Capstone Project

AuE-8230 - Autonomy: Science and Systems

Tanmay Vilas Samak and Chinmay Vilas Samak

Spring 2023

This work is an outcome of the course AuE-8230 "Autonomy: Science and Systems" (Spring 2023) at Clemson University International Center for Automotive Research (CU-ICAR) handled by Dr. Venkat Krovi, Michelin Endowed SmartState Chair Professor of Vehicle Automation at CU-ICAR.

This report was submitted to the Department of Automotive Engineering, CU-ICAR as a part of the course AuE-8230 "Autonomy: Science and Systems".

DISCLAIMER

I certify that all the work and writing that I contributed to here is my own and not acquired from external sources. I have cited sources appropriately and paraphrased correctly. I have not shared my writing with other students (for individual assignments) and other students outside my group (for group assignments), nor have I acquired any written portion of this document from past or present students.

Authors

SUMMARY

This document outlines a capstone project completed by a group of students studying the course AuE-8230 “Autonomy: Science and Systems” at the Clemson University International Center for Automotive Research (CU-ICAR). The project involved designing and implementing an autonomous system (autonomous mobile robot – the TurtleBot3) capable of completing a set of tasks in simulated as well as real-world conditions.

The project involved several different tasks, including wall following, obstacle avoidance, line following, stop sign detection and AprilTag marker tracking. The group used Robot Operating System 2 (ROS 2) framework and Gazebo simulator to develop and test their algorithms in simulation, which they also successfully transferred to reality (sim2real) using the TurtleBot3 robot hardware. They also utilized various methods to optimize and robustify their system's performance.

The autonomy pipeline was divided into three main components: perception, planning and control. The perception component involved using monocular camera and/or 2D planar LIDAR sensor(s) to gather information about the environment, such as identifying cues and detecting obstacles, lines, signs, markers and other objects. The planning component involved using an amalgamation of traditional heuristic approaches and modern machine learning algorithms to analyze the gathered data and make decisions about what actions the system should take. The control component involved implementing the necessary software to act on the generated high level behavior decisions or plans to control the robot's movement.

The document provides a detailed overview of the project's design, implementation, and results, including challenges encountered and lessons learned. The group was able to successfully complete all of the tasks within the given time frame and demonstrated the effectiveness of their autonomous system. Overall, the project showcases the potential of autonomous systems in various applications and highlights the importance of multidisciplinary collaboration in their development.

ACKNOWLEDGEMENT

At the commencement of this report, we would like to add a few words of appreciation for all those who have been a part of this project; directly or indirectly.

We would like to express our immense gratitude towards Department of Automotive Engineering, Clemson University International Center for Automotive Research (CU-ICAR) for offering this course and for providing us with an excellent atmosphere for working on this project.

Our sincere thanks also go to the Automation, Robotics and Mechatronics Laboratory (ARMLab), CU-ICAR for providing all facilities and support to meet our project requirements.

We would also like to express our deep and sincere gratitude to our faculty Dr. Venkat Krovi and teaching assistants Dhruv Mehta and Sumedh Sathe for their valuable guidance, consistent encouragement and timely help. We are also thankful of our peers in the course, who engaged in thoughtful conversations and discussions, which led to cross-pollination of ideas.

Finally, we are grateful to all the sources of information without which this project would be incomplete. It is due to their efforts and research that our report is more accurate and convincing.

Authors

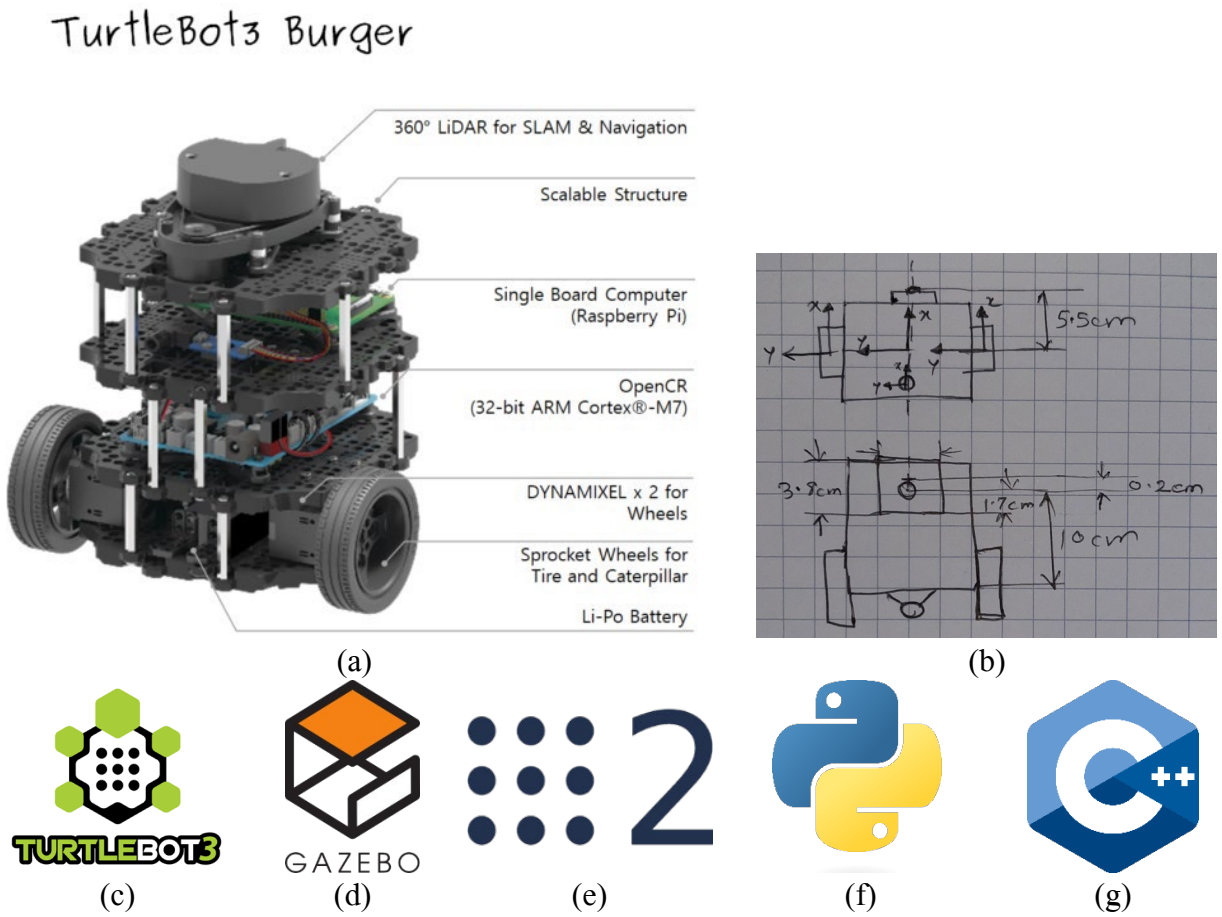
TABLE OF CONTENTS

1. INTRODUCTION

1.1. Project Tools

Following is a brief summary of the hardware and software tools (refer **Figure 1**) that went into designing and implementing the autonomous system for this project:

- **Robot:** TurtleBot3 Burger (with camera)
- **Simulation:** Gazebo
- **Framework:** ROS 2 Foxy
- **Programming:** Python, C++

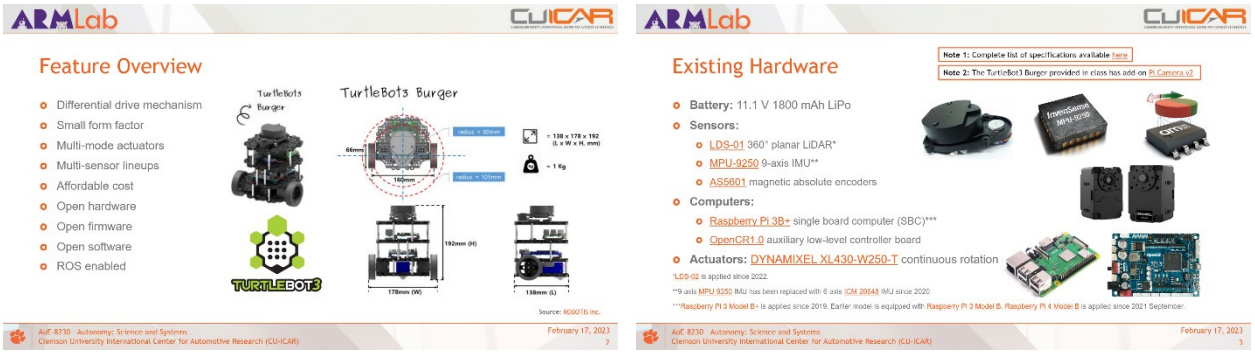


(a) (b) (c) (d) (e) (f) (g)

Figure 1. *Project tools: (a) Turtlebot3 Burger robot hardware; (b) camera modification and transform w.r.t. the robot base link; (c) TurtleBot3 software packages; (d) Gazebo simulator; (e) ROS 2 framework; (f) Python programming language; and (g) C++ programming language.*

The robot used for the project was a baseline TurtleBot3 Burger (refer **Figure 2**) with a custom mounted monocular camera for enabling visual perception. This piece of equipment is a differentially-driven (with supporting caster wheel) mobile robot platform designed for education, research, and hobby purposes. It is a small, affordable, and customizable robot that can be programmed to perform various tasks. The robot is based on the ROS (Robot Operating System) framework and can be controlled using a laptop or a mobile device. We have documented the details of different versions of the robot, hardware and software components as well as potential replacements of certain components here.

For simulation-based development and testing, we made use of the native simulator shipped with the ROS framework called Gazebo. It is an open-source 3D simulation environment for robotics and autonomous systems. It provides a platform for testing and prototyping robots in a virtual environment before deploying them in the real world. Although it has its limitations in terms of simulation fidelity (both physics as well as graphics), it served to be a baseline testing tool for rapid testing and optimization of autonomy algorithms.



(a) (b)

Figure 2. *TurtleBot3 Burger: (a) feature overview; and (b) hardware components. Further details are available here.*

In terms of the middleware framework used for development and deployment of the autonomy algorithms for the project (and assignments) our team exploited Robot Operating System 2 (ROS 2), which is the second version of the Robot Operating System (ROS 1). Overall, it is an open-source framework for building robotic software and was designed to address some of the limitations of the original ROS, such as scalability, security, and real-time performance. ROS 2 is built on top of a new middleware layer called Data Distribution Service (DDS), which provides a more robust and efficient communication between different components of a robot system. This allows ROS 2 to support a wider range of use cases, from small robots to large-scale distributed systems. That being said, it is to be noted that ROS 2 being a relatively new framework significantly lacks resources and community support when compared to ROS 1 distros. Furthermore, the specific distribution of ROS 2 (i.e. Foxy Fitzroy) which was employed for implementing this project has several issues (bugs) being one of the early releases of ROS 2, and as such the patches or workarounds are not available since the end-of-life (EOL) for this particular distribution is scheduled to be May 2023. We have compared different variants of ROS in detail and documented them here.

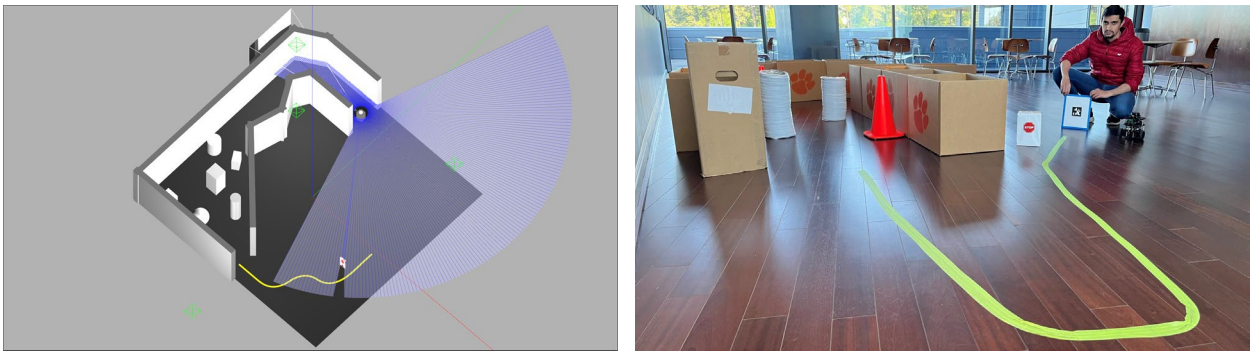
In terms of the languages used for programming, the high-level nodes were written in Python3 (an interpreted language) to ensure code readability and easy integration with modern computer vision and machine learning libraries. However, considering the requirement of "real-time" operation of the robot for safe and efficient autonomous navigation, low-level nodes and drivers were written in C/C++ (a compiled language) to ensure faster and optimal execution.

