Capstone Project

AuE-8230 Autonomy Science and Systems

Group 6

AuE-8230 Autonomy Science and Systems

Group 6

1. Amogh V Reddy
2. Ninad Tarare
3. Priyanshu Rawat
4. Siddharth Thorat

Git Repo Link :

https://github.com/ntarare/AuE8230_Group_6_Repository.git

Acknowledgments

The course AuE8230 Autonomy: Science and Systems give the hands-on experience of learning, and interacting with peer teammates effectively. We would like to express our sincere gratitude for the guidance, inspiration, motivation, and encouragement of our course instructor **Dr. Venkat Narayan Krovi,** and for familiarizing us with the Piazza for the effective class communication environment which extremely solved our issues faced by everyone in hardware or software aspects easily.

We are also indebted to our teaching assistants **Ajinkya Joglekar** and **Ameya Salvi** for the reviews, and suggestions and for giving us valuable support in guiding us whenever required. This project could not have been completed without their assistance and familiarizing us with every task in the course work and assisted us from their past experiences and provided us with enough information required for the successful completion.

We would greatly thank them for their effort and assisting us in the troubleshooting area and for giving us further ideas for the successful completion of tasks in the capstone project.

Our humble thanks to **Dr. Venkat Narayan Krovi, Ajinkya Joglekar, and Ameya Salvi** for giving us a great opportunity to learn and take time in each team at various stages for our successful completion and for providing us with all the facility that was required till the end of Capstone Project.

DATE: -Group 6

05-02-2022

Table of Contents

1. List of figures…… 4
2. Introduction…… 6
   2.1 Hardware Overview……..8
   2.2 Software Overview……...9
   2.3 Overview of capstone project……..11
   2.4 Timeline Projections……....13
3. Literature Review……..15
4. Methodology……..18
5. Discussions……19
   6.1 Challenges……...20
   6.2 Communications……..21
   6.3 ROS Glossary……..22
6. Appendix……29
7. References……..38

1. List of Figures

- Figure - ROS: Nodes, Topics, and Messages
- Figure - TurtleBot 3 burger
- Figure - Raspberry Pi camera V2
- Figure - ROS architecture
- Figure - Rqt_Graph
- Figure - Overview of the GAZEBO Environment
- Figure - Gazebo world
- Figure - Tiny YOLO Demonstration
- Figure - Sift Algorithm
- Figure - GitHub Repository
- Figure - Gantt chart timeline deliverables
- Figure -Overview of a Real test track
- Figure -Different sections of a Test track
- Figure -Wall following using PID controller
- Figure - Wall following and Obstacle avoidance

2. Introduction

Nowadays, Robotic operating systems have become ubiquitous for testing new algorithms, alternative hardware configurations, and prototyping. The new research can perform sharing streamline of new work and integrations. The robotic operating system is a flexible framework for developing robotics software. The ROS was built from the ground up to encourage collaborative robotics software development.

The advantage of being an operating system for robots is that it allows supervising different parts of the robot (control boards, sensors, and actuators) regardless of the programming language that each of them uses in its implementation. The figure illustrates the publisher-subscriber communication between multiple nodes over a topic containing a particular type of message The structure of these message interfaces is defined in the message IDL

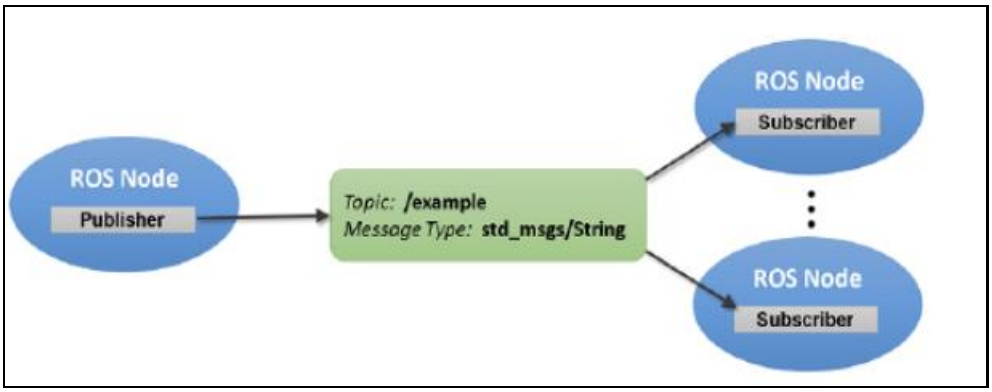


Figure (ROS: Nodes, Topics, and Messages)

The course Autonomy science and systems gave us immersed idea of ROS using a turtle Bot 3 Burger. The turtle Bot – 3 Burger has been successfully made to navigate through different tasks given in Gazebo and then in the real world by the ideology of sim2real. We used TurtleBot3 as it has a great advantage to use in robotics in the specific training research new areas.

1.1 Hardware Overview

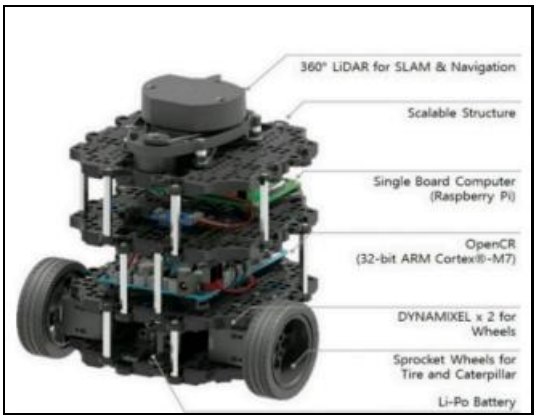


Figure (TurtleBot 3 Burger)

LiDAR sensor:

The robot uses a 360-degree LDS-01 LiDAR sensor. It is a laser scanner that is capable of collecting distance data around the robot with a range of 120 mm to 3,500 mm with an angular resolution of 1 degree, with a sampling rate of 1.8 kHz; and also, allows USB connection for computer and UART for embedded systems. It is located on the top of the robot.

Raspberry Pi development board:

The robot uses a Raspberry Pi 3 Model B development board that is located one level below the LiDAR sensor. Though it does not have more processing power, it is the key to implementing autonomous algorithms in the robot related to machine learning and digital image processing enhancing the advantage of the ability to connect directly with sensors and actuators.

Open CR hardware control board:

This board was developed to support robotic operating embedded systems and is located one level below the Raspberry Pi board. It is implemented around the STM32F7 microcontroller from STMicroelectronics which has an ARM Cortex-M7 core with a floating-point unit. Development can be done with Arduino IDE and Scratch as well as traditionally with firmware. The board contains an inertial measurement unit (IMU) with a three-axis accelerometer, a gyroscope, and a magnetometer, and allows connecting the Raspberry Pi to motors and sensors.

Raspberry Pi Camera:

The high-definition Raspberry Pi camera module v2 is used to make the TurtleBot 3 Burger navigate through the line then to detect the stop sign and finally to detect the April tags.

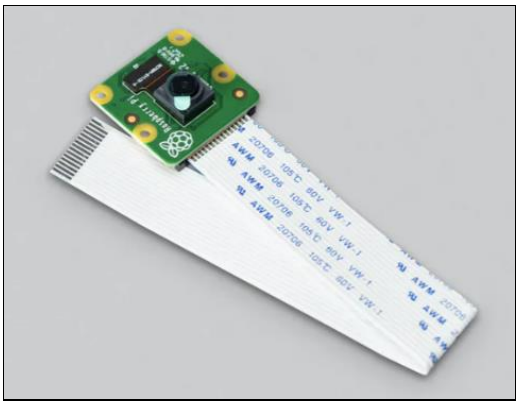


Figure 1(Raspberry Pi camera V2)

Additional Components:

Other significant elements to consider in the robot include the motors and the 11.1 V lithium-polymer (LiPo) battery. It is a differential platform with two independent Dynamixel XL430-W250 motors on each wheel.

1.2 Software Overview

The software architecture involved between the remote PC and the TurtleBot 3 burger:

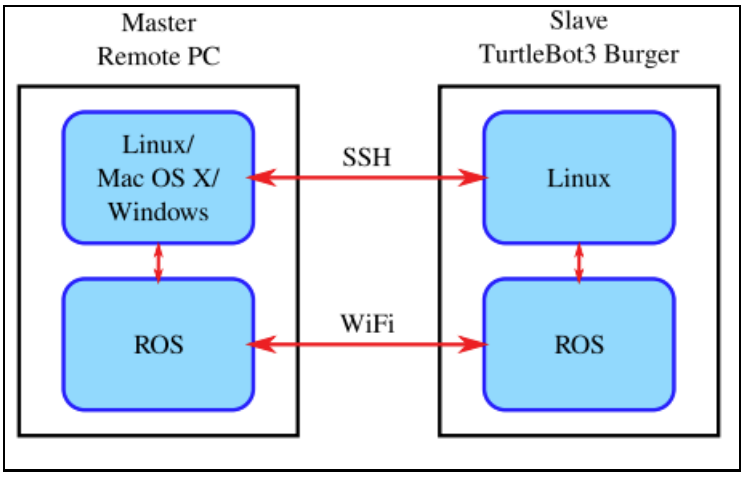


Figure 2(ROS architecture)

The ROS tools will take the advantage of introspection capability through an extensive collection of graphical and the command line which utilizes the debugging and simplifies development. The rviz and rqt the graphical tools where it provides similar functionality for the visualization interfaces. The rviz is the most well-known tool in robotic operating systems which has the visualization of the three-dimensional sensor data types and any URDF-described robot. Besides, the rqt provides a Qt-based framework for developing the graphical interfaces. The plugin of rqt provides the introspection and live visualization of robotic operating systems which shows us the nodes and the connections between them and how are they structured and even we can monitor the encoders or anything which represents as number varies over the time. They even allow us to choose the backend plotting with respective our best needs.

