1 Introduction

The Raspberry Pi today is used extensively in several applications especially in embedded systems and in the world of IoT (Internet of Things) due to its compact size and relatively high performance at a low power consumption. One such use is computer vision, where one can design a security camera and detect suspicious activity. In this report, I will be discussing and be showing an implementation of a robot that stack blocks based on color. The robot moves around the environment randomly and finds blocks of specific colors whilst avoiding obstacles using an ultrasonic sensor after which it goes to the stacking area by means of a line follower.

1.1 Motivation

To test and understand the capabilities of the Raspberry Pi whilst learning how it is being used in computer vision.

1.2 Problem Statement

A fully automated robot that stacks blocks without any human intervention where the blocks are located randomly through out the environment.

2 Design

2.1 Requirements Constraints, and Considerations

The primary requirement is for the robot to be able to stack a particular type of item automatically by searching for them in a given environment. After which it must be able to bring back the item to the starting position so that it may be stacked. The maximum possible number of items that can be stacked will depend on the size of the item itself and the length of the robot arm.

2.2 System Overview

The system involves using the Raspberry pi camera for detecting the colored blocked by applying a mask to each incoming individual frame. The obstacle avoidance utilizesan ultrasonic sensor attached to a servo so it to able

to measure the distances from different angles, primarily at angle 45(left), 90(straight) and 135(right). The Raspberry Pi 4 itself is powered using a 10000 mAh battery that is capable of supplying 5.1V at 2.4 A an can run operate for several hours before needing to recharge . The line following algorithm uses two IR sensor attached under the robot and positioned just above the ground. The robot arm that is being used to grab the block has 4 degrees of freedom i.e. has 4 high torque servo which are powered externally through 2 18650 Li-Po batteries in order to provide the sufficient current. The DC motors on the other hand are powered by 3 18650 Li-Po batteries so as to be move the robot and carry all of the weight of the components mentioned above.

