POLITECNICO DI TORINO

Master's Degree in Mechatronic Engineering

Master's Degree Thesis

Development of an automated benchmark for the analysis of Nav2 controllers

Supervisors

Prof. Marcello CHIABERGE

Eng. Antonio MARANGI

Candidate

Federica SCHENA

APRIL 2024

A Nonna Rita,
che è qui con me in questo giorno speciale.

Acknowledgements

I would like to thank my supervisors Eng. Antonio Marangi and Prof. Marcello Chiaberge, because they believed in me and gave me the opportunity to challenge myself, making my passion on robotics growing more and more.

Also, I wish to extend my special thanks to Eng. Andrea Bertaia Segato, Eng. Simone Monteleone, Marta, Claudio, Alfredo and all of my colleagues that made me feel welcome from the very first moment.

Summary

Demographic trends highlight a significant absolute and relative increase in the population segment aged over seventy-five, with substantial impacts, particularly on the prevalence of many chronic degenerative conditions. Moreover, as stated by the European Council in 2022, the 27% of population in EU over the age of 16 had some form of disability.

The usage of smart wheelchairs has been introduced in order to allow people with reduced mobility to move in public places in a safer and suitable way. Alba Robot srl is a start-up based in Turin, which is developing a cutting-edge micro-mobility platform named SEDIA (Seat Designed for Intelligent Autonomous). The aim of this company is to transform people transportation providing a B2B platform that combines the most innovative technologies, such as AI, IoT, Robotics and Automotive by the usage of self-driving electric vehicles.
This product is composed of a hardware layer and a software layer. The autonomous guide's physical components, which basically consist in its sensors, actuators, motors and all the other sensory devices for environmental perception, are part of the hardware layer. Instead, the software layer is characterized by all the robotic system's higher-level algorithms, decision making processes and real-time coordination. The autonomous navigation system integrated into the wheelchair is built on top of ROS 2 and Nav2. In fact, the company exploits several plugins provided by the meta-package Nav2, in order to customize its own applications or algorithms: costmap layers, planners, controllers, behavior trees and behavior plugins.
Above all, the controllers are particularly crucial for the navigation task, because the way in which the robot deal with this problem depends on them. Nav2 provides 4 different controller plugins: DWB, RPP, TEB and MPPI.

DWB controller is highly configurable through the use of plugins. It implements 3 different types of plugins: plugin-based critics that can be dynamically reconfigured, reweighted and tuned, plugin-based trajectory generation techniques and plugin implementations for common use.

RPP controller implements a variant on the pure pursuit algorithm to track a path, which is called Regulated Pure Pursuit Algorithm. It also implements the basics behind the Adaptative Pure Pursuit Algorithm to vary lookahead distances by current speed.

TEB controller implements the Time Elastic Band method, that locally optimizes the robot's trajectory with respect to trajectory execution time, separation from obstacles and compliance with kynodynamic constraints at runtime.

MPPI controller is a predictive controller that implements the Model Predictive Path Integral algorithm to track a path with adaptative collision avoidance. It contains plugin-based critic functions to impact the behavior algorithm.

Each controller has noticeably different benefits and drawbacks that does not allow a perfectly adaptable navigation to every possible condition The aim of this paper is to provide an automatic benchmark that is capable of deeply analyze every benefit and drawback of each controller, so that it is possible to acknowledge which one is the best that perfectly deals with any task. The simulation tools that have been used are Rviz and Gazebo, while the programming environment is Visual Studio Code and the chosen programming language is Python.
The controllers have been tested in 5 different use cases, concerning simple navigation in an empty environment, obstacle avoidance, sharp bending and abortion of impossible tasks. The analysis evaluates the kinematics of the robot, the smoothness of the path, the capability of the local planner of following the global planner path avoiding detour and of completing a task avoiding collisions. These parameters have been obtained by observing each controller completing one task for 4 times.

In conclusion, the analysis shows that there is no controller working better than the others, because each of them demonstrates remarkable qualities under some conditions. This means that the controllers provided by Nav2 should be merged in order to implement a new customized one.

Table of Contents

List of Tables

List of Figures

Acronyms

AI

Artificial Intelligence

AMCL

Adaptative Monte Carlo Localization

API

Application Programming Interface

APP

Adaptative Pure Pursuit

B2B

Business to Business

BLND

passenger is blind or has reduced vision

BT

Behaviour Tree

CEO

Chief Executive Officer

CPU

Central Processing Unit

DDS

Data Distribution Service

DEAF

passenger is deaf or hard of hearing

DPNA

disabled passenger with intellectual or developmental disability needing assistance

DWA

Dynamic Window Approach

EC

european community

EU

european union

Eurostat

Statistical Office of the European Union

G3M

Global Multimap Mission Manager

GPU

Graphic Processing Unit

HMI

Human-Machine Interface

HJB

Hamilton Jacobi Bellman

ID

Identification

IoT

Internet of Things

ISTAT

Istituto Nazionale di Statistica

LiDAR

Light Detection and Ranging

MIT

Massachusetts Institute of Technology

MPC

Model Predictive Control

MPPI

Model Predictive Path Integral

Nav2

Navigation for Autonomous Vehicle

PCB

printed circuit board

PP

Pure Pursuit

PRM

People with Reduced Mobility

RGB

Red, Green, Blue

RGBD

Red, Green, Blue, Depth

ROS

robot operating system

RPP

Regulated Pure Pursuit

srl

Società a Responsabilità Limitata

SLAM

Simultaneous Localization and Mapping

STVL

Spatio-Temporal Voxel Layer

TEB

Time Elastic Band

TF

Transformed

ToF

Time of Flight

WCHC

wheelchair required; passenger cannot walk any distance and will require the aisle chair to board

WCHR

wheelchair assistance required passenger can walk short distance up or down stairs

WCHS

wheelchair assistance required; passenger can walk short distance, but not up or down stairs

XML

eXtensible Markup Language

Introduction

The concept of disability cannot be easily defined, since it encompasses a heterogeneous range of physical, cognitive, and sensory diversities and abilities.
Until few decades ago, disability was considered only in terms of intrinsic limits of the individual and it was exclusively seen as a medical "issue" that required an individual intervention.
The paradigm that has now been applied is called *'Social Model of Disability'*. It was created in the 80s in order to contrast the traditional medical model: according to this, disability is the result of an interaction between the level of physical or sensory or cognitive or mental limitation of the individual and the environment in which they live. Therefore, disability is largely the result of social factors: if the environment is not accessible or inclusive, disability increases.
In May 2001, the World Health Organization (WHO) approved the International Classification of Functioning, Disability and Health (ICF) with the aim of integrating both the medical and social models. Despite the fact that this is not a condition exclusively associated to elder part of population, disability is predominantly observable in the over seventy-five age group, where chronicity, morbidity, functional impairment, polypharmacy, and socio-health issues play a determining role. Demographic trends highlight a significant absolute and relative increase in the population segment aged over seventy-five (+25 %, equivalent to more than 1,400,000 individuals in the next 10 years), with substantial impacts, particularly on the prevalence of many chronic degenerative conditions. Assisting non-self-sufficient individuals, predominantly (though not exclusively) elderly, has thus become one of the inadequately addressed social emergencies in our country, which is, incidentally, one of the most long-lived in the world [1].
According to ISTAT, in Italy there are 3,100,000 people (5.2% of population) with severe limitations in ordinary activities and half of them (1,500,000) are 75 years old or older.
As stated by the European Council in 2022, the 27% of population in EU over the age of 16 had some form of disability. According to Eurostat estimates, that equals to 101 million people or one in four people adults in the EU. Moreover, the countries that had the highest share of people with disabilities were Latvia (38.5%),

Denmark (36.1%) and Portugal (34%). It is possible to notice that most people with disability are over 65 years old and females[1].

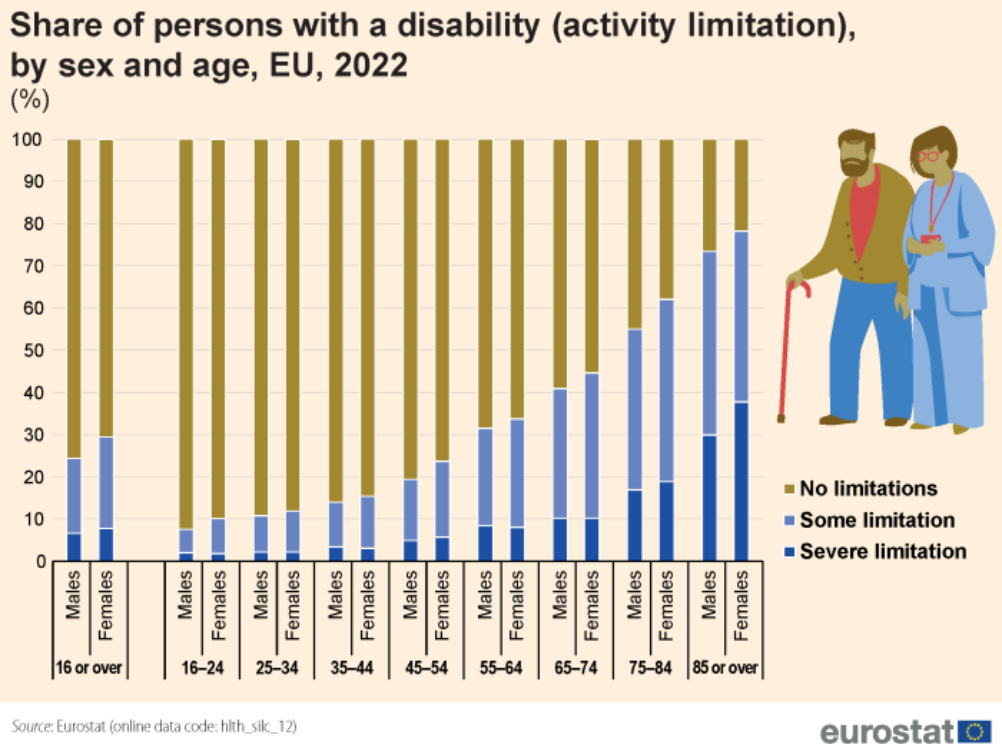


Figure 1: Disability by sex and age

Furthermore, people's life expectancy is increasing throughout the world as a result of improved living standards and medical advances. In the EU, life expectancy at birth is projected to increase from 76.7 years in 2010 to 84.6 years in 2060 for males and from 82.5 years in 2010 to 89.1 years in 2060 for females (EC 2012). It is predicted that the EU population aged 65 and above will almost double by 2060, rising from 87.5 million in 2010 to 152.6 million (EC 2011). The natural ageing process is accompanied by physiological changes which can have significant consequences for mobility [2].
People using walkers, wheelchairs, and crutches or people dealing with debilitating arthritis as well as hip and knee conditions may have difficulty navigating a home, a neighbourhood, or a community. Lack of sidewalks, barriers to entranceways, narrow hallways, the presence of steps, and busy streets can all make mobility more difficult and less safe [3].
According to Regulation (EC) 1107/2006, "Person with disabilities and person with reduced mobility" means any person who has a permanent or temporary

physical, mental, intellectual or sensory impairment which, in interaction with various barriers, may hinder their full and effective use of transport on an equal basis with other passengers or whose mobility when using transport is reduced due to age. Standards used to define different types of disabilities and corresponding needs are:

- BLND: Visually impaired or blind persons
- WCHR: Passengers who cannot walk long distances, but can go up and down steps and move around independently
- WCHS: Passengers who cannot walk long distances and cannot go up and down steps, but can move around inside a public transport independently
- WCHC: Completely immobile passengers, who are not self-sufficient inside a public transport and need assistance at all times
- DEAF: People with hearing disabilities
- DPNA: People with mental or behavioural disabilities

The usage of smart wheelchairs has been introduced in order to allow people with reduced mobility to move in public places in a safer and suitable way.
In Alba Robot's mobility platform autonomy and artificial intelligence are integrated to make people more independent and to provide public buildings (e.g. hospitals, airports and museums) with new solutions for autonomous and assisted mobility. Alba Robot wheelchairs are entirely produced by the company, starting from the PCBs and electric circuit to the chassis. One of the most important parts is the controller, which is used to make the robot able to navigate autonomously in the space. The controllers that are already provided by the Nav2 ROS2 package give different results depending on the environment: some are perfectly capable of following prescribed paths but cannot avoid obstacles, while others have the opposite behaviour. The main object of this thesis is to analyze in depth all of the Nav2 controllers, in order to acknowledge every perks and drawbacks of each of them in every condition.
The structure of this work will be the following:

- **Chapter 1** defines the basic concepts of mobile robot navigation, Nav2 structure and tools;
- **Chapter 2** explains more in details the objective of this project;
- **Chapter 3** describes the method and the metrics that has been used to analyse in a qualitative way the controllers already provided;
- **Chapter 4** shows the obtained results and future implementation of the new customized controller.

Chapter 1

State of the Art

1.1 Navigation fundamental concepts

Mobile robots can move autonomously, without human assistance. When a robot uses an assisting perception system to decide for itself what has to be done to complete a task, that robot is said to be autonomous. In order to coordinate all of the components that make up the robot, it also requires a control system or cognition unit.
Navigation, perception, cognition, and locomotion are the foundational domains of mobile robotics:

- kinematics, dynamics, control theory, and mechanism comprehension are necessary to solve **locomotion** difficulties;
- specialized domains including computer vision and sensor technology, as well as signal processing, are involved in **perception**;
- the analysis of the input data from sensors and the corresponding action taken to complete the tasks of the mobile robots are **cognition** responsibilities;
- **navigation** requires knowledge of planning algorithms, information theory, and artificial intelligence

Navigation, indeed, is the most important aspect in the design of a mobile robot, since it allows motion from one place to another in known or unknown environment, considering the data given by sensors.
Mobile robot navigation consists in the following tasks:

- localization and mapping,
- path planning,
- obstacle avoidance

1.1.1 Localization

Localization plays a fundamental role, since it allows to know the robot's absolute position as well as the its relative position with respect to the target. Moreover, it is essential for the design of the map of the world, which make the robot able to plan a path, recover its position and detect if it has reached the target location. Engineers and researchers have elaborated a diversity of systems, sensors and methods for mobile robot positioning, such as: odometry, inertial navigation, magnetic compasses, active beacons, global positioning systems, landmark navigation and model matching. Another important aspect of localization that has to be mentioned is the accuracy of the map representation, because it must meet the accuracy of the data returned by the robot's sensors. Map representation methods are various, but the most commonly used are: probabilistic map-based localization, Markov localization and Monte Carlo localization.

1.1.2 Path Planning

The aim of path planning is finding the best path that allows the robot to reach the target position from an initial configuration without collisions, neglecting the temporal evolution of motion. Velocities and acceleration are not considered. It defines only one aspect of trajectory planning, which is more completed. As a matter of fact, it calculates the force inputs *u(t)* to move the actuators in such a way that the robot can follow the computed trajectory *q(t)*. Path planning and trajectory planning are included in the motion planning, which is a much vaster concept. There are several different algorithms that can solve the motion planning, which have as main object finding a solution, that is, a feasible path. Over time, they have been complemented by optimization techniques that have sought to minimize the distance traveled by the mobile robot. The earliest work on robot planning was carried out in the late 1960s and early 1970s. Although, the first techniques presented some problems (e.g. too much memory needed to analyze the workspace, the existence of local minima in the potential field that could lead the planner to be trapped in those).
The most used ones are the heuristic planners:

- **A***, which searches all the possible paths to the target and chooses the one with the smallest cost (shortest time, time distance travelled, etc.);

- **Greedy search**, that looks for the locally optimal choice at each stage in order to obtain the global optimum. It is clear that this kind of algorithm cannot provide the best solution, but only a very good approximation of it in the quickest time;

- **Dijkstra's algorithm**, that searches the shortest path between nodes of a graph (roadmaps, discrete workspace, etc);
- **D***, which defines a path starting from the target and working back to the start using a method that is similar to Dijkstra's one.

These algorithms have been modified, focusing on the optimization of several parameters, such as **time needed** (related to productivity), **jerk** (related to the quality of work, accuracy and equipment maintenance), **energy consumption** and **motor effort** (both related to savings).

1.1.3 Obstacle avoidance

A good motion planner must be capable of detecting collisions, between the robot and an obstacle into the environment, and avoiding it, stopping the robot or changing its trajectory. Obstacle avoidance algorithms are based on obstacle detection and obstacle avoidance itself [4]. They can be:

- **map-based**, characterized by the usage of geometric or topological models of the environment. By knowing its own current position in each moment, the robot can calculate distances, that is detecting collisions;
- **mapless-based**, which doesn't make use of any model, but relies only on the system of sensors of the robot to observe the environment.

The Figure 1.1 [5] summarizes all of the main concepts that compose mobile robotics.

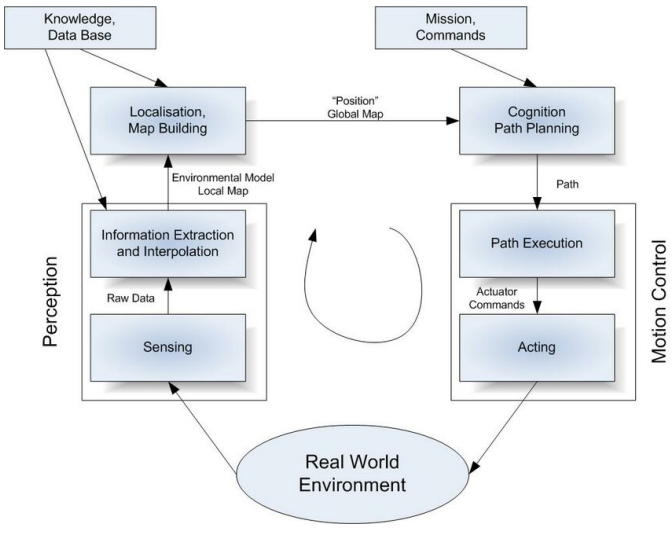


Figure 1.1: Control scheme for mobile robots

1.2 ROS

The Robot Operating System (ROS) is a set of software libraries and tools for building robot applications, created in 2007 by the robotics incubator Willow Garage [6]. Since it has been designed by researchers for researchers, it lacked of some features needed for the development of robust commercial robots [7]. In order to face this problem, ROS2 has been introduced, while pursuing its legacy of a broad ecosystem of distributed, community driven projects [8]. It addresses long-standing tasks concerning security, embedded and real-time support and operating in challenging networking environments.
The software ecosystem is divided into three categories:

- **middleware**: also known as the *plumbing*, it handles message parsers and network APIs to facilitate communication between components.

- **algorithms**: ROS2 provides various algorithms generally used when defining robotics applications, such as perception, SLAM, planning, etc.

- **developer tools**: ROS2 includes a suite of command line and graphical tools for configuration, launch, introspection, visualization, debugging, simulation and logging. There is also a large suite of tools for source management, build processes, and distribution.

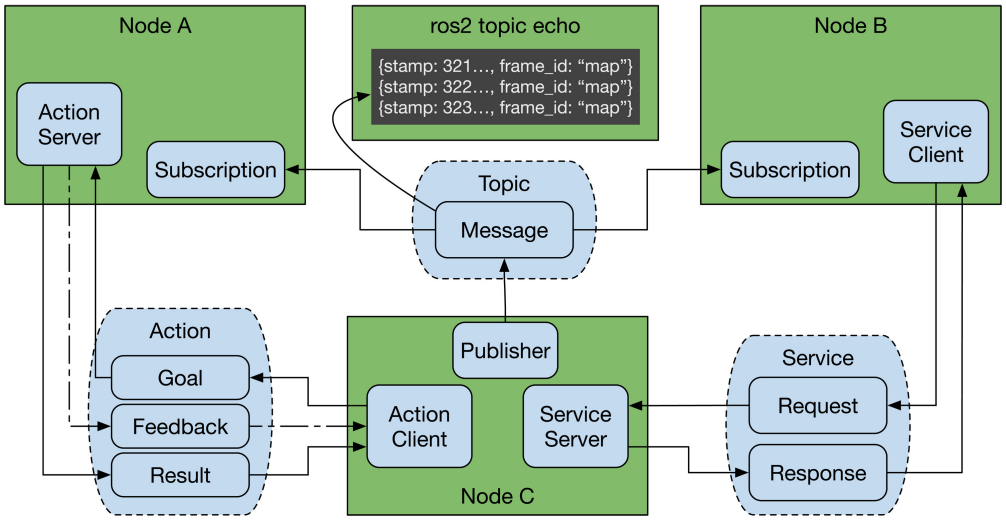


Figure 1.2: ROS 2 architecture

1.2.1 Communication patterns

The ROS 2 API provide access to communication patterns: notably topics, services and actions, which are organized under the concept of *node*. ROS 2 also provides APIs for parameters, timers, launch and other auxiliary tools which can be used to design a robotic system.

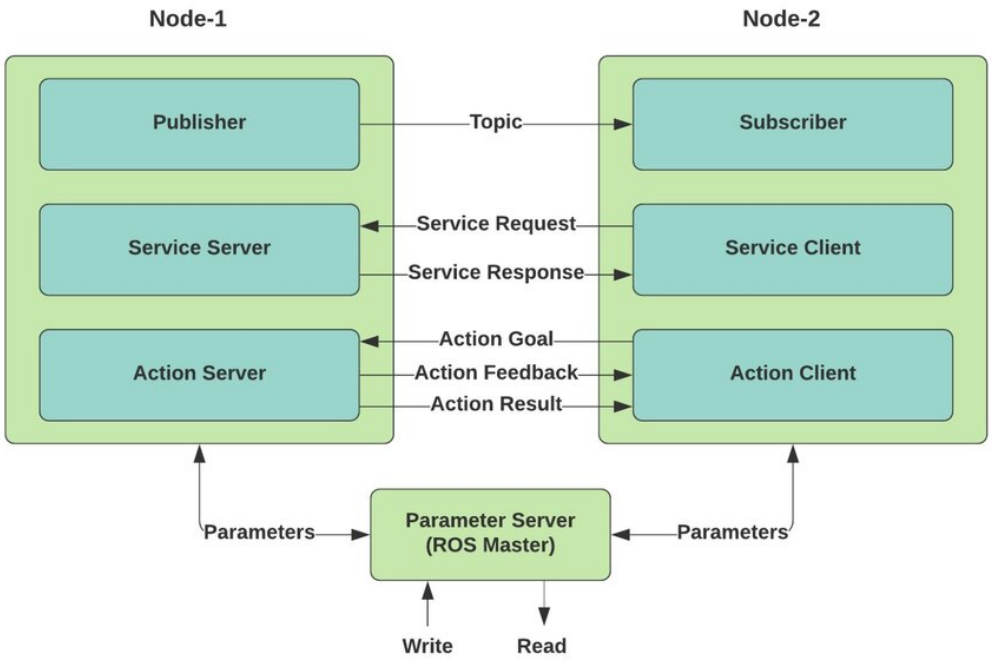


Figure 1.3: ROS 2 communication patterns

Topic

Topics are an asynchronous message passing framework. ROS 2 provides the same publish-subscribe functionality: it is based on the interaction between two actors (*publisher* and *subscriber*). In details, the publisher is responsible of generating and sending messages on a particular topic without knowing what is listening. Messages can contain information, events or notification. The subscriber expresses interest in receiving and sending messages related to a particular channel. Thus, when the publisher sends a message on that topic, all interested subscribers automatically receive the notification and can process the message accordingly. Nevertheless, in ROS 2 this functionality focuses on using asynchronous messaging to organize a system using strongly typed interfaces: the endpoints are organized in a computational graph under the concept of a *node*. The node is an important organizational unit which allows a user to reason about a complex system.

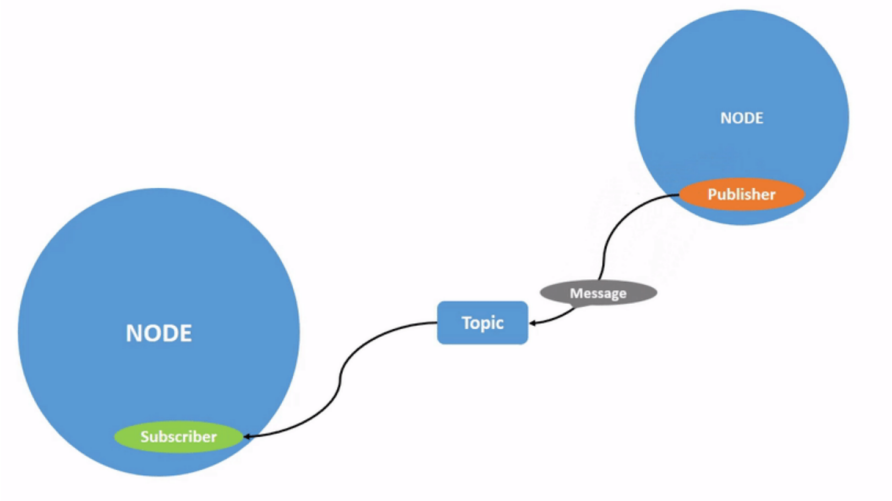


Figure 1.4: ROS 2 topic communication pattern

Services

ROS 2 also provides a pattern based on *request-response*, which is called **service**. This type of communication provides easy data association, which can be useful when ensuring the completeness or the receiving of a task. Moreover, ROS 2 allows a service client's process to not be blocked during a call. Services are organized under a node for organization and introspection, allowing a subsystem's interfaces to appear together in system diagnostics.

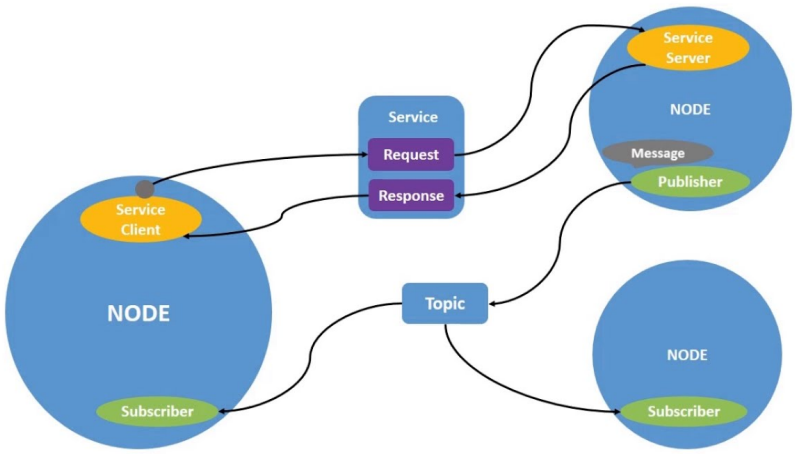


Figure 1.5: ROS 2 service communication pattern

Actions

Actions are asynchronous, goal-oriented communication interfaces that can be terminated and have a request, response, and periodic feedback. They are generally used in long-running tasks, such as navigation or manipulation. Long tasks are controlled by action servers, which are quite similar to canonical servers: a client requests the completion of a task, although, its duration may be much longer. It is possible to block all the other processes until the task is completed and to occasionally check the state of the action through feedbacks.
Feedbacks and results can be collected, spinning the client node to process callback groups, both synchronously, registering callbacks with the action of the client, or asynchronously, requesting information from the shared future object.
The Figure 1.6 shows the scheme of an action on ROS 2.
Additionally, actions are non-blocking and organized under the node as well as services.

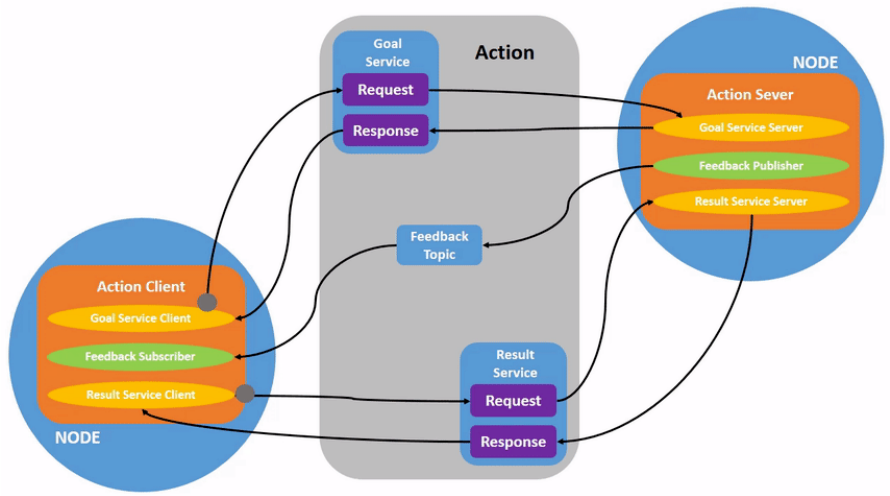


Figure 1.6: ROS2 action communication pattern

1.2.2 Middleware Architecture

The architecture of ROS 2 consists of several abstraction layers distributed across many decoupled packages. This architecture gives the possibility to have multiple solutions for a single required functionality.

Abstraction Layers

During development, abstraction layers are often concealed behind the client library, and developers need to be aware of them only in the case of exceptionally application-specific requirements. For most users, there is simply the client libraries to use. The primary communication APIs are accessible through the client libraries. To be more idiomatic and to utilize language-specific features, they are customized for each programming language. No matter how the system is divided up between compute resources, whether they are on the same computer, in a separate process, or even in a different process, communication remains unaffected. With only few modifications to the source code, a user can distribute their application across numerous computers and processes and even make advantage of cloud computing resources. Over the internet, ROS 2 can establish a connection to cloud resources. The client libraries rely on the `rcl` intermediate interface, which gives all of the client libraries access to shared functionality. All of the client libraries use this library written in C, however it is not necessary.
The essential communication interfaces are provided by the middleware abstraction layer `rmw` (ROS MiddleWare). Users can select different middleware technologies and `rmw` implementations according to a range of limitations such as software license, performance, or supported platforms.
This abstraction layer provides flexibility to ROS 2, allowing it to change over with minimal impact to the systems built on top of it.
The network interfaces (e.g. topics,services, actions) are defined by *Message Types* using an *Interface Description Language* (IDL): ROS 2 uses `.msg` files or `.idl` files.