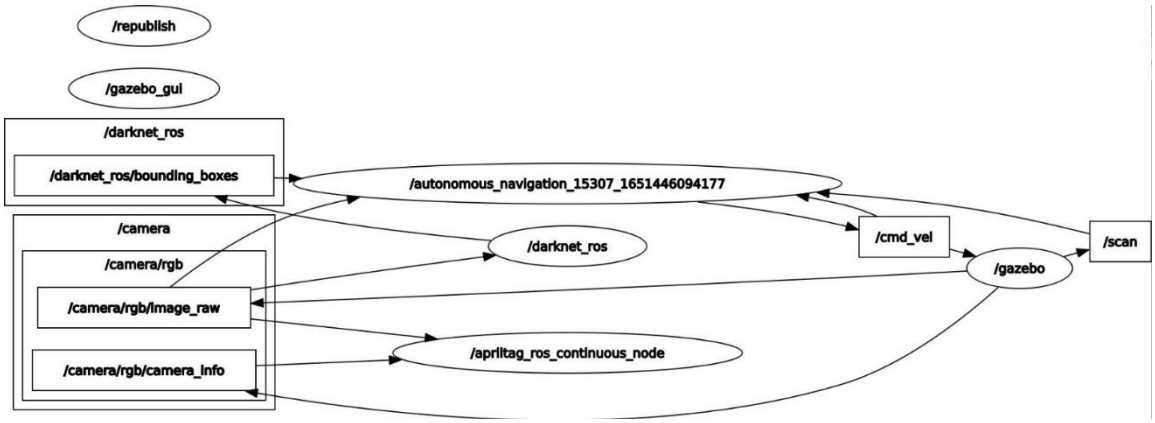


Figure 3(rqt Graph)

Controller:

They are several other techniques that can be used to control mobile robots. One of them is the open-loop control we can use. In an open-loop control system, the speed, distance, and path specifications need to be calculated before which cannot handle the noises in between. On the other hand, the closed-loop control can compensate for the errors in real-time where their inputs are based on real-time conditions. To have accurate control actions the closed-loop control is preferred over the open-loop control system. There are different types of closed-loop control systems used to control but basic in the closed control system is PID controller-Proportional Integral Derivative controller for navigating the TurtleBot3 burger through different tasks in the Gazebo simulation environment and real-time test track.

Gazebo:

A gazebo node consisting of a TurtleBot3 is executed with the help of the roslaunch command. Depending upon the command, the respective simulation environment (world) is displayed along with the TurtleBot. After this, teleoperation nodes launched that publish a stream of Twist messages (values describing the motion in 3D) on cmd_vel topic (a topic that publishes velocity values through keyboard commands). These Twist messages on the cmd_vel topic are accessed byTurtleBot3-diff-drive, a node that controls the position and speed of the robot's wheels. Figures 5 and 6 show the teleoperation simulation and the corresponding rqt_graph.

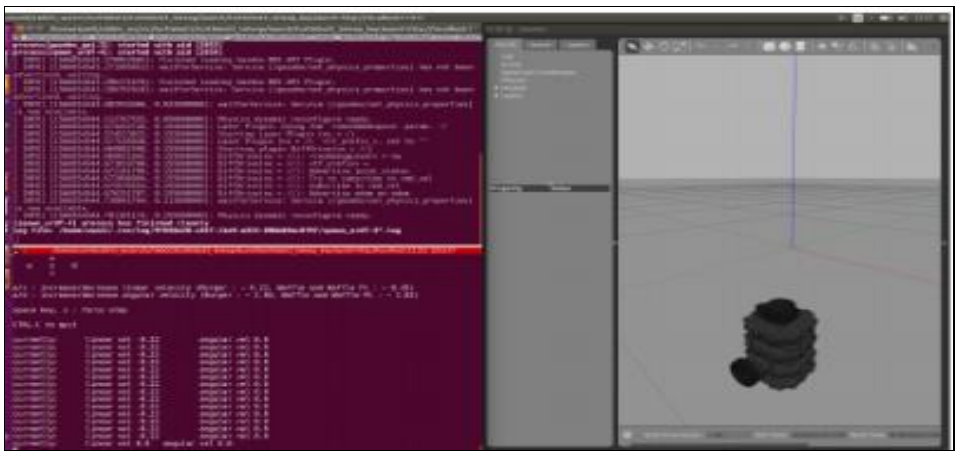
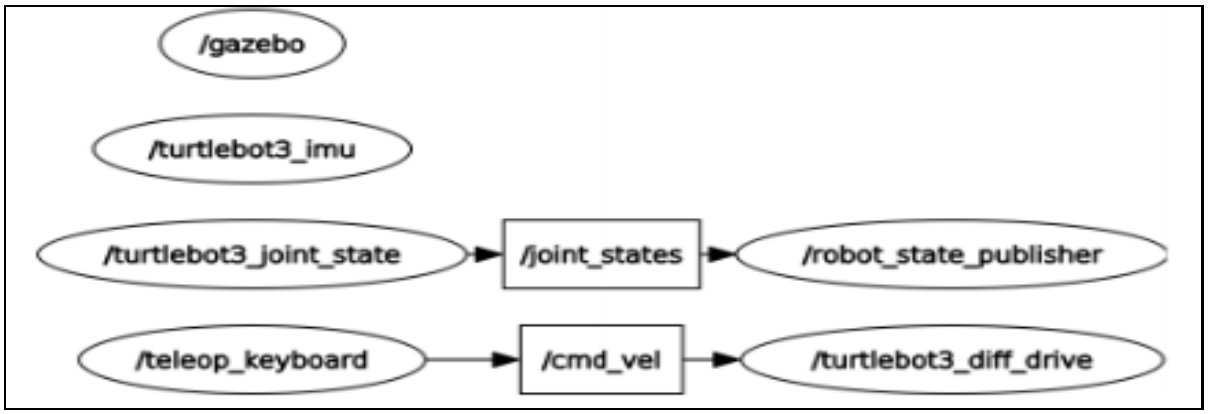


Figure 4(Overview of the GAZEBO Environment)



1.3 Overview of Capstone Project:

- The course AuE 8230 Autonomy: Science and Systems gives us hands-on experience on the gazebo and implementing all the tasks in the real world. The tasks performed in the project are - Wall following
- Obstacle Avoidance
- Line following
- Stop sign detection
- April tags detection

The project gives us the integrating challenge of switching all the tasks performed both in the gazebo and in real-time on TurtleBot 3 Burger. The report gives a detailed report of how we started, integrated, and how switched the different tasks to perform TurtleBot 3 Burger autonomously.

Gazebo Environment:

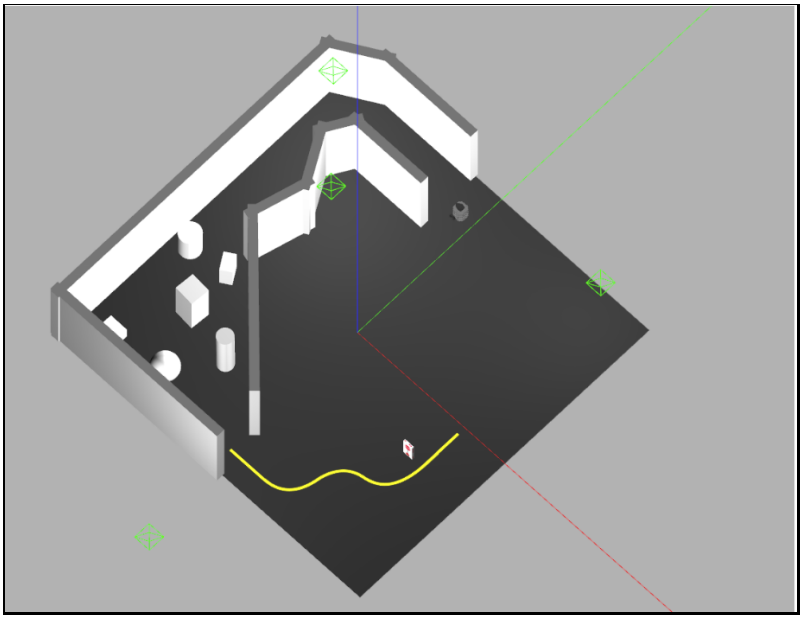


Figure 5 (Gazebo world)

In the above gazebo environment, the TurtleBot3 burger is made to navigate through the wall following the task and then avoid the obstacles placed in between. In addition, the TurtleBot3 burger should ensure that after the obstacle avoidance task it should follow the yellow line as shown in the figure. As the TurtleBot3 burger navigates through the yellow line it should detect the stop sign and should stop for a brief amount of time. Lastly, after the stop sign detection, it should detect and follow the April tag placed near the end of the yellow line.

