# TURTLEBOT(s) Navigation: "Cut & Glue" Trajectories'

**A dissertation submitted in partial fulfilment of the requirements for the degree of**

**BACHELOR OF ENGINEERING in Computer Science in**

**The Queen's University of Belfast**


**Student Name:** Ryan McKee

**Student Number:** 40294886

**Project Supervisor:** Nikolaos Athanasopoulos


Date: 01/05/2024

**SCHOOL OF ELECTRONICS, ELECTRICAL ENGINEERING and COMPUTER SCIENCE**

**CSC3002 – COMPUTER SCIENCE PROJECT**

**Dissertation Cover Sheet**

**Student Name:** Ryan McKee

**Student Number:** 40294886

**Supervisor:** Nikolaos Athanasopoulos

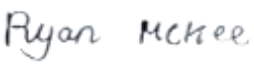
**Declaration of Academic Integrity**

Before submitting your dissertation, please check that the submission:

1. Has a full bibliography attached laid out according to the guidelines specified in the Student Project Handbook
2. Contains full acknowledgement of all secondary sources used (paper-based and electronic)
3. Does not exceed the specified page limit.
4. Is clearly presented and proof-read.
5. Is submitted on, or before, the specified or agreed due date. Late submissions will only be accepted in exceptional circumstances or where a deferment has been granted in advance.
6. If selected as an exemplar, I agree to allow my dissertation to be used as a sample for future students. (Please delete this if you do not agree.)

By submitting your dissertation you declare that you have completed the tutorial on plagiarism at http://www.qub.ac.uk/cite2write/introduction5.html and are aware that it is an academic offence to plagiarize. You declare that the submission is your own original work. No part of it has been submitted for any other assignment and you have acknowledged all written and electronic sources used.

*Student's signature:*



*Date of submission:*

May 1st 2024

# Abstract

This dissertation delves into the intricate implementation of autonomous navigation systems within the ROS2 framework, specifically targeting the Turtlebot3 platform in both simulation and real-world environments, with a primary emphasis on path planning. Comprehensive exploration of ROS essentials, including setup procedures, Turtlebot3 functionality, and an in-depth analysis of the Navigation2 stack, sets the foundation for this research endeavour. Throughout the dissertation, a meticulous focus is placed on path planning algorithms such as A*, Dijkstra, and RRT. These algorithms are not only implemented but also rigorously tested, evaluated, and seamlessly integrated into the Navigation2 stack to enhance robotic motion planning capabilities. The study significantly contributes to the advancement of ROS2 navigation by providing valuable insights into leveraging the ROS framework for robotic applications. By combining theoretical exploration with practical implementation, this research offers a comprehensive understanding of autonomous navigation systems, thereby paving the way for innovative developments in the field of robotics.

## Acknowledgements

I extend my sincere gratitude to the individuals and resources that have contributed to the completion of this dissertation. Firstly, I express heartfelt appreciation to my supervisor, Nikolaos, whose invaluable guidance, and support steered me in the right direction throughout this project.

I am deeply thankful for the myriad platforms and resources that aided me in gaining a comprehensive understanding and overcoming the complexities encountered in working with the Robot Operating System (ROS), Linux, Python, and robotic systems hardware, particularly during the construction of the TurtleBot. The assistance provided by the ROS community, especially in resolving niche issues and comprehending ROS2 and rclpy programming, has been instrumental. I am indebted to The Construct for their insightful resources on ROS programming, and to Anis Koubaa for his Udemy courses, which proved invaluable in enhancing my understanding of ROS theoretical fundamentals and navigation principles.

# Table of Contents

# Table of Figures

# 1. Introduction and Problem Area

The pursuit of autonomous navigation within dynamic environments stands as a pivotal frontier in the ever-evolving landscape of robotics. At its core, this pursuit hinges on the efficacy of planning algorithms, which orchestrate the intricate dance of guiding robots through complex terrains and obstacles [1]. However, translating the theoretical underpinnings of these algorithms into practical solutions for real-world robots, such as the TurtleBot3, reveals a landscape fraught with challenges.

This dissertation embarks on a journey to contribute to the development of a TurtleBot3 robot capable of navigating flat indoor terrains with precision and efficiency. Central to this endeavour is the exploration of a diverse array of path planning algorithm solutions, ranging from graph-based methods like Dijkstra to heuristic models such as A* and probabilistic algorithms like RRT. Each algorithm encapsulates its own set of intricacies, from dynamic obstacle avoidance to accounting for the dimensionality and kinematics of the robot.

Key challenges addressed in this pursuit include localization, mapping, path planning, and seamless integration within a cohesive navigation stack. These challenges necessitate a multidisciplinary approach, spanning computer science, mathematics, artificial intelligence, networking, robotics, and control theory. To navigate this complex landscape efficiently, the dissertation leverages the open-source robotics middleware suite, ROS (Robot Operating System).

Through a combination of experimentation within the ROS Gazebo simulation environment and validation on a real-world TurtleBot3 platform, the proposed solutions will be rigorously tested. This collaborative approach aims to bridge the gap between theoretical exploration and practical implementation, offering valuable insights into the realm of ROS-based robotics and motion planning.

In summary, this dissertation represents a comprehensive exploration of the multifaceted journey undertaken throughout the project. By meticulously documenting the research, challenges, solutions, and implementation processes, it endeavours to offer valuable insights into the nuanced domain of ROS-based robotics and motion planning. As a testament to innovation and perseverance, it aspires to serve as a guiding beacon for future endeavours in the realm of robotic navigation within the ROS ecosystem.

# 2. System Requirements and Specifications

The overarching objective of this project is to develop an autonomous TurtleBot capable of efficiently navigating indoor environments using robust motion planning algorithms within the ROS ecosystem.

## 2.1. System Requirements and Features:

Understanding Mobile Robot Dynamics and Planning Methods:

- Gain proficiency in mobile robot dynamics and various planning methodologies to inform algorithm selection and development.

Identification and Testing of Estimation/SLAM Algorithms:

- Identify and assess estimation/SLAM algorithms suitable for processing LiDAR sensor data.

Identification and Testing of Trajectory Planning Algorithms:

- Identify and evaluate trajectory planning algorithms suitable for the TurtleBot model.
- Validate trajectory planning algorithms through simulation to ensure precise motion tracking.

Development of Planning Algorithms:

- Develop planning algorithms, such as RRT, based on motion primitives and/or reachability-based methods.
- Ensure the algorithms can generate intricate trajectories for seamless navigation in indoor environments.

Assembly and Interface of TurtleBot3 with ROS:

- Assemble TurtleBot3 hardware components and establish interface compatibility with ROS for smooth integration.
- Verify communication and compatibility between TurtleBot3 and ROS components.

Conduct Experiments to Evaluate Navigation System Performance:

- Conduct a series of experiments to assess navigation system performance across varied scenarios.

## 2.2. **Success Criteria:** The success of the project hinges on achieving the following milestones:

- Successful integration of ROS components within simulation environments (e.g., Gazebo).
- Seamless integration of ROS with the real-world TurtleBot3, including successful assembly and interface.
- Implementation of advanced planning algorithms enabling navigation through complex trajectories in simulated and real environments.
- Conducting a comprehensive performance evaluation, showcasing accuracy, efficiency, adaptability, and real-time dynamic navigation strategies. By rigorously evaluating against these criteria throughout the project, I aim to develop a proficiently navigating robot tailored for indoor terrain.

# 3. Environment Setup and Architecture

## 3.1. Architecture Choice

ROS (Robot Operating System) was chosen as the middleware framework due to its open-source nature, which facilitates the development and control of robotic systems. Its standardized and modular architecture abstracts hardware specifics, enabling seamless communication among various robot components [2]. This choice aligns well with TurtleBot 3's framework, promoting efficient development and mitigating the learning curve by focusing on high-level algorithms. ROS comes in multiple iterations, with ROS 2 Humble selected for compatibility with TurtleBot packages and its advanced functionalities, including enhanced real-time and embedded systems support. This version also eliminates the communication middleware between nodes present in ROS 1 'roscore', enhancing efficiency and reliability.

Additionally, ROS contains several built-in tools that enable efficient robotic development. One such tool is Rviz2 [3], a 3D visualization tool for ROS that provides a graphical interface for visualizing sensor data, including LiDAR, and displaying robot trajectories, paths, and poses. Another indispensable feature used throughout the project development is the Gazebo [4] simulation environment, which provides a standardized way to simulate robotic behaviours and models in environments for testing algorithms. ROS2 also provides libraries that allow for programmatic interactions and creations of ROS components such as nodes, topics, services subscribers, and so on. Furthermore, several TurtleBot3-related packages were installed on top of ROS, providing the capability to launch Gazebo TurtleBot simulations with various maps containing static obstacles. These packages also include URDF files for TurtleBot models, such as the Burger model, and built-in TurtleBot SLAM nodes, along with an entire navigation stack.

In conclusion, the project's hardware and software architecture integrate a TurtleBot running Ubuntu Server with ROS2, enabling seamless communication with a remote PC equipped with ROS2 on Ubuntu Desktop. Crucial sensor data, such as LiDAR readings and velocity information, is published by nodes on the TurtleBot over the LAN to the remote PC using the ROS 2 Middleware (RMW) Data Distribution Service (DDS) pub-sub framework [5]. This framework ensures smooth communication between various ROS 2 nodes running on the same subnet, with each ROS device having its own unique ROS_DOMAIN_ID, facilitating communication among ROS instances on the subnet. The remote PC processes this data and sends navigation instructions back to the TurtleBot.

Throughout the project journey, various challenges were encountered that demanded extensive research to resolve, including driver issues and package incompatibility. In this section of the dissertation, we delve into the setup of ROS Humble, aiming to streamline the process and enhance understanding. The methodology outlined here is a result of thorough research and experimentation, heavily influenced by the documentation for Humble [6]. For further insights and information on robot setup, it is recommended to refer to it, though it's noted that several packages, including the turtlebot3_simulations, may present challenges.

## 3.2. **Virtual Machine**

Setting up ROS 2 Humble begins with installing it on Ubuntu 22.04 within a virtual machine environment. While dual booting is often recommended, I encountered several issues related to Intel drivers at the start of this project. As a workaround, I opted to use a hypervisor, specifically VirtualBox. This desktop setup offers a user-friendly development and testing environment compatible with all functionalities required for the project. You can download VirtualBox.

.

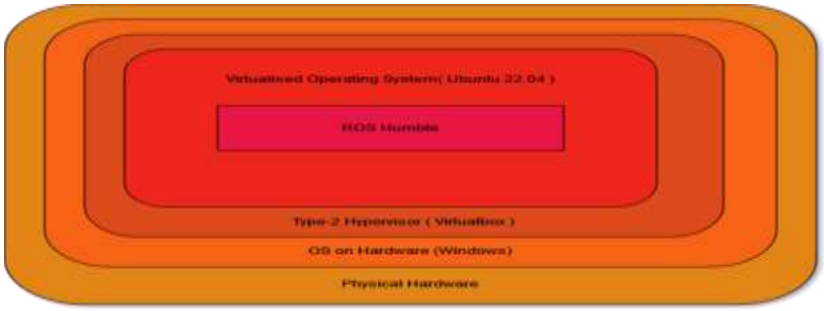



FIGURE 1: ROS VIRTUAL-MACHINE STACK DIAGRAM

When configuring your virtual machine for ROS, ensure that virtualization is enabled in your BIOS. In the network settings, create two adapters. The first adapter should be set to the default NAT connection for use when the virtual machine is on public Wi-Fi. For communication with robots like the TurtleBot over a subnet DDS as discussed above, enable a bridged connection. This allows the host machine and the virtual machine to share the same network resources, ensuring that the virtual machine is on the LAN network.

By default, virtual machines create isolated network connections, which can hinder communication between devices. Therefore, configuring the adapter connection properly is crucial. Avoid sticking with default settings; instead, research the configuration suitable for your network card to prevent issues. Below is an example configuration for the WIFI card for Intel® Wi-Fi 6E AX211 160MHz adapter.

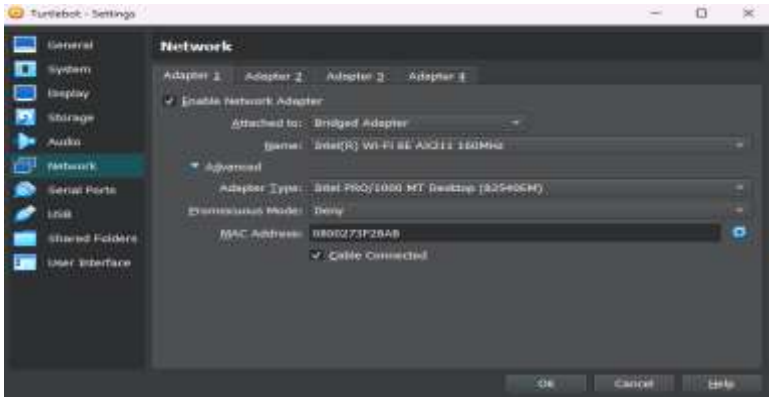



FIGURE 2: VIRTUAL MACHINE BRIDGED ADAPTER CONFIG

## 3.3. ROS2 Install and Configuration

- **Install and Configure ROS Humble Desktop and Tools:** Follow the instructions provided in the ROS documentation [8]
- **Install Development Tools:**
    - **IDE:** Download and install Visual Studio Code
    - **Text Editor:** Install Atom text editor on Ubuntu following the steps.
    - **Terminal:** Install Terminator, a Linux terminal emulator
- **Install Turtlebot3 packages:**
    - $ sudo apt install ros-humble-turtlebot3*
- **Install Gazebo (Simulation Environment for ROS):**
    - $ sudo apt install ros-humble-gazebo-*
- **Install Cartographer (SLAM Packages):**
    - $ sudo apt install ros-humble-cartographer
    - $ sudo apt install ros-humble-cartographer-ros
- **Install Navigation2 Packages:** Follow the instructions provided in the Navigation2 Getting Started Guide and input the following commands in the terminal:
    - $ sudo apt install ros-humble-navigation2
    - $ sudo apt install ros-humble-nav2-bringup
- **Configure .bashrc:** Edit the .bashrc file to set default models for the Gazebo environment and source the file:
    - $ echo TURTLEBOT3_MODEL=burger >> ~/.bashrc
    - $ echo GAZEBO_MODEL_PATH=\$GAZEBO_MODEL_PATH:/opt/ros/humble/share/turtlebot3_gazebo/models" >> ~/.bashrc && $ source ~/.bashrc


**Now ros is installed and you should be able to Launch the Navigation Stack:**

- $ ros2 launch nav2_bringup tb3_simulation_launch.py headless:=False

## 3.4. **Building a ROS2 Colcon workspace**

Colcon, short for "COnvention for Language-independent, Compiler-independent packages," serves as a pivotal command-line tool within the Robot Operating System (ROS) ecosystem, dedicated to the management and building of software projects adhering to specific organizational and structural conventions. These packages encompass diverse elements including source code, configuration files, documentation, and essential resources vital for the seamless construction and execution of software projects. Moreover, Colcon facilitates the management of dependencies, the integration of tests, and the packaging of software, thereby streamlining the development process within ROS environments.

Upon installing ROS on your system, Colcon becomes an integral component, residing typically at '/opt/ros/humble' directory. Within this directory, a constellation of packages awaits, essential for various functionalities. Among them, the 'share' package stands out, housing an array of public packages such as the prominent 'nav2' packages, representing invaluable assets for further exploration and utilization within ROS frameworks.

For further information on Colcon and its functionalities, refer to the official documentation: ()

To create your own packages, you will need to create your own ros workspace to do this is simple:

1. Create the repository: $ mkdir -p ~/ros2_ws/src && cd ~/ros2_ws/src
2. Insert and ros packages/ directories containing packages you want into the src directory.
3. Install dependencies that are required for your packages: $ rosdep install --from-paths src --ignore-src -r -y
4. Colcon build environment this will compile all code; it will also create bootstrap files in launch for sourcing in you. bashrc that will make your packages available to the rest of your ros2 env:
    - $cd ~/ros2_ws/ && colcon build --symlink-install # --symlink-install optional; It creates symbolic links to the build directory in your ros environment which is what your ros environment uses instead of copying your files there therefore you don’t have to build every time you make a change to certain code files.

