error:
805         self._logger.warning(f"Imu Error inside thread: {error}")
806
807 def _distance(self):
808     """
809     Reads in serial robot's distance from an obstacle using ultrasonic sensor l
810     sending a trigger pulse and listening the returning echo to compuse the sep
811     distance from nearby object using triangulation technique. Then it saves c
812     distance in self._depth
813
814     Args:
815         None
816
817     Returns:
818         None
819     """
820     pulse_start = 0
821     pulse_end = 0
822
823     # Define pin allocations
824     trig = 16
825     echo = 18
826     # Setup GPIO board & pins
827     gpio.setup(trig, gpio.OUT)
828     gpio.setup(echo, gpio.IN)
829
830     # Ensure output has no value
831     gpio.output(trig, False)
832     time.sleep(0.01)
833
834     # Generate trigger pulse
```



```
835         gpio.output(trig, True)
836         time.sleep(0.00001)
837         gpio.output(trig, False)
838
839         # Generate echo time signal
840         while gpio.input(echo) == 0:
841             pulse_start = time.time()
842
843         while gpio.input(echo) == 1:
844             pulse_end = time.time()
845
846         pulse_duration = pulse_end - pulse_start
847
848         # Convert time to distance
849         distance = pulse_duration * 17150
850         self._depth = round(distance, 2)
851
852         return self._depth
853
854     def _rotate(self):
855         """
856         Reads in direction of rotation requested, and starts
857         rotating the motors accordingly to achieve desired orientation.
858
859         Args:
860             None
861
862         Returns:
863             None
864         """
865
866         if self._direction == 'r':
867             self._pivotright(60)
868         elif self._direction == 'l':
869             self._pivotleft(60)
870
871
872     def _dist_estimate(self):
873         """
874         Reads in radius of bounding box from bbox detector function, and
875         plugs radius into precomputed correlation function to estimate the
876         depth of toy block from robot.
877
878         Assigns results to self._travel and saves image showing value of distance
879         appended to it in self._img_frame and writes frame to video object self._oi
880
881         Args:
882             None
883
884         Returns:
885             None
886         """
887         # Take image with picamera
888         self._picam_frame()
889         self._img_frame, __, bbox_radius, self._success = cntr_bbox.bbox_detect(se
890
```

```

891     if self._success not in [None, False]:
892         # Plug in pixel size into depth equation
893         distance = (0.0099 * bbox_radius**2) - (1.8846*bbox_radius) + 103.47
894         self._travel = round(distance,2)
895         # Append image of estimated distance on image frame
896         self._img_frame = cntr_bbox.dist_img(self._img_frame,self._travel)
897         # save image into video object
898         self._out.write(self._img_frame)
899 def _corner_detect(self,img,origImg):
900
901     # Create a list to store the x,y location of points
902     pts_loc = []
903
904     # Detect corners from image
905     corners = cv2.goodFeaturesToTrack(img,5,0.01,10)
906
907     if corners is not None:
908
909         corners = np.int0(corners)
910
911         # identify location of corners in image
912         for i in corners:
913             # Extract x,y coordinate of points
914             x,y = i.ravel()
915
916             pts_loc.append([x,y])
917
918             # Draw circle ontop of original image
919             cv2.circle(origImg, (x,y), 3, (0,0,255),-1)
920
921         # Create a column vector from pts list
922         pts_loc = np.array(pts_loc)
923
924         return img, pts_loc, origImg, True
925     else:
926         return img, pts_loc, origImg, None
927
928 def _center_det(self,pt_list, center):
929
930     # Extract x,y points from pt_list
931     x = pt_list[:,0]
932     y = pt_list[:,1]
933
934     # Determine the min and max width & height values
935     # of the points, as if to draw rectangle around arrow
936     x_min = x.min()
937     y_min = y.min()
938
939     x_max = x.max()
940     y_max = y.max()
941
942     # Store height of bounding box
943     vert_dst = y_max - y_min
944
945     # Store width of bounding box
946     horz_dst = x_max - x_min

```

```

947
948     # Compute and store half dimensions of
949     # box, will come later when determining
950     # arrow direction
951     y_half = vert_dst/2 + y_min
952     x_half = horz_dst/2 + x_min
953
954     # Store center of the block
955     #center = [int(round(x_half)), int(round(y_half))]
956     center.append(int(round(x_half)))
957     center.append(int(round(y_half)))
958     # Estimate radius
959     self.bbox_radius = int(round(math.sqrt((x_max - x_half)**2 + (y_max - y_half)**2)))
960
961     return center, self.bbox_radius
962
963 def _bbox_detect(self):
964     """
965     Reads in an image, and then applies preprocessing functions that mask and
966     blur image. Then, attempts to detect corners of an object in frame, and if
967     successful it will run further computation to determine the center and radius
968     of detected object. Finally adds the draws the bounding circle and center point
969     as well as write computed values on original image
970
971     Assigns results to self._travel and saves image showing value of distance
972     appended to it in self._img_frame and writes frame to video object self._out
973
974     Args:
975         image (image): Original colored image taken in from camera
976
977     Returns:
978         bbox_radius (float): Radius of bounding circle detecting object
979         len(pts_loc) (int): The number of corner points identified on object frame
980     """
981     # Initialize list to save corner points and bbox radius
982     bbox_radius = 0
983     pts_loc = []
984     center = []
985
986     # Take image using camera frame
987     self._picam_frame()
988
989     # Prepare image by masking and blurring
990     image_blurred = self._prep_image()
991
992     # Extract bbox radius from image
993     self._img_frame, center, bbox_radius, self._success = cntr_bbox.bbox_detect(image_blurred)
994
995     return self._img_frame, center, bbox_radius, self._success
996
997     # Detect corners from image
998     # img_cnrn, pts_loc, image, var = self._corner_detect(image_blurred, self._img_frame)
999     #
1000     # if var is not None:
1001     #
1002     # Check if corners are detected

```

```

1003 #         if len(pts_loc) > 3:
1004 #             ## Draw contours over an image, if available
1005 #             center, bbox_radius = self._center_det(pts_loc, center)
1006 #
1007 #             block_coordinate = "(" + str(center[0]) + "," + str(center[1]) +
1008 #
1009 #             # Draw circle ontop of original image
1010 #             cv2.circle(image, tuple(center), bbox_radius, (0,255,255),2)
1011 #             cv2.circle(image, tuple(center), 0,(0,0,255),5)
1012 #             cv2.putText(image,block_coordinate,(0,int(self.cy)),self.font,2,
1013 #
1014 #             # Draw a cross at center of frame
1015 #             cv2.line(image,(self.frame_center[0]-100, self.frame_center[1]),
1016 #             cv2.line(image,(self.frame_center[0], self.frame_center[1]-100),
1017 #
1018 #             success = True
1019 #
1020 #             self._logger.info(f"{self._block_now} block is detected")
1021 #             time.sleep(1)
1022 #
1023 #             return image, center, bbox_radius, success
1024 #
1025 #         else:
1026 #             self._logger.info("Not clearly detected")
1027 #             success = False
1028 #
1029 #             return image, center, bbox_radius, success
1030 #
1031 #     else:
1032 #         print("No object of interest in scene")
1033 #         return image, 0, 0, success
1034
1035 self._logger.info(f"Radius {bbox_radius}")
1036 self._logger.info(f"Number of corners/edges detected are {len(pts_loc)}")
1037
1038 #return bbox_radius, center #, len(pts_loc),
1039
1040 def _pivot(self,deg_diff,just_turn=False,steady = False):
1041     """
1042     This function pivots robot to deseired angle
1043
1044     Args:
1045     deg_diff (float): Amount of angular rotation to turn.
1046     Default value is self._turn parameter
1047
1048     Returns:
1049     None
1050
1051     """
1052     self._logger.info("Inside pivot function")
1053     self._task_now = 'rotating'
1054     self._actions[self._task_now] = True
1055
1056     # To know the direction of turn. Notation based on camera object detection
1057     # notation
1058     # thresh = 0.8

```

```

1059     if deg_diff >= 0:
1060         self._direction = 'r'
1061     else:
1062         self._direction = 'l'
1063     # thresh = -0.8
1064
1065     # Initialize change in time for derivative control
1066     zones = [0.4, 0.8, 1]
1067
1068     self._yaw = self._imu_serial()
1069     yaw_start = self._yaw
1070
1071     # Angle diff initialized
1072     yaw_diff0 = deg_diff + 0.001
1073
1074     yaw_final = yaw_start + deg_diff
1075     yaw_diff = yaw_final - yaw_start
1076     yaw_diff_old = yaw_diff
1077
1078     self._logger.info(f"Needed final yaw is {yaw_final} degrees")
1079     yaw_current = yaw_start
1080     # Lower and upper bounds of dutycycle initialized
1081     lower_dc = 70
1082     upper_dc = 90
1083     # Divide the distance traveled into three zones
1084     trip = [x * yaw_diff for x in zones]
1085     # Variables for PID control
1086     derivative = 0
1087     integral = 0
1088
1089     start = time.time()
1090     yaw_diff_list = [0,0]
1091     yaw_diff_list.append(yaw_diff)
1092     window_size = 5
1093     while True:
1094         try:
1095
1096             #prcnt = abs(self._imu_serial() - yaw_start)/deg_diff
1097             prcnt = 1 - yaw_diff/yaw_diff0
1098
1099             # Adopting moving average
1100             # cumsum = np.cumsum(yaw_diff_list)
1101             # cumsum>window_size:] = cumsum>window_size:] - cumsum[:~window_si
1102             # moving_avg = cumsum>window_size - 1:] / window_size
1103
1104             if prcnt >= zones[0] and prcnt <= zones[1]:
1105                 self._dutycycle[self._direction] = self.duty[self._direction]
1106             elif prcnt > zones[1]:
1107                 self._dutycycle[self._direction] = self.duty[self._direction]
1108
1109             # Save robots pose by saving its x, yaw position
1110             self._path.append((round(self.global_x,2),round(self.global_y,2),r
1111             self._dutycycle[self._direction] = np.clip(self._dutycycle[self._d
1112
1113             if abs(yaw_diff) <= 2:
1114                 self._dutycycle[self._direction] = 65

```

```

1115
1116     elif abs(yaw_diff) >= 100:
1117         self._duty_cycle[self._direction] = 100
1118
1119
1120
1121     if abs(yaw_diff) >= 0.8:
1122
1123         if ((np.sign(yaw_diff) == np.sign(yaw_diff_old)) or (abs(yaw_d
1124 #         if (abs(yaw_diff - np.mean(yaw_diff_list[-3:-1])) < 5):
1125 #         if abs(yaw_diff) >= 0.5:
1126             self._logger.info(f"Current yaw {self._imu_serial()}; Angl
1127
1128             if not steady:
1129                 if self._direction == 'r':
1130                     self._pivotright(self._duty_cycle[self._direction])
1131                 elif self._direction == 'l':
1132                     self._pivotleft(self._duty_cycle[self._direction])
1133             else:
1134                 if self._direction == 'r':
1135                     self._pivotright(70)
1136                 elif self._direction == 'l':
1137                     self._pivotleft(70)
1138
1139     else:
1140         self._pwmZero()
1141         total_turn = yaw_start - self._imu_serial()#self.queue.get()
1142         self._logger.info(f"Current yaw {self._imu_serial()}; Total tu
1143         self._logger.info("Robot turn done")
1144
1145         yaw_diff = round(yaw_diff,2)
1146
1147     if not just_turn:
1148         if steady:
1149             #if (abs(yaw_diff) >= 0.):
1150             # Check for overshooting
1151
1152             deg_diff = self._scan4object()
1153
1154             if deg_diff is not None:
1155
1156                 if abs(deg_diff) > 0.5:
1157
1158 #                     time.sleep(1)
1159 #                     # Return pwm duty cycle back to base
1160                     self._duty_cycle[self._direction] = 70
1161
1162                     self._pivot(deg_diff,steady=True)
1163
1164 #                 if abs(yaw_diff) >= 0.6:
1165 #                     self._duty_cycle[self._direction] = 70
1166 #
1167 #                     self._pivot(yaw_diff,steady=True)
1168
1169
1170 #                 elif abs(yaw_diff) > 0.6:

```

```

1171 #
1172 #
1173 #         # Return pwm dutcycle back to base
1174 #         self._dutycycle[self._direction] = 70
1175 #
1176
1177         # Return pwm dutcycle back to base
1178         self._dutycycle[self._direction] = self.duty[self._direction]
1179
1180         # Save robots pose by saving its x, yaw position
1181         self._path.append((round(self.global_x,2),round(self.global_y,2)))
1182
1183         self._logger.info("Final orientation reached")
1184         break
1185
1186         yaw_diff_old = yaw_diff
1187         self._yaw = self._imu_serial()
1188         yaw_diff = yaw_final - self._yaw
1189         yaw_diff_list.append(yaw_diff)
1190
1191         if ((np.sign(yaw_diff) != np.sign(yaw_diff_old)) and (abs(yaw_diff) > 5)):
1192             yaw_diff_list.pop()
1193             yaw_diff = yaw_diff_old
1194         # Filtering data via median thresholding
1195         if len(yaw_diff_list) > 6:
1196             median = np.median(yaw_diff_list[-5:-1])
1197         else:
1198             median = np.median(yaw_diff_list[-3:-1])
1199
1200         if (abs(yaw_diff - median) < 5):
1201             yaw_diff_list.append(yaw_diff)
1202         else:
1203             yaw_diff_list.pop()
1204             yaw_diff = yaw_diff_old
1205             yaw_diff_list.append(yaw_diff_old)
1206
1207         except KeyboardInterrupt:
1208             self._logger.error("Pivot Interrupted")
1209             self._gameover()
1210         except Exception as error:
1211             self._logger.error(f"Error {error}")
1212
1213         self._pwmZero()
1214         self._actions[self._task_now] = False
1215         self._logger.info("Pivot Finished")
1216
1217     def _go2block(self, direction='f', eucl_disp=0, just_drive=False, steady=False)
1218         """
1219         This function drives the robot in a straight path, either forward
1220         or reverse
1221
1222         Args:
1223             direction: Direction of drive, forward or reverse
1224
1225         Returns:
1226             None

```



```

1227
1228
1229 self._task_now = 'driving'
1230 self._actions[self._task_now] = True
1231 self._logger.info("Going toward block (go2block)")
1232
1233 # Divide the separation distance into three zones
1234 # First zone is 70% into drive
1235 # Second zone is 20%
1236 # Third zone is the last 10%
1237 # Each of the zones have their own dutycycle. Robot will
1238 # drive and stop at each zone, scan, align, and resume drive
1239 zones = [0.7, 0.85, 0.9]
1240 thresh = 20
1241 self._success = False
1242 # Check if straight drive mode is requested and steady drive mode
1243 if not just_drive and not steady:
1244     # Use image from camera to estimate distance
1245     self._dist_estimate()
1246     # convert depth to encoder count
1247     orig_dist = self._cm2encoder(self._travel-35)
1248
1249     self._logger.info(f"Distance estimated {self._travel}")
1250     self._logger.info(f"Distance to drive {self._travel - 35}")
1251 elif steady:
1252     # Use image from camera to estimate distance
1253     self._dist_estimate()
1254     # convert depth to encoder count
1255     orig_dist = self._cm2encoder(self._travel-5)
1256     self._logger.info(f"Distance estimated {self._travel}")
1257     self._logger.info(f"Driving slowly {self._travel-5} cms")
1258 elif just_drive:
1259     # Drive straight command
1260     orig_dist = self._feet2encoder(eucl_disp)
1261     self._logger.info(f"Just Driving Distance estimated {orig_dist}")
1262 elif just_drive and steady:
1263     # Drive straight command
1264     orig_dist = self._cm2encoder(eucl_disp)
1265     self._logger.info(f"Just Driving Distance estimated {orig_dist}")
1266
1267 self._logger.info(f"Distance to drive {orig_dist} encoder")
1268 pos_encoder_orig = 0
1269 pos_encoder = 0
1270 orig_error = orig_dist
1271
1272 self.counterFL = np.uint64(0)
1273 self.counterBR = np.uint64(0)
1274 self.buttonBR = int(0)
1275 self.buttonFL = int(0)
1276
1277 # Defining the limit of dutycycle
1278 lower_dc = 70
1279 upper_dc = 100
1280
1281 z1 = 80
1282 z2 = z1 - 20

```



```
1283     z3 = z2 - 10
1284
1285     # Divide up the path in different zones
1286     trip = [x * orig_dist for x in zones]
1287
1288     # Determine if function completed successfully
1289     self._success = False
1290
1291     #time.sleep(1)
1292     yaw0 = self._imu_serial()
1293
1294     # For encoder level control
1295     integral_error = 0
1296     derivative_error = 0
1297     error_encoder0 = 0
1298     error_encoder = 0
1299
1300     # For achieving overall distance
1301     integral = 0
1302     derivative = 0
1303     error0 = 0
1304     error = 0
1305     startD = 0
1306
1307     cnt = 0
1308     # Assign direction of drive from argument
1309     self._direction = direction
1310     # Compute the difference in distance between start state and goal state
1311     self._logger.info("Sending command toward thrusters")
1312     self._logger.info(f"Checking encoder to cm {self._encoder2cm(orig_dist)}")
1313
1314     self._picam_frame()
1315     self._img_frame = cntr_bbox.dist_img(self._img_frame, self._encoder2cm(orig_dist))
1316     #cntr_bbox.img_show("Detecting Block",self._img_frame)
1317
1318     start = time.time()
1319
1320     #To avoid division by zero
1321     if abs(orig_dist) < 0.001:
1322         orig_dist = 0.001
1323
1324     x = 0
1325     y = 0
1326
1327     # Save robots pose by saving its x, yaw position
1328     self._yaw = self._imu_serial()
1329     # Convert imu notation to general +ccw from positive x
1330     theta = -self._yaw
1331
1332     while not self._success:
1333
1334         # by considering the minimum of the two encoders
1335         pos_encoder = int(min(self.counterFL, self.counterBR))
1336
1337         # Global error between start and goal state will be
1338         error = orig_dist - pos_encoder
```