Wall Following:

For wall following, the controller is implemented to follow the path enclosed within the side walls by extracting /scan data from the turtlebot3 LiDAR. A PID-Controller can be used to control the motion of the robot by taking /scan data as input from the left and right sides of the turtlebot3. A Kp value of 0.69 is used in conjunction with a linear velocity of .45. The launch file executes both the python script and the required gazebo environment.

Obstacle Avoidance:

To, avoid obstacles in the path of the turtlebot3 using /scan data from the LiDAR. The code file is written by separating the scan data into different sectors. The launch file executes both the python script and the required gazebo environment containing obstacles.

Line Follower:

For the task of line following, there are different methods, namely, blob tracking and hough line transform. The blob tracking method detects specific regions in the image that have similar properties such as brightness, color, etc. For the Hough line transform, the range of RGB values is given, similar to the blob tracking method, and lines are drawn on the live image using the hough line transform. This line can be followed by the robot using an optimal controller

For the task of line following, we use blob tracking. The blob tracking method detects specific regions in the image that have similar properties such as brightness, color, etc. For line following, blob tracking can be used as the lane has the same color throughout. The color is specified using HSV values, which is a scale that provides a numerical range of a specific color. As the color of the lane was yellow, we specified a range for the color as lower yellow and upper yellow denoting the range of the colors. This method also used masking to detect the blob on the specified color range, increasing the volume of data transfer between the remote PC and the bot. Thus, using the compatible input image was an important condition.

Stop Sign Detection:

The stop sign can be detected using any of the following neural networks -

a. **Yolo v3:**
   - Yolo is a real-time object detecting algorithm. It uses the features from its convolutional neural network to detect objects. The model detects objects in the image by implementing the model at different regions within an image and by the confidence scores for that region. It also draws bounding boxes for the objects within an image. This algorithm can perform at incredible rates on a GPU.

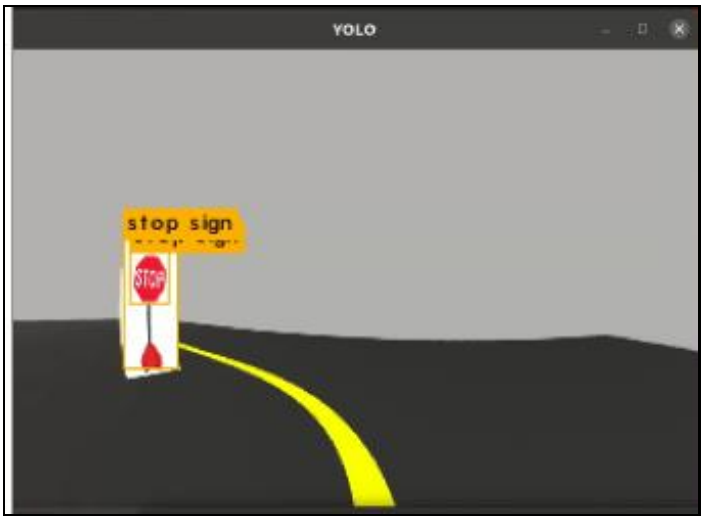


Figure 6(Tiny YOLO Demonstration)

- **Tiny Yolo:** Tiny Yolo is the same as the Yolo model but is much faster but less accurate than the regular Yolo model. It can run at extremely high speeds on a GPU but is also comparatively faster on a CPU.

- **Sift algorithm:** Sift stands for Scale Invariant Feature Transform. It is an algorithm that can be used for object detection. It uses an element called a descriptor in the image, and these descriptors are not subject to any change under various transformations, which can change the appearance of the same image.

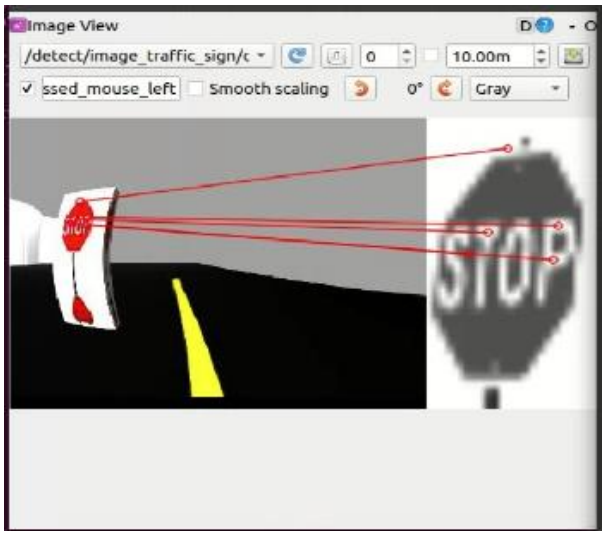


Figure 7(Sift Algorithm)

April Tag Detection:

The launch file continous_detection.launch provided in the package was used to detect the pose and id of the AprilTag. The launch file is placed in the launch folder. For calling the image proc node, we had to use the interensic and extrensic calibration launch file along with camera bring up. The tag size and family were updated in the tags.ymal file which led to smooth detection of tags Attached herewith is the rqt_graph of the Nodes acting to do so.

Next, for the April tag detection task, it was required to spawn another turtlebot under a different namespace with an April tag linked to it. Then, once the turtlebot stops at the stop sign it should follow the remaining segment of the yellow line before detecting and following the April tag.

Integration:

- After performing all the above tasks individually, the main part was to integrate all the tasks into one full python script and run it.
- For this, the LiDAR and Raspberry Pi camera must run simultaneously.

1.4 Timeline Projections:

The demonstrations were scheduled for April 28th, 2022, where we need to demonstrate the TurtleBot 3 burger performing the tasks in the gazebo environment and real-time on the test track. Even, we need to update the final programs used to navigate the TurtleBot 3 burger to perform the tasks in the group's GitHub repository "AuE8230_Group_6_Repository" with videos in it.

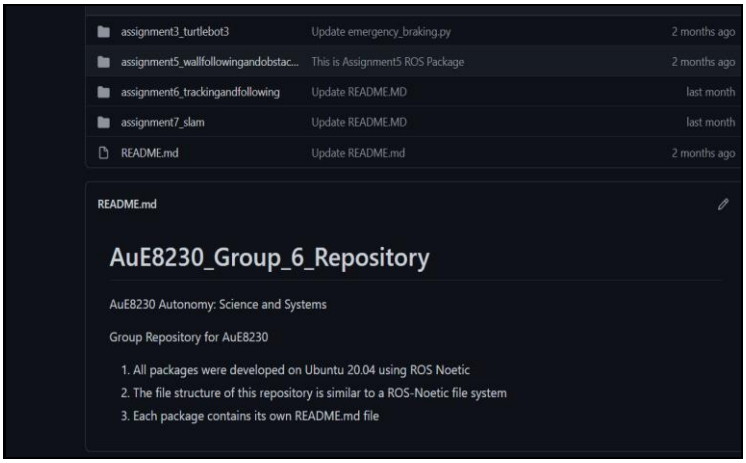


Figure 8(GitHub Repository)

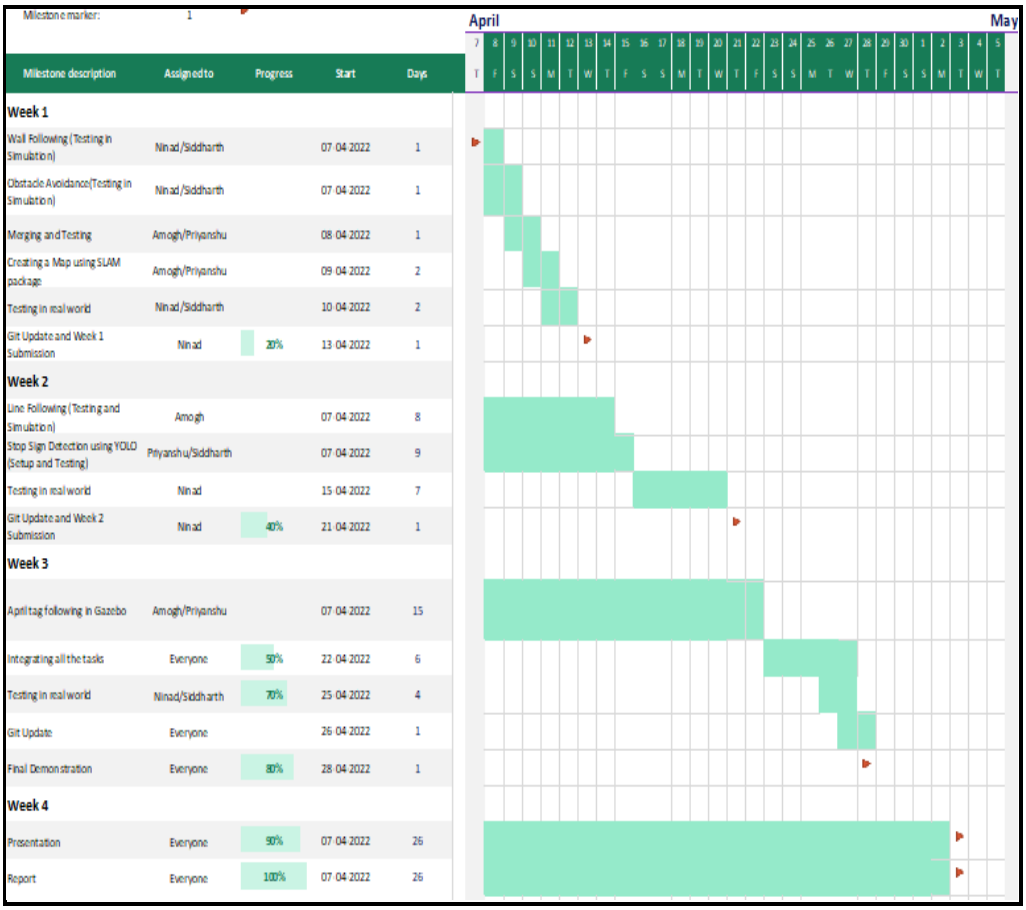


Figure 9(Gantt chart for timeline deliverables)

The above Gantt chart gives the overview of the tasks we divided among ourselves for the successful completion of the capstone project.