1.2. Project Tasks

The project comprises simulation as well as real-world deployment of various autonomy behaviors (refer **Figure 3**).

The original problem statement divided the project into 3 sub-tasks:

- **Task 1 - Wall Following and Obstacle Avoidance:** The robot is initialized in between two walls. It has to keep following the walls without colliding with them until it reaches an obstacle course. Here, the robot should switch its behavior to collision avoidance and traverse towards the other end of the obstacle course, where line following task starts.
- **Task 2 - Line Following and Stop Sign Detection:** The robot must successfully follow the yellow line on the floor. Additionally, the robot should detect a stop sign placed towards the end of this line and halt for some time before resuming its operation.
- **Task 3 - AprilTag Tracking:** The robot should detect and follow an AprilTag marker. This fiducial marker can be moved around manually (real-world) or teleoperated using another robot (simulation).



(a) (b)

Figure 3. *Project environment: (a) simulated world; and (b) real world.*

For our convenience, we divided the project into 5 sub-tasks:

- **Task 1 - Wall Following:** The robot has to keep following the walls without colliding with them until it reaches another zone in the course.
- **Task 2 - Obstacle Avoidance:** The robot should traverse towards the other end of the obstacle course, without colliding with any of them.
- **Task 3 - Line Following:** The robot must successfully follow the yellow line on floor.
- **Task 4 - Stop Sign Detection:** The robot should detect a stop sign placed in the environment and halt for some time before resuming its operation.
- **Task 5 - AprilTag Tracking:** The robot should detect and follow an AprilTag marker. This fiducial marker can be moved around manually (real-world) or teleoperated using another robot (simulation).

1.3. Project Timeline

As mentioned earlier, this project had a multitude of individual tasks that needed to be accomplished to fulfill the objectives of the continuous evaluation criterion. These included the following core tasks:

- **Task 1:** Wall Following
- **Task 2:** Obstacle Avoidance
- **Task 3:** Line Following
- **Task 4:** Stop Sign Detection
- **Task 5:** AprilTag Tracking

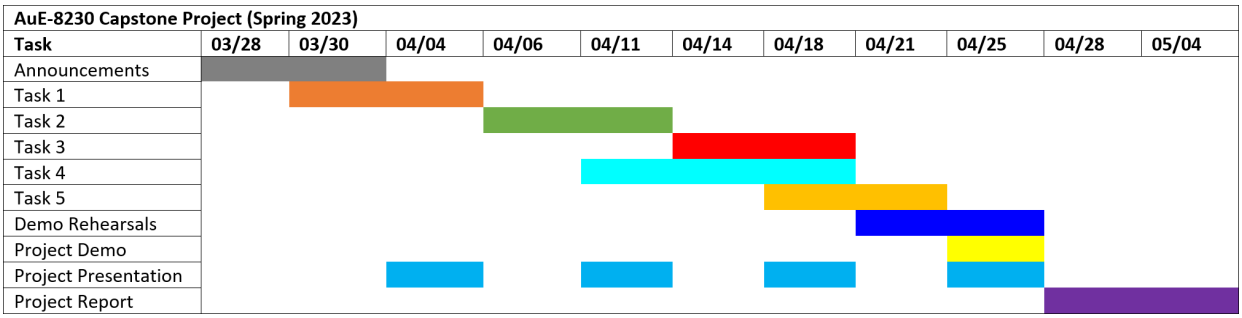


Figure 4. *Project timeline Gantt chart.*

Additionally, logistical tasks such as continuous documentation, demo rehearsals, demo delivery, preparation of slide deck, presentation rehearsals, report writing, etc. had to be accounted for in the overall project plan and strictly followed to ensure successful completion of the project.

Given the overall timeline from 03/28/2023 (with the beginning of initial announcements regarding the capstone project) to 05/04/2023 (end of Spring 2023 semester), the milestones and deliverables of the project were planned as indicated in **Figure 4**, which depicts the Gantt chart. It is to be noted that things such as known parallel commitments were considered while preparing the timeline so as to enable practical adherence rather than coming up with super tight but highly unrealistic timelines.

1.4. Project Management

We divided each of the individual project tasks as well as final integration into a set of responsibilities, which were assigned to individual team members (refer **Table 1**). We shuffled the responsibilities across different tasks to ensure that each individual member is able to develop his skillset in all dimensions.

Table 1. *Responsibility assignment matrix.*

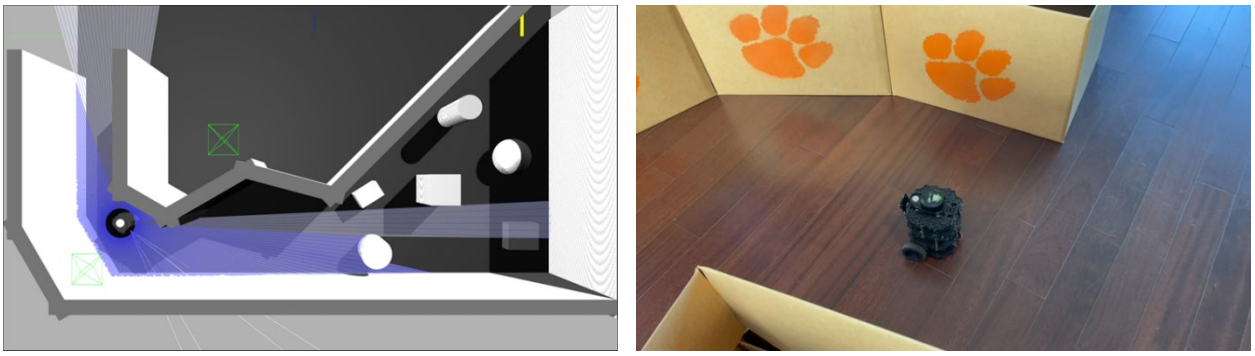
RESPONSIBILITY	Task 1	Task 2	Task 3	Task 4	Task 5	Integration
Algorithm development	Tanmay	Chinmay	Tanmay	Chinmay	Chinmay	Chinmay
Simulation setup	Chinmay	Tanmay	Chinmay	Tanmay	Chinmay	Chinmay
Simulation deployment	Chinmay	Tanmay	Chinmay	Tanmay	Tanmay	Chinmay
Real-world deployment	Tanmay	Chinmay	Tanmay	Chinmay	Tanmay	Tanmay
Git repository management	Chinmay	Chinmay	Chinmay	Tanmay	Chinmay	Tanmay
Documentation	Tanmay	Tanmay	Tanmay	Chinmay	Tanmay	Tanmay

It is to be noted that “responsibility” does not directly indicate “contribution”. Both members contributed equally to this project and have no conflict of interest to declare.

2. INDIVIDUAL TASKS

2.1. Task 1: Wall Following

The wall following task (refer **Figure 5**) demands the robot to keep centered between two adjacent walls at all times despite the curvature and irregularities in the wall surfaces and structure.

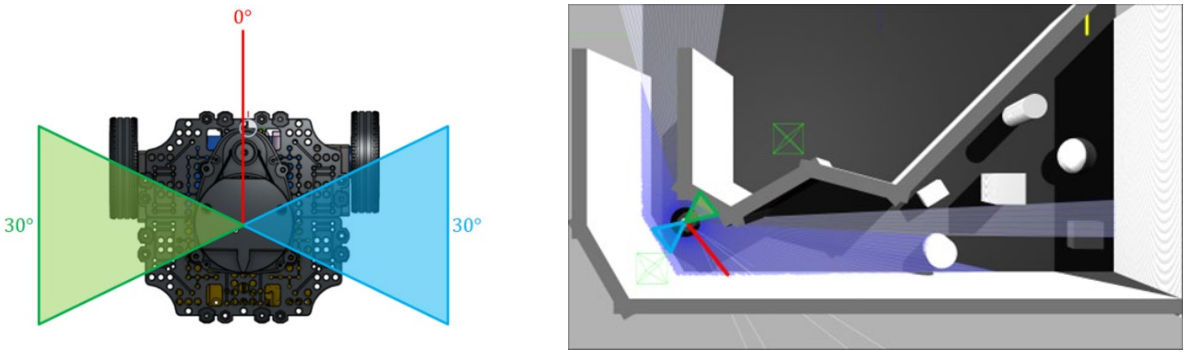


(a) (b)

Figure 5. *Wall following task: (a) simulated world; and (b) real world.*

In order to tackle this problem, we make use of LIDAR sensor and operate on the various sectors of the laser scan point cloud data (refer **Figure 6**). Following are the various sectors we particularly utilize to gain a better understanding of the robot's environment:

- Front laser scan ranging measurement [0]
- Left [75:105] laser scan sector
- Right [255:285] laser scan sector



(a) (b)

Figure 6. *Wall following task: (a) LIDAR ranging coverage; and (b) perception visualization.*

The frontal laser scan reading is used to gauge the frontal distance to collision (e.g. in case the walls are turning in a particular direction) and this information is used to regulate the longitudinal (i.e. linear) velocity of the robot so that the robot slows down in such situations (this is analogous to Adaptive cruise control).

The left and right laser scan sectors are used to compute cross-track error (CTE) from center of the two walls and this error metric is used to regulate the angular velocity of the robot such that the robot behavior is tuned to turn in the direction of (i.e. get attracted to) the center of two walls and get repealed from either walls to avoid collision.

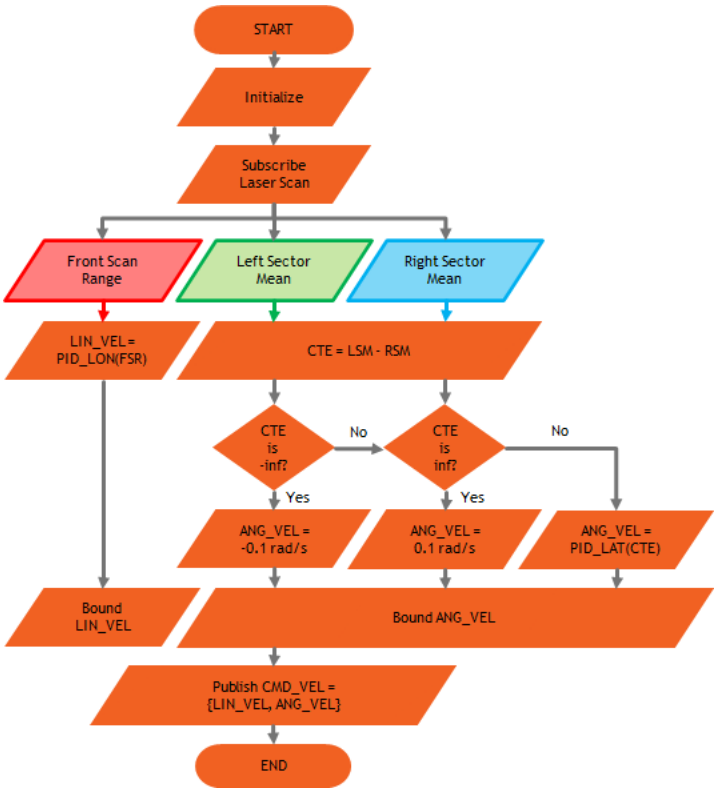
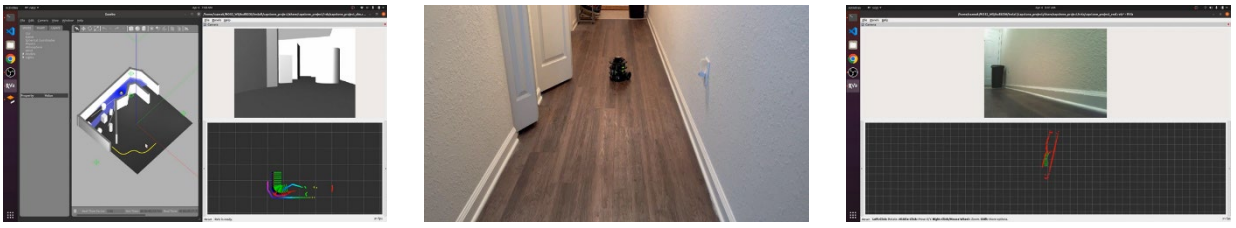


Figure 7. *Wall following task: high-level overview of the process and data flow.*

For executing this planned or intended robot behavior, we have formulated and implemented a de-coupled lateral-longitudinal proportional-integral-derivative (PID) controller architecture with a dynamic first-in-first-out (FIFO) anti-integral wind-up mechanism.

Furthermore, we have certain safety mechanisms in place to tackle the infinite laser scan measurements and unbounded control inputs being fed to the robot. For dealing with the infinite laser scan measurements, we clamp the LIDAR readings using a "min-max bounding" approach and operate on these clamped measurements. For dealing with the unbounded high level control commands for the robot (i.e. linear and angular velocities) based on the output of PID controllers, we impose realistic actuation limits (constraints) on these commands so that the filtered commands that are now fed to the robot are feasible (although the real actuators have physical limits and inherently disregard higher control inputs, it may degrade them, but more importantly, in case of Gazebo simulation which does not impose any such constraints on the robot motion this becomes a crucial way of simulating feasible dynamics so that the algorithms can be transferred to reality with one less thing to worry about).