Architectural Node Patterns

ROS 2 provides a pattern for managing the lifecycle of nodes. They transition through a state machine with **Unconfigured, Inactive, Active** and **Finalized** states. This functionality is essential to coordinate various parts of the distributed asynchronous system.
The machine or the process in which the node should be put depends on the way in which it is used in the larger system. For example, nodes that are written as *components*, can be allocated to any processes as a configuration.

1.2.3 Lifecycle nodes and bonds

Lifecycle nodes contain state machine transitions for bringing up and tearing down of servers. This guarantees freedom to nodes developers on the management of life cycle functionality.

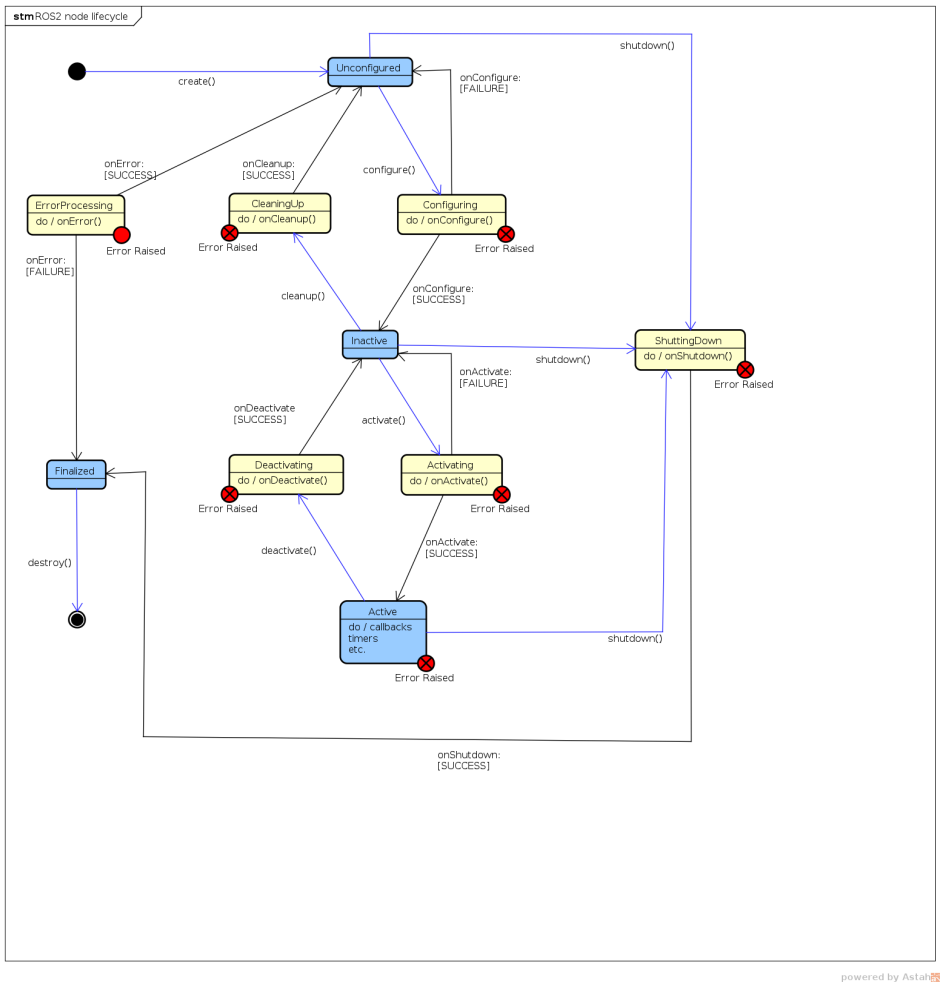


Figure 1.7: Node lifecycle

As shown in Figure 1.7 [9], there four **primary states**:

- **Unconfigured**
- **Inactive**
- **Active**
- **Finalized**

It is possible to transition out of a primary state only when action from an external supervisory process occurs.
There are also six **transition states**:

- **Configuring**

- **CleaningUp**
- **ShuttingDown**
- **Activating**
- **Deactivating**
- **ErrorProcessing**

In the transitions states logic will be executed to determine if transition is successful. Success of failure are communicated via lifecycle management interface to lifecycle management software.
The transitions that undergo supervisory processes are seven:

- **Create**
- **Configure**
- **CleanUp**
- **Activate**
- **Deactivate**
- **Shutdown**
- **Destroy**

At first, the started node is in **unconfigured** state: this means that it is processing the node's constructor, which does not contain any ROS networking setup or parameters. By the launch of the system or lifecycle manager, the node's state must be transitioned to **inactive**. Now, it is possible to **activate** it: when in this state, the node is fully setup to run. The **configuration** stage allow the set up of ROS parameters, ROS networking interfaces and all dynamically allocated memory. The **activation** stage activates ROS networking interfaces and make all the states ready to process information. Finally, in order to set the node in the **finalized** state, it is necessary to transition into the deactivating, cleaning up and shutting down: networking interfaces are deactivated and stop processing, deallocate memory, exit cleanly.

1.3 Behaviour Trees

Behavior trees have become more and more used in the robotic industry, replacing the traditional Hierchical Functional State Machine. A **behaviour tree** is a directed rooted tree where the internal nodes are called *control flow nodes* and leaf nodes are called *execution nodes* [10]. It creates a more scalable and understandable framework for defining applications characterized by multiple steps or states. Each connected node is a *child* or a *parent*: a node without parents is the **root**, while all the other nodes have one parent. The control flow nodes have at least one child. As shown in Figure 1.8, generally the children are placed below the parents.

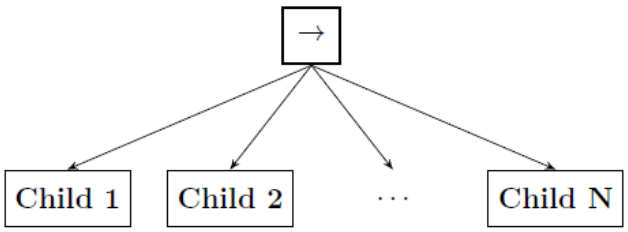


Figure 1.8: Graphical representation of a sequence node with N children

There are four types of control flow nodes:

- fallback
- sequence
- parallel
- decorator

The execution nodes, instead, are divided into two categories:

- action
- condition

All of the described nodes are explained in details in Figure 1.9.

Node type	Symbol	Succeeds	Fails	Running
Fallback	?	If one child succeeds	If all children fail	If one child returns Running
Sequence	→	If all children succeed	If one child fails	If one child returns Running
Parallel	⇉	If $\geq M$ children succeed	If $> N - M$ children fail	else
Action	text	Upon completion	If impossible to complete	During completion
Condition	text	If true	If false	Never
Decorator	◊	Custom	Custom	Custom

Figure 1.9: The node types of a BT

A BT starts its execution from the root node, which sends *ticks* signals to the children. If and only if a child receives a tick, a node is executed. The child returns a *Running* status to the parent if its execution has not finished yet, a *Success* status if it has achieved its goal, a *Failure* status otherwise.
Overall, the formal structure provided by behaviour trees is very useful in order to create complex as well as verifiable systems.

1.4 Nav2

ROS 2 navigation stack metapackage Nav2 is a framework largely used for autonomous navigation of robots. It creates customized and intelligent behaviour using behaviour trees and coordinated independent modular servers: the servers communicate with each other with a behaviour tree (BT) through a ROS interface (such as an action server or service). It is possible for a robot to have various different behaviour trees associated to the accomplishment of multiple types of unique tasks. The structure of the this package is shown in Figure 1.10.

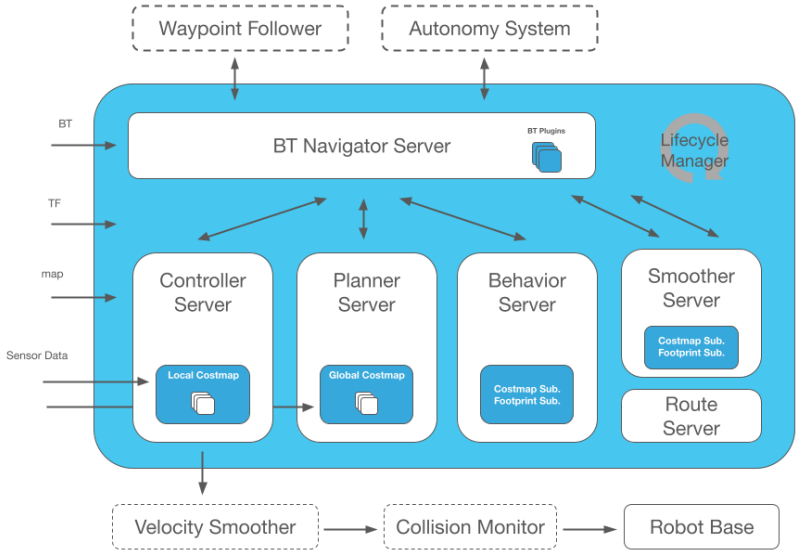


Figure 1.10: Nav2 architecture

The design took into account the requirements for robotics products needs: safety, security, and determinism without loss of generality. Navigation2 is designed on top of ROS 2 to handle functional safety standards and determinism. ROS 2 is built on Data Distribution Service (DDS) communication standard, which is used in critical infrastructure such as aircraft, missile systems, and financial systems. By utilizing DDS security features, ROS 2 enables users to safely transmit

information both within the robot and to cloud services without the need for a dedicated network.
Furthermore, Managed Nodes—referred to as Lifecycle Nodes—are introduced in ROS 2. A lifecycle node implements a server structure with distinct state transitions from instantiation through destruction. This server is created when the program is launched, however it waits for external stimulus before transitioning through a deterministic bringup process. In case of shutdown or error, the server passes from active to finalized state. All servers in Navigation2 make use of managed nodes for deterministic program lifecycle management and memory allocations.
Navigation2 is also highly modular and easy to reconfigure and select at run-time, in order to create a navigation system that can work with various robots in different environments. It uses a *behavior tree navigator* and *task-specific asynchronous servers*. Each server is a ROS 2 node hosting algorithms plugins, that are libraries dynamically loaded at run-time.
The *Behavior Tree Navigator* activates and tracks progress of planner, controller and recovery servers to facilitate navigation, by using a behavior tree to coordinate the navigation tasks. It is possible to create customized navigation behavior by modifying a behavior tree, stored as an XML. BTs are easily reconfigurable with different control flow and condition node types, requiring no programming. A recovery system is then created using BT action node statuses and control flow nodes, in order to trigger unique response after the failure of a specific server.
Using ROS 2, the behavior tree nodes in the navigator can call long running asynchronous servers in other processor cores. A navigation system can efficiently use a much larger quantity of compute resources when multi-core processors are used. To load BT node plugins, the Navigation2 BT navigator uses dynamic libraries. This pattern allows the creation and loading of reusable primitive nodes with a behavior tree XML at run-time without linking to the navigator itself.
Moreover, these nodes can call remote servers on other CPUs in any language with ROS 2 client library support.
Each task-specific server is designed in order to host a ROS 2 server, environmental model, and run-time selected algorithm plugin. They are modular, so that none or many of them can run at the same time to compute actions.
The ROS 2 server is the entry-point for BT navigator nodes and handles cancellation, preemption, or new information requests. These requests are forwarded to the algorithm plugin to complete their task.
Behavior Trees are mostly used to model complex or multi-steps tasks, where navigating to a position is actually a node of that larger task. The BT library used in Nav2 is *BehaviorTree.CPP*.
Nav2 exploits various tools:

- **Map Server** loads and stores maps

- **AMCL** localizes the robot on the map
- **Nav2 planner server** plans a path from a starting point A to a generic point B, avoiding obstacles
- **Nav2 controller server** controls the robot while following the trajectory
- **Nav2 smoother** smooths planned trajectories, in order to guarantee continuity and feasibility
- **Nav2 costmap 2D** creates a costmap representation of the world using sensors data
- **Nav2 behavior trees and BT navigator** build complex robot behaviour through the usage of behaviour trees
- **Nav2 recoveries** when failures occur, it generates recovery behaviours
- **Nav2 waypoint follower** makes the robot following sequential waypoints
- **Nav2 lifecycle manager** manages servers' lifecycle and watchdog
- **Nav2 core** consists of the plugins used by the user in order to customize algorithms and behaviours
- **Collision monitor** detects imminent collisions or danger monitoring raw sensors data
- **Simple commander** is Python API, which allows interaction with Nav2
- **Velocity smoother** smooths output velocities to guarantee dynamic feasibility of commands

1.4.1 Navigation Servers

The action servers are four:

- The core of the navigation task consists in **planners** and **controllers**
- **recoveries** are used to make the system fault-tolerant: in such a way the robot becomes capable of dealing with diverse forms of issues
- **smoothers** improve the quality of the planned paths

These action servers are used to host multiple algorithm plugins to complete various task and the environmental representation used by those algorithms to compute their output.
The planner, smoother and controller servers are configured at runtime, with the names (aliases) and types (pluginlib names) of algorithm to use. These servers then expose an action interface corresponding to their task. The action server is called to process its task when the behaviour tree ticks the corresponding BT node. The action server callback will call the chosen algorithm by its name, which corresponds to a specific algorithm. On this way, the user is able to use classes of algorithms that abstract the algorithm used in the BT.
In the behavior server, each of the behaviours is characterized by its name, while each plugin will expose its own special action server. This is done because of the wide variety of behaviour actions that may be created.
In addition to that, the behavior server contains a costmap subscriber to the local costmap, which receives real-time updates from the controller server, in order to compute its task. On this way, it is possible to avoide the creation of multiple instances of the local costmap, which can be computationally expensive.
Since BT nodes are trivial plugins calling an action, it is possible to create multiple new BT nodes in order to call different action servers with different action types.
All of these servers may be customized or replaced by the user. Each server contains an environmental model relevant for their operation, a network interfaces to ROS 2, and a set of algorithm plugins to be defined at run-time. Those pluings are shown in Figure 1.11.

Type	**Plugin**	**Description**
Control	DWB Controller	Configurable DWA controller
	TEB Controller	Timed-Elastic-Bands controller
Costmap	**Inflation Layer**	Inflate obstacles in costmap
	Non-Persist. Voxel	Maintain only recent voxels
	Obstacle Layer	Raycast 2D obstacles
	STVL	Temporal 3D sparse voxel grid
	Static Layer	Loads static map into costmap
	Voxel Layer	Raycast 3D obstacles
Planner	**NavFn Planner**	Holonomic A* expansion
Recovery	Back Up	Back out of sticky situations
	Clear Costmap	Clear erroneous measurements
	Spin	Rotate to clear free space
	Wait	Waitout time based obstacles

Figure 1.11: Nav2 provided plugins

Planners and controllers

The task of the *global planner* is to compute the shortest route to a goal, while the **controller**, also known as *local planner* in ROS 1, uses local information to compute the best local path and control signals.
The planner has access to a global environmental representation and sensor data, which are stored into its buffer. It can be written for several reasons, such as the computation of the shortest path, of the complete coverage path, of path along sparse or predefined routes, etc.
The controller, instead, can be written in order to follow a path, board an elevator, interface with a tool, etc.

Behaviors

Recovery behaviors' goal is to autonomously handle unknown or failure conditions. For example, faults in the perception system could lead to an environmental representation full of fake obstacles. In this case, the *clear costmap recovery* would be triggered to allow the robot to move. Another significant example, that show the importance of this server is when the robot is stuck due to dynamic obstacles or poor control. In this case, instead, backing up or spinning in place, allow the robot to move into free space.
Recovery behaviors are called from the leaves in the behavior tree and carried out by recovery server. By convention, these behaviors are ordered from the most conservative to the most aggressive actions.
Moreover, they can be specific to its subtree (e.g. global planner or controller) or system level in the subtree that contains only recoveries in case of system failures. The most used ones generally are: **Clear Costmap**, which is a recovery to clear costmap layers in case of perception system failure, **Spin**, to clear out free space and steer the robot away from possible failures (e.g. the robot perceives itself to be too etrapped to back out), **Wait**, to wait in case of time-base obstacle (e.g. human traffic or collecting more sensor data).

1.4.2 State Estimation

For state estimation, Navigation2 follows ROS transformation tree standard to exploit many modern tools. This includes **Robot Localization**, which is a general sensor fusion solution using Extended or Unscented Kalman Filters.
It is used to provide smoothed base odometry from N arbitrary sources, which often includes wheel odometry, multiple IMUs and visual odometry algorithms.
The main transformations that need to be provided are two:

- **map** to **odom**, provided by a positioning system (localization, mapping, SLAM)

- **odom** to **base_link**, provided by an odometry system

When making use of the rich positioning, odometry and SLAM projects available in the Nav2 community, it is suggested to follow the **REP 105** standard convention. It says that a robot should have at least TF tree containing a full *map -> odom -> base_link -> [sensor frames]*. TF2 is the time-invariant transformation library in ROS 2, which is used to represent and obtain time synchronized transformations. However, since locally filtered poses are not sufficient to account for integrated odometric drift, a global localization solution is needed: it is the job of the global positioning system to provide the map -> odom transformation, while the odometry system provides the odom -> base_link transformation. Nav2 provides two solutions for the global localization:

- **AMCL**, which is an implementation of the Adaptative Monte Carlo Localization that uses a particle filter to localize a robot in a given occupancy grid using omni-directional or differential motion models

- **SLAM** Toolbox, which is a configurable graph-based SLAM system using 2D pose graphs and canonical scan matching to generate a map and serialized files for multi-session mapping.

1.4.3 Environmental representation

The environmental representation is the way the robot perceives the workspace. It is also the result of the combination of various algorithms and data sources in a single space. This space is then used by controllers, planners and recoveries to compute their tasks safely and efficiently. The current environmental representation is a **costmap**: a regular 2D grid of cells containing a cost from unknown, free, occupied or inflated cost. This costmap is then searched, in order to compute a global plan, or sampled, to compute local control efforts.
Various costmap layers are implemented in order to allow a single costmap to be coherently updated by a number of data sources and algorithms. Each layer can extend or modify the costmap it inherits, then forward the new information to planners and controllers.
The most used layers are generally: **Static Layer**, which uses the static map provided by a SLAM pipeline or loaded from disk, and initialize occupancy information, **Inflation Layer**, that inflates lethal obstacles in costmap with exponential decay by convolving the collision footprint of the robot, **Spatio-Temporal Voxel Layer**, that maintains temporal 3D sparse volumetric voxel grid that decays over time via sensor models from the laser and RGBD cameras.
STVL maintains a 3D representation environment to project obstacles into the planning space, even though the costmap is 2D. Also, this particular layer make

use of temporal-based measurement persistence to maintain an accurate view of the world in the presence of dynamic obstacles. Moreover, Navigation2 provides **costmap filters** to mark areas on maps with some additional features or behavioral changes. They are implemented as plugins called "filters", as they are filtering a costmap by spatial annotations marked on filter masks.
In order to make a filtered costmap and change robot's behavior in annotated areas, the filter plugin reads the data coming from the filter mask. This data is linearly transformed into a feature map in a filter space. Having this transformed feature map along with a map/costmap, any sensor data and current robot coordinate filters can update the underlying costamp and change the behavior of the robot depending on where it is.
The functionality that could be used to implement costmap filters are:

- **keep-out/safety zones**, where the robot will never enter
- **speed restrictions areas**, where the maximum speed of robots will be limited
- **preferred lanes** for robots moving in industrial environments and warehouse

These are two variations of the costmap in the Nav2 stack:

- **global costmap**, which is used for global planning, which consists in developing long-term plans for the entire environment. It combines everything that has been perceived and stored by the robot from its previous visits, like the static map, which generally contains immovable elements (e.g. walls).
- **local costmap**, which is used to plan locally and avoid obstacles. It represents everything the robot can learn about the current position by sensors, like visible walls or moving people.

1.4.4 Waypoint Follower

The Nav2 waypoint follower is a basic feature of a navigation system that make the robot follow *waypoints* to reach multiple destinations. It includes a plugin interface for specific task executors: this is very useful for completing specific tasks, such as picking up a box, taking a picture, or waiting for user input.
It can be used as a sample application in an on-robot solution. Nonetheless, it can be used for more complex scenarios: where the robot perform many complex tasks in complete autonomy.
Neither approach is better than other, and the distinction is often very clear for a given business case. According to the tasks the robot is completing, the environment, and the available cloud resource, the choice between these two approaches has to be made.

Chapter 2

Objective of the Thesis

2.1 Alba Robot

Alba Robot srl is a company based in Turin, specifically in the start-up incubator of Politecnico di Torino I3P. It was founded in 2019 by Andrea Bertaia Segato, who is the CEO, in response of an individual need: a brilliant woman named Alba at 90 years old lost her autonomy being forced to use a wheelchair due to knee arthritis. The objective of this company is to change the way People with Reduced Mobility (PRM) move and improve their independence in every day life. In order to accomplish this, the most innovative technologies, such as AI, IoT, Automotive and Robotics, are combined in an end-to-end B2B micro-mobility platform using self-driving electric vehicles with attractive Italian design.

Figure 2.1: SEDIA prototype model

Many trials are pending with other companies like International Airlines Group and an archeological site in Middle East. Moreover, the vehicles have been presented during various events: Gitex in Dubai, Airport PRM Leadership Conference in Paris, SMAU in Milan and Dubai Airshow.
Alba Robot is one of the first mover in the world thanks to its business model and to the type of products and technologies that it is developing.
This paper focuses on the SEDIA (SEat Designed for Intelligent Autonomy) project, which is a cutting-edge mobility platform developed by Alba Robot srl. It is a micromobility platform which transforms people transportation by using autonomous vehicle fleets navigating in big facilities (e.g. airports, museums and hospitals).
The autonomous navigation system integrated into the wheelchair, built on top of ROS 2 and Nav2, is characterized by the hardware architecture and the software architecture.
The autonomous guide's physical parts, such as its sensors, actuators, motors, and other sensory technologies for environment perception, are all part of the hardware layer. These hardware components regulate the motions of the guide and gather information from the surroundings, giving the software layer the inputs it needs to process and decide. However, the software architecture concentrates on the robotic system's higher-level algorithms, decision-making processes, and real-time coordination, allowing the system to navigate autonomously. It covers path planning, object recognition, obstacle avoidance, and other algorithms.
The team at Alba Robt is constanlty working on the development of the SEDIA platform in order to enhance its functionality and performance. The most recent technologies are being used in the platform's development, as it will hopefully become an essential tool for individuals with restricted movement.
The hardware and the software are designed in such a way they can work together seamlessly: the software gives commands to the hardware based on its perception of the environment.This enables the guide to navigate around various obstacles and terrain types while adjusting in real time as necessary.

2.2 Hardware Layer

The aim of the hardware layer (low level) is gather information about the environment that surrounds the vehicle, in order to provide it to the software layer (high level). Various sensors are utilized by the vehicle to do that:

- **depth cameras**: it uses pixels that are associated to the distance from the camera (depth). Some cameras have both an RGB and depth system, which allows the capture of pixels with all four values, named RGBD (Figure 2.2). It exploits some useful technologies such as ToF and stereo vision.
- **LiDAR**: it is a ranging device, which measures the distance to a target. It

sends a short laser pulse and records the time lapse between outgoing light pulse and the detection of the reflected (back-scattered) light pulse [11]. It can create a detailed 3D map of the object or environment by repeating this process multiple times (Figure 2.3).

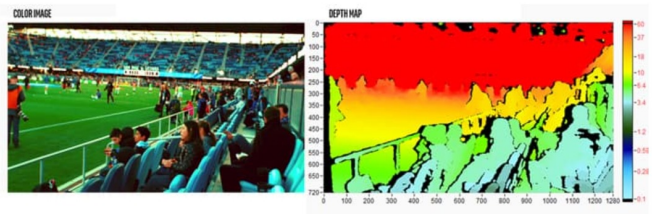


Figure 2.2: depth camera example

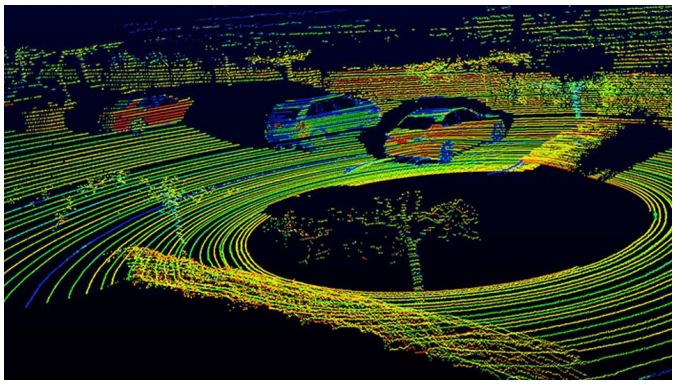


Figure 2.3: LiDAR example

2.3 Software Layer

Since SEDIA must be able to navigate in small to large facilities with multiple floors, it is needed to manage more than only one map. This feature used to be problematic, because:

- maps are implicitly linked to one floor, hence managing multi floor facilities implies the ability to manage multiple maps
- large maps are difficult to maintain since changing a part of it would inevitably affect the remaining part
- large maps are more expensive from a computation point of view and for large maps (e.g. greater than $\sim 8000\,m^2$) navigation is no longer ensured

For these reasons, but not only, it is needed to add a multimap management system to SEDIA.

2.3.1 Navigation plugins

This project uses some plugins that Nav2 provides in order to make users able to create their own custom applications or algorithms with: costmap layer, planner, controller, behavior tree, and behavior plugins [12].
As **Behavior-Tree Navigator** it is has been used the *NavigateToPoseNavigator*: point-to-point navigation via a behavior tree action server.
The **Costmap Layers** used in the project are basically 3: *Static Layer*, which gets static map and loads occupancy information into costmap, *Inflation Layer*, which inflates lethal obstacles in costmap with exponential decay, *Obstacle Layer* that maintains persistent 2D costmap from 2D laser scans with raycasting to clear free space, and *Denoise Layer*, which filters the standalone obstacles or small obstacles group induced by noise.
Sometimes, it is also used a **Costmap Filter**: *Keepout Filter*, that maintains keep-out/safety zones and preferred lanes for moving, and *Speed Filter*, which limits the maximum velocity of robot in speed restriction areas, and *Binary Filter*, that enables binary (boolean) mask behavior to trigger actions.
It is also used a **Smoother**: *Constrained Smoother*, which optimizes multiple criteria (e.g. smoothness, distance from obstacles) using a constraints problem solver.
The used **Behaviors** are: *Clear Costmap*, that is a service that clears the given costmap when wrong perception occurs or robot is stuck, *Spin*, which is a rotate behavior of configurable angles to clear out free space and nudge the robot out of potential local failures, *Back Up*, a back up behavior of configurable distance to back out of a situation where the robot is stuck, and **Wait**, a wait behavior with configurable time to wait in case of time based obstacle (e.g. human traffic).
The used *Waypoint Task Executor* is *WaitAtWaypoint*, which a plugin that executes a wait behavior on waypoint arrivals.
As **Goal Checker**, the *SimpleGoalChecker* is used: a plugin that checks if the robot is within the translational distance and rotational distance of goal. Instead, as **Progress Checker**, it is used the *SimpleProgressChecker*: it checks if the robot was able to make progress towards a goal moving a minimum distance in a given time.
The currently used **Planner** for the project is *SmacPlanner2D*: it implements the 2D A* using either 4 or 8 connected neighborhoods with smoother and multi-resolution query. It supports differential, omnidirectional and legged robots.
The **Controller** that the company has used for years is *TEB Controller*, which is a controller similar to a MPC. It is suitable for Ackermann, differential and

holonomic robots. However, since it is no more maintained, it is currently used the *MPPI Controller*: a predictive MPC controller with modular and custom cost functions that can accomplish many tasks. As well as the previous one, this is suitable for differential, omnidirectional and Ackermann robots.

2.4 The Problem

With the configuration that has been described, it is definitely possible to obtain a very good level of automation of the vehicle, but it is clear that the controllers that are already provided by the Nav2 stack cannot cover all the possible scenarios. For example, the MPPI controller allows the robot to move smoothly in an empty environment, but when it approaches an obstacles, the distance between the two becomes dangerously small.
The implementation of the autonomous navigation is not trivial per se, but one of the biggest problem is the versatility that a controller should have in order to deal with any situation. In order to face this problem, it is necessary to carefully analyze each of the provided controller, so that it is possible to grasp every benefit and drawback. In this paper it is shown the development of an automated benchmark that take into account the robot of the company in 5 different use cases:

- navigation in an empty world
- navigation in a world with sparse static obstacles
- navigation in a world with one static obstacle, characterized by a sharp bend
- navigation in a restricted area
- navigation in a small restricted area

Chapter 3

Method

3.1 Nav2 controller plugins

This project has started studying all the controllers that Nav2 provides:

- DWB
- RPP
- TEB
- MPPI

3.1.1 DWB

The DWB controller is the successor to the base local planner and DWA controller in ROS 1.
The DWA controller implements the **Dynamic Window Approach** [13] to local robot navigation on a plane. This approach searches for the commands that controls the robot directly in the space of velocities. However, in the first step of the algorithm the space is reduced to the velocities that are reachable under the dynamic constraints and safe with respect to the obstacles. In the second step the velocity that maximizes the objective function is chosen from the remaining velocities.
In other words, a single cycle of this algorithm can be described as follows:

- **Search space**: the search space of possible velocities is reduced in 3 steps:
    a. **Circular trajectories**: only circular trajectories (curvatures) uniquely determined by pairs (v, ω) of translational and rotational velocities are taken into account by the dynamic window approach. This yields a velocity search space that is two-dimensional.

b. **Admissible velocities**: Because of the restriction to admissible velocities, only safe trajectories are surely considered. A pair (v, ω) is considered admissible if the robot is able to stop before it reaches the closest obstacle to the corresponding curvature.

c. **Dynamic Window**: it restricts the admissible velocities to those that can be reached within a short time interval given the limited accelerations of the robot.