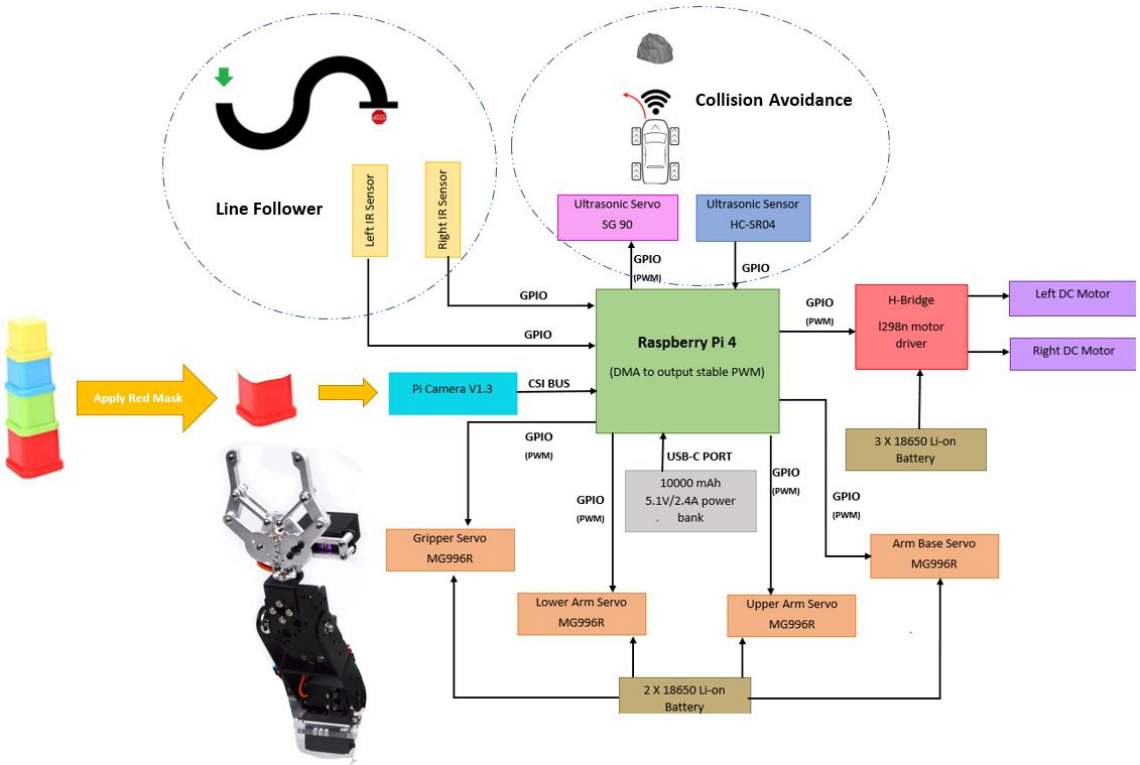

Figure 1: System Overview.

2.3 Component Design

The robot was assembled using 1mm acrylic sheets in order to provide two additional layers(floors) to make space for the Raspberry pi and the power bank. Furthermore, the robot is made up of multiple components. The names of these components and their placement are shown in figures 2, 3 & 4.

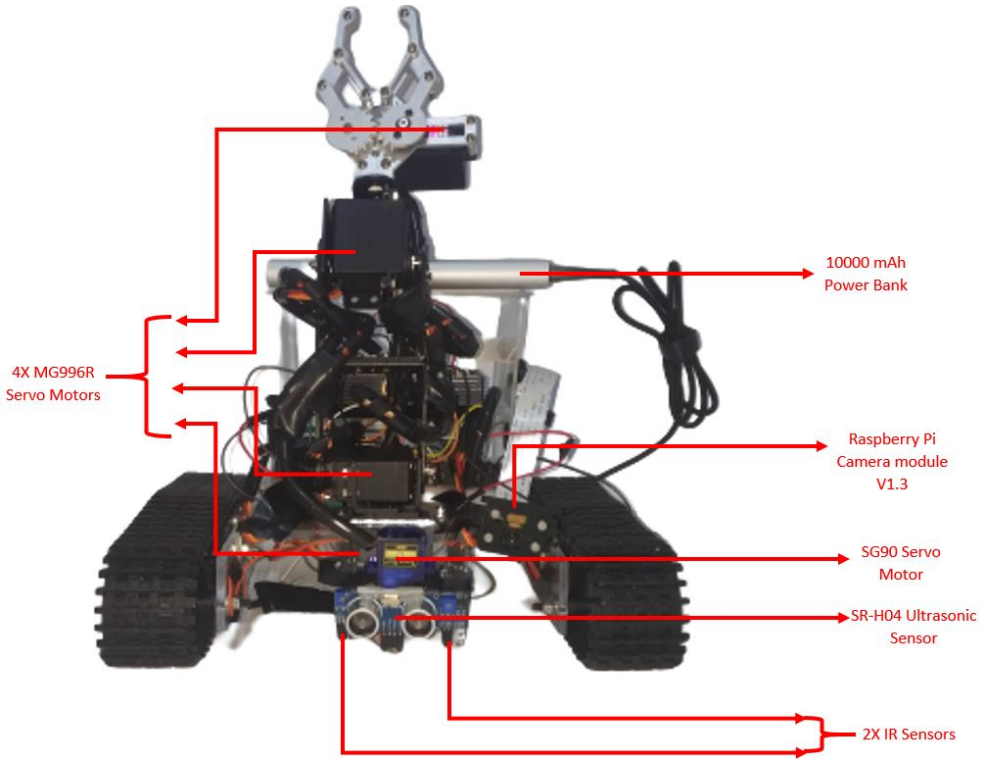

Figure 2: Front View.

