Case Study - Autonomous Systems : Cartesian Robot and Small World

Raniel Maranan
Master of Engineering: Artificial
Intelligence for Smart Sensors
and Actuators
Deggendorf Institute of
Technology
Cham, Germany
raniel.maranan@stud.th-deg.de
Manojkumar Mohankumar
Master of Engineering: Artificial
Intelligence for Smart Sensors
and Actuators
Deggendorf Institute of
Technology
Cham, Germany
manoj.mohankumar@stud.th-
deg.de
Dalmeet Singh
Master of Engineering: Artificial
Intelligence for Smart Sensors
and Actuators
Deggendorf Institute of
Technology
Cham, Germany
dalmeet.singh @stud.th-deg.de

Chethan Srinivasareddy
Master of Engineering: Artificial
Intelligence for Smart Sensors
and Actuators
Deggendorf Institute of
Technology
Cham, Germany
chethan.srinivasareddy
@stud.th-deg.de

Abstract— **In this case study, the group studies the coordination system method of using the Robotic operating system (ROS) for operating a TurtleBot in Gazebo. The main objective of the group is to develop a turtlebot 3 navigation system that can park in a predetermined location in the new world which was supposed to be the final task.The real time data which the team accessed with the help of cameras or sensors are used for Gmapping technique in order to do SLAM navigation. This paper also gives an overview of the Gazebo system modeling method and also gives details about the outcomes and test results. In addition to that the team must do the task includes developing a URDF file for the cartesian robot in SolidWorks(Task1), SLAM navigation (Task2) developing a turtlebot navigation with a parking system using the parking feature in the Gazebo (Task3). The team used a python script for the final task (Task 4) which uses the idea of a coordinate system for parking.**

Keywords— **Turtlebot3, SLAM navigation, Autonomous Navigation, Gazebo, ROS, SolidWorks, URDF**

I. INTRODUCTION M

Traffic-related accidents cause the terrible loss of 1.2 million lives annually. The tragic fact that car crashes take two lives on average every incidence tells the story of how often they occur [1].In addition to the enormous death toll, millions of people are dealing with injuries in the wake of the disaster, some of which result in permanent disability. Many of these accidents are the result of avoidable human error, such as weariness or carelessness with traffic regulations.

Despite this dreadful reality, a solution remains in self-driving vehicles. Automated robots driving in place of human drivers could be a major solution for minimizing these kinds of accidents. Technology must be able to minimize human errors for cars to be operated accurately, without human error or fatigue.

But making the dream of driverless vehicles a reality is challenging. Despite significant advancements in specific fields like traffic sign identification and vehicle navigation [2-4], there remains a significant research gap. A complete plan to integrate these developments into a self-controlling decision-making system is certainly missing, considering the complete research performed in these specific areas. Even though research has been done in these specific areas, one noticeable thing is the absence of an entire plan that combines these advancements into an autonomous decision-making framework.

Computer vision research, which has demonstrated impressive accuracy in processing visuals and identifying traffic signs, is an essential part of this project. Unfortunately, subsequent manages and their impact are not given sufficient attention to the present attention-getting on the recognition and detection stage. It is not enough simply to see the signs; we have to understand them and make judgments based on this understanding.

The unknown issue represents how to bring together these parts into a single platform that can navigate and identify traffic signs and make decisions on its own. By completing this paper, we could be on the edge of a period in which everyone uses safer, more secure methods of transportation, and tragedies driven by human mistake become an issue of the past.

In addition to these tasks, the team designed a Cartesian robot and converted it into a URDF file. The file stored in the URDF format is suitable for the Gazebo simulation. These integrations give the group detailed knowledge about robotic design and control.

II. Requirments

For the case study of autonomous systems, the team was assigned many tasks to complete so the team can have a good understanding about ROS and a TurtleBot. The main objective of the case study is to perform different task within the ROS software, the team had to create their assigned robot and import into gazebo, simulate SLAM, Navigation, Parking, generate an assigned world into gazebo and perform a parking simulation.

With the case study, the group wants to accomplish these requirements:

- Download and install all software needed (Ubuntu, ROS Noetic).
- Understand and perform how to generate and import worlds or models into gazebo.
- Understand and perform different simulations useable in ROS like SLAM Mapping and Navigation.
- Understand and perform different simulations of Autorace World within ROS like camera and lane calibrations, sign detection and parking.
- Combine all the tasks that was learned to perform the final task of generating assigned world and performing parking simulation.

When considering the requirements for the system, constraints were some a factor that the group must implement:

- Using ROS Noetic
- Design cartesian robot
- Perform simulation in small world that was assigned to the team.

Based on the concept of the system requirements it is explained to allow a better understanding of the functionality and concept.

A. *High-Level Overview R*

The case study for autonomous systems involves many systems and processes that will allow the group to successfully complete the case study. With high level overview, the group will be explaining those systems and processes. It will allow the readers to have a good perspective, understanding about the key components and functionality regarding the case study. With 3 main tasks, the group will be focusing on cartesian robot, ROS simulation and small world – parking simulation.

Talk about downloading packages

1) *Cartesian Robot*

When designing the groups assigned robot, the group first had to understand the functionality of the robot, and the key features the robot has compared to others. Cartesian robots are a type of robot that has 3 linear joints that use the cartesian coordinates of X, Y and Z. Typically, these robotic movements are only done on 1 axis and would execute a sliding motion. The group designed their robot with one of the bodies of the robot only sliding in the respected axis and cannot rotate.

