Final Project Lab report for CISC 3060

Fall 2022

Jan C. Bierowiec

Bethany Fernandez

1.0 Objective

The objective of the final project was to create a system in which two nodes: a color tracking node, and an obstacle avoidance node would communicate with each other and accomplish a task. This task was for the robot to move towards four colored targets: orange, black, white, and orange, reach these colors avoiding obstacles, stopping and taking a screenshot of the color target recording both the time and position of the robot going from one color goal to the next. To come up with these nodes, a design was derived based on using both the tracking node and potential field navigation learned throughout the course, and then implemented. The design and implementation of these nodes will be discussed in detail followed by a result analysis using these nodes.

2.0 Design

2.1 System Design

The system design involved using two nodes, a OANode and a CTrackingNode. The OANode was the obstacle avoidance node, and the CTrackingNode was the color tracking node. The OANode used Potential Field Navigation to move the robot away from each obstacle, whereas CTrackingNode used the trackingNode as reference for the robot to sense each of the four colors and move towards each of them recording the time, and position of each color target goal.

2.2 Topics, Publishers, and Subscriptions

The topics used in the CTrackingNode were the MotionTopic, ImageTopic, and HeadingTopic. The Heading Topic is published to the image and motion topics. This allows the CTrackingNode and OANode to communicate with each other sending both the linear and angular velocity needed for the proper heading, regarding where a color target is and moving the robot towards it using the image topic, and being able to move around obstacles, along the way using the motion topic. The MotionTopic and ImageTopic were all subscribed to the HeadingTopic. The MotionTopic was used to help identify when the robot should stop moving or move around when it sees the target or obstacle. The ImageTopic was used to iterate each color target communicating with the OA node to move around obstacles and stop the robot a given distance away from the color target. The topics used in the OANode were the

MotionTopic, LaserTopic, PoseTopic, and HeadingTopic. The Motion Topic is published to the laser, pose, and heading topics. Having the motion topic as publisher allows the OA node to handle all of the movement between the two nodes, and sends the linear and angular velocities to the robot for navigation. The LaserTopic, PoseTopic, and HeadingTopic were all subscribed to the MotionTopic. The laser topic was used to determine whether or not an object is too close to an obstacle, and it was used to determine where the target goal was. The pose topic was used to identify the location of the robot in the environment it was in. The heading topic lastly was a topic shared between the color tracking node and the obstacle avoidance node which sent the linear and angular velocity to the proper heading enabling the robot to move towards the goals and avoiding obstacles along the way doing so.

2.3 How the nodes should work together

The two nodes were supposed to work together in a way that would allow the CTrackingNode and OANode to communicate with each other, sending both the linear and angular velocity needed for the proper heading. The color tracking node would iterate for each of the four colors in the world and it would identify where the color target is. Once this information was determined, this would be communicated to the obstacle avoidance node prompting the robot to move towards the color code while simultaneously avoiding the obstacles that were in the way of the robot. Once the robot reached the target, the color tracking node would be used again to record the robot's position and time reaching that specific target. Following that a new iteration would begin as the robot would move onto the next target.

2.4 Design of CTrackNode

Most of the CtrackNode was inspired to be used from the trackTargetT3.py file. Most of the pseudocode for designing this node is provided in Figure 1.

```
function trackNode ():
while rospy is not shutdown
        Show the turtlebot camera
        Get the current image shape
        Angular velocity and linear velocity are set to default values 0.3 and 0
        if m['m00'] > 0:
                delta x, angular velocity, and distance are calculated
                if target in range:
                        Text output on screenshot
                end-if
        Publish image message
end-trackNode
```

Figure 1: Pseudocode presenting the design of the CTrackingNode for color targets that the robot has to reach for every iteration.