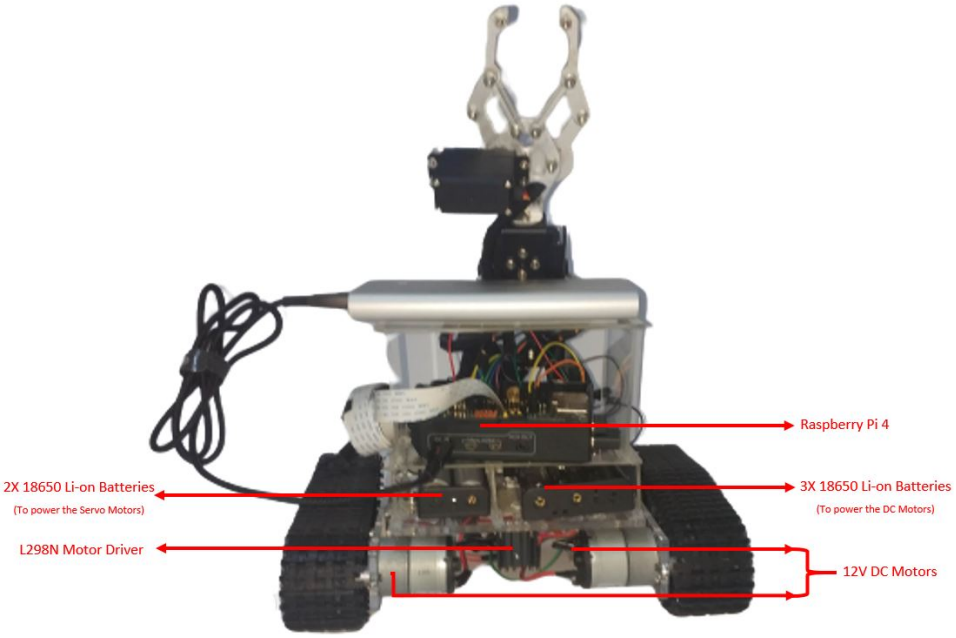

Figure 3: Rear View.

Figure 4: Side View.

2.4 Design Process

2.4.1 Statechart

The robot can be controlled manually using a Bluetooth controller, as shown in figure 5, and is primary used for debugging and finding and tuning the parameters of the angles of the servos and the speed of the DC motors so as to determine the right values needed for accurate grabbing of a block and the speed of robot respectively. Moreover, it helps to regain back control during the testing phase of the automatic mode in case something goes wrong and thus preventing the robot from crashing and getting damaged.