In addition to the approach described above (refer **Figure 7**), we also tried different variations such as having different fields of view (FOVs), orientation and number of laser scan sectors, however the configuration highlighted in **Figure 6** was observed to work optimally under a variety of conditions. Furthermore, we tried computing the CTE based on minimum, maximum, mean, and weighted mean of the laser scan sectors. Since the wall following problem was symmetric in nature (maintaining the position across the center of two walls) the averaging method proved most efficient under a variety of conditions.



(a) (b) (c)

Figure 8. *Wall following task: (a) deployment and visualization in simulation [video]; (b) deployment in reality [video]; and (c) visualization in reality [video].*

The results indicate (refer **Figure 8**) successful wall following behavior in simulation as well as real world conditions despite the variations in robot initialization and the curvature and irregularities in the wall surfaces and structure.

2.2. Task 2: Obstacle Avoidance

The obstacle avoidance task (refer **Figure 9**) demands the robot to repel away from (i.e. avoid collision with) obstacles in proximity. An implicit objective of this task is also to guide the robot towards the line in the environment (without any queue regarding the location of the line whatsoever).

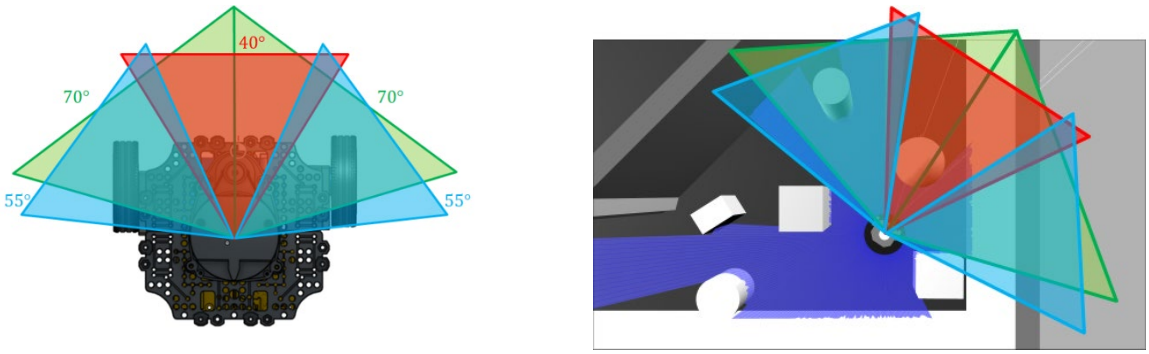
(a) (b)

Figure 9. *Obstacle avoidance task: (a) simulated world; and (b) real world.*

In order to tackle this problem, we make use of LIDAR sensor and operate on the various sectors of the laser scan point cloud data (refer **Figure 10**). Following are the various sectors we particularly utilize to gain a better understanding of the robot's environment:

- Front laser scan sector [0:20]+[340:360]
- Oblique laser scan sectors [0:70] and [290:360]
- Side laser scan sectors [30:85] and [275:330]

The cumulative overlap of these sectors span little less than 180 degrees since the exact side sectors are exploited for the wall following task. That being said, we have formulated and implemented a 3 stage attention mechanism wherein the robot analyzes the risk level (either safely away from obstacles or fairly away from obstacles or very close to obstacles) in terms of collision with proximal obstacles and uses a light-weight pseudo potential field method to negotiate its way through the obstacle course. The aggressiveness in robot's behavior to avoid obstacle(s) increases with the associated risk level as discussed earlier (the most aggressive behavior being turn-on-spot and the least one being to continue going straight).



(a) (b)

Figure 10. *Obstacle avoidance task: (a) LIDAR ranging coverage; and (b) perception visualization.*

For executing this planned or intended robot behavior, we have formulated and implemented a de-coupled lateral-longitudinal proportional-integral-derivative (PID) controller architecture with a dynamic first-in-first-out (FIFO) anti-integral wind-up mechanism.

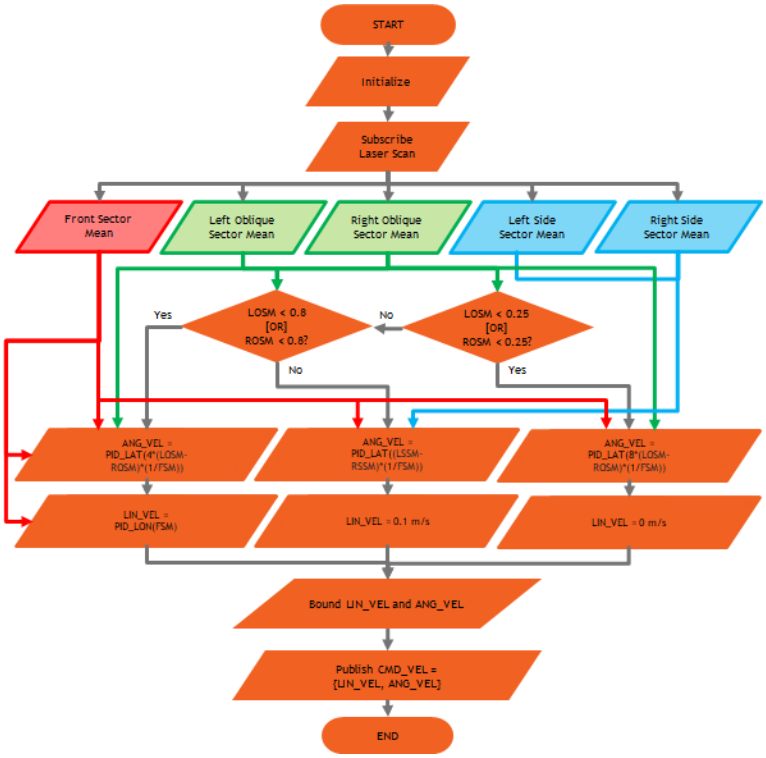
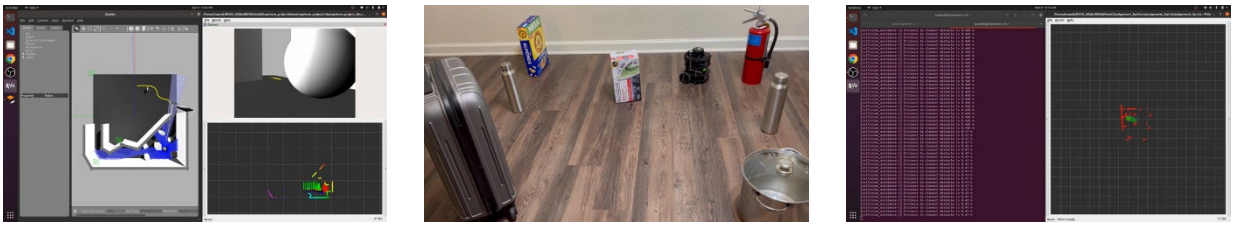


Figure 11. *Obstacle avoidance task: high-level overview of the process and data flow.*

Furthermore, we have certain safety mechanisms in place to tackle the infinite laser scan measurements and unbounded control inputs being fed to the robot. For dealing with the infinite laser scan measurements, we clamp the LIDAR readings using a "min-max bounding" approach and operate on these clamped measurements. For dealing with the unbounded high level control commands for the robot (i.e. linear and angular velocities) based on the output of PID controllers, we impose realistic actuation limits (constraints) on these commands so that the filtered commands that are now fed to the robot are feasible (although the real actuators have physical limits and inherently disregard higher control inputs, it may degrade them, but more importantly, in case of Gazebo simulation which does not impose any such constraints on the robot motion this becomes a crucial way of simulating feasible dynamics so that the algorithms can be transferred to reality with one less thing to worry about).

In addition to the approach described above (refer **Figure 11**), we also tried different variations such as having different fields of view (FOVs), orientation and number of laser scan sectors, however the configuration highlighted in **Figure 10** was observed to work optimally under a variety of conditions. Furthermore, we tried computing the distance to collision based on minimum, maximum, mean, and weighted mean of the laser scan sectors. Since there is a possibility of the robot getting stuck in the presence of a symmetric obstacle field, the weighted averaging method proved most efficient under a variety of conditions. This is because this method introduces asymmetry (or instability) into the system which then forces the robot to come out of this grid-lock configuration/situation.

One particular issue or challenge with the obstacle avoidance task was to appropriately weigh safety (collision avoidance) vs. performance (avoiding grid-locks or over-spooked U-turns to forcefully guide the robot towards the line in the environment without any queue regarding the location of the line whatsoever) of the behavior for optimal performance.



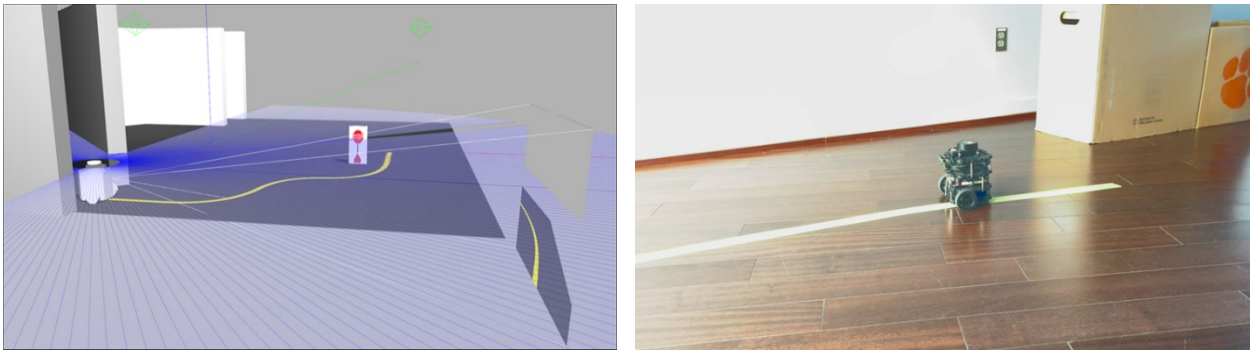
(a) (b) (c)

Figure 12. *Obstacle avoidance task: a) deployment and visualization in simulation [video]; (b) deployment in reality [video]; and (c) visualization in reality [video].*

The results indicate (refer **Figure 12**) successful obstacle avoidance behavior in simulation as well as real world conditions despite several variations in robot initialization and obstacle placement.

2.3. Task 3: Line Following

The line following task (refer **Figure 13**) demands the robot to follow or track the line of a particular color on the ground/floor of the environment (without any queue regarding the global location of the beginning of the line whatsoever).



(a) (b)

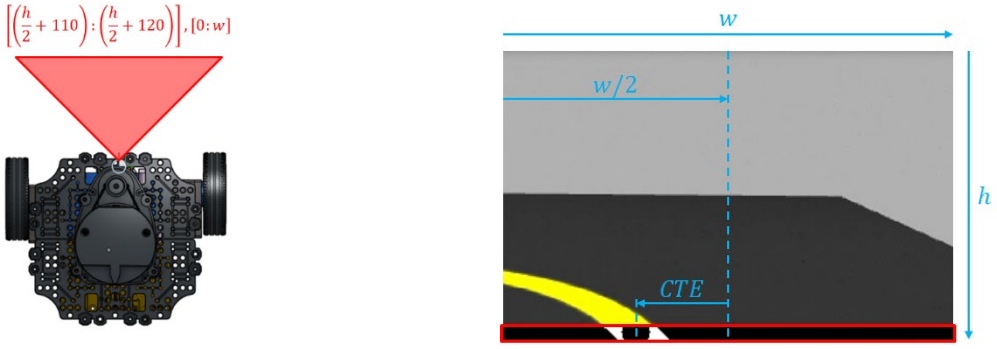
Figure 13. *Line following task: (a) simulated world; and (b) real world.*

In order to tackle this problem, we make use of the monocular camera sensor and operate on a specific region of interest (ROI) of the overall field of view (FOV) of the incoming image spanning 320x240 pixels (refer **Figure 14**). Specifically, we select the bottom-most 10 pixels

from the image (in terms of height) and keep the entire width as given by the following expression:

$$\left[\left(\frac{h}{2}+110\right):\left(\frac{h}{2}+120\right)\right],\ [0:w]$$

One of the prominent reasons for cropping the image according to desired ROI as the first step was to relax computational burden on the further stages of the pipeline.



(a) (b)

Figure 14. *Line following task: (a) camera field of view; and (b) perception visualization.*

The cropped image is transformed from red-green-blue (RGB) color space to hue-saturation-value (HSV) color space (refer **Figure 15**) also known as hue-saturation-brightness (HSB) in order to robustify the pipeline against light intensity variations. The reason this works is because the "value" channel or coordinate of the HSV color space indicates the brightness of the pixel intensities and as such once tuned appropriately with lower and upper thresholds, can work for different lighting conditions. Another good representation to achieve this robustness could be the hue-saturation-lightness (HSL) color space which is quite similar to HSV representation.