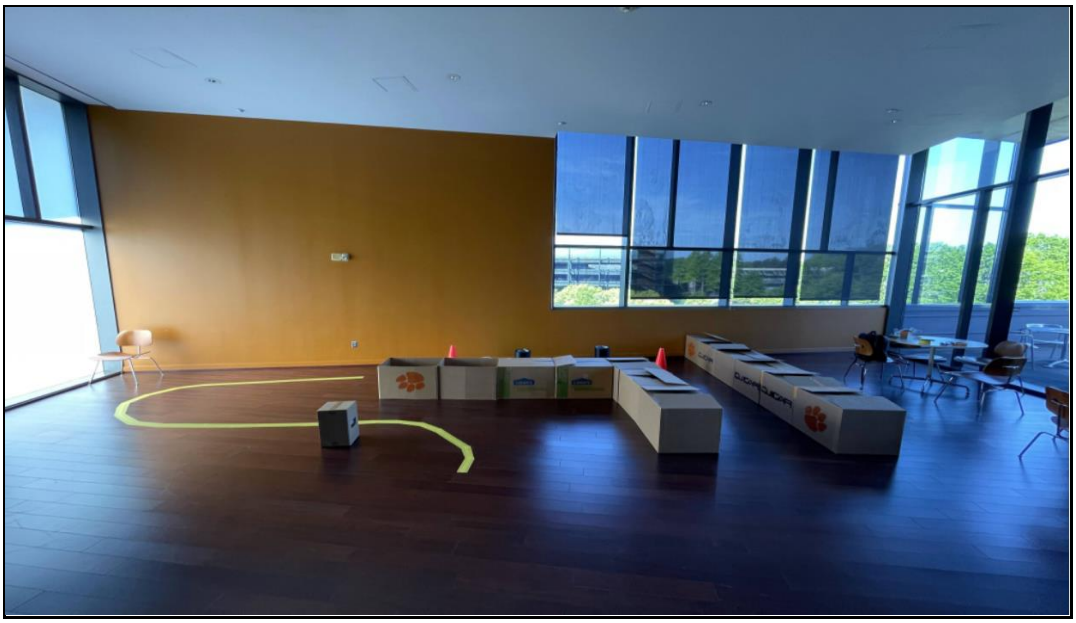


Figure 10(Overview of a Real test track)

Figure 11 (Different sections of a Test track)

The above figure represents the final test track where the TurtleBot 3 burger is made to accomplish certain tasks step-by-step till the end of April tags detection.

2.Literature Review:

Evaluation of ROS and Gazebo Simulation Environment using TurtleBot3 robot:

We have learned the usability of ROS and Gazebo for robotic applications which includes the implementation and simulating it in the real world. In addition to that, even it gives a brief explanation of rviz and rqt graphs while teleoperating the TurtleBot 3 Burger in the simulation on Gazebo. It gives a brief explanation of teleoperation implemented in the gazebo simulator and on the TurtleBot3. This report gives a brief description of the implementation of ROS-GAZEBO with an example for clear understanding along with the TurtleBot3. It even gives us an outline view of the ROS graphical tools to visualize the system. This paper even describes the importance and drawbacks of robotic operating systems in real-world applications.

TURTLEBOT 3 AS A ROBOTICS EDUCATION PLATFORM:

We learned about the different functions performed by the TurtleBot 3 which provided us with a brief idea about the GitHub with its commands. It even guides us on how to set up the Bash aliases and replace the lengthy commands with shorter versions. It even provides the functionality information about the hardware abstraction, localization, navigation, and visualization. It even covers the basic concepts involved in the robotic operating system (publisher, subscriber, nodes, topics) and configurations involved in the ubuntu software. In addition, it provides the documentation and the exercises with examples for giving us a complete idea of the topics. Here, every topic is given on a milestone basis to create great interest to learn further and get hands-on experience by easily implementing it. While completion of this report is deemed more challenging than initially estimated.

Manipulation Task Simulation using ROS and Gazebo

In this report, they give us a basic idea of the manipulation of different tasks using a robotic operating system in a gazebo environment. Even provides how to operate the gazebo and execute accurate commands to teleoperate to navigate. Also, provides information on how we can compete for the grasp and place using Gazebo virtual world and robotic operating systems. This report gives a brief explanation of ROS architecture and compatibilities. In addition to that, it explains how we need to visualize the ROS graphical tools during the communication between the nodes and allows the users to configure and modify them.

Obstacle Detection and Avoidance Using TurtleBot

This report generalizes the obstacle avoidance technique which must be simulated with TurtleBot 3 in Gazebo, ROS. It explains how much TurtleBot 3 would move in a linear direction and angular direction. While navigating, the TurtleBot 3 needs to detect the

obstacle placed in front, so this report lets us know what parameters need to be changed for the successful navigation of TurtleBot in the path. It provides brief information about what packages to be built or can be installed to run the commands to navigate the TurtleBot in the environment. In addition, it provides the different intensive testing cases of the TurtleBot and lets us know the issues while navigating the environment.

Wall following using PID controller

We focused on how to use the PID controller to give the TurtleBot3 a perfect inline motion. The wall following works on the principle of calculating both the side distances and averaging them to follow the centerline. If the bot has to calculate distances at every point and average it the error and noises would make the motion very unstable and not uniform. Hence, we use a PID controller where the proportional corrects the current value, the integral examines the past values and the derivative gives the future value-adding a correction to the errors and noises. This gives the TurtleBot a uniform and linear motion.

```
left_dist = sum(left)/len(left) # average distance of obstacles on the left
err_side = left_dist-right_dist # estimating the error for P-Controller
```

Figure 12(Wall following using PID controller)

3.Methodology

Wall following and obstacle avoidance using TurtleBot

The task of wall following was the first task to be accomplished in the project. The values of the linear and angular velocities were to be altered, mostly increased for linear and decreased for angular for optimal control in the real world. A P controller was implemented to perform wall following and obstacle avoidance. The minimum distances from the obstacles had to also be changed to be optimal for the real world.

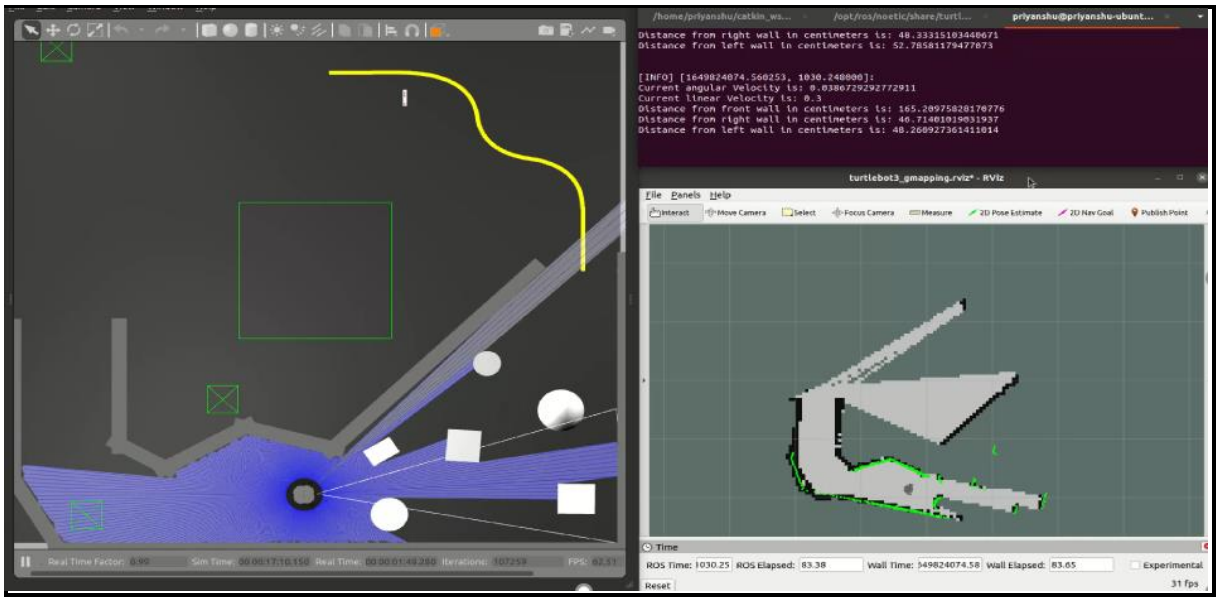


Figure 13 (Wall following and Obstacle avoidance)

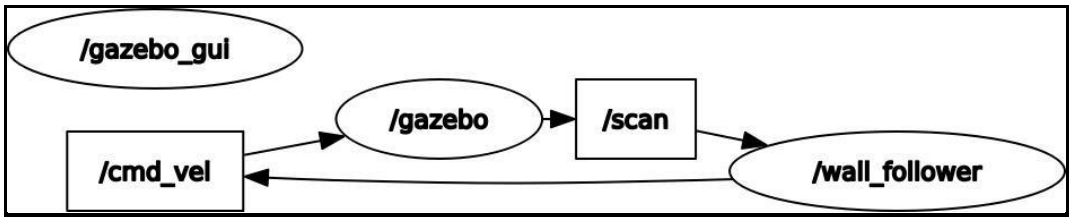


Figure 14(rqt graph for wall following and obstacle avoidance)

Line Follower using TurtleBot

- For the task of line following, we use blob tracking. The blob tracking method detects specific regions in the image that have similar properties such as brightness, color, etc. For line following, blob tracking can be used as the lane has the same color throughout. The color is specified using HSV values, which is a scale that provides a numerical range of a specific color. As the color of the lane was yellow, we specified a range for the color as lower yellow and upper yellow denoting the range of the colors. This method also used masking to detect the blob on the specified color range, increasing the volume of data transfer between the remote PC and the bot. Thus, using the compatible input image was an important condition.