The layout is similar to that of the trackTargetT3.py file, with some adjustments. The CTrackNode was declared to be a function and in that function, the TurtleBot Camera was called to show a camera as the robot would move, and then the imageTopic would be referenced to obtain the shape of the color target. The angular velocities and linear velocities were set to default values 0.3 and 0.0 respectively. 0.3 for the angular velocity so that the robot can rotate by default searching for a color target as opposed to standing at rest. Once these values were determined an if statement was created that would enable the robot to locate the color target, and once located, the distance from the target and angular velocity would all be calculated. Once the robot would move towards the color target, the robot would then stop in front of the the target and take a screenshot of the target recording the time it took the robot to go from its starting point to the target position, the position of the robot when it reached the target, as well as the order of reaching each target. This data would be recorded as text in the screenshot. Once this was completed, the if-else statement ended and the image message would be published. Once the color tracking node would finish, the program would go to the main function and iterate the next color target. Once the color targets would all be reached, the program would end, the robot would stop moving, and the system would shut down. The screenshots with the data would then be accessed by the user. Additionally, below is the

3.0 Implementation

3.1 Preliminary Implementation

The CtrackingNode was implemented by adjusting the original trackNode given in the trackTargetT3.py file. In regards to implementing the text on each of the screenshots, three texts were created with the content provided for each of the targets reached, and then calculations were made to update the data of the goal, position and time as the robot went from target to target. The target colors themselves were located in the main function of the program with the upper and lower bounds of the orange, yellow, black, and white colors. Once the bounds were created, the four target colors were placed in an array, with that array passed in a for loop, referencing the CTrackNode. This node was subscribed to the ImageTopic and MotionTopic and published on the HeadingTopic. This makes sense because the track code needed access to a camera in order to track the color of the targets it was looking for, and once that target was found, this linked with the MotionTopic to move the robot towards the color target while avoiding obstacles. This information would be published onto the HeadingTopic, which linked the CTrackingNode to the OANode so that nodes would be able to communicate with each other. Figure 2 provides a visual representation of the full CTrackingNode.

3.2 Variable Implementation

A function was created for the CtrackingNode and within that function, the subscribers and publishers were initialized. The message was set to be a twist message and a variable start was declared that would measure the time of the movement of the robot. A while loop was then created which showed an image for the Turtlebot robot camera. A height and width was declared to get the current size of the image detected by the robot. Following that a variable called targetImage was created which looked for a

binary target image. 0 being wherever the image was not located and 1 being where the color target was located. Another image show was then created, which allowed the robot to track color and locate it. A default angular and linear velocity were declared being set to 0.3 and 0.0 respectively. 0.3 being set for the angular velocity so that the robot would have a default rotating speed, otherwise it would remain stationary. Figure 2 below shows the python code file representing this declaration of variable process.

```
def trackNode(targetCol):
    '''center the robot camera on the target if in view'''
    global gCurrentImage

    rospy.init_node('trackNode',anonymous=True)
    # create windows to show camera and processing
    cv2.namedWindow('Turtlebot Camera')#, cv2.WINDOW_AUTOSIZE)
    cv2.namedWindow('Target')#, cv2.WINDOW_AUTOSIZE)

    imageSub = rospy.Subscriber(imageTopic,Image,callbackImage)
    heading_pub = rospy.Publisher(headingTopic, Twist, queue_size=10)
    rospy.sleep(5) # wait for callbacks to catch up

    rate = rospy.Rate(10)
    msg=Twist()
    start=rospy.get_time()

    while not rospy.is_shutdown():
        #just show what the camera sees now
        cv2.imshow('Turtlebot Camera', cv2.resize(gCurrentImage,(320,240)) )
        #get height h and width w of current image
        h,w = gCurrentImage.shape[0], gCurrentImage.shape[1]
        # make a binary image that is 0 except where the color is in range
        targetImage = cv2.inRange(gCurrentImage,targetCol[0],targetCol[1])
        cv2.imshow('Target',cv2.resize(targetImage,(320,240)) )

        # variable used to help stop loop and track how far is center
        trackCntr = 0

        # tracking algorithm
        avel=0.3 # default velocity, so robot 'spins' when no target in view
        lvel=0.0
        # extract the moments of the target image
```

Figure 2: Here is an image portraying the complete CTrackingNode program.