```

1339 prcnt = round(1 - error/orig_dist,3)
1340
1341 if (pos_encoder % 150) == 0:
1342
1343     # Save robots pose by saving its x, yaw position
1344     x = self._encoder2feet(pos_encoder) * math.cos(math.radians(theta))
1345     y = self._encoder2feet(pos_encoder) * math.sin(math.radians(theta))
1346
1347     self._path.append((round(self.global_x + x,2),round(self.global_y +
1348
1349 #         if (self._distance() < 30):
1350 if ((10 - self.global_x < 1 or self.global_x < 0.9) or ((10 - self
1351     self._logger.info("Too close to border. Stop and Exit straight
1352     self._pwmZero()
1353     break
1354
1355     #Record encoder states to txt file
1356     outstring = str(round(self.global_x + x,2)) + ' ' + str(round(self
1357     self.f.write(outstring)
1358
1359
1360 if prcnt < zones[0] and self._direction == 'f' and not steady:
1361     if cnt == 0:
1362         lower_dc = 50
1363         upper_dc = 60
1364
1365         error = orig_dist - pos_encoder
1366         pos_encoder_orig = pos_encoder
1367
1368         self.duty[self._direction] = [55,55]
1369         self.dutycycle[self._direction] = self.duty[self._direction]
1370         cnt += 1
1371
1372         self._picam_frame()
1373         self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
1374         self._out.write(self._img_frame)
1375
1376 elif (prcnt >= zones[0] and prcnt < zones[1]) and self._direction == 'f'
1377     if cnt == 1:
1378         lower_dc = 35
1379         upper_dc = 50
1380
1381         pos_encoder_orig += pos_encoder
1382
1383         self.duty[self._direction] = [40,40]
1384         self.dutycycle[self._direction] = self.duty[self._direction]
1385         cnt += 1
1386
1387         self._picam_frame()
1388         self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
1389         self._out.write(self._img_frame)
1390
1391 elif (prcnt >= zones[1] and prcnt < zones[2]) and self._direction == 'f'
1392     if cnt == 1:
1393         lower_dc = 28
1394         upper_dc = 32

```

```

1395         self._pwmZero()
1396         pos_encoder_orig += pos_encoder
1397
1398         self.duty[self._direction] = [30,30]
1399         self._dutycycle[self._direction] = self.duty[self._direction]
1400         cnt += 1
1401
1402         self._picam_frame()
1403         self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
1404         self._out.write(self._img_frame)
1405     elif prcnt >= zones[2]:
1406         if cnt == 2:
1407             lower_dc = 25
1408             upper_dc = 28
1409
1410             pos_encoder_orig += pos_encoder
1411
1412             self.duty[self._direction] = [22,22]
1413             self._dutycycle[self._direction] = self.duty[self._direction]
1414             cnt += 1
1415
1416             self._picam_frame()
1417             self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
1418             self._out.write(self._img_frame)
1419
1420
1421         self._dutycycle[self._direction] = [np.clip(self._dutycycle[self._dire
1422             np.clip(self._dutycycle[self._dire
1423
1424         if steady:
1425             self._dutycycle[self._direction] = [30,30]
1426
1427         if self._direction == 'f':
1428             # Drive robot towards direction
1429             self._forward(tuple(self._dutycycle[self._direction]))
1430         elif self._direction == 'rev':
1431             # Drive robot towards direction
1432             self._reverse(tuple(self._dutycycle[self._direction]))
1433
1434         # Count encoders for left and right motors
1435         if int(gpio.input(12)) != int(self.buttonBR):
1436             self.buttonBR = int(gpio.input(12))
1437             self.counterBR += 1
1438
1439         if int(gpio.input(7)) != int(self.buttonFL):
1440             self.buttonFL = int(gpio.input(7))
1441             self.counterFL += 1
1442
1443         # PD Controller for encoder balance
1444         error_encoder = self.counterFL - self.counterBR
1445
1446         if error_encoder > 25:
1447             # Give power to corresponding motors
1448             self._dutycycle[self._direction][0] -= 0.1 * self.duty[self._dire
1449             self._dutycycle[self._direction][1] += 0.1 * self.duty[self._dire
1450         elif error_encoder < -25:

```

```

1451         # Give power to corresponding motors
1452         self._duty_cycle[self._direction][0] += 0.1 * self.duty[self._direction][0]
1453         self._duty_cycle[self._direction][1] -= 0.1 * self.duty[self._direction][1]
1454     else:
1455         # Give power to corresponding motors
1456         self._duty_cycle[self._direction][0] = self.duty[self._direction][0]
1457         self._duty_cycle[self._direction][1] = self.duty[self._direction][1]
1458
1459
1460     if (error <= thresh) and (direction in ['f', 'rev']):
1461         self._yaw = self._imu_serial()
1462         angle_diff = yaw0 - self._yaw
1463         self._logger.info(f"counterBR: {self.counterBR} counterFL: {self.counterFL}")
1464         self._logger.info(f"Angle shifted: {angle_diff} deg ")
1465         self._logger.info(f"Estimated distance traveled: {self._encoder2cm}")
1466         self.counterFL = np.uint64(0)
1467         self.counterBR = np.uint64(0)
1468         self.buttonBR = int(0)
1469         self.buttonFL = int(0)
1470         self._logger.info("Arrived in front of block")
1471         self._pwmZero()
1472         self._success = True
1473         self._actions[self._task_now] = False
1474
1475         # Convert imu notation to general +ccw from positive x
1476         theta = -self._yaw
1477
1478         # Save robots pose by saving its x, yaw position
1479         x = self._encoder2feet(pos_encoder) * math.cos(math.radians(theta))
1480         y = self._encoder2feet(pos_encoder) * math.sin(math.radians(theta))
1481         self.global_x += x
1482         self.global_y += y
1483         #self.global_yaw += self._yaw
1484
1485         self._path.append((round(self.global_x, 2), round(self.global_y, 2), round(self.global_yaw, 2)))
1486
1487         self._logger.info(f"x value {round(self.global_x, 2)} and y value {round(self.global_y, 2)}")
1488
1489         # Take image and save to video object
1490         self._picam_frame()
1491         self._img_frame = cntr_bbox.dist_img(self._img_frame, self._encoder2cm)
1492         #cntr_bbox.img_show("Detecting Block", self._img_frame)
1493         self._out.write(self._img_frame)
1494
1495         #Record encoder states to txt file
1496         outstring = '#' * 30 + '\n'
1497         self.f.write(outstring)
1498         outstring = 'End of Driving {self._direction}' + '\n'
1499         self.f.write(outstring)
1500         outstring = '#' * 30 + '\n'
1501         self.f.write(outstring)
1502         break
1503
1504     # Return pwm duty cycle back to base
1505     self._duty_cycle[self._direction] = self.duty[self._direction]
1506

```

```

1507 def _scan4object(self):
1508     """
1509     This function scans for block in environment by turning in place
1510     until it finds the block. If it doesn't find the block after full 360
1511     degree turn, it returns false to say not found
1512
1513     Args:
1514         None
1515
1516     Returns:
1517         None
1518     """
1519
1520     self._task_now = 'scanning'
1521     self._actions[self._task_now] = True
1522     self._logger.info(f"Scanning for {self._block_now} block started")
1523
1524     scan_turn = 0
1525
1526     while True:
1527         # Detect object in frame
1528
1529         self._picam_frame()
1530         #self._img_frame, ave_center, bbox_radius, success = self._bbox_detect
1531         self._img_frame, ave_center, bbox_radius, success = cntr_bbox.bbox_det
1532
1533         if success not in [None, False]:
1534             # Object found
1535             diff = ave_center[0] - self.frame_center[0]
1536             # Convert the pixel difference to degrees
1537             self._turn = diff * cntr_bbox.pixel2deg
1538
1539             if (self._turn > 180 and self._turn <= 360):
1540                 self._turn = self._turn - 360
1541
1542             self._yaw = -self._turn
1543             self._path.append((round(self.global_x, 2), round(self.global_y, 2), r
1544
1545             self._success = True
1546             self._status['found'] = True
1547             self._task_now = 'scanning'
1548             self._actions[self._task_now] = False
1549             return self._turn
1550
1551         else:
1552             # Pivot the robot to a predefined turn angle
1553             self._pivot(self.turn)
1554             # Read the distance from obstacle for reference
1555             self._distance()
1556             # Keep track of turn
1557             #self._yaw += self.turn
1558             scan_turn += self.turn
1559             self._yaw += -self._turn
1560             self._path.append((round(self.global_x, 2), round(self.global_y, 2), r
1561
1562         # if scan_turn >= 90:

```

```

1563 #         # Scan the other side
1564 #         self._pivot(-self.turn)
1565 #         # Add scan angle
1566 #         scan_turn += -self.turn
1567 #         self._yaw += self._turn
1568 #         self._path.append((round(self.global_x,2),round(self.global_y,2),
1569 #
1570 #         if scan_turn <= -90:
1571 #             self._status['found'] = False
1572 #             self._success = False
1573 #             self._task_now = 'scanning'
1574 #             self._actions[self._task_now] = False
1575 #             self._logger.warning("No objects in nearby vicinity")
1576 #             return None
1577
1578         if abs(scan_turn) >= 360:
1579             self._status['found'] = False
1580             self._success = False
1581             self._task_now = 'scanning'
1582             self._actions[self._task_now] = False
1583             self._logger.warning("No objects in nearby vicinity")
1584             return None
1585
1586 def _pick_up(self):
1587     """
1588     This function drives the robot to steadily approach and picks up toy block
1589
1590     Args:
1591         None
1592
1593     Returns:
1594         None
1595     """
1596     self._success = False
1597     self._task_now = 'picking'
1598     self._actions[self._task_now] = True
1599
1600     # Take image and save to video object
1601     self._picam_frame()
1602     #cntr_bbox.img_show("Detecting Block",self._img_frame)
1603     self._out.write(self._img_frame)
1604
1605     while not self._success:
1606         # Open gripper
1607         self._servo_cntrl(self.grip_state['opened'])
1608         self._gripper_state = self.grip_state['opened']
1609         self._logger.info("Servo Opened")
1610         # Take image and save to video object
1611         self._picam_frame()
1612         #cntr_bbox.img_show("Detecting Block",self._img_frame)
1613         self._out.write(self._img_frame)
1614         # Align robot fine turn rotation
1615         degree = self._scan4object()
1616
1617     #         while True:
1618     #             if degree is not None:

```

```

1619 #         if abs(degree) > 0.6:
1620 #             self._pivot(degree, steady=True)
1621 #         else:
1622 #             break
1623 #     else:
1624 #         break
1625 # Scan block and drive slowly towards it
1626 self._logger.info("Driving slowly toward block")
1627 self._go2block(steady=True)
1628 self._logger.info("Slow drive completed")
1629
1630 # Take image and save to video object
1631 self._picam_frame()
1632 #cntr_bbox.img_show("Detecting Block",self._img_frame)
1633 self._out.write(self._img_frame)
1634
1635 # Close gripper
1636 self._servo_cntrl(self.grip_state['closed'])
1637 self._gripper_state = self.grip_state['closed']
1638 self._logger.info("Servo Closed")
1639
1640 # Take image and save to video object
1641 self._picam_frame()
1642 #cntr_bbox.img_show("Detecting Block",self._img_frame)
1643 self._out.write(self._img_frame)
1644
1645 #if (abs(radius2 - radius1) < 5) or (abs(center2[0] - center1[0]) < 3)
1646 self._logger.info(f"The {self._block_now} block has been gripped succe:
1647 self._success = True
1648 self._status['picked_up'] = True
1649 self._actions[self._task_now] = False
1650
1651 # Read saved image and send via email
1652 self._email_media()
1653
1654 def _go2(self,destination=(0,0),area=''):
1655     """
1656     This function computes the difference in pose between the robot and the de:
1657     then estimates the displacement vector along with the angle of rotation nee:
1658     to align the robot head on. It then calls the pivot and go2block functions:
1659     and command the robot to enter the construction zone.
1660
1661     Args:
1662         destination (tuple): contans x,y location of destination
1663         area (str): Describes where that destination is
1664
1665     Returns:
1666         None
1667     """
1668     self._logger.info(f"Initiating travel to {area}")
1669     self._success = False
1670     self._task_now = 'returning'
1671     self._actions[self._task_now] = False
1672
1673     # Read last position from the appended path
1674     # self.global_x = self._path[-1][0]

```



```

1675 # self.global_y = self._path[-1][1]
1676 #self.global_yaw = self._path[-1][2]
1677 delta_x = destination[0]-self.global_x
1678 delta_y = destination[1]-self.global_y
1679 self._yaw = self._imu_serial()
1680
1681 self._logger.info(f"Pose Now: X- {self.global_x} and Y- {self.global_y} and")
1682 # Compute the euclidean distance between current robot location and constr
1683 disp = math.sqrt((delta_x)**2 + (delta_y)**2)
1684 # Calculate the yaw angle difference between these two points using slope
1685 rad_diff = math.atan2(delta_y,delta_x)
1686 # Convert from radians to angle
1687 target_angle = math.degrees(rad_diff)
1688 # Ensure angle is between [0, 360]
1689 target_angle %= 360
1690
1691 self._logger.info(f"Computed target angle is {target_angle}")
1692
1693 # # Place angle from [0,360] to [-180,0) U [0,180]
1694 if target_angle >=-180 and target_angle <=180:
1695     target_angle = -target_angle
1696 elif (target_angle >= 180 and target_angle <360):
1697     target_angle = 360 - target_angle
1698 elif (target_angle > -360 and target_angle <=-180):
1699     target_angle = -(target_angle + 360)
1700
1701 self._logger.info(f"Normalized target angle is {target_angle}")
1702 # # Normalize angle difference to range [-180, 180) degrees
1703 # target_angle = (target_angle + 180) % 360 - 180
1704 # # Angle difference from current and target
1705 # deg_diff = self._yaw - target_angle
1706 # deg_diff %= 360
1707
1708 # Based on the quadrant of the point heading towards normalize
1709 # the target angle
1710 # if delta_x < 0:
1711 #     if delta_y > 0:
1712 #         #target_angle = 180 - target_angle
1713 #         pass
1714 #     else:
1715 # # target_angle = target_angle - 180
1716 #         pass
1717 # else:
1718 #     if delta_y < 0:
1719 # # target_angle = -target_angle
1720 #         pass
1721
1722 # convert target angle with respect to imu configuration
1723 # target_angle = -target_angle
1724
1725 # Angle difference from current and target
1726 deg_diff = target_angle - self._yaw
1727 #deg_diff %= 360
1728
1729 # if (deg_diff > 180 and deg_diff <360):
1730 #     deg_diff = (deg_diff - 360) # Turning to the right

```



```

1731 #         elif (deg_diff > -360 and deg_diff <=-180):
1732 #             deg_diff = (deg_diff + 360) # Turning to the left
1733
1734 #         # Place angle from [0,360] to [-180,0) U [0,180]
1735 #         if (deg_diff > 180 and deg_diff <=360):
1736 #             deg_diff = deg_diff - 360
1737 #         # Normalize angle difference to range [-180, 180) degrees
1738 #         deg_diff = (deg_diff + 180) % 360 - 180
1739
1740 self._logger.info(f"Yaw that vehicle will pivot to {deg_diff}")
1741 # Pivot robot to face construction zone
1742 self._pivot(deg_diff, just_turn=True)
1743 self._logger.info(f"Started driving straight to {area}")
1744 # Drive robot forward to construction zone
1745 self._go2block(eucl_disp = disp, just_drive = True)
1746 self._logger.info(f"Arrived at {area}")
1747
1748 # New global yaw
1749 self.global_yaw = self._imu_serial()
1750
1751 self._success = True
1752 self._status['arrived'] = True
1753 self._actions[self._task_now] = False
1754
1755 def _place(self):
1756     """
1757     This function places toy block
1758
1759     Args:
1760         None
1761
1762     Returns:
1763         None
1764     """
1765     self._success = False
1766     self._task_now = 'placing'
1767     self._actions[self._task_now] = True
1768
1769     # Open gripper
1770     self._servo_cntrl(self.grip_state['opened'])
1771     self._gripper_state = self.grip_state['opened']
1772     self._logger.info(f"Gripper Opened to place {self._block_now} block")
1773
1774     # Drive in reverse
1775     self._go2block(direction='rev', eucl_disp=7.5, just_drive=True, steady=True)
1776     time.sleep(1)
1777     self._logger.info("Backed away from toy block")
1778
1779     # Close gripper
1780     self._servo_cntrl(self.grip_state['closed'])
1781     self._gripper_state = self.grip_state['closed']
1782     self._logger.info("Gripper Closed to prepare for next order")
1783
1784     self._success = True
1785     self._status['placed'] = True
1786     self._actions[self._task_now] = False

```

```

1787
1788     self._logger.info(f"{self._block_now} block is now placed inside construct:
1789
1790     # Read saved image and send via email
1791     self._email_media()
1792
1793 def _localize(self):
1794     """
1795     This function uses distance sensor to read from borderwall
1796     so that the robot's exact position in arena can easily be
1797     identified
1798
1799     Args:
1800         None
1801
1802     Returns:
1803         None
1804     """
1805     self._success = False
1806     self._task_now = 'localizing'
1807     self._actions[self._task_now] = True
1808
1809     # pivot the robot to positive y first and read from distance sensor
1810     yaw_final = -90
1811     yaw_current = self._imu_serial()
1812     deg_diff = yaw_final - yaw_current
1813
1814     #deg_diff %= 360
1815     # Normalize angle difference to range [-180, 180) degrees
1816     if (deg_diff > 180 and deg_diff < 360):
1817         deg_diff = 360 - deg_diff
1818     elif (deg_diff > -360 and deg_diff <= -180):
1819         deg_diff = -(deg_diff + 360)
1820
1821     # Turn towards borderwall
1822     self._pivot(deg_diff, just_turn=True)
1823
1824     # Now read distance sensor value
1825     self._distance()
1826     delta_ft = self._cm2feet(self._depth)
1827     self._yaw = self._imu_serial()
1828     theta = -self._yaw
1829     x1 = delta_ft * math.cos(math.radians(theta))
1830     y1 = delta_ft * math.sin(math.radians(theta))
1831
1832     # Extract y pose by normalizing from border wall
1833     if self.global_y > 5:
1834         self.global_y = 10 - y1
1835     else:
1836         self.global_y = y1
1837
1838     # pivot the robot to negative x first and read from distance sensor
1839     yaw_final = -179
1840     yaw_current = self._imu_serial()
1841     deg_diff = yaw_final - yaw_current
1842     #deg_diff %= 360

```