5. Now source your environment in your ~/.bashrc: $ echo source ~/ros2_ws/install/setup.bash && source ~/.bashrc

You should be able to ros2 run/launch you packages in your own personal colcon environment. If you want to use the code created during this project git clone this GitLab repository into your ~/ros2_ws/src/ folder and follow the proceeding steps: ( GitLab repo: [0] )

# 4. Foundational Understanding

## 4.1. Exploring ROS (Robot Operating System) Theory

Ros is complex and has many features. To work with ROS optimally you will need some pre-requisites, Understanding of Linux and bash command line and knowledge of c++ is required if you want to work with the navigation2 stack that I will talk about later in this dissertation however for most ros programming a knowledge of python is enough to get by. In ros2 the programming library you will use to interact with in python is rclpy and when working with c++ the library will be rclcpp for further information about this check these links. When working with these libraries you will be creating nodes, services, action servers and topics and subscribers.

### ROS Nodes

Nodes in ROS2 represent individual processes, each fulfilling specific tasks like robot control, sensor data subscription, or localization. This modular approach facilitates both ease of use and the development of scalable robotic systems. In ROS2, nodes communicate with one another utilizing various methods such as the publish-subscribe message pattern, services, actions, or topics.

***Node Communication**:* For modular nodes to work in conjunction with each other they need to communicate information, there are several communication middleware types that aid in this providing slightly different communication methods which can be used for multiple reasons. The publish-subscribe model, client-service model and action messaging model for communication are used between nodes depending on node requirements.

- **Topics:** A topic is a communication intermediary where communication can take place. One or more nodes can publish a message to a topic at one time and this message will then be distributed to the nodes that are subscribed to that topic [9].

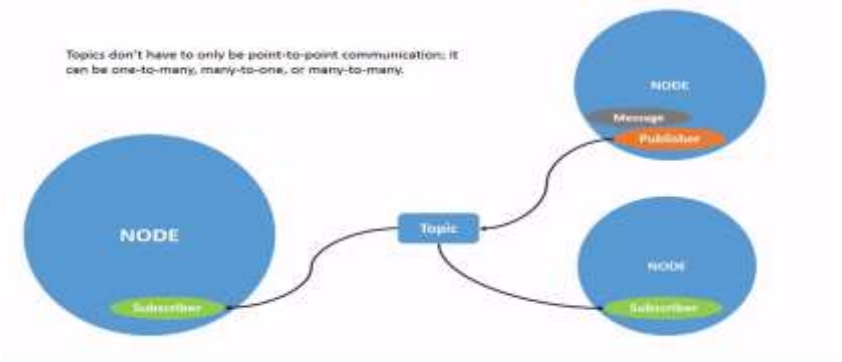



FIGURE 3: ROS TOPICS DIAGRAM [9]

- **Services:** Services provide an asynchronous communication mechanism between nodes, allowing one node to request a particular operation from another node and receive a response. Services are typically used for short-lived tasks or operations that require immediate feedback [10].

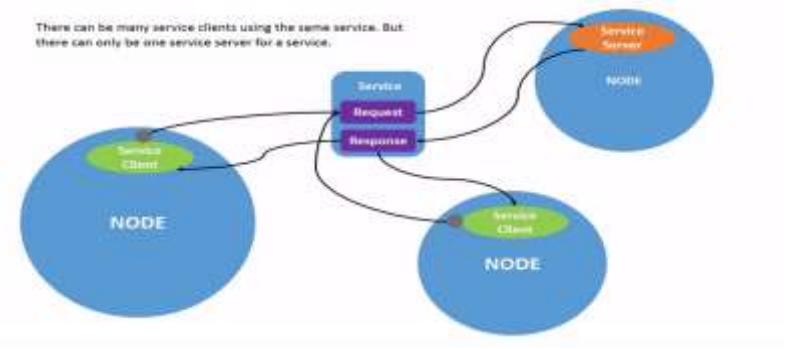



FIGURE 4: ROS SERVICE DIAGRAM [10]

- **Actions:** Actions are built on topics and services. Their functionality is like services, exception actions are preemptable (cancel while executing). They also provide steady feedback, as opposed to services which return a single response. Actions us a client-server model, like publisher subscriber model. An "action client" node sends a goal to an "action server" node that acknowledges the goal and returns a stream of feedback and a result. [11]

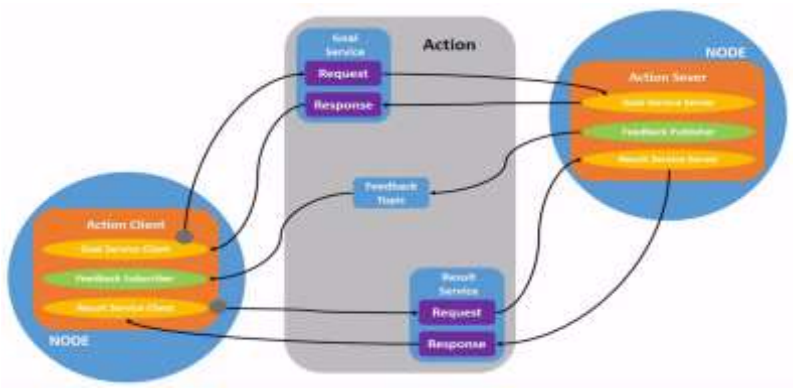



FIGURE 5: ROS ACTION DIAGRAM [11]

## ROS Interfaces

Interfaces (when creating interfaces, they cannot be made in a package with build-type ament_rclpy they need to be made in ament_cmake package): Interfaces are used to define the structure and semantics of messages exchanges between communication of nodes. They provide a clear and standardized way to specify the data exchanged between ROS 2 entities (services, topics, action servers). ( more information on ros interfaces: [12])

**Message Interfaces:** Message interfaces with file type of '.mgs' specify the fields and data types of a message that is passed to topics. 'mgs' can take several different fields which make up the message each of these fields can be a pre-defined ros2 datatype contained in the 'std_msgs' package or can contain other interfaces which contain their own individual fields.

```
# CustomMessage.msg
Int32 message_number
String message
```

**Service interfaces:** Service interfaces with file type .srv define the fields and data types of a service request and response. Again, fields are separated by spaces but the srv message has a request and response section separated by '---'

```
# CustomMessage.srv
string request
---
Int32 response_num
custom_messages_pkg/CustomMessage response
```

**Action interfaces**: Action interfaces define the goal result and feedback messages of a ros2 action these interfaces are of type .action and are split into 3 sections, goal, result and feedback and look like the following. (We won't be using this in this dissertation so no need to worry about this one)

```
# MoveRobot.action
geometry_msgs/PoseStamped goal_pose
---
boolean success
___
Geometry_msgs/PoseStamped current_pose
```

## 4.2. TurtleBot3 burger

### Real Robot

Our TurtleBot 3 is structured around a Raspberry Pi single-board computer (SBC), functioning as the central hub for control, coordination, and sensor data interpretation. It orchestrates the operations of the robot's motors, sensors, and actuators. Powered by Ubuntu 22.04 Server with ROS 2 Humble, the Raspberry Pi executes the robot control software, processing sensor data and disseminating it across the LAN via dedicated nodes running on the TurtleBot. Simultaneously, it remains responsive to commands relayed through topics.

FIGURE 6: TURTLEBOT3 IMAGE

Upon installation and activation of the TurtleBot 3 bring-up packages, the system employs Dynamixel robot actuators and LDS-01 Lidar sensor. The actuators are accessible both remotely and locally through the '/cmd_vel' topic enable seamless interaction with navigation algorithms.

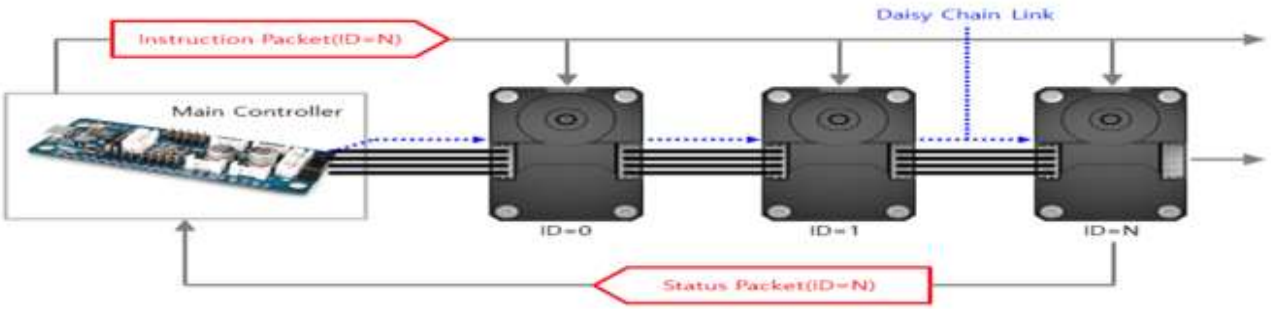



FIGURE 7: ACTUATOR DIAGRAM

When Turtlebot3 has been had it's 'bringup' script ran the TurtleBot 3 subscribes to the '/cmd_vel' topic, which necessitates messages of interface '.msg' type 'geometry_msgs/msg/Twist'. This message format consists of two Vector3 commands: linear and angular. The linear Vector3 signifies the rate of change in a robot's position over time. Conversely, angular velocity variable denotes rotational motion around a fixed point in this case the turtlebot3's central frame. Specifically, for the turtlebot3, only the x component of the linear Vector3 is utilized, defining the linear velocity in meters per second (m/s) and angular velocity is measured in radians per second (rad/s), is determined by the z argument in the velocity message. For example, angular {z = 2} denotes the robot

turning at a rate of 2 radians per second, corresponding to a full 360° rotation in a second. Similarly, linear{x=1} denotes the robot moving at a speed of 1 meter per second.

Messages can be published via this command to the /cmd_vel topic:
$ ros2 topic pub /cmd_vel geometry_msgs/msg/Twist “linear: x: 0.1 y: 0.0 z: 0.0  angular: x: 0.0 y: 0.0 z: 0.2”

These commands serve as pivotal instructions for guiding the robot, complementing its advanced sensor suite which includes a 360-degree LiDAR sensor (LSD-01), an IMU (Inertial Measurement Unit), and wheel encoders. Together, these sensors play a critical role in precisely perceiving the robot's surroundings, providing essential data for localization, and effectively addressing the challenge of determining the robot's position within its environment relative to a specific reference frame (which will be discussed further later).

The LDS-01 LiDAR sensor, the second primary topic published by the TurtleBot, is instrumental in enhancing TurtleBot3 Burger's SLAM (Simultaneous Localization and Mapping) capabilities. Functioning as a laser distance sensor, the LDS emits a modulated infrared laser while completing a full rotation. It meticulously processes the returning signals to derive precise angle and distance measurements. Operating at a rotation speed ranging between 4.83 and 5.17 revolutions per second, each degree of the sensor's 360-degree  rotation yields ray measurements in meters. This extensive dataset is disseminated to the '/scan' topic, employing the '/sensor_msgs/msg/LaserScan' interface message to ensure seamless compatibility and accessibility within the ROS suite.

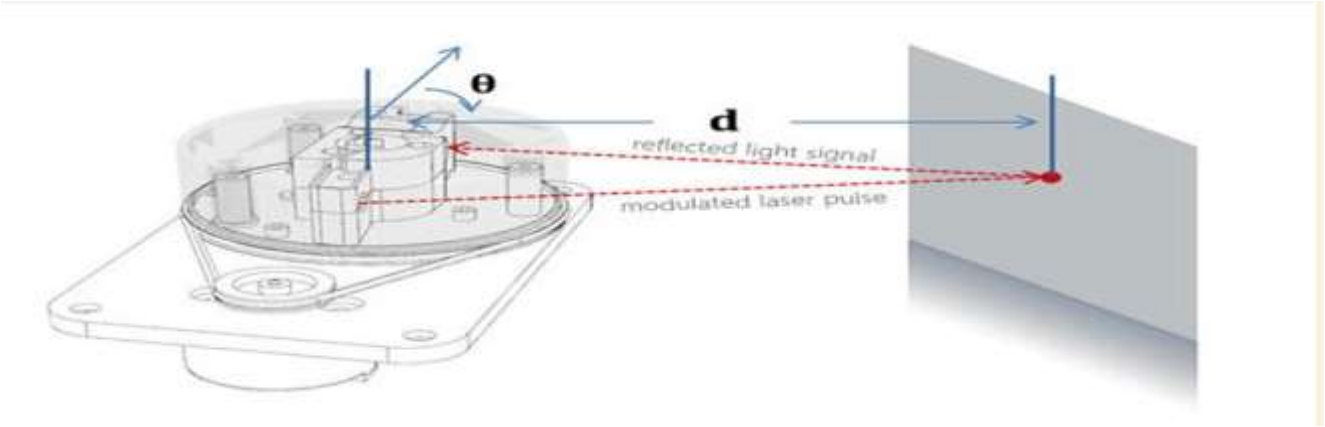



FIGURE 8: LIDAR DIAGRAM

The SBC processes data from these sensors in real-time to build maps of the environment, localize the robot within the map and avoid obstacles while navigating. SBE provides the communication allowing turtlebot to communicate with remote devices over network to exchange information with other devices or systems. The SBC provides a platform for software development and experimentation and the sbc provides a way for the how to interpret and interface with the turtlebot3 burger for the purpose of monitoring robot status configure settings and visualize sensor data or mapping results in real-time.

Our raspberry pi will contain ubuntu 22.04 server which was burned onto a micro-ssd and plugged into the raspberry pi. It will be configured to exist on the local network and allow ssh communication with remote devices as well as well as connect to Ros node's that exist on the same network will allow us to communicate with the turtlebot remotely and execute nodes that the robot will be able to communicate remote pc's that start ROS nodes.

FIGURE 9: VIDEO SHOWING TURTLEBOT3 AFTER COMPLETING ITS CONSTRUCTION

When constructing this robot, I adhered closely to the guidelines outlined by the ROBOTIS team, as detailed in their documentation([3]). It's worth emphasizing a crucial aspect of configuring the robot for network access: ensuring connection to a LAN through a router that solely demands a username and password for access. This became evident during our assembly process, particularly when confronted with the challenge posed by public Wi-Fi networks that necessitate login via a pop-up window, rendering it inaccessible to a command-line operating system.

## Simulation Robot

In our exploration of the ROS2 simulation environment, Gazebo emerges as a crucial tool for replicating real-world scenarios with exceptional fidelity. Central to Gazebo's capability is the utilization of standardized formats, namely URDF (Unified Robot Description Format) and SDF (Simulation Description Format), which define robots and environments within the simulation.

URDF, with its focus on kinematics and visual properties, is apt for simpler models and basic simulations. Conversely, SDF provides a comprehensive approach, incorporating detailed physical properties necessary for complex simulations requiring accurate physics representation.

In our study, where we focus on the Turtlebot operating in static environments, the relatively straightforward nature of both the robot and its operational context led us to opt for SDFs. These SDFs, conveniently accessible in

the GitLab project directory for Nav2 Custom Planner Bringup under the "models" subdirectory, accommodate the detailed physical properties essential for our simulation needs.

Examining the Turtlebot3 Burger model's description in XML format, we uncover several key components intricately defining the robot's structure and functionality:

1. **Model Tag**: Serving as the foundational element, this encapsulates the comprehensive description of the TurtleBot3 Burger model.
2. **Pose**: Defines the robot's initial pose, encompassing its precise position (x, y, z) and orientation (roll, pitch, yaw).
3. **Links**: Define various physical components integral to the robot's architecture, including the base footprint, base link, IMU link, laser scanner (LIDAR) sensor, wheels, and caster back link.
4. **Joints**: Articulate the connections between links and their relative movement capabilities, such as fixed joints and revolute joints.
5. **Plugins**: Augment the simulation's capabilities, enhancing the robot's behaviour, including simulating the differential drive mechanism and facilitating visualization and control.

This detailed XML file serves as a blueprint, meticulously documenting the TurtleBot3 Burger model's physical attributes, sensor configurations, and kinematic characteristics. It lays the groundwork for precise simulation of the robot's behaviour within environments like Gazebo, ensuring accuracy and reliability in virtual experimentation and development.

For those seeking further insights into building Gazebo models and leveraging its capabilities, a tutorial is available at the Gazebo Model Building Tutorial, offering valuable guidance for navigating the intricacies of simulation development.