- **Optimization**: The objective function

$$G(v, \omega) = \sigma(\alpha \cdot heading(v, \omega) + \beta \cdot dist(v, \omega) + \gamma \cdot vel(v, \omega)) \tag{3.1}$$

is maximized. This function trades off the following aspects with respect to current position and orientation of the robot:

a. **Target heading**: *heading* is a measure of progress towards the goal location. It is maximal if the robot moves directly towards the target.

b. **Clearance**: *dist* is the distance to the closest obstacle on the trajectory. The robot's urge to avoid an object increases with its distance from it.

c. **Velocity**: *vel* is the forward velocity of the robot and supports fast movements.

The σ function smoothes the weighted sum of these 3 components.

Search space

Given that each curvature is uniquely determined by the velocity vector (v_i, ω_i), the robot has to determine velocities (v_i, ω_i) (one for each of the n time intervals between t_0 and t_n) in order to generate a trajectory to specified goal point for the next n time intervals. The resulting trajectory must not intersect with any obstacle. The search space for these vectors is exponential in the number of the considered intervals.
In order to make the optimization feasible, the dynamic window approach take into account only the first time interval and assumes that the velocities in the remaining *n-1* time intervals are constant. In other words. the accelerations are assumed equal to 0 in $[t_1, t_n]$. It is possible to make these assumptions without loss of generalities because:

- the reduced search space is two-dimensional and consequently tractable
- the search is repeated after each time interval
- if there are no new commands, velocities will remain constant

Obstacles in the closer environment of the robot imposes restrictions on the rotational and translational velocities. For instance, the maximal admissible speed on curvature depends on the distance to the next obstacle on this curvature.
A velocity is considered admissible if the robot is able to stop before it reaches this obstacle. Hence, the set of admissible velocities V_a is defined as:

$$V_a = \{v, \omega | \sqrt{2 \cdot dist(v, \omega) \cdot \dot{v}_b} \wedge \omega \leq \sqrt{2 \cdot dist(v, \omega) \cdot \dot{\omega}_b}\} \tag{3.2}$$

where *dist(v,ω)* represents the distance to the closest obstacle on the corresponding curvature, for a given velocity *(v,ω)*, $\dot{v}_b$ and $\dot{\omega}_b$ represent the accelerations for breakage.
V_a is the set of velocities that allow the navigation of the robot avoiding collision with an obstacle.
Moreover, the overall search space is reduced to the *dynamic window*, which contains exclusively the velocities that can be reached within the next time interval, so that the limited accelerations that the motors can exert can be taken into account. Considering t as the time interval during which the accelerations $\dot{v}$ and $\dot{\omega}$ will be applied and (v_a, ω_a) as the actual velocity, the dynamic window V_d is defined as:

$$V_d = \{(v, \omega) | v \varepsilon [v_a - \dot{v} \cdot t, v_a + \dot{v} \cdot t] \wedge \omega \varepsilon [\omega_a - \dot{\omega} \cdot t, \omega_a + \dot{\omega} \cdot t]\} \tag{3.3}$$

It is possible to notice that the dynamic window is centered around the actual velocity, while the extensions depend on the accelerations applicable by the motor. All the curvatures that doesn't belong to the dynamic window cannot be reached within the next time interval and thus are not considered for the obstacle avoidance. In the end, the resulting search space is obtained by the intersection of the restricted areas: the space of possible velocities V_s, the space of admissible velocities V_a and the dynamic window V_d

$$V_f = V_s \cap V_a \cap V_d. \tag{3.4}$$

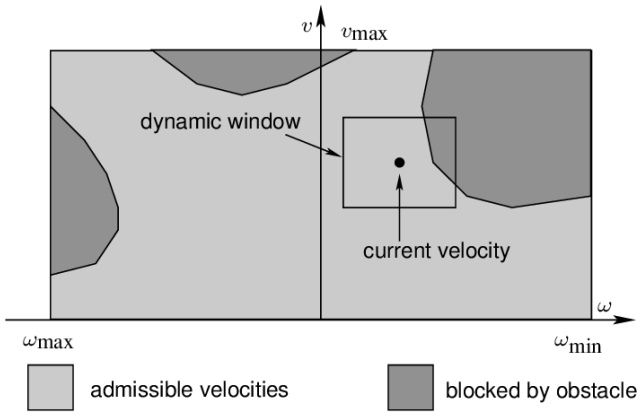


Figure 3.1: Resulting search space

Maximization of the Objective Function

Once it has been determined the restricted search space V_r, it has to be chosen a velocity belonging to it. At this point, the maximum of the objective function (Equation 3.1) is computed over V_r to include the *target heading, clearance* and *velocity* criteria. This is done by discretization of V_r.
The **target heading** measures the alignment of the robot with the target direction. It is obtained by 180-θ, where θ is the angle of the target point relative to the robot's heading direction. It is computed for a predicted position of the robot, because the direction changes with the different velocities.
It is possible to assume that the robot navigates with the selected velocity during the next time interval, however, for a realistic measurement, the dynamics of the rotation has to be taken into account. Thus, θ is computed at the position in which the robot exerts maximal deceleration after the next interval.
The **clearance** (*dist(v,ω)* function) defines the distance to the closest obstacle that intersects with the curvature. In case of no obstacles on the curvature, this parameter is set to a large constant.
The progress of the robot on the corresponding trajectory is evaluated by the function *velocity(v,ω)*.
All these components of the objective function are normalized to [0,1].

Implementation

DWB controller was created in ROS 1 by David Lu at Locus Robotics as part of the `robot_navigation` project. It was then ported in ROS 2 for use in Nav2 as critic-based controller algorithm. It enhances the qualities of DWA by implementing 3 types of plugins: **trajectory generator plugins**, **critic plugins** and plugin implementations for **common use**.
Trajectory Generator Plugins generate the set of possible trajectories that should be evaluated by the critics. The output command velocity depends on the trajectory with the highest score. The trajectory generators provided by Nav2 are:

- `StandardTrajectoryGenerator`
- `LimitedAccelGenerator`

Critic Plugins score the trajectories generated by the trajectory generator. The chosen command velocity is determined by the sum of the scores of the variety of plugins that could be loaded:

- `BaseObstacle` scores a trajectory based on where the path passes over the costmap

- `ObstacleFootprint` scores a trajectory based on verifying all points along the robot's footprint don't touch an obstacle marked in the costmap
- `GoalAlign` scores a trajectory based on how well aligned trajectory is with the goal pose
- `GoalDist` scores a trajectory based on how close the trajectory gets the robot to the goal pose
- `PathAlign` scores a trajectory based on how well it is aligned to the path provided by the global planner
- `PathDist` scores a trajectory based on how far it end up from the path provided by the global planner
- `PreferForward` scores a trajectory that move the robot forwards more highly
- `RotateToGoal`, which only allows the rotation of the robot to the goal orientation when it is close enough to the goal location
- `Oscillation` makes the robot able to avoid just moving backward and forwards
- `Twirling` prevents holonomic robots from spinning as they make their way to the goal

3.1.2 RPP

RPP controller implements a variant on the **Pure Pursuit Algorithm** to track a path.

PP

Considering a path P as an ordered list of points $P=\{p_0,p_1,...,p_n\}$ where $p_i=(x_i,y_i) \in P$, a local trajectory planner is a function f that defines the linear and angular velocity to track a reference path P_t at time t.

$$(v_t, w_t) = f(P_t) \tag{3.5}$$

It is introduced graphically in Figure 3.2 [14].

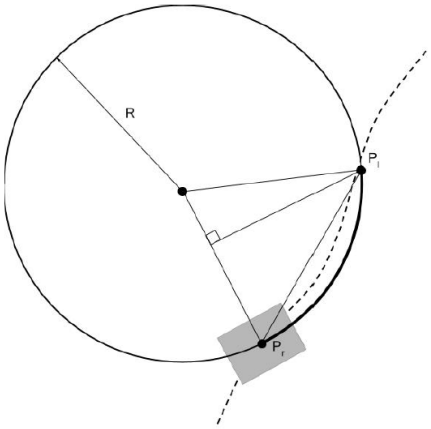


Figure 3.2: Pure Pursuit Goal Point

At first, it determines the closest point p_r on P_t to the robot position. The lookahead point p_l is then determined as the first p_i at least L (lookahead distance) away from p_r. In formula:

$$dist(p_i) = \sqrt{(x_r - x_i)^2 + (y_r - y_i)^2} p_l = p_i \in P_t, \begin{cases} dist(p_{i-1}) < L \\ dist(p_i) \geq L \end{cases} \tag{3.6}$$

Knowing p_l, it is possible to determine the curvature of the circle using simple geometry. Thus, representing P_t in vehicle base coordinates, P'_t, the curvature can be represented as

$$K = \frac{2y'_l}{L^2} \tag{3.7}$$

where K is the path curvature required to drive the robot from its starting position to the lookahead carrot, y'_l is the lateral coordinate of the lookahead point p'_l, and L is the desired distance between p_r and p_l. At this point, the commands can be sent to a robot controller. This process is then updated at the desired rate.
The velocity of travel v_t and the distance L along the path used to select the lookahead point p_l are the crucial parameters of this path tracker. The lookahead point p_l is at distance L from the robot, tuned to achieve an acceptable trade-off between oscillations centered around the path (shorter distances) and slower convergence (longer distances).
The response of the pure pursuit tracker looks similar to the step response of a second order dynamic system, and the value of L tends to act as a damping factor.

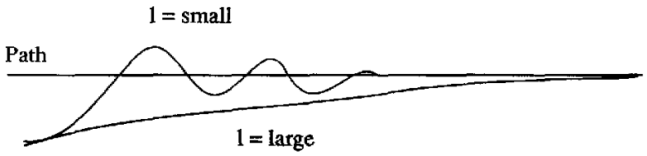


Figure 3.3: Pure Pursuit response

Considering a circle of arbitrary curvature in Figure 3.4, which is a path of constant curvature, it should obviously should have one lookahead distance that can be used for path following.

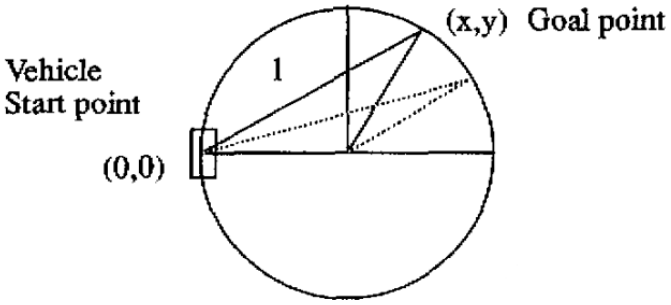


Figure 3.4: Generic circle

It is possible to notice that an isosceles triangle within the circle has the lookahead distance as the base extending from the starting point to the goal point, while the sides are simply the radius of the circle. The lookahead distance that can form this isosceles triangle will satisfy the conditions on the curvature and, therefore, define the curvature of this circle. Clearly, many lookahead distances ranging between 0 and 2r can satisfy them [15].
Nontheless, there are several drawbacks of this approach: the Pure Pursuit is known to have overshoot or undershoot behaviors (which results in deviations) in high curvature situations. This is a major issue only for smaller-scale applications (e.g. industrial and consumer robots), but it does not particularly affect autonomous driving applications because it typically has a minimum turning radius limit. Also, it does not specify any translational specification criteria during execution. This feature on one hand could be unsafe, because without described methods, almost all known variants use a constant traslation velocity profile. On the hand, it can be beneficial as it allows having a great deal of flexibility with different linear velocity profiles.

PP Variants

In order to increase path tracking stability, they have been proposed variants of this algorithm by varying the computations of the lookahead point. MIT's entry into the DARPA Urban Challenge implemented the **Adaptative Pure Pursuit (APP)** algorithm for lane following while varying the lookahead distances proportionally to the translational velocity [16]. A mapping of lookahead distances to velocities is required so that it is possible to obtain an acceptable trade-off between oscillation and slower convergence to the path. In formula:

$$L_t = v_t l_t \tag{3.8}$$

where L_t is the lookahead distance, v_t is the translational velocity and l_t is a lookahead gain representing the amount of time to project v_t forward.
Recently, a new variant has been created: it modifies the process of determining the lookahead point, adjusting it off the path and containing a heuristic change to the translational velocity to address the case of short-cutting during path tracking in substantial turns. However, this policy does not solve the problem facing deployment of PP techniques to general mobile robotics applications, but only to Ackermann vehicles.
Moreover, Pure Pursuit nor its variations account for dynamic effects of the vehicle, because PP is geometrically derived, thus vehicle dynamics are not modeled in this style of path tracking algorithm.

RPP

The **Regulated Pure Pursuit** algorithm was designed for service and industrial mobile robots in real world by Shrijit Singh and Steve Macenski while at Samsung Research as part of the Nav2 working group.
It provides methods for adapting robot's translational velocity to current conditions: regulation cost functions. They provide excellent performance in a range of real-world mobile robot scenarios, derived from standard demands on lowering robot velocities when maneuvering around sharp bends or in small spaces. The regulated Pure Pursuit algorithm firstly determines p_r and all prior points $\{p_0,...,p_j\}$ are permanently pruned from the stored path to prevent unnecessary future transformations of obsolete data. Then, it transforms the input path P_t $\{p_0,...,p_j,|p_r,...,p_n\}$ into the robot's base coordinates frame P'_t and prune it. On this way, the curvature of the path can be obtained by 3.7. For every point p_i where $dist(p'_r,p'_i) \gg L_t$, the modified path $(\{p'_r,...,p'_n\})$ is similarly pruned, because the points are far enough away to never need to be taken into consideration at t. These far path points continue to be held in the stored path for future iterations, until a

new path is received.
At this point, RPP will utilize the same lookahead selection mechanics as Adaptative Pure Pursuit formulated in Equation 3.8. This makes the lookahead distance L_t proportional to the speed v_t and a lookahead gain l_t, so that longer distances are used while moving faster. The lookahead point p_l is chosen using this distance. Interpolating between path points noticeably improve smoothness on sparse paths (with 0.1 m - 1.0 m resolution) at autonomous vehicle speeds, however this does not contribute much benefit at the service and industrial robot speeds using typical grid map planning resolutions (0.025 m - 0.1 m).
Moreover, the desired linear velocity v_t is still processed by the curvature and proximity heuristics. Both heuristics are applied to linear velocity and then the maximum of the two are taken.
The **Curvature Heuristic** is needed in order to slow the robot to v_t' during sharp turns into partially observable environments (e.g. when entering or exiting hallways and aisles commonly found in retail, warehouses, factories, schools and shopping malls). This feature makes the navigation significantly safer when making blind turns. This heuristic is applied to the linear velocity v_t when the change in curvature θ is above a minimum threshold T_θ. This minimal radius prevents velocity scaling in small turns or path changes that do no require slowing. The curvature velocity v_t' returned by this heuristic is selected as:

$$v_t' = \begin{cases} v_t & K > T_k, \\ \frac{v_t}{r_min \ K} & K \leq T_k \end{cases} \tag{3.9}$$

where r_{min} is the minimum turning radius to apply the heuristic. This formulation reduces mathematically a simple error calculation between the minimum radius and the radius of a circle represented by K.
The **Proximity Heuristic** is applied to the linear velocity v_t when the robot becomes in close proximity to dynamic obstacles or fixed infrastructure. This heuristic slows the robot when in constrained environments where it has a high probability to collide. The likelihood of collision is minimized reducing the speed near fixed infrastructures, so that the impact of small path variations in tight spaces is reduced. Lowering the speed of industrial and service robots in close proximity to dynamic agents, like humans, is a common safety requirement - allowing a robot to reactively stop faster to prevent potential injury. To adapt the response to different systems, the heuristic's linear formulation lowers the speed by the ratio d_O/d_{prox} with a gain α. This linear heuristic, showed in Equation 3.10, has a broad range of α which may be finely tuned by a system designer and derives from the well-used Adaptative Pure Pursuit's formulation.

$$v_t' = \begin{cases} v_t \frac{alpha \, d_O}{d_{prox}} & d_O \leq d_{prox} \\ v_t & d_O > d_{prox} \end{cases} \tag{3.10}$$

In Equation 3.10 d_{prox} is the proximity distance to obstacles to apply the heuristic, d_O is the current distance to an obstacle, and α is a gain to scale the heuristic function for aggressive behavior, with the requirement that $\alpha \leq 1.0$. Higher values of α reduce the velocity of the robot in proximity to obstacles more expeditiously. The value of d_{prox} should be chosen according to the system requirements of a robot's application for how close an obstacle can be before the robot begins slowing its maximum velocity.
After velocity regulation, it can be determined the path curvature using Equation 3.5. The angular velocity is computed using the regulated velocity, not desired linear velocity: this prevents consequential undershoot behavior relative to target curvatures. Thus, the angular velocity can be calculated as:

$$\omega_t = v'_t K \tag{3.11}$$

The final step of the algorithm is to check the path tracking command for current or imminent collisions. A given angular velocity ω_t and regulated linear velocity v'_t can be projected forward in time, resulting in a circular arc. Points on the arc are sampled at the grid map cell resolution forward for a set duration. Collision checking is done based on a duration to collision (instead of to the lookahead point), so that the robot is always, at minimum, a set duration from collision. At slow speeds, it may not be sensible to collision check to a lookahead point tens of meters, or hundreds of seconds, away. Instead, in restricted areas where the current velocity commands might not be admissible a short distance—but long time—away, a temporal schema enables precise manipulations.

Implementation

The implementation of this algorithm is in C++ and combines Pure Pursuit, Adaptative Pure Pursuit and Regulated Pure Pursuit. Also, it is parameterized, in order to enable each specific behavior. On this way, it is possible to quickly evaluate and tune these features by simply changing a handful of parameters from the available binaries.
The robot can also be configured to reduce its speed as it gets closer to the desired goal pose. Because of this feature, the robot stops more gradually: the translational velocity is lowered proportional to the remaining distance, up to a minimum viable velocity to make progress, when the robot is close enough to the target.
This controller is suitable for use on all types of robots, such as differential, legged, and Ackermann steering vehicles. When applied to omni-directional platforms, it is not able to fully leverage the lateral movements of the base. The configuration of this controller is done by tuning the following parameters:

- `desired_linear_vel` - the desired maximum linear velocity to use

- `lookahead_dist` - the lookahead distance to use to find the lookahead point
- `min_lookahead _dist` - the minimum lookahead distance threshold when using velocity scaled lookahead distances
- `max_lookahead_dist` - the maximum lookahead distance threshold when using velocity scaled lookahead distances
- `lookahead_time` - the time to project the velocity by to find the velocity scaled lookahead distance (lookahead gain)
- `rotate_to_heading_angular_vel` - if rotate to heading is used, this is the angular velocity to use
- `use_velocity_scaled_lookahead_dist` - whether to use the velocity scaled lookahead distances or constant `lookahead_distance`
- `min_approach_linear_velocity` - the minimum velocity threshold to apply when approaching the goal
- `use_collision_detection` - whether to enable collision detection
- `max_allowed_time_to_collision_up_to_carrot` - the time to project a velocity command to check for collisions when `use_collision_detection` is `true`. It is limited to maximum distance of lookahead distance selected.
- `use_regulated_linear_velocity_scaling` - whether to use regulated features for curvature
- `use_cost_regulated_linear_velocity_scaling` - whether to use the regulated features for proximity to obstacles
- `cost_scaling_dist` - the minimum distance from an obstacle to trigger the scaling of linear velocity, if `use_cost_regulated_linear_velocity_scaling` is `true`. The value set should be smaller or equal to the `inflation_radius` set in the inflation layer of costmap, since inflation is used to compute the distance from obstacles
- `cost_scaling_gain` - A multiplier gain, which should be less or equal to 1.0, used to further scale the speed when an obstacle is within `cost_scaling_dist`. Lower values reduces speed more quickly.
- `inflation_cost_scaling_factor` - it should be exactly the same of the value of the `cost_scaling_factor` set for the inflation layer in the local costmap

- `regulated_linear_scaling_min_radius` - the turning radius for which the regulation features are triggered
- `regulated_linear_scaling_min_speed` - the minimum speed for which the regulated features can be send, to ensure process is still achievable even in high cost spaces with high curvature.
- `use_fixed_curvature_lookahead` - enables fixed lookahead for curvature detection
- `curvature_lookahead_dist` - distance to lookahead to determine curvature for velocity regulation purposes. It is only used if `use_fixed_curvature_lookahead` is enabled
- `use_rotate_to_heading_min_angle` - the difference in the path orientatation and the starting robot orientation to trigger a rotation in place, if `use_rotate_to_heading` is enabled
- `max_angular_accel` - maximum allowable angular acceleration while rotating to heading, if enabled
- `max_robot_pose_search_dist` - maximum integrated distance along the path to bound the search for the closest pose to the robot.This is set by default to the maximum costmap extent, so it shouldn't be set manually unless there are loops within the local costmap

The used topics are `lookahead_point` and `lookahead_arc`.In the first, messages of `geometry_msgs/PointStamped` type, defining the current lookahead point on the path, are published. In the second, the `nav_msgs/Path` messages are published: they define the drivable arc between the robot and the carrot. Arc length depends on `max_allowed_time_to_collision_up_to_carrot`, forward simulating from the robot pose at the commanded `Twist` by that time. In a collision state, the last published arc will be the points leading up to, and including, the first point in collision.

3.1.3 TEB

The TEB local planner is based on the method called Time Elastic Band, which locally optimizes the robot's trajectory with respect to trajectory execution time, separation from obstacles and compliance with kinodynamic constraints at runtime [17].
The classic "Elastic Band" is described in terms of a sequence of n intermediate robot poses $x_i = (x_i, y_i, \beta_i)^T \in \mathbb{R}^2 \times S^1$, in the following denoted as a configuration

including position x_i, y_i and orientation β_i of the robot in the related frame ({map} Figure 3.5):

$$Q = \{x_i\}_{i=0...n} \quad n \in \mathbb{N} \tag{3.12}$$

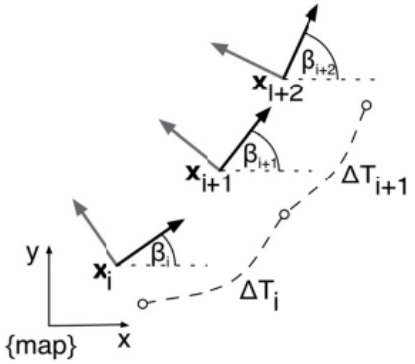


Figure 3.5: Sequences of configurations

The **"Timed Elastic Band" (TEB)** is augmented by the time intervals between two consecutive configurations, resulting in a sequence of *n-1* time differences ΔT_i:

$$\tau = \{\Delta T_i\}_{i=0...n-1} \tag{3.13}$$

Each time differences defines the time that the robot takes to transit from one configuration to the next one in sequence. The TEB method is a tuple of both sequences:

$$B := (Q, \tau) \tag{3.14}$$

The main goal is to adapt and optimize the TEB in terms of configurations as well as time intervals, by a weighted multi-objective optimization in real-time:

$$f(B) = \sum_k \gamma_k f_k \tag{3.15}$$

$$B^* = \arg\min_B f(B) \tag{3.16}$$

B^* denotes the optimized TEB, f(B) denotes the objective function.
Many components of the objective function are local with respect to B, because they only depend on a few number of consecutive configurations. This property yields in a sparse system matrix, for which specialized fast and efficient large scale numerical optimization methods are available (like Least-Squares).
The objective functions of the TEB belong to two types: constraints, such as velocity and acceleration limits formulated in terms of penalty functions, and objectives with respect to trajectory, such as shortest or fastest path or clearance

from obstacles.
Sparse constrained optimization algorithms are not readily available in robotic frameworks (e.g. ROS) in a freely usable implementation. Hence, for what concerns the "Time Elastic Band" method, these constraints are formulated as obejctives in terms of a piecewise continuous, differentiable cost function that penalizes the violation of a constraint:

$$e_\Gamma(x, x_t, \epsilon, S, n) \simeq \begin{cases} (\frac{x-(x_r-\epsilon)}{S})^n & if x > x_r - \epsilon \\ 0 & otherwise \end{cases} \tag{3.17}$$

where x_r denotes the bound. S, n and ϵ affect the the accuracy of the approximation: S defines the scaling, n the polynomial order and ϵ a small translation of the approximation. For example in Figure 3.6 there are two different realizations of Equation 3.17: Approximation 1 is obtained by setting $n=2$,$S=0.1$,$\epsilon=0.1$, while Approximation 2 (which is noticeably the stronger approximation between the two) by setting $n=2$, $S=0.05$ and $\epsilon=0.1$. Both of them approximates the constraint $x_r=0.4$.

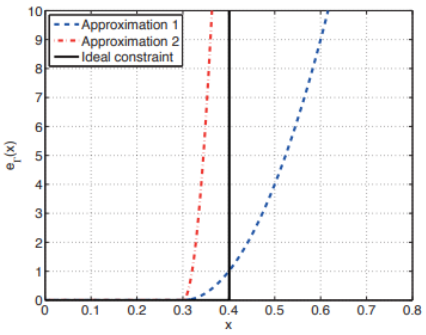


Figure 3.6: Polynomial approximation of constraints

The TEB takes into consideration avoiding static or dynamic obstacles as well as reaching the intermediate way points of the original path at the same time. The only difference between these two objective functions is that obstacles repel the elastic band, while way points attract it. The objective function depends on the minimal separation $d_{min,j}$ between the TEB and the way point or obstacle z_j.

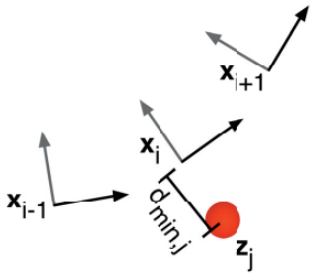


Figure 3.7: Minimal distance between TEB and way point or obstacle

In the case of way points, the distance is bounded from above by a maximal target radius $r_{p_{max}}$ (Equation 3.18). Instead, in the case of obstacles it is bounded from below by a minimal distance $r_{o_{min}}$ (Equation 3.19.

$$f_{path} = e_\Gamma(d_{min,j}, r_{p_{max}}, \epsilon, S, n) \tag{3.18}$$

$$f_{ob} = e_\Gamma(-d_{min,j}, -r_{o_{min}}, \epsilon, S, n) \tag{3.19}$$

The signs of the separation $d_{min,j}$ and the bound $r_{o_{min}}$ must be swapped in order to realize a bounding from below.

Similar penalty function, as in the case of geometric constraints, is exploited to define dynamic constraints on robot velocity and acceleration. The euclidean or angular distance between two successive configurations x_i, x_{i+1} and the time interval ΔT_i for the transition between both poses are used in order to compute the mean translational and rotational velocities.

$$v_i \simeq \frac{1}{\Delta T_i} \left\| \begin{pmatrix} x_{i+1} - x_i \\ y_{i+1} - y_i \end{pmatrix} \right\| \tag{3.20}$$

$$\omega_i \simeq \frac{\beta_{i+1} - \beta_i}{\Delta T_i} \tag{3.21}$$

The euclidean distance approximates the true length of the circular path between two consecutive poses accurately enough, because configurations are very close to each other. Since the acceleration connects two successive mean velocities, three consecutive configurations with two corresponding time intervals are taken into consideration:

$$a_i = \frac{2(v_{i+1} - v_i)}{\Delta T_i + \Delta T_{i+1}} \tag{3.22}$$

where the three consecutive configurations are substituted by their two related velocities.

The rotational acceleration is computed in the same way as Equation 3.22, taking into account angular velocities instead of the linear ones.

The classic elastic band method objective is obtaining the shortest path. Nonetheless, since this new approach takes into account temporal information, it is possible to replace or combine the shortest path objective with the fastest path one. The objective of the **fastest path** is easily achieved by:

$$f_k = (\sum_{i=1}^{n} \Delta T_i)^2 \tag{3.23}$$

This objective leads to a fastest path in which the intermediate configurations are uniformly separated in time rather than space.

Implementation

The control flow of the implemented TEB is shown in Figure 3.8.

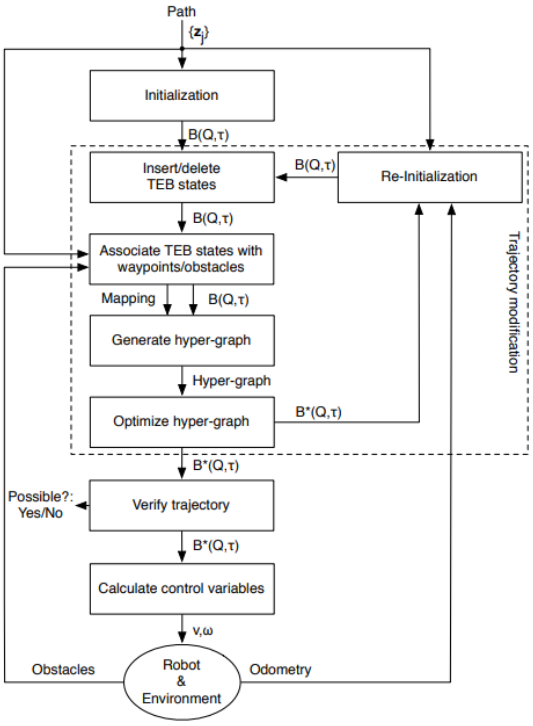


Figure 3.8: Control flow of TEB implementation

In the initialization phase default timing information, adhering the dynamic and kinematic constraints, are added to the initial path. On this way it becomes

an *initial trajectory*.
At each iteration, new configurations are added or previous ones are deleted dynamically by the algorithm, in order to adjust the spatial and temporal resolution to the remaining trajectory length or planning horizon. A hysteresis is implemented to avoid oscillations.
The optimization problem is then transformed into a hyper-graph and solved with large scale optimization algorithms for sparse systems. The required hyper-graph is characterized by an unlimited amount of connected nodes of one single edge. This means that an edge can connect more than two nodes.
The TEB problem (Equation 3.15) can be transformed into a hyper-graph that has configurations and time differences as nodes. The edges connecting them represent the given objective functions f_k or constraint functions (Figure 3.9).