```

1843
1844     # Normalize angle difference to range [-180, 180] degrees
1845     if (deg_diff > 180 and deg_diff < 360):
1846         deg_diff = 360 - deg_diff
1847     elif (deg_diff > -360 and deg_diff <= -180):
1848         deg_diff = -(deg_diff + 360)
1849
1850     # Turn towards borderwall
1851     self._pivot(deg_diff, just_turn=True)
1852
1853     # Now read distance sensor value
1854     self._distance()
1855     delta_ft = self._cm2feet(self._depth)
1856     self._yaw = self._imu_serial()
1857     theta = -self._yaw
1858     x2 = delta_ft * math.cos(math.radians(theta))
1859     y2 = delta_ft * math.sin(math.radians(theta))
1860
1861     # Extract x pose by normalizing from border wall
1862     if self._x > 5:
1863         self.global_x = 10 - x2
1864     else:
1865         self.global_x = x2
1866
1867     # Add detected pose of robot into path list
1868     self._path.append((round(self.global_x,2),round(self.global_y,2),round(self.global_yaw,2)))
1869
1870     self._success = True
1871     self._actions[self._task_now] = False
1872     self._logger.info(f"Localization complete")
1873
1874 def _go2site(self):
1875     """
1876     This function commands the robot to drive back to the center
1877     of the arena or near the cluttered environment to resume delivering
1878     order
1879
1880     Args:
1881         None
1882
1883     Returns:
1884         None
1885     """
1886
1887     self._logger.info("Initiating travel to cluttered environment")
1888     # Compute the euclidean distance between current robot location and constr
1889     disp = math.sqrt((self._x - self.clutter_env[0])**2 + (self._y - self.clut
1890     # Calculate the yaw angle difference between these two points using slope
1891     rad_diff = math.atan2(self.clutter_env[0] - self._x, self.clutter_env[1] -
1892     # Convert from radians to angle
1893     deg_diff = math.degrees(rad_diff)
1894     # Ensure angle is between [0, 360]
1895     deg_diff %= 360
1896     # Place angle from [0,360] to [-180,0) U [0,180]
1897     if (deg_diff > 180 and deg_diff <= 360):
1898         deg_diff = deg_diff - 360

```

```

1899
1900     self._logger.info("Orienting to cluttered environment")
1901     # Pivot robot to face construction zone
1902     self._pivot(deg_diff)
1903     self._logger.info("Started driving straight to cluttered environment")
1904     # Drive robot foward to construction zone
1905     self._go2block(eucl_disp = disp, just_drive = True)
1906     self._logger.info("Arrived at cluttered environment")
1907
1908     def _go_home(self):
1909         """
1910         This function commands the robot to drive back to its home location
1911         marking the end of the challenge.
1912
1913         Args:
1914             None
1915
1916         Returns:
1917             None
1918         """
1919         self._logger.info("Initiating travel to home base")
1920         # Compute the euclidean distance between current robot location and constr
1921         disp = math.sqrt((self._x - self.start_pt[0])**2 + (self._y - self.start_p
1922         # Calculate the yaw angle difference between these two points using slope
1923         rad_diff = math.atan2(self.start_pt[0] - self._x, self.start_pt[1] - self.
1924         # Convert from radians to angle
1925         deg_diff = math.degrees(rad_diff)
1926         # Ensure angle is between [0, 360]
1927         deg_diff %= 360
1928         # Place angle from [0,360] to [-180,0) U [0,180]
1929         if (deg_diff > 180 and deg_diff <=360):
1930             deg_diff = deg_diff - 360
1931
1932         self._logger.info("Orienting to home base")
1933         # Pivot robot to face construction zone
1934         self._pivot(deg_diff)
1935         self._logger.info("Started driving straight to home base")
1936         # Drive robot foward to construction zone
1937         self._go2block(eucl_disp = disp, just_drive = True)
1938         self._logger.info("Arrived at home")
1939         self._logger.info("Thank you for playing game.")
1940         self._logger.info("Powering off.")
1941         self._end_game = True
1942
1943     def plot_path(self):
1944         """
1945         Plot the path of the robot during the grand challenge.
1946
1947         Args:
1948             None
1949         Return:
1950             None
1951         """
1952         # Take in the path List as an array 2D NumPy array
1953         path = np.array(self._path)
1954         x = path[:,0]

```

```

1955     y = path[:,1]
1956
1957     # Create figure and plot
1958     fig, ax = plt.subplots(1,1)
1959
1960     fig.suptitle('Motor Encoder Analysis')
1961
1962     ax.plot(x,y,ls='solid', color='blue',linewidth=2, label=f'{self._name} Path')
1963
1964     # 3) Label the axes, title, legend
1965     ax.set_ylabel('Y')
1966     ax.set_xlabel('X')
1967     ax.set_xlim(0,10)
1968     ax.set_ylim(0,10)
1969     plt.savefig('grandChallenge-robotPath.png')
1970     plt.show()
1971     plt.close()
1972
1973     def start(self, order: List):
1974         """
1975         The sequence of actions occuring while baron robot works to pick and place
1976         all toy blocks to the construction zone. This function lists the steps needed
1977         for the robot to complete the orders passed to it at the beginning of competition.
1978
1979         Args:
1980             order: List[Blocks] List of all toy blocks needed to complete challenge
1981
1982         Returns:
1983             None
1984         """
1985
1986         try:
1987             # Assign order from object initialization
1988             self._order = order
1989
1990             # First wait until startup QR Code has been read
1991             while True:
1992                 if self._wait2start():
1993                     break
1994
1995             while len(self._order) > 0:
1996                 """While Robot is still active, resume challenge"""
1997                 if (not self._end_game):
1998
1999                     self._block_now = self._order.pop(0)
2000                     self._order_no +=1
2001
2002                     # Begin challenge with choesn block
2003                     self._logger.info("-" * 20)
2004                     self._logger.info(f"*** Picking Up {self._block_now} block")
2005                     self._logger.info("-" * 20)
2006
2007                     # Track the number of times robot drives to search area
2008                     search = 0
2009
2010                     # Drive robot to center of arena, get view of toy blocks and blocks

```

```

2011 # extracting blocks
2012 self._go2(destination=self.clutter_env,area='cluttered environ
2013
2014 while True:
2015
2016     # Begin by scanning for toy block object
2017     degree = self._scan4object()
2018
2019     # Take image and save to video object
2020     #cntr_bbox.img_show("Detecting Block",self._img_frame)
2021     self._out.write(self._img_frame)
2022
2023     # Drive toward block if successful, else pivot to
2024     # a free space and drive
2025     if self._success:
2026         #self._logger(f"Pivoting Robot towards {self._block_now}
2027         self._pivot(degree,steady=True)
2028
2029         # Take image and save to video object
2030         self._picam_frame()
2031         __, __, __, __ = cntr_bbox.bbox_detect(self._img_frame, :
2032         #cntr_bbox.img_show("Detecting Block",self._img_frame)
2033         self._out.write(self._img_frame)
2034
2035         self._logger.info(f"Drive Robot forward to {self._block_now}
2036         self._go2block()
2037
2038         # # Update global x and y
2039         # self.global_x += self._x
2040         # self.global_y += self._y
2041
2042         ##self._logger(f"{self._name} is now at x- {self.global_x} y- {self.global_y}")
2043
2044         # Take image and save to video object
2045         self._picam_frame()
2046         #cntr_bbox.img_show("Arrived at Block",self._img_frame)
2047         self._out.write(self._img_frame)
2048
2049         ##self._logger(f"Pick up {self._block_now} block with {self._block_now}")
2050         self._pick_up()
2051
2052         # Drive robot to construction zone to place block
2053         ##self._logger(f"Take {self._block_now} block to construction zone")
2054         self._go2(destination=self.const_zone,area='construction zone')
2055
2056         #self._logger(f"Place {self._block_now} block inside construction zone")
2057         self._place()
2058
2059         #self._logger("Localizing to get accurate location of block")
2060         self._localize()
2061
2062         ##self._logger(f"Pick up and place the next order in the list")
2063         self._success = True
2064         break
2065     else:
2066         self._logger("No toy blocks found in current pose. Check environment")

```

```

2067         #self._logger("try again.")
2068
2069         # Drive robot to the predetermined survey locations
2070         if search > 1:
2071             self._go2(destination=self._survey_loc[search],area=
2072         else:
2073             #self._logger(f"Robot can't find {self._block_now}
2074             #self._logger("In pursuit of time, moving on to ne
2075             break
2076             search += 1
2077
2078     else:
2079         #self._logger(f"Mission complete. Take robot back to home base")
2080         self._go2(destination=self.start_pt,area='home base')
2081         return
2082
2083     except KeyboardInterrupt as error:
2084         self._logger.error("Error occurred: %s",error)
2085         self._gameover()
2086         time.sleep(1)
2087         return None
2088     except ValueError as error:
2089         self._logger.error("Error %s", error, exc_info=True)
2090         self._gameover()
2091         time.sleep(1)
2092         return None
2093

```

class Robot:

[▼ View Source](#)

```

41 class Robot(object):
42     """
43     A class defining basic attributes of Baron robot that
44     will be autonomously achieving objective
45
46     Attributes:
47         name (str): Name of robot
48         _x (float): horizontal position of robot from start postion
49         _y (float): vertical position of robot from start position
50         _yaw (float): yaw orientation from +x as horizontal [0,180] or (-180,0)
51         _travel(float): estimating the distance to drive and reach block
52         _turn (float): keeps the next turn needed by block
53         _depth (float): measures distance of obstacle in front of robot from image
54         _path (list): stores the path traveled by robot as list of (x,y) points and
55         _pwms (list): stores a list of pwms for the robot motors
56         _duty cycle (dict): holds the baseline pwm duty cycle for forward,reverse,
57             pivotleft and pivotright
58         _success (bool): keeps track whether a given task is successful or not
59         _gripper_state (str): stores gripper duty cycle state as opened or closed
60         _order(list): List of blocks to pickup
61         _end_game (bool): tells competition is over by victory or failure
62         _order_no (int): keeps track of number of orders completed
63         _block_now (str): the block being pursued now

```



```

64     _task_now (str): keeps track of the task being completed at the moment
65     _actions (dict): holding list of all service actions robot performs with b
66                     on which process is being done at a given time
67     _status (dict): a dictionary keeping track of completed actions along with
68     _img_frame (img): saves current image frame
69     _out (cv2 video obj): gather all frames into a video
70     _logger (log): Log statements for debugging and status over the terminal
71     _kp (float): proportional gain
72     _ki (float): integral gain
73     _kd (float): derivative gain
74     _direction (str): tells weather robot is moving forward or reverse
75     _event (event): event object to end the thread call
76     _pause_event (event): event object to pause and resume thread
77     imu_reader (thread): threading object to read from imu during forward run
78     _ser (serial): object reading serial input streams from imu sensor
79     _img_saved (img): keeping track of saved image to send in an email
80     _survey_loc (List): contains x,y locations for the robot to navigate and :
81                       so that to extract toy blocks for mission
82
83     ""
84
85     # define the codec and create VideoWriter object
86     fourcc = cv2.VideoWriter_fourcc(*'XVID')
87     # Gripper States
88     grip_state = dict([('closed',2.5),('opened',7.5)])
89     # Matrix for Image frame
90     image = np.empty((480*640*3,),dtype=np.uint8)
91     # define the codec and create VideoWriter object
92     fourcc = cv2.VideoWriter_fourcc(*'XVID')
93     # Open .txt file to save data
94     f = open('robot_path_final.txt','a')
95
96     dutysset = [('f', [70,70]), ('rev', [30,30]),('l', 100),('r',100)]
97
98     duty = dict(dutysset)
99     font = cv2.FONT_HERSHEY_COMPLEX_SMALL
100
101     # turn angel
102     turn = 15
103
104     # detect radius
105     bbox_radius = 0
106
107     # Initialize FL and BR button count
108     counterBR = np.uint64(0)
109     counterFL = np.uint64(0)
110     buttonBR = int(0)
111     buttonFL = int(0)
112
113     # Proportional Control Gain
114     Kp = 1
115     Ki = 0.05
116     Kd = 0.08
117
118     # Important Locations in map in units of feet
119     origin = [0,0]

```



```

120     start_pt = [1,1]
121     clutter_env = [5,5]
122     const_zone = [2.25,6.5]
123
124     # create object to read camera
125     video = cv2.VideoCapture(0)
126
127     frame_center = (320,240)
128     (cx,cy) = (frame_center[0]/4.5, frame_center[1]/8)
129
130     queue = queue.Queue()
131
132     img_folder = 'challenge_pics'
133
134     counter = 0
135
136     # Environment assessment points
137     scan_spots = [(7.5,3),(7.5,8)]
138
139     # Global x, y, positions and yaw from fixed reference
140     # frame (0,0)
141     global_x = 1
142     global_y = 1
143     global_yaw = 0
144
145     def __init__(self, name: str):
146         """
147         Initialize Robot attributes
148
149         Args:
150         name: str - takes name of robot
151
152         Returns:
153         None
154         """
155
156         self.name = name
157         self._x = self.start_pt[0]
158         self._y = self.start_pt[1]
159         self._yaw = 0.0
160         self._travel = 0
161         self._turn = 0
162         self._depth = 0
163         self._path = [(1,1,0)]
164         self._pwms = []
165         self._duty_cycle = self.duty
166         self._success = False
167         self._gripper_state = self.grip_state['closed']
168         self._order = []
169         self._end_game = False
170         self._order_no = 0
171         self._block_now = ''
172         self._task_now = ''
173         self._actions = dict([('driving',False),('rotating',False),('picking',False),('dropping',False)])
174         self._status = dict([('started',False),('picked_up',False),('found',False),('lost',False)])
175         self._img_frame = self.image

```

```

176     self._out = cv2.VideoWriter('grand_chall_vid.avi', self.fourcc, 3, (640, 480))
177     self._logger = logging.getLogger(name)
178     self._kp = self.Kp
179     self._ki = self.Ki
180     self._kd = self.Kd
181     self._direction = 'f'
182     self._event = threading.Event()
183     self._pause_event = threading.Event()
184     self.imu_reader = threading.Thread(target = self._imu_serial_straight, args=())
185     self._ser = serial.Serial('/dev/ttyUSB0', 19200)
186     self._img_saved = self.image
187     self._survey_loc = self.scan_spots
188
189     # Set up logger with Debug initially
190     self._logger.setLevel(logging.DEBUG)
191     # Create console handler and set level to debug
192     console_handler = logging.StreamHandler()
193     console_handler.setLevel(logging.DEBUG)
194     # Log terminal messages to a file
195     file_handler = logging.FileHandler('gc_log_3.log')
196     file_handler.setLevel(logging.INFO)
197     # Create formatter
198     formatter = logging.Formatter('%(name)s - %(levelname)s - %(message)s')
199     # Add formatter to handler
200     console_handler.setFormatter(formatter)
201     file_handler.setFormatter(formatter)
202     # Add console handler to logger
203     self._logger.addHandler(console_handler)
204     self._logger.addHandler(file_handler)
205     # Initialize gpios and pwms
206     self._init()
207
208
209     @property
210     def name(self):
211         return self._name
212
213     @name.setter
214     def name(self, name):
215         self._name = name
216
217     def _init(self):
218         """
219         Initialize gpio pins and set pwms to command DC motors, encoders and
220         servo
221
222         Args:
223             None
224
225         Returns:
226             None
227         """
228
229         gpio.cleanup()
230         gpio.setmode(gpio.BOARD)
231

```

```
232     # Setup GPIO pin(s)
233     gpio.setup(36, gpio.OUT) # Servo
234
235     gpio.setup(31, gpio.OUT) # IN1
236     gpio.setup(33, gpio.OUT) # IN2
237     gpio.setup(35, gpio.OUT) # IN3
238     gpio.setup(37, gpio.OUT) # IN4
239
240     gpio.setup(7, gpio.IN, pull_up_down = gpio.PUD_UP)
241     gpio.setup(12, gpio.IN, pull_up_down = gpio.PUD_UP)
242
243     self._pwms.clear()
244
245     # initialize pwm signal to control motor
246     pwm01 = gpio.PWM(31, 50) # BackLeft motor
247     pwm11 = gpio.PWM(33, 50) # FrontLeft motor
248     pwm22 = gpio.PWM(35, 50) # FrontRight motor
249     pwm02 = gpio.PWM(37, 50) # BackRight motor
250     pwmS = gpio.PWM(36, 50) # Servo
251
252     self._pwms = [pwm01, pwm11, pwm22, pwm02, pwmS]
253
254     for pwm in self._pwms:
255         pwm.start(0)
256
257     def _pwmZero(self):
258         """
259         Stops motors by zeroing pwm values in all pins
260         controlling DC motors
261
262         Args:
263             None
264
265         Returns:
266             None
267         """
268
269
270         self._pwms[0].ChangeDutyCycle(0)
271         self._pwms[1].ChangeDutyCycle(0)
272         self._pwms[2].ChangeDutyCycle(0)
273         self._pwms[3].ChangeDutyCycle(0)
274
275     def _gameover(self):
276         """
277         Terminates run by stopping pwms and cleaning up
278         gpio pins
279
280         Args:
281             None
282
283         Returns:
284             None
285         """
286
287         self._pwmZero()
```