Stop-sign Detection

The approaches used for stop-sign detection are as follows:

- **Yolo v3**
  Yolo v3 is a heavier model and can be effective only when it is implemented along with a GPU. We had problems in getting the GPUs to run along with the Yolo and this led us to look for other alternative methods as mentioned.

- **Sift Algorithm**
  The Sift algorithm does not require GPU and is very fast and accurate in detecting the object. But the major drawback is that this method is affected by the conditions of the environment. The brightness of the object to be detected can affect the efficiency of the algorithm. This algorithm fails to account for the dynamic nature of the surrounding environment.

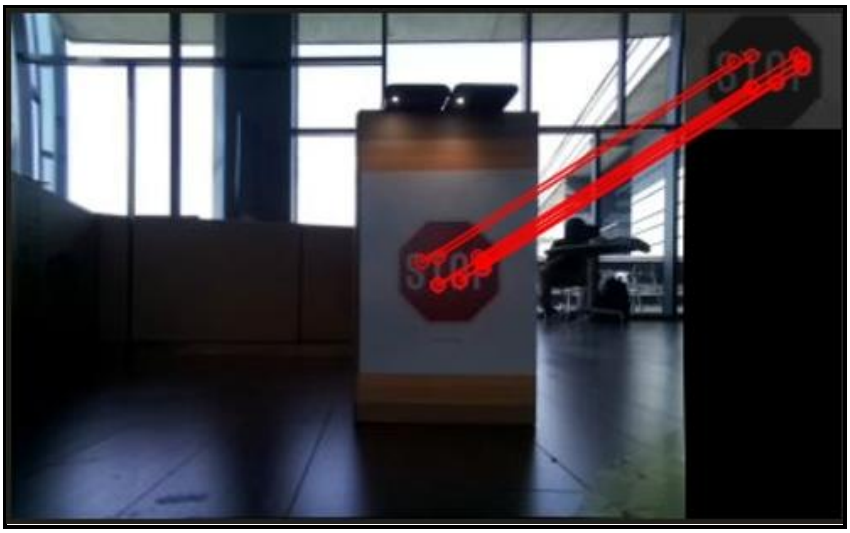


Figure 15 (SIFT)

- **Tiny YOLO**
  Tiny Yolo as said is faster and a lighter model at the cost of its accuracy. But the priority of the solution was that the model has to be lighter and faster. This could help us reduce the volume of data transfer between the remote PC and the turtlebot. Thus, we implemented the tiny Yolo approach on the CPU for sign detection.

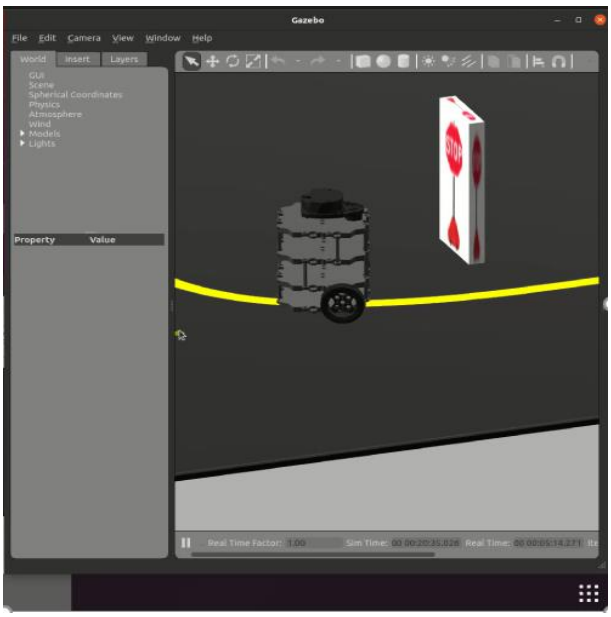


Figure 16(Stop sign detection with Line following)

April tag detection using TurtleBot

For this task, we had to spawn a different turtlebot with an April tag linked to its back. The main challenge was to launch the new turtlebot under a different namespace. For this, we had to –

- Create a urdf file for the second turtlebot with the parameters defined for the April tag.
- Create a launch file to launch both the robots under a different namespace along with the gazebo environment.

After, following the above steps we were able to launch both the TurtleBot in the gazebo environment. Visualizing the April tag in the gazebo environment was a challenge since the texture file must be added to the gazebo model file with properly defined parameters. Furthermore, to teleop the TurtleBot with the April tag, the /cmd_vel topic in the turtlebot3_teleop_key was to be changed to the velocity topic of the TurtleBot.

The

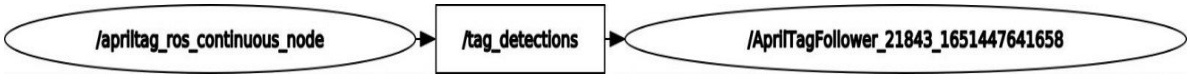


Figure 17(rqt graph for april tag detection)

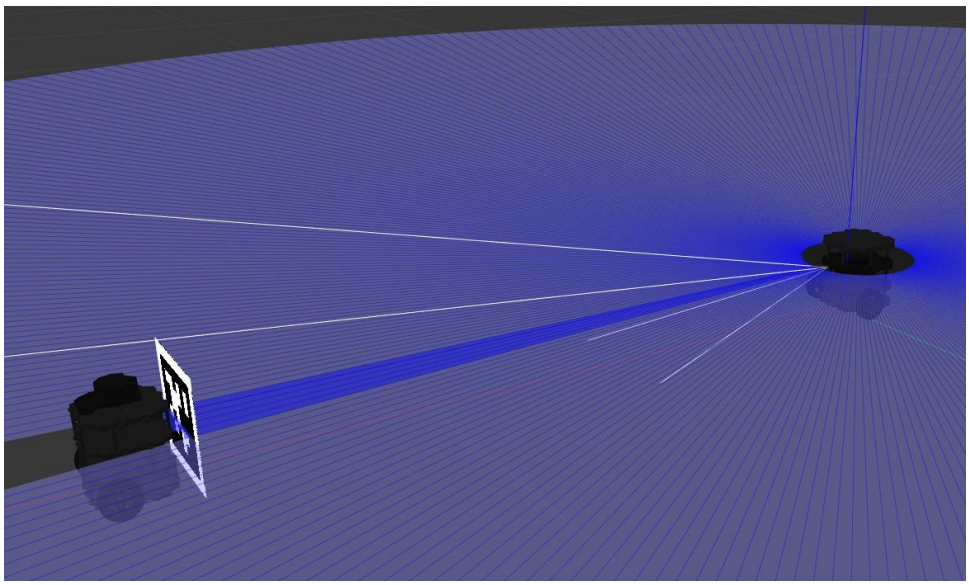


Figure 18 (april tag following in empty world in gazebo)

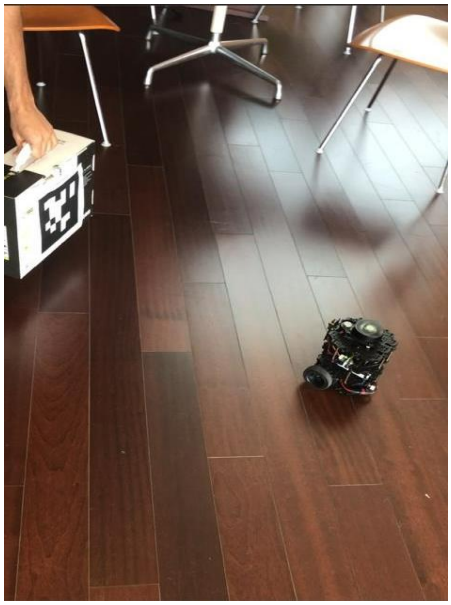


Figure 19 (april tag following in real-world)