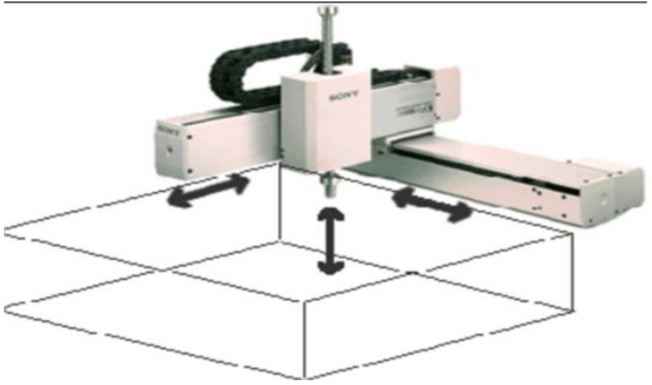


Figure 1 - Cartesian Robot

When it came to designing the robot, the group was able to use the software of SolidWorks which is a 3D design software that has many capabilities for any designers wishes. SolidWorks also worked hand in hand with converting the model into a URDF file which then will be used to generate in the ROS software using Gazebo.

URDF is a useful tool because it helps to model robotic systems which can be used to simulate and work in ROS. When converting the models to URDF, the group must assign joints and linkages from the robot. This will allow the system to know the specific movements the robot must perform; those movements could include the direction and type of motion required. For the groups case study, the movement would be the sliding motion on the axis.

Generating the cartesian robot model into gazebo was simple because all the work was making the requirements in the SolidWorks to URDF file and then when importing into

Gazebo, the group had to call up or open the URDF file into gazebo.

2) *ROS Simuluation*

When performing the simulation tasks in ROS, the tasks involved some features like generating a world or robot, SLAM and Navigation. Those are the 3 main components of performing a navigation simulation within ROS. Having the group install certain packages for ROS, TurtleBot and Simulation will help obtain certain scripts to successful complete the simulations. A lot of the actions involve using the terminal window and calling up on certain files that will help perform certain tasks.

When generating a world or robot into Gazebo, the user would use the command of roslaunch and then use the file name to open a world or robot into Gazebo. Within the packages that were installed by the group members, launch files has already been created and is easily accessible. Regarding the other features like performing SLAM and Navigation, it is similar to opening a world in Gazebo. Similar to launch files, files have been created within the packages and members of the team would have to call up certain tasks with the use of roslaunch.

Using SLAM is a very straight forward task. SLAM stands for Simultaneous Localization and Mapping. SLAM involves the robot creating a map of its surrounding and simultaneously localizing itself within the environment. It will achieve all those processes by using a software in ROS called RVIZ. Where it will display the surrounding of the robot based on the sensor reading and a lot displaying where the robot is located. When a robot is first introduced into a new environment, the environment will not all appear at once especially with all the details of the environment. Members of the group is required to use tool called teleportation which allows the members of the group control the robot within the environment with the use of the computers keyboard. Once a member of the group is controlling the robot, the member must move the robot around the environment so the robot can construct a map of the new environment with the use of an algorithm called GMapping. The key is to move to every corner, spaces where there are obstacles or object to capture every detail with the use of the robots' sensors.

Navigation is one of the last tasks to use after constructing a map using SLAM then generating the map into gazebo. Navigation contributes to the autonomous movement the robot will perform. There are two tools the members used, which are 3D initial pose which allows the robot to calibrate its location and have its sensors aware of the surrounding. 3D navigation where the user will choose a point within the map with an arrow pointing at the direction the robot must be facing. This feature will allow the robot to move autonomously without the help of any keyboards, the robot will then perform path planning and obstacle avoidance. Where path planning will create the right path from the generated map and at the same time use sensors such as depth cameras to detect any objects.

3) *Small World – Parking Simulation*

The parking simulation task is very similar idea as performing the navigation but with an extra feature of setting a parking position and performing the parking simulation.

Using the repository packages from the github page of worlds [1], the group was able to obtain the small world models and world files to generate the world into Gazebo. The primary components on generating the world were to clone the github files and then adding the model's directory to Gazebo model path. To add the model path, the members of the group had to add a line of commands that will export the world into the gazebo and change some files names within the script so the ROS system can determine where the file is located.

Regarding simulating parking, it is a straightforward concept. Where the group assigned a parking location with the help of the github page of turtlebot [2]. Which created nodes that can use the concept of navigation to reach its parking location goal. It will use the information gathered from SLAM to perform the simulation of parking is very similar to performing navigation. The biggest different is that a member of the group will be calling up a command that will automatically tell the robot to autonomously drive to the parking location by itself. Wherever the robot may be, it will use the process of performing navigation but does not require a member of the group to click a point on the map.

B. *Key Variables and Data Structures.*

1) *Cartesian Robot*

Designing a cartesian robot involved having key variables that will result in the correct concept. When the group designed the robot, the group did not have any certain restriction involving the size or parameters of the robot. Without knowing what the future of the task regarding the cartesian robot will be, so the group decided to create a simple design that can be easily modified to the respected dimensions from any specific task. The group's focus was having the robot perform a sliding motion within one axis. Which means if the robot is sliding up and down on the y-axis, that body component should not be moving in other axis unless a new body component is added.

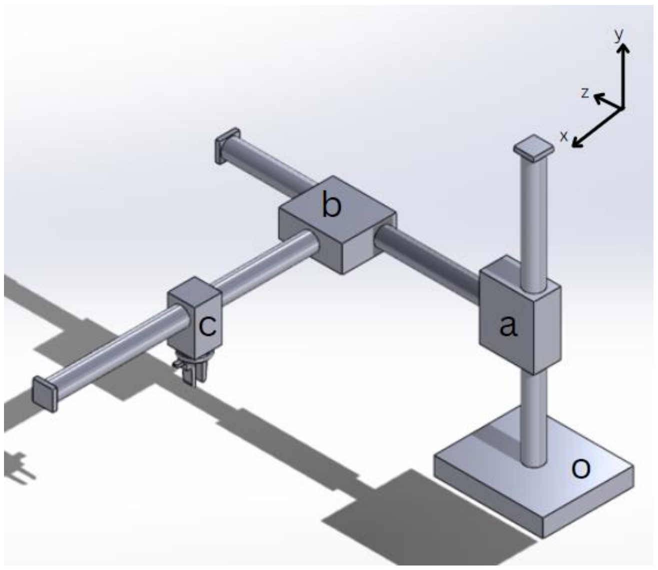


Figure 2 - Group Design of Cartesian Robot

Other than the motion, the group needed to specify certain joins and linkages within the robot. Specifying the joints and linkages will help determine the connections between different parts of the robot, how those parts will move and know the segments of the robot within the model. In Figure 2, the group has specified:

- Part O and rod O as a fixed component
- Part A and rod O is a linkage 1.
- Part A and rod A is a Joint 1.
- Part B and rod A is a linkage 2.
- Part B and rod B is a joint 2.
- Part C and rob B is a linkage 3.
- Part c and gripper is a joint 3.

Within the code, the dimensions and the information about the linkages and joints are all variables that will help model the robot within Gazebo. Designing the robot, the group assigned limits of movement for part a,b,c which will be transferred into variable when converting to URDF files. All of these are useful information because those parameters would be used to convert the design from SolidWorks to a URDF file. When considering the conversion process typically normal parameters mention above is included with material properties, meshes, geometries, and original and pose information. Once all these variables are processed, URDF file will be written as an XML format, where it will define each component and apply any of the constraints needed.

2) *ROS Simuluation*

ROS simulations requires to perform different task like SLAM and navigation. Within those task, each one will have different key varaibles, structures and charcterists to succesfully perform.

SLAM has different key variables but one of them is mapping, it involves constructing a new environment using an algorithm called Gmapping. This algorithm will help create a map of the surrounding area. Depending on the map size, issues can occur during the process because of the map being very large or having different types of terrain. This will result in a large amount of memory and a long period of processing. Another key variable are robot position and sensor measurements. These variables are considered two different key concepts but are executed simultaneity. A robot will determine or estimate the position it is in with the help of its sensor like cameras or depth sensors. This will give a proper estimation of the position. Once the robot determines its location, members of the group can visually see the robot on the map. The limitations with these two variables are the accuracy of the sensors, the quality of the map created and the physical motion of the robot.

Navigation is also like SLAM, where it has different key variables. A set of key variables include the global and local path, where it will plan out the path the robot requires to take beforehand with the help of global path and will refresh the predicted path that is left when upcoming objects or obstacles occur and this is the process of local path. If the robot is in a moving environment, then it will have difficulties determining the new path with the pace of the moving obstacles. The navigation goal is another key and important variable because it will help set the goal of where the members of the group would like the robot to move to autonomously. The limit of the variable is setting the goal of the location with the parameters of the robot and map that has been created.

3) *Small World – Parking Simulation*

Performing parking simulation will involve many of the key variables used in SLAM and Navigation but it will require additional variables like parking statues, parking information, vehicle status and control input. With regarding the parking status and information, these variables describe the location and characteristics of the parking space which can include the size and where in the world is it located but also the occupancy of the space, the maneuvers required to reach the space and processes needed when parked. Having these purposes for these variables will differentiate the performance it accomplishes to other task like navigation. At the same time, it will also come with limitations like if the parking space is the right size for the robot, is the space noticeable and the sequence that is required to approach the space acceptable.

Vehicle status is also a key variable for parking simulation because it will determine the location of the robot in the environment it is placed. Determining the position and orientation will require the robot to move towards the parking goal similar to the estimated position, global and local path. Like the other variable, the dynamic environment can result to issues on reach its parking position.

Control input is an important key variable for parking simulation or with any movement task. Regarding control inputs, it is used to determine the vehicle outputs required to complete the task like motor speed, steering angle, sensor movement, acceleration and breaking. All of the output are required to bring the robot to the desired parking position because it will use certain sequences of accelerating, braking and steering to precisely park itself in the position. Just like with any robots, there are limitations to the functionality of those movements. Where brake timing is not as quick, acceleration takes time, the motor and actuators are set to limits. Having those limitations will affect the performance of the robot.

Using the small world, isn't as different compared to performing the parking simulation in Autorace. With both worlds, the group uses pre-made packages that is required to be called up to perform. With regarding the Autorace, it uses cameras to determine the location and position of the robot because it will detect the parking signs and lanes to position itself to the designated parking spot. With the small world, the group assigned a parking position using coordinates and uses simple sensors to determine the location of the robot. The group is required to call up commands to perform the task and within those commands, the group had to open the script to change the location of the desired parking spot. Similar to the Autorace, parking in the small world and parking in the Autorace world would use the same sequences of movement, measurements and processes but just using different methods of determining itself.

III. CONCEPT

A. Algorithmic Steps ***C***

There are various steps involved in achieving this task. We have made blocks based on the relevance of the task. These blocks will give a big picture of which processes are performed.
1.Simulation setup.
2. Mapping and localization.
3.Autonomous navigation.
4.Parking simulation.

Simulation Setup
Mapping and localisation
Au

1. Simulation setup:
In this block, modelling of the robot and the simulation environment will take place. since we are using the turtlebot package, which comes with some pre-modelled robots in handy. We are importing custom worlds for simulations of robots, which are defined in URDF. This will lay the foundation for navigation in a simulated environment.
The following flow chart will provide an overview of this process.