```
288         self._pwms[-1].ChangeDutyCycle(self.grip_state['closed'])
289         time.sleep(1)
290         for pwm in self._pwms:
291             pwm.stop()
292         gpio.cleanup()
293
294     def _forward(self,vals):
295         """
296         Commands DC motors to drive robot forward by sending
297         Dutycycles to respective pwm pins
298
299         Args:
300             None
301
302         Returns:
303             None
304         """
305         # Left wheels
306         self._pwms[0].ChangeDutyCycle(vals[0])
307         self._pwms[1].ChangeDutyCycle(0)
308         # Right wheels
309         self._pwms[2].ChangeDutyCycle(0)
310         self._pwms[3].ChangeDutyCycle(vals[1])
311
312     def _reverse(self,vals):
313         """
314         Commands DC motors to drive robot in reverses by sending
315         Dutycycles to respective pwm pins
316
317         Args:
318             None
319
320         Returns:
321             None
322         """
323         # Left wheels
324         self._pwms[0].ChangeDutyCycle(0)
325         self._pwms[1].ChangeDutyCycle(vals[0])
326         # Right wheels
327         self._pwms[2].ChangeDutyCycle(vals[1])
328         self._pwms[3].ChangeDutyCycle(0)
329
330     def _pivotleft(self,vals):
331         """
332         Commands DC motors to pivot robot Left by sending
333         Dutycycles to respective pwm pins
334
335         Args:
336             None
337
338         Returns:
339             None
340         """
341         # Left wheels
342         self._pwms[0].ChangeDutyCycle(0)
343         self._pwms[1].ChangeDutyCycle(vals)
```

```
344     # Right wheels
345     self._pwms[2].ChangeDutyCycle(0)
346     self._pwms[3].ChangeDutyCycle(vals)
347
348     def _pivotright(self,vals):
349         """
350         Commands DC motors to pivot robot right by sending
351         Dutycycles to respective pwm pins
352
353         Args:
354             None
355
356         Returns:
357             None
358         """
359
360         # Left wheels
361         self._pwms[0].ChangeDutyCycle(vals)
362         self._pwms[1].ChangeDutyCycle(0)
363         # Right wheels
364         self._pwms[2].ChangeDutyCycle(vals)
365         self._pwms[3].ChangeDutyCycle(0)
366
367     def _servo_cntrl(self,duty_cycle):
368         """
369         Commands servo motors to open and close gripper by sending
370         Dutycycles to respective pwm pins
371
372         Args:
373             None
374
375         Returns:
376             None
377         """
378
379         self._pwms[-1].ChangeDutyCycle(duty_cycle)
380         time.sleep(1)
381
382         img = cntr_bbox.servo_img(duty_cycle)
383         self._img_frame = cntr_bbox.dist_img(img, self._distance())
384
385         #cntr_bbox.img_show("Servo status",self._img_frame)
386         self._out.write(img)
387
388     def _picam_frame(self):
389         """
390         Calls picamera object to take image frame from the pi camera
391         and applies corrections by flipping to reflect reality
392
393         Args:
394             None
395
396         Returns:
397             None
398         """
399         # Take picture with camera
```

```
400     cntr_bbox.camera.capture(self.image, format="bgr")
401     image = self.image.reshape((480,640,3))
402
403     self._img_frame = cv2.flip(image,1)
404
405     def _mask_color(self, imageHSV):
406
407         if self._block_now == 'green':
408             # Trail Green Block - LAB
409             minHSV = np.array([46,88,121])
410             maxHSV = np.array([65,141,255])
411         elif self._block_now == 'red':
412             # Trail Red block - LAB
413             minHSV = np.array([151,107,147])
414             maxHSV = np.array([255,255,255])
415         elif self._block_now == 'blue':
416             # Trail Blue block - LAB
417             minHSV = np.array([72,94,97])
418             maxHSV = np.array([126,188,212])
419
420         maskHSV = cv2.inRange(imageHSV, minHSV, maxHSV)
421
422         return maskHSV
423
424     def _prep_image(self):
425         """
426         Processes images by applying HSV masking and gaussian blur
427         for use by latter object detection functions
428
429         Args:
430             None
431
432         Returns:
433             None
434         """
435
436         # Convert image from BGR to HSV space
437         imageHSV = cv2.cvtColor(self._img_frame, cv2.COLOR_BGR2HSV)
438
439         # mask the green light from HSV and convert to grayscale
440         mask = self._mask_color(imageHSV)
441
442         # Mask HSV masked image of arrow
443         blurred = cv2.GaussianBlur(mask, (11,11), 0)
444
445         return blurred
446
447     def _grip_checker(self):
448         """
449         Processes images by applying a rectangular mask over desired
450         region of interest to verify whether gripper has held toy block
451
452         Args:
453             None
454
455         Returns:
```

```

456         self._img_frame (img): Masked image
457         """
458         self._logger.info("Inside Grip Checker Function")
459         # Create a mask of gripped block in image
460         # Save the shape of the image array
461         (height, width, c) = self._img_frame.shape
462
463         # Create mask of image just like picture
464         mask = np.zeros_like(self._img_frame[:, :, 0])
465
466         # Create a white rectangle
467         x,y,w,h = round(width/2)-130, round(height/2)-30, 250, 300
468         cv2.rectangle(mask, (x,y), (x+w, y+h), 255, -1)
469
470         # Extract the image that aligns in the rectangle
471         self._img_frame = cv2.bitwise_and(self._img_frame, self._img_frame, mask=mask)
472
473         return self._img_frame
474
475     def _wait2start(self):
476         """
477         Waits for the competition start QR code to be visible,
478         and reads QR code in, which if corresponds to "ENPM701"
479         returns a True boolean to begin competition
480
481         Args:
482         None
483
484         Returns:
485         start (bool): True/False based on detection of expected QR
486         """
487         # Define detector
488         detector = cv2.QRCodeDetector()
489
490         # Check if program start initiated
491         start = False
492
493         cnt = 0
494
495         self._logger.info("Waiting for start cue . . . ")
496
497         try:
498
499             for frame in cntr_bbox.camera.capture_continuous(cntr_bbox.rawCapture,
500
501                 # grab the current frame
502                 img = frame.array
503
504                 img = cv2.flip(img,1)
505
506                 data, bbox, _ = detector.detectAndDecode(img)
507
508                 if(bbox is not None):
509                     for i in range(len(bbox)):
510                         cv2.line(img, tuple(bbox[i][0]),tuple(bbox[(i+1)%len(bbox)]
511                         cv2.putText(img, data, (int(bbox[0][0][0]),int(bbox[0][0][1]

```

```

512
513         if data:
514             self._logger.info(f"Data: {data}")
515             pass
516
517         # Show results to the screen
518         cv2.imshow("QR Code detector",img)
519         key = cv2.waitKey(1) & 0xFF
520
521         # write frame into file
522         self._out.write(img)
523
524         # clear the stream in preparation for the next frame
525         cntr_bbox.rawCapture.truncate(0)
526
527         # Break out of loop by pressing the q key
528         if(key == ord("q")):
529             self._logger.warn("Program Terminated")
530             self.video.release()
531             cv2.destroyAllWindows()
532             break
533
534         if data == "ENPM701":
535             self._logger.info("Cue Received \n")
536             self._logger.info('@' * 45)
537             self._logger.info("Starting Grand Challenge! \n")
538             self._logger.info('#' * 45)
539             start = True
540             self.video.release()
541             cv2.destroyAllWindows()
542             time.sleep(3)
543             break
544
545     except Exception as error:
546         self._logger.error(error)
547         self.video.release()
548         cv2.destroyAllWindows()
549
550     return start
551
552     def _email_media(self):
553
554         # Take a picture of gripped block to email
555         self._picam_frame()
556         # Append status of servo on image
557         self._img_frame = cntr_bbox.servo_img(self._gripper_state)
558         self._out.write(self._img_frame)
559         # Save to designated directory
560         cv2.imwrite(f'{self.img_folder}/{self._block_now}_{self.counter}.png', self
561         #time.sleep(1)
562         # Read saved image
563         img_name = f'{self.img_folder}/{self._block_now}_{self.counter}.png'
564
565         # send email to user with images
566         smtpUser = 'ykebede2@terpmail.umd.edu'
567         smtpPass = 'QwE@d1219'

```



```
568
569     toAdd = ['ENPM809TS19@gmail.com', 'yosephcollege@gmail.com']
570     fromAdd = smtpUser
571
572     f_time = datetime.now().strftime('%a %d %b @ %H:%M')
573     subject = f'ENPM701-GrandChallenge-{f_time}-yosephK-{self._block_now}-{self._counter}'
574
575     self.counter += 1
576
577     msg = MIMEMultipart()
578     msg['Subject'] = subject
579     msg['From'] = fromAdd
580     msg['To'] = ",".join(toAdd)
581
582     msg.preamble = "Image @ " + f_time
583
584     body = email.mime.text.MIMEText("Baron Robot image: " + f_time)
585     msg.attach(body)
586
587     fp = open(img_name, 'rb')
588     img = MIMEImage(fp.read())
589     fp.close()
590     msg.attach(img)
591
592     s = smtplib.SMTP('smtp.gmail.com', 587)
593     s.ehlo()
594     s.starttls()
595     s.ehlo()
596     s.login(smtpUser, smtpPass)
597     s.sendmail(fromAdd, toAdd, msg.as_string())
598     s.quit()
599
600     self._logger.info("Email delivered!")
601
602     def _meter2encoder(self, x_dist):
603         """
604         Reads in distance values in meters and converts to
605         encoder counts
606
607         Args:
608             None
609
610         Returns:
611             encod (int): converted encoder counts from distance in m
612         """
613         encod = round(float(x_dist / (2*math.pi*0.0325))*960)
614
615         return encod
616
617     def _encoder2meter(self, encd):
618         """
619         Reads in encoder values and converts them to meters
620
621         Args:
622             None
623
```

```
624     Returns:
625         cms (float): converted encoder counts to distance in m
626     """
627     dist = round(float((encd / 960) * 2*math.pi*0.0325),4)
628
629     return dist
630
631 def _cm2encoder(self,x_dist):
632     """
633     Reads in distance values in centimeters and converts to
634     encoder counts
635
636     Args:
637         None
638
639     Returns:
640         encod (int): converted encoder counts from distance in cms
641     """
642     dist = x_dist/100
643     encod = self._meter2encoder(dist)
644     return encod
645
646 def _encoder2cm(self,encd):
647     """
648     Reads in encoder values and converts them to centimeters
649
650     Args:
651         None
652
653     Returns:
654         cms (float): converted encoder counts to distance in cms
655     """
656     dist = self._encoder2meter(encd)
657     cms = round(dist*100,4)
658
659     return cms
660
661 def _feet2encoder(self,x_dist):
662     """
663     Reads in distance values in feet and converts to
664     encoder counts
665
666     Args:
667         None
668
669     Returns:
670         encod (int): converted encoder counts from distance in feet
671     """
672     dist = round(float(x_dist * 0.3048),4)
673     encod = self._meter2encoder(dist)
674     return encod
675
676 def _encoder2feet(self,x_encod):
677     """
678     Reads in encoder values and converts to
679     feet
```

```
680
681     Args:
682         None
683
684     Returns:
685         feet (float): converted encoder counts from distance in feet
686     """
687     meter = self._encoder2meter(x_encod)
688     feet = round(meter * 3.28084,4)
689
690     return feet
691
692 def _feet2cm(self,x_dist):
693     """
694     Reads in distance values in feet and converts to
695     encoder counts
696
697     Args:
698         None
699
700     Returns:
701         encod (int): converted encoder counts from distance in feet
702     """
703     dist = round(float(x_dist * 12 * 2.54),3)
704
705     return dist
706
707 def _cm2feet(self,x_dist):
708     """
709     Reads in distance values in feet and converts to
710     encoder counts
711
712     Args:
713         None
714
715     Returns:
716         encod (int): converted encoder counts from distance in feet
717     """
718     dist = round(float(x_dist / (12 * 2.54)),2)
719
720     return dist
721
722 def _imu_serial(self):
723     """
724     Reads in serial robot's yaw angle value from imu sensor, cleans buffer,
725     and converts it into a float data type. Convers reading from 0-360 to
726     -180 < yaw < 180.
727
728     Args:
729         None
730
731     Returns:
732         self._yaw (float): robots yaw in degrees
733     """
734     self._ser.reset_input_buffer()
735     self._ser.flush()
```

```

736
737     while True:
738
739         try:
740             # Read for imu from serial
741             if(self._ser.in_waiting > 0):
742                 #
743                 # Strip serial stream of extra characters
744                 line = self._ser.readline()
745
746                 line = line.rstrip().lstrip()
747
748                 line = str(line)
749                 line = line.strip('"')
750                 line = line.strip("'")
751
752                 # Return float
753                 line = float(line)
754
755                 if (line > 180 and line <=360):
756                     line = line - 360
757
758                 self._yaw = line
759                 return self._yaw
760
761         except Exception as error:
762             self._logger.error(f"Imu Error: {error}")
763
764     def _imu_serial_straight(self,queue):
765         """
766         Reads in serial robot's yaw angle value from imu sensor, cleans buffer,
767         and converts it into a float data type. Converts reading from 0-360 to
768         -180 < yaw < 180.
769
770         Only difference from self._imu_serial() is that this is run in a separate
771         thread which is controlled by events to start, pause, and stop function call
772
773         Args:
774             None
775
776         Returns:
777             self._yaw (float): robots yaw in degrees
778         """
779
780         while not self._event.is_set():
781
782             self._ser.reset_input_buffer()
783             self._ser.flush()
784
785             try:
786                 while self._pause_event.is_set():
787                     # Read for imu from serial
788                     if(self._ser.in_waiting > 0):
789                         line = self._ser.readline()
790
791                         line = line.rstrip().lstrip()

```

```
792
793         line = str(line)
794         line = line.strip("'")
795         line = line.strip("b'")
796
797         # Return float
798         line = float(line)
799
800         if (line > 180 and line <=360):
801             line = line - 360
802
803         queue.put(line)
804
805     except Exception as error:
806         self._logger.warning(f"Imu Error inside thread: {error}")
807
808 def _distance(self):
809     """
810     Reads in serial robot's distance from an obstacle using ultrasonic sensor l
811     sending a trigger pulse and listening the returning echo to compuse the sep
812     distance from nearby object using triangulation technique. Then it saves c
813     distance in self._depth
814
815     Args:
816         None
817
818     Returns:
819         None
820     """
821     pulse_start = 0
822     pulse_end = 0
823
824     # Define pin allocations
825     trig = 16
826     echo = 18
827     # Setup GPIO board & pins
828     gpio.setup(trig, gpio.OUT)
829     gpio.setup(echo, gpio.IN)
830
831     # Ensure output has no value
832     gpio.output(trig, False)
833     time.sleep(0.01)
834
835     # Generate trigger pulse
836     gpio.output(trig, True)
837     time.sleep(0.00001)
838     gpio.output(trig, False)
839
840     # Generate echo time signal
841     while gpio.input(echo) == 0:
842         pulse_start = time.time()
843
844     while gpio.input(echo) == 1:
845         pulse_end = time.time()
846
847     pulse_duration = pulse_end - pulse_start
```

```

848
849     # Convert time to distance
850     distance = pulse_duration * 17150
851     self._depth = round(distance, 2)
852
853     return self._depth
854
855 def _rotate(self):
856     """
857     Reads in direction of rotation requested, and starts
858     rotating the motors accordingly to achieve desired orientation.
859
860
861     Args:
862         None
863
864     Returns:
865         None
866     """
867
868     if self._direction == 'r':
869         self._pivotright(60)
870     elif self._direction == 'l':
871         self._pivotleft(60)
872
873 def _dist_estimate(self):
874     """
875     Reads in radius of bounding box from bbox detector function, and
876     plugs radius into precomputed correlation function to estimate the
877     depth of toy block from robot.
878
879     Assigns results to self._travel and saves image showing value of distance
880     appended to it in self._img_frame and writes frame to video object self._oi
881
882     Args:
883         None
884
885     Returns:
886         None
887     """
888     # Take image with picamera
889     self._picam_frame()
890     self._img_frame, __, bbox_radius, self._success = cntr_bbox.bbox_detect(se
891
892     if self._success not in [None, False]:
893         # Plug in pixel size into depth equation
894         distance = (0.0099 * bbox_radius**2) - (1.8846*bbox_radius) + 103.47
895         self._travel = round(distance,2)
896         # Append image of estimated distance on image frame
897         self._img_frame = cntr_bbox.dist_img(self._img_frame,self._travel)
898         # save image into video object
899         self._out.write(self._img_frame)
900 def _corner_detect(self,img,origImg):
901
902     # Create a List to store the x,y location of points
903     pts_loc = []

```

```
904
905 # Detect corners from image
906 corners = cv2.goodFeaturesToTrack(img,5,0.01,10)
907
908 if corners is not None:
909
910     corners = np.int0(corners)
911
912     # identify location of corners in image
913     for i in corners:
914         # Extract x,y coordinate of points
915         x,y = i.ravel()
916
917         pts_loc.append([x,y])
918
919         # Draw circle ontop of original image
920         cv2.circle(origImg, (x,y), 3, (0,0,255),-1)
921
922     # Create a column vector from pts List
923     pts_loc = np.array(pts_loc)
924
925     return img, pts_loc, origImg, True
926 else:
927     return img, pts_loc, origImg, None
928
929 def _center_det(self,pt_list, center):
930
931     # Extract x,y points from pt_list
932     x = pt_list[:,0]
933     y = pt_list[:,1]
934
935     # Determine the min and max width & height values
936     # of the points, as if to draw rectangle around arrow
937     x_min = x.min()
938     y_min = y.min()
939
940     x_max = x.max()
941     y_max = y.max()
942
943     # Store height of bounding box
944     vert_dst = y_max - y_min
945
946     # Store width of bounding box
947     horz_dst = x_max - x_min
948
949     # Compute and store half dimensions of
950     # box, will come later when determining
951     # arrow direction
952     y_half = vert_dst/2 + y_min
953     x_half = horz_dst/2 + x_min
954
955     # Store center of the block
956     #center = [int(round(x_half)), int(round(y_half))]
957     center.append(int(round(x_half)))
958     center.append(int(round(y_half)))
959     # Estimate radius
```