4. INTEGRATION:

- For integrating all the tasks, we used the line following code as our base code which will keep on running all the time with switch-cases built in it to switch to different tasks.
- Firstly, we set mode=0 which will run the controller for the wall following and obstacle avoidance.

```
if self.mode==0 and self.apriltags==0:
    """ Desired angular and linear velocity """
    self.move.angular.z = np.clip(err_side*kp_side,-1.2, 1.2)
    self.move.linear.x  = np.clip(err_front*kp_front,-0.1,0.2)
```

Figure 20 (controller for wall following and obstacle avoidance)

- Secondly, once the camera detects the yellow line, the line following code finds a centroid for it after which the mode is set to 1, which will change the controller to that defined for line following.

```
try:
    cx, cy = m['m10']/m['m00'], m['m01']/m['m00']
    self.mode = 1
except ZeroDivisionError:
    cx, cy = height/2, width/2
```

Figure 21 (Switch case for line following)

```
# controller
if self.mode==1 and self.apriltags==0:
    print('line following is running!')
    err_x = cx - width/2
    twist_object = Twist()
    twist_object.linear.x = 0.08
    twist_object.angular.z = -err_x/450
```

Figure 22 (Controller for line following)

- After which a flag for stop sign was built in the code to publish zero linear and angular velocity once the yolo v3 algorithm detects a stop sign. Furthermore, logic was added such that the robot stops only once for 3 seconds.

```
def stop_callback(self, msg):
    self.stop = 0
    if msg.bounding_boxes[len(msg.bounding_boxes)- 1].id == 11:
        self.stop = 1
```

Figure 23 (Switch case for stop sign detection)

```
    if self.stop == 1:
        if self.stop_once == 1:
            print('stop sign detected!')
            rospy.sleep(3)
            twist_object.linear.x = 0
            twist_object.angular.z = 0
            self.moveTurtlebot3_object.move_robot(twist_object)
            rospy.sleep(3)
            self.stop = 0
            self.stop_once = 0
```

Figure 24 (Controller for stop sign detection)

- After this, we added a switch case for the April tag detection which prioritize the controller for the April tag following over other controllers within the code.

```
def apriltag_callback(self, tags):
    self.apriltags = len(tags.detections)
    if self.apriltags>0 and self.stop_once == 0:
        print('april tag detection is running!')
        self.x = tags.detections[0].pose.pose.pose.position.x
        self.z = tags.detections[0].pose.pose.pose.position.z
        linear_vel = 1
        angular_vel = 2
        #velocity controller
        self.move.linear.x = self.z*linear_vel #desired linear velocity
        self.move.angular.z = self.x*angular_vel #desired angular velocity
        #publishing velocity
        self.vel_pub.publish(self.move)
        # rate.sleep()
```

Figure 25 (Switch case and controller for April tag following)

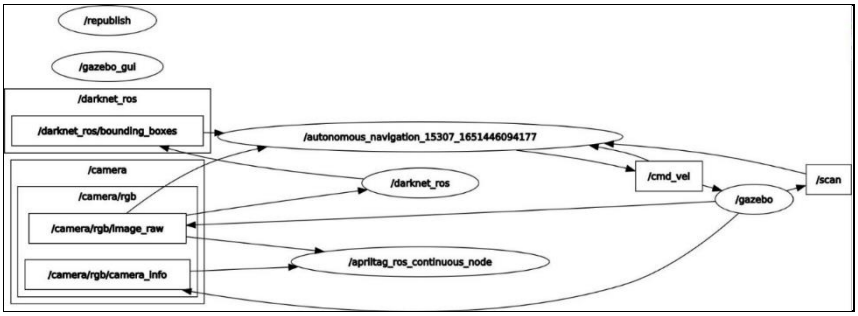
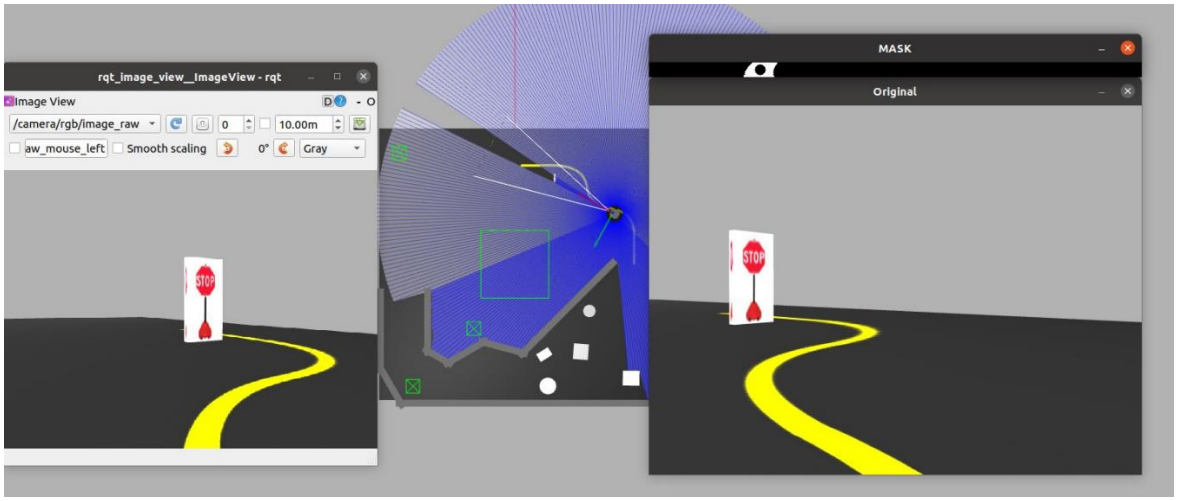


Figure 26 (rqt graph for wall following, obst



acle avoidance, line-following, and stop sign detection)

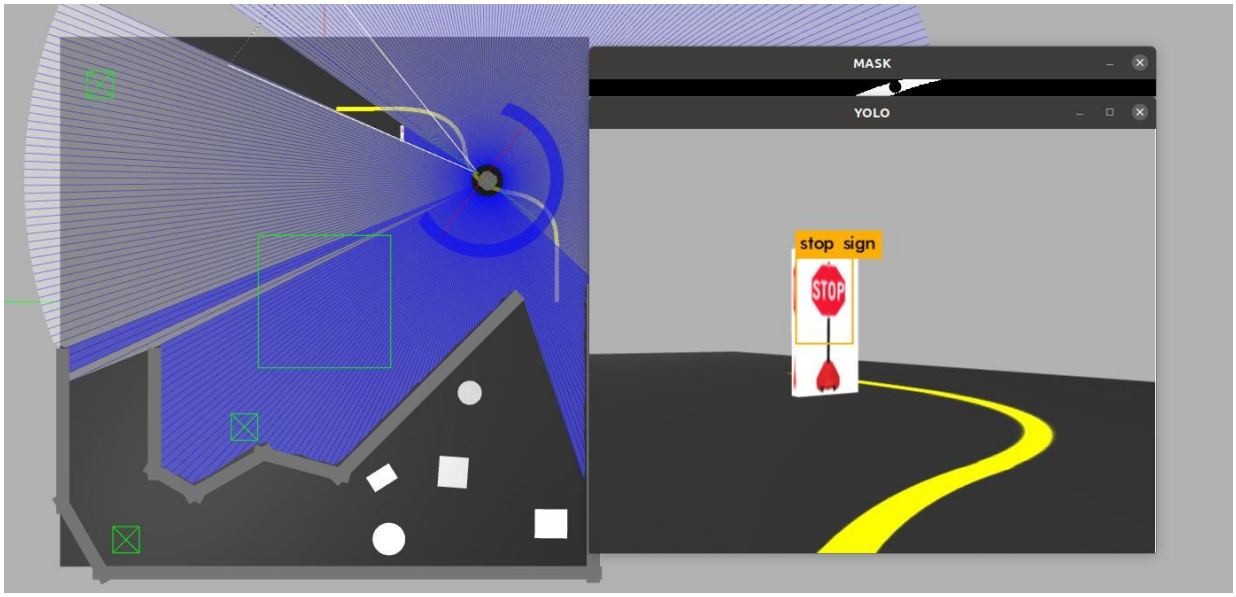


Figure 27 (Stop sign detection in the gazebo world)