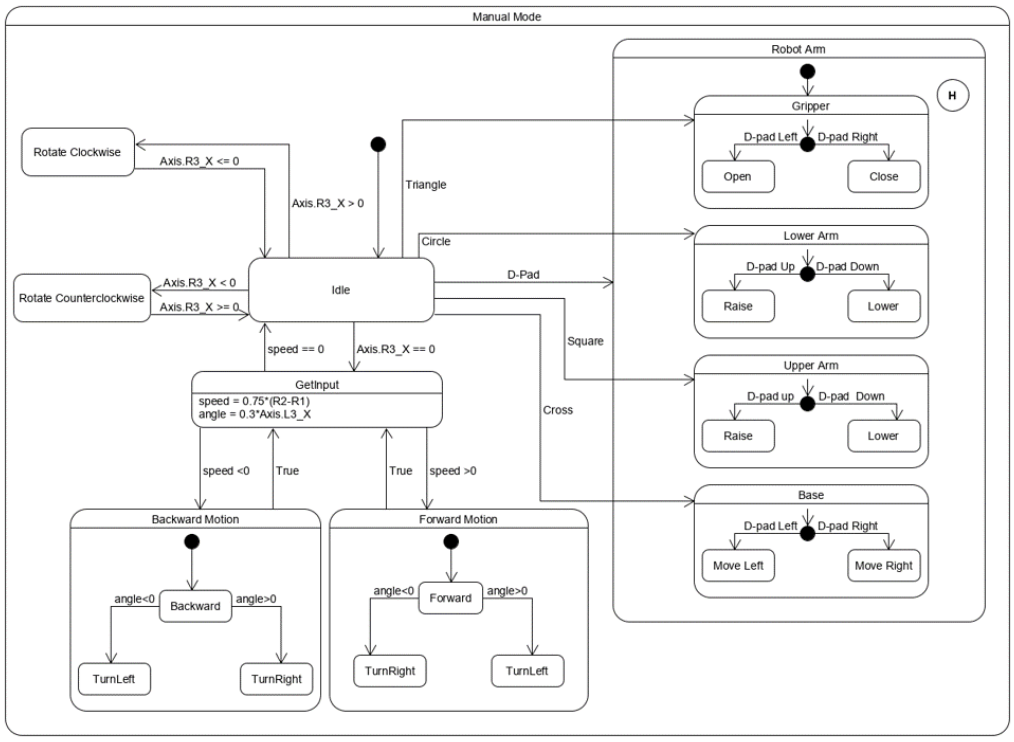

Figure 5: Manual Mode FSM.

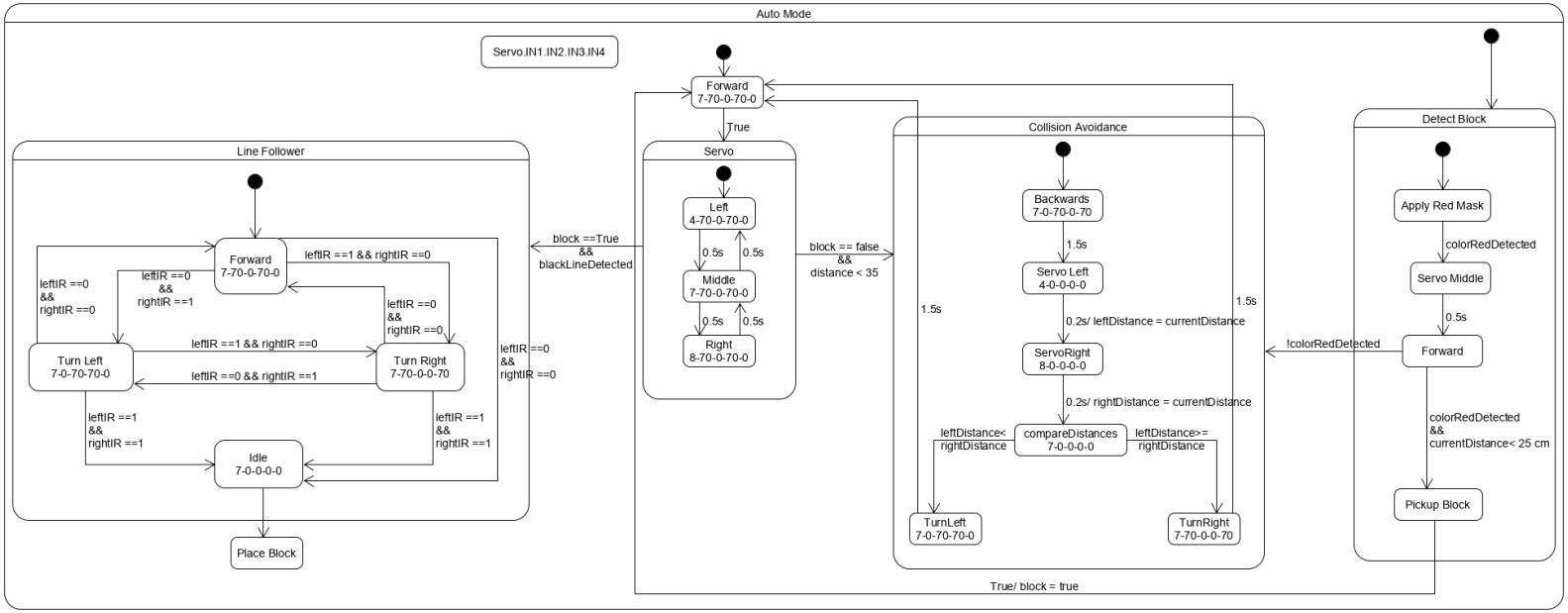

Figure 6: Automatic Mode FSM.