3.3 Variables Used when Image was Found Implementation

When the robot would rotate and locate the color target, an if statement would be used to calculate the distance of the robot from the image, as well as the target area size of the screenshot the robot would take once reaching the target position. A value delx would calculate the center of the image, and the angular velocity would be recalculated so that it would be proportional to the ratio of the image measured. A variable dist was then declared to measure the area as a fraction of the image size. The target area size was also declared as A and epi. Once these declarations were completed, an if statement was created which compared the target area size of the color target to the area as a fraction of the

image size. If the dist was greater than A and epi, then the linear velocity became -0.1. Otherwise if the dist was less than the difference of A and epi, then the linear velocity was set to 0.2. Below Figure 3 portrays the implementation of this code.

```
# extract the moments of the target image
m = cv2.moments(targetImage)
if m['m00']>0 and trackCntr > 50: # skip if the target image has non nonzero regions
    # how far is the X center of target  from X center of image
    delx = w/2 - m['m10']/m['m00']
    avel = 0.4*delx/w # use this to generate a proportional ang velocity
    dist= m['m00']/(h*w)  # area as a fraction of the image size
    A,epi=70,10 # target area size, controls how close ti get to target
    if dist>A+epi:
        lvel = -0.1
    elif dist<A-epi:
        lvel=0.2
```

Figure 3: Portraying the process in the program once the image was found

3.4 Printing Data and Exiting the Node

Once the target image is located, and the robot reaches it, the robot takes a screenshot and saves it as a jpeg. The code implementation of this is provided in Figure 4 below.

```
                else: # target now in range
                    tnum = 1 # target number

                    for i in tnum:
                        tnum = tnum + 1

                    elapsed = rospy.get_time() - start
                    txt1 = " Goal: " + str(round(tnum - 1, 2))
                    txt2 = " Loc: " + str(round(dist, 2))
                    txt3 = " Time to goal: " + str(round(elapsed, 2))
                    font = cv2.FONT_SANS_SERIF
                    black = (0, 0, 0)
                    cv2.putText(gCurrentImage, txt1, (0, 100), font, 2, black)
                    cv2.putText(gCurrentImage, txt2, (0, 150), font, 2, black)
                    cv2.putText(gCurrentImage, txt3, (0, 200), font, 2, black)

                    filename = "Goal"+str(tnum)+".jpg"

                    cv2.imwrite(filename, gCurrentImage)
                    return

                print ("offset from image center =",round(delx,2),
                       " => avel=",round(avel,2))
                print ("size of target =",round(dist,2)," =>lvel=",round(lvel,2))
                trackCntr = round(dist,2)
            msg.linear.x,msg.angular.z=lvel,avel # publish velocity
            print(msg.linear.x, " ", msg.angular.z)
            heading_pub.publish(msg)
            cv2.waitKey(1) # need to do this for CV2 windows to show up
```

Figure 4: Printing the results of the robot reaching each of the target colors

A value tnum was created to record the target number of each of the four targets. Once the robot reaches each of the targets, the goal, location, and time would be recorded and when the screenshot would be taken, this data would be recorded as text on the image. The linear velocity, angular velocity, offset from the image center, as well as the size of the target would also be printed, however on the

terminal. Once this process finished, the if statements would exit and the velocity would be published and the heading topic would publish the Twist message. Once this process would finish, the color target would iterate through the process again. Figure 5 shows an image of the main function and how the color targets are iterated using an array to store all the colors and a for loop to go through each color goal in the CTrackingNode.

```
def callback_shutdown():
    print("Shutting down")
    pub = rospy.Publisher(motionTopic, Twist, queue_size=1)
    msg = Twist()
    msg.angular.z=0.0
    msg.linear.x=0.0
    pub.publish(msg)
    rospy.sleep(5)
    return

if __name__ == '__main__':
    # identify/center the RGB color range of the target
    try:
        rospy.on_shutdown(callback_shutdown)

        # target colors
        targetColorO = [(0,30,75), (5,50,89)]           # orange target
        targetColorY = [(0,80,80), (10,110,110)]        # yellow target
        targetColorB = [(0,0,0), (20,20,20)]            # black target
        targetColorW = [(240,240,240), (255,255,255)]   # white target

        # array for colored targets
        targetColorArray = [targetColorO, targetColorY, targetColorB, targetColorW]

        # ffor loop to go through each target
        for i in targetColorArray:
            trackNode(i)

    except  rospy.ROSInterruptException:
        pass
#
```