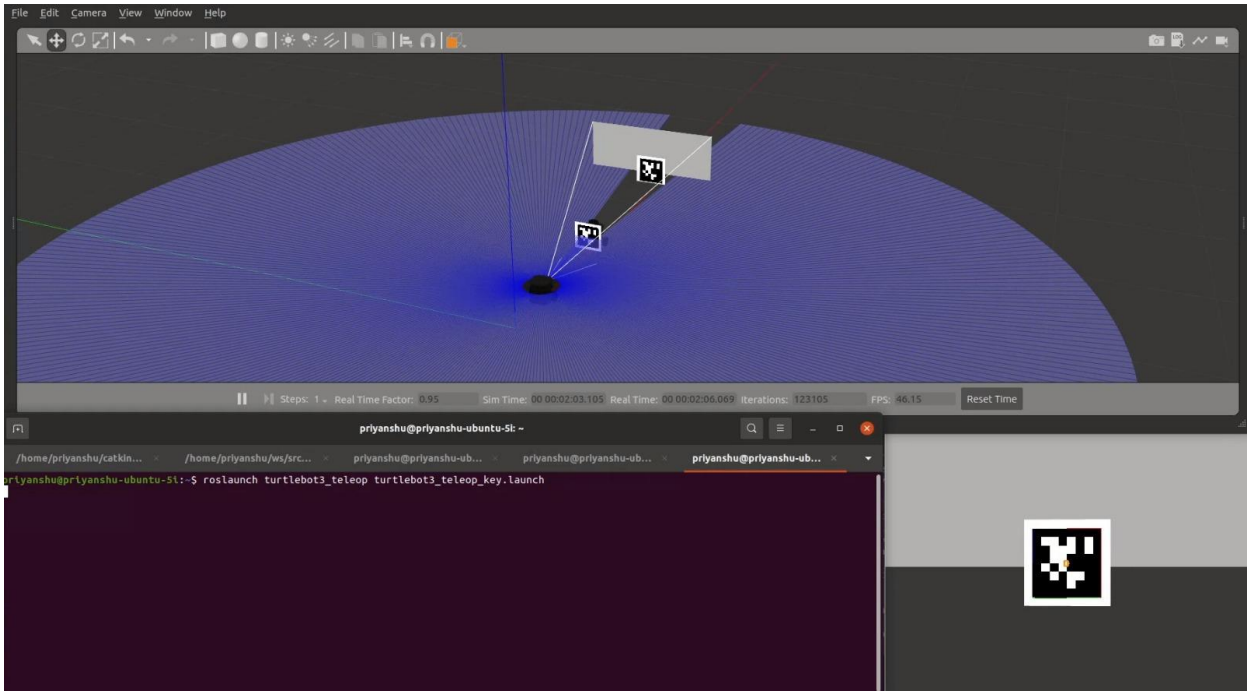


Figure 28 (April tag following in the gazebo world)

5.Discussions

5.1 Challenges:

Challenges faced during wall following task:

1. Keeping the TurtleBot in constant motion
2. The distances between the boxes were not constant and we had to implement the P controller to give the bot a linear and uniform motion
3. During the wall following the TurtleBot3 burger should detect the wall during navigating through the test track by avoid collision

Challenges faced during Obstacle detection task:

- Round obstacles were difficult to detect.
- The height of an obstacle in the gazebo was less than the height of Li-DAR on TurtleBot.
- The obstacle in the real world with varying widths and circular shapes tends to go undetected by the Li-DAR.
- There was a blind zone in the real world where the threshold set on all 3 sides of the LiDAR range was equal and the TurtleBot was stuck in a blind situation.

Challenges faced during Line following task:

- Real-world HSV values.

Challenges faced during stop sign detection task:

- The switching of the program should be accurate so that the TurtleBot3 burger can detect the stop sign and stop for at least three seconds after completion of the line following task.
- The Raspberry Pi camera is mounted at a certain height so that it can detect the stop sign.
- For the installation of the YOLO package, the system should have a good GPU to install all the packages accurately.

Challenges faced during the April tag detection task:

- Replicating the texture of the April tag in the gazebo

Challenges faced during the integration of tasks:

- There are too many nodes running parallelly on a single topic such as the camera topic.
- The flags for each module switch in an instance, running their respective logic and instructions.

Challenges faced while shift to real world:

- Firstly, we had to adjust the threshold distances according to real world feasible values
- We also had to tune the linear and angular speed parameters according to real world values

- We finally had to tune HSV values based on actual test timing on the day of presentation
- We also had to setup wireless network such that the communication between the TurtleBot and Remote Pc is seamless and fast

5.2 Communications

Communication is the exchange of vital ideas with other people. Our team communication is how we define our team members to interact with each other and accepted all social behaviors of a team (expressing ideas, concerns, resolving conflicts) and master group communications to collaborate essentially and effectively.

The piazza communication is the common platform provided which helped us to discuss with the other teammates and this platform allowed the instructors and teaching assistants to guide us in the issues we faced during the project essentially and effectively.

The WhatsApp group was created for effective communication which helped to solve the tasks and resolved the issues faced whenever required in the given time for the completion of different tasks

5.3 Glossary of ROS:

Nodes:

- Nodes are processes (executable programs) that perform computation, such as calculations, processing, and specific functions (reading a laser, for example).

Master:

- The Master deals with name registration, publication, subscription, and so on. It also includes a parameter server that allows them to be saved centrally. Without the Master, nodes would not be able to find each other, exchange messages, or invoke services.

Messages:

- A message is simply a data structure. Nodes communicate with each other through messages.

Topics:

- A topic is a name used to identify a message. The topics can be published and subscribed to.

Bags:

- Bags are a format for storing data records in a file that can later be played. For example, saving data from sensors that can then be reproduced in later simulation

6.Conclusions

The Capstone project concludes the course, “Autonomy: Science and systems” for the spring semester. This course has helped us learn a lot of new concepts in the field of autonomy and implementing them on a turtlebot3 was an experience of another level.

This project gave us an understanding of the differences between the simulation and real-world problems that we will encounter when working on similar problems in the future. It also gives an idea about how integrating modules can become a tedious and tough task if it is not planned and executed effectively.

There is a vast ocean of knowledge and topics that are left to learn on the table. This project has just given a glimpse of the things that are yet to be learned and mastered. This report just serves as the problems encountered and tasks accomplished over the course of this semester.

7. Appendix

Wall Following and Obstacle Avoidance

```
#!/usr/bin/env python3

import rospy
import numpy as np
from geometry_msgs.msg import Twist
from sensor_msgs.msg import LaserScan

def PID_side(err_side, Kp_side=2.5): #defing a function for P-Controller
    return Kp_side*err_side

def PID_front(err_front, Kp_front=1): #defing a function for P-Controller
    return Kp_front*err_front

def wallfollow(data): # defining a wall following function

    data = list(data.ranges[0:360]) # storing LiDAR data

    """ For lateral control """

    scan = [data[i] for i in range(len(data)) if data[i]<8] # taking only the values less than '8'

    right = scan[-90:-16]
    right_dist = sum(right)/len(right) # average distance of obstacles on the right
    left = scan[16:90]
    left_dist = sum(left)/len(left) # average distance of obstacles on the left

    err_side = left_dist-right_dist # estimating the error for P-Controller

    """ For longitudnal control """

    front_dist = min(min(scan[0:5], scan[(len(scan)-5):len(scan)])) # front distance

    err_front = front_dist-0.2 # setting desired distance to be 0.2 for sim -- 0.35 for real-world

    """ Desired angular and linear velocity """

    move.angular.z = np.clip(PID_side(err_side),-1.5,1.5)
    move.linear.x   = np.clip(PID_front(err_front),-0.1,0.3) # max linear vel to 0.3 for sim -- 0.4 for real-
world

    if move.linear.x < 0.01:
        move.angular.z = -0.3

    rospy.loginfo('')
    print("Current angular Velocity is: %s" % move.angular.z)
    print("Current linear Velocity is: %s" % move.linear.x)
    print("Distance from front wall in centimeters is: %s" % (front_dist*100))
```

```
    print("Distance from right wall in centimeters is: %s" % (right_dist*100))
    print("Distance from left wall in centimeters is: %s" % (left_dist*100))
    print('\n')

rospy.init_node('wall_follower')
move = Twist()
pub = rospy.Publisher("/cmd_vel",Twist, queue_size=10)
sub = rospy.Subscriber("/scan",LaserScan, wallfollow)
while not rospy.is_shutdown():
    pub.publish(move)
    pass
rospy.spin()
```

Line Follower

```
#!/usr/bin/env python3
import rospy
import cv2
import numpy as np
from cv_bridge import CvBridge, CvBridgeError
from geometry_msgs.msg import Twist
from sensor_msgs.msg import Image
from move_robot import MoveTurtlebot3

class LineFollower(object):

    def __init__(self):
        self.bridge_object = CvBridge()
        self.image_sub = rospy.Subscriber("/camera/rgb/image_raw", Image,self.camera_callback)
        self.moveTurtlebot3_object = MoveTurtlebot3()

    def camera_callback(self, data):
        # We select bgr8 because its the OpneCV encoding by default
        cv_image = self.bridge_object.imgmsg_to_cv2(data, desired_encoding="bgr8")

        # We get image dimensions and crop the parts of the image we dont need
        height, width, channels = cv_image.shape
        crop_img = cv_image[int((height/2)+100):int((height/2)+120)][1:int(width)]
        #crop_img = cv_image[340:360][1:640]

        # Convert from RGB to HSV
        hsv = cv2.cvtColor(crop_img, cv2.COLOR_BGR2HSV)

        # Define the Yellow Colour in HSV

        """
        To know which color to track in HSV use ColorZilla to get the color registered by the camera in BGR and convert to HSV.
        """
        #hsv_yellow = cv.cvtColor(yellow, cv2.COLOR_BGR2HSV)

        # Threshold the HSV image to get only yellow colors
        lower_yellow = np.array([20,100,100])#([0,0,200])
        upper_yellow = np.array([130,255,255])#([50,255,255])
        mask = cv2.inRange(hsv, lower_yellow, upper_yellow)

        # Calculate centroid of the blob of binary image using ImageMoments
        m = cv2.moments(mask, False)

        try:
            cx, cy = m['m10']/m['m00'], m['m01']/m['m00']
        except ZeroDivisionError:
            cx, cy = height/2, width/2
```

```
        # Draw the centroid in the resultut image
        # cv2.circle(img, center, radius, color[, thickness[, lineType[, shift]]])
        cv2.circle(mask,(int(cx), int(cy)), 10,(0,0,255),-1)
        cv2.imshow("Original", cv_image)
        cv2.imshow("MASK", mask)
        cv2.waitKey(1)

        # controller
        err_x = cx - width/2
        twist_object = Twist()
        twist_object.linear.x = 0.1
        twist_object.angular.z = -err_x/1500

        rospy.loginfo("ANGULAR VALUE SENT===>"+str(twist_object.angular.z))
        # Make it start turning
        self.moveTurtlebot3_object.move_robot(twist_object)

    def clean_up(self):
        self.moveTurtlebot3_object.clean_class()
        cv2.destroyAllWindows()

def main():
    rospy.init_node('line_following_node', anonymous=True)
    line_follower_object = LineFollower()
    rate = rospy.Rate(5)
    ctrl_c = False
    def shutdownhook():
        # Works better than rospy.is_shutdown()
        line_follower_object.clean_up()
        rospy.loginfo("Shutdown time!")
        ctrl_c = True
    rospy.on_shutdown(shutdownhook)
    while not ctrl_c:
        rate.sleep()

if __name__ == '__main__':
        main()
```