Figure 7: Overall System FSM.

As shown in the automatic mode FSM in figure 6, the robot initially moves forward and whilst doing so continues measuring its left, front an right distances. When the the distance between it and the obstacle is less than 35cm, then the collision avoidance algorithm gets activated where it moves backwards for 1.5 seconds. After which it checks the left and right distances by changing the angle of the servo motor where the ultrasonic sensor is attached and evaluates whether the left distance is greater than the right distance and if so it moves to the left, else it move to the right. While all this is happening the block detection is running concurrently where the the camera is continuously applying a red mask to each frame to detect a red block. Once a red block gets detected, it moves towards the block slowly while aligning itself based on the center x axis position of the block relative to the width of the frame. When the distance between the block and the robot is less than 25cm , the robot arm moves down and initiates to grab the block. The robot then proceeds to randomly move around the environment but this time the block detection algorithm is disabled and is instead is searching for a black line which will be detected by the IR sensors. Once detected, it first aligns the robot to the black line and then switches on the line following algorithm where it compares the reading for the left and right IR sensors and bases its movement on that. When both the IR sensor senses a black line , the robot stops indicating the destination has been reached after which the arm is lowered gradually and the block is placed.

The automatic mode and manual mode switches seamlessly by pressing on the share button on the dualshock 4 Bluetooth controller as shown in figure 7.

2.5 Hardware Schematic

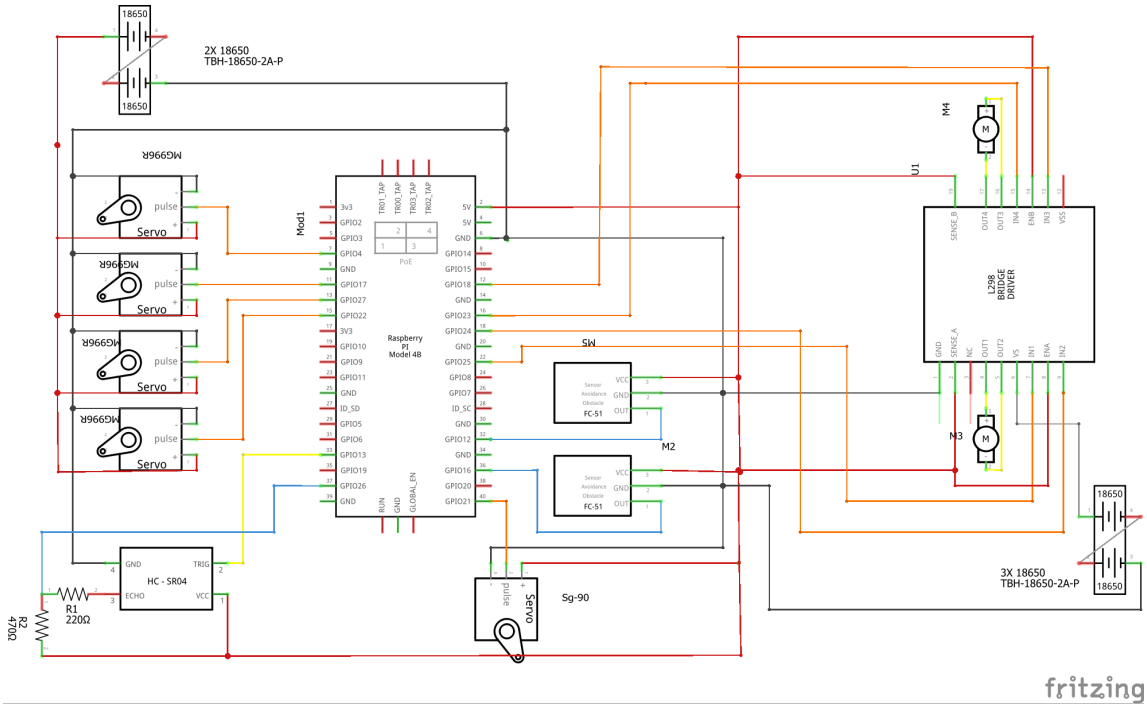

Figure 8: Hardware connections of the overall system.