Figure 5: Main function in the program which iterates the color targets through an array

4.0 The OANode

4.1 Design

Most of the CtrackNode was inspired to be used from the Potential Field Navigation. Most of the pseudocode for designing this node is provided in Figure 6.

```
function OA_Node ():
while rospy is running:
        wait for lin. And ang. vels from color node
        if linear velocity = 0
                spin in place to help CT look for target
```

```
        else
                smoothed_vels = potential field algorithm()
                set new linear and angular velocity to smoothed_vels
        end-if
        Publish motion message
end-while
end-OA_NOde
```

Figure 6: Pseudocode presenting the design of the OANode for obstacle avoidance when the robot locates the color targets without hitting obstacles along the way.

The OANode waits for the angular and linear velocities from the CTrackingNode, which were by default set to 0.3 and 0.0 respectively as mentioned in the implementation of the CTrackingNode. Following that, if-statements are used to determine the conditions of the linear velocity. If the linear velocity is 0, the robot will spin until the color target is located. Otherwise, when the color target is located and the robot will move towards the target calling the potential field navigation algorithm and once called, new linear and angular velocities will be created for the robot to move towards the color targets. Once this process is finished, the motion message is published and the OANode ends.

4.2 Preliminary Implementation

The OA Node was implemented for the robot to move from each colored target to the next whilst avoiding hitting the blue obstacles. The OANode was declared as a function with the MotionTopic as the publisher. The LaserTopic, PoseTopic, and HeadingTopic were all subscribers to the MotionTopic. Following that global variables were declared for the heading, pose, and laser, as well as list for obstacles. Figure 7 below shows a demonstration of this code.

```
                    norm = eucdist(field[0],field[1],0.0,0.0)
                    field = field/norm

                    return field


                def oa_node():
                    rospy.init_node('OA_Node', anonymous = True) #initialize the node

                    #Set up Publishers
                    pub = rospy.Publisher(motionTopic, Twist, queue_size = 10)

                    #Set up Subscribers
                    rospy.Subscriber(laserTopic, LaserScan, laserCallback)
                    rospy.Subscriber(poseTopic, Odometry, poseCallback)
                    rospy.Subscriber(headingTopic, Twist, headingCallback) #provided by CT node

                    rate = rospy.Rate(10)   #rate is 10 Hz

                    #Declare global variables in function
                    global gHeading, gPose, gLaserDist, gObstacleList, gLaserDist

                    prior_avel,prior_lvel= gHeading.angular.z, gHeading.linear.x

                    #Obstacle Avoidance Algorithm
                    while not rospy.is_shutdown():

                        msg = Twist()

                        if gBumperLeft or gBumperRight:
```

Figure 7: Declaration of the OANode along with the publishers, subscribers, and declaration of global

variables

4.3 Algorithm Implementation

Once the global variables subscribers, and publishers, were declared, a while loop was created for the obstacle avoidance algorithm. The first half of the algorithm can be seen in Figure 8 below, and the second half can be seen in Figure 9.

```
#Obstacle Avoidance Algorithm
while not rospy.is_shutdown():

    msg = Twist()

    if gBumperLeft or gBumperRight:
        print('Bump!')

    if prior_lvel != 0:
        target = pot_field(gPose[0],gPose[1],gPose[0]+ gLaserDist ,gPose[1] + gLaserDist, gObstacleList)
        targetTheta = math.atan2(target[1],target[0])
        delTheta =  targetTheta - gPose[2]
        mag = eucdist(target[0],target[1],0,0)

        # check for error 'wrapping around'
        if delTheta>math.pi or delTheta<-math.pi:
            if targetTheta>0:
                delTheta -= 2*math.pi
            else:
                delTheta += 2*math.pi

        # scale velocities
        avel = 1.0 *(delTheta)
        lvel = 0.3 * mag

        #smooth the velocity signal
        msg.angular.z = 0.5*(avel+prior_avel)
        msg.linear.x = 0.5*(lvel+prior_lvel)
        pub.publish(msg)
```