```

960     self.bbox_radius = int(round(math.sqrt((x_max - x_half)**2 + (y_max - y_half)**2)))
961
962     return center, self.bbox_radius
963
964     def _bbox_detect(self):
965         """
966         Reads in an image, and then applies preprocessing functions that mask and
967         blur image. Then, attempts to detect corners of an object in frame, and if
968         successful it will run further computation to determine the center and radius
969         of detected object. Finally adds the draws the bounding circle and center,
970         as well as write computed values on original image
971
972         Assigns results to self._travel and saves image showing value of distance
973         appended to it in self._img_frame and writes frame to video object self._out
974
975         Args:
976             image (image): Original colored image taken in from camera
977
978         Returns:
979             bbox_radius (float): Radius of bounding circle detecting object
980             len(pts_loc) (int): The number of corner points identified on object frame
981         """
982         # Initialize list to save corner points and bbox radius
983         bbox_radius = 0
984         pts_loc = []
985         center = []
986
987         # Take image using camera frame
988         self._picam_frame()
989
990         # Prepare image by masking and blurring
991         image_blurred = self._prep_image()
992
993         # Extract bbox radius from image
994         self._img_frame, center, bbox_radius, self._success = cntr_bbox.bbox_detect(image_blurred)
995
996         return self._img_frame, center, bbox_radius, self._success
997
998     # Detect corners from image
999     # img_cnrn, pts_loc, image, var = self._corner_detect(image_blurred, self._img_frame)
1000     #
1001     # if var is not None:
1002     #
1003     #     # Check if corners are detected
1004     #     if len(pts_loc) > 3:
1005     #         ## Draw contours over an image, if available
1006     #         center, bbox_radius = self._center_det(pts_loc, center)
1007     #
1008     #         block_coordinate = "(" + str(center[0]) + "," + str(center[1]) +
1009     #
1010     #         # Draw circle ontop of original image
1011     #         cv2.circle(image, tuple(center), bbox_radius, (0,255,255),2)
1012     #         cv2.circle(image, tuple(center), 0,(0,0,255),5)
1013     #         cv2.putText(image,block_coordinate,(0,int(self.cy)),self.font,2,
1014     #
1015     #         # Draw a cross at center of frame

```



```

1016 #         cv2.line(image,(self.frame_center[0]-100, self.frame_center[1]),
1017 #         cv2.line(image,(self.frame_center[0], self.frame_center[1]-100),
1018 #
1019 #         success = True
1020 #
1021 #         self._logger.info(f"{self._block_now} block is detected")
1022 #         time.sleep(1)
1023 #
1024 #         return image, center, bbox_radius, success
1025 #
1026 #     else:
1027 #         self._logger.info("Not clearly detected")
1028 #         success = False
1029 #
1030 #         return image, center, bbox_radius, success
1031 #
1032 #     else:
1033 #         print("No object of interest in scene")
1034 #         return image, 0, 0, success
1035
1036 self._logger.info(f"Radius {bbox_radius}")
1037 self._logger.info(f"Number of corners/edges detected are {len(pts_loc)}")
1038
1039 #return bbox_radius, center #, len(pts_loc),
1040
1041 def _pivot(self,deg_diff,just_turn=False,steady = False):
1042     """
1043     This function pivots robot to deseired angle
1044
1045     Args:
1046         deg_diff (float): Amount of angular rotation to turn.
1047             Default value is self._turn parameter
1048
1049     Returns:
1050         None
1051
1052     """
1053     self._logger.info("Inside pivot function")
1054     self._task_now = 'rotating'
1055     self._actions[self._task_now] = True
1056
1057     # To know the direction of turn. Notation based on camera object detection
1058     # notation
1059     # thresh = 0.8
1060     if deg_diff >= 0:
1061         self._direction = 'r'
1062     else:
1063         self._direction = 'l'
1064     # thresh = -0.8
1065
1066     # Initialize change in time for derivative control
1067     zones = [0.4, 0.8, 1]
1068
1069     self._yaw = self._imu_serial()
1070     yaw_start = self._yaw
1071

```

```

1072     # Angle diff initialized
1073     yaw_diff0 = deg_diff + 0.001
1074
1075     yaw_final = yaw_start + deg_diff
1076     yaw_diff = yaw_final - yaw_start
1077     yaw_diff_old = yaw_diff
1078
1079     self._logger.info(f"Needed final yaw is {yaw_final} degrees")
1080     yaw_current = yaw_start
1081     # Lower and upper bounds of dutycycle initialized
1082     lower_dc = 70
1083     upper_dc = 90
1084     # Divide the distance traveled into three zones
1085     trip = [x * yaw_diff for x in zones]
1086     # Variables for PID control
1087     derivative = 0
1088     integral = 0
1089
1090     start = time.time()
1091     yaw_diff_list = [0,0]
1092     yaw_diff_list.append(yaw_diff)
1093     window_size = 5
1094     while True:
1095         try:
1096
1097             #prcnt = abs(self._imu_serial() - yaw_start)/deg_diff
1098             prcnt = 1 - yaw_diff/yaw_diff0
1099
1100             # Adopting moving average
1101             # cumsum = np.cumsum(yaw_diff_list)
1102             # cumsum>window_size:] = cumsum>window_size:] - cumsum[:window_size]
1103             # moving_avg = cumsum>window_size - 1:] / window_size
1104
1105             if prcnt >= zones[0] and prcnt <= zones[1]:
1106                 self._dutycycle[self._direction] = self.duty[self._direction]
1107             elif prcnt > zones[1]:
1108                 self._dutycycle[self._direction] = self.duty[self._direction]
1109
1110             # Save robots pose by saving its x, yaw position
1111             self._path.append((round(self.global_x,2),round(self.global_y,2),r
1112             self._dutycycle[self._direction] = np.clip(self._dutycycle[self._d
1113
1114             if abs(yaw_diff) <= 2:
1115                 self._dutycycle[self._direction] = 65
1116
1117             elif abs(yaw_diff) >= 100:
1118                 self._dutycycle[self._direction] = 100
1119
1120
1121
1122             if abs(yaw_diff) >= 0.8:
1123
1124                 if ((np.sign(yaw_diff) == np.sign(yaw_diff_old)) or (abs(yaw_d
1125                 # if (abs(yaw_diff - np.mean(yaw_diff_list[-3:-1])) < 5):
1126                 # if abs(yaw_diff) >= 0.5:
1127                     self._logger.info(f"Current yaw {self._imu_serial()}; Angl

```

```

1128
1129         if not steady:
1130             if self._direction == 'r':
1131                 self._pivotright(self._dutycycle[self._direction])
1132             elif self._direction == 'l':
1133                 self._pivotleft(self._dutycycle[self._direction])
1134         else:
1135             if self._direction == 'r':
1136                 self._pivotright(70)
1137             elif self._direction == 'l':
1138                 self._pivotleft(70)
1139
1140     else:
1141         self._pwmZero()
1142         total_turn = yaw_start - self._imu_serial()#self.queue.get()
1143         self._logger.info(f"Current yaw {self._imu_serial()}, Total turn {total_turn}")
1144         self._logger.info("Robot turn done")
1145
1146         yaw_diff = round(yaw_diff,2)
1147
1148         if not just_turn:
1149             if steady:
1150                 #if (abs(yaw_diff) >= 0.):
1151                 # Check for overshooting
1152
1153                 deg_diff = self._scan4object()
1154
1155                 if deg_diff is not None:
1156
1157                     if abs(deg_diff) > 0.5:
1158
1159                         #
1160                         time.sleep(1)
1161                         # Return pwm dutcycle back to base
1162                         self._dutycycle[self._direction] = 70
1163
1164                         self._pivot(deg_diff,steady=True)
1165
1166                     if abs(yaw_diff) >= 0.6:
1167                         self._dutycycle[self._direction] = 70
1168
1169                     self._pivot(yaw_diff,steady=True)
1170
1171                 elif abs(yaw_diff) > 0.6:
1172
1173                     # Return pwm dutcycle back to base
1174                     self._dutycycle[self._direction] = 70
1175
1176                     self._pivot(yaw_diff)
1177
1178                 # Return pwm dutcycle back to base
1179                 self._dutycycle[self._direction] = self.duty[self._direction]
1180
1181                 # Save robots pose by saving its x, yaw position
1182                 self._path.append((round(self.global_x,2),round(self.global_y,2)))
1183

```

```

1184         self._logger.info("Final orientation reached")
1185         break
1186
1187     yaw_diff_old = yaw_diff
1188     self._yaw = self._imu_serial()
1189     yaw_diff = yaw_final - self._yaw
1190     yaw_diff_list.append(yaw_diff)
1191
1192     #         if ((np.sign(yaw_diff) != np.sign(yaw_diff_old)) and (abs(yaw_diff) > 5)):
1193     #             yaw_diff_list.pop()
1194     #             yaw_diff = yaw_diff_old
1195     # Filtering data via median thresholding
1196     #         if len(yaw_diff_list) > 6:
1197     #             median = np.median(yaw_diff_list[-5:-1])
1198     #         else:
1199     #             median = np.median(yaw_diff_list[-3:-1])
1200
1201     #         if (abs(yaw_diff - median) < 5):
1202     #             yaw_diff_list.append(yaw_diff)
1203     #         else:
1204     #             yaw_diff_list.pop()
1205     #             yaw_diff = yaw_diff_old
1206     #             yaw_diff_list.append(yaw_diff_old)
1207
1208     except KeyboardInterrupt:
1209         self._logger.error("Pivot Interrupted")
1210         self._gameover()
1211     except Exception as error:
1212         self._logger.error(f"Error {error}")
1213
1214     self._pwmZero()
1215     self._actions[self._task_now] = False
1216     self._logger.info("Pivot Finished")
1217
1218     def _go2block(self, direction='f', eucl_disp=0, just_drive=False, steady=False)
1219         """
1220         This function drives the robot in a straight path, either forward
1221         or reverse
1222
1223         Args:
1224             direction: Direction of drive, forward or reverse
1225
1226         Returns:
1227             None
1228
1229         """
1230         self._task_now = 'driving'
1231         self._actions[self._task_now] = True
1232         self._logger.info("Going toward block (go2block)")
1233
1234         # Divide the separation distance into three zones
1235         # First zone is 70% into drive
1236         # Second zone is 20%
1237         # Third zone is the last 10%
1238         # Each of the zones have their own dutycycle. Robot will
1239         # drive and stop at each zone, scan, align, and resume drive

```

```

1240     zones = [0.7, 0.85, 0.9]
1241     thresh = 20
1242     self._success = False
1243     # Check if straight drive mode is requested and steady drive mode
1244     if not just_drive and not steady:
1245         # Use image from camera to estimate distance
1246         self._dist_estimate()
1247         # convert depth to encoder count
1248         orig_dist = self._cm2encoder(self._travel-35)
1249
1250         self._logger.info(f"Distance estimated {self._travel}")
1251         self._logger.info(f"Distance to drive {self._travel - 35}")
1252     elif steady:
1253         # Use image from camera to estimate distance
1254         self._dist_estimate()
1255         # convert depth to encoder count
1256         orig_dist = self._cm2encoder(self._travel-5)
1257         self._logger.info(f"Distance estimated {self._travel}")
1258         self._logger.info(f"Driving slowly {self._travel-5} cms")
1259     elif just_drive:
1260         # Drive straight command
1261         orig_dist = self._feet2encoder(eucl_disp)
1262         self._logger.info(f"Just Driving Distance estimated {orig_dist}")
1263     elif just_drive and steady:
1264         # Drive straight command
1265         orig_dist = self._cm2encoder(eucl_disp)
1266         self._logger.info(f"Just Driving Distance estimated {orig_dist}")
1267
1268     self._logger.info(f"Distance to drive {orig_dist} encoder")
1269     pos_encoder_orig = 0
1270     pos_encoder = 0
1271     orig_error = orig_dist
1272
1273     self.counterFL = np.uint64(0)
1274     self.counterBR = np.uint64(0)
1275     self.buttonBR = int(0)
1276     self.buttonFL = int(0)
1277
1278     # Defining the limit of dutycycle
1279     lower_dc = 70
1280     upper_dc = 100
1281
1282     z1 = 80
1283     z2 = z1 - 20
1284     z3 = z2 - 10
1285
1286     # Divide up the path in different zones
1287     trip = [x * orig_dist for x in zones]
1288
1289     # Determine if function completed successfully
1290     self._success = False
1291
1292     #time.sleep(1)
1293     yaw0 = self._imu_serial()
1294
1295     # For encoder Level control

```

```

1296     integral_error = 0
1297     derivative_error = 0
1298     error_encoder0 = 0
1299     error_encoder = 0
1300
1301     # For achieving overall distance
1302     integral = 0
1303     derivative = 0
1304     error0 = 0
1305     error = 0
1306     startD = 0
1307
1308     cnt = 0
1309     # Assign direciton of drive from argument
1310     self._direction = direction
1311     # Compute the difference in distance between start state and goal state
1312     self._logger.info("Sending command toward thrusters")
1313     self._logger.info(f"Checking encoder to cm {self._encoder2cm(orig_dist)}")
1314
1315     self._picam_frame()
1316     self._img_frame = cntr_bbox.dist_img(self._img_frame, self._encoder2cm(orig_dist))
1317     #cntr_bbox.img_show("Detecting Block",self._img_frame)
1318
1319     start = time.time()
1320
1321     #To avoid division by zero
1322     if abs(orig_dist) < 0.001:
1323         orig_dist = 0.001
1324
1325     x = 0
1326     y = 0
1327
1328     # Save robots pose by saving its x, yaw position
1329     self._yaw = self._imu_serial()
1330     # Convert imu notation to general +ccw from positive x
1331     theta = -self._yaw
1332
1333     while not self._success:
1334
1335         # by considering the minimum of the two encoders
1336         pos_encoder = int(min(self.counterFL, self.counterBR))
1337
1338         # Global error between start and goal state will be
1339         error = orig_dist - pos_encoder
1340         prcnt = round(1 - error/orig_dist, 3)
1341
1342         if (pos_encoder % 150) == 0:
1343
1344             # Save robots pose by saving its x, yaw position
1345             x = self._encoder2feet(pos_encoder) * math.cos(math.radians(theta))
1346             y = self._encoder2feet(pos_encoder) * math.sin(math.radians(theta))
1347
1348             self._path.append((round(self.global_x + x, 2), round(self.global_y + y, 2)))
1349
1350             #
1351             if (self._distance() < 30):
1352                 if ((10 - self.global_x < 1 or self.global_x < 0.9) or ((10 - self

```

```
1352         self._logger.info("Too close to border. Stop and Exit straight")
1353         self._pwmZero()
1354         break
1355
1356         #Record encoder states to txt file
1357         outstring = str(round(self.global_x + x,2)) + ' ' + str(round(self
1358         self.f.write(outstring)
1359
1360
1361     if prcnt < zones[0] and self._direction == 'f' and not steady:
1362         if cnt == 0:
1363             lower_dc = 50
1364             upper_dc = 60
1365
1366             error = orig_dist - pos_encoder
1367             pos_encoder_orig = pos_encoder
1368
1369             self.duty[self._direction] = [55,55]
1370             self._dutycycle[self._direction] = self.duty[self._direction]
1371             cnt += 1
1372
1373             self._picam_frame()
1374             self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
1375             self._out.write(self._img_frame)
1376
1377     elif (prcnt >= zones[0] and prcnt < zones[1]) and self._direction == 'f':
1378         if cnt == 1:
1379             lower_dc = 35
1380             upper_dc = 50
1381
1382             pos_encoder_orig += pos_encoder
1383
1384             self.duty[self._direction] = [40,40]
1385             self._dutycycle[self._direction] = self.duty[self._direction]
1386             cnt += 1
1387
1388             self._picam_frame()
1389             self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
1390             self._out.write(self._img_frame)
1391
1392     elif (prcnt >= zones[1] and prcnt < zones[2]) and self._direction == 'f':
1393         if cnt == 1:
1394             lower_dc = 28
1395             upper_dc = 32
1396             self._pwmZero()
1397             pos_encoder_orig += pos_encoder
1398
1399             self.duty[self._direction] = [30,30]
1400             self._dutycycle[self._direction] = self.duty[self._direction]
1401             cnt += 1
1402
1403             self._picam_frame()
1404             self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
1405             self._out.write(self._img_frame)
1406     elif prcnt >= zones[2]:
1407         if cnt == 2:
```

```

1408         lower_dc = 25
1409         upper_dc = 28
1410
1411         pos_encoder_orig += pos_encoder
1412
1413         self.duty[self._direction] = [22,22]
1414         self._dutycycle[self._direction] = self.duty[self._direction]
1415         cnt += 1
1416
1417         self._picam_frame()
1418         self._img_frame = cntr_bbox.dist_img(self._img_frame, self._en
1419         self._out.write(self._img_frame)
1420
1421
1422         self._dutycycle[self._direction] = [np.clip(self._dutycycle[self._dire
1423             np.clip(self._dutycycle[self._dire
1424
1425         if steady:
1426             self._dutycycle[self._direction] = [30,30]
1427
1428         if self._direction == 'f':
1429             # Drive robot towards direction
1430             self._forward(tuple(self._dutycycle[self._direction]))
1431         elif self._direction == 'rev':
1432             # Drive robot towards direction
1433             self._reverse(tuple(self._dutycycle[self._direction]))
1434
1435         # Count encoders for left and right motors
1436         if int(gpio.input(12)) != int(self.buttonBR):
1437             self.buttonBR = int(gpio.input(12))
1438             self.counterBR += 1
1439
1440         if int(gpio.input(7)) != int(self.buttonFL):
1441             self.buttonFL = int(gpio.input(7))
1442             self.counterFL += 1
1443
1444         # PD Controller for encoder balance
1445         error_encoder = self.counterFL - self.counterBR
1446
1447         if error_encoder > 25:
1448             # Give power to corresponding motors
1449             self._dutycycle[self._direction][0] -= 0.1 * self.duty[self._dire
1450             self._dutycycle[self._direction][1] += 0.1 * self.duty[self._dire
1451         elif error_encoder < -25:
1452             # Give power to corresponding motors
1453             self._dutycycle[self._direction][0] += 0.1 * self.duty[self._dire
1454             self._dutycycle[self._direction][1] -= 0.1 * self.duty[self._dire
1455         else:
1456             # Give power to corresponding motors
1457             self._dutycycle[self._direction][0] = self.duty[self._direction][0]
1458             self._dutycycle[self._direction][1] = self.duty[self._direction][1]
1459
1460
1461         if (error <= thresh) and (direction in ['f','rev']):
1462             self._yaw = self._imu_serial()
1463             angle_diff = yaw0 - self._yaw

```