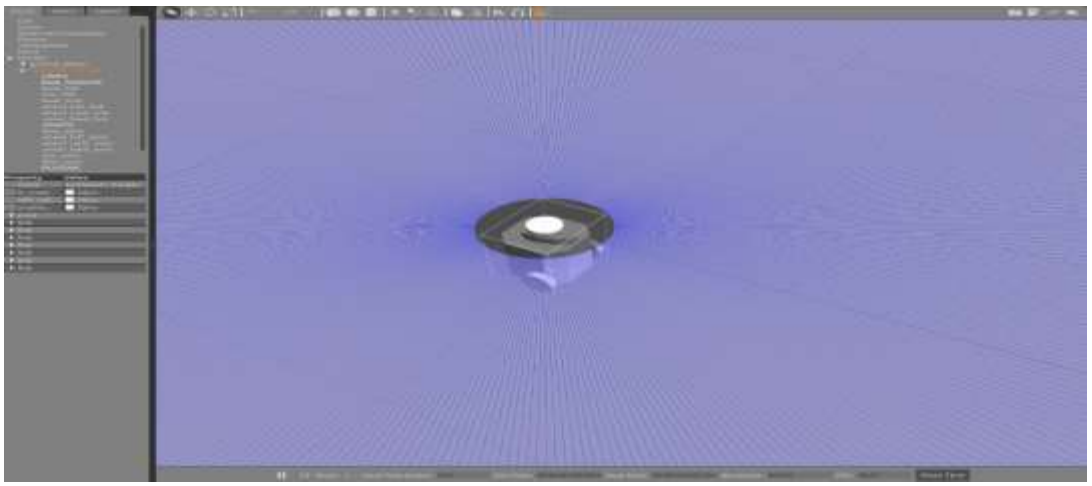


FIGURE 10: TURTLEBOT3 SIMULATION MODEL IN GAZEBO ENVIRONMENT

# 5. Autonomous Robot Navigation

## 5.1. Initial experiment: Wall Follower Algorithm

In pursuit of leveraging the foundational principles of ROS and delving deeper into pivotal concepts essential for project advancement, we embarked on a programming endeavour within both simulated and real-world environments. The primary goal was to gain profound insights into the effective utilization of various navigation tools available to the Turtlebot for use in more advanced algorithms. To achieve this, we meticulously crafted a wall follower algorithm designed to navigate a straightforward, obstacle-free environment reminiscent of a square layout. Below is a breakdown of the algorithm:

1. The robot pauses briefly to initialize.
2. It assesses the distance to the wall on its right using data from the /scan laserscan topic at a 0-degree angle.
3. If a valid distance reading is obtained:
    - If the path ahead is clear:
        - It dynamically adjusts its speed by publishing velocity commands to the /cmd_vel topic based on the proximity of the wall.
        - If the wall is distant, it makes a slight adjustment to the right.
        - If the wall is too close, it makes a slight adjustment to the left.
        - If the distance is deemed safe, it proceeds forward.
    - If an obstacle obstructs the path:
        - It halts its motion and executes a left turn to navigate around the obstacle.

This algorithm represents a foundational exploration into ROS-based robotics, meticulously integrating foundational knowledge. It serves as a demonstrative showcase of the intricate synergy among sensors, nodes, and control mechanisms, ultimately culminating in autonomous navigation within a controlled environment. Notably, the utilization of the rclpy Colcon package for deploying the wall follower node underscores the seamless integration of indispensable software components vital for enabling robotic functionality within the ROS ecosystem.

FIGURE 10: WALL FOLLOWER ALGORITHM IN ACTION: REAL-WORLD VS SIMULATION

## 5.2. SLAM (Simultaneous Localization and Mapping)

Continuing from the initial algorithm but before embarking on navigation tasks, it's essential to generate a comprehensive representation of the environment surrounding the TurtleBot3 in its environment as it serves as a foundation for navigation and allows the application of path-finding algorithms of computer understandable representations of the real world. Achieving this requires employing SLAM (Simultaneous Localization and Mapping), a crucial solution for navigating unfamiliar terrains while simultaneously creating a map of the environment.: SLAM operates through two concurrent processes: localization and mapping. [13]

**Localization**: Localization utilizes mobile robot sensors to determine its position within the environment. It primarily employs two approaches: Passive Localization and Active Localization. In Passive Localization, the robot deduces its position by continuously analysing sensor data, refining its estimate over time based on static objects in the environment. Despite its effectiveness, this method introduces randomness and uncertainty due to its non-deterministic nature. Conversely, Active Localization involves the robot following a predetermined trajectory designed to minimize localization uncertainty.

**Mapping**: Mapping refers to the process of creating or updating a map of the environment using sensor data such as LiDAR or cameras. This map-building process occurs simultaneously with localization and is pivotal for accurately estimating the robot's position and orientation within the environment. When mapping, the representation of the map can be output in several different ways, with the main one being in an occupancy grid.

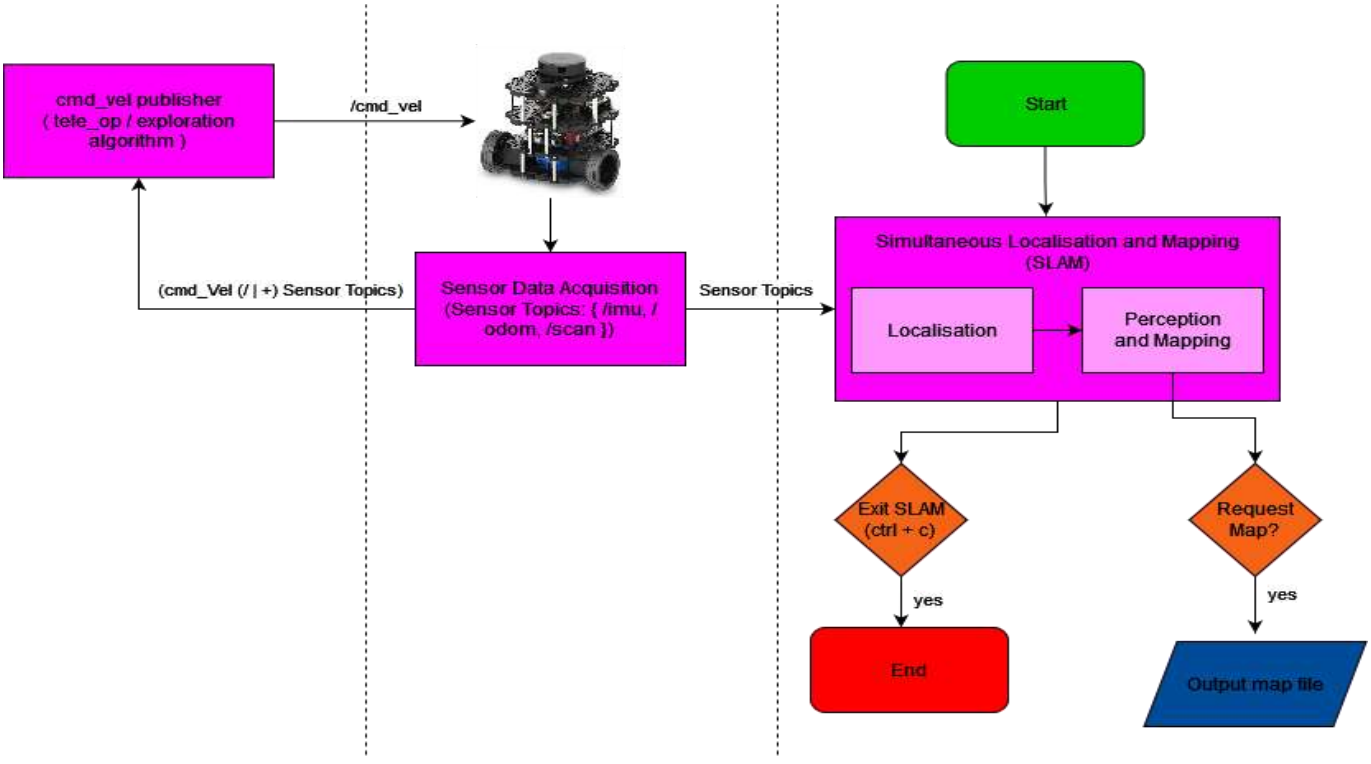



FIGURE 11: FLOWCHART OF SLAM PROCESS

**SLAM Flow Processes**

Following the flow diagram shown above there are three concurrent processes that run when slam is operating these processes are:

- **Cmd_vel Publisher**: This process involves tele-operating the robot to explore the environment autonomously or using maze-search-like algorithms, which utilize sensor data to autonomously navigate unexplored parts of the environment.
- **Sensor Data Acquisition**: This process occurs on the Raspberry Pi of the TurtleBot when the TurtleBot3 bring-up launch file has been run. It interprets information from the robotic sensors operating in real-time and publishes them to individual ROS topics. These topics include /scan for laser scan data and /odom for actuator telemetry used for localization. This process also subscribes to the /cmd_vel topic, translating linear and angular information into machine code for robot actuators.
- **SLAM Process**: The SLAM process localizes and maps the environment using algorithms like gmapping, cartographer, and others. For this dissertation, we opt for the cartographer algorithm developed by Google, as it's a highly advanced open-source SLAM system suitable for our navigation pipeline.

There are several different SLAM algorithms each having their own independent implementation of localisation and mapping techniques and a range of other techniques they may use to aid it including gmapping and cartographer. Cartographer being state-of-the-art slam algorithm developed by google and integrated with ros for easy use therefore making it the wisest choice for our slam for a combination of accuracy, robustness, efficiency, and localization tasks in robotics applications. Cartographer utilizes active localization [14], processing scan data, performing scan matching, loop closure detection, and global optimizations to localize the robot within the environment and build an accurate map. Its phases include:

- **Data Acquisition:** Cartographer collects sensor data, including LiDAR, IMU, and odometry, to understand the environment and the robot’s motion.
- **Feature Extraction:** Relevant features are extracted from sensor data to represent distinctive points in the environment, serving as landmarks for localization and mapping.
- **Scan Matching:** Cartographer aligns consecutive sensor readings to estimate the robot’s trajectory as it moves through the environment, finding the best match between current sensor data and the previously built map.
- **Loop Closure Detection:** Cartographer detects loop closures, improving map consistency and accuracy by correcting accumulated errors in the robot trajectory when revisiting previously mapped areas.
- **Global Optimization:** After loop closures are detected, Cartographer refines the map and robot’s trajectory through global optimization, ensuring consistency and accuracy across the entire map.

- **Map Representation:** Cartographer builds and maintains a map using collected sensor data and the estimated robot trajectory, which may include occupancy grids, point clouds, or 2D/3D representations. In our case, we utilize occupancy grids.

### Cartographer SLAM in simulation and real life

Assuming 'cartographer' packages have been installed it's pretty simple to start working with cartographer like the flow chart shows above you will need to start the three processes, you navigation method in this case tele-operation, you need to start your gazebo/|+rviz depending on if you are running on simulation of real world robot and finally you will run the cartographer node:

From Cartographer SLAM, two crucial components are generated: the map.pgm file and a corresponding YAML file.

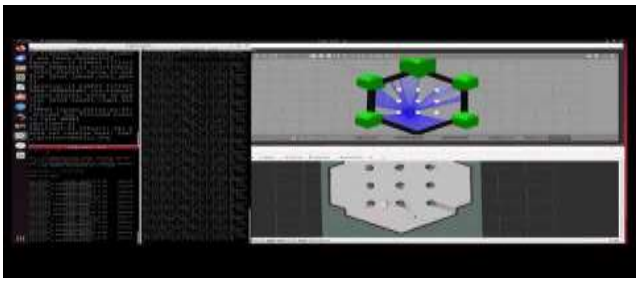


FIGURE 11: SLAM CARTOGRAPHER SIMULATION

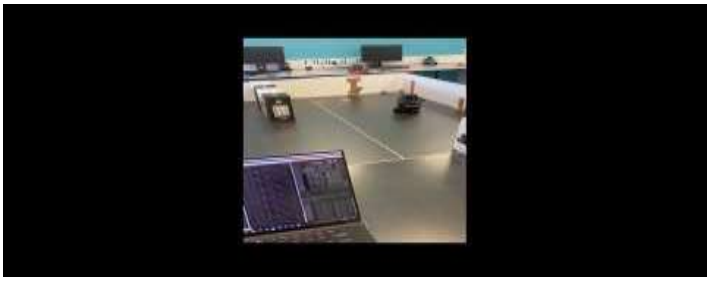


FIGURE 12: SLAM CARTOGRAPHER REAL WORLD

**1. map.pgm File:** The map.pgm file serves as the 2D representation of the environment, essential for discrete path planning algorithms. Each cell within the .pgm file is assigned one of three states:

- **White (255 in grayscale):** Indicates occupied cells within the environment.
- **Black (0 in grayscale):** Represents unoccupied cells.
- **Gray (grayscale values ranging from 0 to 255):** Denotes cells of unknown occupancy, reflecting uncertainty associated with each cell.

These distinct states allow for precise mapping of the environment and aid in path planning strategies.

**2. YAML File:** Accompanying the map.pgm file, the YAML file contains vital metadata crucial for interpreting the map's characteristics:

- **Resolution:** Specifies the resolution of the map.
- **Map Origin:** Defines the origin point of the map.
- **Cell Occupancy Thresholds:** Indicates thresholds for occupied and free space cells.
- **Mode:** In this case, the mode is trinary, indicating the presence of three cell states: occupied, unoccupied, and unknown.

Together, the map.pgm and YAML files provide comprehensive information about the environment's layout, occupancy status, and resolution, facilitating accurate navigation and path planning for the TurtleBot3.

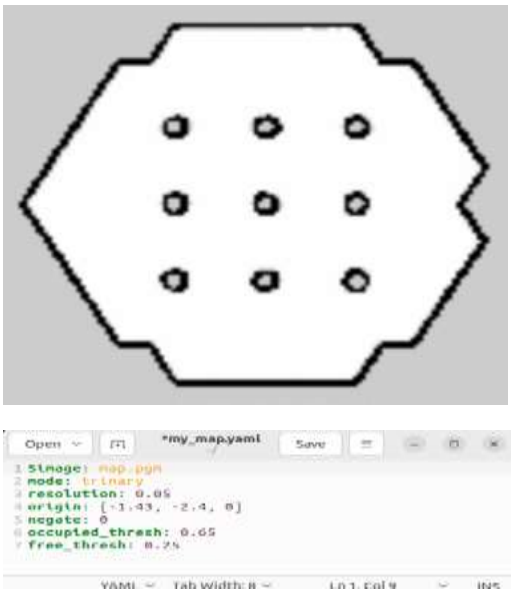



FIGURE 13: TB3_WORLD MAP

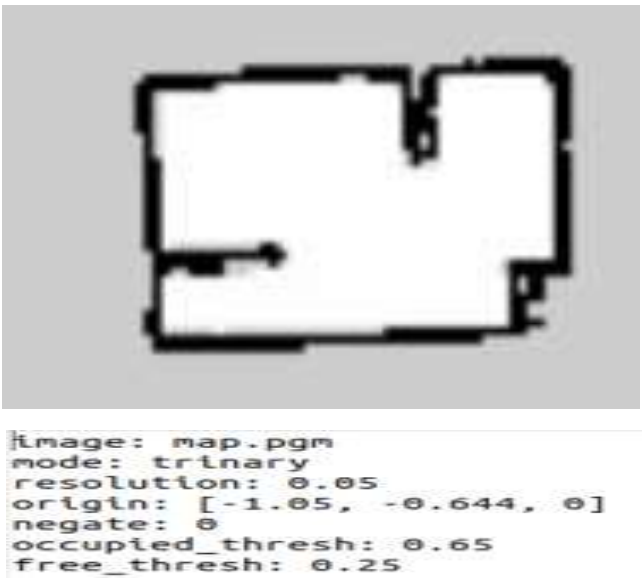



FIGURE 14: REAL WORLD MAP

2D occupancy grids are crucial for robotic navigation. They provide an environment representation and enable path planning algorithms to generate safe trajectories. By updating grid cells based on sensor data, robots can avoid collisions and dynamically adjust their paths. Additionally, occupancy grids support localization, aiding robots in estimating their position accurately.

## 5.3. Introduction to Autonomous Navigation

With a 2D representation of the robot's environment in hand, we can now explore the application of autonomous navigation to our TurtleBot3. Robotic path planning is a multifaceted domain involving the automated navigation of mechanical systems endowed with sensory, actuation, and computational capabilities. It encompasses a rich variety of planning algorithms meticulously tailored to meet the unique demands and limitations of robotic systems. While terms like "motion planning" and "trajectory planning" are sometimes used interchangeably in discussions of path planning, they signify distinct methodologies. Motion planning primarily concerns itself with devising routes without delving into the intricacies of robot dynamics and constraints, whereas trajectory planning takes these factors into careful consideration [13]. However, regardless of the specific algorithm used, robotic path planning typically consists of two primary components:

**High-Level Specifications from Human Operators**: The initial step in robotic path planning involves receiving overarching directives from human operators. These directives may encompass tasks such as guiding the robot from its current position to a designated goal location.