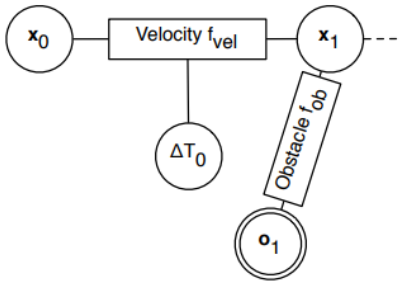


Figure 3.9: Velocity and obstacle objective function formulated as a hyper graph

Figure 3.9 shows an example of hyper-graph with two configurations: one time difference and a point-shaped obstacle. The mean velocity, which is related to the euclidean distance between two configurations and the necessary navigation time, is required by the velocity bounding objective function. As a result, it creates an edge that links those B states. The obstacle requires one edge connected to the nearest configuration. Since the obstacle node is fixed (double circle), its parameters (position) are not affected by optimization algorithms.
Control variables v and ω can be computed to directly command the robot navigation system, only after verifying the optimized TEB.
The re-initialization step occurs before every new iteration, in order to check new and changing way-points. It is particularly useful when way-points are received after analyzing short-range camera or laser-scan data.

3.1.4 MPPI

MPPI is predictive controller (local trajectory planner) that implements the **Model Predictive Path Integral** algorithm to track a path with adaptive collision

avoidance.
The basis of this algorithm derives from a stochastic trajectory method, which is characterized by the path integral control framework [18].

Stochastic Trajectory Optimization

Using random sampling of trajectories, path integral control offers a mathematically solid methodology for developing optimal control algorithms. One of the most useful features of the developed algorithms in the path integral framework is that they require no derivatives of dynamics or cost. This makes the estimation of the system dynamics and the design of cost function more flexible and robust.
The typical derivation of path integral control is based on an exponential transformation of the value function of the optimal control problem. It also requires that only the actuated states are affected by the noise in the system. In such a way, it is possible to transform the stochastic Hamilton-Jacobi-Bellman equation into a linear partial differential equation, that can be eventually transformed into a path integral, using the Feynman-Kaclemma. The resulting path integral takes the form of an expectation over trajectories under the uncontrolled dynamics of the system, which is then approximated using Monte-Carlo sampling to compute the control for the current state.
However, it is possible to provide a new formulation which offers two benefits over the standard path integral approach:

- it permits noise in both directly and indirectly actuated situations by relaxing the condition between the noise and control authority
- rather than only providing a formula for the initial time, it explicitly gives the optimal settings throughout the full time horizon

Considering stochastic dynamical systems with the state and controls at time t denoted $x_t \in \mathbb{R}^n$ and $u_t \in \mathbb{R}^m$, these dynamics are disturbed by the Brownian motion $dw \in \mathbb{R}^p$. Let $\mathrm{u}(\cdot) : [t_0, T] \to \mathbb{R}^m$ be the function mapping time to control inputs, and the $\tau : [t_0, T] \to \mathbb{R}^n$ be the trajectory of the system.
In the classical stochastic optimal control setting, the objective is finding a control sequence $\mathrm{u}(\cdot)$ such that:

$$u^*(\cdot) = \arg\min_{u(\cdot)} \mathbb{E}_{\mathbb{Q}} \left[\phi(x_T, T) + \int_{t_0}^{T} \mathcal{L}(x_t, u_t, t) dt \right] \tag{3.24}$$

where the expectation is taken with respect to the dynamics: $dx = F(x_t, u_t, t)dt + B(x_t, t)\mathrm{dw}$. The costs are considered to be composed of an arbitrary state-dependent term and quadratic control cost:

$$\mathcal{L}(x_t, u_t, t) = q(x_t, t) + \frac{1}{2} u_t^T R(x_t, t) u_t \tag{3.25}$$

and dynamics which are affine in controls:

$$F(x_t, u_t, t) = f(x_t, t) + G(x_t, t)u_t \tag{3.26}$$

The controlled dynamics induce another **probability measure on the space trajectories**, denoted as $\mathbb{Q}(\mathrm{u})$. Moreover, it needs to be computed a **relative entropy term** between the uncontrolled distribution $\mathbb{P}$ and the controlled distribution $\mathbb{Q}(\mathrm{u})$. It can be done by applying Girsanov's theorem [19] [1]:

$$\mathbb{D}_{KL}(\mathbb{Q}(u)||\mathbb{P}) = \frac{1}{2}\int_{t_0}^{T} u_t^T G(x_t, t)^T \Sigma(x_t, t)^{-1} G(x_t, t) u_t dt \tag{3.27}$$

where $\Sigma(x_t, t) = B(x_t, t)B(x_t, t)^T$.
Assuming that the control cost matrix takes the form

$$R(x_t, t) = \lambda G(x_t, t)^T \Sigma(x_t, t)^{-1} G(x_t, t) \tag{3.28}$$

it is possible to set the following equation:

$$\mathbb{E}_{\mathbb{Q}(u)}[S(\tau)] + \lambda \mathbb{D}_{KL}(\mathbb{Q}||\mathbb{P}) = \mathbb{E}_{\mathbb{Q}(u)}\left[S(\tau) + \frac{1}{2}\int_{t_0}^{T} u_t^T R(x_t, t) u_t dt\right] \tag{3.29}$$

where $\lambda \in \mathbb{R}^+$, $S(\tau)$ is the state dependent cost-to-go term $\phi(x_T, T) + \int_{t_0}^{T} q(x_t, t)dt$. Moreover, it is possible to derive the form of optimal probability measure $\mathbb{Q}^*$ in terms of the Radon-Nykodym derivative [2] with respect to the uncontrolled dynamics:

$$\frac{d\mathbb{Q}^*}{d\mathbb{P}} = \frac{exp(-\frac{1}{\lambda}S(\tau)}{\mathbb{E}_{\mathbb{P}}\left[exp(-\frac{1}{\lambda}S(\tau))\right]} \tag{3.31}$$

---

[1] Let $\gamma = \{\gamma_t \in [0, T]\}$ be a $\{\mathcal{F}_{\sqcup}\}$ - predictable process such that $\mathbb{E}^{\mathbb{P}}\left[exp\left(\frac{1}{2}\int_0^T \gamma_t^2 dt\right)\right] < \infty$. There exist a measure $\mathbb{Q}$ on $(\Omega, \mathcal{F})$ such that:

G1. $\mathbb{Q}$ is equivalent to $\mathbb{P}$

G2. $\frac{d\mathbb{Q}}{d\mathbb{P}} = exp\left[-\int_0^T \gamma_t dW_t - \frac{1}{2}\int_0^T \gamma_t^2 dt\right]$

G3. The process $\tilde{W} = \left\{\tilde{W}_t : t \in [0, T]\right\} defined as \tilde{W}_t = W_t + \int_0^t \gamma_s ds$ is a $(\{\mathcal{F}_t\}, \mathbb{Q})$ - Brownian motion.

[2] The Radon-Nikodym derivative defines the measure $\mathbb{Q}^*$ by means of the equation

$$\mathbb{Q}^*(A) = \int_A \frac{d\mathbb{Q}^*}{d\mathbb{P}} d\mathbb{P} \tag{3.30}$$

Connecting the information theoretic notions of free energy, relative entropy and classical optimal control theory, yields to a method for computing a control law independet of the HJB-equation. In such a way, the minimization problem can be solved by moving the probability distribution induced by the controller $\mathbb{Q}(\mathrm{u})$ as close as possible to the optimal probability measure $\mathbb{Q}^*$, defined by the Radon-Nikodym derivative $\frac{d\mathbb{Q}^*}{d\mathbb{P}}$. Using the relative entropy between $\mathbb{Q}^*$ and $\mathbb{Q}(\mathrm{u})$ as a notion of distance:

$$u^*(\cdot) = \arg\min_{u(\cdot)} \mathbb{D}_{\mathbb{KL}}(\mathbb{Q}^*||\mathbb{Q}(u)) \tag{3.32}$$

This formulation is very useful, because the optimal probability measure $\mathbb{Q}^*$ takes the form of a softmax [20] [3] function with temperature λ. If a trajectory has a low cost, then $\frac{d\mathbb{Q}^*}{d\mathbb{P}}$ has a high value. Hence, if a trajectory is drawn at random from $\mathbb{Q}^*$, it is likely to have a very low cost. If $\mathbb{Q}(\mathrm{u})$ is able to accurately approximate $\mathbb{Q}^*$, then appyling the controls $\mathrm{u}(\cdot)$ to the system will have a high probability of generating a low cost trajectory.
Now, it is possible to minimize the relative entropy between optimal distribution $\mathbb{Q}^*$ and the distribution induced by the controller $\mathbb{Q}(\mathrm{u})$. Applying the definition of relative entropy:

$$\mathbb{D}_{\mathbb{KL}}(\mathbb{Q}^*||\mathbb{Q}(u)) = \mathbb{E}_{\mathbb{Q}^*}\left[\log\left(\frac{d\mathbb{Q}^*}{d\mathbb{Q}(u)}\right)\right] \tag{3.34}$$

It is necessary to find an expression for the Radon-Nikodym derivative $\frac{d\mathbb{Q}^*}{d\mathbb{Q}(u)}$, to optimize this function:

$$\frac{\mathbb{Q}^*}{d\mathbb{Q}(u)} = \frac{d\mathbb{Q}^*}{d\mathbb{P}}\,\frac{d\mathbb{P}}{d\mathbb{Q}(u)} \tag{3.35}$$

At this point it is possible to apply Girsanov's theorem to compute $\frac{d\mathbb{P}}{d\mathbb{Q}(u)}$:

$$\frac{d\mathbb{P}}{d\mathbb{Q}(u)} = \exp(\mathcal{D}(\tau, u(\cdot))) \tag{3.36}$$

[3] The softmax function takes as input a vector z of K real numbers, and normalizes it into a probability distribution consisting of K probabilities proportional to the exponentials of the input numbers. For a vector z of K real numbers, the standard (unit) softmax function $\sigma : \mathbb{R}^K \mapsto (0,1)^K$, where $K \geq 1$, is defined by the formula:

$$\sigma(z)_i = \frac{e^{z_i}}{\sum_{j=1}^{K} e^{z_j}} \qquad for\, i = 1, ..., K\, and\, z = (z_1, ..., z_K) \in \mathbb{R}^K \tag{3.33}$$

In general, it can be used another base $b > 0$, instead of e. For example, $b = e^{\beta}$ or $b = e^{-\beta}$ where the reciprocal of β is sometimes referred to as the temperature.

with

$$\begin{aligned}\mathcal{D}(\tau, u(\cdot)) = &- \int_0^T u_t^T G(x_t,t)^T \Sigma(x_t,t)^{-1} B(x_t,t) dw^{(0)} \\ &+ \frac{1}{2}\int_0^T u_t^T G(x_t,t)^T \Sigma(x_t,t)^{-1} G(x_t,t) u_t dt\end{aligned} \tag{3.37}$$

where $dw^{(0)}$ is a Brownian motion with respect to $\mathbb{P}$ (i.e. $\mathbb{E}_{\mathbb{P}}\left[\int_0^t dw^{(0)}\right] = 0, \ \forall t$). Applying Equation 3.31:

$$\mathbb{D}_{KL}(\mathbb{Q}^*||\mathbb{Q}(u)) = \mathbb{E}_{\mathbb{Q}^*}\left[\log\left(\frac{\exp(-\frac{1}{\lambda}S(\tau))\exp(\mathcal{D}(\tau,u(\cdot)))}{\mathbb{E}_{\mathbb{P}}[\exp(-\frac{1}{\lambda}S(\tau))]}\right)\right] \tag{3.38}$$

It could be re-written as

$$\mathbb{D}_{KL}(\mathbb{Q}^*||\mathbb{Q}(u)) = \mathbb{E}_{\mathbb{Q}^*}\left[-\frac{1}{\lambda}S(\tau) + \mathcal{D}(\tau,u(\cdot)) - \log\left(\mathbb{E}_{\mathbb{P}}\left[\exp\left(-\frac{1}{\lambda}S(\tau)\right)\right]\right)\right] \tag{3.39}$$

Since $S(\tau)$ does not depend on the controls u$(\cdot)$, it is possible to remove the first and last terms from the minimization:

$$\arg\min_{u(\cdot)} \mathbb{D}_{KL}(\mathbb{Q}^*||\mathbb{Q}(u)) = \arg\min_{u(\cdot)} \mathbb{E}_{\mathbb{Q}^*}[\mathcal{D}(\tau,u(\cdot))] \tag{3.40}$$

The goal is to find the function u$^*(\cdot)$ that minimizes Equation 3.40. However, considering only the class of step functions is enough because the control is obviously applied in discrete times:

$$u_t = \begin{cases} \vdots \\ u_j \quad if\ j\Delta t \leq t < (j+1)\Delta t \\ \vdots \end{cases} \tag{3.41}$$

where $j = \{0,1,...N\}$. Applying this formulation to $\mathcal{D}(\tau, u(\cdot))$:

$$\mathcal{D}(\tau,u(\cdot)) = -\sum_{j=0}^{N} u_j^T \int_{t_j}^{t_{j+1}} \mathcal{G}(x_t,t) dw^{(0)} + \frac{1}{2} u_j^T \int_{t_j}^{t_{j+1}} \mathcal{H}(x_t,t) dt u_j \tag{3.42}$$

where

i. $\mathcal{G}(x,t) = G(x,t)^T \Sigma(x,t)^{-1} B(x,t)$

ii. $\mathcal{H}(x,t) = G(x,t)^T \Sigma(x,t)^{-1} G(x,t)$

iii. $N = T/\Delta t$

Since u_j does not depend on the trajectory, when the expectation operator is applied, it is obtained $\mathbb{E}_{\mathbb{Q}^*}[\mathcal{D}(\tau, u(\cdot))]$ equal to:

$$\begin{aligned}\mathbb{E}_{\mathbb{Q}^*}[\mathcal{D}(\tau, u(\cdot))] = &-\sum_{j=0}^{N} u_j^T \mathbb{E}_{\mathbb{Q}^*}\left[\int_{t_j}^{t_{j+1}} \mathcal{G}(x_t, t) dw^{(0)}\right] \\ &+ \sum_{j=0}^{N} \frac{1}{2} u_j^T \mathbb{E}_{\mathbb{Q}^*}\left[\int_{t_j}^{t_{j+1}} \mathcal{H}(x_t, t) dt\right] u_j\end{aligned} \tag{3.43}$$

It is obvious that this formulation is convex with respect to each u_j. This means that, it is sufficient to take the gradient of Equation 3.43 with respect to u_j, set it equal to zero and solve it for u_j, in order to find u_j^*. The result is:

$$u_j^* = \mathbb{E}_{\mathbb{Q}^*}\left[\int_{t_j}^{t_{j+1}} \mathcal{H}(x_t, t) dw^{(0)}\right] + \sum_{j=0}^{N} \frac{1}{2} u_j^T \mathbb{E}_{\mathbb{Q}^*}\left[\int_{t_j}^{t_{j+1}} \mathcal{G}(x_t, t) dw^{(0)}\right] \tag{3.44}$$

For small Δt it is possible to approximate:

$$\int_{t_j}^{t_{j+1}} \mathcal{H}(x_t, t) dt \approx \mathcal{H}(x_{t_j}, t_j) \Delta t \tag{3.45}$$

$$\int_{t_j}^{t_{j+1}} \mathcal{G}(x_t, t) dw^{(0)} \approx \mathcal{G}(x_{t_j}^{t_{j+1}} dw^{(0)} \tag{3.46}$$

This yields to:

$$u_j^* = \frac{1}{\Delta t} \mathbb{E}_{\mathbb{Q}^*}[\mathcal{H}(x_{t_j}, t_j)]^{-1} \mathbb{E}_{\mathbb{Q}^*}\left[\mathcal{G}(x_{t_j}, t_j) \int_{t_j}^{t_{j+1}} dw^{(0)}\right] \tag{3.47}$$

The sampling cannot be done from the $\mathbb{Q}^*$ distribution, thus it is needed to change the expectation in order to have it with respect to the uncontrolled dynamics $\mathbb{P}$.

$$u_j^* = \frac{1}{\Delta t} \mathbb{E}_{\mathbb{P}}\left[\frac{\exp(-\frac{1}{\lambda} S(\tau)) \mathcal{H}(x_{t_j}, t_j)}{\mathbb{E}_{\mathbb{P}}[\exp(-\frac{1}{\lambda} S(\tau))]}\right] \mathbb{E}_{\mathbb{P}}\left[\frac{\exp(-\frac{1}{\lambda} S(\tau)) \mathcal{G}(x_{t_j}, t_j) \int_{t_j}^{t_{j+1}} dw^{(0)}}{\mathbb{E}_{\mathbb{P}}[\exp(-\frac{1}{\lambda} S(\tau))]}\right] \tag{3.48}$$

To approximate the controls, trajectories can be directly sampled from $\mathbb{P}$.

Matrix formulation

Supposing the control matrix and diffusion matrix have the form:

$$G = \begin{pmatrix} 0 \\ G_c \end{pmatrix}, \qquad B(x_t) = \begin{pmatrix} B_a(x_t) & 0 \\ 0 & B_c \end{pmatrix} \tag{3.49}$$

In this specific case, there are no correlations between noise in the states that are directly actuated either non-directly actuated, and the diffusion for the directly actuated states is state-independent. Hence, the covariance matrix becomes:

$$\Sigma(x) = \begin{pmatrix} B_a(x)B_a(x)^T & 0 \\ 0 & B_c B_c^T \end{pmatrix} \tag{3.50}$$

The terms $\mathcal{H}(x_{t_j})$ and $\mathcal{G}(x_t)$ are no longer state dependent, thus can be reduced to:

$$\mathcal{H} = G_c^T(B_c B_c^T)^{-1}G_c \quad \mathcal{G} = G_c^T(B_c B_c^T)^{-1}B_c \tag{3.51}$$

This means that the these matrices can be pulled out of the expectation, because they are state-independent:

$$u_j^* = \frac{1}{\Delta t}\mathcal{H}^{-1}\mathcal{G}\left(\mathbb{E}_{\mathbb{P}}\left[\int_{t_j}^{t_{j+1}} \frac{exp(-\frac{1}{\lambda}S(\tau))dw^{(0)}}{\mathbb{E}_{\mathbb{P}}[\exp(-\frac{1}{\lambda}S(\tau))]}\right]\right) \tag{3.52}$$

Numerical Approximation

At this point, it is necessary to numerically approximate Equation 3.52. However, they must be addressed two different problems:

1. re-writing the equation for sampling in discrete time
2. finding a way to perform importance sampling with Equation 3.52, because the expectation is with respect to the uncontrolled dynamics

In discrete time the dynamics of the system is:

$$dx_{t_j} = (f(x_{t_j}, t_j) + G(x_{t_j}, t_j)u_j)\Delta t + B(x_{t_j}, t_j)\epsilon_j\sqrt{\Delta t} \tag{3.53}$$

where ϵ_j is a vector where each entry is a standard normal random variable. Substituting $\epsilon_j\sqrt{\Delta t}$:

$$u_j^* = \frac{1}{\Delta t}\mathcal{H}^{-1}\mathcal{G}\left(\mathbb{E}_p\left[\frac{\exp(-\frac{1}{\lambda}S(\tau))\epsilon_j\sqrt{\Delta_j}}{\mathbb{E}_p[\exp(-\frac{1}{\lambda}S(\tau))]}\right]\right) \tag{3.54}$$

where p is the probability distribution corresponding to the discrete time uncontrolled dynamics (i.e. Equation 3.53 with u_{t_j} set to zero). Instead of sampling from p, it can be sampled from a different probability distribution q_u^v, which corresponds to sampling from the dynamics:

$$dx_{t_j} = (f(x_{t_j}, t_j) + G(x_{t_j}, t_j)u_j)\Delta t + B_E(x_{t_j}, t_j)\epsilon_j\sqrt{\Delta t} \tag{3.55}$$

where the new diffusion matrix is

$$B_E(x_t) = \begin{pmatrix} B_a(x_t) & 0 \\ 0 & \nu B_c \end{pmatrix} \tag{3.56}$$

with $\nu \leq 1$.
When sampling from this distribution, the designer gets to choose:

- the initial controls from which the sampling is centered about
- the magnitude of the exploration variance defined by ν

The **likelihood ratio** between the two distribution must be calculated in order to sample from q_u^v instead of p[21]. Inserting the likelihood ratio corresponds to changing the running cost from $q(x_t, t)$ to:

$$\begin{aligned} \tilde{q}(x_t, u_t, \epsilon_t, t) = q(x_t, t) + \frac{1}{2} u_t^T R u_t + \lambda u^T \mathcal{G} \frac{\epsilon}{\sqrt{\Delta t}} \\ + \frac{1}{2} \lambda (1 - \nu^{-1}) \frac{\epsilon^T}{\sqrt{\Delta t}} B_c^T (B_c B_c^T)^{-1} B_c \frac{\epsilon}{\sqrt{\Delta t}} \end{aligned} \tag{3.57}$$

The first two terms are penalties for shifting the mean of the exploration away from zero, instead the last term is a penalty for sampling from an over-aggressive variance. $B_c \frac{\epsilon}{\sqrt{\Delta t}}$ is the effective change in the control input due to noise.
When sampling from a distribution with non-zero control input, the Equation 3.52 also changes. This happens because the random variable under consideration shifts from the zero mean term $B_c \epsilon_j \sqrt{\Delta t}$ to the non-zero mean term $G_c u \Delta t + B_E \epsilon_j \sqrt{\Delta t}$. Let $\tilde{S}(\tau)$ be:

$$\tilde{S}(\tau) = \phi(x_T, T) + \sum_{j=0}^{N} \tilde{q}(x_t, u_t, \epsilon_t, t) \Delta t \tag{3.58}$$

On this way it is possible to obtain the resultant iterative update rule:

$$u_j^* = u_j + \mathcal{H}^{-1} \mathcal{G} \left(\mathbb{E}_{q_u^\nu} \left[\frac{\exp(-\frac{1}{\lambda} \tilde{S}(\tau) \frac{\epsilon_j}{\sqrt{\Delta t}}}{\mathbb{E}_{q_u^\nu} \left[\exp\left(-\frac{1}{\lambda} \tilde{S}(\tau) \right) \right]} \right] \right) \tag{3.59}$$

Approximating the term inside the parenthesis:

$$\sum_{k=1}^{K} \left(\frac{\exp\left(-\frac{1}{\lambda} \tilde{S}(\tau_k) \right) \frac{\epsilon_{j,k}}{\sqrt{\Delta t}}}{\sum_{k=1}^{K} \exp(-\frac{1}{\lambda} \tilde{S}(\tau_k))} \right) \tag{3.60}$$

where trajectory τ_k is drawn from the sampling dynamics 3.55. The iterative update rule can be seen as computing the new control based on a reward weighted average over trajectories.

Implementation

The MPPI predicitive controller has been created by Aleksei Budyakov, while adapted and developed for Nav2 by Steve Macenski.
Equations 3.59 and 3.60 offer an iterative update law that can be applied in a model predictive control setting. In this specific setting, optimization takes place dynamically: after optimizing the trajectory, a single control input is executed before re-optimization takes place. The path integral control derivation shown here provides a formula for optimizing the sequence of controls, not just the current time instance. This allows to warm start the optimization by reusing the portion of the control sequence that has not yet been run. On this way, the performance of the algorithm is enhanced, since, in general, only a limited number of iterations can be performed per timestep for a complex system operating at a suitable control frequency. The majority of the computation required for the path integral control update rule can be done in parallel, which is another important feature of real-time. The model predictive path integral control is shown in Figure 3.10.

Given: K: Number of samples;
N: Number of timesteps;
$(\mathbf{u}_0, \mathbf{u}_1, ...\mathbf{u}_{N-1})$: Initial control sequence;
$\Delta t, \mathbf{x}_{t_0}, \mathbf{f}, \mathbf{G}, \mathbf{B}, \mathbf{B}_E$: System/sampling dynamics;
$\phi, q, \mathbf{R}, \lambda$: Cost parameters;
$\mathbf{u}_{\text{init}}$: Value to initialize new controls to;

while *task not completed* **do**
  for $k \leftarrow 0$ **to** $K-1$ **do**
    $\mathbf{x} = \mathbf{x}_{t_0}$;
    for $i \leftarrow 1$ **to** $N-1$ **do**
      $\mathbf{x}_{i+1} = \mathbf{x}_i + (\mathbf{f} + \mathbf{G}\mathbf{u}_i)\Delta t + \mathbf{B}_E \epsilon_{i,k}\sqrt{\Delta t}$;
      $\tilde{S}(\tau_k) = \tilde{S}(\tau_k) + \tilde{q}(\mathbf{x}_i, \mathbf{u}_i, \epsilon_{i,k}, t_i)$;
  for $i \leftarrow 0$ **to** $N-1$ **do**
    $\mathbf{u}_i =$
    $\mathbf{u}_i + \boldsymbol{\mathcal{H}}^{-1}\boldsymbol{\mathcal{G}}\left[\sum_{k=1}^{K}\left(\frac{\exp\left(-\frac{1}{\lambda}\tilde{S}(\tau_k)\right)\frac{\epsilon_{j,k}}{\sqrt{\Delta t}}}{\sum_{k=1}^{K}\exp\left(-\frac{1}{\lambda}\tilde{S}(\tau_k)\right)}\right)\right]$;
  send to actuators($\mathbf{u}_0$);
  for $i \leftarrow 0$ **to** $N-2$ **do**
    $\mathbf{u}_i = \mathbf{u}_{i+1}$;
  $\mathbf{u}_{N-1} = \mathbf{u}_{\text{init}}$
  Update the current state after receiving feedback;
  check for task completion;

Figure 3.10: Model Predictive Path Integral Control

The control implementation for Nav2 contains plugin-based critic functions to impact the behavior of the algorithm:

- Constraint Critic

- Goal Angle Critic
- Goal Critic
- Obstacles Critic
- Cost Critic
- Path Align Critic
- Path Angle Critic
- Prefer Forward Critic
- Twirling Critic

The most important features of this controller are:

- predictive MPC trajectory planner
- plugin-based critics can be easily swapped out, tuned or replaced by the user
- highly optimized CPU-only performance using vectorization and tensor operations
- support of a number of common motion models, including Ackermann, Differential Drive and Omni-Directional
- fallback mechanisms are included to handle soft-failures before escalating to recover behaviors
- high unit test coverage, documentation and parameter guide
- easy extensibility to support modern search variants of MPPI

3.2 Settings of the analysis

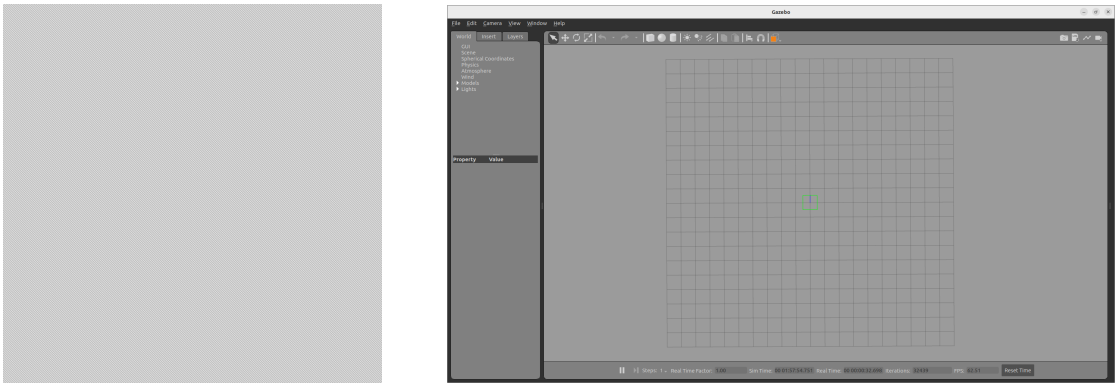
At this point, they have been chosen the use cases and the performance indexes that had to be analyzed.
They have been created 5 different simulation environments:

1. **empty world**, to evaluate simple navigation without external influence (Figure 3.11a and 3.11b)
2. world including **sparse static obstacles**, to evaluate collision avoidance (Figure 3.12a and 3.12b)

3. world including only **one static obstacle**, to evaluate smoothness in sharp curves (Figure 3.13a and 3.13b)

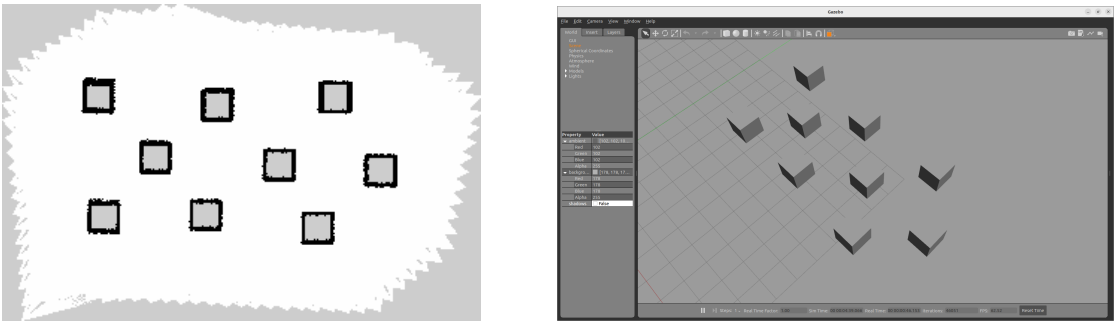
4. **restricted world, with obstacles close to the robot**, to evaluate the capability of the robot of immediate abortion of impossible tasks (Figure 3.14a and 3.14b)

5. **restricted world, with obstacles distant from the robot**, to evaluate the capability of the robot to manage impossible tasks (Figure 3.15a and 3.15b)



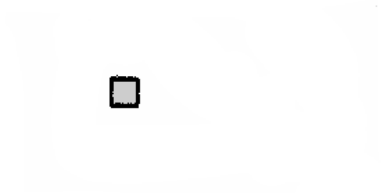
(a) Map pf the empty world. **(b)** Simulation environment of the empty world.