Although the Raspberry pi does come with 4 hardware PWM pins, my implementation of the stacker requires 9 PWM pins, 4 for the high torques servo(robot arm), 1 for the servo motor where the ultrasonic sensor is attached, and another 4 for the L293D Module. The reason for using 4 PWM signals(IN1,IN2,IN3 and IN4) rather than 2(Enable 1 and Enable 2) for the L293D is because the maximum output voltage of a GPIO pin of 3.3V which is 66% of 5V. And so if I were to use the latter configuration, then that means 66% from the enable line and 66% from the INPUT pins which is essentially (66% of 66%) 43.6% duty cycle which should severely limit the speed of the robot. As shown from hardware schematic in figure 8, an external servo driver is not being used to generate the PWM signals for the servo but insteadm I am using the Pi-blaster library which gives Hardware Timed(DMA) PWM signal which is significantly more stable than a software generated PWM signal and uses very little of the CPU and is indistinguishable from a hardware generated PWM signal. The best part is that the library can effectively

use all 32 GPIO pins to generate PWM signals. However it is to be noted that using this requires using the DMA controller on the Raspberry pi and as a result the sound will not be generated properly but for purpose of this project, sound is not required.
By default, the library generates 100Hz PWM signals. Although, supplying this frequency to the servos will allow them to handle larger loads however will cause them to draw excess current and overheat(at least for the servos that I am using which are meant to operate at 50Hz). The following codes generates a duty cycle of 80% on pin 17 using Pi-blaster.

```
os.system("echo ""{}={}"" >/dev/pi-blaster".format(17,0.8))
```

2.6 Code

2.6.1 Collision Avoidance

The following code shows the collision avoidance algorithm and is being run concurrently on a separate thread. The servo, where the ultrasonic sensor is attached, continuously changes its angles in the following sequence: left, center right & center and whilst doing so measures the distance of that direction. If the distance at any time is less than 35 cm, then the robot moves backwards for 1.5 seconds and then measures and compares the left and right distances by changing the servo angle with a delay of 0.75 seconds to give the servo sufficient time to get in position. If it is found that the right distance is greater than the left distance then the robot moves to the right else it moves to the left.

```
def collisionAvoidance(self):
       while True:

           if not self.loop:
               break

           self.backward()
           sleep(1.5)
           self.stop()
           print("STOP SUCCESS")

           if not self.loop:
```

```
            break
        os.system("echo ""{}={}"" >
            /dev/pi-blaster".format(self.servoPin,sensorLeft))
        sleep(0.75)

        leftDistance = self.getDistance()

        print("TURN1 SUCCESS")

        if not self.loop:
            break
        os.system("echo ""{}={}"" >
            /dev/pi-blaster".format(self.servoPin,sensorRight))
        sleep(0.75)
        rightDistance = self.getDistance()

        os.system("echo ""{}={}"" >
            /dev/pi-blaster".format(self.servoPin,sensorCenter))
        sleep(0.75)

        if not self.loop:
            break

        if rightDistance > leftDistance:
            print("right > left")
            self.right()
            sleep(0.75)
            break

        else:
            print("left > right")
            self.left()
            sleep(0.75)
            break

    def moveServoandCheckDistance(self,value):
        if self.loop:
            self.forward()
            os.system("echo ""{}={}"" >
                /dev/pi-blaster".format(self.servoPin,value))
```

```
            sleep(0.2)
            if self.getDistance() < 35:
                self.collisionAvoidance()

    def run(self):
        while True:
            if self.loop:
                self.moveServoandCheckDistance(sensorLeft)
                self.moveServoandCheckDistance(sensorCenter)
                self.moveServoandCheckDistance(sensorRight)
                self.moveServoandCheckDistance(sensorCenter)
```