Figure 8:

First an if statement was created that would print a bump message if the robot would hit a wall or an obstacle. Secondly, if the prior linear velocity was not initially 0.0 but a non zero number, then a series of calculations would be made calculating the target, target angle, the angle of the image as the robot is getting closer to it, and lastly the magnitude or distance of the robot to the target. Following these calculations for these variables, an error test was set up to compare the target angle and angle of the image. Following that, the angular and linear velocity became scaled according to the angle of the image as the robot would become closer to it and the linear velocity would be multiplied by the magnitude or distance the robot is from the color target. Following that a signal for smooth velocity was created which calculated the angular and linear velocities based on the prior values of the linear and angular velocities and these values would be published as a Twist message. Once this algorithm would run, the program would sleep and a message saying when the robot reached the target would be in the terminal. Once the robot would reach its destination of the target color, the location would also be provided in the terminal. Once all of this is completed, the node would finish execution and the CTrackingNode would be communicated with in order to iterate through the next color target. This would execute until all the color targets within the color target array in the CTrackingNode would finish. Once the last color target is

located, and the last iteration of the OANode has been completed, then the two programs would finish and the results would be portrayed.

```
            # scale velocities
            avel = 1.0 *(delTheta)
            lvel = 0.3 * mag

            #smooth the velocity signal
            msg.angular.z = 0.5*(avel+prior_avel)
            msg.linear.x = 0.5*(lvel+prior_lvel)
            pub.publish(msg)

            prior_avel,prior_lvel=avel,lvel

        msg.linear.x, msg.angular.z = prior_lvel, prior_avel
        pub.publish(msg)

        rate.sleep()

        #print  ("Done ","d=",round(eucdist(goalx,goaly,gPose[0], gPose[1]),2),end="")
        #print  (" Loc= ",round(gPose[0],2), round(gPose[1],2))


    return

#Shutdown function; important for any node involving movement
def callback_shutdown():
    print("Shutting down...")
    pub = rospy.Publisher(motionTopic, Twist, queue_size = 10)
    msg = Twist() #creates ROS message that determines lin. and ang. velocities.
    msg.linear.x, msg.angular.z = 0,0 #send message to stop the motors.
    pub(msg)

    return
```

Figure 9: Second part and continuation of the obstacle avoidance algorithm

5.0 System Testing

When testing out nodes, the CTrackingNode nodes were able to recognize the different color targets based on the lower and upper bounds inputted for the colors. The program would not however take screenshots of the individual goals, the time elapsed between going from one target to another or the location of the robot upon reaching each target. Only the first screenshot for the goal was taken as the for loop within the CtrackNode did not work properly updating after each color target. The order and updating of the color target was correct, however screenshotting each one after moving towards each other was not successful. The robot was able to successfully move from one color target to the next, without hitting the obstacles. This was a success with the program.

6.0 Conclusion

In short, this system was designed with an intention to track a goal, a colored target, and for the robot tracking that goal, to be able to avoid obstacles when moving towards each goal. The robot was supposed to be able to locate these goals and obstacles accordingly so as to not hit the obstacles, and

stop in front of the target colors in a way that takes in a fraction of the image size of the camera, and being able to record the location and time of the robot as it reached each target. Despite having a design for implementing the program. The nodes were not able to communicate with each other. No results have been provided, however a general layout as to how the node and system might work has been developed. A proposal to solve the problem has been made, however the problem itself could not be solved due to a lack of communication between the nodes.