Figure 3.11: Empty world simulation.

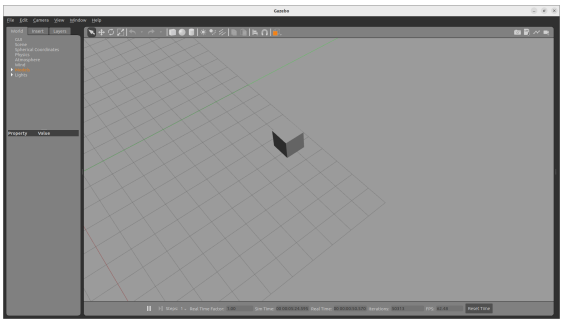


(a) Map of the world with static obstacles. **(b)** Simulation environment with static obstacles.

Figure 3.12: Static Obstacles simulation.

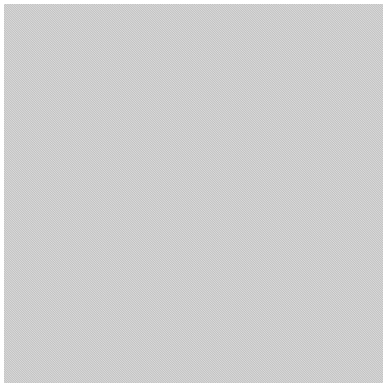


(a) Map of the world with only one static obstacle.

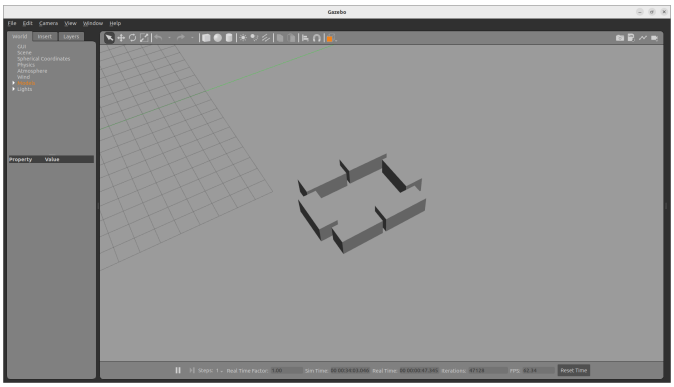


(b) Simulation environment with only one static obstacles.

Figure 3.13: One Static Obstacle simulation.

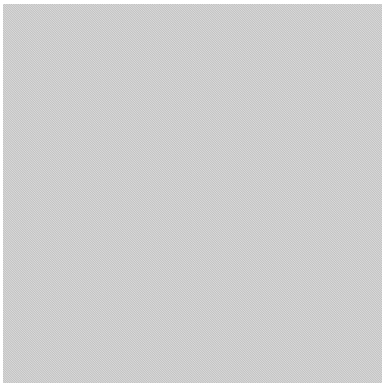


(a) Map of the world with small restricted area.

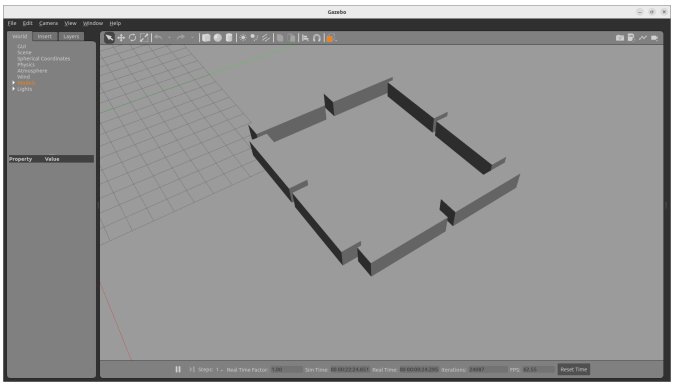


(b) Simulation environment with small restricted area.

Figure 3.14: Small restricted area simulation.



(a) Map of the world with large restricted area.



(b) Simulation environment with large restricted area.

Figure 3.15: Large restricted area simulation.

Each simulation consisted in the robot's execution of the same task repeated 4 times, in order to get average information on multiple occurrences. In such a way, the analysis is more significant.
Moreover, it is possible to notice that in the last 2 use cases, the maps are empty, despite the presence of obstacles in the simulation. This has been done in order to test the controllers capability of generating paths without knowing a priori the map of the simulation environment and without the usage of the obstacle and inflation layers.
The chosen parameters for all the simulations are:

- **Success rate** - percentage of successful tasks over the total number of trials
- **Average linear speed** - average of robot's speed on the x axis in m/s
- **Average length of the path navigated by the robot** in m
- **Average time taken** - average of time taken by the robot in order to successfully reach the goal in s
- **Average integrated jerk** [22] - average of the jerk on x axis in m^2/s^6
- **Maximum Centripetal Acceleration** in m/s^2
- **Maximum velocity** in m/s
- **Maximum path error** - maximum linear displacement between the global planned path and local planned one in m

- **Minimum path error** - minimum linear displacement between the global planned path and local planned one in m
- **Average path error** - average linear displacement between the global planned path and local planned one in m
- **Average heading error** and its **standard deviation** - average angular displacement between the global planned path and local planned one in *degrees*
- **Average Energy Consumption** [22] in J

For what concerns the simulation including obstacles, they have been also taken into consideration the **minimum distance from obstacles** and its **standard deviation** (in m).
Instead, the restricted area simulation analysis, included also the computation of the **minimum time for abortion** (in s), which gives the time between the first time the robot stops, trying to re-plan a new path, and the last time, when the robot finally aborts the task.

3.3 Code implementation

The project is developed in Python and consists in:

- 2 files used for the launch of the simulations - `tb3_simulation_only_launch.py` and `controller_benchmark_bringup.py`
- 1 file containing the tuned parameters - `controller_benchmark.yaml`
- 1 file containing the specifics on the starting and the target points and the code saving data of the simulation (state of completion of the task, velocity of the robot, the list of poses of the robot, costmaps and global path planned) - `metrics.py`
- 1 file containing the code that processes data, in order to provide the parameters chosen for the analysis - `process_data.py`

Each of the described files, except for `tb3_simulation_only_launch.py`, are specifically implemented for each simulation, in order to better manage the shifting from one use case to the other.
The computation of the parameters is described in the following section.

Perfomance indexes

The **success rate** is basically computed dividing the number of successfully completed tasks over the total number of task.

```
str(round(np.sum(tasks_results[i])/len(tasks_results)*100))+"%"
```

This formulation is repeated for the number of controllers taken into account.

The **average linear speed** is computed on this way:

```
def getTaskAvgLinearSpeed(controller_twists):
linear_x = []
for twist_stamped in controller_twists:
    linear_x.append(twist_stamped.twist.linear.x)
return  np.average(linear_x)
```

This function creates an array (*linear_x*), fills it with all the robot's velocities sampled during the simulation and then returns its average.

The **average length of the path navigated by the robot** is computed by summing iteratively the square root of the sum of squares of the dispacements along x and y axes.

```
def getPathLength(path):
path_length = 0
x_prev = path.poses[0].pose.position.x
y_prev = path.poses[0].pose.position.y
for i in range(1, len(path.poses)):
    x_curr = path.poses[i].pose.position.x
    y_curr = path.poses[i].pose.position.y
    path_length = path_length + math.sqrt((x_curr-x_prev)**2 + (
y_curr-y_prev)**2)
    x_prev = x_curr
    y_prev = y_curr

return path_length
```

The **average time taken** is computed by doing the time difference between the timestamps of the first and last poses

```
def getTaskTimes(controller_poses):

```

```
    return  (controller_poses[-1].header.stamp.nanosec/1e09+
controller_poses[-1].header.stamp.sec)-\
            (controller_poses[0].header.stamp.nanosec/1e09+
controller_poses[0].header.stamp.sec)
```

The **maximum centripetal acceleration** is computed along with the **average integrated jerk** in the following method:

```
def getControllerMSJerks(controller_twists, controller_poses):
    linear_x = []
    linear_y = []
    angular_z = []
    coordinates_x = []
    coordinates_y = []
    acc_centripetal = []

    time_passed = 0.0
    for twist_stamped in controller_twists:
        linear_x.append(twist_stamped.twist.linear.x)
        linear_y.append(twist_stamped.twist.linear.y)
        angular_z.append(twist_stamped.twist.angular.z)
        time_passed+=twist_stamped.header.stamp.nanosec/1e09+
twist_stamped.header.stamp.sec

    for poses in controller_poses:
        coordinates_x.append(poses.pose.position.x)
        coordinates_y.append(poses.pose.position.y)

    for i in range(len(controller_poses)):
        if angular_z[i] < 0:
            angular_z[i] = abs(angular_z[i])
        if angular_z[i] > 1e-2:
            r = linear_x[i]/angular_z[i]
            if r > 1e-2:
                acc_centripetal.append((linear_x[i]**2)/r)
        else:
            acc_centripetal.append(0.0)

    end = controller_twists[-1].header.stamp.nanosec/1e09+
controller_twists[-1].header.stamp.sec
    start= controller_twists[0].header.stamp.nanosec/1e09+
controller_twists[0].header.stamp.sec
    dt = (end-start)/len(controller_twists)

    # Discrete derivative of linear velocity = linear acceleration
    linear_acceleration_x = np.diff(linear_x,axis = 0) / dt

    # Discrete derivative of linear acceleration = linear jerk
```

```
38      linear_jerk_x = np.diff(linear_acceleration_x,axis=0) / dt
39
40      # Mean Squared jerk Wininger, Kim, & Craelius (2009)
41      ms_linear_jerk_x = 0.0
42      for jerk in linear_jerk_x:
43          ms_linear_jerk_x += jerk**2
44
45      if len(linear_jerk_x)>0:
46          rmse_linear_jerk = math.sqrt((ms_linear_jerk_x)/len(
    linear_jerk_x))
47      else:
48          rmse_linear_jerk = np.nan
49
50      acc = np.max(acc_centripetal)
51
52      return rmse_linear_jerk, acc
```

In details, at first they have been created 5 different arrays containing respectively samplings of the robot's linear velocity along x axis, samplings of the robot's angular velocity along z axis, x coordinates of the robot's poses, y coordinates of the robot's poses and samplings of the robot's centripetal acceleration. After filling the arrays, it has been computed the infinitesimal time period between a pose and the following one, in order to differentiate the robot's linear acceleration along x axis, and, consequently, obtain the linear jerk along x axis. Finally, this method returns the maximum centripetal acceleration and the root mean square of the linear jerk of one controller executing only one task.

The **maximum velocity** is computed by the execution of this method:

```
1       def getMaxVel(controller_twists):
2       linear_x = []
3       for twist_stamped in controller_twists:
4           linear_x.append(twist_stamped.twist.linear.x)
5
6       return np.max(linear_x)
```

Hence, the computation is much similar to the one implemented for the definition of the robot's average speed. The only difference is that, obviously, it returns the maximum sampled value of the robot's linear velocity.

The **path error** and the **heading error** are computed by the same function:

```
1       def getControllerPathHeadingError(path,controller_poses):
2
```

```
    x = []
    y = []
    q = []
    # Getting x and y of controller
    for controller_coordinates in controller_poses:
        x.append(controller_coordinates.pose.position.x)
        y.append(controller_coordinates.pose.position.y)
        q.append(controller_coordinates.pose.orientation)

    xp = []
    yp = []
    qp = []
    for path_poses in path.poses:
        xp.append(path_poses.pose.position.x)
        yp.append(path_poses.pose.position.y)
        qp.append(path_poses.pose.orientation)

    err = []
    head = []

    # for each point along controller trajectory
    for i in range(len(xp)):
        d_min = 100.0
        # for each couple of points along global trajectory
        for j in range(len(x)):

            d = math.sqrt((x[j]-xp[i])**2+(y[j]-yp[i])**2)

            if d < d_min:
                d_min = d
                jmin = j

            [Xp,Yp,Zp] = QuatToEulerAngle(qp[i])
            [Xc,Yc,Zc] = QuatToEulerAngle(q[jmin])

        if Zc < 0:
            Zc = Zc*-1
        if Zp < 0:
            Zp = Zp*-1

        head.append(abs(Zc-Zp))
        err.append(d_min)
    err = err[:-4]

    return (head, err)
```

Firstly, they have been filled 3 arrays, containing the coordinates and the quaternions (orientation) of the robot, and other 3 arrays, containing the coordinates and the quaternions planned. Then, they have been initialized 2 arrays: the path

error array and the heading error array. The first one is filled with the minimum distance between the robot's pose and the correspondent planned pose, while the second one with the difference between the orientation of the robot in the same pose and in the correspondent planned one. Moreover, it is possible to notice that the last 4 path error values are neglected: it must be done because sometimes the robot stops when the distance between itself and the target point is sufficiently small.
Finally, the maximum, minimum and average values are computed in a for loop.
The **average energy consumption** is computed by this method:

```
def getEnergy(controller_twists, controller_poses):
    linear_x = []
    coordinates_x = []
    coordinates_y = []
    time_passed = 0.0
    for twist_stamped in controller_twists:
        linear_x.append(twist_stamped.twist.linear.x)
        time_passed+=twist_stamped.header.stamp.nanosec/1e09+
    twist_stamped.header.stamp.sec

    for poses in controller_poses:
        coordinates_x.append(poses.pose.position.x)
        coordinates_y.append(poses.pose.position.y)

    end = controller_twists[-1].header.stamp.nanosec/1e09+
    controller_twists[-1].header.stamp.sec
    start= controller_twists[0].header.stamp.nanosec/1e09+
    controller_twists[0].header.stamp.sec
    dt = (end -start)/len(controller_twists)

    energy = 0.0
    for i in range(len(controller_poses)-1):
        displacement_x = controller_poses[i+1].pose.position.x-
    controller_poses[i].pose.position.x
        displacement_y = controller_poses[i+1].pose.position.y-
    controller_poses[i].pose.position.y
        energy = energy + displacement_x**2/dt + displacement_y**2/dt
    )

    return energy
```

In details, after filling the arrays of the samplings of robot's linear velocity, x coordinates and y coordinates of its poses, and computing the infinitesimal time period between a pose and the following one, the energy spent for making the robot move by that time is computed. The computation consists in the execution of the integral of the squares of the velocity along x and y axes. Since the simulations are

obviously done in discrete time, this computation can be simplified by summing the squares of the displacement along x and y axes over the infinitesimal time period. Finally, this function returns the total energy expenditure of a controller executing only one task by summing all these values.

The **minimum distance from obstacles** is computed on this way:

```
1     def getObstacleDistances(tasks_local_costmaps,
      local_costmap_resolution,tasks_controller_local_costmaps_meta,
      tasks_poses):
2
3     # Will contains minimum obstacle distance for each controller
4     controllers_obstacle_distances_min = []
5     controllers_min_obstacle_distances_std = []
6     # for each task / navigation
7
8     for task_local_costmaps, task_poses, task_local_costmaps_meta in
      zip(tasks_local_costmaps, tasks_poses,
      tasks_controller_local_costmaps_meta):
9         # for each controller
10        for controller_local_costmaps, controller_poses,
      controller_local_costmaps_meta in zip(task_local_costmaps,
      task_poses, task_local_costmaps_meta):
11            min_obst_dist = 1e10
12            # Will contain history of nearest obstacle distance of
      the controller
13            # for this task, used to compute man and std
14            controller_task_min_obstacle_distances = []
15
16            for local_costmap, pose, local_costmap_meta in zip(
      controller_local_costmaps, controller_poses,
      controller_local_costmaps_meta):
17                # Definition of the origin of each costmap
18                origin_x = local_costmap_meta.origin.position.x
19                origin_y = local_costmap_meta.origin.position.y
20
21                #Definition of the current position of the robot
22                x_r = pose.pose.position.x
23                y_r = pose.pose.position.y
24
25                xmin = 0
26                ymin = 0
27
28                if local_costmap.max()>FATAL_COST:
29                    # Look for obstacle
30                    obstacles_indexes = np.where(local_costmap>
      FATAL_COST)
```

```
31                      obstacles_indexes_x = obstacles_indexes[1]
32                      obstacles_indexes_y = obstacles_indexes[0]
33
34                      iteration_minimum_distance =  1e10
35                      for x, y in zip(obstacles_indexes_x,
    obstacles_indexes_y):
36
37                          obstacle_distance = math.sqrt((x*
    local_costmap_resolution + origin_x - x_r)**2 + (y*
    local_costmap_resolution + origin_y - y_r)**2)
38
39                          if obstacle_distance <
    iteration_minimum_distance:
40                              iteration_minimum_distance =
    obstacle_distance
41
42                          if obstacle_distance < min_obst_dist:
43                              min_obst_dist = obstacle_distance
44
45                  else:
46                      iteration_minimum_distance = np.nan
47                  if iteration_minimum_distance == 1e10:
48                      iteration_minimum_distance = np.nan
49
50                  controller_task_min_obstacle_distances.append(
    iteration_minimum_distance)
51              controllers_obstacle_distances_min.append(np.nanmin(
    controller_task_min_obstacle_distances))
52              controllers_min_obstacle_distances_std.append(np.nanstd(
    controller_task_min_obstacle_distances))
53          if (min_obst_dist == 1e10):
54              min_obst_dist = np.nan
55
56      return controllers_obstacle_distances_min,
    controllers_min_obstacle_distances_std
```

This function computes, for each controller executing only one task, the minimum distance between the robot and an obstacle in each local costmap. Basically, when the maximum value of the local costmap is greater than 253 (`FATAL_COST`), they have been saved all the indexes of the local costmap containing values greater than `FATAL_COST`. Then, the specific position of the obstacle is determined by multiplying the indexes times the resolution of the local costmap and summing them to the coordinates of the origin of the local costmap. Finally, the distance is computed doing the difference between the obstacle's and robot's positions.

The **time for abortion** is computed by this function:

```
    def getMaxTimeAbortion(controller_twists):
    linear_x = []
    angular_z = []
    for twist_stamped in controller_twists:
        linear_x.append(twist_stamped.twist.linear.x)
        angular_z.append(twist_stamped.twist.angular.z)
        cnt = 0
        for i in range(len(linear_x)-1):
            if linear_x[i] == min(linear_x):
                if min(linear_x)<0.5:
                    cnt += 1
                    if cnt == 1:
                        end = controller_twists[-1].header.stamp.
nanosec/1e09+controller_twists[-1].header.stamp.sec
                        start= controller_twists[i].header.stamp.
nanosec/1e09+controller_twists[i].header.stamp.sec
                        time_passed = end-start
                elif min(linear_x)==0.5:
                    time_passed = 0.0

    return time_passed
```

where it has been calculated the time passed between the first time in which the robot stops, trying to recalculate the path that has to be navigated, and the last time, when the robot ends the execution of the task. This calculation returns 0 in the specific case of RPP controller, because it does not make the robot stops before it reaches a pose that is close enough to the obstacle.

Chapter 4

Obtained results

4.1 Empty World simulation

Figure 4.1 and 4.2 show the robot in the simulation environment.

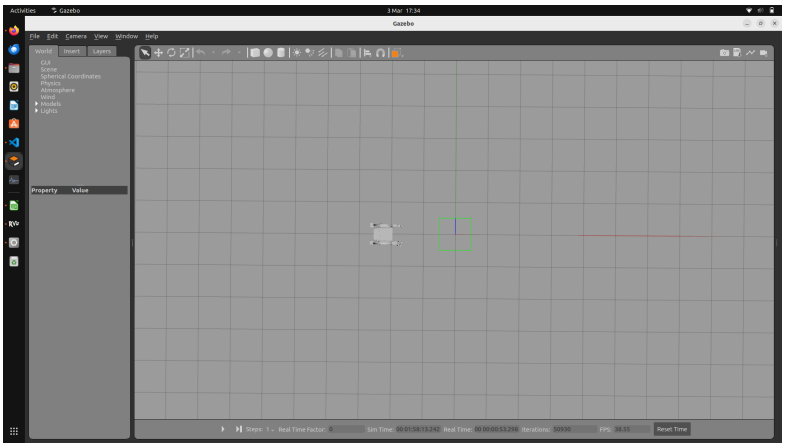


Figure 4.1: Empty World simulation.

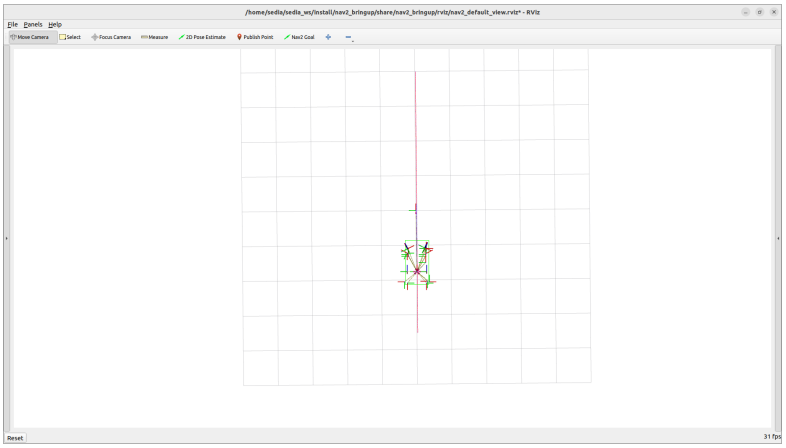


Figure 4.2: Empty world map.

The simulation in the empty world provided the following results:

Performance indexes	**DWB**	**RPP**	**TEB**	**MPPI**
Success rate	100.00%	100.00%	100.00%	100.00%
Avg linear speed (*m/s*)	0.466/0.5	0.5/0.5	0.476/0.5	0.459/0.5
Avg path length (*m*)	7.238	7.231	7.247	7.236
Avg time taken (*s*)	15.498	14.512	15.14	15.66
Avg integrated x jerk (m^2/s^6)	1.811	0.393	1.445	0.834
Max velocity (*m/s*)	100.00%	100.00%	100.00%	100.00%
Max local planned path error (*m*)	0.062	0.063	0.051	0.061
Min local planned path error (*m*)	0.00	0.00	0.00	0.00
Avg local planned path error (*m*)	0.023	0.017	0.017	0.017
Avg heading error (°)	0.495	0.373	0.358	0.349
Std heading error (°)	0.005	0.001	0.023	0.025
Energy Consumption (*J*)	3.509	3.647	3.55	3.475

Table 4.1: Results of the simulation with the Empty World simulation

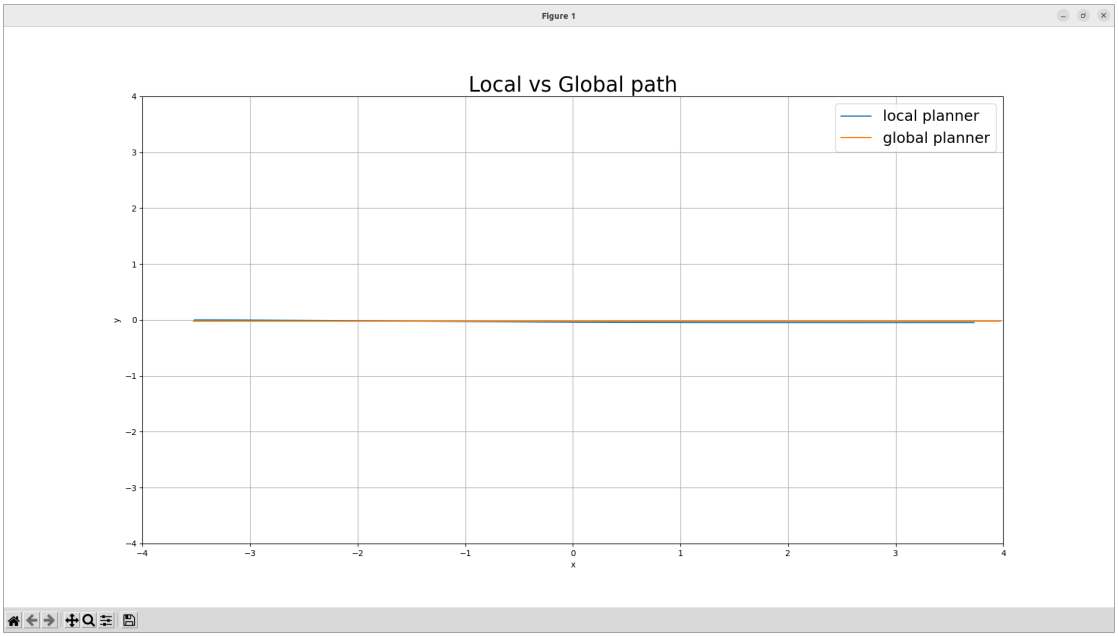


Figure 4.3: Difference between global planner and DWB in the Empty World simulation.

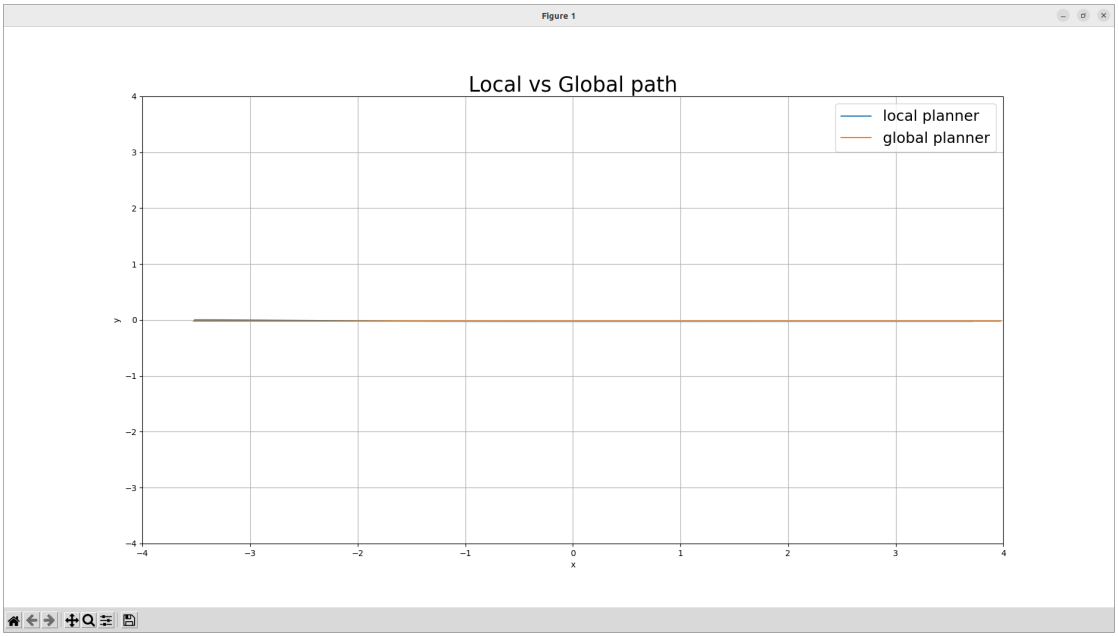


Figure 4.4: Difference between global planner and RPP in the Empty World simulation.

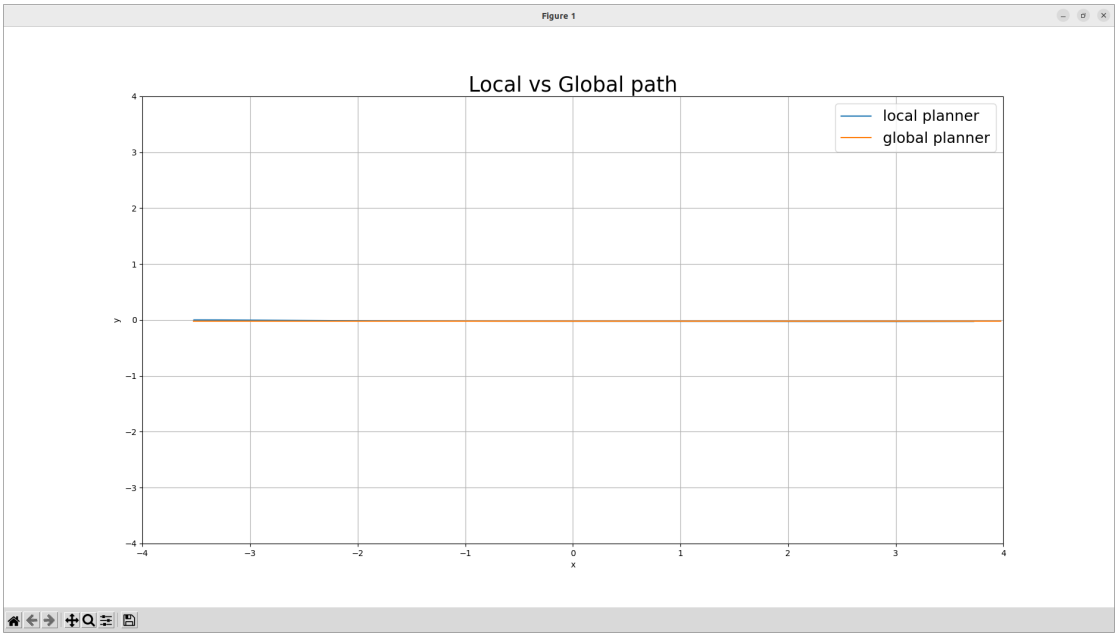


Figure 4.5: Difference between global planner and TEB in the Empty World simulation.