2.6.2 Color Detection

The first thing is to determine the lower and upper limits of each color in hsv. Since I have three colored blocks, it means I would require a lower and upper limit for each totaling to six parameters. The parameters were determined through tuning and testing but these value will vary slightly based on the lighting conditions of the environment. The following code captures a frame and applies to it the upper and lower limit by performing a bitwise_AND operation on each pixel. Figures 10, 11 and 12 show the result of this operation by applying a red, orange and yellow mask respectively.

```
if blockColor == Color.RED:
    lower = red_lower
    upper = red_upper
elif blockColor == Color.ORANGE:
    lower = orange_lower
    upper = orange_upper
elif blockColor == Color.YELLOW:
    lower = yellow_lower
    upper = yellow_upper

fps = FPS().start()
frame = vs.read()

hsv=cv2.cvtColor(frame,cv2.COLOR_BGR2HSV)

#finding the range of red,orange and yellow color in the image
```

```
color=cv2.inRange(hsv, lower, upper)

#Morphological transformation, Dilation
kernal = np.ones((5 ,5), "uint8")

color=cv2.dilate(color, kernal)
res=cv2.bitwise_and(frame, frame, mask = color)
```

Figure 9: Raw Frame.

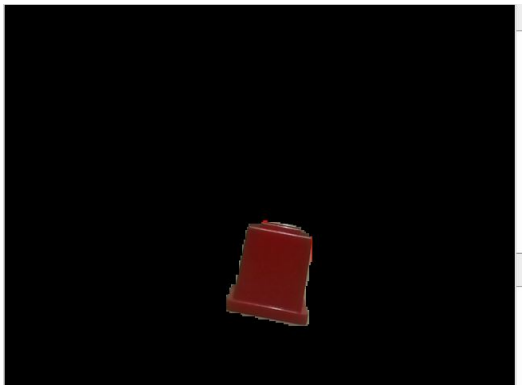

Figure 10: Applying a red mask.

Figure 11: Applying a orange mask.

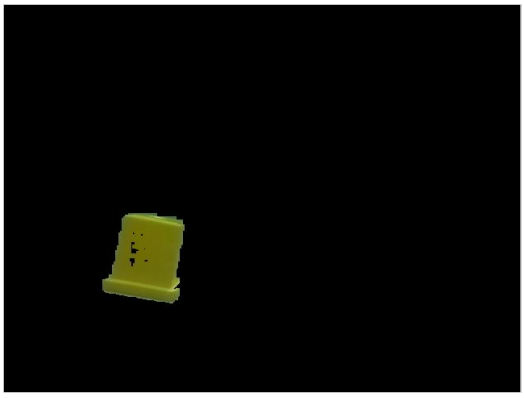

Figure 12: Applying a yellow mask.

The next step is to bound the color and find its edges and this can be done using findContours function in the openCV library. However there will be slight colors anomalies being detected so we adjust the area of the contour that is large enough for the block to be detected and the other slight color variations to be omitted and can use the contours to draw a rectangle around its edges. Once a large enough color is detected, the obstacle avoidance is disabled and the robot moves forward towards the color slowly since the framerate on average is 13 fps. In order to align the robot with the block, the boundingRect function returns the starting x, y coordinates of the drawn rectangle as well as the width and height. We can use this to find the mid point of the rectangle using the formula (x + (x+y))/2. Based on this center and its position in relation to the width of the window(my window size is