```

1464     self._logger.info(f"counterBR: {self.counterBR} counterFL: {self.c
1465     self._logger.info(f"Angle shifted: {angle_diff} deg ")
1466     self._logger.info(f"Estimated distance traveled: {self._encoder2cm
1467     self.counterFL = np.uint64(0)
1468     self.counterBR = np.uint64(0)
1469     self.buttonBR = int(0)
1470     self.buttonFL = int(0)
1471     self._logger.info("Arrived in front of block")
1472     self._pwmZero()
1473     self._success = True
1474     self._actions[self._task_now] = False
1475
1476     # Convert imu notation to general +ccw from positive x
1477     theta = -self._yaw
1478
1479     # Save robots pose by saving its x, yaw position
1480     x = self._encoder2feet(pos_encoder) * math.cos(math.radians(theta))
1481     y = self._encoder2feet(pos_encoder) * math.sin(math.radians(theta))
1482     self.global_x += x
1483     self.global_y += y
1484     #self.global_yaw += self._yaw
1485
1486     self._path.append((round(self.global_x,2),round(self.global_y,2),r
1487
1488     self._logger.info(f"x value {round(self.global_x,2)} and y value {
1489
1490     # Take image and save to video object
1491     self._picam_frame()
1492     self._img_frame = cntr_bbox.dist_img(self._img_frame, self._encod
1493     #cntr_bbox.img_show("Detecting Block",self._img_frame)
1494     self._out.write(self._img_frame)
1495
1496     #Record encoder states to txt file
1497     outstring = '#' * 30 + '\n'
1498     self.f.write(outstring)
1499     outstring = 'End of Driving {self._direction}' + '\n'
1500     self.f.write(outstring)
1501     outstring = '#' * 30 + '\n'
1502     self.f.write(outstring)
1503     break
1504
1505     # Return pwm dutcycle back to base
1506     self._dutycycle[self._direction] = self.duty[self._direction]
1507
1508     def _scan4object(self):
1509         """
1510         This function scans for block in environment by turning in place
1511         until it finds the block. If it doesn't find the block after full 360
1512         degree turn, it returns false to say not found
1513
1514         Args:
1515             None
1516
1517         Returns:
1518             None
1519         """

```

```

1520
1521 self._task_now = 'scanning'
1522 self._actions[self._task_now] = True
1523 self._logger.info(f"Scanning for {self._block_now} block started")
1524
1525 scan_turn = 0
1526
1527 while True:
1528     # Detect object in frame
1529
1530     self._picam_frame()
1531     #self._img_frame, ave_center, bbox_radius, success = self._bbox_detect
1532     self._img_frame, ave_center, bbox_radius, success = cntr_bbox.bbox_det
1533
1534     if success not in [None, False]:
1535         # Object found
1536         diff = ave_center[0] - self.frame_center[0]
1537         # Convert the pixel difference to degrees
1538         self._turn = diff * cntr_bbox.pixel2deg
1539
1540         if (self._turn > 180 and self._turn <= 360):
1541             self._turn = self._turn - 360
1542
1543         self._yaw = -self._turn
1544         self._path.append((round(self.global_x, 2), round(self.global_y, 2), r
1545
1546         self._success = True
1547         self._status['found'] = True
1548         self._task_now = 'scanning'
1549         self._actions[self._task_now] = False
1550         return self._turn
1551
1552     else:
1553         # Pivot the robot to a predefined turn angle
1554         self._pivot(self.turn)
1555         # Read the distance from obstacle for reference
1556         self._distance()
1557         # Keep track of turn
1558         #self._yaw += self.turn
1559         scan_turn += self.turn
1560         self._yaw += -self._turn
1561         self._path.append((round(self.global_x, 2), round(self.global_y, 2), r
1562
1563     #
1564     #
1565     #
1566     #
1567     #
1568     #
1569     #
1570     #
1571     #
1572     #
1573     #
1574     #
1575     #

```

```

1576 #         self._logger.warning("No objects in nearby vicinity")
1577 #         return None
1578
1579         if abs(scan_turn) >= 360:
1580             self._status['found'] = False
1581             self._success = False
1582             self._task_now = 'scanning'
1583             self._actions[self._task_now] = False
1584             self._logger.warning("No objects in nearby vicinity")
1585             return None
1586
1587 def _pick_up(self):
1588     """
1589     This function drives the robot to steadily approach and picks up toy block
1590
1591     Args:
1592         None
1593
1594     Returns:
1595         None
1596     """
1597     self._success = False
1598     self._task_now = 'picking'
1599     self._actions[self._task_now] = True
1600
1601     # Take image and save to video object
1602     self._picam_frame()
1603     #cntr_bbox.img_show("Detecting Block",self._img_frame)
1604     self._out.write(self._img_frame)
1605
1606     while not self._success:
1607         # Open gripper
1608         self._servo_cntrl(self.grip_state['opened'])
1609         self._gripper_state = self.grip_state['opened']
1610         self._logger.info("Servo Opened")
1611         # Take image and save to video object
1612         self._picam_frame()
1613         #cntr_bbox.img_show("Detecting Block",self._img_frame)
1614         self._out.write(self._img_frame)
1615         # Align robot fine turn rotation
1616         degree = self._scan4object()
1617
1618         #         while True:
1619         #             if degree is not None:
1620         #                 if abs(degree) > 0.6:
1621         #                     self._pivot(degree, steady=True)
1622         #                 else:
1623         #                     break
1624         #             else:
1625         #                 break
1626         # Scan block and drive slowly towards it
1627         self._logger.info("Driving slowly toward block")
1628         self._go2block(steady=True)
1629         self._logger.info("Slow drive completed")
1630
1631         # Take image and save to video object

```

```

1632         self._picam_frame()
1633         #cntr_bbox.img_show("Detecting Block",self._img_frame)
1634         self._out.write(self._img_frame)
1635
1636         # Close gripper
1637         self._servo_cntrl(self.grip_state['closed'])
1638         self._gripper_state = self.grip_state['closed']
1639         self._logger.info("Servo Closed")
1640
1641         # Take image and save to video object
1642         self._picam_frame()
1643         #cntr_bbox.img_show("Detecting Block",self._img_frame)
1644         self._out.write(self._img_frame)
1645
1646         #if (abs(radius2 - radius1) < 5) or (abs(center2[0] - center1[0]) < 3)
1647         self._logger.info(f"The {self._block_now} block has been gripped succe
1648         self._success = True
1649         self._status['picked_up'] = True
1650         self._actions[self._task_now] = False
1651
1652         # Read saved image and send via email
1653         self._email_media()
1654
1655     def _go2(self,destination=(0,0),area=''):
1656         """
1657         This function computes the difference in pose between the robot and the des
1658         then estimates the displacement vector along with the angle of rotation nee
1659         to align the robot head on. It then calls the pivot and go2block function
1660         and command the robot to enter the construction zone.
1661
1662         Args:
1663             destination (tuple): contans x,y location of destination
1664             area (str): Describes where that destination is
1665
1666         Returns:
1667             None
1668         """
1669         self._logger.info(f"Initiating travel to {area}")
1670         self._success = False
1671         self._task_now = 'returning'
1672         self._actions[self._task_now] = False
1673
1674         # Read last position from the appended path
1675         # self.global_x = self._path[-1][0]
1676         # self.global_y = self._path[-1][1]
1677         #self.global_yaw = self._path[-1][2]
1678         delta_x = destination[0]-self.global_x
1679         delta_y = destination[1]-self.global_y
1680         self._yaw = self._imu_serial()
1681
1682         self._logger.info(f"Pose Now: X- {self.global_x} and Y- {self.global_y} an
1683         # Compute the euclidean distance between current robot location and constr
1684         disp = math.sqrt((delta_x)**2 + (delta_y)**2)
1685         # Calculate the yaw angle difference between these two points using slope
1686         rad_diff = math.atan2(delta_y,delta_x)
1687         # Convert from radians to angle

```

```

1688 target_angle = math.degrees(rad_diff)
1689 # Ensure angle is between [0, 360]
1690 target_angle %= 360
1691
1692 self._logger.info(f"Computed target angle is {target_angle}")
1693
1694 # # Place angle from [0,360] to [-180,0) U [0,180]
1695 if target_angle >=-180 and target_angle <=180:
1696     target_angle = -target_angle
1697 elif (target_angle >= 180 and target_angle <360):
1698     target_angle = 360 - target_angle
1699 elif (target_angle > -360 and target_angle <=-180):
1700     target_angle = -(target_angle + 360)
1701
1702 self._logger.info(f"Normalized target angle is {target_angle}")
1703 # # Normalize angle difference to range [-180, 180) degrees
1704 # target_angle = (target_angle + 180) % 360 - 180
1705 # # Angle difference from current and target
1706 # deg_diff = self._yaw - target_angle
1707 # deg_diff %= 360
1708
1709 # Based on the quadrant of the point heading towards normalize
1710 # the target angle
1711 # if delta_x < 0:
1712 #     if delta_y > 0:
1713 #         #target_angle = 180 - target_angle
1714 #         pass
1715 #     else:
1716 # # target_angle = target_angle - 180
1717 #         pass
1718 # else:
1719 #     if delta_y < 0:
1720 # # target_angle = -target_angle
1721 #         pass
1722
1723 # convert target angle with respect to imu configuration
1724 # target_angle = -target_angle
1725
1726 # Angle difference from current and target
1727 deg_diff = target_angle - self._yaw
1728 #deg_diff %= 360
1729
1730 # if (deg_diff > 180 and deg_diff <360):
1731 #     deg_diff = (deg_diff - 360) # Turning to the right
1732 # elif (deg_diff > -360 and deg_diff <=-180):
1733 #     deg_diff = (deg_diff + 360) # Turning to the Left
1734
1735 # # Place angle from [0,360] to [-180,0) U [0,180]
1736 # if (deg_diff > 180 and deg_diff <=360):
1737 #     deg_diff = deg_diff - 360
1738 # # Normalize angle difference to range [-180, 180) degrees
1739 # deg_diff = (deg_diff + 180) % 360 - 180
1740
1741 self._logger.info(f"Yaw that vehicle will pivot to {deg_diff}")
1742 # Pivot robot to face construction zone
1743 self._pivot(deg_diff, just_turn=True)

```

```
1744 self._logger.info(f"Started driving straight to {area}")
1745 # Drive robot foward to construction zone
1746 self._go2block(eucl_disp = disp, just_drive = True)
1747 self._logger.info(f"Arrived at {area}")
1748
1749 # New global yaw
1750 self.global_yaw = self._imu_serial()
1751
1752 self._success = True
1753 self._status['arrived'] = True
1754 self._actions[self._task_now] = False
1755
1756 def _place(self):
1757     """
1758     This function places toy block
1759
1760     Args:
1761         None
1762
1763     Returns:
1764         None
1765     """
1766     self._success = False
1767     self._task_now = 'placing'
1768     self._actions[self._task_now] = True
1769
1770     # Open gripper
1771     self._servo_cntrl(self.grip_state['opened'])
1772     self._gripper_state = self.grip_state['opened']
1773     self._logger.info(f"Gripper Opened to place {self._block_now} block")
1774
1775     # Drive in reverse
1776     self._go2block(direction='rev', eucl_disp=7.5, just_drive=True, steady=True)
1777     time.sleep(1)
1778     self._logger.info("Backed away from toy block")
1779
1780     # Close gripper
1781     self._servo_cntrl(self.grip_state['closed'])
1782     self._gripper_state = self.grip_state['closed']
1783     self._logger.info("Gripper Closed to prepare for next order")
1784
1785     self._success = True
1786     self._status['placed'] = True
1787     self._actions[self._task_now] = False
1788
1789     self._logger.info(f"{self._block_now} block is now placed inside construct:
1790
1791     # Read saved image and send via email
1792     self._email_media()
1793
1794 def _localize(self):
1795     """
1796     This function uses distance sensor to read from borderwall
1797     so that the robot's exact position in arena can easily be
1798     identified
1799
```

```
1800     Args:
1801         None
1802
1803     Returns:
1804         None
1805     """
1806     self._success = False
1807     self._task_now = 'localizing'
1808     self._actions[self._task_now] = True
1809
1810     # pivot the robot to positive y first and read from distance sensor
1811     yaw_final = -90
1812     yaw_current = self._imu_serial()
1813     deg_diff = yaw_final - yaw_current
1814
1815     #deg_diff %= 360
1816     # Normalize angle difference to range [-180, 180) degrees
1817     if (deg_diff > 180 and deg_diff < 360):
1818         deg_diff = 360 - deg_diff
1819     elif (deg_diff > -360 and deg_diff <= -180):
1820         deg_diff = -(deg_diff + 360)
1821
1822     # Turn towards borderwall
1823     self._pivot(deg_diff, just_turn=True)
1824
1825     # Now read distance sensor value
1826     self._distance()
1827     delta_ft = self._cm2feet(self._depth)
1828     self._yaw = self._imu_serial()
1829     theta = -self._yaw
1830     x1 = delta_ft * math.cos(math.radians(theta))
1831     y1 = delta_ft * math.sin(math.radians(theta))
1832
1833     # Extract y pose by normalizing from border wall
1834     if self.global_y > 5:
1835         self.global_y = 10 - y1
1836     else:
1837         self.global_y = y1
1838
1839     # pivot the robot to negative x first and read from distance sensor
1840     yaw_final = -179
1841     yaw_current = self._imu_serial()
1842     deg_diff = yaw_final - yaw_current
1843     #deg_diff %= 360
1844
1845     # Normalize angle difference to range [-180, 180) degrees
1846     if (deg_diff > 180 and deg_diff < 360):
1847         deg_diff = 360 - deg_diff
1848     elif (deg_diff > -360 and deg_diff <= -180):
1849         deg_diff = -(deg_diff + 360)
1850
1851     # Turn towards borderwall
1852     self._pivot(deg_diff, just_turn=True)
1853
1854     # Now read distance sensor value
1855     self._distance()
```



```

1856 delta_ft = self._cm2feet(self._depth)
1857 self._yaw = self._imu_serial()
1858 theta = -self._yaw
1859 x2 = delta_ft * math.cos(math.radians(theta))
1860 y2 = delta_ft * math.sin(math.radians(theta))
1861
1862 # Extract x pose by normalizing from border wall
1863 if self._x > 5:
1864     self.global_x = 10 - x2
1865 else:
1866     self.global_x = x2
1867
1868 # Add detected pose of robot into path list
1869 self._path.append((round(self.global_x,2),round(self.global_y,2),round(self.global_yaw,2)))
1870
1871 self._success = True
1872 self._actions[self._task_now] = False
1873 self._logger.info(f"Localization complete")
1874
1875 def _go2site(self):
1876     """
1877     This function commands the robot to drive back to the center
1878     of the arena or near the cluttered enviornment to resume delivering
1879     order
1880
1881     Args:
1882         None
1883
1884     Returns:
1885         None
1886     """
1887
1888     self._logger.info("Initiating travel to cluttered environment")
1889     # Compute the euclidean distance between current robot location and construction zone
1890     disp = math.sqrt((self._x - self.clutter_env[0])**2 + (self._y - self.clutter_env[1])**2)
1891     # Calculate the yaw angle difference between these two points using slope triangle
1892     rad_diff = math.atan2(self.clutter_env[1] - self._y, self.clutter_env[0] - self._x)
1893     # Convert from radians to angle
1894     deg_diff = math.degrees(rad_diff)
1895     # Ensure angle is between [0, 360]
1896     deg_diff %= 360
1897     # Place angle from [0,360] to [-180,0) U [0,180]
1898     if (deg_diff > 180 and deg_diff <=360):
1899         deg_diff = deg_diff - 360
1900
1901     self._logger.info("Orienting to cluttered environment")
1902     # Pivot robot to face construction zone
1903     self._pivot(deg_diff)
1904     self._logger.info("Started driving straight to cluttered environment")
1905     # Drive robot foward to construction zone
1906     self._go2block(eucl_disp = disp,just_drive = True)
1907     self._logger.info("Arrived at cluttered environment")
1908
1909 def _go_home(self):
1910     """
1911     This function commands the robot to drive back to its home location

```



```

1912     marking the end of the challenge.
1913
1914     Args:
1915         None
1916
1917     Returns:
1918         None
1919     """
1920     self._logger.info("Initiating travel to home base")
1921     # Compute the euclidean distance between current robot location and construction zone
1922     disp = math.sqrt((self._x - self.start_pt[0])**2 + (self._y - self.start_pt[1])**2)
1923     # Calculate the yaw angle difference between these two points using slope and angle
1924     rad_diff = math.atan2(self.start_pt[0] - self._x, self.start_pt[1] - self._y)
1925     # Convert from radians to angle
1926     deg_diff = math.degrees(rad_diff)
1927     # Ensure angle is between [0, 360]
1928     deg_diff %= 360
1929     # Place angle from [0,360] to [-180,0) U [0,180]
1930     if (deg_diff > 180 and deg_diff <=360):
1931         deg_diff = deg_diff - 360
1932
1933     self._logger.info("Orienting to home base")
1934     # Pivot robot to face construction zone
1935     self._pivot(deg_diff)
1936     self._logger.info("Started driving straight to home base")
1937     # Drive robot forward to construction zone
1938     self._go2block(eucl_disp = disp, just_drive = True)
1939     self._logger.info("Arrived at home")
1940     self._logger.info("Thank you for playing game.")
1941     self._logger.info("Powering off.")
1942     self._end_game = True
1943
1944     def plot_path(self):
1945         """
1946         Plot the path of the robot during the grand challenge.
1947
1948         Args:
1949             None
1950         Return:
1951             None
1952         """
1953         # Take in the path list as an array 2D NumPy array
1954         path = np.array(self._path)
1955         x = path[:,0]
1956         y = path[:,1]
1957
1958         # Create figure and plot
1959         fig, ax = plt.subplots(1,1)
1960
1961         fig.suptitle('Motor Encoder Analysis')
1962
1963         ax.plot(x,y,ls='solid', color='blue',linewidth=2, label=f'{self._name} Path')
1964
1965         # 3) Label the axes, title, legend
1966         ax.set_ylabel('Y')
1967         ax.set_xlabel('X')

```