April Tag:

```
#!/usr/bin/env python3

import rospy
from geometry_msgs.msg import Twist
from apriltag_ros.msg import AprilTagDetectionArray

class AprilTagFollower():

    def __init__(self):
        self.x=0
        self.z=0
        #initializing a node
        rospy.init_node('AprilTagFollower',anonymous=True)
        #defining publisher
        self.pub = rospy.Publisher('/cmd_vel',Twist,queue_size=10)
        #defining subscriber
        self.sub = rospy.Subscriber('/tag_detections',AprilTagDetectionArray,self.pose)
        #subscriber -Apriltagdetection array callback function that stores the tag detection array
coordinates in xyz world frame
        self.rate = rospy.Rate(10)

    def pose(self,data):
        #extracting pose of pril tag
        self.x = data.detections[0].pose.pose.pose.position.x
        self.z = data.detections[0].pose.pose.pose.position.z

    def pril_tag_follower(self):
            self.move = Twist()
            while not rospy.is_shutdown():
                    k_linear_vel = 0.2
                    k_angular_vel = 2
                    #velocity controller
                    self.vel_msg.linear.x =self.z*k_linear_vel #desired linear velocity
                    self.vel_msg.angular.z = -self.x*k_angular_vel #desired angular velocity
                    #publishing velocity
                    self.pub.publish(self.vel_msg)
                    self.rate.sleep()

if __name__=='__main__':
        tagfollow = AprilTagFollower()
        tagfollow.april_tag_follower()
```

Integrated Script

```
#!/usr/bin/env python3
from email.errors import ObsoleteHeaderDefect
from statistics import mode
import rospy
import cv2
import numpy as np
from cv_bridge import CvBridge, CvBridgeError
from geometry_msgs.msg import Twist
from sensor_msgs.msg import Image
from sensor_msgs.msg import LaserScan
from move_robot import MoveTurtlebot3
from darknet_ros_msgs.msg import BoundingBoxes
from apriltag_ros.msg import AprilTagDetectionArray

class LineFollower(object):
    def __init__(self):
        self.flag = 0
        self.mode = 0
        self.stop = 0
        self.auto_nav_object = 0
        self.stop_once = 1
        self.apriltags = 0
        self.move = Twist()
        self.bridge_object = CvBridge()
        self.image_sub =
rospy.Subscriber("/camera/image",Image,self.camera_callback)
        self.stop_sign_subscriber = rospy.Subscriber('/darknet_ros/bounding_boxes' ,
BoundingBoxes, self.stop_callback)
        self.vel_pub = rospy.Publisher("/cmd_vel",Twist, queue_size=10)
        self.laser_sub = rospy.Subscriber("/scan",LaserScan,
self.obstacle_avoidance_callback)
        self.tag_sub = rospy.Subscriber('/tag_detections', AprilTagDetectionArray,
self.apriltag_callback)
        self.moveTurtlebot3_object = MoveTurtlebot3()

    def stop_callback(self, msg):
        self.stop = 0
        if msg.bounding_boxes[len(msg.bounding_boxes)- 1].id == 11:
            self.stop = 1
```

```
def camera_callback(self, data):
    # We select bgr8 because its the OpneCV encoding by default
    cv_image = self.bridge_object.imgmsg_to_cv2(data, desired_encoding="bgr8")

    # We get image dimensions and crop the parts of the image we dont need
    height, width, channels = cv_image.shape
    crop_img = cv_image[int((height/2)+100):int((height/2)+120)][1:int(width)]

    # Convert from RGB to HSV
    hsv = cv2.cvtColor(crop_img, cv2.COLOR_BGR2HSV)

    # Threshold the HSV image to get only yellow colors
    lower_yellow = np.array([20,100,100])
    upper_yellow = np.array([70,255,255])
    mask = cv2.inRange(hsv, lower_yellow, upper_yellow)

    # Calculate centroid of the blob of binary image using ImageMoments
    m = cv2.moments(mask, False)

    try:
        cx, cy = m['m10']/m['m00'], m['m01']/m['m00']
        self.mode = 1
    except ZeroDivisionError:
        cx, cy = height/2, width/2

    # Draw the centroid in the resultant image
    # cv2.circle(img, center, radius, color[, thickness[, lineType[, shift]]])
    cv2.circle(mask,(int(cx), int(cy)), 7, (0,0,255),-1)
    cv2.imshow("Original", cv_image)
    cv2.imshow("MASK", mask)
    cv2.waitKey(1)

    # controller
    if self.mode==1 and self.apriltags==0:
        print('line following is running!')
        err_x = cx - width/2
        twist_object = Twist()
        twist_object.linear.x = 0.08
        twist_object.angular.z = -err_x/450
        if self.stop == 1:
```

```
            if self.stop_once == 1:
                print('stop sign detected!')
                rospy.sleep(3)
                twist_object.linear.x = 0
                twist_object.angular.z = 0
                self.moveTurtlebot3_object.move_robot(twist_object)
                rospy.sleep(3)
                self.stop = 0
                self.stop_once = 0

            # Make it start turning
            self.moveTurtlebot3_object.move_robot(twist_object)

    def clean_up(self):
        self.moveTurtlebot3_object.clean_class()
        cv2.destroyAllWindows()

    def obstacle_avoidance_callback(self, laserscan): # defining a wall following
function

        """ For lateral control """
        laserscan = list(laserscan.ranges[0:360]) # storing LiDAR data

        right = laserscan[-90:-20]
        right_dist = sum(right)/len(right) # average distance of obstacles on the right
        left = laserscan[20:90]
        left_dist = sum(left)/len(left) # average distance of obstacles on the left
        err_side = left_dist-right_dist # estimating the error for P-Controller

        """ For longitudnal control """

        # front_dist = min(min(i for i in laserscan[(len(laserscan)-20):len(laserscan)] if
i>0), min(i for i in laserscan[0:20] if i>0)) # front distance
        # if len(front_dist) == 0:
        #     front_dist = 5

        front_dist = []
        front_right = laserscan[-20:]
        front_left = laserscan[0:20]
        if(len(front_left)>0 and len(front_right)>0):
            for i in range(0,len(front_left)):
```

```
            if front_left[i] != 0:
                front_dist.append(front_left[i])
        for i in range(0,len(front_right)):
            if front_right[i] != 0:
                front_dist.append(front_right[i])

        front_dist = min(front_dist)

    err_front = front_dist-0.4 # setting desired distance to be 0.2 for sim -- 0.35 for real-world
    kp_side = 1
    kp_front = 0.5

    if self.mode==0 and self.apriltags==0:
        """ Desired angular and linear velocity """
        self.move.angular.z = np.clip(err_side*kp_side,-1.2, 1.2)
        self.move.linear.x  = np.clip(err_front*kp_front,-0.1,0.2) # max linear vel to 0.3 for sim -- 0.4 for real-world

        print('obstacle avoidance is running!')
        # if move.linear.x < 0.01 and move.linear.x > 0:
        #     move.angular.z = 0.3
        self.vel_pub.publish(self.move)

def apriltag_callback(self, tags):
    self.apriltags = len(tags.detections)
    if self.apriltags>0:
        print('april tag detection is running!')
        self.x = tags.detections[0].pose.pose.pose.position.x
        self.z = tags.detections[0].pose.pose.pose.position.z
        linear_vel = 0.07
        angular_vel = 1.2
        #velocity controller
        self.move.linear.x = self.z*linear_vel #desired linear velocity
        self.move.angular.z = -self.x*angular_vel #desired angular velocity
        #publishing velocity
        self.vel_pub.publish(self.move)
        # rate.sleep()

if __name__ == '__main__':
    rospy.init_node('autonomous_navigation', anonymous=True)
```

```
rate = rospy.Rate(5)
auto_nav_object = LineFollower()
ctrl_c = False
def shutdownhook():
    auto_nav_object.clean_up()
    rospy.loginfo("shutdown time!")
    ctrl_c = True
rospy.on_shutdown(shutdownhook)
while not ctrl_c:
    rate.sleep()
```

References

- (PDF) Evaluation of ROS and Gazebo Simulation Environment using TurtleBot3 robot (researchgate.net)
- TurtleBot3 (robotis.com)
- Documentation - ROS Wiki