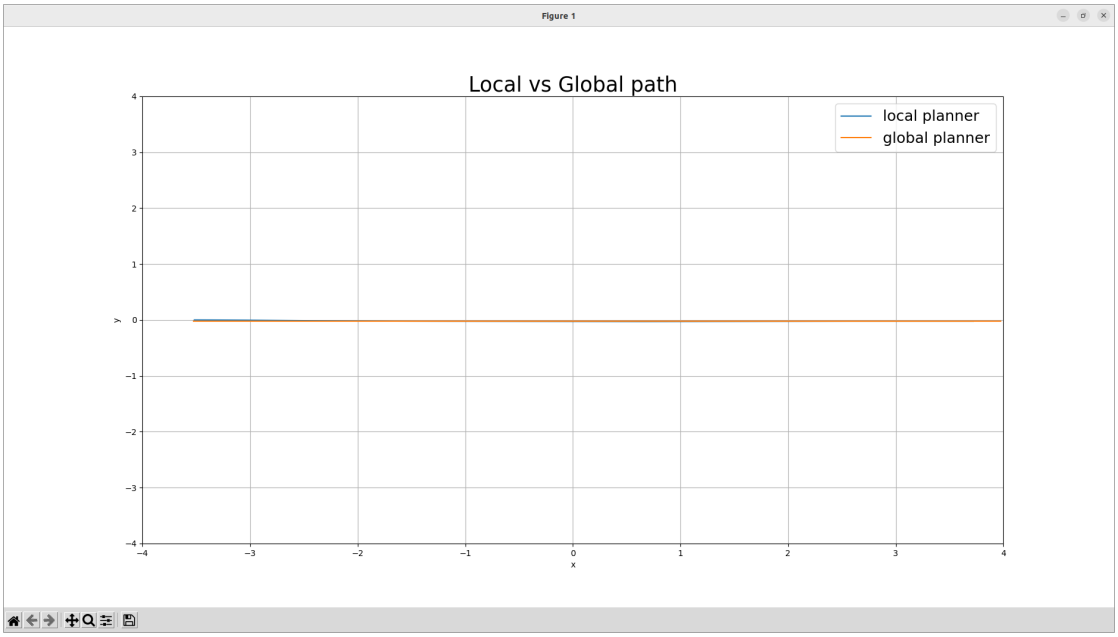


Figure 4.6: Difference between global planner and MPPI in the Empty World simulation.

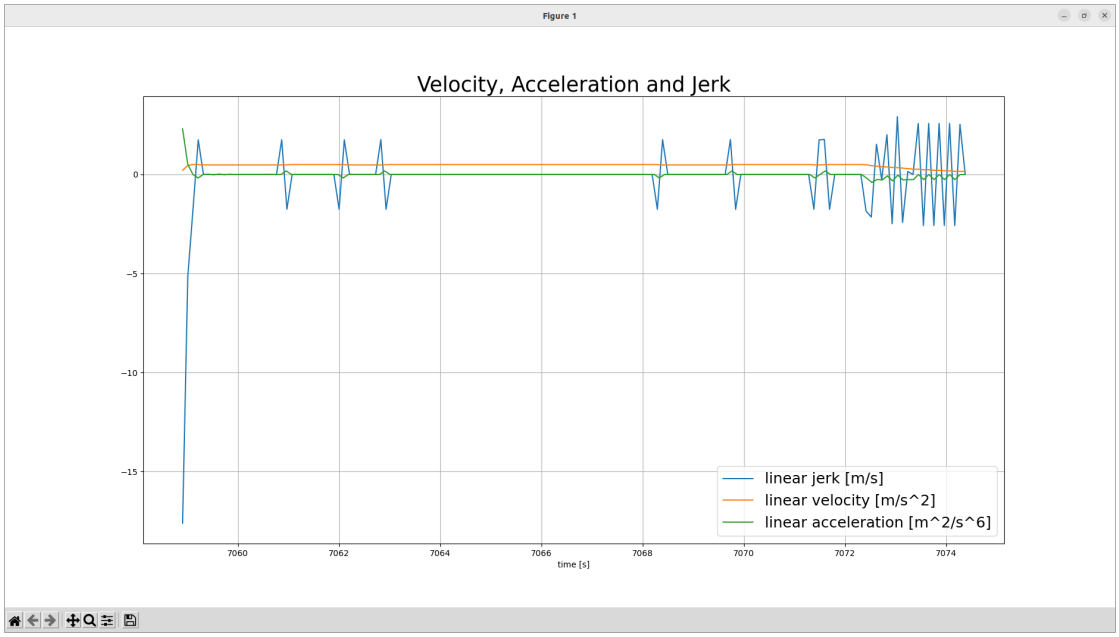


Figure 4.7: Velocity, Acceleration and Jerk of DWB controller.

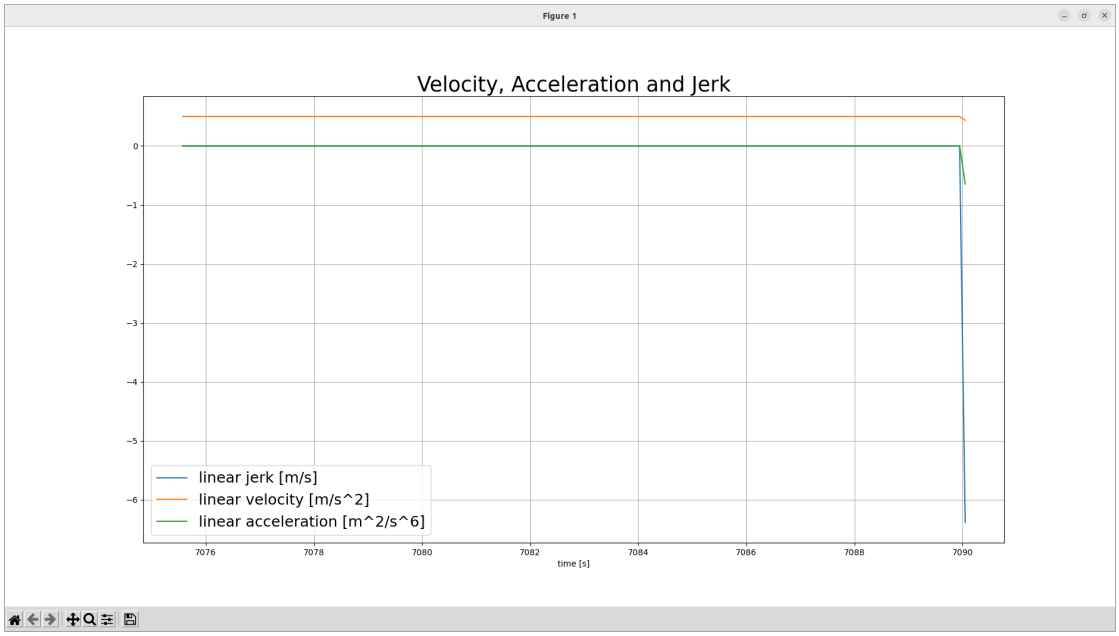


Figure 4.8: Velocity, Acceleration and Jerk of RPP controller.

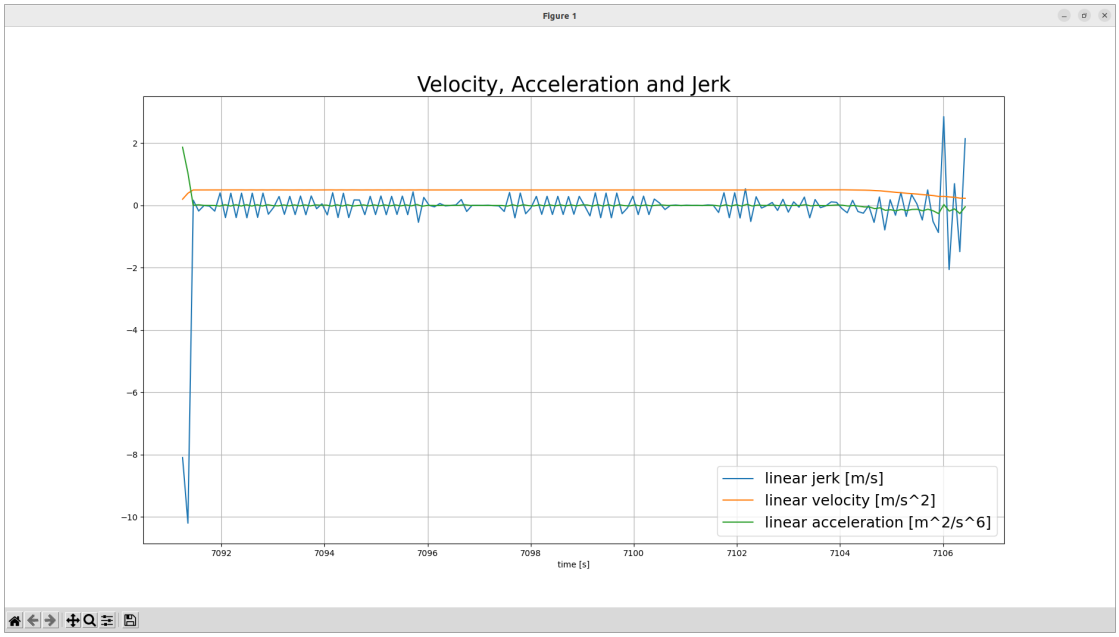


Figure 4.9: Velocity, Acceleration and Jerk of TEB controller.

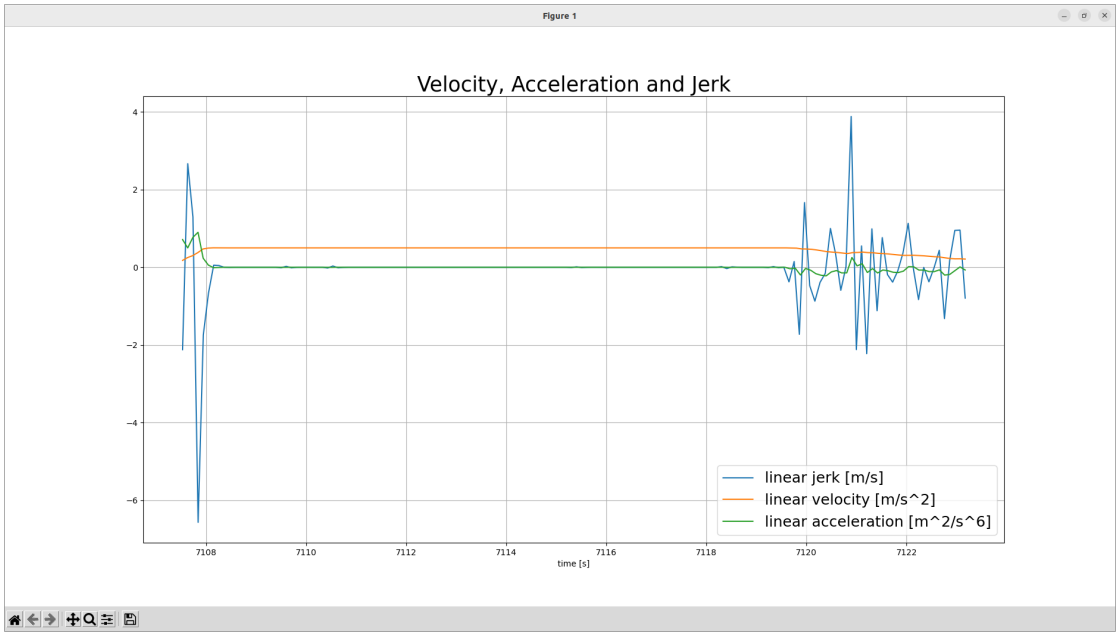


Figure 4.10: Velocity, Acceleration and Jerk of MPPI controller.

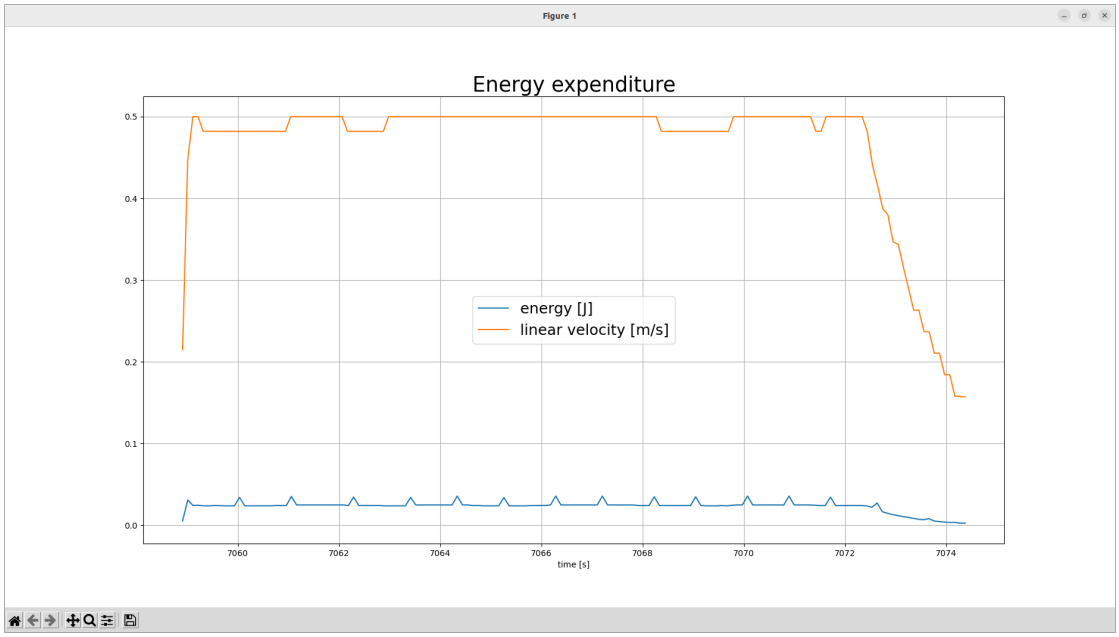


Figure 4.11: Energy expenditure and velocity of DWB.

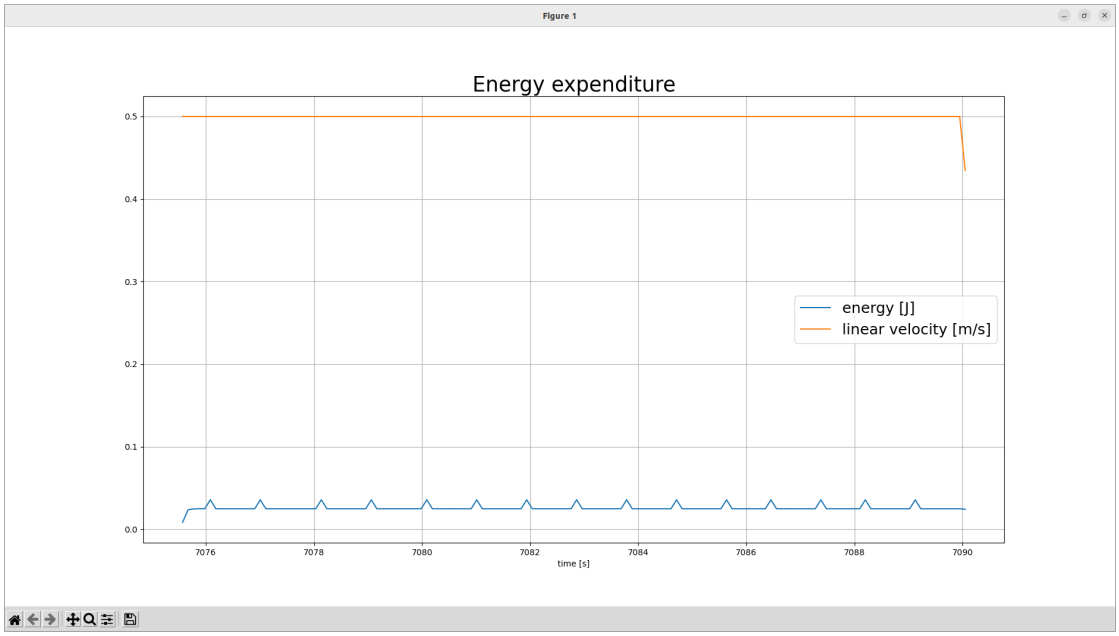


Figure 4.12: Energy expenditure and velocity of RPP.

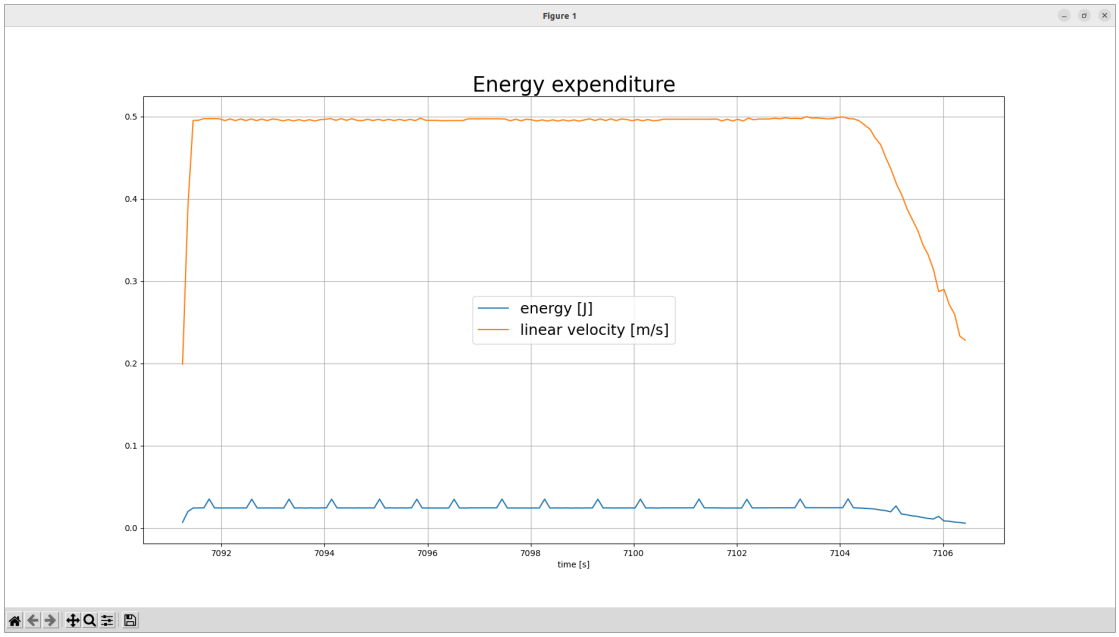


Figure 4.13: Energy expenditure and velocity of TEB.

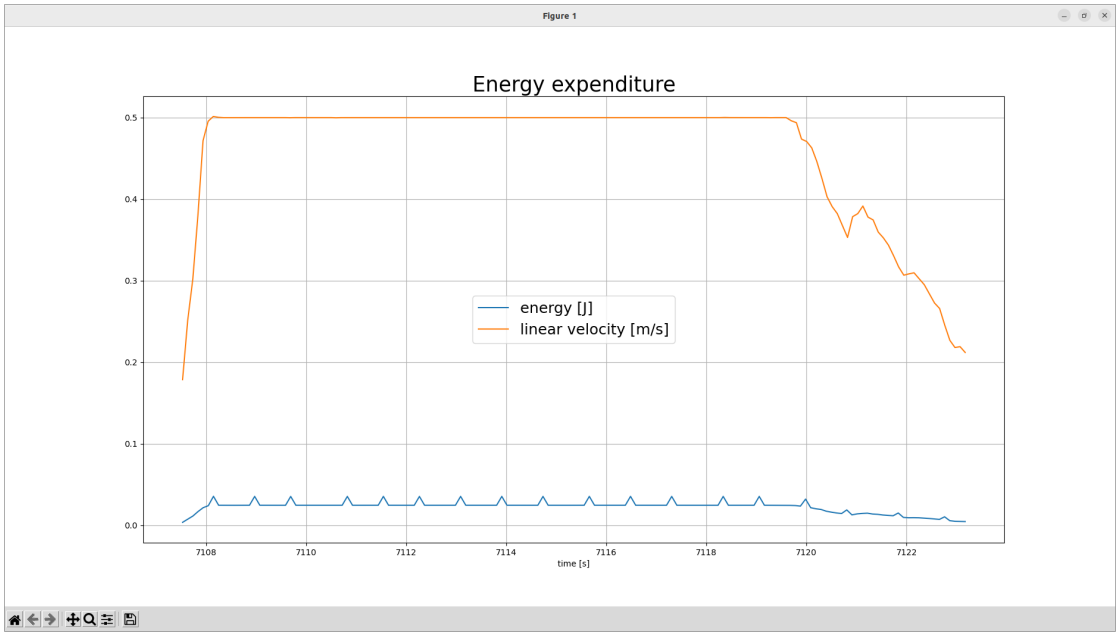


Figure 4.14: Energy expenditure and velocity of MPPI.

4.1.1 Observations

It is possible to notice that the **RPP** controller is the fastest one: it reaches the goal at maximum possible speed (0.5 m/s) spending less time than all the other controllers and with the minimum error between the local and global planned path.

The energy consumption is quite similar to every controller, however the RPP has a slightly higher value because it does not make the robot decelerate before reaching the target point. Instead, it suddenly stops, spending more energy than the other controllers. The link between the average linear speed and the energy expenditure is clear in Figure 4.11, Figure 4.12, Figure 4.13 and Figure 4.14: the energy consumption decreases when the robot's linear speed decreases gradually.

The DWB controller's acceleration seems to significantly change in a very short period of time, because the peaks in Figure 4.7 reaches the highest value above all the controllers taken into account. Instead, MPPI and TEB controllers accelerate and decelerate in a smoother way. However, the RPP, as expected, navigates with the same maximum velocity through the whole path, until it reaches the goal at 0 m/s.

4.2 Static Obstacles simulation

Figure show the robot in the simulation environment with the static obstacles.

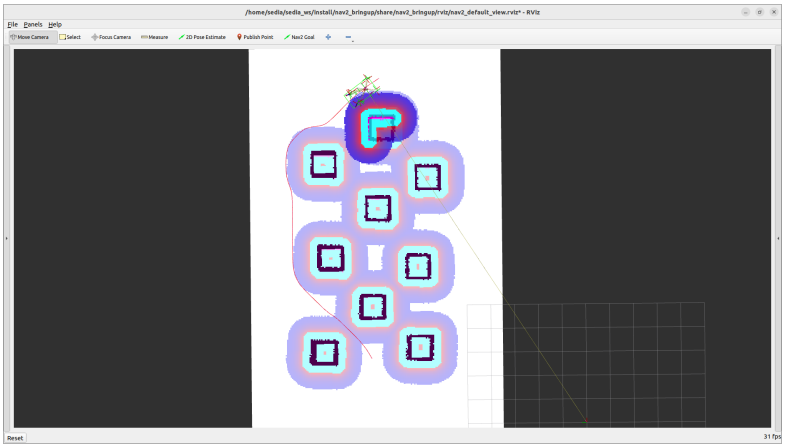


Figure 4.15: Static Obstacles world simulation.

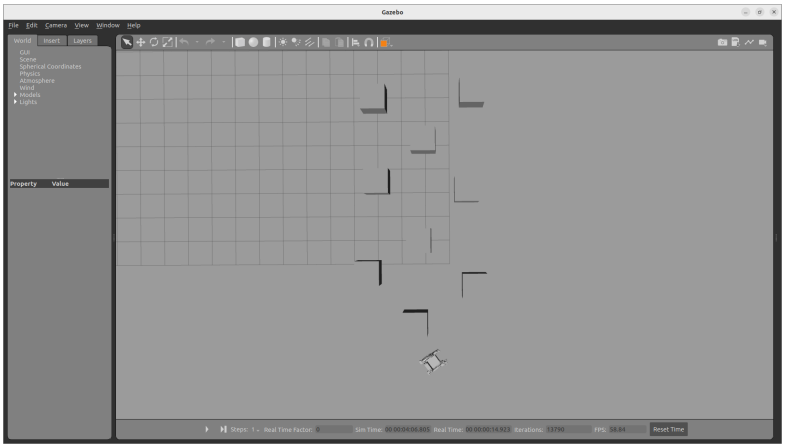


Figure 4.16: Static Obstacles world map.

The results obtained by this simulations are shown the Table 4.2:

Controller	**DWB**	**RPP**	**TEB**	**MPPI**
Success rate	100.00%	100.00%	100.00%	100.00%
Avg linear speed (*m/s*)	0.482/0.5	0.494/0.5	0.479/0.5	0.475/0.5
Avg path length (*m*)	14.776	14.883	14.685	14.352
Avg time taken (*s*)	30.704	30.107	30.637	30.049
Min distance from obstacle (*m*)	0.646	0.636	0.622	0.571
Std distance from obstacle (*m*)	0.221	0.216	0.239	0.304
Avg integrated x jerk (m^2/s^6)	2.266	0.193	1.376	0.601
Max centripetal acceleration (m/s^2)	0.242	0.215	0.172	0.149
Max velocity (*m/s*)	100.00%	100.00%	100.00%	100.00%
Max local planned path error (*m*)	0.115	0.074	0.137	0.741
Min local planned path error (*m*)	0.00	0.00	0.00	0.00
Avg local planned path error (*m*)	0.044	0.026	0.069	0.217
Avg heading error (°)	5.23	3.663	6.92	11.771
Std heading error (°)	0.019	0.024	0.053	1.48
Energy Consumption (*J*)	7.26	7.443	7.151	7.071

Table 4.2: Results of the Static Obstacles simulation

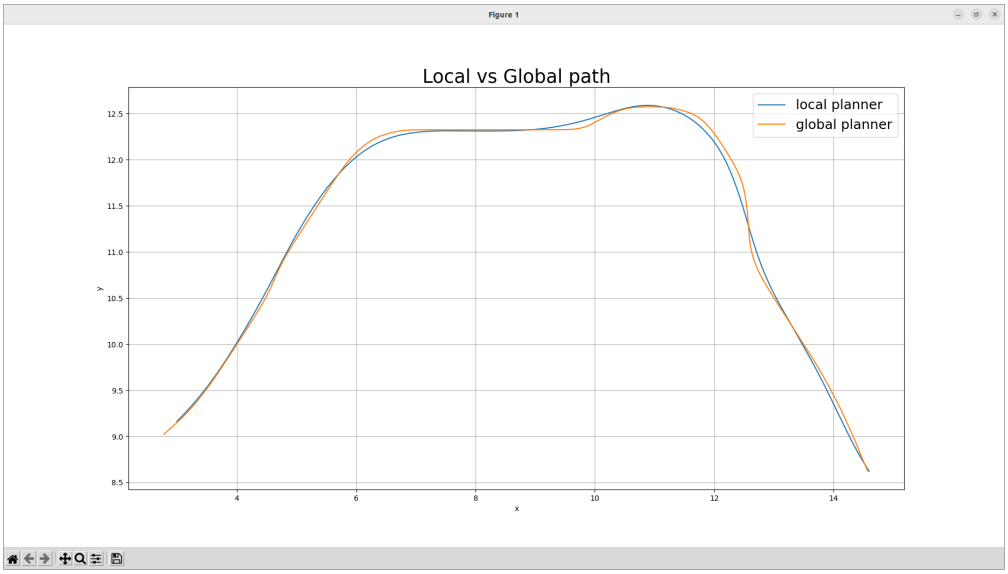


Figure 4.17: Difference between global planner and DWB in the Static Obstacles simulation.

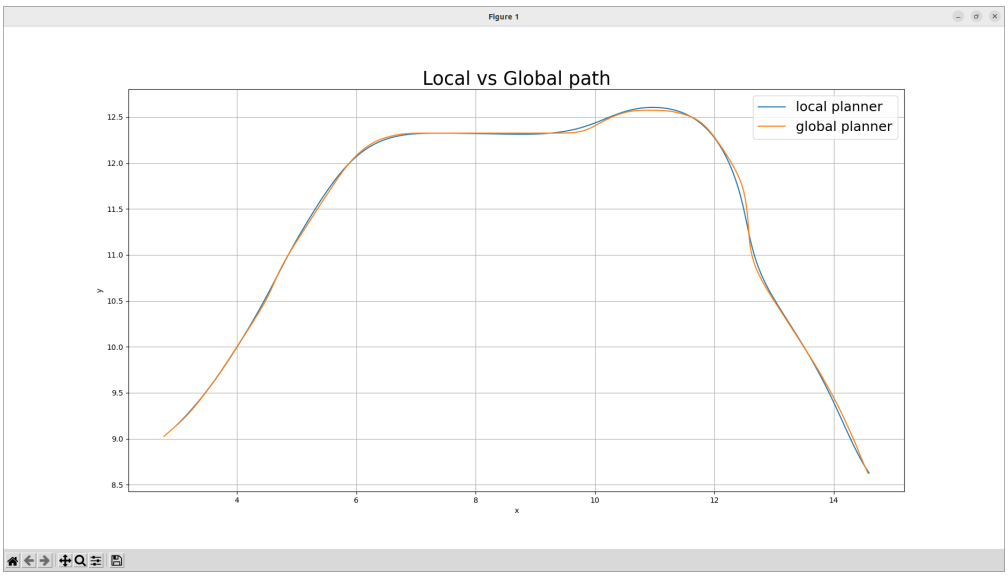


Figure 4.18: Difference between global planner and RPP in the Static Obstacles simulation.