```

1968 ax.set_xlim(0,10)
1969 ax.set_ylim(0,10)
1970 plt.savefig('grandChallenge-robotPath.png')
1971 plt.show()
1972 plt.close()
1973
1974 def start(self, order: List):
1975     """
1976     The sequence of actions occuring while baron robot works to pick and place
1977     all toy blocks to the construction zone. This function lists the steps need
1978     for the robot to complete the orders passed to it at the beginning of comp
1979
1980     Args:
1981         order: List[Blocks] list of all toy blocks needed to complete challenge
1982
1983     Returns:
1984         None
1985     """
1986
1987     try:
1988         # Assign order from object initialization
1989         self._order = order
1990
1991         # First wait until startup QR Code has been read
1992         while True:
1993             if self._wait2start():
1994                 break
1995
1996         while len(self._order) > 0:
1997             """While Robot is still active, resume challenge"""
1998             if (not self._end_game):
1999
2000                 self._block_now = self._order.pop(0)
2001                 self._order_no +=1
2002
2003                 # Begin challenge with choesn block
2004                 self._logger.info("-" * 20)
2005                 self._logger.info(f"*** Picking Up {self._block_now} block")
2006                 self._logger.info("-" * 20)
2007
2008                 # Track the number of times robot drives to search area
2009                 search = 0
2010
2011                 # Drive robot to center of arena, get view of toy blocks and b
2012                 # extracting blocks
2013                 self._go2(destination=self.clutter_env,area='cluttered environ
2014
2015                 while True:
2016
2017                     # Begin by scanning for toy block object
2018                     degree = self._scan4object()
2019
2020                     # Take image and save to video object
2021                     #cntr_bbox.img_show("Detecting Block",self._img_frame)
2022                     self._out.write(self._img_frame)
2023

```

```

2024 # Drive toward block if successful, else pivot to
2025 # a free space and drive
2026 if self._success:
2027     #self._logger(f"Pivoting Robot towards {self._block_now}")
2028     self._pivot(degree,steady=True)
2029
2030     # Take image and save to video object
2031     self._picam_frame()
2032     __, __, __, __ = cntr_bbox.bbox_detect(self._img_frame, :
2033     #cntr_bbox.img_show("Detecting Block",self._img_frame)
2034     self._out.write(self._img_frame)
2035
2036     self._logger.info(f"Drive Robot forward to {self._block_now}")
2037     self._go2block()
2038
2039     # # Update global x and y
2040     # self.global_x += self._x
2041     # self.global_y += self._y
2042
2043     ##self._logger(f"{self._name} is now at x- {self.global_x} y- {self.global_y}")
2044
2045     # Take image and save to video object
2046     self._picam_frame()
2047     #cntr_bbox.img_show("Arrived at Block",self._img_frame,
2048     self._out.write(self._img_frame)
2049
2050     ##self._logger(f"Pick up {self._block_now} block with {self._pick_up}")
2051     self._pick_up()
2052
2053     # Drive robot to construction zone to place block
2054     ##self._logger(f"Take {self._block_now} block to construction zone")
2055     self._go2(destination=self.const_zone,area='construction')
2056
2057     #self._logger(f"Place {self._block_now} block inside construction zone")
2058     self._place()
2059
2060     #self._logger("Localizing to get accurate location of block")
2061     self._localize()
2062
2063     ##self._logger(f"Pick up and place the next order in the list")
2064     self._success = True
2065     break
2066 else:
2067     #self._logger("No toy blocks found in current pose. Check the area")
2068     #self._logger("try again.")
2069
2070     # Drive robot to the predetermined survey locations
2071     if search > 1:
2072         self._go2(destination=self._survey_loc[search],area=self._survey_area[search])
2073     else:
2074         #self._logger(f"Robot can't find {self._block_now} block")
2075         #self._logger("In pursuit of time, moving on to next block")
2076         break
2077     search += 1
2078
2079 else:

```

```

2080         #self._logger(f"Mission complete. Take robot back to home base")
2081         self._go2(destination=self.start_pt, area='home base')
2082         return
2083
2084     except KeyboardInterrupt as error:
2085         self._logger.error("Error occurred: %s", error)
2086         self._gameover()
2087         time.sleep(1)
2088         return None
2089     except ValueError as error:
2090         self._logger.error("Error %s", error, exc_info=True)
2091         self._gameover()
2092         time.sleep(1)
2093         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 position
- **_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 debugging 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):** contains x,y locations for the robot to navigate and scan so that to extract toy blocks for mission

Robot(name: str)[▼ View Source](#)

```

145     def __init__(self, name: str):
146         """
147         Initialize Robot attributes
148
149         Args:
150             name: str - takes name of robot
151
152         Returns:
153             None
154         """
155
156         self.name = name
157         self._x = self.start_pt[0]
158         self._y = self.start_pt[1]
159         self._yaw = 0.0
160         self._travel = 0
161         self._turn = 0
162         self._depth = 0
163         self._path = [(1,1,0)]
164         self._pwms = []
165         self._duty_cycle = self.duty
166         self._success = False
167         self._gripper_state = self.grip_state['closed']
168         self._order = []
169         self._end_game = False
170         self._order_no = 0
171         self._block_now = ''
172         self._task_now = ''
173         self._actions = dict([('driving', False), ('rotating', False), ('picking', False), ('found', False)])
174         self._status = dict([('started', False), ('picked_up', False), ('found', False)])
175         self._img_frame = self.image
176         self._out = cv2.VideoWriter('grand_chall_vid.avi', self.fourcc, 30, (640, 480))
177         self._logger = logging.getLogger(name)
178         self._kp = self.Kp
179         self._ki = self.Ki
180         self._kd = self.Kd
181         self._direction = 'f'
182         self._event = threading.Event()
183         self._pause_event = threading.Event()

```

```

184     self.imu_reader = threading.Thread(target = self._imu_serial_straight,
185     self._ser = serial.Serial('/dev/ttyUSB0', 19200)
186     self._img_saved = self.image
187     self._survey_loc = self.scan_spots
188
189     # Set up logger with Debug initially
190     self._logger.setLevel(logging.DEBUG)
191     # Create console handler and set level to debug
192     console_handler = logging.StreamHandler()
193     console_handler.setLevel(logging.DEBUG)
194     # Log terminal messages to a file
195     file_handler = logging.FileHandler('gc_log_3.log')
196     file_handler.setLevel(logging.INFO)
197     # Create formatter
198     formatter = logging.Formatter('%(name)s - %(levelname)s - %(message)s')
199     # Add formatter to handler
200     console_handler.setFormatter(formatter)
201     file_handler.setFormatter(formatter)
202     # Add console handler to logger
203     self._logger.addHandler(console_handler)
204     self._logger.addHandler(file_handler)
205     # Initialize gpios and pwms
206     self._init()

```

Initialize Robot attributes

Arguments:

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

Returns:

None

def plot_path(self):

▼ View Source

```

1944     def plot_path(self):
1945         """
1946         Plot the path of the robot during the grand challenge.
1947
1948         Args:
1949             None
1950         Return:
1951             None
1952         """
1953         # Take in the path list as an array 2D NumPy array
1954         path = np.array(self._path)
1955         x = path[:,0]
1956         y = path[:,1]
1957
1958         # Create figure and plot
1959         fig, ax = plt.subplots(1,1)
1960
1961         fig.suptitle('Motor Encoder Analysis')

```

```

1962
1963         ax.plot(x,y,ls='solid', color='blue',linewidth=2, label=f'{self._name}')
1964
1965         # 3) Label the axes, title, legend
1966         ax.set_ylabel('Y')
1967         ax.set_xlabel('X')
1968         ax.set_xlim(0,10)
1969         ax.set_ylim(0,10)
1970         plt.savefig('grandChallenge-robotPath.png')
1971         plt.show()
-----

```

Plot the path of the robot during the grand challenge.

Arguments:

- None

Return:

None

def start(self, order: List):

▼ View Source

```

1974     def start(self, order: List):
1975         """
1976         The sequence of actions occuring while baron robot works to pick and p
1977         all toy blocks to the construction zone. This function lists the steps
1978         for the robot to complete the orders passed to it at the beginning of
1979
1980         Args:
1981             order: List[Blocks] list of all toy blocks needed to complete chal
1982
1983         Returns:
1984             None
1985         """
1986
1987         try:
1988             # Assign order from object initialization
1989             self._order = order
1990
1991             # First wait until startup QR Code has been read
1992             while True:
1993                 if self._wait2start():
1994                     break
1995
1996             while len(self._order) > 0:
1997                 """While Robot is still active, resume challenge"""
1998                 if (not self._end_game):
1999
2000                     self._block_now = self._order.pop(0)
2001                     self._order_no +=1
2002
2003                     # Begin challenge with choesn block
2004                     self._logger.info("-" * 20)

```



```

2005 self._logger.info(f"*** Picking Up {self._block_now} block
2006 self._logger.info("-" * 20)
2007
2008 # Track the number of times robot drives to search area
2009 search = 0
2010
2011 # Drive robot to center of arena, get view of toy blocks a
2012 # extracting blocks
2013 self._go2(destination=self.clutter_env,area='cluttered env
2014
2015 while True:
2016
2017     # Begin by scanning for toy block object
2018     degree = self._scan4object()
2019
2020     # Take image and save to video object
2021     #cntr_bbox.img_show("Detecting Block",self._img_frame)
2022     self._out.write(self._img_frame)
2023
2024     # Drive toward block if successful, else pivot to
2025     # a free space and drive
2026     if self._success:
2027         #self._logger(f"Pivoting Robot towards {self._block_now} block")
2028         self._pivot(degree,steady=True)
2029
2030     # Take image and save to video object
2031     self._picam_frame()
2032     __, __, __, __ = cntr_bbox.bbox_detect(self._img_frame)
2033     #cntr_bbox.img_show("Detecting Block",self._img_frame)
2034     self._out.write(self._img_frame)
2035
2036     self._logger.info(f"Drive Robot forward to {self._block_now} block")
2037     self._go2block()
2038
2039     # # Update global x and y
2040     # self.global_x += self._x
2041     # self.global_y += self._y
2042
2043     ##self._logger(f"{self._name} is now at x- {self.global_x} y- {self.global_y}")
2044
2045     # Take image and save to video object
2046     self._picam_frame()
2047     #cntr_bbox.img_show("Arrived at Block",self._img_frame)
2048     self._out.write(self._img_frame)
2049
2050     ##self._logger(f"Pick up {self._block_now} block with {self._pick_up}")
2051     self._pick_up()
2052
2053     # Drive robot to construction zone to place block
2054     ##self._logger(f"Take {self._block_now} block to construction zone")
2055     self._go2(destination=self.const_zone,area='construction zone')
2056
2057     #self._logger(f"Place {self._block_now} block inside construction zone")
2058     self._place()
2059
2060     # self._logger("Localizing to get accurate location")

```



```

2061         self._localize()
2062
2063         ##self._logger(f"Pick up and place the next order
2064         self._success = True
2065         break
2066     else:
2067         #self._logger("No toy blocks found in current pose
2068         #self._logger("try again.")
2069
2070         # Drive robot to the predetermined survey location
2071         if search > 1:
2072             self._go2(destination=self._survey_loc[search])
2073         else:
2074             #self._logger(f"Robot can't find {self._block_
2075             #self._logger("In pursuit of time, moving on t
2076             break
2077         search += 1
2078
2079     else:
2080         #self._logger(f"Mission complete. Take robot back to home base
2081         self._go2(destination=self.start_pt,area='home base')
2082         return
2083
2084     except KeyboardInterrupt as error:
2085         self._logger.error("Error occurred: %s",error)
2086         self._gameover()
2087         time.sleep(1)
2088         return None
2089     except ValueError as error:
2090         self._logger.error("Error %s", error, exc_info=True)
2091         self._gameover()
2092         time.sleep(1)
2093         return None

```

The sequence of actions occurring 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](#)

```
1  import cv2
2  import os
3  import imutils
4  import numpy as np
5  import matplotlib.pyplot as plot
6  import math
7  import picamera
8  from picamera.array import PiRGBArray
9  from picamera import PiCamera
10 import time
11 #from Locomotion.drive_robot import pivot, imu_serial,pwmZero,pwms
12
13 import smtplib
14 from smtplib import SMTP
15 from smtplib import SMTPException
16 import email
17 from email.mime.image import MIMEImage
18 from email.mime.text import MIMEText
19 from email.mime.multipart import MIMEMultipart
20 from datetime import datetime
21 import imaplib
22
23 # initialize the Raspberry Pi camera
24 camera = PiCamera()
25 camera.resolution = (640, 480)
26 camera.vflip = True
27 camera.framerate = 25
28 rawCapture = PiRGBArray(camera, size=(640,480))
29 time.sleep(2)
30
31 font = cv2.FONT_HERSHEY_COMPLEX_SMALL
32
33 # # Read original RGB Image from Library
34 # block = cv2.imread("greenBlock.png")
35 pics = []
36
37 ave_center = [0,0]
38 bbox_radius = 0
39
40 frame_center = (320,240)
41
42 (cx,cy) = (frame_center[0]/4.5, frame_center[1]/8)
43
44 # define the codec and create VideoWriter object
45 fourcc = cv2.VideoWriter_fourcc(*'XVID')
46 out = cv2.VideoWriter('blockRetrieval.avi', fourcc, 3, (640, 480))
47
```

```
48 # Open .txt file to save data
49 f = open('pxlradius_distance-02.txt','w')
50
51 image = np.empty((480*640*3,),dtype=np.uint8)
52
53 detected = False
54
55 obj = ''
56
57 pixel2deg = 0.061
58
59 # Initialize servo gripper states
60 closed = 2.5
61 half = 5
62 open_full = 7.5
63
64 # # create object to read camera
65 # video = cv2.VideoCapture(0)
66 #
67 # if (video.isOpened() == False):
68 #     print("Error reading video")
69
70 # def cv2_cam_frame():
71 #     # Pick image from video stream
72 #     success, image = video.read()
73 #
74 #     return image
75
76 def picam_frame():
77     global image
78
79     # Take picture with camera
80     camera.capture (image, format="bgr")
81     image = image.reshape((480,640,3))
82
83     image = cv2.flip(image,1)
84
85
86     return image
87
88 def img_show(name, img):
89     cv2.imshow(name,img)
90     # key = cv2.waitKey(1) & 0xFF
91     cv2.waitKey(0)
92     # Write frame to video
93     # out.write
94     cv2.destroyAllWindows()
95
96 def dist_img(img,dist):
97
98     dist = str(dist) + "cm"
99     data = f"{obj} Sodar: " + dist
100
101     cv2.putText(img, data, (355,30),cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,0),2)
102
103     return img
```

```

104
105 def servo_img(grip_state):
106
107     img = picam_frame()
108     data = str(grip_state)
109     data = "Duty Cycle: " + data + "%"
110
111     cv2.putText(img, data, (20,30),cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,0),2)
112
113     if grip_state == half:
114         cv2.putText(img, "Half-Opened", (20,55),cv2.FONT_HERSHEY_SIMPLEX,1,(0,125,125),2)
115     elif grip_state == open_full:
116         cv2.putText(img, "Fully-Opened", (20,55),cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,255),2)
117     elif grip_state == closed:
118         cv2.putText(img, "Fully-Closed", (20,55),cv2.FONT_HERSHEY_SIMPLEX,1,(255,0,0),2)
119
120     return img
121
122 def mask_color(image, imageHSV):
123     global obj
124     # HSV bounds
125     # minHSV = np.array([41,74,163])
126     # maxHSV = np.array([63,159,255])
127     # # Trail Green Block - LAB
128     # minHSV = np.array([47,56,172])
129     # maxHSV = np.array([255,255,255])
130
131     if obj == 'green':
132         # Trail Green Block - LAB
133         minHSV = np.array([46,88,121])
134         maxHSV = np.array([65,141,255])
135     elif obj == 'red':
136         # Trail Red block - LAB
137         minHSV = np.array([151,107,147])
138         maxHSV = np.array([255,255,255])
139     else:
140         # Trail Blue block - LAB
141         minHSV = np.array([72,94,97])
142         maxHSV = np.array([126,188,212])
143
144     # # Mask the gripper
145     # mask_gripper = np.zeros(image.shape[:2], np.uint8)
146     # mask_gripper[0:0, 380:0] = 255
147     #
148     # # Cut out gripper
149     # mask_img = cv2.bitwise_and(image, image, mask = mask_gripper)
150     maskHSV = cv2.inRange(imageHSV, minHSV, maxHSV)
151
152     return maskHSV
153
154 def prep_image(image):
155
156     # Convert image from BGR to HSV space
157     imageHSV = cv2.cvtColor(image,cv2.COLOR_BGR2HSV)
158
159     # mask the green light from HSV and convert to grayscale