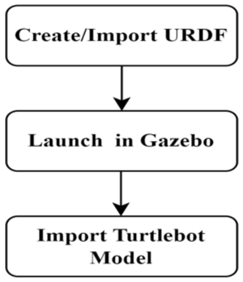


Figure 3 – *Simulation setup flowchart*

- URDF creation and import: we clone worlds from GitHub and import an assigned custom world named small_office.world to turtlebot3_gazebo.
- Launch in Gazebo: We create a launch file for the same world, and we spawn the world into the gazebo environment. using the below command
"$ roslaunch turtlebot3_gazebo offce_small.launch"
- TurtleBot model import: using the below commands, we can spawn the compatible TurtleBot model in that world. We have used the 'burger' model throughout this task.
"$ export TURTLEBOT3_MODEL=burger"

2. Mapping and localization:
In this block, Simultaneous Localization and Mapping (SLAM) is implemented using GMapping to create a map of our virtual environment, which we have imported in the above task. This is a crucial step in autonomous navigation because it will enable the robot to perceive the environment and objects and have an estimated location. So that it can use its awareness for efficient path planning and to perform autonomous navigation.
The steps below are included to achieve this task.

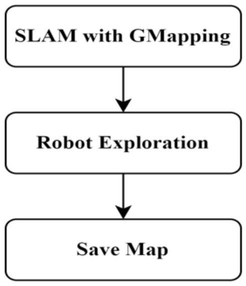


Figure 4 – *Mapping & localization flowchart*

- SLAM with Gmapping: In the new terminal, we import the same robot model and launch the SLAM node with Gmapping. Based on the captured sensory data, it will build a map. The following commands facilitate running SLAM and recorded maps.
"$ export TURTLEBOT3_MODEL=burger"
"$ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping"

- Robot exploration: To record the map, the robot must move through the corners and lengths of the world. We control the robot manually with a keyboard. To control this, we launch the teleoperation node, which enables us to control the robot through the keyboard.
  The commands are as follows,
  "$ export TURTLEBOT3_MODEL=burger"
  "$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch"
- Save map: Once we record the map thoroughly by moving the robot to each corner, we must save the map so that we can use it for locating the robot and performing autonomous navigation.
  To save the map, we use the command below.
  "$ rosrun map_server map_saver -f ~/map"

3. Autonomous Navigation:
The tasks in this block will make the robot realise the ultimate goal of autonomous navigation. Here, the robot will leverage the map generated in the previous task to locate itself on the map. We will enable the robot to realise its accurate location by overlapping the generated map with a simulated environment and locating the robot on the map. This is a pivotal step to reaching the objective of a robot having self-navigation through its surroundings.

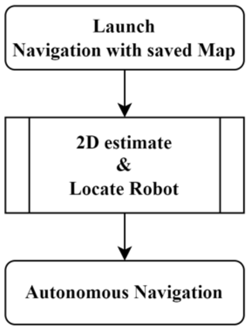


Figure 5 – *Autonomous navigation flowchart*

- Launch Navigation: To perform the navigation in the RViz environment, in the new terminal, we freshly launch small_office.world and imported TurtleBot model as well. Then launch the navigation node with the saved map. following commands will do the job.
  "$ export TURTLEBOT3_MODEL=burger"
  "$ roslaunch turtlebot3_gazebo small_office.launch"
  "$ roslaunch turtlebot3_navigation turtlebot3_navigation.launch map_file:=$HOME/map.yaml"

•

- 2D estimate and Locating robot: By using the "2D Pose Estimate" button in the RViz .Click on the map where the actual robot is located and drag the large green arrow toward the direction where the robot is facing.
  repeating until the LDS sensor data is overlayed on the saved map. To locate and move the robot we again used a teleoperation node.
- Autonomous Navigation: we can perform autonomous navigation by selecting the goal location on the map. we can do that using the "2D Nav Goal" button in the RViz menu. It tries to reach its goal by optimizing its path.

4.Automated Parking simulation:
In this block robot will perform the automated parking in existing navigation system using customized parking node that will provide the coordinated for parking in the map.
In other words, robot perform precise parking maneuvers at specified input co-ordinates.

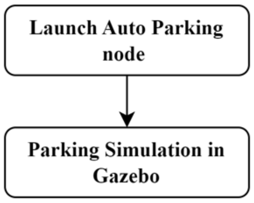


Figure 6 – *Auto-Parking flowchart*

- launch Parking node: Here we launch the auto parking node, where a robot will move to the assigned location irrespective of its current location. Here robot will identify the location-based map co-ordinates.
  "$cd ~/home/robot/catkin_ws/turtlebot "
  "$emacs -nw parking_position.py"

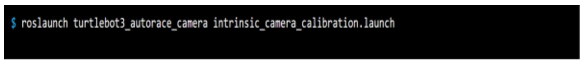
B. Key Functions/Methods ***D***

Highlight and explain any important functions or methods used in the code. Describe their purpose, input parameters, return values, and any side effects. If there are complex calculations or operations within the code, explain them step by step.

C. Dependencies and External Libraries

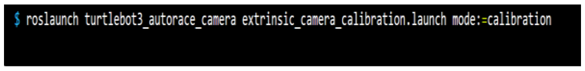
A. OpenCV

- **Calibration Algorithms**: OpenCV supports tried-and-true camera calibration algorithms such as the Tsai grid-based calibration or Zhang's method. These methods calculate intrinsic parameters which are essential to rectifying and undistorting images, like focal length, main point, and distortion of the lens factors.



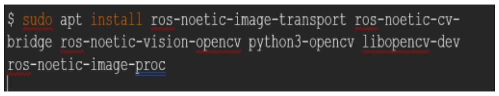
```
$ roslaunch turtlebot3_autorace_camera intrinsic_camera_calibration.launch
```

- **Estimating Extrinsic Parameters**: In multi-camera systems or when integrating cameras with other sensors that are important to determine the corresponding positions among several cameras, or extrinsic parameters. Accurate distances between cameras can be made practicable with the help of OpenCV by estimating such parameters.



```
$ roslaunch turtlebot3_autorace_camera extrinsic_camera_calibration.launch mode:=calibration
```

- **ROS Integration**: ROS nodes can directly use OpenCV features because of the OpenCV's easy compatibility, and which is made easier through the 'cv_bridge' package. This integration makes it simpler to add calibrated cameras into computer vision applications or ROS-based robotic systems via allowing real-time camera calibration.