**Low-Level Descriptions of Movement:** This ladder aspect encompasses planning algorithms and behaviours crucial for enabling the robot to autonomously navigate its environment based on the high-level specifications provided. These algorithms and behaviours act as intermediaries, translating human intentions into actionable instructions for the robot to follow.

A standard low-level navigation stack typically comprises a hierarchical planning system with various components designed to fulfil specific tasks:

- **Global Path Planning**: This component determines a high-level route from the robot's current location to the specified destination. Algorithms such as A* (A-star), Dijkstra, or Rapidly Exploring Random Trees (RRT) are commonly employed for optimal or feasible pathfinding, considering obstacles and terrain.
- **Local Path Planning**: Following the global plan, local path planning facilitates safe navigation along the generated route while dynamically avoiding obstacles. Algorithms like Dynamic Window Approach (DWA), Potential Fields, or Model Predictive Control (MPC) generate smooth, collision-free trajectories, considering robot dynamics and environmental changes.
- **Obstacle Avoidance**: Static and dynamic obstacle detection and avoidance are crucial for safe navigation. Sensor data from devices like lidar and cameras are utilized to perceive obstacles in real-time. This information is integrated into the planning process to adjust the robot's trajectory accordingly.
- **Recovery Behaviours**: Unforeseen circumstances, such as getting stuck or encountering impassable terrain, require the system to incorporate recovery strategies. These strategies, including backtracking,

replanning, and local exploration, enable the robot to recover from navigation failures and continue making progress toward the goal.

- **Feedback Control**: Feedback control mechanisms, such as PID (Proportional-Integral-Derivative) control, ensure the accurate execution of planned trajectories. By adjusting the robot's velocity and steering commands based on sensor feedback, these mechanisms enable precise tracking of the desired path.

## 5.4. ROS2 Navigation2 Stack

Navigating through a low-level navigation stack might seem straightforward in discussion, but its implementation is far from trivial, demanding a significant amount of time—time that might not be readily available during a project. Fortunately, ROS2 provides a solution in the form of the widely adopted navigation2 stack [15] Navigation2, or Nav2, offers a comprehensive framework with pre-implemented packages tailored to handle navigation complexities, including path planning, cost map retrieval, and local planning. It standardizes the process of defining behaviour trees, making it easier to configure behaviours and packages using standardized formats, a topic I will delve into further below.

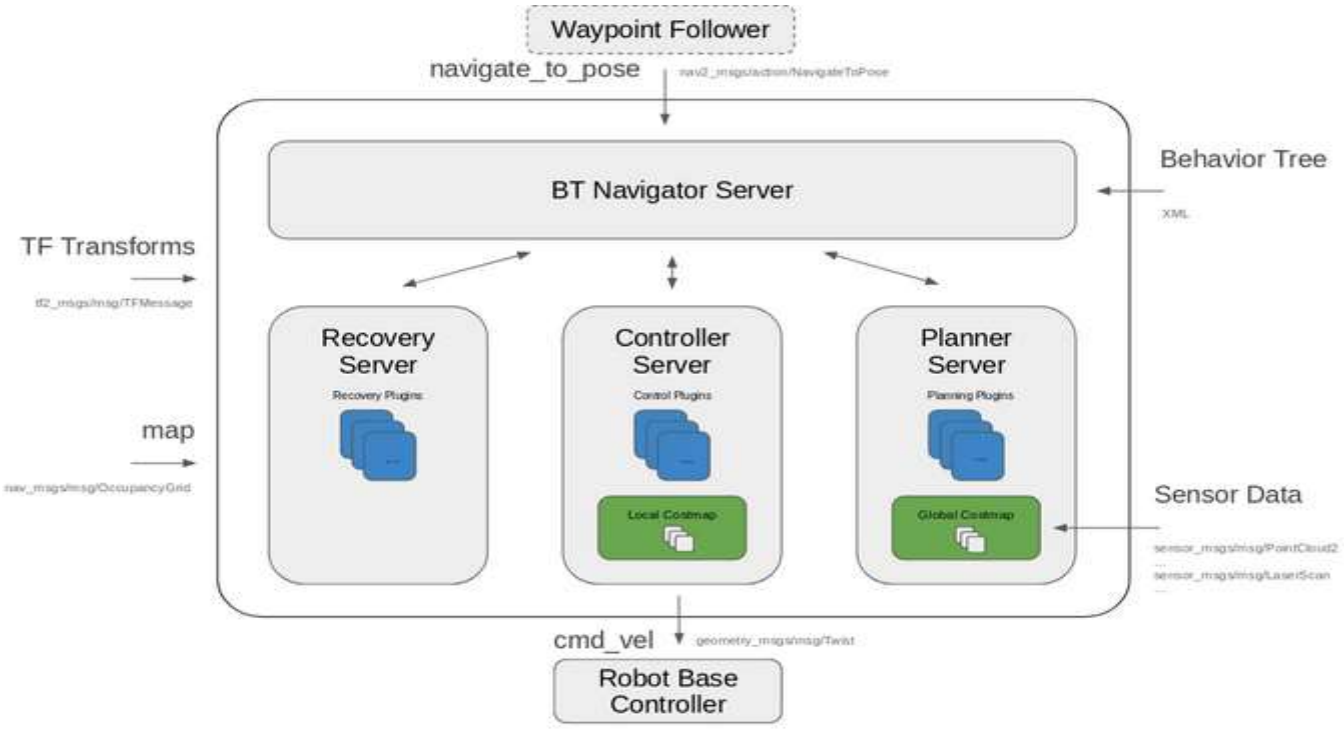



FIGURE 13: NAV2 LIFECYCLE MANAGER DIAGRAM

The Nav2 lifecycle manager, depicted above, plays a crucial role in orchestrating the operation of all nodes within the navigation stack. It manages the lifecycle of each node, ensuring seamless coordination and execution when the navigation stack is initialized. At the heart of Nav2's functionality lies the Behaviour Tree (BT) Navigator server, defined by an .xml file. This file outlines the behaviour trees comprising different behaviours necessary for effective navigation. Unlike traditional finite state machines, behaviour trees enable concurrent execution of multiple nodes, allowing for more flexible and robust navigation strategies.

To understand behaviour trees better, consider them as hierarchical structures composed of nodes representing various behaviours or tasks. These nodes are organized in a tree-like fashion, with parent nodes controlling the execution flow of their child nodes. At the lowest level, you have actions or tasks, while higher-level nodes dictate the overall decision-making process. Behaviour trees offer advantages over traditional methods by providing a more intuitive and scalable approach to robot behaviour control.

The behaviour tree referenced by Nav2 is configured through a YAML file, typically named nav2_params.yaml. This file defines the plugins associated with each behaviour, including critical components like the Planner server and Global cost map, which are indispensable for effective navigation planning and execution. By harnessing the power of behaviour trees and modular plugin-based architectures, Nav2 facilitates the development of sophisticated navigation systems with remarkable ease, enabling developers to create intelligent and adaptable robotic behaviours suited for real-world scenarios.

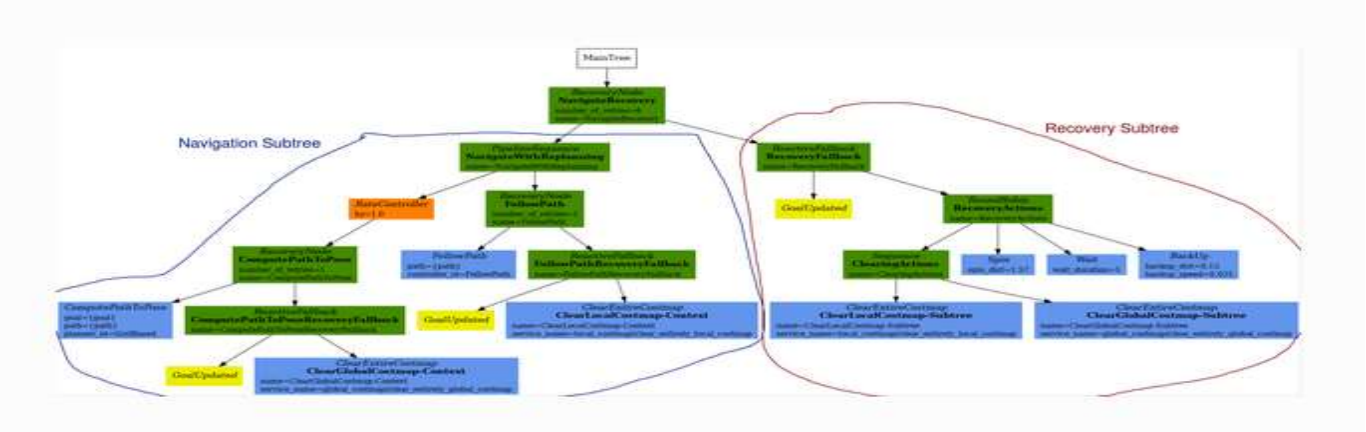



FIGURE 14: NAV2 NAVIGATE_TO_POSE_W_REPLANNING_AND_RECOVERY' BEHAVIOR TREE DIAGRAM

An exemplary instantiation of this concept is depicted in the default behaviour tree, denoted as 'navigate_to_pose_w_replanning_and_recovery,' contained within the nav2_bt_navigator package. This behaviour tree stands as the linchpin of my project, owing to its elegant simplicity and extensive coverage of fundamental navigation behaviours, encompassing recovery (e.g., autonomously resolving scenarios like collisions with obstacles) and replanning (automatic recalibration of navigation routes upon deviations) mechanisms. The sole modification made to this XML representation was the adjustment of the planner frequency in Hertz, aimed at exploring alternative planner algorithms, a topic to be elaborated further in this dissertation. This XML representation encapsulates a behaviour tree meticulously crafted for robotics or AI agents, integrating features

such as recovery mechanisms, periodic replanning, path-following actions, goal update checks, and fallback strategies for failure mitigation. For deeper insights into behaviour trees, readers are directed to the tutorial on behaviour trees for ROS2 robotics decision-making and control available at [18].

The operationalization of the behaviour tree defined within Nav2 is facilitated through a .params file, employed in launching the Navigation2 stack. This .params file not only integrates the behaviour tree but also delineates the parameters for each individual behaviour. Sample .params files can be in the nav2_bringup/params directory. Throughout the project, the nav2_params.yaml file served as the cornerstone for parameter files used in launching the Navigation2 stack, albeit with slight adjustments to accommodate specific project requisites. Notably, modifications were made to the planner server parameters to enable seamless integration of different navigation2 planner servers or a custom nav2_custom_planner_client, as expounded upon later in this dissertation.

Nav2 encompasses a plethora of internal components meticulously designed to tackle challenges pertaining to mobile robot dynamics. These components encompass tools for map loading, serving, and storage, as well as localization within a map and path planning within intricate environments.

Leveraging the Navigation2 framework, I conducted comprehensive tests utilizing A* and Dijkstra algorithms in both simulated and real-world scenarios.

To facilitate the integration of these components, the nav2_bringup package furnishes .launch files, employed in ROS to simultaneously launch multiple nodes, each configured with distinct parameters. These parameters dictate various factors such as the map file (e.g., map.pgm) to be utilized, whether Gazebo should be initiated for simulation purposes, and the selection of robot and world models.

Central to the harmonious orchestration of these components is the nav2_bringup.launch.py file. This file orchestrates the launch of the Navigation2 stack alongside the defined behaviour tree and its associated nodes, in conjunction with simulation and RVIZ environments. These environments facilitate real-time scrutiny of the robot's behaviour and performance in both simulated and real-world contexts, thereby furnishing invaluable insights for development and testing endeavours.

With this knowledge at our disposal, we are equipped to launch the navigation2 stack. By default, the planner_server harnesses the navFN planner_server, which, in turn, employs Dijkstra planning as its default mode but can readily be configured to utilize A*. Which is in my launch_files/simulation/navigation directory which contains pre-defined launch params including headless:=False which means that the simulation environment will be started the map is the slam output, params_file is a params_file containing default params which means the Dijkstra planner server will be ran.

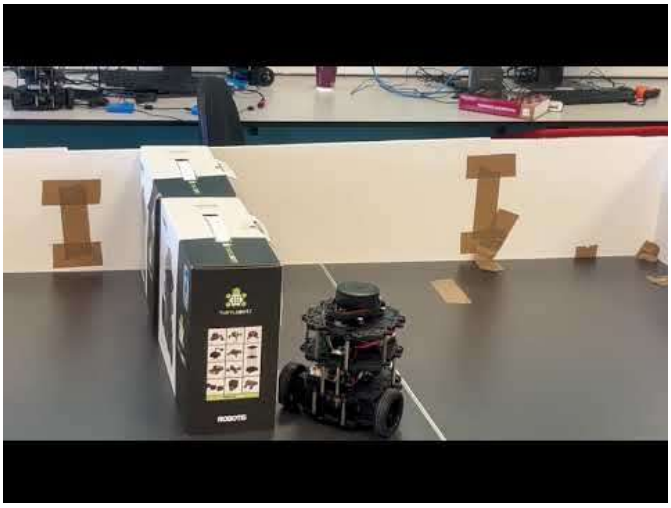


FIGURE 16: NAVFN DIJKSTRA REAL LIFE

FIGURE 15: NAVFN A* REAL LIFE

## 5.5. Navigation 2 custom planner plugin

In my endeavour to enhance navigation capabilities within the ROS2 Navigation2 stack, I embarked on developing custom global path planners, focusing specifically on RRT and RRT* algorithms. Despite the existence of pre-existing, well-optimized planning algorithms, my research revealed a notable gap in the implementation of these planners within the ROS2 ecosystem.

Furthermore, I aimed to broaden the scope of my work by implementing my interpretations of the classic Dijkstra and A* algorithms. This decision was driven by the desire to conduct a comprehensive comparison between heuristic and probabilistic path planning approaches, ensuring parity in optimization levels and hyperparameters for a fair assessment.

During the initial stages of implementation, I encountered a challenge regarding the integration of custom planners into the Nav2 stack. While a tutorial existed for creating a global path planner plugin, it was tailored to a simple straight-line planner. Despite successfully implementing this baseline model, I soon discovered discrepancies in the documentation. Specifically, the instructions provided in the params file failed to enable seamless integration of the plugin, necessitating further investigation and troubleshooting. You can find the tutorial at [17]

Working version of the planner_server configuration for the params file:

```
planner_server:
  ros__parameters:
    plugins: ["GridBased"]
    use_sim_time: True
```

```
GridBased:
  plugin: "nav2_straightline_planner/StraightLine"
  interpolation_resolution: 0.1
```

As I contemplated configuring the straight-line plugin for integration into my algorithm and leveraging rclpy for this purpose, I soon realized a significant impediment. Plugins for the Nav2 package are restricted to C++ due to the utilization of pluginlib for dynamically loading plugins, a mechanism that supports only C++ plugins [16]. Given my extensive familiarity with Python and the RCLPY library from the several course's I had took previously, this restriction posed a substantial challenge since I pivoted to implementing this planner near the end of the project.

Consequently, I opted for an alternative approach. Rather than attempting to directly implement the global planner plugin within the Nav2 stack, I devised a solution centered around a service named "/custom_planner_server." This service is designed to receive requests encompassing essential planning data, such as the current cost map, robot pose, and goal pose. Subsequently, it processes these inputs and returns a message containing a sequence of poses necessary to navigate from the current position to the goal within the cost map.