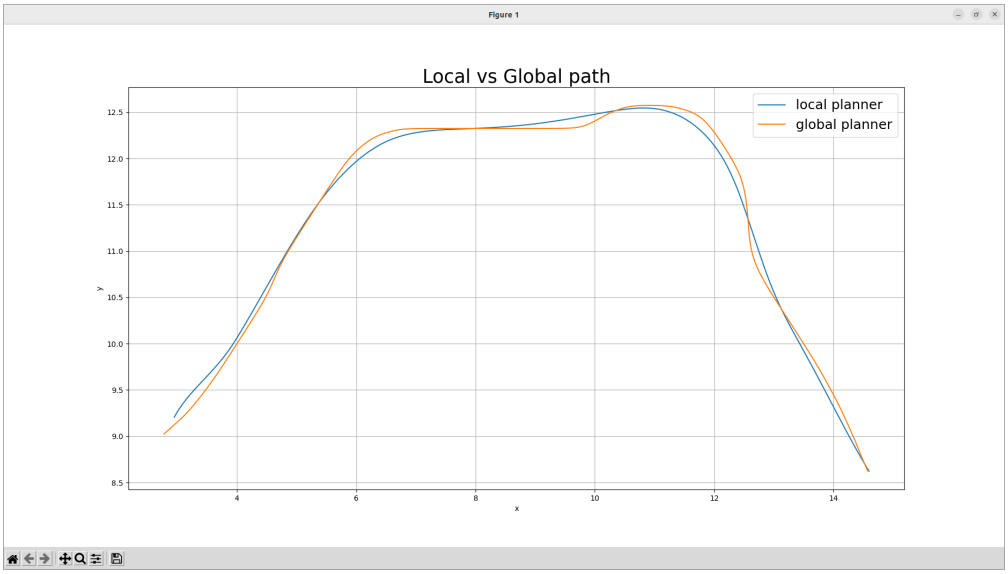


Figure 4.19: Difference between global planner and TEB in the Static Obstacles simulation.

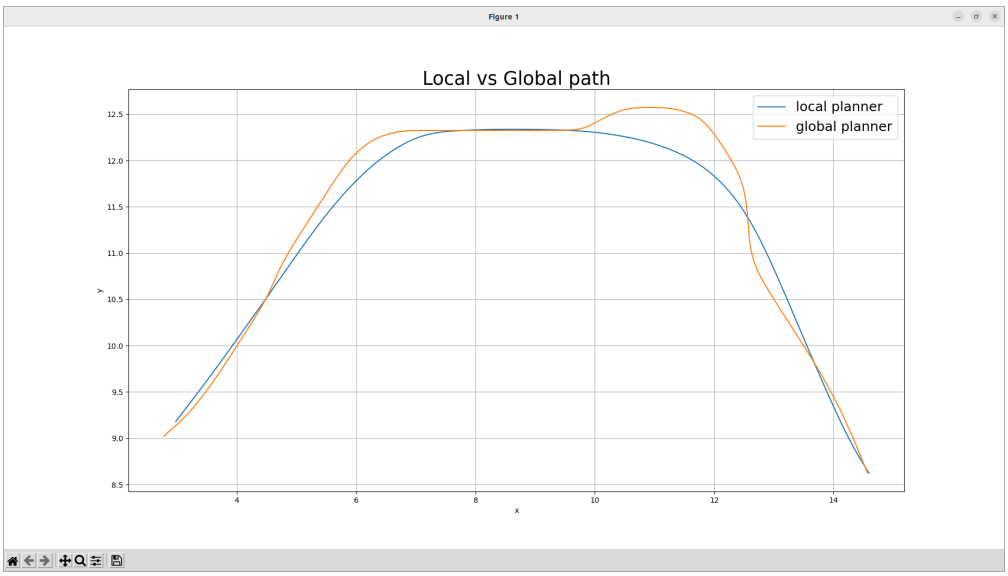


Figure 4.20: Difference between global planner and MPPI in the Static Obstacles simulation.

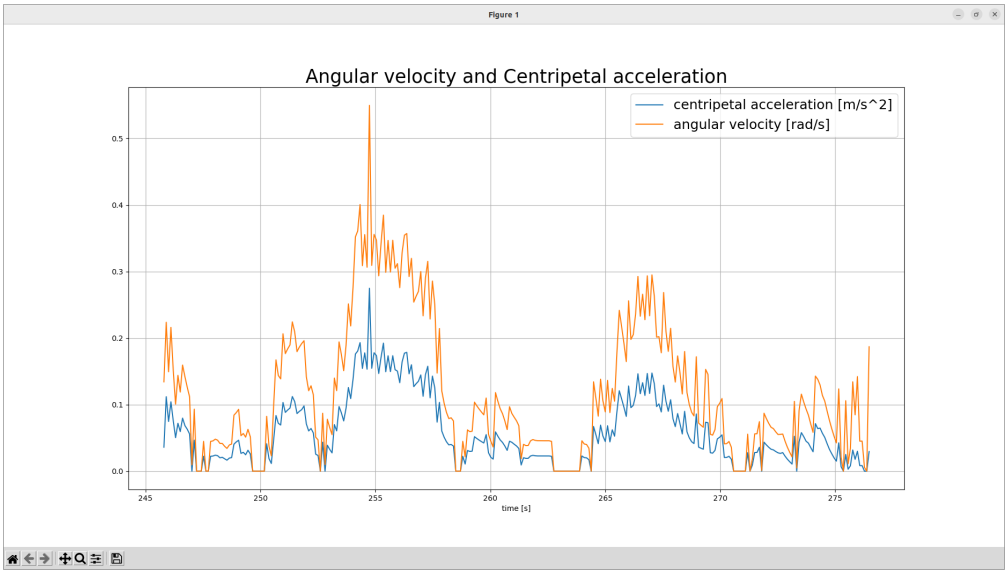


Figure 4.21: Angular velocity and Centripetal acceleration of DWB in the Static Obstacles simulation.

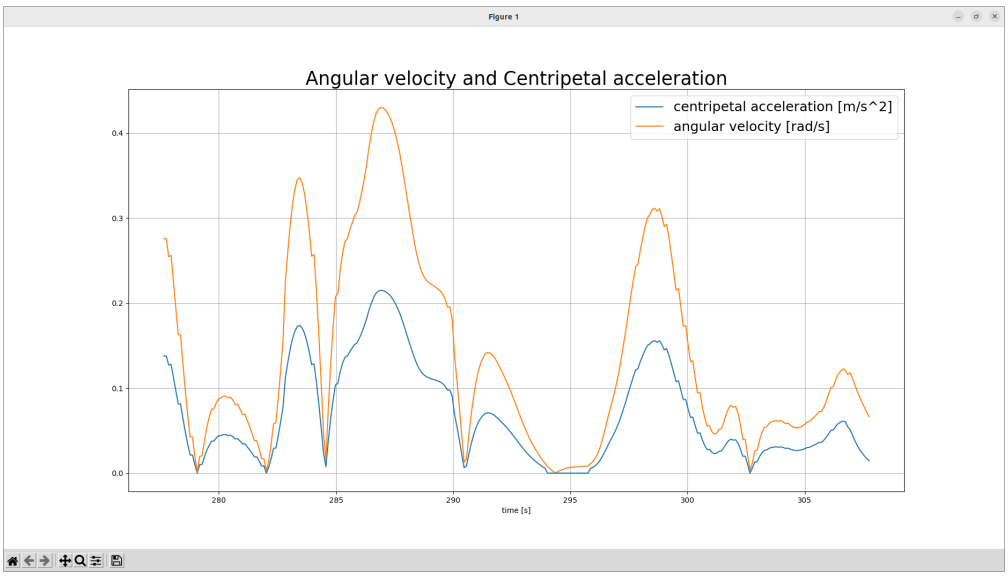


Figure 4.22: Angular velocity and Centripetal acceleration of RPP in the Static Obstacles simulation.

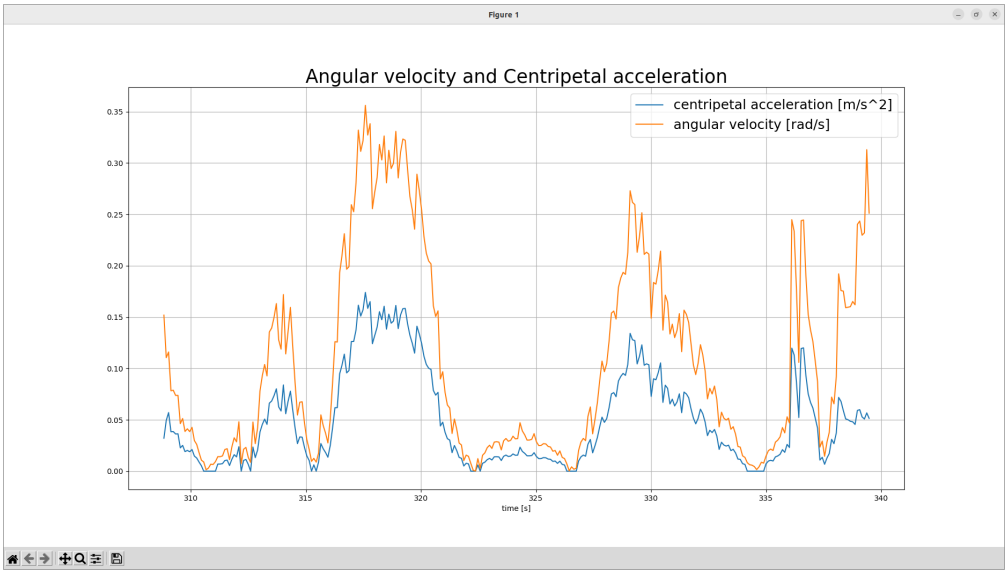


Figure 4.23: Angular velocity and Centripetal acceleration of TEB in the Static Obstacles simulation.

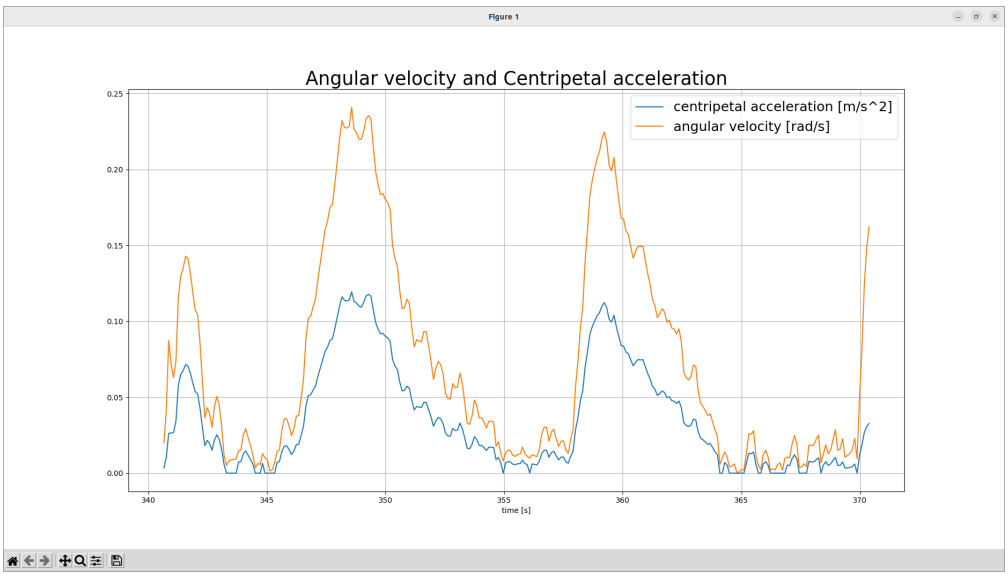


Figure 4.24: Angular velocity and Centripetal acceleration of MPPI in the Static Obstacles simulation.

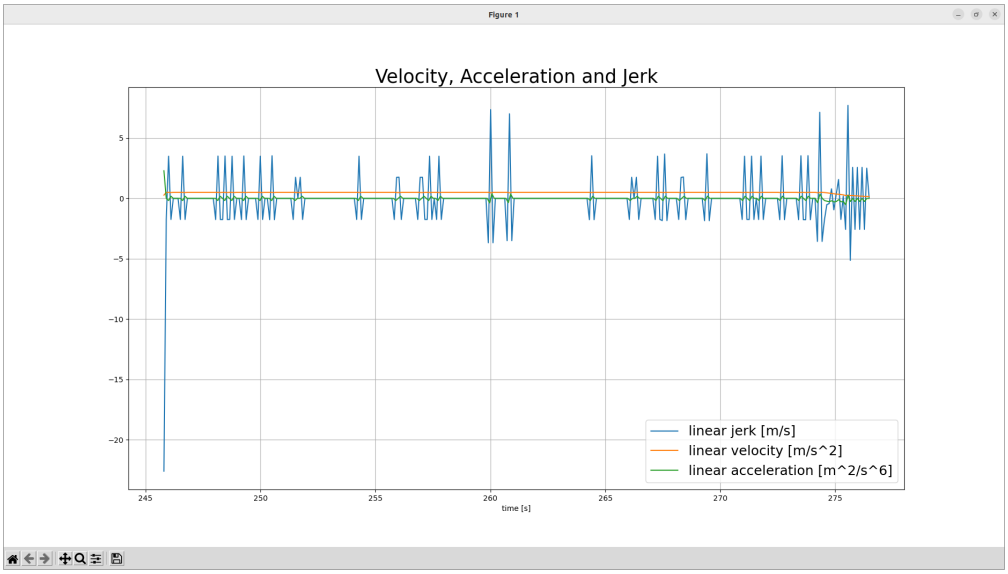


Figure 4.25: Velocity, Acceleration and jerk of DWB in the Static Obstacles simulation.

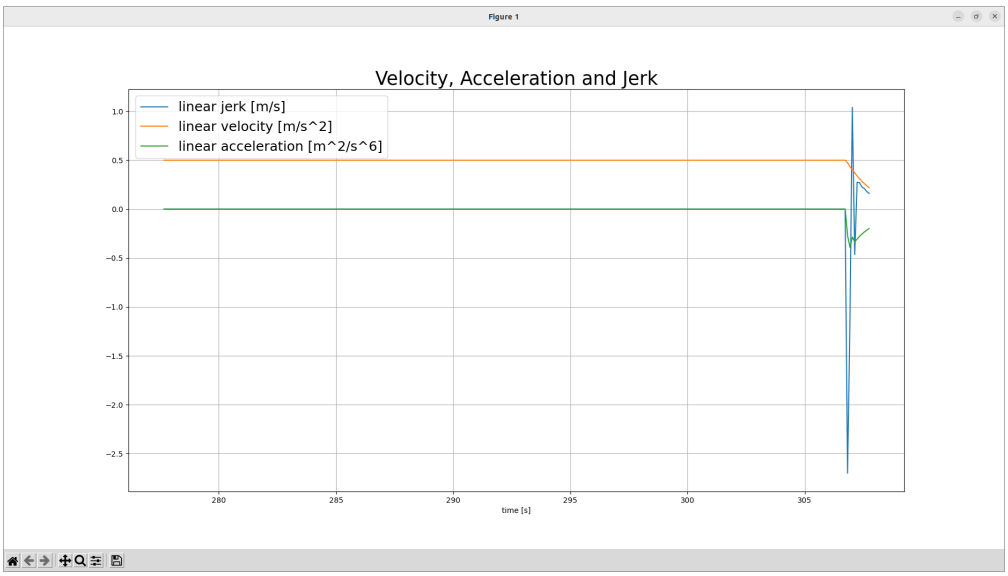


Figure 4.26: Velocity, Acceleration and jerk of RPP in the Static Obstacles simulation.

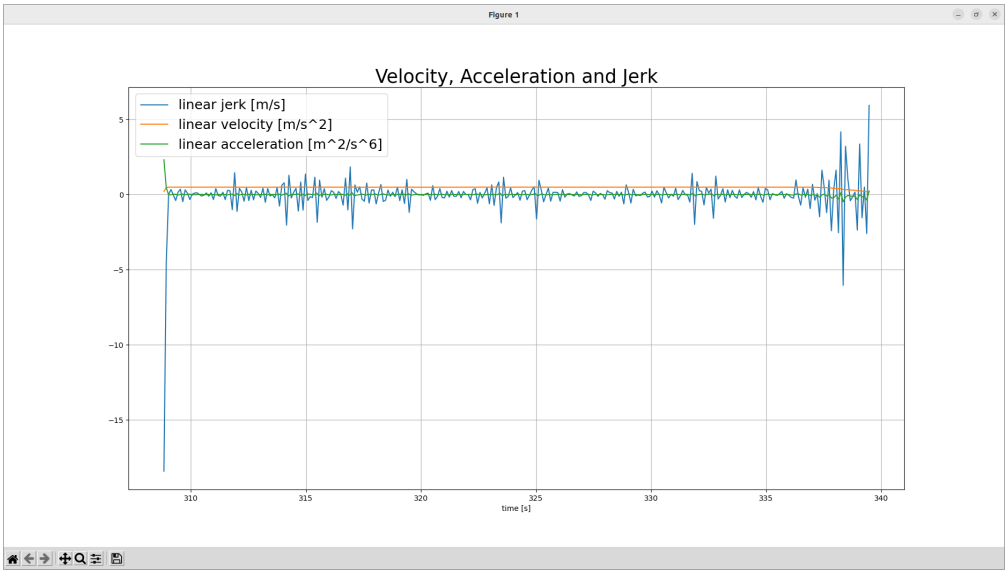


Figure 4.27: Velocity, Acceleration and jerk of TEB in the Static Obstacles simulation.

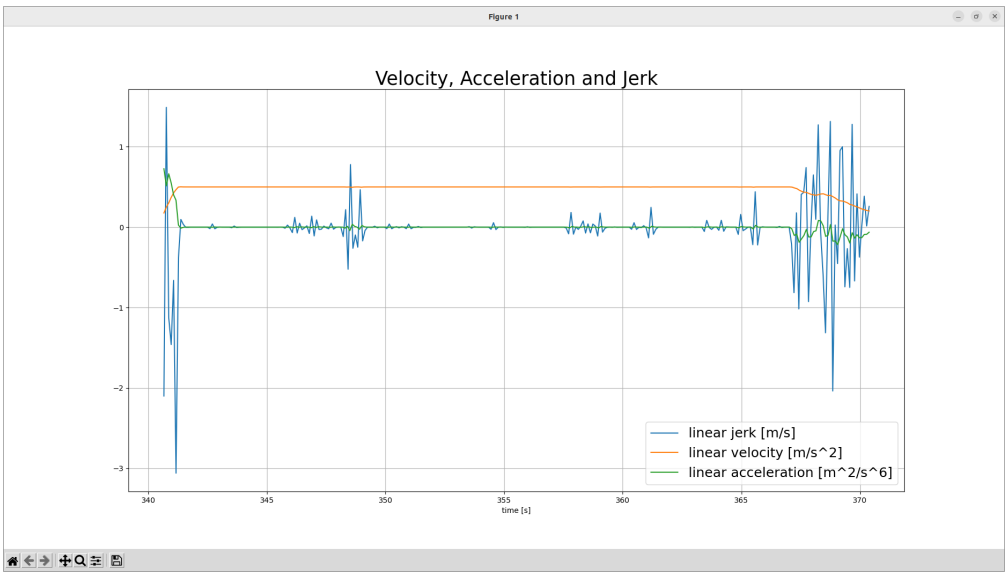


Figure 4.28: Velocity, Acceleration and jerk of MPPI in the Static Obstacles simulation.

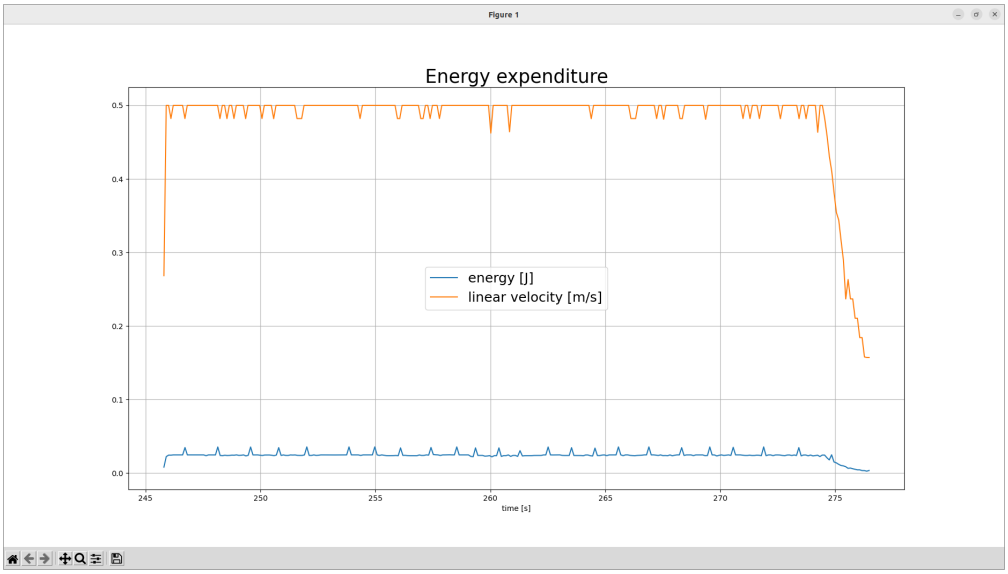


Figure 4.29: Energy expenditure and velocity of DWB in the Static Obstacles simulation.

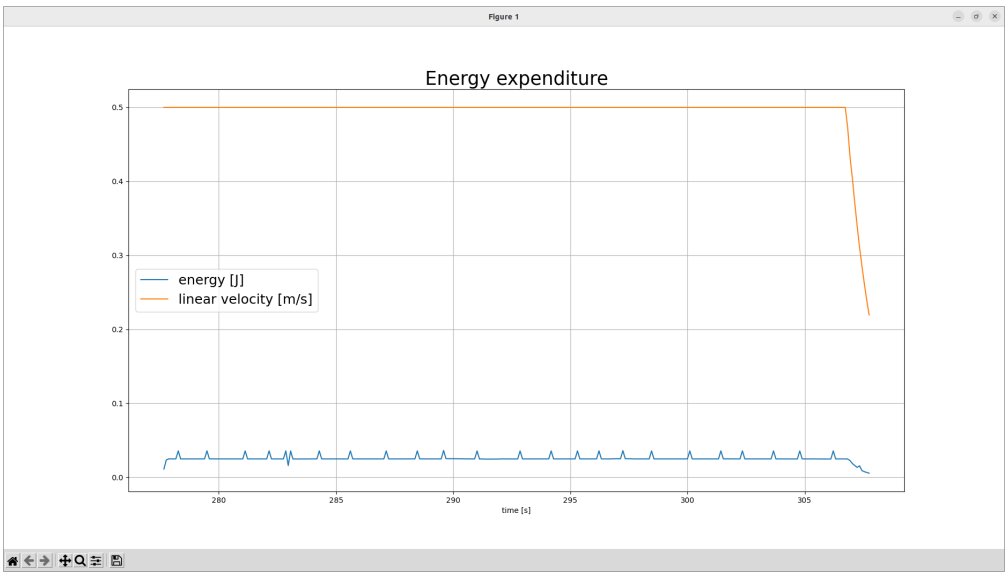


Figure 4.30: Energy expenditure and velocity of RPP in the Static Obstacles simulation.

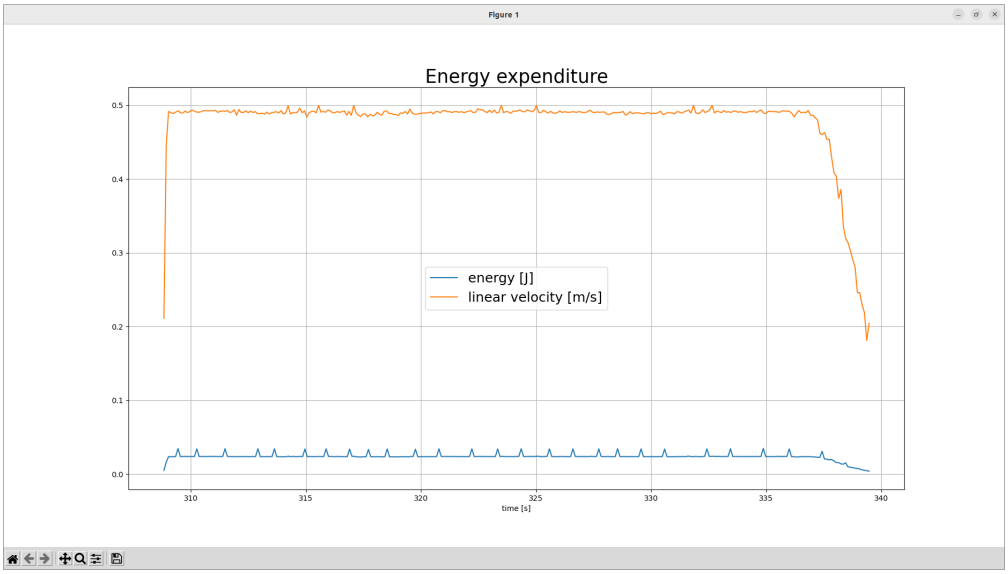


Figure 4.31: Energy expenditure and velocity of TEB in the Static Obstacles simulation.

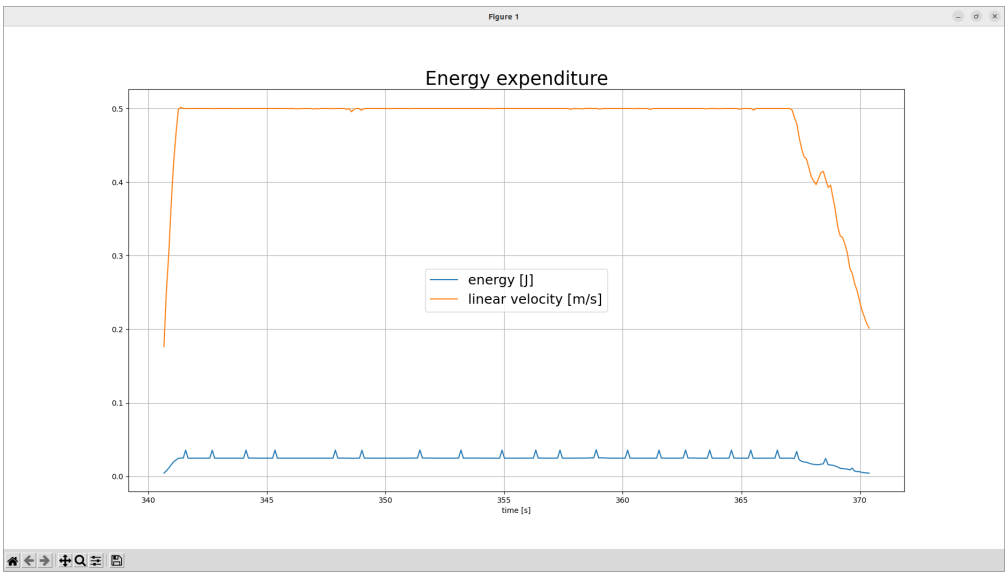


Figure 4.32: Energy expenditure and velocity of MPPI in the Static Obstacles simulation.

4.2.1 Observations

It is possible to notice that all of the controllers taken into account have completed the task successfully. The time taken to complete the tasks and the length of the navigated path are quite similar between one controller and the others, because on one hand DWB and RPP controllers are faster but navigate longer paths, while on the other hand TEB and MPPI are slower but cut corners in such a way that the navigated path is smaller. Moreover, the minimum distances from obstacles are similar as well, because of the presence of a narrow passage (Figure 4.33).

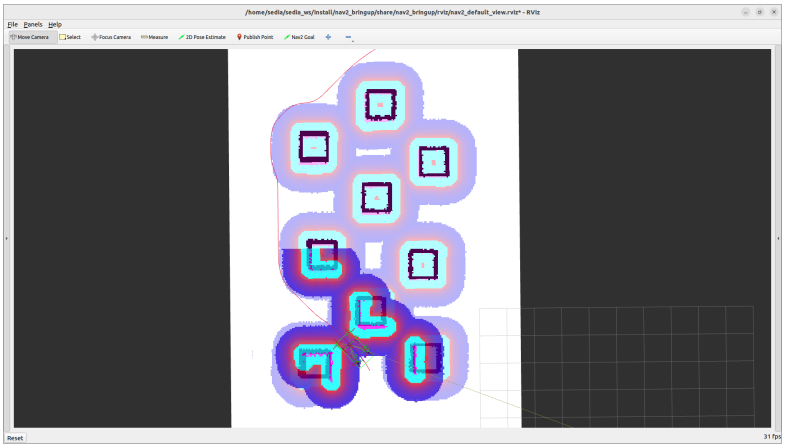


Figure 4.33: Narrow passage in the Static Obstacles simulation.

The difference between the local and the global planned path is noticeably higher in the MPPI controller case, because it cuts the corners more than all the others (Figure 4.20), navigating dangerously close to the obstacles. This does not allow a fully smooth navigation.

Nonetheless, the DWB controller seems to be the less smooth above all the controllers taken into account: as shown in Figure 4.21, it has the highest angular velocity (0.55 rad/s), thus the highest centripetal acceleration, and the highest peaks of linear jerk (Figure 4.25).