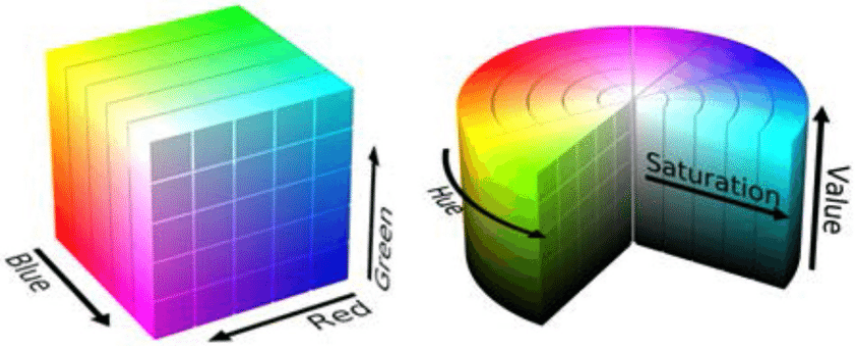


Figure 15. *Line following task: RGB vs. HSV color space visualization. [Source: V. Popov, et al.]*

Once the color space transformation is accomplished, we calculate weighted average of pixel intensities to detect pixel cluster(s), also known as blob(s) with similar features/properties. This cluster is then processed to calculate moments which help in computing the centroid of the cluster in image space. Based on the lateral image center coordinate (half the image width) and the centroid of the cluster, deviation of the robot from the line can be computed which can now be used as an error metric in a feedback control framework to continuously analyze and compensate for.

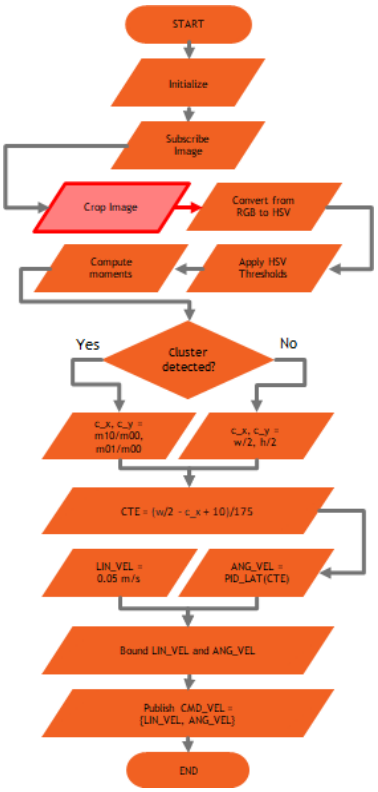


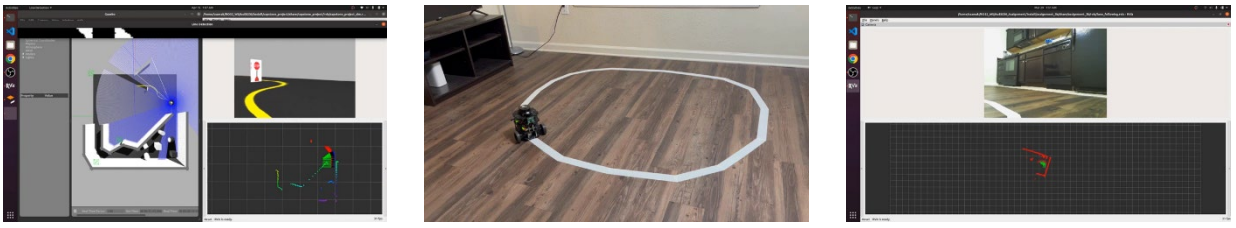
Figure 16. *Line following task: high-level overview of the process and data flow.*

For executing this planned or intended robot behavior, we have formulated and implemented a lateral proportional-integral-derivative (PID) controller architecture with a dynamic first-in-first-out (FIFO) anti-integral wind-up mechanism. For the longitudinal control aspect, we fixed a slow velocity value to ensure smooth maneuvers (i.e. avoid jerks in motion) and ensure continuous detections despite potential latencies in the pipeline (which is more probable when all the individual tasks are integrated in a finite state automaton).

Furthermore, we have certain safety mechanisms in place to tackle the unbounded high level control commands for the robot (i.e. linear and angular velocities) based on the output of PID controllers. For dealing with this issue, we impose realistic actuation limits on these commands so that the filtered commands that are now fed to the robot are feasible (although the real actuators have physical limits and inherently disregard higher control inputs, it may degrade them, but more importantly, in case of Gazebo simulation which does not impose any such constraints on the robot motion this becomes a crucial way of simulating feasible dynamics so that the algorithms can be transferred to reality with one less thing to worry about).

In addition to the approach described above (refer **Figure 16**), we also tried different variations such as having different regions of interest (ROIs), different upper and lower thresholds for HSV pixel filtering, however the aforementioned ROI and HSV thresholds noted here were observed to work optimally under a variety of conditions. Furthermore, we tried physically tilting (along the local pitch axis by ~15°) the camera so as to capture the portion of line closer to the robot (which turned out to be extremely beneficial in cases of highly sharp/acute turns). It is worth mentioning that although tilting the camera seems beneficial in the first glance, a good balance needs to be established so as not to compromise performance of other tasks such as stop sign detection or AprilTag tracking.

One particularly important consideration with image transport is that although uncompressed image transport might be acceptable for simulation, compressed image transport is highly recommended for real-world deployments in order to reduce on-board computational burden and over-the-air image transport latency significantly.



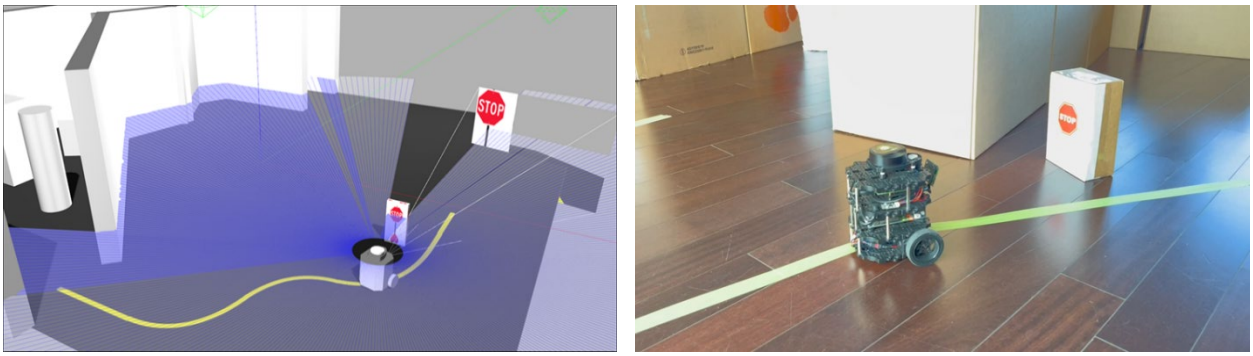
(a) (b) (c)

Figure 17. *Line following task: (a) deployment and visualization in simulation [video]; (b) deployment in reality [video]; and (c) visualization in reality [video].*

The results indicate (refer **Figure 17**) successful line following behavior in simulation as well as real world conditions despite several variations in robot initialization and lighting conditions.

2.4. Task 4: Stop Sign Detection

The stop sign detection task (refer **Figure 18**) demands the robot to come to a complete stop if a stop sign is detected in close proximity.



(a) (b)

Figure 18. *Stop sign detection task: (a) simulated world; and (b) real world.*

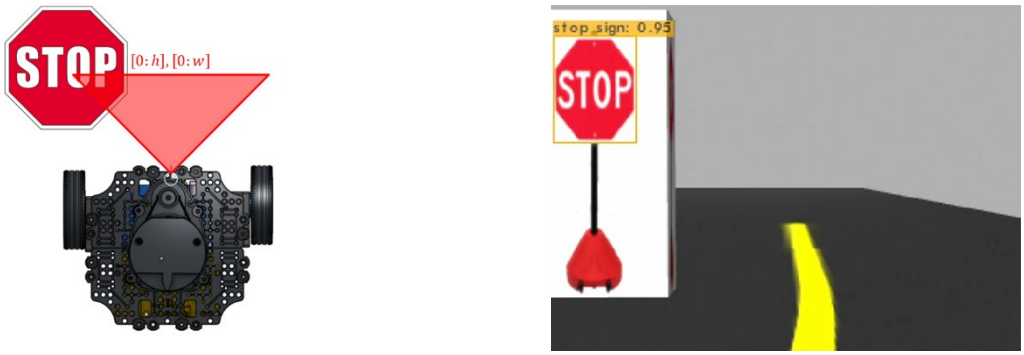
In order to tackle this problem, we make use of the monocular camera sensor and operate on the entire field of view (FOV) of the incoming image spanning 320x240 pixels (refer **Figure 19**) as given by the following expression:

$$[0:h],\ [0:w]$$

One of the prominent reasons for operating on the entire FOV of the incoming image is to detect the stop sign anywhere in the scene that is visible to the camera (e.g. on right as well as left side of the line/road).

At the outset, we configure & run Tiny-YOLO V7 model inference on incoming image. Upon valid availability of detections, we then verify the detection class (which needs to be a stop sign), confidence (which needs to be greater than at least 50%) & bounding box area of the stop sign (which depends upon the metric size of the stop sign as well as the desired stopping

distance). If all these conditions are met, we command the robot to come to a complete stop for a predetermined stopping time of 4 seconds. Once the timer has elapsed the robot continues its motion based on the applicable heuristic.



(a) (b)

Figure 19. *Stop sign detection task: (a) camera field of view; and (b) perception visualization.*

Furthermore, we have certain safety mechanisms in place to tackle the unbounded high level control commands for the robot (i.e. linear and angular velocities) based on the output of PID controllers. For dealing with this issue, we impose realistic actuation limits (constraints) on these commands so that the filtered commands that are now fed to the robot are feasible (although the real actuators have physical limits and inherently disregard higher control inputs, it may degrade them, but more importantly, in case of Gazebo simulation which does not impose any such constraints on the robot motion this becomes a crucial way of simulating feasible dynamics so that the algorithms can be transferred to reality with one less thing to worry about).

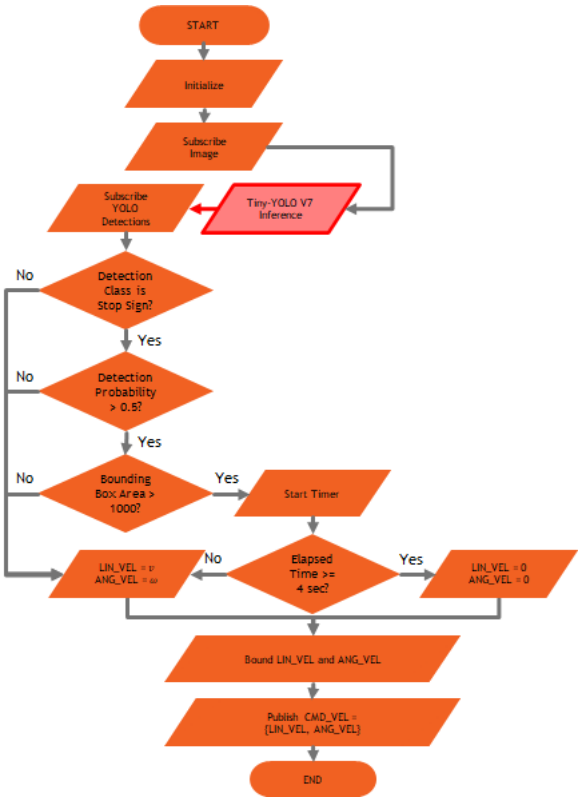
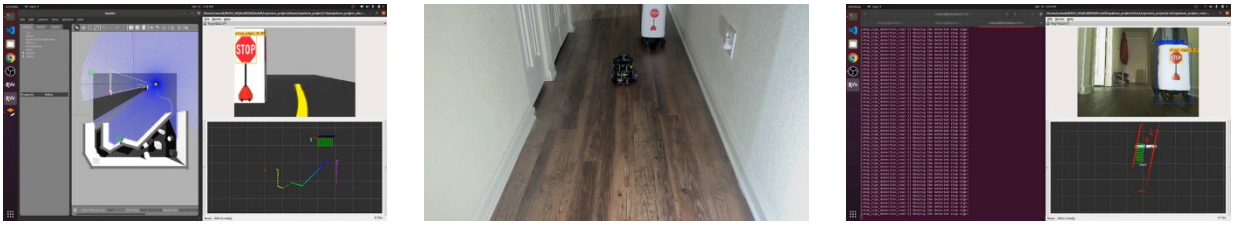


Figure 20. *Stop sign detection task: high-level overview of the process and data flow.*

In addition to the approach described above (refer **Figure 20**), we also tried different variations such as having different YOLO model versions, different confidence limits and bounding box areas, and also implementing and comparing computer vision (CV) and machine learning (ML) based algorithms. It was observed that the Tiny-YOLO V7 model was the fastest and most robust in terms of stop sign detection with respect to the aforementioned confidence limit and bounding box area choices based on the metric size of the stop sign as well as the desired stopping distance. As expected, comparison of CV and ML models revealed the clear superiority of the latter in terms of robustness to variations and was therefore chosen over the former.