640) we then find the right set of values through experimentation and alight the robot left and right.

```
(_,contours,hierarchy)=cv2.findContours(color,cv2.RETR_TREE,\
cv2.CHAIN_APPROX_SIMPLE)

for pic, contour in enumerate(contours):

    area = cv2.contourArea(contour)

    if area > 300:

            if obstacleAvoidance.loop:
                print("Block Found!")
                obstacleAvoidance.setLoop(False)
                blockDetected = True
                os.system("echo ""{}={}"" >
                    /dev/pi-blaster".format(servoPin,0.07))# center
                slowForward()


            x,y,w,h = cv2.boundingRect(contour)
            block_X_Center = (x+x+w)/2

            frame = cv2.rectangle(frame,(x,y),(x+w,y+h),(0,0,255),2)
#
    cv2.putText(frame,"red",(x,y),cv2.FONT_HERSHEY_SIMPLEX, 0.7,
    (0,0,255))

            if block_X_Center > 480:
                slowRight()
            elif block_X_Center < 290:
                slowLeft()
            else:
                slowForward()
            break
```

2.6.3 Line Follower

After the block is picked up, the collision avoidance is reactivated and it roams around the environment until any of the IR sensors detect a black line. The robot now needs to transition from roaming around randomly to aligning itself to the line so the line follower algorithm may be started.

The black line in the environment was placed in such a way that the robot can only enter the line from one side only. And knowing this, there can be three possible scenarios. The first being when both the IR sensors detect the line, then the lineBoth function is executed and the robot moves slowly to the right until both IR sensors detect a negative reading after which the lineFollower is activated. The second scenario is only when the left IR sensor detects the line and then executes the lineLeft function. The last scenario is when only the right IR detects a line and then executes which the lineRight function. The latter two scenarios have not been tested thoroughly due to the breakage of one of the wheels.
As for the actual line follower algorithm itself, if only the left IR sensor detects a line then the robot turns left. In the same way, if only the right IR sensor detects a line, then the robot turns right. If both the IR sensors detect a line, that means it has reached the staking area and stops else it moves forward if both sensors detect no line.

```
def lineFollower():
    while True:
        print("Following Line")
        left_detect = IO.input(12)
        right_detect = IO.input(16)

        if left_detect == 0 and right_detect == 0:
            forward()
        if left_detect == 0 and right_detect == 1:
            right()
        if left_detect == 1 and right_detect == 0:
            left()
        if left_detect == 1 and right_detect == 1:
            stop()
            break;

```

```
def lineBoth():
    print("lineBoth")
    left()
    while True:
        left_detect = IO.input(12)
        right_detect = IO.input(16)

        if right_detect == 0 or left_detect == 0:
            lineFollower()
            break
def lineLeft():
    print("lineLeft")
    left()
    while True:
        left_detect = IO.input(12)
        right_detect = IO.input(16)
        if left_detect == 0:
            lineFollower()
            break
def lineRight():
    print("lineRight")
    left()
    while True:
        left_detect = IO.input(12)
        right_detect = IO.input(16)

        if right_detect == 0 and left_detect == :
            lineLeft()
            break
        elif right_detect == 1 and left_detect == 1:
            lineBoth()
            break
```

2.7 Results and analysis

Note: During the testing phase, one of the wheels came loose and came out and I am no longer able to attach it and as a result the transition to the line Following state is incomplete though the actual line follower works.

Both the Collision Avoidance and block detection were running perfectly in parallel without any issues. It was able to detect blocks from quite a distance away.

2.7.1 Collision Avoidance

The robot moved as it should while not hitting a single obstacle.

2.7.2 Color Detection

The actual edge detection of each color was very accurate. However, when it came to alignment of the robot with respect to the blocks positions, the adjustments were a bit too aggressive and as a result caused the robot arm to deviate slightly from the center of the block. This was attributed to the difference between the speeds of the left and right DC motors despite providing the same voltage. Furthermore, if the block is at an angle then the ultrasonic sensor cannot detect it due to the sound waves not bouncing back and hence the robot can overshoot.

Figure 13: Bounding all three color and drawing a rectangle.

2.7.3 Line Follower

As mentioned earlier the second and third scenarios were not thoroughly tested and so cannot conclude the effectiveness of the transitioning to the line following state. Although the actual line follower worked as intended and was able to make follow the line fairly smoothly.