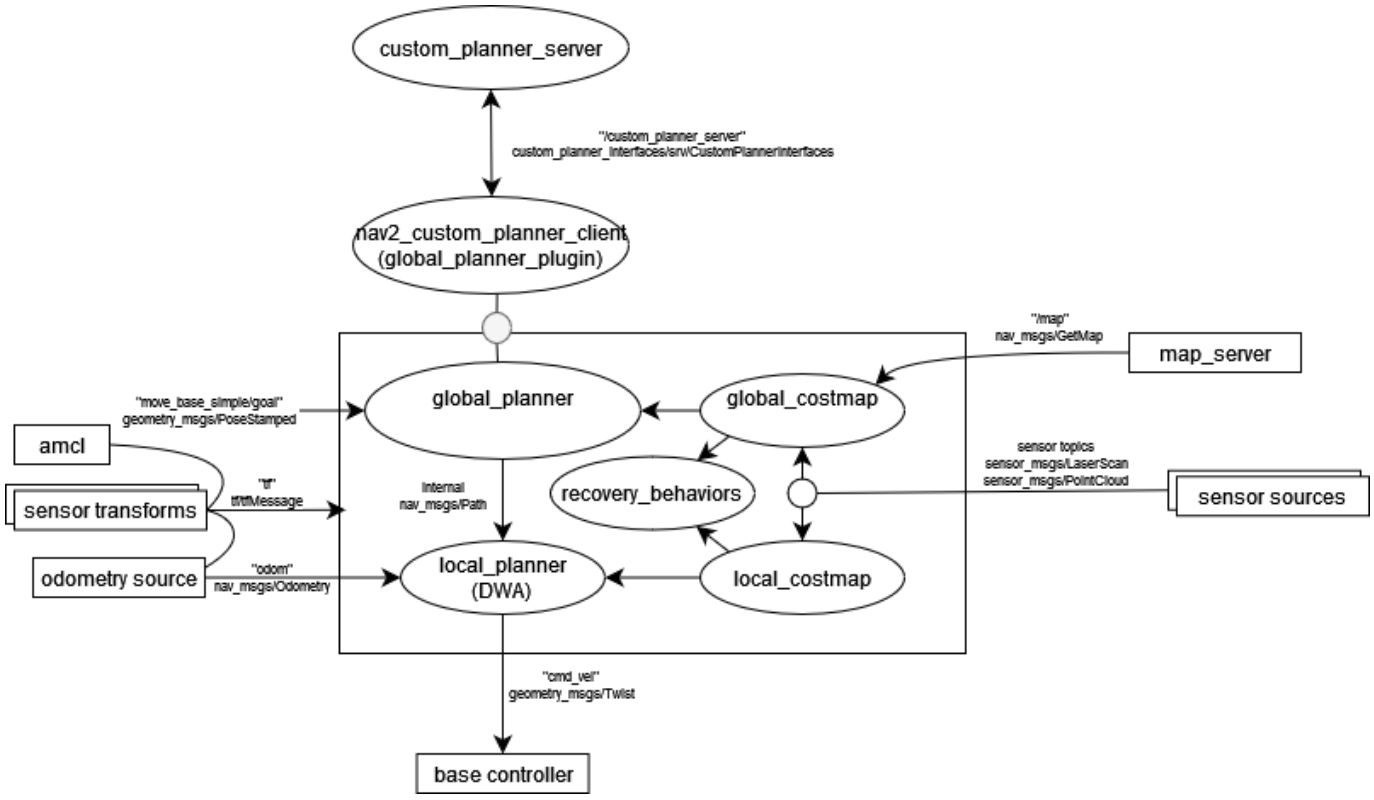



**FIGURE 17:** NAV2 STACK ARCHITECTURE WITH CUSTOM PLANNER PLUGIN COMPONENTS

Above shows a high-level diagram of how I implemented this. The nodes within the box are the behaviors that create the navigation plans which outside the boxes are topics which generate information about the robot environment for use by the navigation2 stack.

The global planner plugin employs the AMCL (Adaptive Monte Carlo Localization) method, a probabilistic algorithm tailored for 2D robot localization within a known map. AMCL initializes a particle filter and refines its estimates by assimilating laser scans and odometry data, yielding pose estimates with associated uncertainty. The crux of its operation lies in the dead reckoning transform, which aligns the robot's odometry data with its baseframe—a pivotal aspect for accurate localization. Here, the baseframe denotes the fundamental reference frame of the robot, typically centered around its physical core or a defined point on its chassis. Meanwhile, the odometry frame encapsulates the transformations induced by the robot's actuators, providing crucial motion data. By integrating data from these frames, AMCL furnishes precise estimations of the robot's position relative to the broader spatial coordinates of the environment.

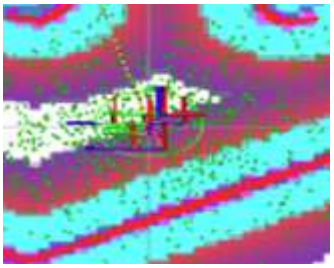


FIGURE 18: ILLUSTRATES THE CORE COMPONENTS OF THE LOCALIZATION PROCESS. THE GREEN PARTICLES DEPICT THE ENSEMBLE CONSTITUTING THE PARTICLE FILTER, REPRESENTING VARIOUS POSSIBLE POSITIONS OF THE ROBOT WITHIN THE ENVIRONMENT. STACKED ATOP THESE PARTICLES ARE THE BASE-FRAME AND ODOMETRY FRAME, PIVOTAL REFERENCE FRAMES UTILIZED IN THE DEAD RECKONING PROCESS. THROUGH THE INTEGRATION OF DATA BETWEEN THESE FRAMES, THE ALGORITHM DEDUCES THE ROBOT'S RELATIVE POSITION AND ORIENTATION, WHICH ARE THEN TRANSFORMED INTO THE MAP FRAME FOR ACCURATE POSE ESTIMATION. THIS VISUALIZATION ENCAPSULATES THE ESSENCE OF THE LOCALIZATION ALGORITHM, SHOWCASING HOW THE FUSION OF SENSOR DATA AND FRAME TRANSFORMATIONS FACILITATES PRECISE LOCALIZATION WITHIN THE MAPPED ENVIRONMENT.

Parameters within the algorithm govern filter behaviour, while a laser model deciphers scan data and an odometry model estimates robot motion. AMCL furnishes vital services for global localization and map setting. It facilitates the transformation of laser scans into the odometry frame and broadcasts the global-to-odometry transform. Continuously updating its belief about the robot's pose, AMCL harmonizes sensor data and motion information to enhance localization accuracy within 2D environments. Moreover, it ensures the alignment of laser scans with the odometry frame, solidifying the transform between the laser and base frames upon the first receipt of laser data—albeit it cannot handle a dynamically moving laser concerning the base. During operation, AMCL estimates the transformation from the base frame to the global frame, albeit only publishing the transform between the global and odometry frames. This strategic approach mitigates drift arising from Dead Reckoning, with the published transforms being temporally future dated. The data disseminated via the amcl topic serves as the initial pose for the global_planner algorithm upon invocation.

For further insights, refer to the official documentation: [19].

For further information on transformations: ([20])

The global_planner module interfaces with the global_costmap topic, which emits a structured message of type 'nav2_msgs/Costmap'. This message is instrumental in furnishing essential data for navigation planning. Within its payload, key attributes such as map resolution, width, height, and the map's origin frame, as defined by 'geometry_msgs/Pose', are provided. However, the crux of this message lies in its encapsulation of the cost map's cell values, crucial for informed decision-making in navigation tasks.

Unlike a conventional occupancy grid, a costmap operates on a layered architecture, integrating multiple layers of information to derive composite cost values. Each layer within the costmap framework can be configured with distinct update frequencies, data sources, and parameters. In the context of our turtlebot3 algorithm, three primary layers are utilized: the static layer, obstacle layer, and inflation layer.

The static layer represents a fixed map of the environment, typically derived from the output of a SLAM algorithm, such as the occupancy grid generated by our SLAM algorithm. This layer provides a foundational representation of the environment's structure.

In contrast, the obstacle layer operates in real-time, dynamically detecting and incorporating information about moving objects within the robot's vicinity. This layer enables the robot to react to dynamic obstacles swiftly and adapt its trajectory accordingly.

Finally, the inflation layer adds a protective buffer zone around detected obstacles to accommodate the robot's size and navigational uncertainties. This additional margin of safety ensures smoother navigation by mitigating the risk of collisions.

The configuration of the costmap, including its layer specifications, can be customized through the 'nav2_custom_planner_params.yaml' file. By default, the system leverages the capabilities of the 'nav2_costmap_2d' packages to define the obstacle, static, and inflation layers. While custom plugins offer the flexibility to tailor the costmap's behaviour further, such endeavours require careful consideration of system dynamics and objectives. As such, I opt not to explore this avenue, preferring a more conservative approach to maintain system stability and reliability.

For further information: [21]

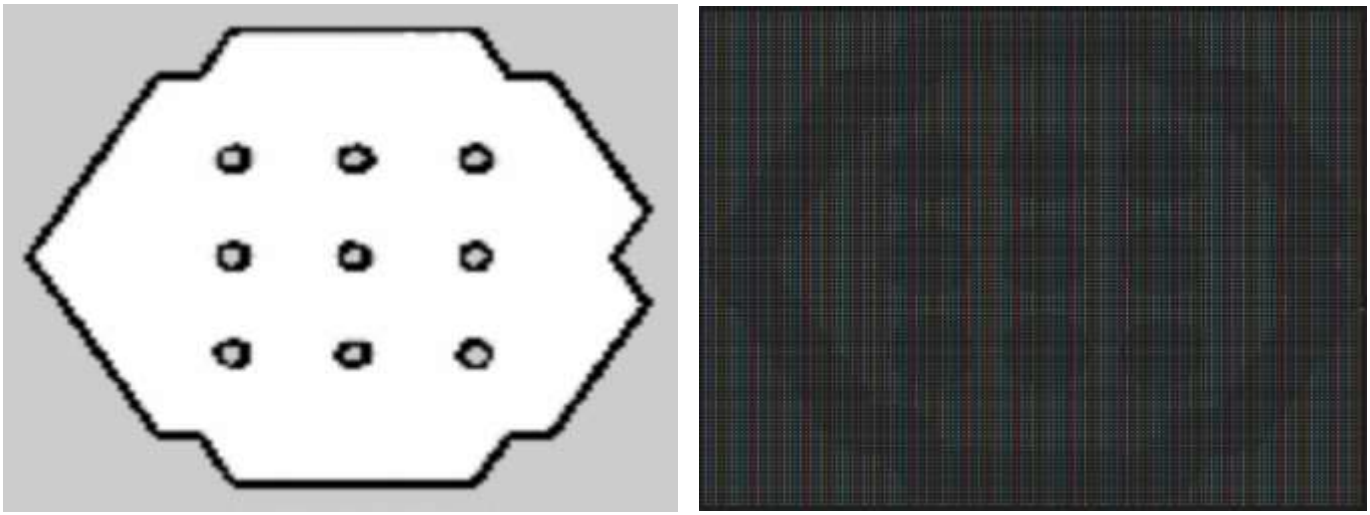


FIGURE 20: THIS ILLUSTRATION PRESENTS THE THREE-STATE OCCUPANCY GRID ALONGSIDE ITS CSV REPRESENTATION.

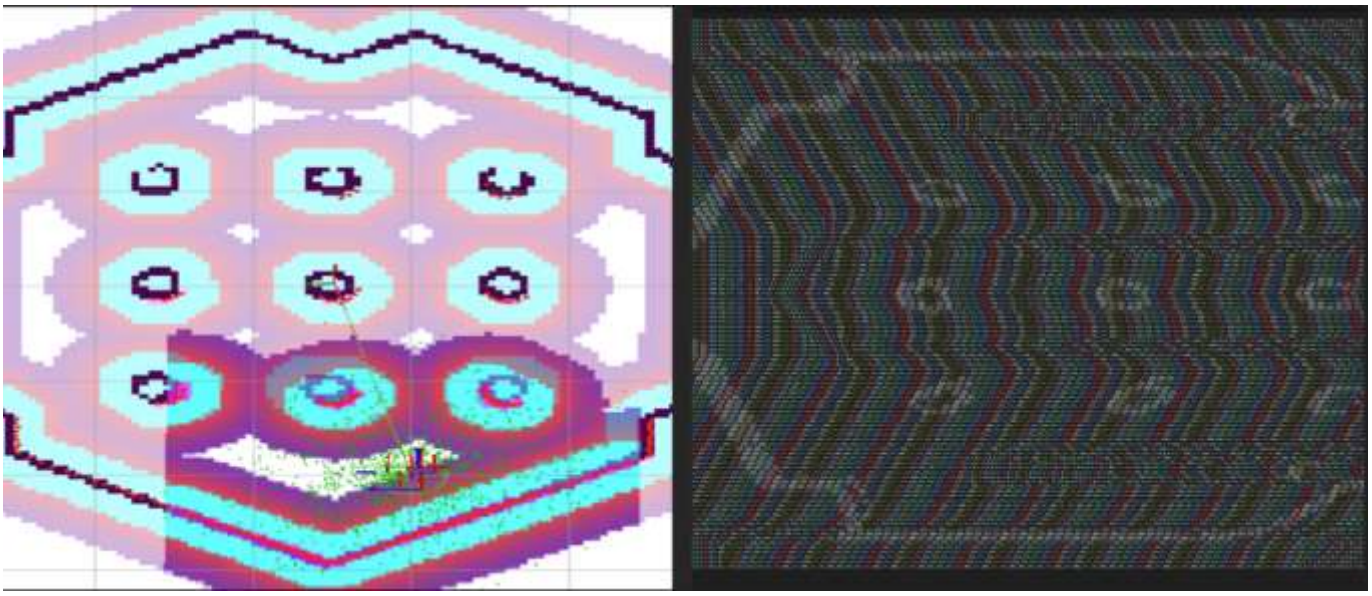


FIGURE 19: DISPLAYS THE COST MAP AND ITS CSV REPRESENTATION. THE MAP DEPICTS SPATIAL COST VALUES INDICATING NAVIGATIONAL DIFFICULTY OR OBSTACLE PRESENCE. EACH CELL VALUE RANGES FROM 0 TO 254. THE CSV FORMAT ON THE RIGHT PROVIDES STRUCTURED DATA FOR DEEPER ANALYSIS AND ALGORITHMIC ADJUSTMENTS.

Analysis of the navigation stack above shows that this start pose generated from ACML and the goal pose set by human at navigation start when the input/algorithm input and the cost map are the inputs that we can expect for our custom planner. And the output is nav_msgs/Path which contains an array of poses therefore that for the client it will interpret all the input data and send it in an efficient format using my custom. srv interface format 'custom_planner_interfaces/srv/CustomPlannerInterface which will contain:

```
geometry_msgs/PoseStamped start_pose
geometry_msgs/PoseStamped end_pose
nav_msgs/OccupancyGrid occupancy_grid
---
bool success
std_msgs/Float32MultiArray path  # [[x,y], [x, y],…,[x,y]]
```

Using this message format I then created the nav2_custom_planner_client package based on the straightline planner. I kept most of the package the same but changed a few key things, especially the createPlan function. The createPlan function now carries out this algorithm.

1. An empty nav_msgs::msg::Path object named global path is created. This will be the path that the function returns.

2. The function checks if the frame_id of the start and goal poses match the global frame. If they don't, an error or info message is logged and the empty global_path is returned.

3. A service request of type custom_planner_interfaces::srv::CustomPlannerInterface::Request is created and populated with the start and goal poses.

4. The occupancy grid for the request is constructed. The occupancy grid is a representation of the environment where each cell represents a state of occupancy. The grid's metadata (resolution, width, height, origin) is set using the costmap's properties. The grid's data is resized to match the grid's dimensions.

5. The occupancy grid's data is populated with the costmap's data. This is done by iterating over each cell in the grid and setting its value to the corresponding cost from the costmap.

6. The populated request is sent to the planner client using the sendRequest function. The result, which is the planned path, is stored in global path.

7. If the planned path is not empty, the goal pose is added to the end of the path.

8. The header information for the returned path is set to the current time and the global frame.

9. The function returns the global path which is then sent onto the local_planner.

After the successful creation of the nav2_custom_planner_client, attention turned to the implementation of the custom_planner_server. Operating as a node within the Robot Operating System (ROS), the PlannerServer class

assumes the responsibility of handling path planning requests and employs a variety of algorithms including Dijkstra, A*, RRT, and RRT*. This class follows a structured workflow:

- **Initialization (init method):** The class is initialized with parameters such as the node name, chosen planner type, a flag for log saving, and the log file path. It establishes a service named 'custom_planner_server' to manage requests, utilizing the _planner_callback method for processing.
- **Planner Callback (_planner_callback method):** Triggered upon request reception, this method logs the request receipt and records the start time. It then extracts the occupancy grid metadata from the request and converts it into a 2D list. Subsequently, it translates the start and goal poses into grid cell coordinates, invoking the planner to compute the path. If no viable path is found, a failure response is issued. Conversely, if a path is discovered, it undergoes smoothing and conversion into a ROS message format before being returned within the response. Optionally, logging is facilitated by creating a log file.
- **Call Planner (_call_planner method):** This method selects the appropriate planner algorithm based on the requested type, initializes an instance of the chosen planner with the grid, and executes the find_path method to generate the path.
- **Main Function:** Serving as the program's entry point, this function parses command-line arguments to obtain planner type specifications, log-saving preferences, and log file paths. Subsequently, it initializes the ROS client library (rclpy), instantiates a PlannerServer node, and perpetuates node activity through spinning until manual shutdown or exceptions occur. Upon cessation of node activity, proper cleanup routines are executed, including node destruction and shutdown of the ROS client library.