One particularly important consideration with image transport is that although uncompressed image transport might be acceptable for simulation, compressed image transport is highly recommended for real-world deployments in order to reduce on-board computational burden and over-the-air image transport latency significantly.



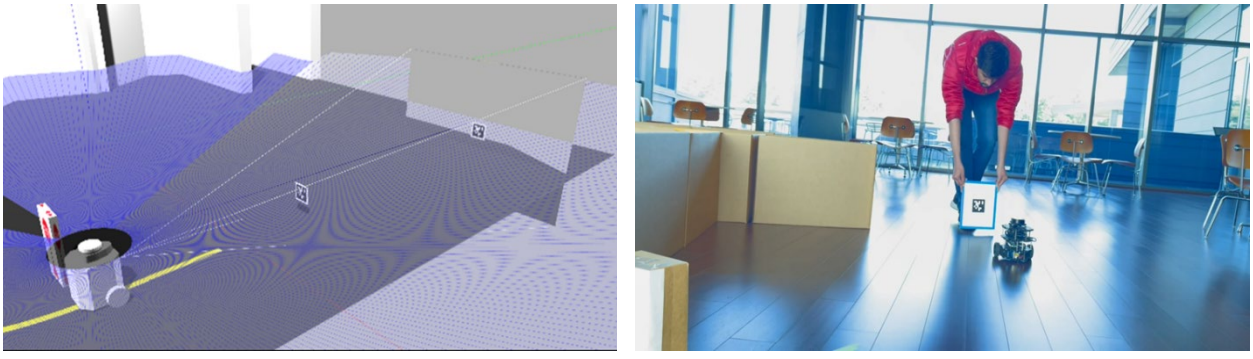
(a) (b) (c)

Figure 21. *Stop sign detection task: (a) deployment and visualization in simulation [video]; (b) deployment in reality [video]; and (c) visualization in reality [video].*

The results indicate (refer **Figure 21**) successful stop sign detection behavior in simulation as well as real world conditions despite several variations in robot initialization and lighting conditions.

2.5. Task 5: AprilTag Tracking

The AprilTag tracking task (refer **Figure 22**) demands the robot to track an AprilTag marker in the environment (and preferably come to a complete stop if too close to the marker in order to avoid collision with it and also in a way to ensure that the marker does not go out of the FOV of the camera).



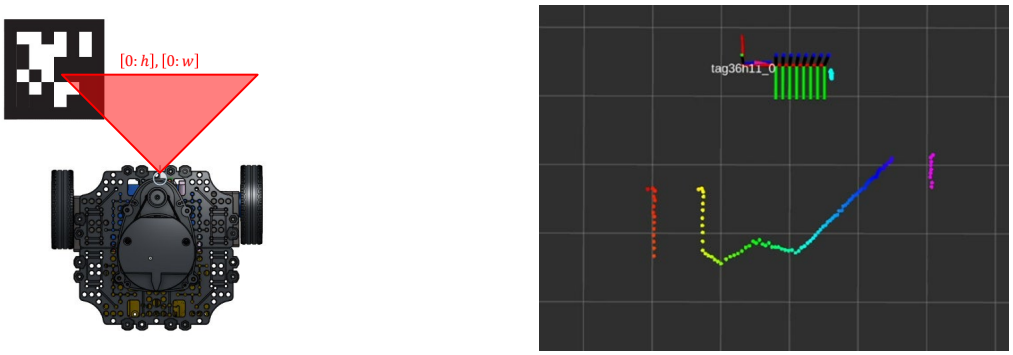
(a) (b)

Figure 22. *AprilTag tracking task: (a) simulated world; and (b) real world.*

In order to tackle this problem, we make use of the monocular camera sensor and operate on the entire field of view (FOV) of the incoming image spanning 320x240 pixels (refer **Figure 23**) as given by the following expression.

$$[0:h],\ [0:w]$$

One of the prominent reasons for operating on the entire FOV of the incoming image is to detect the AprilTag marker anywhere in the scene that is visible to the camera.



(a) (b)

Figure 23. *AprilTag tracking task: (a) camera field of view; and (b) perception visualization.*

At the outset, we define a static homogeneous rigid-body transform from robot's base link to the camera (since this is not defined by default owing to the fact that the camera was added by us as a modification to the TurtleBot3 Burger in simulation as well as reality and hence the default packages as expected fail to capture it).

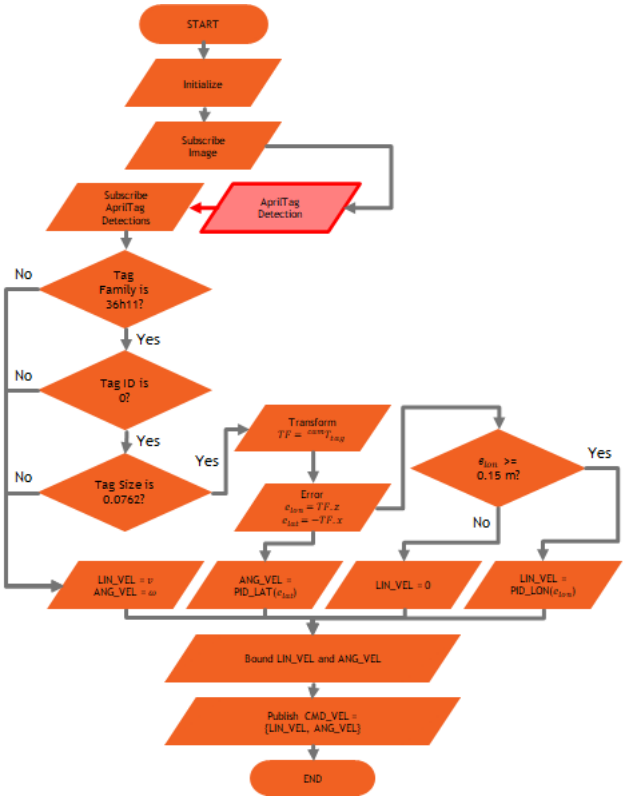


Figure 24. *AprilTag tracking task: high-level overview of the process and data flow.*

Next, we fire up the AprilTag marker detection node which filters detections based on tag family, tag ID as well as the tag size. The particular choice of including the tag size as a filtering criteria was so as to make our robot robust against detections of unintended AprilTag markers (e.g. those of other teams just lying around or coming in camera FOV for some reason).

With a valid transformation between robot and its camera, we compute the 6D metric pose of the AprilTag marker w.r.t. robot's camera using the calibrated intrinsic parameters (further details and code available here) for our specific camera module (refer **Figure 19**). The information pertaining to the lateral and longitudinal deviation of the marker w.r.t. the robot is extracted using the negative x coordinate and positive z coordinate respectively, which can now be used as error metrics in a feedback control framework to continuously analyze and compensate for.

For executing this planned or intended robot behavior, we have formulated and implemented a de-coupled lateral-longitudinal proportional-integral-derivative (PID) controller architecture with a dynamic first-in-first-out (FIFO) anti-integral wind-up mechanism.

Furthermore, we have certain safety mechanisms in place to tackle the unbounded high level control commands for the robot (i.e. linear and angular velocities) based on the output of PID controllers. For dealing with this issue, we impose realistic actuation limits (constraints) on these commands so that the filtered commands that are now fed to the robot are feasible (although the real actuators have physical limits and inherently disregard higher control inputs, it may degrade them, but more importantly, in case of Gazebo simulation which does not impose any such constraints on the robot motion this becomes a crucial way of simulating feasible dynamics so that the algorithms can be transferred to reality with one less thing to worry about).

In addition to the approach described above (refer **Figure 24**), we also tried different variations such as having different print qualities (affecting the detections due to anti-aliasing of marker edges), tag IDs and tag sizes. Although the print quality (to a certain degree), tag ID and size doesn't affect the detection performance, we down-scoped these to be high quality (to ensure maximum robustness against environmental variations), 0 and 0.0762 meters respectively to keep these unique to our team and thereby ensure optimal performance.

One particularly important consideration with image transport is that although uncompressed image transport might be acceptable for simulation, compressed image transport is highly recommended for real-world deployments in order to reduce on-board computational burden and over-the-air image transport latency significantly.

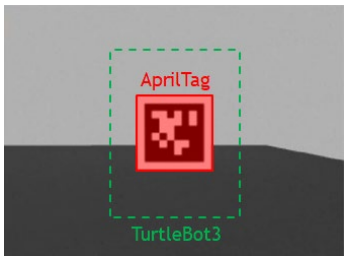
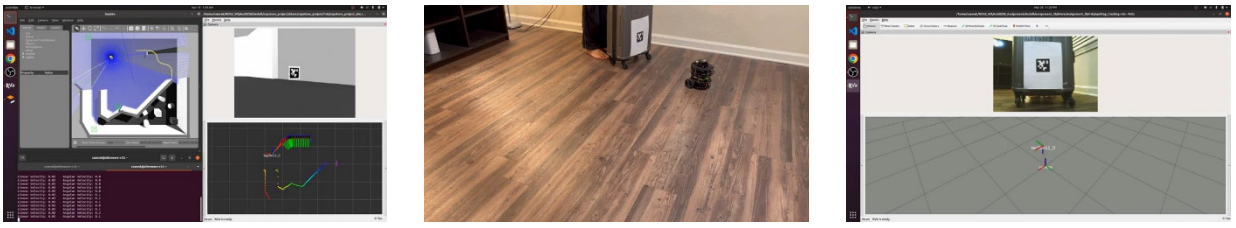


Figure 25. *AprilTag tracking task: Setting up invisible TurtleBot3 in simulation with AprilTag marker.*

From an implementation perspective (in simulation, which was more challenging for this task), we defined another TurtleBot3 and attached an AprilTag marker of appropriate family, ID and size to it. Furthermore, we removed all sensors from this robot and made it invisible to have a realistic and clean/aesthetic appearance (refer **Figure 25**).



(a) (b) (c)

Figure 26. *AprilTag tracking task: (a) deployment and visualization in simulation [video]; (b) deployment in reality [video]; and (c) visualization in reality [video].*

The results indicate (refer **Figure 26**) successful AprilTag tracking behavior in simulation as well as real world conditions despite several variations in robot initialization and lighting conditions.

3. PROJECT INTEGRATION

3.1. Finite State Automaton

As depicted in **Figure 27**, we start by initializing all the necessary nodes and check if LIDAR as well as camera data are available. This is crucial since our finite state machine (FSM) depends on redundant perception from both these sensors (to detect environmental cues) for state transition, not to mention the successful execution of all tasks/behaviors needs continuous information input from either of these sensors.