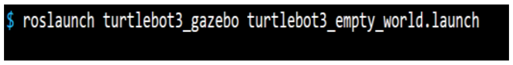


```
$ sudo apt install ros-noetic-image-transport ros-noetic-cv-
bridge ros-noetic-vision-opencv python3-opencv libopencv-dev
ros-noetic-image-proc
```

B.Gazebo

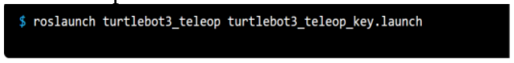
Gazebo is one of the simulation tools which the team use commonly for designing and evaluating robotic systems, which frequently works in collaboration with ROS (Robot Operating System). In addition to that robotic systems can be virtually tested and evaluated in a simulated environment and will provide an overview of how it works.

- **Simulation Environment** : Gazebo simulation environment can be used for modelling of robots, sensors, and their interactions with other objects. Thereby, the user avoids the need for actual robots allowing professionals to examine robot response in different circumstances and simulate various scenarios.



```
$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
```

- **ROS Integration:** Gazebo is one of the easiest compatibility tool with ROS which allows interaction among ROS nodes with a simulated environment.This integration can be used to communicate between controllers, robotic systems, and ROS-based algorithms which is easy in Gazebo modelling the interactions between all these components in the real world.



```
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

- **Robot model Representation:** Gazebo uses URDF (Unified Robot Description Format) or SDF (Simulation Description Format) files to represent robot designs. Thereby giving full explanations of the robot's sensors and other components. The files obtained from Gazebo are precise in simulation and display of the robot in a virtual environment.

- **Sensor Simulation**: Sensors, such as cameras, LIDAR, depth sensors, and IMUs, are able to be used in Gazebo. With this simulated environment, users can test and evaluate algorithms for processing the sensor data.

- **Testing and Validation**: The Gazebo provides a repeatable environment by which the user can test and control the system using a suitable algorithm. Moreover, it helps to reduce design and development time and also minimizes costs through enabling robotic systems to be tested and refined before they were run on actual hardware.

C. rviz

Rviz or ROS visualization is a 3D visualization tool for robots, sensors and algorithms that gives the robot perceptron in real or in a simulated world. The sensor data which we obtained from rviz is used to understand what is going on in the Robot environment. The rviz figure* is shown below

The initial command we used to launch rviz is

```
$ roscore
```

And in a new terminal tab, type

```
$rosrun rviz rviz -d
path/to/your/config_file.rviz
```

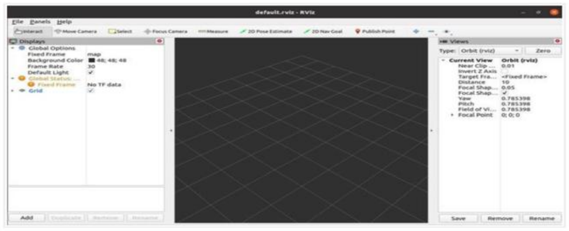


Fig* RViz

- **What is the difference between Gazebo and rviz** ?

Gazebo and rviz are the tools we used to design, develop and simulate in the ROS environment.

Rviz is a 3D visualization tool designed for debugging the sensor and robot state. With the help of a graphical interface the user can collect the datas from a 3D environment using LIDAR, sensors and depth cameras. In addition to that the user can change the values

As a advanced robotics simulator, Gazebo gives a 3-D structure for modeling robots sensors and their interactions in each environment. Its use of a physics engine allows exact modeling of components including momentum effects, and sensor results, which are crucial for refining algorithms to assess the behaviors of machines in different circumstances.

Even though the responsibilities and functions are entirely different, it supports each other in robotics: Gazebo creates virtual worlds where robotics can perform and, in the meantime, Rviz evaluates and collect the data generated by such environments. In addition to that combined functionality can accelerate forward developments in robotics and automation by enabling researchers, developers, and engineers to successfully simulate, visualize, and further develop robotic systems.

D.Solidworks

In this project the team used SolidWorks for creating a cartesian robot. It gives a way to design different parts, joints and linkages and help the group to create and simulate the actual framework of the robot. This program helps to change the design in the URDF file which is suitable for simulations in Gazebo and allows additional virtual testing and analysis.

E.External Packages

ros-noetic-joy

This package helps the user to communicate with the ROS with the help of a joystick by offering a manual control of motion.

ros-noetic-teleop-twist-joy

This package is used to convert the joysticks input into twist commands which allows a smooth and simple robot motion. It is best for situations like robot testing and exploration.

Twist-keyboard-teleop-noetic-ros:

This package makes it easier for the user to integrate the joystick with the ROS. The operational keys are W, X,A,S and D. This is one of the most important teleoperation designs which help the user to control and operate the robot.