The PlannerServer class is designed to be versatile and expandable, facilitating the seamless integration of new path planning algorithms. It encompasses pre-processing and post-processing steps, refined through iterative experimentation, to handle data format conversions and path smoothing. This adaptability aligns with Navigation 2's support for plugin development, enabling the incorporation of additional planners into the system framework.

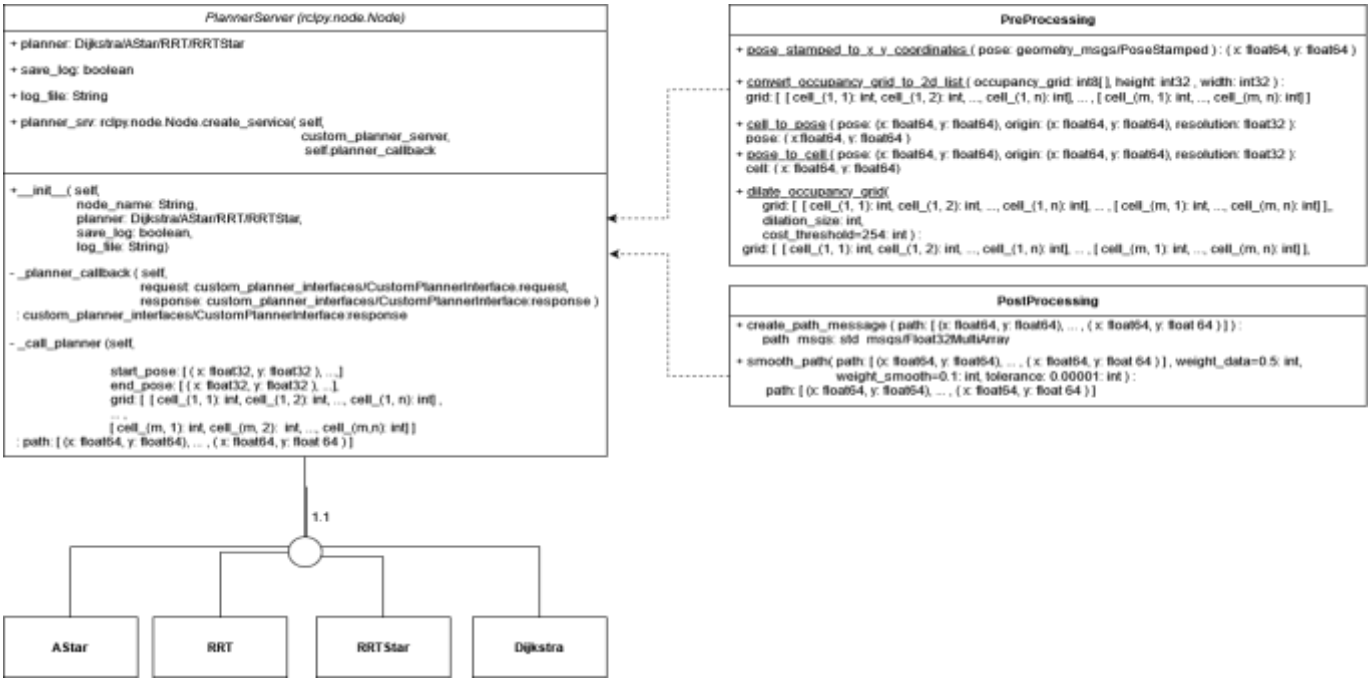



FIGURE 21: CUSTOM_PLANNER_SERVER UML DIAGRAM

The planner server then was configured with arguments in the main which allowed for setting rrt, rrt_star, a_star and dijkstra below is the command to run the custom_planner_server.

```
ryan@TB3:~$ ros2 run custom_planner_server planner_server --planner rrt
[INFO] [1714522326.975865831] [custom_planner_server]: rrt Planner Server Node has Started........
[INFO] [1714522474.727689307] [custom_planner_server]: Received request........
[INFO] [1714522474.741658247] [custom_planner_server]: calling rrt planner.....
[INFO] [1714522475.437924772] [custom_planner_server]: Response Sent: Path Found.............. (Response time: 0.6952083110809326 seconds)

[INFO] [1714522495.485996197] [custom_planner_server]: Received request........
[INFO] [1714522495.486469238] [custom_planner_server]: calling rrt planner.....
[INFO] [1714522495.507559756] [custom_planner_server]: Response Sent: Path Found.............. (Response time: 0.020983219146728516 seconds)
```

FIGURE 22: DEMONSTRATES THE CUSTOM PLANNER SERVER IN ACTION, ACTIVELY RECEIVING AND PROCESSING REQUESTS.

## 5.6. Planner Modules

To facilitate efficient navigation within its operational environment, the custom_planner_server incorporates dedicated algorithms for global pathfinding. I developed separate and autonomous modules for four distinct types of path planning algorithms: AStar, RRT, RRTStar, and Dijkstra. Each module seamlessly integrates into the custom_planner_server, activated based on the specified argument during node initialization.

### Dijkstra Planner

The initial global planner created and tested was the Dijkstra module, featuring a meticulously crafted implementation of Dijkstra's algorithm tailored for testing purposes. While fully functional on its own, its design is optimized for seamless integration with the custom_planner_pkg. Dijkstra's algorithm, renowned for its proficiency in identifying the shortest path between a starting node and all other nodes in a graph, finds extensive applications in route planning, navigation, and diverse domains beyond.

The Dijkstra class is initialized with crucial parameters: a grid representation of the environment and a meticulously calibrated cost threshold. Through iterative experimentation, I determined the optimal threshold to be 230, balancing performance and safety by ensuring the algorithm avoids overly risky paths. This threshold, refined through careful testing, strikes a delicate equilibrium between obstacle avoidance and efficient pathfinding. Structured as a 2D list, the grid serves as a spatial map for navigation, while the cost threshold acts as a beacon, defining the upper limit for acceptable path costs. The find_path method orchestrates the algorithm's execution, guiding it from a designated starting point to an endpoint within the grid. Leveraging a priority queue (heap), the algorithm prioritizes nodes based on their accumulated cost, a core principle of Dijkstra's methodology. Additionally, it integrates a sophisticated cost threshold mechanism, allowing the algorithm to bypass paths with excessively high costs, ensuring optimal route selection in complex spatial environments.

.

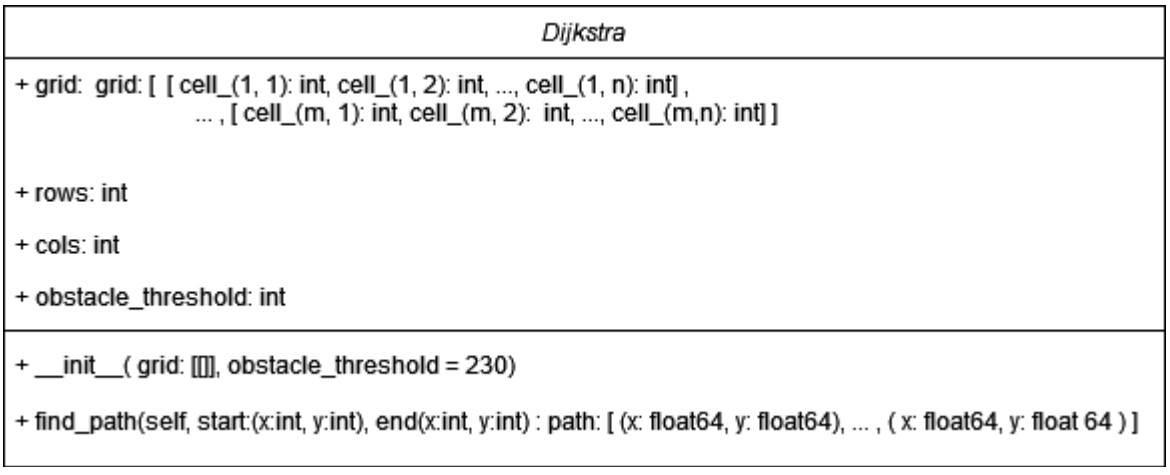



FIGURE 23: DIJKSTRA UML

The outcomes of these Integration tests yielded the following results:

| Test Number | Time Taken (seconds) | Path Cost | Path Visualization |
|---|---|---|---|
| **1** | 0.017369985580444336 | 3444 | [A1] |
| **2** | 0.009747982025146484 | 3990 | [A2] |
| **3** | 0.002007722854614258 | 2102 | [A3] |
| **4** | 0.011528968811035156 | 5316 | [A4] |
| **5** | 0.005010128021240234 | 3847 | [A5] |

The integration testing of the Dijkstra module reveals promising results, with consistent performance across various scenarios. The time taken for each test execution ranged from approximately 0.002 to 0.017 seconds, demonstrating the algorithm's efficiency in pathfinding. Path costs also varied, reflecting the algorithm's adaptability to different environments, with costs ranging from 2102 to 5316. These results align with Dijkstra's time complexity of O((V+E) log V) and space complexity of O(V), indicating its scalability and suitability for real-world navigation applications. Despite its computational efficiency, Dijkstra's algorithm exhibits limitations such as the necessity for complete information and potential inefficiency in graphs with negative edge weights. However, its ability to identify the shortest path between nodes in a graph makes it a valuable tool for route planning and navigation systems. In conclusion, the Dijkstra module presents a reliable and efficient solution for global pathfinding, offering a balance of performance and adaptability essential for navigating complex spatial environments.

## A* Planner

The A* algorithm stands as a cornerstone in pathfinding, renowned for its prowess in discovering the least-cost path from a starting node to a designated goal node. Leveraging a best-first search strategy, A* intelligently explores neighboring nodes while being guided by a heuristic function, often the Manhattan distance, to estimate the cost from the current node to the goal. This heuristic-driven exploration minimizes node traversal, particularly advantageous in grid-based environments where efficiency is paramount. The algorithm's initialization involves setting up essential data structures, including a priority queue and auxiliary dictionaries to track costs and parent-child relationships. Through iterative refinement, A* dynamically identifies the optimal path, balancing computational efficiency with path optimality.

Below showcases the UML diagram of the AStar class, encapsulating various functions essential for efficient pathfinding.

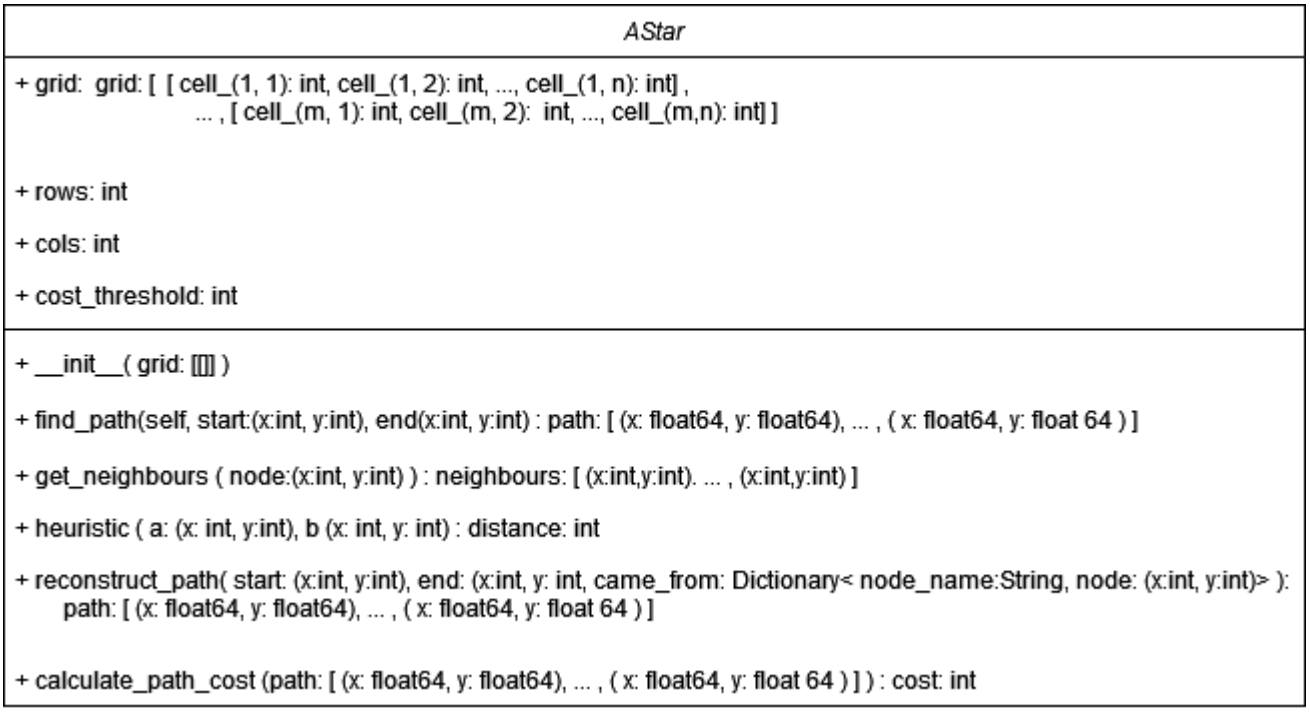



FIGURE 24: ASTAR CLASS UML

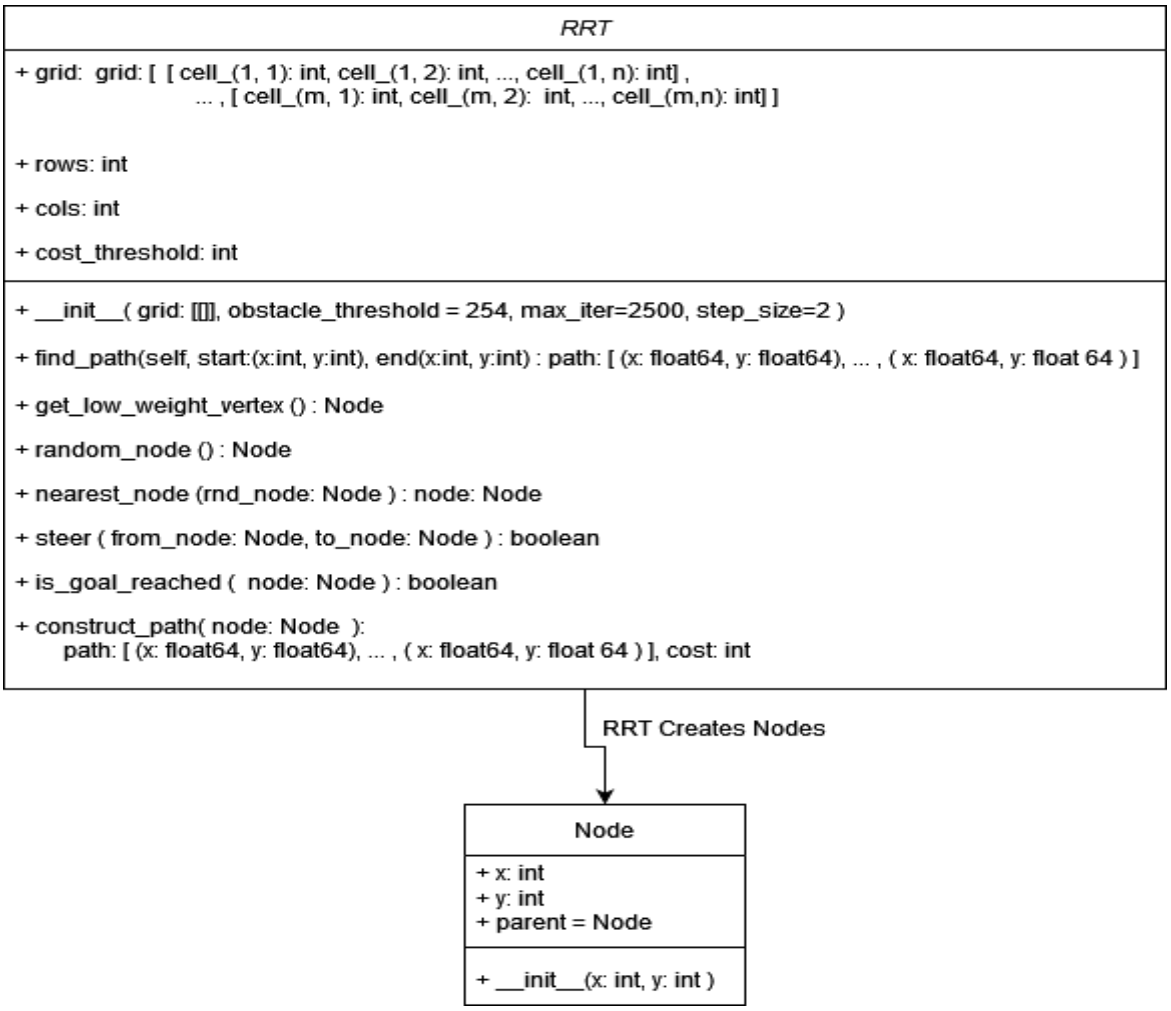
The test suite outcomes are summarized as follows:

| Test Number | Time Taken (seconds) | Path Cost | Path Visualization |
|---|---|---|---|
| **1** | 0.002999544143676758 | 5396 | [A6] |
| **2** | 0.005657196044921875 | 5822 | [A7] |
| **3** | 0.0011937618255615234 | 2412 | [A8] |
| **4** | 0.009800434112548828 | 5769 | [A9] |
| **5** | 0.0039861202239990234 | 5822 | [A10] |

The A* algorithm offers a robust solution for pathfinding, excelling in identifying the optimal path from a starting node to a goal node with the least cost. By employing a best-first search approach guided by a heuristic function like the Manhattan distance, A* efficiently explores neighbouring nodes, significantly reducing node traversal, especially beneficial in grid-based environments. Through its initialization, A* sets up essential data structures, facilitating iterative refinement to dynamically identify the most efficient path. The provided UML diagram encapsulates the algorithm's essential functionalities, ensuring its efficacy in pathfinding tasks. Integration tests conducted validate its functionality, demonstrating consistent performance across various scenarios, with minimal time taken for path generation and varying path costs, reflecting adaptability to different environments. The algorithm's time and space complexity, although theoretically O(b^d), can be optimized with the utilization of effective heuristic functions. In conclusion, A* presents a powerful and versatile solution for navigation tasks, offering a balance of computational efficiency and path optimality essential for real-world applications.