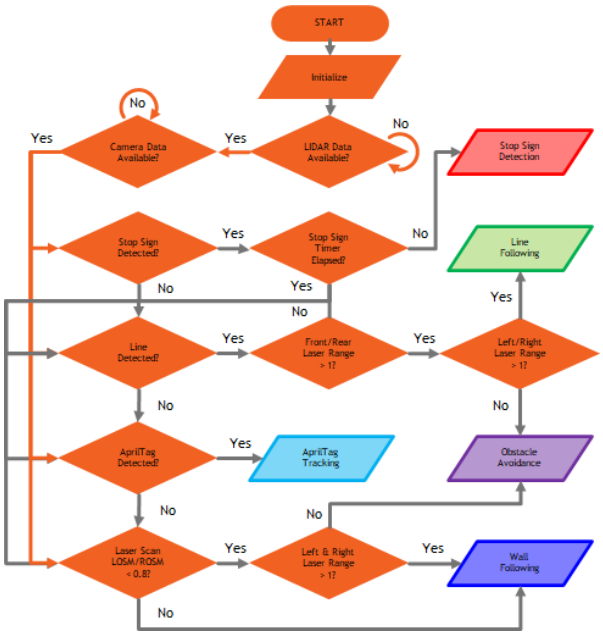


Figure 27. *Finite state automaton for automatic mode switching: high-level overview of the process and data flow.*

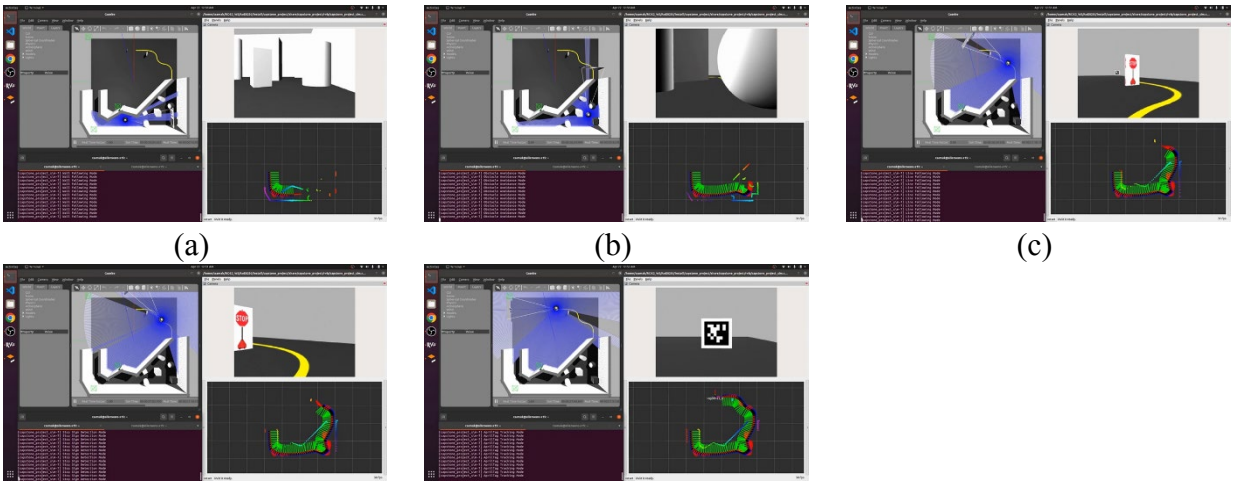
In terms of priority scheduling, stop sign obedience takes precedence (utmost priority) followed by line following followed by AprilTag tracking followed by obstacle avoidance followed by wall following.

In what follows, we discuss the automatic state transition feature of our pipeline and in order to limit the complexity of the explanation, we do NOT explicitly reiterate (since it is already covered in the previous sections) on all the different conditions/criteria that need to be satisfied for activating/driving a particular state.

The stop sign detection state flag is set true if the stop sign is detected and the stopping timer has not yet elapsed. The line following state flag is set true if line is detected, the robot is not stopped at stop sign and if front or rear and left or right laser scan measurements are greater than 1 m (this way, we get rid of any potential false line detections when performing wall following/obstacle avoidance). The AprilTag tracking state flag is set true if a tag is detected, line is not detected and the robot has not stopped at a stop sign. The obstacle avoidance flag is set true if tag is not detected, line is not detected, robot has not stopped at a stop sign and if laser scan left oblique scan measurement (LOSM) or right oblique scan measurement (ROSM) is less than 0.8 m. Finally, the wall following flag is set true if tag is not detected, line is not detected, and if the robot has not stopped at a stop sign.

3.2. Simulation Deployment

The results indicate (refer **Figure 28**) successful automatic behavior switching (state transition) of the robot based on unprivileged cues from the environment in simulated conditions despite several variations in robot initialization and environmental conditions.



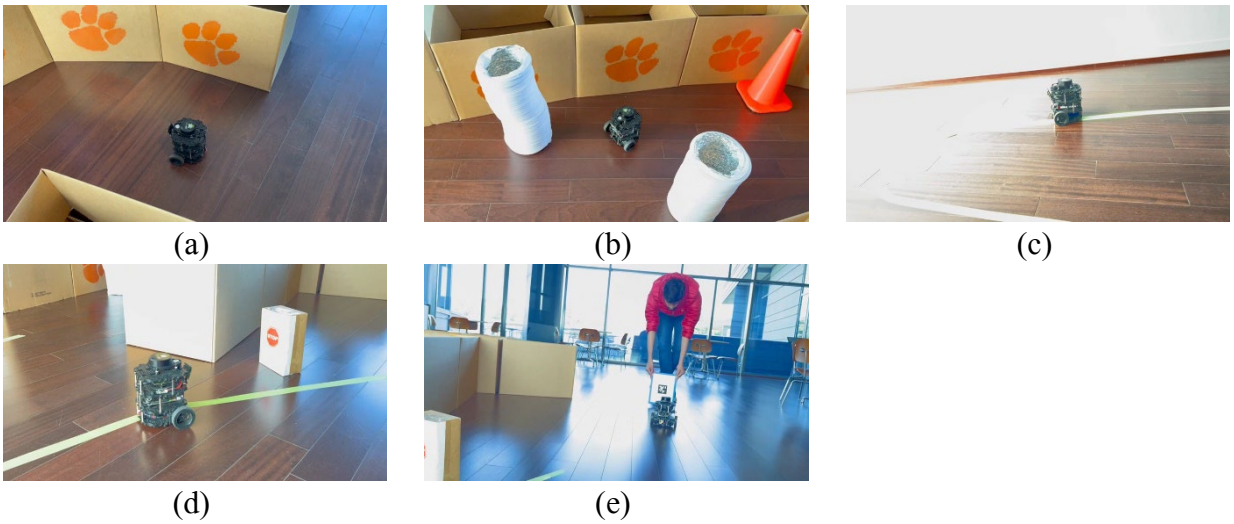
(a) (b) (c)

(d) (e)

Figure 28. *Simulation deployment [video]: (a) wall following mode; (b) obstacle avoidance mode; (c) line following mode; (d) stop sign detection mode; and (e) AprilTag tracking mode.*

3.3. Real-World Deployment

The results indicate (refer **Figure 29**) successful automatic behavior switching (state transition) of the robot based on unprivileged cues from the environment in real world conditions despite several variations in robot initialization and environmental conditions.



(a) (b) (c)

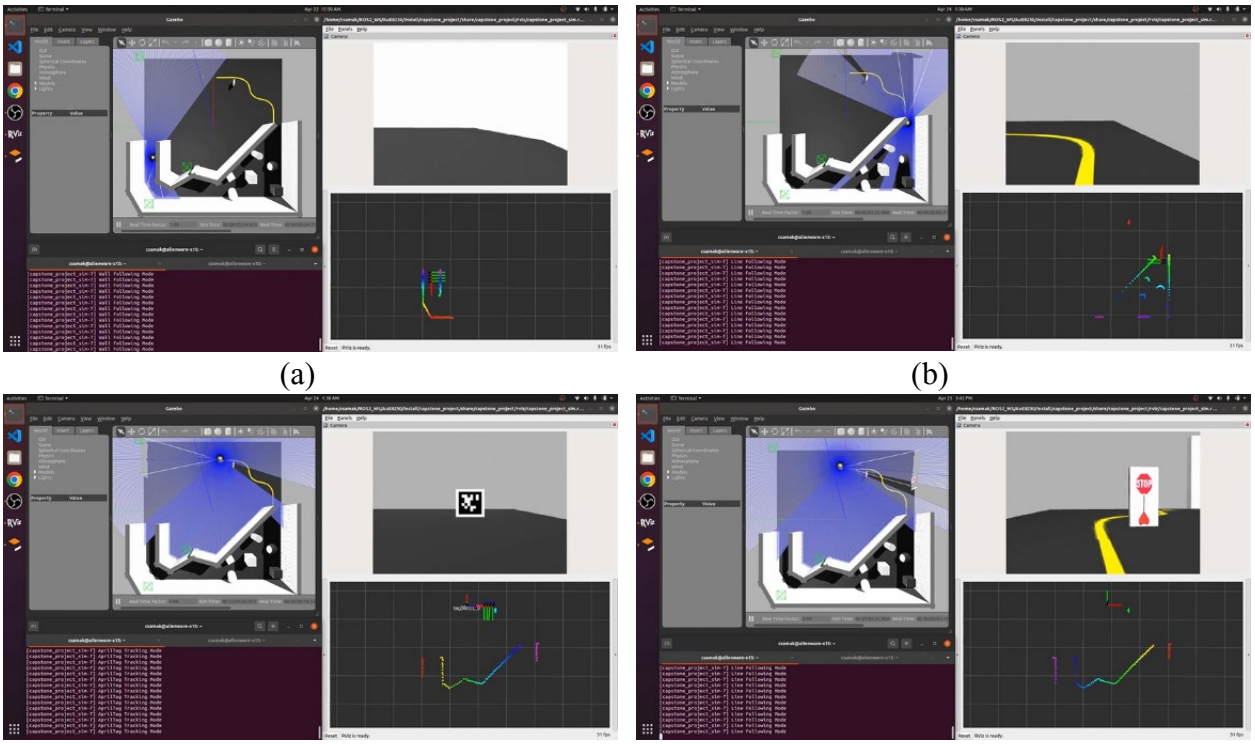
(d) (e)

Figure 29. *Real-world deployment [video]: (a) wall following mode; (b) obstacle avoidance mode; (c) line following mode; (d) stop sign detection mode; and (e) AprilTag tracking mode.*

3.4. Robustness Testing

In order to analyze the robustness of the automatic behavior switching mechanism as well as all the individual tasks/behaviors/states, we performed variability analysis in both simulation and real world conditions. This included things like starting from a different location (starting

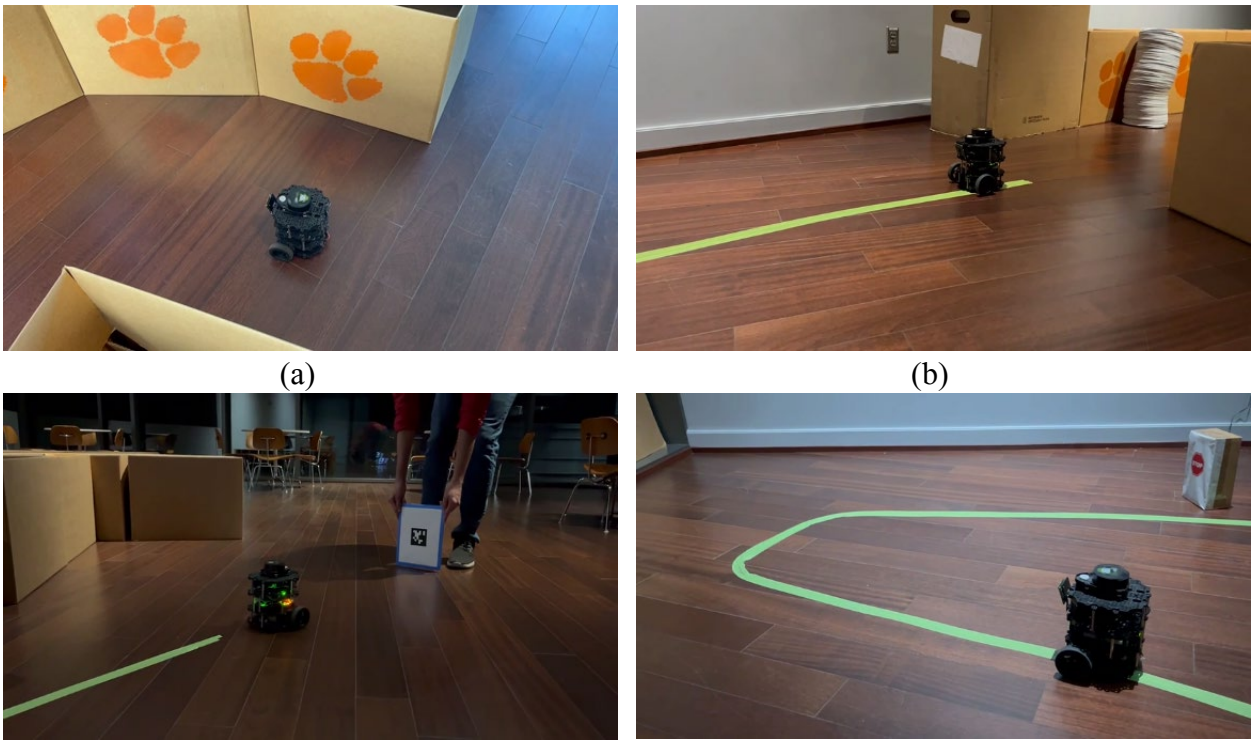
with different behavior), traversing the entire course in forward and inverse directions, and most importantly, driving during different times of the day as well as during the night.



(a) (b)

(c) (d)

Figure 30. *Robustness testing experiments in simulation: (a) forward direction, start with wall following [video]; (b) forward direction, start with line following [video]; (c) forward direction, start with AprilTag tracking [video]; and (d) inverse direction, start with line following [video].*



(a) (b)

(c) (d)

Figure 31. *Robustness testing experiments in real world: (a) day, forward direction, start with wall following [video]; (b) night, forward direction, start with line following [video]; (c) night, forward direction, start with AprilTag tracking [video]; and (d) night, inverse direction, start with line following [video].*

The results indicate (refer **Figure 30** and **Figure 31**) successful automatic behavior switching (state transition) of the robot based on unprivileged cues from the environment as well as individual behavior execution in both simulation as well as real world conditions despite several variations in robot initialization and environmental conditions.