```
Control Your TurtleBot3
Moving around:
     w
  a  s  d
     x
```

w/x : increase/decrease linear velocity
a/d : increase/decrease angular velocity
space key, **s** : force stop

ros-noetic-laser-proc:

The package is developed for analyzing laser data to help the navigation and to develop accurate maps.Theis is very much important for object detection or environmental mapping.

Ros-noetic-rgbd-launch:

It gives the robot a way to sense and respond to their surroundings by using the launch files for RGB-D cameras. This is useful in navigation over the maps with various kinds of obstacles.

Ros-noetic-amcl

The Adaptive Monte Carlo Localization (AMCL) package is essential for robot localization with various environments. It is very useful when the precise location of a robot is required like navigation in dynamic spaces.

sudo apt install ros-noetic-dynamixel-sdk:

The Super user Do or sudo can be used to run as superuser or administrator. It controls command line use for Linux based on Debian. APT will perform the action and suggest what it should install the specified package. The package which is installed is called ros-noetic-dynamixel-sdk and it is used for controlling the motors

sudo apt install ros-noetic-turtlebot3-msgs:

This command will isntall the turtlebot3- msgs package. In ROS systems use this package for communication with other components through messages.

sudo apt install ros-noetic-turtlebot:

This package will install the ros-noetic turtlebot ROS package. It includes software components related to the open-source robot platform.

IV. Result

A. *Performance Consideration* ***C***

If applicable, discuss any performance considerations or optimizations made in the code. Explain the reasoning behind those optimizations and their impact on efficiency or resource usage.

B. *Example and Usage* ***D***

Provide examples or sample inputs to demonstrate how the code is used in practice. Walk the reader through the execution of the code with specific inputs and expected outputs. This helps clarify the code's functionality and usage.

C. *Error Handling and Edge Cases* ***C***

Validation of Parking Coordinates: Will verify the parking co-ordinates whether these are within the boundaries of the map. Else throws error stating invalid co-ordinates.

Timeouts and Retries: Allowing the TurtleBot to reach the goal to within 60 sec otherwise restarting robot to move towards goal again.

Interrupt Exception: Included mechanism to shut down on user request using “CTRL+C”.

ConclU **Cartesian Robot Design:**

- What are the key variables representing the physical characteristics of the Cartesian robot (e.g., dimensions, joints, links)?
- How are these variables structured to accurately model the robot in the code?
- Are there constraints or limitations imposed on these variables?

V. Conlusion

When trying to complete the case study, the group encountered some blockages along the way. The group first had issues regarding properly running ROS and simulating some tasks. The group is having this issue because of the type of computers some of the group members are using.

. Some of the issues the group encountered with running the program was render generation in gazebo, packages install, pixelated movements, slowly start up time and software crashes. At the end, the group focused on members with ROS running smoothly and completed the task with those members’ computers.

Generating the small world also gave some problems for the group since some rendering and SLAM mapping was tricky. Without properly rendering the models for the small office into the gazebo, it will result in inaccurate SLAM Mapping. By having other members of the group also generating the small world, the group was able to have multiple maps and chose the best one.

Overall, the aim of the case study was to simulate tasks with ROS that was a success. With having the group being assigned with a cartesian robot and successfully design one on SolidWorks, convert it to URDF files and generated the robot in gazebo was the first but many accomplishments. The group was able to understand and practice different task/nodes that was useable in ROS and on TurtleBot Performing SLAM Mapping to create maps for new environments of the robot which will help to navigate the robot autonomously by performing Navigation. After, the group moved on simulating autonomous driving, where it involved camera calibration, lane and sign detection, simple driving task and finally being able to park the TurtleBot. All these tasks, was to set the group for success to complete the final task which was to create a map of a world that was assigned, perform navigation, assigned a parking position and then calling the parking position so the robot can drive autonomously to the spot. The group is able to get a lot from these case study and with accomplishing all the different types of tasks and have the knowledge about ROS, the group feels confident to use these experience in the future.

VI. Outlook **D**

References

[1] M. Peden, “World report on road traffic injury prevention,” WHO library Cataloguing in Publication Data, 2014.
World report on road traffic injury prevention (who.int)

[2] E. M. Nebot, H. D. Whyte, “A high integrity navigation architecture for outdoor autonomous vehicles,” Journal ofRobotics and Autonomous Systems, Vol. 26, No. 2-3, 1999, pp. 81-97.
https://www.scribd.com/document/91515282/Robot-Localization-and-Map-Building

[3] C. Hofner, G. Schmidt, “Path planning and guidance techniques for an autonomous mobile cleaning robot,” IEEE/RSJ/GI International Conference on Intelligent Robots and Systems, Vol. 1, 1994, pp. 610-617.
https://www.researchgate.net/publication/4065816_Combining_random_and_data-driven_coverage_planning_for_underwater_mine_detection

[4] S. Ishikawa, H. Kuwamoto, S. Ozawa, “Visual navigation of an autonomous vehicle using white line recognition,” IEEE Transaction On Pattern Analysis and Machine Intelligence, Vol. 10, No. 5, 1988, pp. 743-749.
https://www.jstage.jst.go.jp/article/jrobomech/31/2/31_212/_article/-char/ja/

APPENDIX

A. Commands

1) Simulations

PRE-REQUISITES

• Download UBUNTU 20.04 and Install on your PC

• Install ROS Noetic on Remote PC:

```
$ sudo apt update
$ sudo apt upgrade
$ wget hEps://raw.githubusercontent.com/ROBOTIS-GIT/roboJs_tools/master/install_ros_noeJc.sh
$ chmod 755 ./install_ros_noetic.sh
$ bash ./install_ros_noetic.sh
```

• Install Dependent ROS Packages:

```
$ sudo apt-get install ros-noetic-joy ros-noetic-teleop-twist-joy \
 ros-noetic-teleop-twist-keyboard ros-noetic-laser-proc \
 ros-noetic-rgbd-launch ros-noetic-rosserial-arduino \
 ros-noetic-rosserial-python ros-noetic-rosserial-client \
 ros-noetic-rosserial-msgs ros-noetic-amcl ros-noetic-map-server \
 ros-noetic-move-base ros-noetic-urdf ros-noetic-xacro \
 ros-noetic-compressed-image-transport ros-noetic-rqt* ros-noetic-rviz \
 ros-noetic-gmapping ros-noetic-navigation ros-noetic-interactive-markers
```

• Install Turtlebot3 Packages:

```
$ sudo apt install ros-noetic-dynamixel-sdk
$ sudo apt install ros-noetic-turtlebot3-msgs
$ sudo apt install ros-noetic-turtlebot3
```

• $ sudo apt-get update

• $ sudo apt-get upgrade

• Install Simulation Package:

```
$ cd ~/catkin_ws/src/
$ git clone -b noetic-devel https://github.com/ROBOTIS-GIT/turtlebot3_simulations.git
$ cd ~/catkin_ws && catkin_make
```

Launch Simulation World in Gazebo

• Run Roscore from PC: //run it everytime before bringup.