## Rapidly exploring Random Tree (RRT)

Rapidly exploring Random Tree (RRT) is a widely applied probabilistic, sampling-based algorithm lauded for its effectiveness in path planning across diverse domains, particularly in robotics and autonomous vehicles. This algorithm dynamically explores the search space by randomly sampling configurations and incrementally constructing a tree structure towards these samples, facilitating rapid exploration of expansive environments. The algorithm's initialization involves setting up parameters such as the environment grid, obstacle threshold, maximum iteration count, and step size. Key methods, including random node generation, nearest node finding, and node validation, are pivotal in steering the tree expansion towards the goal while adhering to constraints. Additionally, the algorithm employs goal checking and path construction mechanisms to iteratively construct a path from the start to the end node. Integration tests, conducted to assess feasibility and optimality, validate the algorithm's performance through rigorous evaluation of time taken, path cost, and visualization quality. Despite its simplicity, RRT boasts a time and space complexity of O(n), where n represents the number of iterations, making it suitable for real-world applications with varying environmental complexities.



Testing like with the other modules was conducted in the 'tests' module which contains a 'test_rr..py file it utilizes an csv representation of a cost map to create a 2d list input in the 'test_grids/tb3_world_costmap.csv' and with several different start and end poses set we test the feasibility through analyzing the visualizations of the paths generated but also the optimality by looking at the final costs of paths generated coupled with time taken for paths to generate.

The following is the test results produced by the integration tests on the module.

| Test Number | Time Taken (seconds) | Path Cost | Path Visualization |
|---|---|---|---|
| **1** | 0.057216644287109375 | 11931.0 | [A11] |
| **2** | 0.3798708915710449 | 17851.0 | [A12] |
| **3** | 0.0023860931396484375 | 3320.0 | [A13] |
| **4** | 0.12484169006347656 | 5089.0 | [A14] |
| **5** | 0.0341036319732666 | 5424.0 | [A15] |

The integration tests conducted on the Rapidly Exploring Random Tree (RRT) algorithm provide valuable insights into its performance and reliability. The algorithm demonstrates efficiency in generating paths, with varying time taken ranging from 0.002 to 0.379 seconds across different test scenarios. Path costs, reflective of the algorithm's adaptability to diverse environments, range from 3320.0 to 17851.0, showcasing its ability to navigate through complex spatial landscapes. With a time and space complexity of O(n), where n represents the number of iterations, RRT offers scalability and suitability for real-world applications, making it particularly attractive for robotics and autonomous vehicles operating in dynamic environments. However, the algorithm's simplicity comes with limitations, such as the lack of optimality guarantees and potential sensitivity to parameter tuning. Despite these drawbacks, RRT presents a promising solution for path planning tasks, offering rapid exploration and adaptability essential for navigating expansive and uncertain environments.

## RRT Star

The RRT* algorithm enhances the RRT framework with key improvements aimed at refining path quality and exploration efficiency. Building upon the RRT class, RRTStar introduces the search_radius_factor parameter to determine the radius for node rewiring, offering flexibility in exploration strategies. After node addition, the rewire method evaluates nearby nodes within a specified radius, dynamically rerouting through the new node if it leads to a lower-cost path, thus enhancing path refinement during exploration. The cost method calculates path costs by summing distances from each node to its parent, facilitating efficient evaluation of potential path improvements. Additionally, RRT* incorporates the get_low_weight_vertex method to select vertices with low edge weights, typically below 128 in the grid representation. The find_path method orchestrates path discovery, incorporating dynamic rewiring and leveraging the cost method to assess path quality. These enhancements collectively empower RRT* to navigate complex environments more effectively, resulting in higher-quality paths and improved exploration efficiency compared to its predecessor.

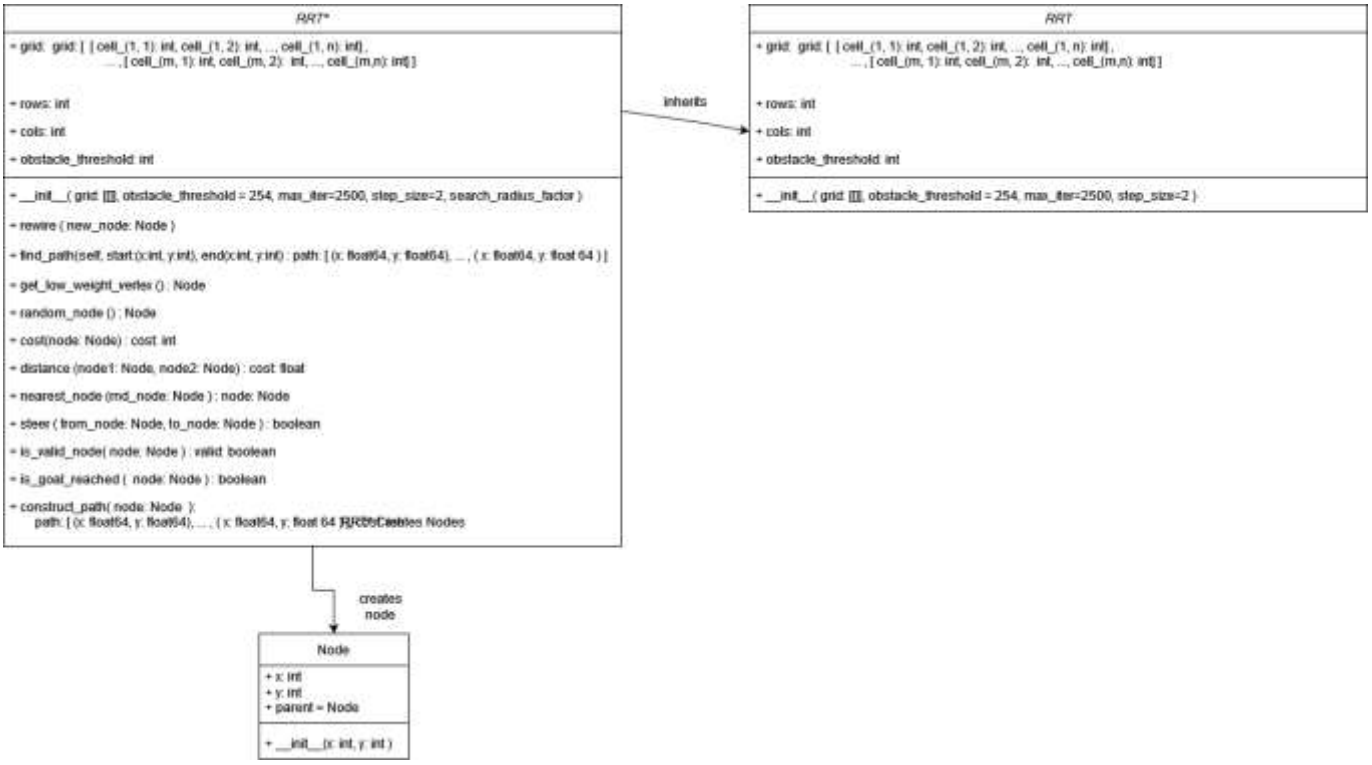



FIGURE 25: RRT* UML DIAGRAM

| Test Number | Time Taken (seconds) | Path Cost | Path Visualization |
|---|---|---|---|
| **1** | 0.47685980796813965 | 11861 | [A16] |
| **2** | 0.1312718391418457 | 9571 | [A17] |
| **3** | 0.012151479721069336 | 7867 | [A18] |
| **4** | 0.05119442939758301 | 5450 | [A19] |
| **5** | 0.05552029609680176 | 5024 | [A20] |

The RRT* algorithm demonstrates a time complexity of O(n log n), where 'n' represents the number of nodes. This complexity arises from the search for the nearest node, which takes O(log n) time, and potential rewiring of the tree, which takes O(n) time. Such time complexity is favorable for real-world applications, offering efficient pathfinding in various environments. Integration tests further validate the algorithm's performance, with relatively low time taken for path generation across different scenarios. For instance, the test results show that the algorithm consistently produced paths within reasonable time frames, ranging from approximately 0.01 to 0.48 seconds. Additionally, the varying path costs observed in the integration tests suggest the algorithm's adaptability to different environments and scenarios, further reinforcing its efficacy for navigation tasks. Despite these positive outcomes, the issue of unsmooth path lines causing erratic behavior in navigation highlights a potential area for improvement, emphasizing the importance of refining path smoothing techniques to enhance the algorithm's performance further.

# 6. Evaluation of Integration and Simulation Testing of Path Planning Algorithms

## 6.1. Integration test results

Upon thorough analysis of various path planners, it's clear that A* emerges as the most efficient and optimal solution. Leveraging its heuristic-driven approach, A* consistently generates low-cost paths within reasonable computation times, making it particularly suitable for grid-based environments. Its capacity to efficiently explore neighbouring nodes while guided by heuristics positions it as a leading pathfinding algorithm. However, there remains room for enhancement in A*, especially in dynamic environments where adaptive heuristics or alternative heuristic functions could further bolster its performance.

Following A*, Dijkstra presents itself as a dependable alternative, albeit with slightly reduced efficiency and optimality. Despite consistently finding paths with moderate to good efficiency and manageable costs, Dijkstra's exhaustive search methodology may limit its efficacy in larger, intricate environments. Strategies like bidirectional Dijkstra or integrating heuristic guidance hold promise in addressing these constraints and elevating its performance to approach that of A*.

Meanwhile, the probabilistic nature of RRT and RRT* renders them suitable for navigating expansive and uncertain environments, albeit with varying levels of efficiency and optimality. RRT, known for its swift exploration, tends to yield higher-cost paths compared to A* and Dijkstra. RRT*, with its improvements in path quality and exploration efficiency, represents a step forward from RRT but still trails A* in performance. Refinements in sampling strategies, exploration techniques, and heuristics could further enhance both RRT and RRT* to strike a better balance between exploration speed and path optimality. Overall, while each path planner boasts strengths and weaknesses, continual refinement and optimization are essential to unlocking their full potential across diverse real-world scenarios.

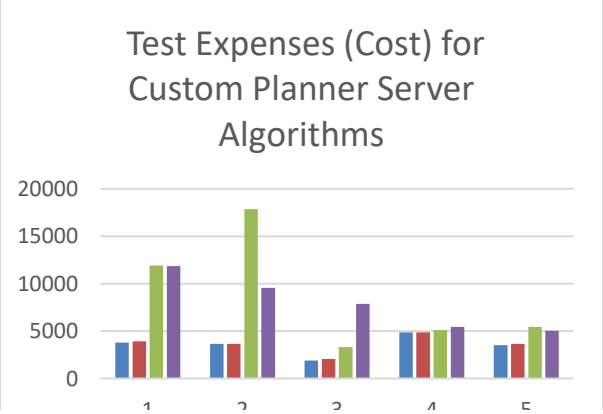



FIGURE 27: INTEGRATION TESTING COSTS PER TEST GRAPH

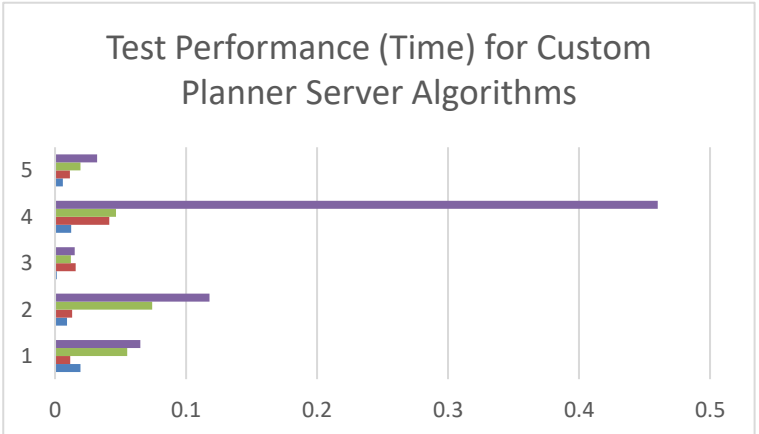



FIGURE 26: INTEGRATION TESTING TIME PER TEST GRAPH

## 6.2. Simulation Testing

In addition to integration testing, simulation testing was conducted on various path planning algorithms, including navFN A-star, Dijkstra, as well as custom implementations of RRT, RRT*, Dijkstra, and A*. Utilizing the ROS commander API, multiple goal poses were set for the TB3 robot to navigate through, allowing for a comparative assessment of algorithm performance within a simulated environment.