4. CODE INTEGRATION

4.1. Robot Setup

The simulated robot required the following steps to be setup:

- `turtlebot3_burger.urdf` from `turtlebot3_description` package was modified to define the fixed links and joints (static transforms) for the camera module for simulating TurtleBot3 Burger with a camera in Gazebo simulator.
- `model.sdf` from `turtlebot3_gazebo` package was modified to define links, physical properties as well as the fixed joints (static transforms) for the camera module for simulating TurtleBot3 Burger with a camera in Gazebo simulator.
- We defined another TurtleBot3 and attached an AprilTag marker of appropriate family, ID and size to it. Furthermore, we removed all sensors from this robot's `model.sdf` and made it invisible to have a realistic and clean/aesthetic appearance.

The only modification to the physical robot was tilting the camera by ~15° to better capture the line (especially sharp curves).

4.2. Environment Setup

The simulated environment required the following steps to be setup:

- `capstone_project.world` was defined from scratch to setup the environment with ground, walls, obstacles, traffic sign and mobile AptilTag marker. Issues such as unstable obstacles have been rectified with our implementation.
- `capstone_project.sdf` was defined to spawn the modified TurtleBot3 Burger with a camera in the `capstone_project.world` environment.

The real-world environment was setup by the TAs (Dhruv Mehta and Sumedh Sathe), and we really appreciate their efforts. We only changed the environment slightly for carrying out variability analysis to test the robustness of our algorithms.

4.3. Dependencies

Our implementation assumes the following dependencies:

- TurtleBot3 Burger Robot Hardware with TurtleBot3 SBC Image
- ROS2 Foxy Fitzroy on Ubuntu 20.04 Focal Fossa
- TurtleBot3 Packages - Included within our repository
- TurtleBot3 Simulations Packages - Included within our repository
- TurtleBot3 Messages Package - Included within our repository
- TurtleBot3 Dynamixel SDK Packages - Included within our repository

4.4. Scripts

The ROS2 package `capstone_project` for this project hosts the following Python scripts:

- `wall following sim.py` makes the robot follow walls by maintaining equal distance from them. The robot makes use of de-coupled longitudinal and lateral PID controllers (with FIFO integral anti-windup mechanism) acting on frontal distance to collision (based on 0° laser scan reading) and relative distance from walls (based on difference in means of left and right laser scan sectors spanning 30° each) respectively for motion control. A simple finite state machine is implemented to account for inf scan measurements beyond LIDAR maximum range. The script is setup to wait 4 seconds for simulation to initialize properly and then run for eternity to demonstrate dynamically variable wall following application.

- `obstacle avoidance sim.py` makes the robot avoid obstacles by maintaining a safe distance from them. The robot makes use of de-coupled longitudinal and lateral PID controllers (with FIFO integral anti-windup mechanism) acting on frontal distance to collision (based on mean of frontal laser scan sector of 40°), distance to collision from oblique left and right sectors spanning 70° each, and distance to collision from left and right sectors spanning 55° each respectively for motion control. A pseudo-potential field method is implemented with a hierarchical 3 stage attention mechanism to act on close, fairly far and safely far obstacles. A finite state machine is implemented to account for turning left, turning right, going straight and cautiously rotating on the spot in case of too many obstacle in proximity. The script is setup to wait 4 seconds for simulation to initialize properly and then run for eternity to demonstrate dynamically variable obstacle avoidance application.

- `line following sim.py` makes the robot perform line following operation. The robot is commanded to move with 0.05 m/s linear velocity. The incoming RGB camera frame (320x240 px) is cropped according to the region of interest (ROI) of 10 px height and entire width, converted to HSV color space, and masked into a binary image using upper and lower HSV thresholds for bright yellow color. The binary image is then used to calculate moments (weighted average of image pixel intensities) to detect cluster of pixels (i.e. blob), compute the centroid of this blob, and then this centroid is used to calculate error (deviation) of robot from line center. A PID controller (with FIFO integral anti-windup mechanism) operates on this error to accordingly command the robot's angular velocity (rad/s) to perform line following operation. The script is setup to wait 4 seconds for everything to initialize properly and then run for eternity to demonstrate dynamically variable lane following application.

- `stop sign detection sim` makes the robot come to a complete stop for a brief amount of time if a stop sign is detected within a threshold distance. The incoming RGB camera frame (320x240 px) is passed through a pretrained Tiny-YOLO V7 model in inference mode. The resultant object detections are filtered on the fly based upon their class, classification confidence, and area of the bounding box in image space (implicitly giving away the detection and localization information of the stop sign). A finite state machine is implemented to act on the live filtered detections and determine the exact instance to come to a complete stop based on the preset threshold bounding box area (making sure that the robot doesn't stop too near or too far away from the stop sign). The script is setup to wait 4 seconds for simulation to initialize properly and then run for eternity to demonstrate dynamically variable obstacle avoidance application.

- `apriltag_tracking_sim.py` makes the robot detect and track AprilTag markers. The incoming camera feed from the robot is fed to the AprilTag detection pipeline from which the relative 6D pose of the AprilTag marker with respect to the robot camera is extracted. A static rigid body homogeneous transform is defined and applied between robot and camera frames to account for their relative pose. The robot makes use of de-coupled longitudinal and lateral PID controllers (with FIFO integral anti-windup mechanism) acting on the positive translational Z and negative translational X components of robot relative marker pose respectively for motion control. The script is setup to wait 4 seconds for everything to initialize properly and then run for eternity to demonstrate dynamically variable AprilTag tracking application.

- `apriltag_teleop.py` makes the AprilTag marker (attached to an invisible TurtleBot3) move around based on the linear and angular velocity commands given by human from standard computer keyboard interface. `w/x` keys increase/decrease linear velocity, `a/d` keys increase/decrease angular velocity and `s` key applies emergency brakes. The script is setup to spin indefinitely to perform dynamically variable AprilTag marker teleoperation as required/demanded.

- `capstone_project_sim.py` makes the robot perform all the aforementioned tasks by autonomous mode-switching behavior solely depending upon salient cues from the environment (without any human intervention). The incoming laser scan pattern and camera frame features from the robot are analyzed so as to deduce the current functional requirement for the robot. The algorithm has been robustified against false detections (e.g. to activate line following mode, it is ensured that there are no obstructions within on either of the two sides and either front or behind the robot, detection confidence and bounding box area of stop sign is continuously analyzed to avoid stopping randomly, etc.). In terms of priority scheduling, stop sign obedience takes precedence (utmost priority) followed by line following followed by AprilTag tracking followed by obstacle avoidance followed by wall following. The script is setup to wait 4 seconds for everything to initialize properly and then run for eternity to demonstrate dynamically reliable execution of the capstone project.

- `wall_following_real.py` makes the robot follow walls by maintaining equal distance from them. The robot makes use of de-coupled longitudinal and lateral PID controllers (with FIFO integral anti-windup mechanism) acting on frontal distance to collision (based on 0° laser scan reading) and relative distance from walls (based on difference in means of left and right laser scan sectors spanning 30° each) respectively for motion control. A simple finite state machine is implemented to account for 0 scan measurements beyond LIDAR maximum range. The script is setup to wait 4 seconds for simulation to initialize properly and then run for eternity to demonstrate dynamically variable wall following application.

- `obstacle_avoidance_real.py` makes the robot avoid obstacles by maintaining a safe distance from them. The robot makes use of de-coupled longitudinal and lateral PID controllers (with FIFO integral anti-windup mechanism) acting on frontal distance to collision (based on mean of frontal laser scan sector of 40°), distance to collision from oblique left and right sectors spanning 20° each, and distance to collision from left and

right sectors spanning 40° each respectively for motion control. A pseudo-potential field method is implemented with a hierarchical 3 stage attention mechanism to act on close, fairly far and safely far obstacles. The sector sizes are different from simulation owing to the fact that real-world LIDAR will pick up objects other than just the walls or obstacles. A finite state machine is implemented to account for turning left, turning right, going straight and cautiously rotating on the spot in case of too many obstacle in proximity. The script is setup to wait 4 seconds for simulation to initialize properly and then run for eternity to demonstrate dynamically variable obstacle avoidance application.

- `line following real.py` makes the robot perform line following operation. The robot is commanded to move with 0.05 m/s linear velocity. The incoming RGB camera frame (320x240 px) is uncompressed, cropped according to the region of interest (ROI) of 10 px height and entire width, converted to HSV color space, and masked into a binary image using upper and lower HSV thresholds for light (fluorescent) yellow color. The binary image is then used to calculate moments (weighted average of image pixel intensities) to detect cluster of pixels (i.e. blob), compute the centroid of this blob, and then this centroid is used to calculate error (deviation) of robot from line center. A PID controller (with FIFO integral anti-windup mechanism) operates on this error to accordingly command the robot's angular velocity (rad/s) to perform line following operation. The script is setup to wait 4 seconds for everything to initialize properly and then run for eternity to demonstrate dynamically variable lane following application.

- `stop sign detection real` makes the robot come to a complete stop for a brief amount of time if a stop sign is detected within a threshold distance. The incoming compressed RGB camera frame (320x240 px) is uncompressed and passed through a pretrained Tiny-YOLO V7 model in inference mode. The resultant object detections are filtered on the fly based upon their class, classification confidence, and area of the bounding box in image space (implicitly giving away the detection and localization information of the stop sign). A finite state machine is implemented to act on the live filtered detections and determine the exact instance to come to a complete stop based on the preset threshold bounding box area (making sure that the robot doesn't stop too near or too far away from the stop sign). The script is setup to wait 4 seconds for simulation to initialize properly and then run for eternity to demonstrate dynamically variable obstacle avoidance application.

- `apriltag tracking real.py` makes the robot detect and track AprilTag markers. The incoming compressed camera feed from the robot is uncompressed and fed to the AprilTag detection pipeline from which the relative 6D pose of the AprilTag marker with respect to the robot camera is extracted. A static rigid body homogeneous transform is defined and applied between robot and camera frames to account for their relative pose. The robot makes use of de-coupled longitudinal and lateral PID controllers (with FIFO integral anti-windup mechanism) acting on the positive translational Z and negative translational X components of robot relative marker pose respectively for motion control. The script is setup to wait 4 seconds for everything to initialize properly and then run for eternity to demonstrate dynamically variable AprilTag tracking application.

- `capstone_project_real.py` makes the robot perform all the aforementioned tasks by autonomous mode-switching behavior solely depending upon salient cues from the environment (without any human intervention). The incoming laser scan pattern and camera frame features from the robot are analyzed so as to deduce the current functional requirement for the robot. The algorithm has been robustified against false detections (e.g. to activate line following mode, it is ensured that there are no obstructions within 1.0 m on either of the two sides and either front or behind the robot, detection confidence and bounding box area of stop sign is continuously analyzed to avoid stopping randomly, etc.) to ensure seamless mode-switching under a multitude of environmental conditions. In terms of priority scheduling, stop sign obedience takes precedence (utmost priority) followed by line following followed by AprilTag tracking followed by obstacle avoidance followed by wall following. The script is setup to wait 4 seconds for everything to initialize properly and then run for eternity to demonstrate dynamically reliable execution of the capstone project.

4.5. Launch Files

The ROS2 package `capstone_project` for this project hosts the following launch files:

- `project_world.launch.py` launches Gazebo simulator with `capstone_project.sdf` as well as the `robot_state_publisher.launch.py`.

- `wall_following_sim.launch.py` launches Gazebo simulator with `wall_following.sdf`, the `robot_state_publisher.launch.py`, the `wall_following_sim.py` script as well as an RViz window to visualize the camera feed, laser scan and odometry estimates as well as a relative transformation between robot and AprilTag marker(s).

- `obstacle_avoidance_sim.launch.py` launches Gazebo simulator with `obstacle_avoidance.sdf`, the `robot_state_publisher.launch.py`, the `obstacle_avoidance_sim.py` script as well as an RViz window to visualize the camera feed, laser scan and odometry estimates as well as a relative transformation between robot and AprilTag marker(s).

- `line_following_sim.launch.py` launches Gazebo simulator with `line_following.sdf`, the `robot_state_publisher.launch.py`, the `line_following_sim.py` script as well as an RViz window to visualize the camera feed, laser scan and odometry estimates as well as a relative transformation between robot and AprilTag marker(s).

- `stop_sign_detection_sim.launch.py` launches Gazebo simulator with `stop_sign_detection.sdf`, the `robot_state_publisher.launch.py`, the `darknet_ros.launch.py` with `capstone_sim.yaml` configuration, the `stop_sign_detection_sim.py` script as well as an RViz window to visualize the camera feed, laser scan and odometry estimates as well as a relative transformation between robot and AprilTag marker(s).