```

```
160     mask = mask_color(image, imageHSV)
161
162     # Mask HSV masked image of arrow
163     blurred = cv2.GaussianBlur(mask,(11,11), 0)
164
165     return blurred
166
167 def corner_detect(img,origImg):
168
169     # Create a list to store the x,y location of points
170     pts_loc = []
171
172     # Detect corners from image
173     corners = cv2.goodFeaturesToTrack(img,5,0.01,10)
174
175     if corners is not None:
176
177         corners = np.int0(corners)
178
179         # identify location of corners in image
180         for i in corners:
181             # Extract x,y coordinate of points
182             x,y = i.ravel()
183
184             pts_loc.append([x,y])
185
186             # Draw circle ontop of original image
187             cv2.circle(origImg, (x,y), 3, (0,0,255),-1)
188
189             # Create a column vector from pts list
190             pts_loc = np.array(pts_loc)
191
192             return img, pts_loc, origImg, True
193     else:
194         return img, pts_loc, origImg, None
195
196 def center_det(pt_list, center):
197     global bbox_radius
198     # Extract x,y points from pt_list
199     x = pt_list[:,0]
200     y = pt_list[:,1]
201
202     # Determine the min and max width & height values
203     # of the points, as if to draw rectangle around arrow
204     x_min = x.min()
205     y_min = y.min()
206
207     x_max = x.max()
208     y_max = y.max()
209
210     # Store height of bounding box
211     vert_dst = y_max - y_min
212
213     # Store width of bounding box
214     horz_dst = x_max - x_min
215
```

```

216     # Compute and store half dimensions of
217     # box, will come later when determining
218     # arrow direction
219     y_half = vert_dst/2 + y_min
220     x_half = horz_dst/2 + x_min
221
222     # Store center of the block
223     #center = [int(round(x_half)), int(round(y_half))]
224     center.append(int(round(x_half)))
225     center.append(int(round(y_half)))
226     # Estimate radius
227     bbox_radius = int(round(math.sqrt((x_max - x_half)**2 + (y_max - y_half)**2)))
228
229     return center, bbox_radius
230
231
232 def bbox_detect(image, image_blurred):
233     global bbox_radius
234     global obj
235
236     center = []
237     radius = 0
238     success = None
239
240     # Extract frame from camera
241     image = picam_frame()
242     #
243     # Prep image for object detection
244     img_blurred = prep_image(image)
245
246     # Detect corners from image
247     img_crnr, pts_loc, image, var = corner_detect(image_blurred, image)
248
249     if var is not None:
250
251         # Check if corners are detected
252         if len(pts_loc) > 3:
253             ## Draw contours over an image, if available
254             center, bbox_radius = center_det(pts_loc, center)
255
256             block_coordinate = "(" + str(center[0]) + "," + str(center[1]) + ")"
257
258             # Draw circle ontop of original image
259             cv2.circle(image, tuple(center), bbox_radius, (0,255,255),2)
260             cv2.circle(image, tuple(center), 0,(0,0,255),5)
261             cv2.putText(image,block_coordinate,(0,int(cy)),font,2,(0,0,0),2)
262
263             # Draw a cross at center of frame
264             cv2.line(image,(frame_center[0]-100, frame_center[1]),(frame_center[0]+
265             cv2.line(image,(frame_center[0], frame_center[1]-100),(frame_center[0],
266
267             success = True
268
269             print(f"{obj} block is detected")
270             time.sleep(1)
271

```

```
272         return image, center, bbox_radius, success
273
274     else:
275         print("Not clearly detected")
276         success = False
277
278         return image, center, bbox_radius, success
279
280     else:
281         print("No object of interest in scene")
282         return image, 0, 0, success
283
284 def scanObject(color):
285     global ave_center
286     global obj
287     global bbox_radius
288
289     detected = False
290     obj = color
291     # Rotate every small angle, take a picture to detect existence of toy block
292     stat = 0
293     turn_angle = 15
294     deg_diff = 0
295     yaw = imu_serial()
296     yaw_orig = yaw
297
298     while not detected:
299
300         image, ave_center, bbox_radius, success = bbox_detect()
301
302         # write frame into file
303         # out.write(image)
304         if success:
305             diff = ave_center[0]-frame_center[0]
306             # Convert the pixel difference to degrees
307             deg_diff = diff * pixel2deg
308
309             detected = True
310
311             return detected, deg_diff, image
312
313         else:
314             # Turn the robot by 15 degrees
315             #stat += turn_angle
316
317             return detected, turn_angle, image
318
319         yaw = pivot(deg_diff= turn_angle)
320         time.sleep(1)
321
322
323     if stat >=360:
324         print("Could not detect block from area")
325         print("Drive to a different location and try again")
326         break
327
```

```

328     else:
329         print("Desired object has been located")
330
331     return detected, deg_diff, image
332
333 def dist_estimate():
334     # Read in pixel radius from image
335     global bbox_radius
336
337     img, __, bbox_radius, __ = bbox_detect()
338
339     # Plug in pixel size into depth equation
340     distance = (0.0099 * bbox_radius**2) - (1.8846*bbox_radius) + 103.47
341
342     # Append image of estimated distance on image frame
343     image = dist_img(img, round(distance, 2))
344
345     return distance, image
346
347 def email_media(obj):
348
349     image = picam_frame()
350
351     # send email to user with images
352     smtpUser = 'ykebede2@terpmail.umd.edu'
353     smtpPass = 'QwE@$d1219'
354
355     toAdd = ['ENPM809TS19@gmail.com', 'yosephcollege@gmail.com']
356     fromAdd = smtpUser
357
358     f_time = datetime.now().strftime('%a %d %b @ %H:%M')
359     subject = f'{obj} block picked up for transportation to construction area: ' + .
360
361     msg = MIMEMultipart()
362     msg['Subject'] = subject
363     msg['From'] = fromAdd
364     msg['To'] = ",".join(toAdd)
365
366     msg.preamble = "Image @ " + f_time
367
368     body = email.mime.text.MIMEText("Baron Robot image: " + f_time)
369     msg.attach(body)
370
371     fp = open(sodar_img, 'rb')
372     img = MIMEImage(fp.read())
373     fp.close()
374     msg.attach(img)
375
376     fp = open(grip_img, 'rb')
377     img = MIMEImage(fp.read())
378     fp.close()
379     msg.attach(img)
380
381     s = smtplib.SMTP('smtp.gmail.com', 587)
382     s.ehlo()
383     s.starttls()

```



```
384     s.ehlo()
385     s.login(smtpUser, smtpPass)
386     s.sendmail(fromAdd, toAdd, msg.as_string())
387     s.quit()
388
389     print("Email delivered!")
390
391     # if __name__ == "__main__":
392     #
393     #     global obj
394     #     i = 20
395     #     block = ['g', 'b', 'r']
396     #
397     #     while True:
398     #
399     #         obj = block.pop(0)
400     #         obj = 'g'
401     #
402     #         detected, deg_diff = detectObject()
403     #
404     #         # Read image from picamera
405     #         image = picam_frame()
406     #
407     #         # to accomodate for pi camera mount
408     #         image = cv2.flip(image, 1)
409     #
410     #         # Draw bbox
411     #         radius = bbox_detect(image)
412     #
413     #         outstring = str(i) + ',' + str(radius) + '\n'
414     #         f.write(outstring)
415     #
416     #         i += 10
417     #
418     #     f.close()
419     #
420     #     for i in range(40,200,30):
421     #
422     #         # Read an image from library
423     #         image = cv2.imread(f"block_pics/blocks_{i}.png")
424     #
425     #         # Convert image from BGR to HSV space
426     #         imageHSV = cv2.cvtColor(image,cv2.COLOR_BGR2HSV)
427     #
428     #         tmp = cv2.cvtColor(block, cv2.COLOR_BGR2GRAY)
429     #
430     #         _, alpha = cv2.threshold(tmp, 0, 200, cv2.THRESH_BINARY)
431     #
432     #         b, g, r = cv2.split(block)
433     #
434     #         rgba = [b, g, r, alpha]
435     #
436     #         dst = cv2.merge(rgba,4)
437     #
438     #         # mask the green light from HSV and convert to grayscale
439     #         mask = mask_color(image, imageHSV)
```

```

440 #
441 #     blurred = cv2.GaussianBlur(mask,(11,11), 0)
442 #
443 #     # Detect corners from image
444 #     img_crnr, pts_loc, image = corner_detect(mask, image)
445 #
446 #     ## Draw contours over an image, if available
447 #     center, radius = center_det(pts_loc)
448 #
449 #     block_coordinate = "(" + str(center[0]) + "," + str(center[1]) + ")"
450 #
451 #     # Draw circle ontop of original image
452 #     cv2.circle(image, center, radius, (0,255,255),2)
453 #     cv2.circle(image, center, 0,(0,0,255),5)
454 #     cv2.putText(image,block_coordinate,(0,int(cy/2)),font,2,(0,0,0),2)
455 #
456 #     # Draw a cross at center of frame
457 #     cv2.line(image,(frame_center[0]-100, frame_center[1]),(frame_center[0]+100,frame_center[1]),(0,0,255),2)
458 #     cv2.line(image,(frame_center[0], frame_center[1]-100),(frame_center[0], frame_center[1]+100),(0,0,255),2)
459 #
460 #     cv2.imshow('pixel map of block',image)
461 #     cv2.imshow('black mask',mask)
462 #     cv2.waitKey(0)
463 #     cv2.destroyAllWindows()
464 #     pxl_radius.append(radius)
465 #
466 #     outstring = str(i) + ' ' + str(radius) + '\n'
467 #     f.write(outstring)

```

def picam_frame():

[▼ View Source](#)

```

77 def picam_frame():
78     global image
79
80     # Take picture with camera
81     camera.capture (image, format="bgr")
82     image = image.reshape((480,640,3))
83
84     image = cv2.flip(image,1)
85
86
87     return image

```

def img_show(name, img):

[▼ View Source](#)

```

89 def img_show(name, img):
90     cv2.imshow(name,img)
91     #     key = cv2.waitKey(1) & 0xFF
92     cv2.waitKey(0)
93     # Write frame to video
94     # out.write

```

95 `cv2.destroyAllWindows()`

def `dist_img`(img, dist):

▼ View Source

```
97 def dist_img(img,dist):
98
99     dist = str(dist) + "cm"
100     data = f"{obj} Sodar: " + dist
101
102     cv2.putText(img, data, (355,30),cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,0),2)
103
104     return img
```

def `servo_img`(grip_state):

▼ View Source

```
106 def servo_img(grip_state):
107
108     img = picam_frame()
109     data = str(grip_state)
110     data = "Duty Cycle: " + data + "%"
111
112     cv2.putText(img, data, (20,30),cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,0),2)
113
114     if grip_state == half:
115         cv2.putText(img, "Half-Opened", (20,55),cv2.FONT_HERSHEY_SIMPLEX,1,(0,125,125))
116     elif grip_state == open_full:
117         cv2.putText(img, "Fully-Opened", (20,55),cv2.FONT_HERSHEY_SIMPLEX,1,(0,0,255))
118     elif grip_state == closed:
119         cv2.putText(img, "Fully-Closed", (20,55),cv2.FONT_HERSHEY_SIMPLEX,1,(255,0,0))
120
121     return img
```

def `mask_color`(image, imageHSV):

▼ View Source

```
123 def mask_color(image, imageHSV):
124     global obj
125     # HSV bounds
126     # minHSV = np.array([41,74,163])
127     # maxHSV = np.array([63,159,255])
128     # # Trail Green Bock - LAB
129     # minHSV = np.array([47,56,172])
130     # maxHSV = np.array([255,255,255])
131
132     if obj == 'green':
133         # Trail Green Bock - LAB
134         minHSV = np.array([46,88,121])
135         maxHSV = np.array([65,141,255])
136     elif obj == 'red':
```

```
137         # Trail Red block - LAB
138         minHSV = np.array([151,107,147])
139         maxHSV = np.array([255,255,255])
140     else:
141         # Trail Blue block - LAB
142         minHSV = np.array([72,94,97])
143         maxHSV = np.array([126,188,212])
144
145     # # Mask the gripper
146     # mask_gripper = np.zeros(image.shape[:2], np.uint8)
147     # mask_gripper[0:0, 380:0] = 255
148     #
149     # # Cut out gripper
150     # mask_img = cv2.bitwise_and(image, image, mask = mask_gripper)
151     maskHSV = cv2.inRange(imageHSV, minHSV, maxHSV)
152
153     return maskHSV
```

def `prep_image`(image):[▼ View Source](#)

```
155 def prep_image(image):
156
157     # Convert image from BGR to HSV space
158     imageHSV = cv2.cvtColor(image,cv2.COLOR_BGR2HSV)
159
160     # mask the green light from HSV and convert to grayscale
161     mask = mask_color(image, imageHSV)
162
163     # Mask HSV masked image of arrow
164     blurred = cv2.GaussianBlur(mask,(11,11), 0)
165
166     return blurred
```

def `corner_detect`(img, origImg):[▼ View Source](#)

```
168 def corner_detect(img,origImg):
169
170     # Create a list to store the x,y location of points
171     pts_loc = []
172
173     # Detect corners from image
174     corners = cv2.goodFeaturesToTrack(img,5,0.01,10)
175
176     if corners is not None:
177
178         corners = np.int0(corners)
179
180         # identify location of corners in image
181         for i in corners:
182             # Extract x,y coordinate of points
183             x,y = i.ravel()
184
```

```
185         pts_loc.append([x,y])
186
187         # Draw circle ontop of original image
188         cv2.circle(origImg, (x,y), 3, (0,0,255),-1)
189
190         # Create a column vector from pts list
191         pts_loc = np.array(pts_loc)
192
193         return img, pts_loc, origImg, True
194     else:
195         return img, pts_loc, origImg, None
```

def center_det(pt_list, center):

▼ View Source

```
197 def center_det(pt_list, center):
198     global bbox_radius
199     # Extract x,y points from pt_list
200     x = pt_list[:,0]
201     y = pt_list[:,1]
202
203     # Determine the min and max width & height values
204     # of the points, as if to draw rectangle around arrow
205     x_min = x.min()
206     y_min = y.min()
207
208     x_max = x.max()
209     y_max = y.max()
210
211     # Store height of bounding box
212     vert_dst = y_max - y_min
213
214     # Store width of bounding box
215     horz_dst = x_max - x_min
216
217     # Compute and store half dimensions of
218     # box, will come later when determining
219     # arrow direction
220     y_half = vert_dst/2 + y_min
221     x_half = horz_dst/2 + x_min
222
223     # Store center of the block
224     #center = [int(round(x_half)), int(round(y_half))]
225     center.append(int(round(x_half)))
226     center.append(int(round(y_half)))
227     # Estimate radius
228     bbox_radius = int(round(math.sqrt((x_max - x_half)**2 + (y_max - y_half)**2)))
229
230     return center, bbox_radius
```

def bbox_detect(image, image_blurred):

▼ View Source

```
233 def bbox_detect(image, image_blurred):
234     global bbox_radius
235     global obj
236
237     center = []
238     radius = 0
239     success = None
240
241     # # Extract frame from camera
242     # image = picam_frame()
243     #
244     # # Prep image for object detection
245     # img_blurred = prep_image(image)
246
247     # # Detect corners from image
248     img_crnr, pts_loc, image, var = corner_detect(image_blurred, image)
249
250     if var is not None:
251
252         # # Check if corners are detected
253         if len(pts_loc) > 3:
254             ## # Draw contours over an image, if available
255             center, bbox_radius = center_det(pts_loc, center)
256
257             block_coordinate = "(" + str(center[0]) + "," + str(center[1]) + ")"
258
259             # # Draw circle ontop of original image
260             cv2.circle(image, tuple(center), bbox_radius, (0,255,255),2)
261             cv2.circle(image, tuple(center), 0,(0,0,255),5)
262             cv2.putText(image,block_coordinate,(0,int(cy)),font,2,(0,0,0),2)
263
264             # # Draw a cross at center of frame
265             cv2.line(image,(frame_center[0]-100, frame_center[1]),(frame_center[0]+
266             cv2.line(image,(frame_center[0], frame_center[1]-100),(frame_center[0],
267
268             success = True
269
270             print(f"{obj} block is detected")
271             time.sleep(1)
272
273             return image, center, bbox_radius, success
274
275         else:
276             print("Not clearly detected")
277             success = False
278
279             return image, center, bbox_radius, success
280
281     else:
282         print("No object of interest in scene")
283         return image, 0, 0, success
```

def scanObject(color):[▼ View Source](#)

```
285 def scanObject(color):
286     global ave_center
287     global obj
288     global bbox_radius
289
290     detected = False
291     obj = color
292     # Rotate every small angle, take a picture to detect existence of toy block
293     stat = 0
294     turn_angle = 15
295     deg_diff = 0
296     yaw = imu_serial()
297     yaw_orig = yaw
298
299     while not detected:
300
301         image, ave_center, bbox_radius, success = bbox_detect()
302
303         # write frame into file
304     #     out.write(image)
305         if success:
306             diff = ave_center[0]-frame_center[0]
307             # Convert the pixel difference to degrees
308             deg_diff = diff * pixel2deg
309
310             detected = True
311
312             return detected, deg_diff, image
313
314         else:
315             # Turn the robot by 15 degrees
316             #stat += turn_angle
317
318             return detected, turn_angle, image
319
320             yaw = pivot(deg_diff= turn_angle)
321             time.sleep(1)
322
323
324             if stat >=360:
325                 print("Could not detect block from area")
326                 print("Drive to a different location and try again")
327                 break
328
329         else:
330             print("Desired object has been located")
331
332     return detected, deg_diff, image
```

def dist_estimate():[▼ View Source](#)

```

334 def dist_estimate():
335     # Read in pixel radius from image
336     global bbox_radius
337
338     img, __, bbox_radius, __ = bbox_detect()
339
340     # Plug in pixel size into depth equation
341     distance = (0.0099 * bbox_radius**2) - (1.8846*bbox_radius) + 103.47
342
343     # Append image of estimated distance on image frame
344     image = dist_img(img,round(distance,2))
345
346     return distance, image

```

def email_media(obj):

▼ View Source

```

348 def email_media(obj):
349
350     image = picam_frame()
351
352     # send email to user with images
353     smtpUser = 'ykebede2@terpmail.umd.edu'
354     smtpPass = 'QwE@$d1219'
355
356     toAdd = ['ENPM809TS19@gmail.com','yosephcollege@gmail.com']
357     fromAdd = smtpUser
358
359     f_time = datetime.now().strftime('%a %d %b @ %H:%M')
360     subject = f'{obj} block picked up for transportation to construction area: ' + .
361
362     msg = MIMEMultipart()
363     msg['Subject'] = subject
364     msg['From'] = fromAdd
365     msg['To'] = ",".join(toAdd)
366
367     msg.preamble = "Image @ " + f_time
368
369     body = email.mime.text.MIMEText("Baron Robot image: " + f_time)
370     msg.attach(body)
371
372     fp = open(sodar_img, 'rb')
373     img = MIMEImage(fp.read())
374     fp.close()
375     msg.attach(img)
376
377     fp = open(grip_img, 'rb')
378     img = MIMEImage(fp.read())
379     fp.close()
380     msg.attach(img)
381
382     s = smtplib.SMTP('smtp.gmail.com', 587)
383     s.ehlo()
384     s.starttls()

```



```
385     s.ehlo()  
386     s.login(smtpUser, smtpPass)  
387     s.sendmail(fromAdd, toAdd, msg.as_string())  
388     s.quit()  
389     . . . . .
```



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