(https://navigation.ros.org/commander_api/index.html )

Commander API Tests Goal Poses:

| | Goal1 | Goal2 | Goal3 | Goal4 | Goal5 | Goal6 |
|---|---|---|---|---|---|---|
| **X** | 1.504 | 3.826 | 2.162 | 0.147 | 2.599 | 0.447 |
| **Y** | -1.4 | 0.831 | 1.274 | 0.891 | 0.175 | 1.907 |
| **w** | 0.999 | 0.77 | 0.267 | 0.65 | 0.572 | 0.653 |

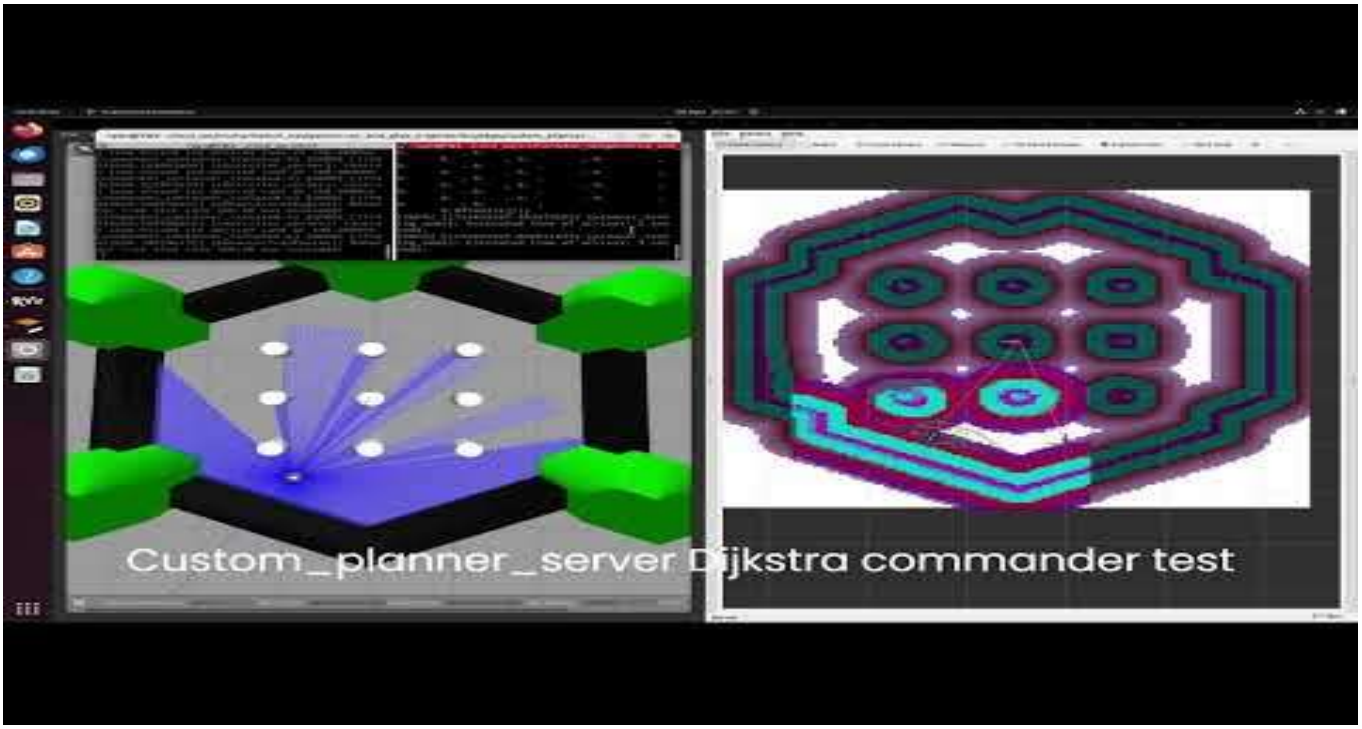



FIGURE 28: NAVIGATION ALGORITHMS SIMULATION TESTING

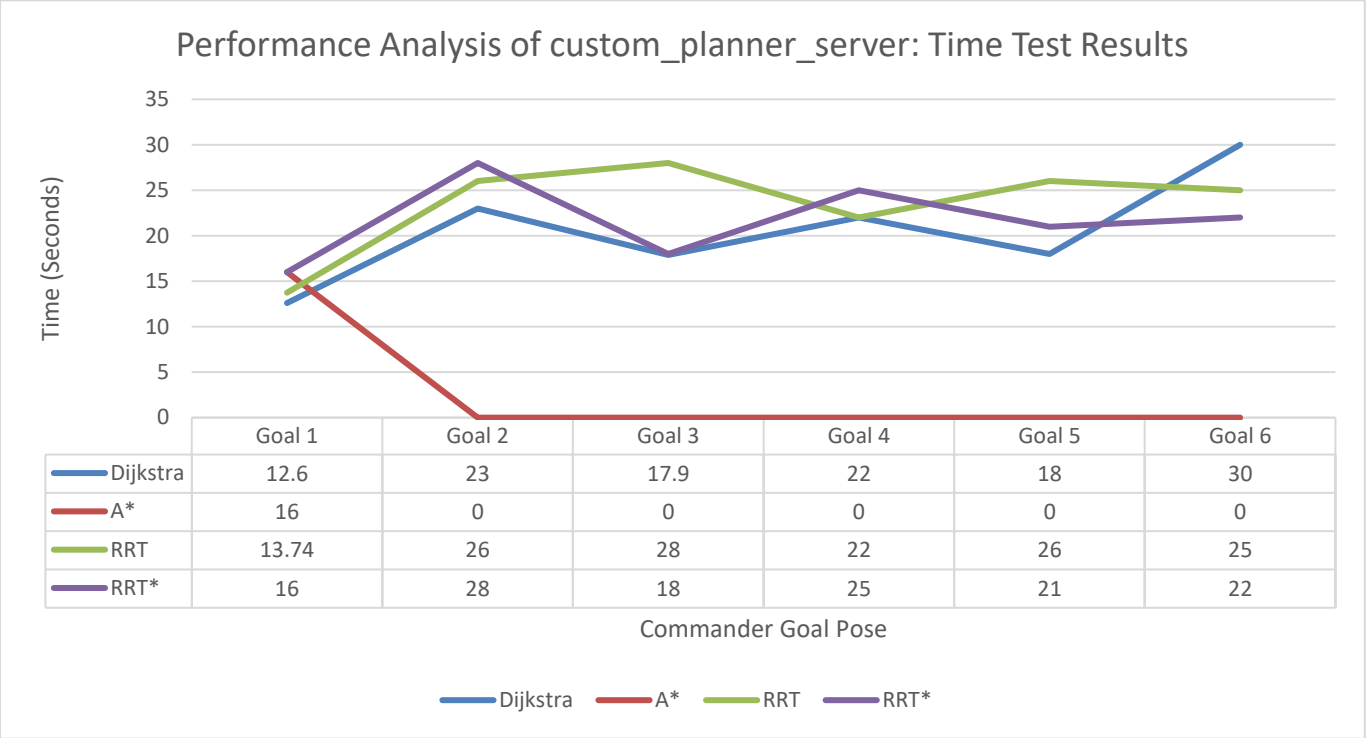



**FIGURE 29: CUSTOM_PLANNER SIMULATION TEST TIME ANALYSIS GRAPH**

The generated times, though based on visual observations and thus subject to some margin of error, provide valuable insights. They suggest that our algorithms perform admirably, closely matching or even surpassing the efficiency of navFN A-star and Dijkstra implementations in this environment. However, it's worth noting that our custom A* implementation exhibited difficulty in generating paths through cells with values exceeding the obstacle threshold of 200, resulting in paths through obstacles during simulation. While this issue warrants further analysis and rectification in future iterations, given the project's time constraints, it wasn't possible to delve deeper into resolving it.

Nevertheless, based on the integration test results and comparison with planners in the navigation2 stack, it's evident that our custom navigation stack houses the most effective algorithm, likely owing to the environment's small and static nature. Additionally, our newly implemented RRT and RRT* algorithms showcased commendable performance. It's notable that each of our navigation algorithms performed comparably to those provided by navfn, as evidenced in the video. Thus, overall, this implementation can be deemed largely successful.

# 7.0 System Evaluation and Conclusions

The implementation and customization of the ROS2 Navigation2 stack for autonomous navigation on the TurtleBot3 have represented a substantial undertaking. Across the project's duration, we delved into various facets of autonomous navigation, encompassing high-level path planning, local navigation, obstacle avoidance, and recovery behaviours. Leveraging the robustness inherent in the Navigation2 stack, we seamlessly integrated custom planner plugins, thereby augmenting the system's navigation prowess.

The Behaviour Tree (BT) architecture furnished by Navigation2 proved instrumental, offering a flexible and intuitive means of behaviour control that allowed for the concurrent execution of multiple navigation tasks. Harnessing behaviour trees facilitated the development of intricate navigation strategies, inclusive of robust recovery mechanisms, dynamic replanning strategies, and goal updates.

While challenges in integrating custom planner plugins arose due to compatibility constraints, we devised innovative solutions, such as service-based planning, to surmount these hurdles. The Custom Planner Server underscored the adaptability and expandability of our navigation system, enabling the seamless integration of novel planning algorithms.

## 7.1. System Evaluation

The performance of the autonomous navigation system underwent rigorous evaluation across simulated and real-world scenarios. Leveraging algorithms like A* and Dijkstra for global path planning, we scrutinized the system's efficiency and efficacy in navigating complex environments. Notably, the system showcased robustness in dynamic obstacle avoidance, dynamically adjusting its trajectory to circumvent collisions in real-time.

The integration of custom planner plugins, notably RRT and RRT*, broadened the system's capabilities, facilitating probabilistic path planning in demanding environments. Despite initial integration challenges, the Custom Planner Server emerged as a versatile solution, facilitating the incorporation of diverse planning algorithms.

## 7.2. Conclusions

In summary, the autonomous navigation system exhibited commendable performance in autonomously executing navigation tasks, fulfilling the project's goals of developing a robust and adaptable navigation framework for the TurtleBot3 platform. All the primary code referenced in this project is available in the GitLab repository. Please feel free to utilize it to facilitate any ROS-related work you undertake in the future [0].

## 7.3. **Future work**

While the system's performance met expectations, there remains ample room for enhancement. Future endeavours could focus on optimizing path planning algorithms through iterative experimentation, thereby enhancing navigation efficiency and robustness. Additionally, exploring additional sensor fusion options, incorporating machine learning techniques with cameras alongside lidar data, holds promise for improving obstacle detection capabilities. Implementing dynamic replanning strategies to adapt algorithms to evolving environments and unforeseen obstacles in real-time could further bolster navigation efficacy. Furthermore, there's potential to bridge the simulation-custom_planner_server navigation stack to real-world testing, enabling the evaluation of RRT and RRT* algorithms in real-world scenarios. Although time constraints precluded these advancements during the project's duration, the process should be straightforward, primarily involving parameter adjustments in the navigation launch file.

# References


[0] 40294886/turtlebot_navigation-cut_and_glue_trajectories. Gitlab repository. Available at: https://gitlab.eeecs.qub.ac.uk/40294886/turtlebot_navigation-cut_and_glue_trajectories.

[1] Gaspraetto, A., Boscariol, P., Lanzutti, A., & Vidoni, R. (March 2015). Path Planning and Trajectory Planning Algorithms: A General Overview. ResearchGate. Available at: https://www.researchgate.net/publication/282955967_Path_Planning_and_Trajectory_Planning_Algorithms_A_General_Overview.

[2] Stanford Artificial Intelligence Lab and Open AI. (June 2023). ROS (Robot Operating System) Information. Wikipedia. Available at: https://en.wikipedia.org/wiki/Robot_Operating_System.

[3] Turtlebot. (n.d.). RVIZ2 Information. Retrieved from https://turtlebot.github.io/turtlebot4-user-manual/software/rviz.html.

[4] Open Robotics. (n.d.). Gazebo Information. Retrieved from https://gazebosim.org/home.

[5] Górecki, R., Husarion. (n.d.). ROS Networking Information. Available at: https://husarion.com/tutorials/ros2-tutorials/6-robot-network/.

[6] ROBOIS. (n.d.). Turtlebot3 Quick Start Guide. Available at: https://emanual.robotis.com/docs/en/platform/turtlebot3/quick-start/.

[7] Open Robotics. (2024). ROS Humble Installation Guide. Available at: https://docs.ros.org/en/humble/Installation/Ubuntu-Install-Debians.html.

[8] Thomas, D. (2018). Colcon Build Tool Documentation. Available at: https://colcon.readthedocs.io/en/released/.

[9] Open Robotics. (2024). ROS2 Guide on Topics. Available at: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Nodes/Understanding-ROS2-Nodes.html.

[10] Open Robotics. (2024). ROS2 Guide on Services. Available at: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Services/Understanding-ROS2-Services.html.

[11] Open Robotics. (2024). ROS2 Guide on Actions. Available at: https://docs.ros.org/en/foxy/Tutorials/Beginner-CLI-Tools/Understanding-ROS2-Actions/Understanding-ROS2-Actions.html.

[12] Open Robotics. (2024). ROS2 Guide on Interfaces. Available at: https://docs.ros.org/en/foxy/Concepts/About-ROS-Interfaces.html.

[13] LaVelle, S. M. (2016). Planning Algorithms. Lavalle.pl. Available at: https://lavalle.pl/planning/.

[14] Dwijotomo, A., Abdul Rahmam, M. A., Mohammed Ariff, M. H., Zamzuri, H., & Wan Azree, W. M. H. (2019). Cartographer SLAM. MDPI. Available at: https://www.mdpi.com/2076-3417/10/1/347.

[15] Open Navigation. (2023). Available at: https://navigation.ros.org/.

[16] Writing a Planner Plugin for ROS2 Using Python. (n.d.). Available at: https://answers.ros.org/question/357237/writing-a-planner-plugin-for-ros2-using-python/.

[17] Straight-line Planner Tutorial. (2019.). https://docs.nav2.org/plugin_tutorials/docs/writing_new_nav2planner_plugin.html

[18] Tutorial on Behavior Trees. ( The Construct). https://app.theconstruct.ai/live_class/8e39921b-f25e-4bdf-b8af-d71cc0f560c9/

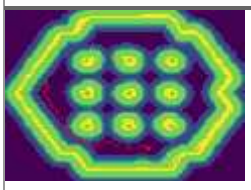
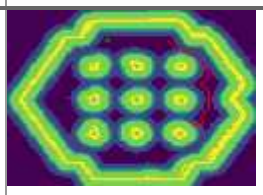
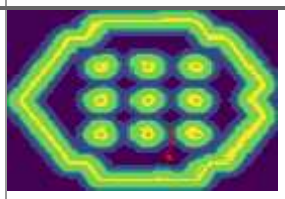
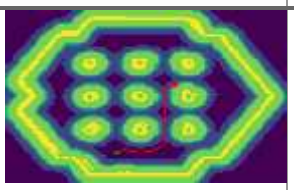
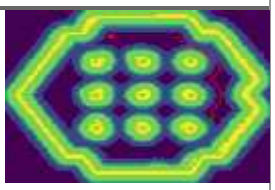
[19] AMCL ROS Wiki. (2020). http://wiki.ros.org/amcl

[20] Rigid Body Transformations. (n.d.). Available at: https://www.rosroboticslearning.com/rigid-body-transformations.

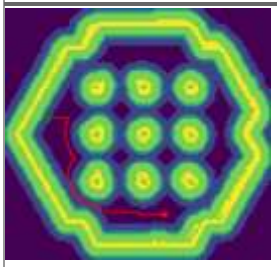
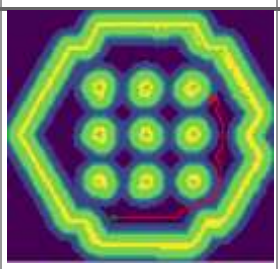
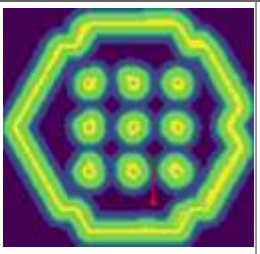
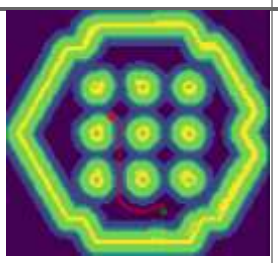
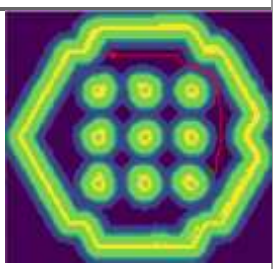
[21] Configuring Costmaps. (n.d.). Available at: https://navigation.ros.org/configuration/packages/configuring-costmaps.html.

# Appendix

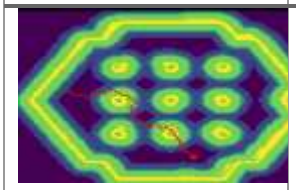
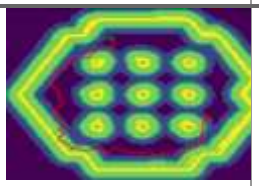
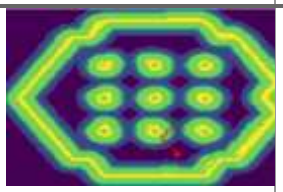
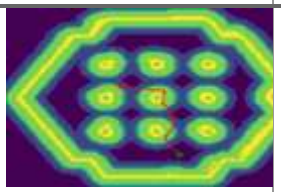
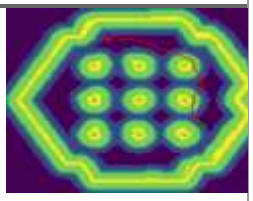
**[A1] – [A5]:** These are the visualizations of the paths generated by the dijkstra Integration Tests

| [A1] | [A2] | [A3] | [A4] | [A5] |
|---|---|---|---|---|
|  |  |  |  |  |

[**A6] – [A10]:** These are the visualizations of the paths generated by A* Integration Tests.

| [A6] | [A7] | [A8] | [A9] | [A10] |
|---|---|---|---|---|
|  |  |  |  |  |

**[A11] – [A15]:** These are the visualizations of the RRT Integration Tests

| [A11] | [A12] | [A13] | [A14] | [A15] |
|---|---|---|---|---|
|  |  |  |  |  |

**[A16] – [A20]:** These are the visualizations of the RRT* Integration Tests

| [A16] | [A17] | [A18] | [A19] | [A20] |
|---|---|---|---|---|