- apriltag_tracking_sim.launch.py launches Gazebo simulator with apriltag_tracking.sdf, the robot_state_publisher.launch.py, the apriltag_node with tags_36h11.yaml configuration, the apriltag_tracking_sim.py script as well as an RViz window to visualize the camera feed, laser scan and odometry estimates as well as a relative transformation between robot and AprilTag marker(s).

- capstone_project_sim.launch.py launches Gazebo simulator with capstone_project.sdf, the robot_state_publisher.launch.py, the darknet_ros.launch.py with capstone_sim.yaml configuration, the apriltag_node with tags_36h11.yaml configuration, the capstone_project_sim.py script as well as an RViz window to visualize the camera feed, laser scan and odometry estimates as well as a relative transformation between robot and AprilTag marker(s).

- wall_following_real.launch.py launches the wall_following_real.py script as well as an RViz window to visualize the camera feed, laser scan and odometry estimates as well as a relative transformation between robot and AprilTag marker(s).

- obstacle_avoidance_real.launch.py launches the obstacle_avoidance_real.py script as well as an RViz window to visualize the camera feed, laser scan and odometry estimates as well as a relative transformation between robot and AprilTag marker(s).

- line_following_real.launch.py republishes incoming compressed images (used for reducing on-board computational burden significantly) on image/compressed topic to image/uncompressed topic as uncompressed images, launches the line_following_real.py script as well as an RViz window to visualize the camera feed, laser scan and odometry estimates as well as a relative transformation between robot and AprilTag marker(s).

- stop_sign_detection_real.launch.py republishes incoming compressed images (used for reducing on-board computational burden significantly) on image/compressed topic to image/uncompressed topic as uncompressed images, launches the darknet_ros.launch.py with capstone_real.yaml configuration, the stop_sign_detection_real.py script as well as an RViz window to visualize the camera feed, laser scan and odometry estimates as well as a relative transformation between robot and AprilTag marker(s).

- apriltag_tracking_real.launch.py republishes incoming compressed images (used for reducing on-board computational burden significantly) on image/compressed topic to image/uncompressed topic as uncompressed images, launches the apriltag_node with tags_36h11.yaml configuration, the static_transform_publisher to define relative transform between camera and robot, the apriltag_tracking_real.py script as well as an RViz window to visualize the camera feed, laser scan and odometry estimates as well as a relative transformation between robot and AprilTag marker(s).

- `capstone_project_real.launch.py` republishes incoming compressed images (used for reducing on-board computational burden significantly) on `image/compressed` topic to `image/uncompressed` topic as uncompressed images, launches the `darknet_ros.launch.py` with `capstone_real.yaml` configuration, the `apriltag_node` with `tags_36h11.yaml` configuration, the `capstone_project_real.py` script as well as an RViz window to visualize the camera feed, laser scan and odometry estimates as well as a relative transformation between robot and AprilTag marker(s).

4.6. Building Instructions

Following are the step-by-step building instructions:

1. Make a directory `ROS2_WS` to act as your ROS2 workspace.

   ```
   $ mkdir -p ~/ROS2_WS/src/
   ```

2. Clone our repository:

   ```
   $ git clone https://github.com/Tinker-Twins/Autonomy-Science-And-Systems.git
   ```

3. Move `capstone_project` directory with required ROS2 packages to the source space (`src`) of your `ROS2_WS`.

   ```
   $ mv ~/Autonomy-Science-And-Systems/Capstone\ Project/capstone_project/ ~/ROS2_WS/src/
   ```

4. [Optional] Remove the unnecessary files.

   ```
   $ sudo rm -r Autonomy-Science-And-Systems
   ```

5. Build the packages.

   ```
   $ cd ~/ROS2_WS
   ```

   ```
   $ colcon build
   ```

6. Source the `setup.bash` file of your `ROS2_WS`.

   ```
   $ echo "source ~/ROS2_WS/install/setup.bash" >> ~/.bashrc
   ```

   ```
   $ source ~/.bashrc
   ```

4.7. Execution Instructions

Following are the execution instructions for simulation deployments:

1. Capstone Project:

   ```
   user@computer:~$ ros2 launch capstone_project capstone_project_sim.launch.py
   ```

2. Wall Following:

   ```
   user@computer:~$ ros2 launch capstone_project wall_following_sim.launch.py
   ```

3. Obstacle Avoidance:

```
user@computer:~$ ros2 launch capstone_project obstacle_avoidance_sim.launch.py
```

4. Line Following:

```
user@computer:~$ ros2 launch capstone_project line_following_sim.launch.py
```

5. Stop Sign Detection:

```
user@computer:~$ ros2 launch capstone_project stop_sign_detection_sim.launch.py
```

6. AprilTag Tracking:

```
user@computer:~$ ros2 launch capstone_project apriltag_tracking_sim.launch.py
user@computer:~$ ros2 run capstone_project apriltag_teleop
```

Following are the execution instructions for real-world deployments:

1. Connect to the TurtleBot3 SBC via Secure Shell Protocol (SSH):

```
user@computer:~$ sudo ssh <username>@<ip.address.of.turtlebot3>
user@computer:~$ sudo ssh ubuntu@192.168.43.48
```

2. Bringup TurtleBot3:

```
ubuntu@ubuntu:~$ ros2 launch turtlebot3_bringup robot.launch.py
ubuntu@ubuntu:~$ ros2 launch v4l2_camera camera.launch.py
```

3. Capstone Project:

```
user@computer:~$ ros2 launch capstone_project capstone_project_real.launch.py
```

4. Wall Following:

```
user@computer:~$ ros2 launch capstone_project wall_following_real.launch.py
```

5. Obstacle Avoidance:

```
user@computer:~$ ros2 launch capstone_project obstacle_avoidance_real.launch.py
```

6. Line Following:

```
user@computer:~$ ros2 launch capstone_project line_following_real.launch.py
```

7. Stop Sign Detection:

```
user@computer:~$ ros2 launch capstone_project stop_sign_detection_real.launch.py
```

8. AprilTag Tracking:

```
user@computer:~$ ros2 launch capstone_project apriltag_tracking_real.launch.py
```

5. LEARNINGS AND REMARKS

5.1. Challenges Faced

The most significant challenge we faced during the demonstration and occasionally throughout the semester was pertaining to the ROS 2 DDS communication framework. This included the fact that the domain ID is not robust to network traffic fluctuations and if multiple devices are connected/discoverable, topics/nodes are no longer discoverable over the network (including both Ethernet and Wi-Fi, although the latter is more probable). The problem of wireless overlap with limited (3) number of distinctly available channels further intensifies this issue over a limited bandwidth of 2.5 GHz that we had. To further add to this, conflicting quality of service (QoS) profiles for sensors (requiring best effort communication) and actuators (requiring reliable communication) is something that causes the topics to freeze and messages to be dropped intermittently for prolonged durations of time. It is worth noting that these issues are more specific with the hardware-software configuration of Raspberry Pi + Remote PC + ROS 2 Foxy for reasons unknown (which was unfortunately our case) and there are a multitude of open issues on community forums pertaining to these issues. However, since ROS 2 Foxy has a scheduled end of life (EOL) in May 2023, further efforts to rectify these issues seem to have been discontinued. Also, on a related note, migration from Fast DDS (as in case of ROS 2 Foxy) to Cyclone DDS is observed with future ROS 2 distros (e.g. ROS 2 Humble).

(a) (b)

Figure 32. *Networking issues: (a) general issue with wireless networking [link], which becomes more prominent with DDS communication framework; and (b) Unresolved issue of ROS 2 Foxy (Fast DDS) nodes and topics becoming undiscoverable with wired (Ethernet) or wireless (Wi-Fi) communication [link].*

Additionally, following were some more challenges that we faced while implementing the required modules of this capstone project.

- Unavailable ROS 2 drivers, packages and lack of resources for getting started and troubleshooting.
- ROS 2 TurtleBot3 URDF is much larger in size (inflated) as compared to ROS 1 URDF and even the physical (real-world) robot therefore incorporating it with same world posed issues such as robot getting stuck in tight gaps.
- Gazebo world and texture setup for ROS 2 was something that we had to do on our own (files provided by TAs were unusable) but we took the opportunity and fixed the "wobbly obstacle" bug in the original files provided by TAs.
- Gazebo occasionally results in residual errors and crashes, for which the "known-to-work" troubleshooting tips have been added to README.md.
- Practical considerations for sim2real transition are non-exhaustive, however, prominent tips pertaining to this have been added to README.md.

- Automatic mode switching leaves very little room for fail-safe mechanisms (e.g. difference in a line ending vs. the robot wearing off the line is very difficult to perfectly classify and account for).
- As a personal choice, we wanted to avoid giving the robot access to privileged information (e.g. map of environment, fiducial markers of different family/ID to switch behavioral mode, etc.) or equipment (e.g. better/redundant sensors, flashlights, etc.) so this was more of an challenge we created for ourselves since we wanted to really apply the core concepts of algorithm robustification, which was the major focus of this course.

5.2. Practical Considerations

We summarize some of the practical considerations we took into account while working on this project:

- With TurtleBot3, it is either possible to subscribe to sensor data or publish actuator commands (but NOT both) when interfaced with ROS-2 over a single Quality of Service (QoS) profile of the underlying Data Distribution Service (DDS) or Real-Time Publish Subscribe (RTPS) implementation (also, ROS-2 supports multiple DDS/RTPS implementations). Hence, we have created different QoS profiles for subscribing to sensor data (using "best effort" communication and small queue depth) and publishing actuator commands (using "reliable" communication and relatively large queue depth).

- Laser scan sector FOV and image processing thresholds will be different for simulation and real-world deployments since Gazebo simulator does not capture "truly" realistic characteristics of the sensors and rendering of the environment; not to mention that simulated world is "perfect" and comprises only walls, obstacles, line, stop sign and AprilTag, whereas real world comprises various other objects/people with different materials and lighting conditions.

- PID controller gains will be different for simulation and real-world deployments since Gazebo simulator does not capture "truly" realistic dynamics of the robot and its environment.

- Although uncompressed image transport might be acceptable for simulation, compressed image transport is highly recommended for real-world deployments in order to reduce on-board computational burden and over-the-air image transport latency significantly.

- Initial (first-instance) data availability on subscribed topics is verified (recursion is observed to be a computational burden), however, intermittent data losses and undiscoverable topics pose major issues in wireless ROS-2 framework (specifically due to stringent QoS requirements for sensors and actuators).

- Robot motion constraints (`max_lin_vel = 0.22 m/s` and `max_ang_vel = 2.84 rad/s`) need to be imposed on the controller to ensure bounded control input publication (especially for simulations since Gazebo simulator does not impose any such constraints on the robot).

We hope that these considerations will be useful for the readers who wish to recreate this project or implement a similar project.

5.3. Troubleshooting Tips

We summarize some of the troubleshooting tips we discovered while working on this project:

- In case Gazebo simulator crashes, hangs or ceases to launch properly, try killing all instances of `gzserver` and `gzclient` using the following command:

  ```
  user@computer:~$ killall gzserver and killall gzclient
  ```

- Problem of unavailability of data on topics or undiscoverable nodes itself (mostly the case with distributed-networked real-world deployment) can be potentially tackled by using a dedicated network isolated of any other open access points in proximity (to avoid wireless overlap and interference from other networks or devices).

We hope that these tips will be useful for the readers who wish to recreate this project or implement a similar project.

6. SUPPLEMENTAL MATERIALS

All source files used for implementation of various assignments (including mini-assignments) and the capstone project are openly available on GitHub. Following is a link to our repository: https://github.com/Tinker-Twins/Autonomy-Science-And-Systems

Here, apart from basic information regarding the course, group name and group members, we have added collapsible sections highlighting the high-level outcomes of each milestone (i.e. mini-assignment or assignment or project). Furthermore, a dedicated directory for each of the milestones has a detailed markdown file (README.md) of its own.

Particularly for the capstone project, all source files can be accessed at: https://github.com/Tinker-Twins/Autonomy-Science-And-Systems/tree/main/Capstone%20Project

In terms of documentation, this directory includes setup instructions, description of all nodes and launch files, practical considerations for sim2real transfer, troubleshooting tips, list of dependencies, building instructions, execution instructions (for simulation as well as reality), and finally, the results and outcomes.

Although the primary aim of using a Git repository for this project (as well as other assignments of this course) was to enable version control and collaborative development, this also gave us the opportunity to document all the progress using the markdown files (README.md). Despite challenges and scarcity of time, we invested a significant amount of time and effort in building and maintaining our repository throughout the duration of this course (and project). We hope that this repository will help the professor, the teaching assistants (TAs) as well as the reviewing committee in better understanding our efforts, outcomes and learnings from this project (and the overall course in general) and potentially serve as a documentation for us (or others) when trying to explore similar approaches. That being said, we completely respect the academic integrity moral code and as such in no way promote direct use of any part of our repository (as a whole or in part) by current and/or future students taking this course.