```
$ roscore
```

• Bringup Turtlebot3 in empty world: (New Terminal)

```
$ export TURTLEBOT3_MODEL=${TB3_MODEL} // use burger , waffle in place of ( ${TB3_MODEL} )
$ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch
//Turtlebot3 will launch in Gazebo
```

OR

• Bringup Turtlebot3 in world: (New Terminal)

• $ export TURTLEBOT3_MODEL=waffle // use burger , waffle in place of (${TB3_MODEL})

• $ roslaunch turtlebot3_gazebo turtlebot3_empty_world.launch

• //Turtlebot3 will launch in Gazebo

OR

• Bringup Turtlebot3 in House: (New Terminal)

• $ export TURTLEBOT3_MODEL=waffle_pi // use burger,waffle in place of (${TB3_MODEL})

• $ roslaunch turtlebot3_gazebo turtlebot3_world.launch
//Turtlebot3 will launch in Gazebo

• Launch Teleoperation Node: (New Terminal)

```
$ export TURTLEBOT3_MODEL=${TB3_MODEL} // use burger , waffle in place of ( ${TB3_MODEL} )
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

//Turtlebot3 teleop key will launch and we can control it using keyboard keys while keeping the
terminal open in one corner. If the node is successfully launched, the following instruction will be
appeared to the terminal window.

Launch SLAM Simulation

• Launch simulation world: (New Terminal)

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

• Launch SLAM Node: (New Terminal)

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_slam turtlebot3_slam.launch slam_methods:=gmapping
```

• Launch Teleoperation Node: (New Terminal)

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
```

//Turtlebot3 SLAM will launch in RViZ.
If the node is successfully launched, the following instruction will be appeared to the terminal
window.Once SLAM node is successfully up and running, TurtleBot3 will be exploring unknown area
of the map using teleoperation.Start exploring and drawing the map.

• Save Map:

```
$ rosrun map_server map_saver -f ~/map. // See the drawn map in the attached file.
```

RUN NAVIGATION Simulation

• Launch simulation world: (New Terminal)

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_gazebo turtlebot3_world.launch
```

• Launch the Navigation:

```
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_navigation turtlebot3_navigation.launch map_file:=$HOME/map.yaml
```

// This will open the previous saved map in rviz and further instructions can be performed there.

• Estimate Initial Pose:

// Click the 2D Pose Estimate button in the RViz menu.
// Click on the map where the actual robot is located and drag the large green arrow toward the

direction where the robot is facing.
// Repeat first two steps until the LDS sensor data is overlayed on the saved map.
// Launch keyboard teleoperation node to precisely locate the robot on the map.
$ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
// Move the robot back and forth a bit to collect the surrounding environment information and
narrow down the estimated location of the TurtleBot3 on the map which is displayed with tiny green
arrows.
//Terminate the keyboard teleoperation node by entering Ctrl + C to the teleop node terminal in
order to prevent different cmd_vel values are published from multiple nodes during Navigation.
• Set Navigation goal:
// Click the 2D Nav Goal button in the RViz menu.
// Click on the map to set the destination of the robot and drag the green arrow toward the direction
where the robot will be facing.
• This green arrow is a marker that can specify the destination of the robot.
• The root of the arrow is x, y coordinate of the destination, and the angle θ is determined by the
orientation of the arrow.
• As soon as x, y, θ are set, TurtleBot3 will start moving to the destination immediately.
// See the small video in the attached file.

2) Small World Parking

#launch small world
$ export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_autorace_detect my_launch_file.launch

#launch navigation
export TURTLEBOT3_MODEL=burger
$ roslaunch turtlebot3_navigation turtlebot3_navigation.launch
map_file:=$HOME/office_small.yaml

roslaunch turtlebot3_navigation turtlebot3_navigation.launch map_file:=$HOME/map.yaml

#display coordinates of robot
rosrun tf tf_echo /office_small /base_link
rosrun tf tf_echo /map /base_link

#set parking
cd ~/home/robot/catkin_ws/turtlebot
emacs -nw parking_position.py

modify line 83
position = {'x': 1.22, 'y' : 2.56}

Crtl + x
Crtl + s

#launch Parking
cd ~/catkin_ws/turtlebot
export TURTLEBOT3_MODEL=burger
python3 parking_position.py

3) Auto_race

- Install additional packages

$ sudo apt install ros-noetic-image-transport ros-noetic-cv-bridge ros-noetic-vision-opencv python3-opencv libopencv-dev ros-noetic-image-proc

Camera Calibration

1.Open a new terminal and enter
$ roscore
2. open a new terminal and enter
$ roslaunch turtlebot3_autorace_camera raspberry_pi_camera_publish.launch
3. execute rqt_image_view_on

$ rqt_image_view

Extrinsic camera

1. Open a new terminal and launch Gazebo.
$ roslaunch turtlebot3_gazebo turtlebot3_autorace_2020.launch
2. Open a new terminal and launch the intrinsic camera calibration node.
$ roslaunch turtlebot3_autorace_camera intrinsic_camera_calibration.launch
3. Open a new terminal and launch the extrinsic camera calibration node.
$ roslaunch turtlebot3_autorace_camera extrinsic_camera_calibration.launch mode: = calibration
4. Execute rqt
$ rqt

- Select plugins >visualization> Image_view.
- Select /camera/image_extrinsic_calib/compressed and /camera/image_projected_compensated

  The first selection shows an image with a red trapezoidal shape and later shows the ground projected view like a Bird's eye view

- Execute rqt_reconfigure .

  $ rosrun rqt_reconfigure rqt_reconfigure
- Adjust the parameters in camera image projection and camera image compensation projection.
  1. Change the camera image projection values. It affects camera/image_extrinsic_calib/compressed topic.
  2. Intrinsic camera calibration modifies the perspective of the image in red trapezoid.
- Updating camera calibration

  After that, overwrite each value onto the YAML files in turtlebot3_autorace_camera/calibration/extrinsic_calibration/. This will save the current calibration parameters so that they can be loaded later.

- To check the calibration results

After completing calibrations, run the step by step instructions below to check the calibration result.

1. Close all of terminal.
2. Open a new terminal and launch Autorace Gazebo simulation. The roscore will be automatically launched with the **roslaunch** command.
   $ roslaunch turtlebot3_gazebo turtlebot3_autorace_2020.launch
3. Open a new terminal and launch the intrinsic calibration node
   $ roslaunch turtlebot3_autorace_camera intrinsic_camera_calibration.launch
4. Open a new terminal and launch the extrinsic calibration node
5. $ roslaunch turtlebot3_autorace_camera extrinsic_camera_calibration.launch
6. Open a new terminal and launch rqt image viewer.
   $ rqt_image_view
7. With successful calibration settings, the bird eye view image should appear as below
   /camera/image_projected_compensated topic is selected.

Lane detection

1. Place the TurtleBot3 in between yellow and white lanes.
2. Open a new terminal and launch Autorace Gazebo simulation. The roscore will be automatically launched with the roslaunch command.
   $ roslaunch turtlebot3_gazebo turtlebot3_autorace_2020.launch
3. Open a new terminal and launch the intrinsic calibration node
   $ roslaunch turtlebot3_autorace_camera intrinsic_camera_calibration.launch
4. Open a new terminal and launch the extrinsic calibration node.
   $ roslaunch turtlebot3_autorace_camera extrinsic_camera_calibration.launch

5. Open a new terminal and launch the lane detection calibration node.
   $ roslaunch turtlebot3_autorace_detect detect_lane.launch mode:=calibration
6. Open a new terminal and launch the rqt
   $ rqt
7. Launching the rqt Image Viewer:
8. Launch the rqt image viewer by selecting **Plugins** > **Visualization** > **Image View**. Multiple rqt plugins can be run simultaneously.
9. Display three topics at each image viewer
   /detect/image_lane/compressed
10. Open a new terminal and execute rqt_reconfigure
    $ rosrun rqt_reconfigure rqt_reconfigure
11. Click **detect_lane** then adjust parameters so that yellow and white colors can be filtered
12. open lane.yml and need to modify the values. It will change the camera parameters. close the terminal of rqt_configure and detetect_lane terminal using ctrl +c
13. Open a new terminal and launch the node below.
    $ roslaunch turtlebot3_autorace_driving turtlebot3_autorace_control lane.launch

Traffic sign detection.

1. Open a new terminal and launch Autorace Gazebo simulation. The roscore will be automatically launched with the **roslaunch** command.
   $ roslaunch turtlebot3_gazebo turtlebot3_autorace_2020.launch
2. Open a new terminal and launch the teleoperation node. Drive the TurtleBot3 along the lane and stop where traffic signes can be clearly seen by the camera.
   $ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
3. Open a new terminal and launch the rqt_image_view.
   $ rqt_image_view
4. Select the camera / image compensated topic to display the camera image.
5. Capture each traffic sign from the rqt image view and crop unnecessary part of image. For the best performance, it is recommended to use original traffic sign images used in the track.
6. Save the images in the turtlebot3_autorace_detect package **/turtlebot3_autorace_2020/turtlebot3_autorace_detect/image/**. The file name should match with the name used in the source code.
7. Construction.png, intersection.png, left.png, right.png, parking.png, stop.png, tunnel.pngfile names are used by default.
8. Open a new terminal and launch the intrinsic calibration node
   $ roslaunch turtlebot3_autorace_camera intrinsic_camera_calibration.launch
9. Open a new terminal and launch the extrinsic calibration node
   $ roslaunch turtlebot3_autorace_camera extrinsic_camera_calibration.launch1
10. Open a new terminal and launch the traffic sign detection node.
    A specific mission for the ***mission*** argument must be selected among below.Interesection, construction,

parking, level_crossing, tunnel$ roslaunch turtlebot3_autorace_detect detect_sign.launch mission:=SELECT_MISSION11)

11. Open a new terminal and launch the rqt image view plugin.
    $ rqt_image_view

Parking

1. Close all terminals or terminate them with Ctrl +C
2. Open a new terminal and launch Autorace Gazebo simulation. The roscore will be automatically launched with the **roslaunch** command.
   $ roslaunch turtlebot3_gazebo turtlebot3_autorace_2020.launch
3. Open a new terminal and launch the intrinsic calibration node
   $ roslaunch turtlebot3_autorace_camera intrinsic_camera_calibration.launch

   Open a new terminal and launch the keyboard teleoperation node. Drive the TurtleBot3 along the lane and stop before the stop traffic sign
   $ roslaunch turtlebot3_teleop turtlebot3_teleop_key.launch
5. open a new terminal and launch the autorace core node with a specific mission name.
   $ roslaunch turtlebot3_autorace_core turtlebot3_autorace_core.launch mission:=level_crossing
6. Open a new terminal and launch the Gazebo mission node
   $ roslaunch turtlebot3_autorace_core turtlebot3_autorace_mission.launch
7. Run the level crossing mission by setting the mode to 2 using the below code
   $ rostopic pub -1 /core/decided_mode std_msgs/UInt8 "data: 2

B. Google Drive Link