The TEB and MPPI have the lowest energy consumption, because they decrease their speed more gradually than the others.

In conclusion, the **TEB**, despite its high maximum local planned path error, is the controller that better deal with this specific situation. The RPP should not be taken into account because of its high centripetal acceleration.

4.3 One Obstacle Simulation

Figure 4.34 and 4.35 show the robot in the simulation environment.

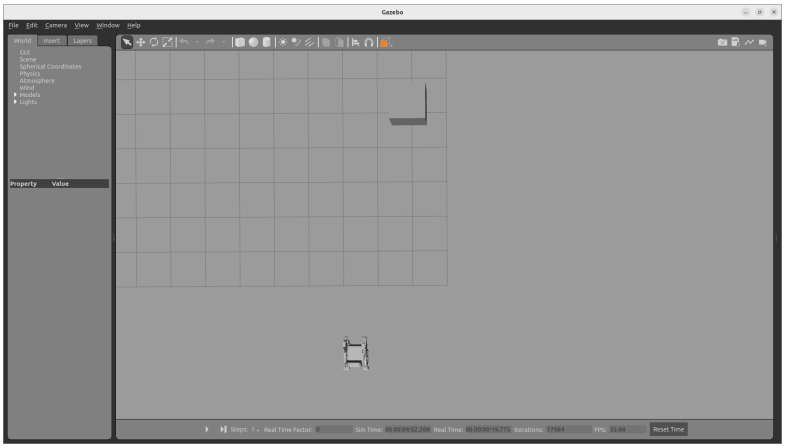


Figure 4.34: One Obstacle simulation environment.

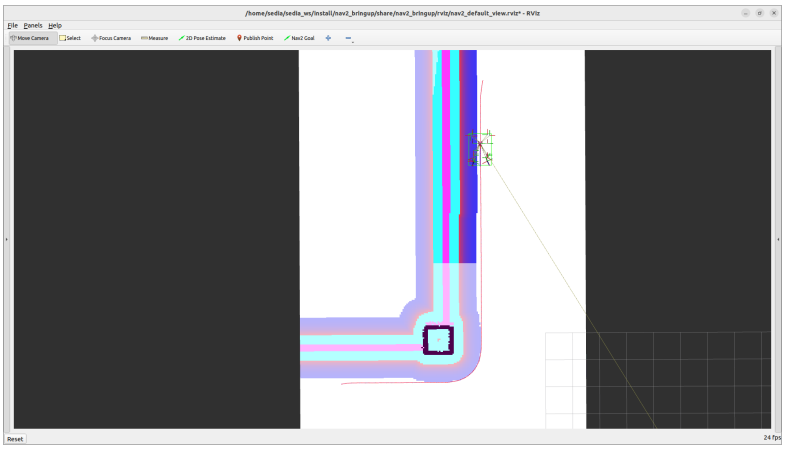


Figure 4.35: One Obstacle simulation map.

The obtained results are shown in Table 4.3.

Controller	**DWB**	**RPP**	**TEB**	**MPPI**
Success rate	100.00%	100.00%	100.00%	100.00%
Avg linear speed (m/s)	0.478/0.5	0.495/0.5	0.479/0.5	0.471/0.5
Avg path length (m)	15.614	15.612	15.42	15.247
Avg time taken (s)	32.685	31.553	32.103	32.288
Min distance from obstacle (m)	0.909	0.916	0.734	0.604
Std distance from obstacle (m)	0.077	0.087	0.188	0.169
Avg integrated x jerk (m^2/s^6)	2.776	0.251	1.285	0.641
Max centripetal acceleration (m/s^2)	0.242	0.235	0.198	0.185
Max velocity (m/s)	100.00%	100.00%	100.00%	100.00%
Max local planned path error (m)	0.07	0.056	0.254	0.377
Min local planned path error (m)	0.00	0.00	0.00	0.00
Avg local planned path error (m)	0.028	0.019	0.088	0.112
Avg heading error ($°$)	2.083	1.634	5.959	5.777
Std of heading error ($°$)	0.043	0.052	0.224	0.098
Energy Consumption (J)	7.608	7.807	7.517	7.424

Table 4.3: Results of the One Obstacle simulation

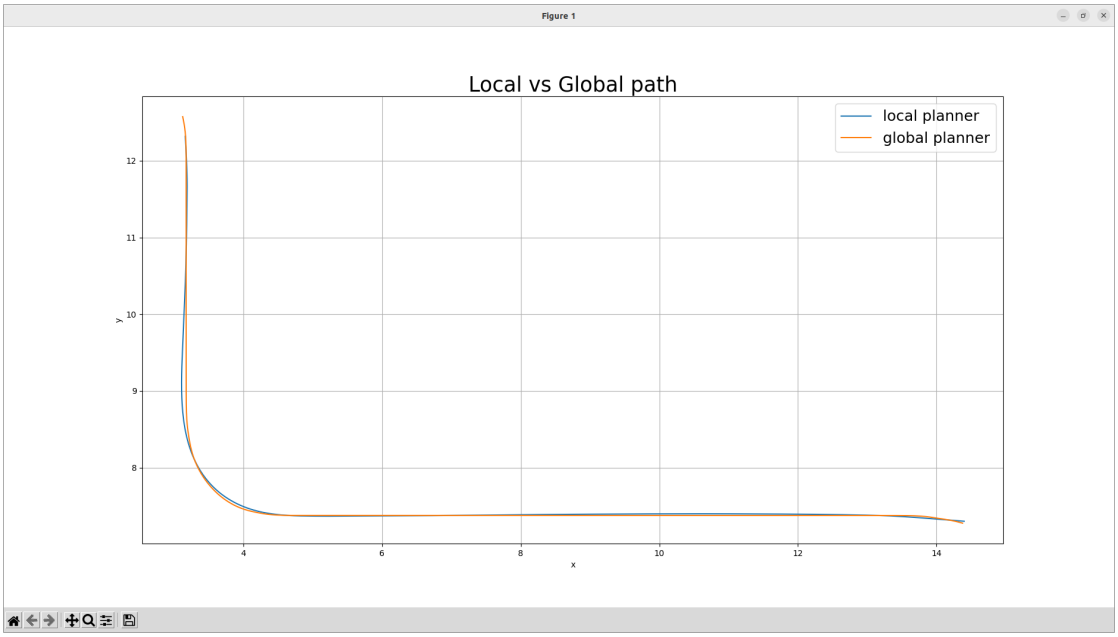


Figure 4.36: Difference between global planner and DWB in One Obstacle simulation.

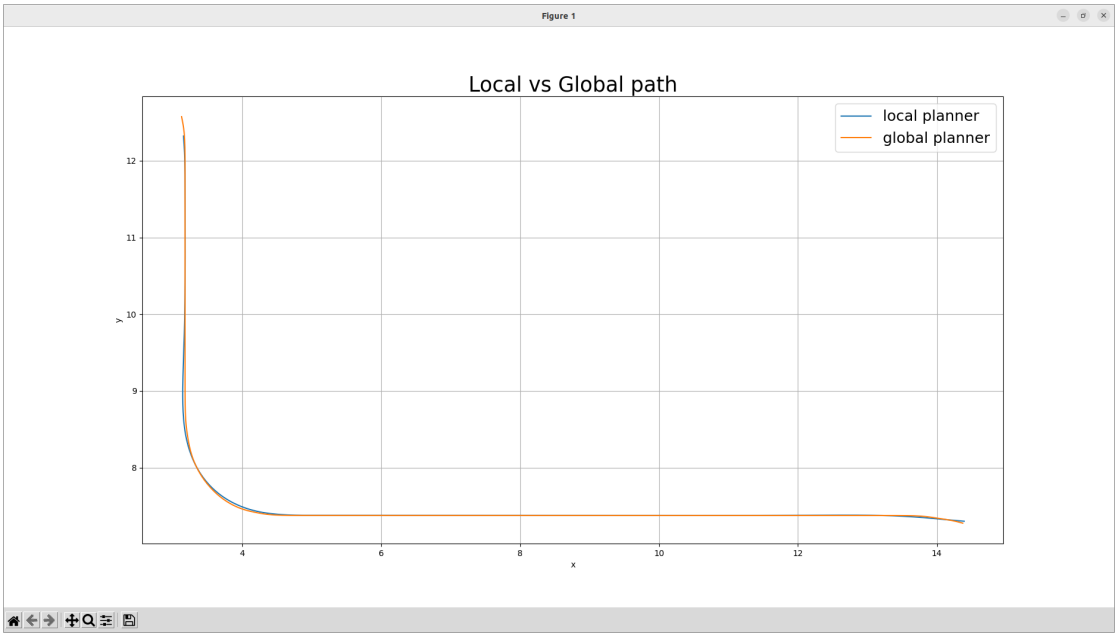


Figure 4.37: Difference between global planner and RPP in One Obstacle simulation.

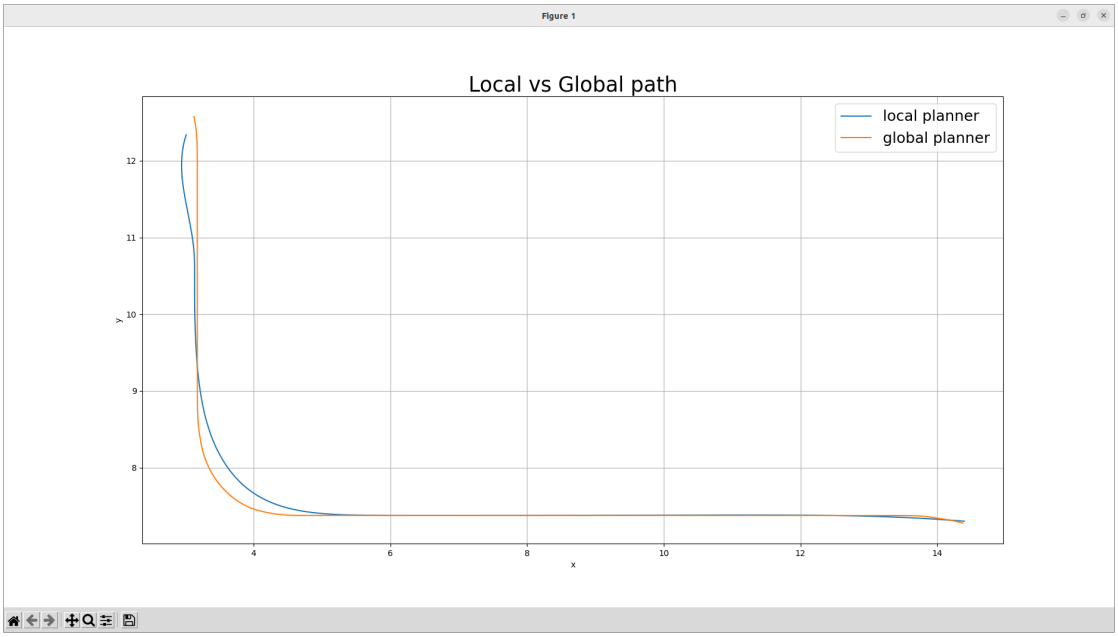


Figure 4.38: Difference between global planner and TEB in One Obstacle simulation.

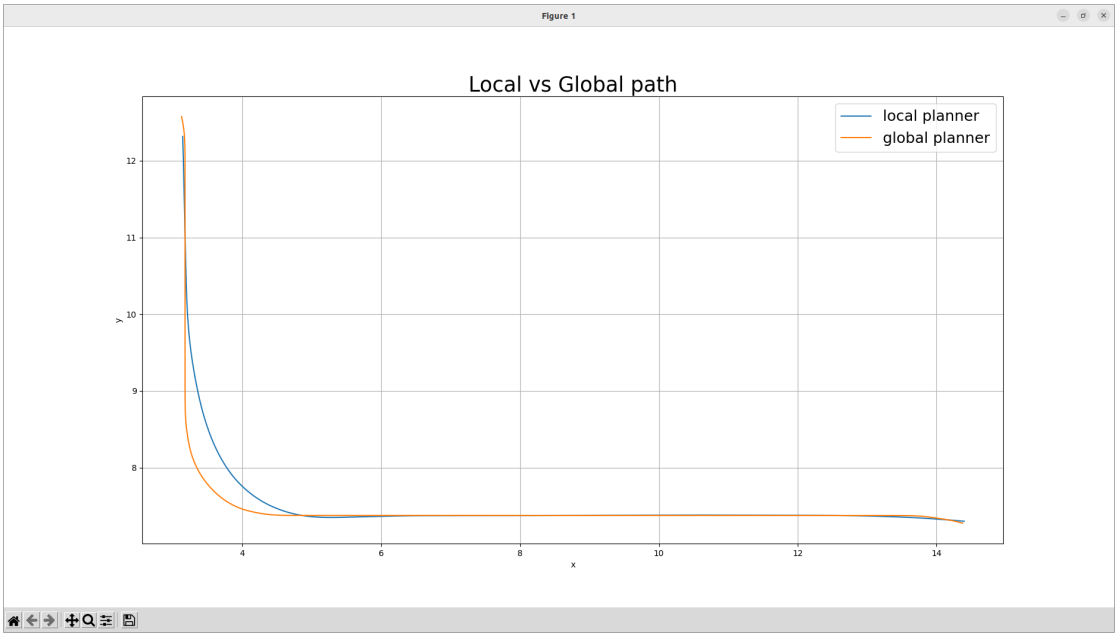


Figure 4.39: Difference between global planner and MPPI in One Obstacle simulation.

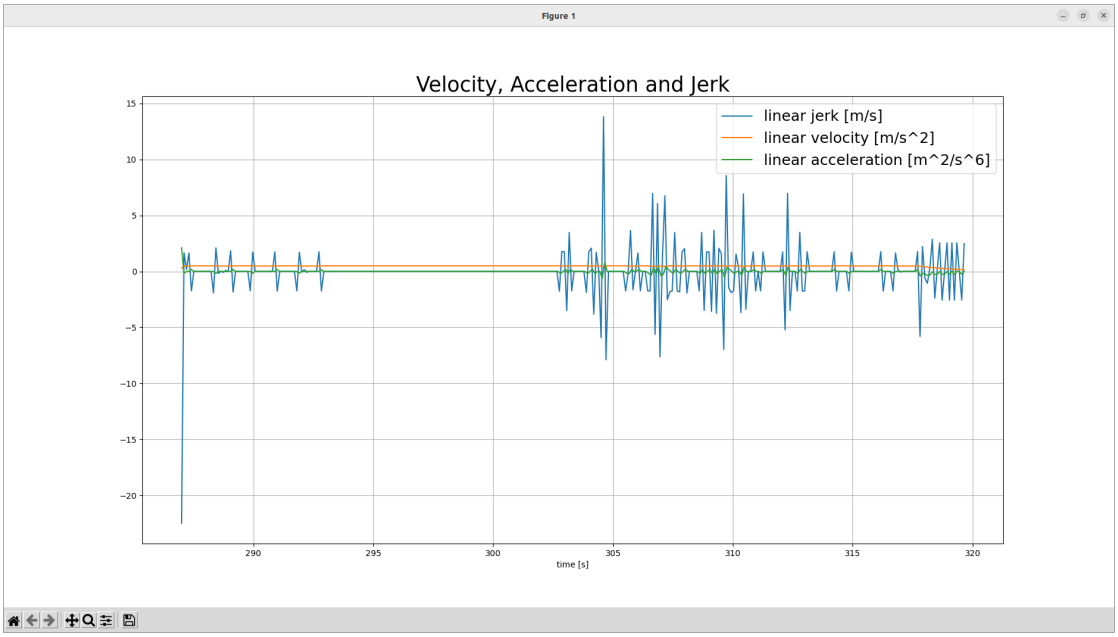


Figure 4.40: Velocity, Acceleration and Jerk of DWB in the One Obstacle simulation.

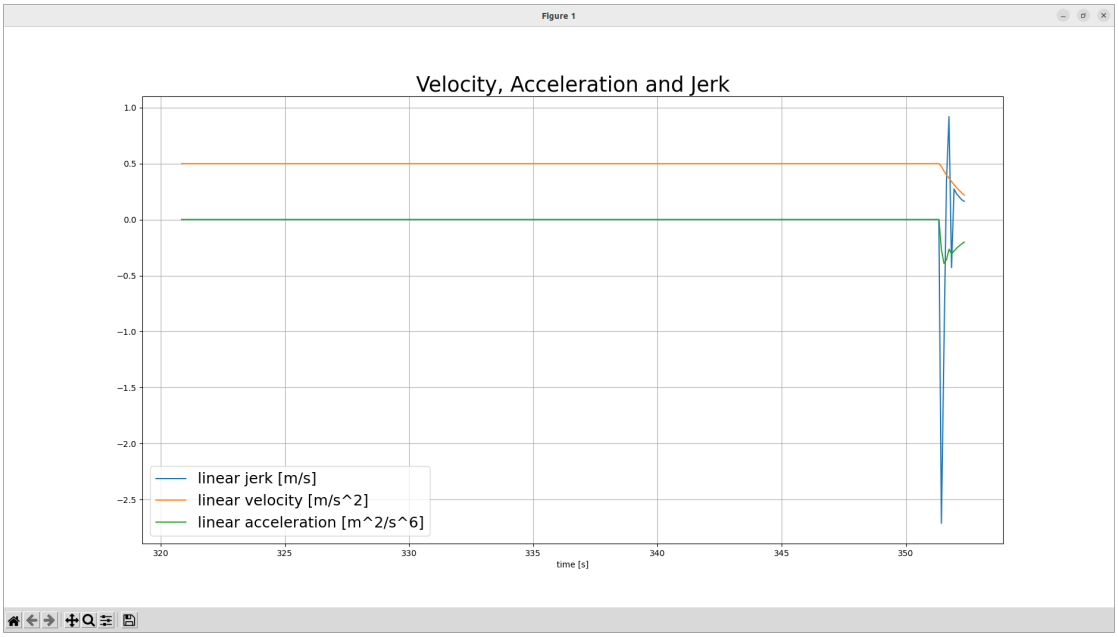


Figure 4.41: Velocity, Acceleration and Jerk of RPP in the One Obstacle simulation.

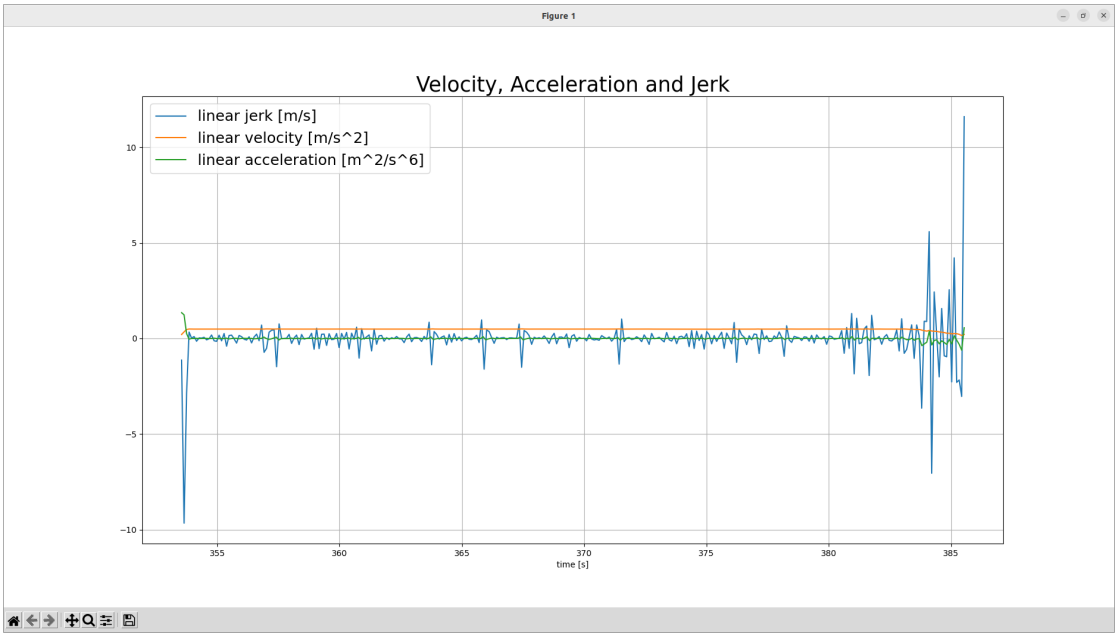


Figure 4.42: Velocity, Acceleration and Jerk of TEB in the One Obstacle simulation.

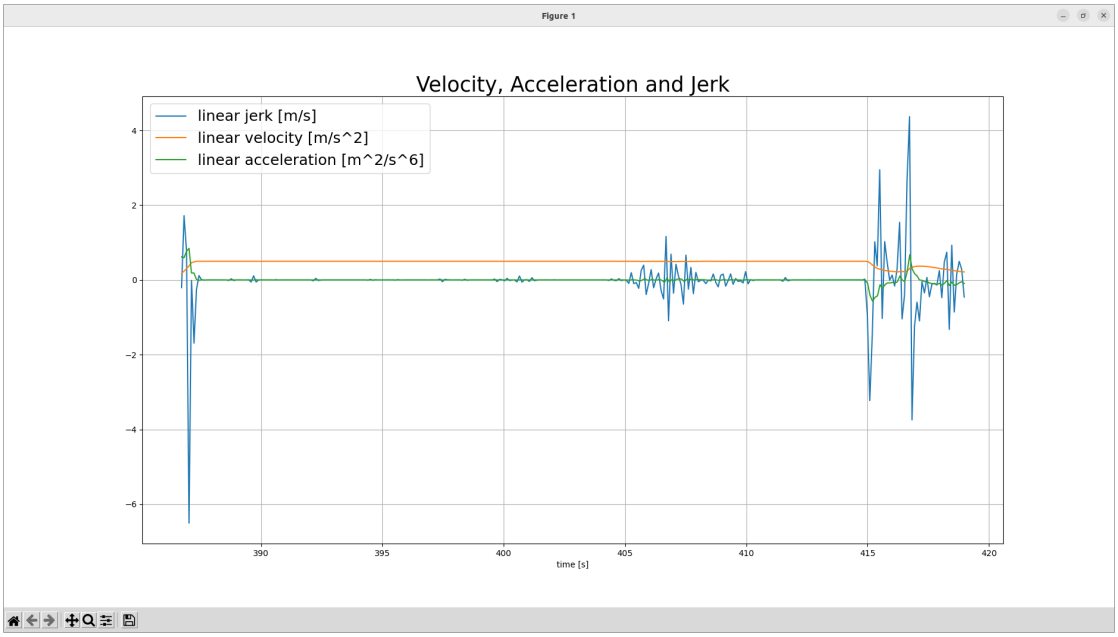


Figure 4.43: Velocity, Acceleration and Jerk of MPPI in the One Obstacle simulation.

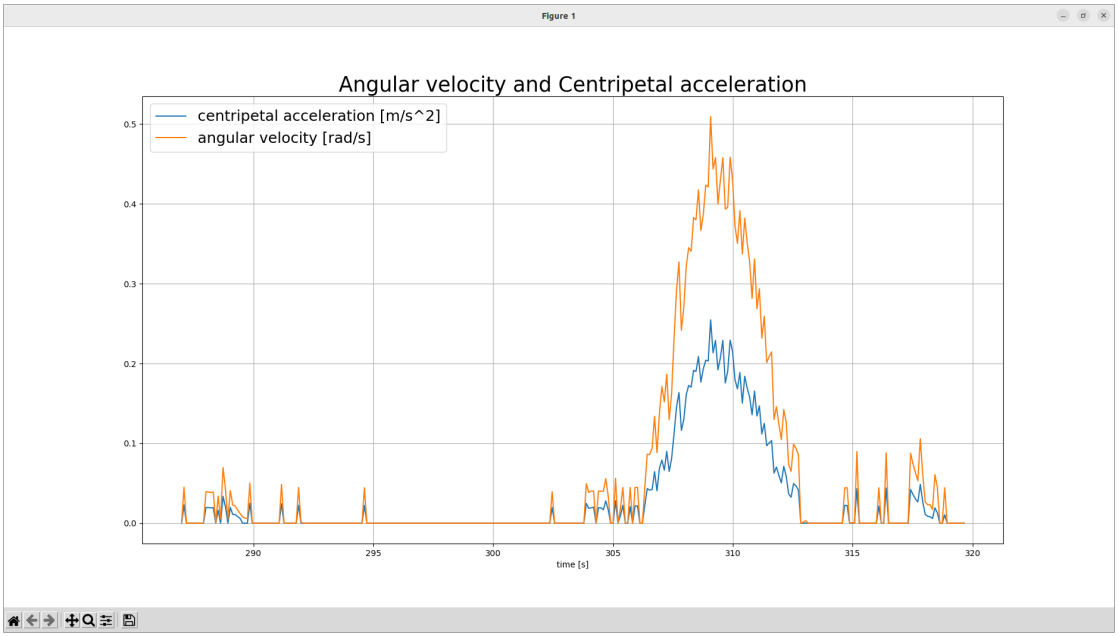


Figure 4.44: Angular velocity and Centripetal acceleration of RPP in the One Obstacle simulation.

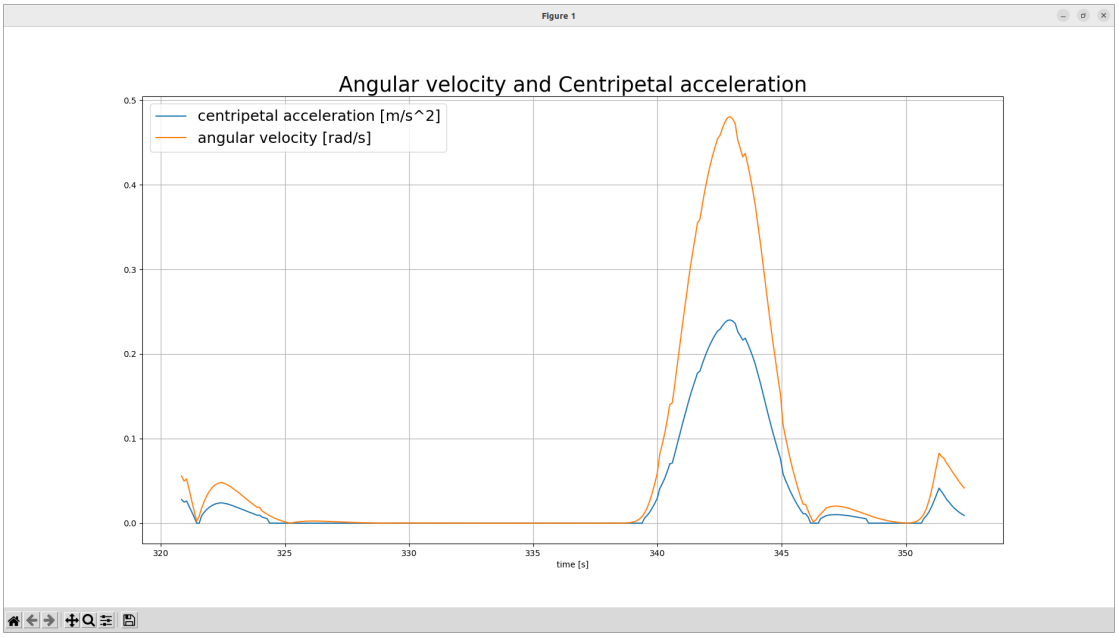


Figure 4.45: Angular velocity and Centripetal acceleration of RPP in the One Obstacle simulation.

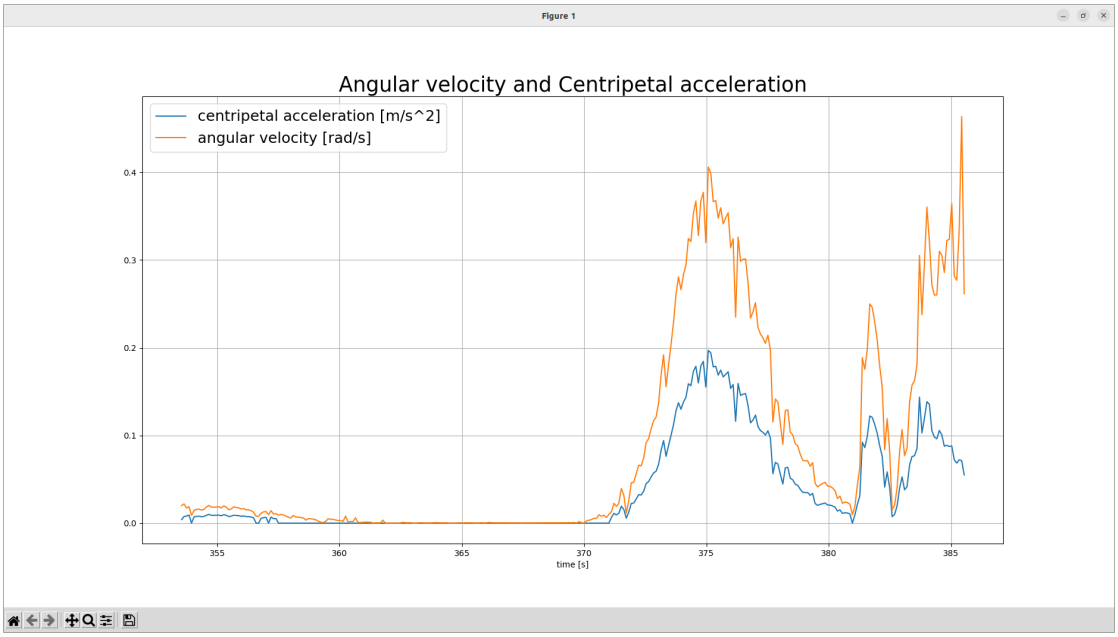


Figure 4.46: Angular velocity and Centripetal acceleration of TEB in the One Obstacle simulation.

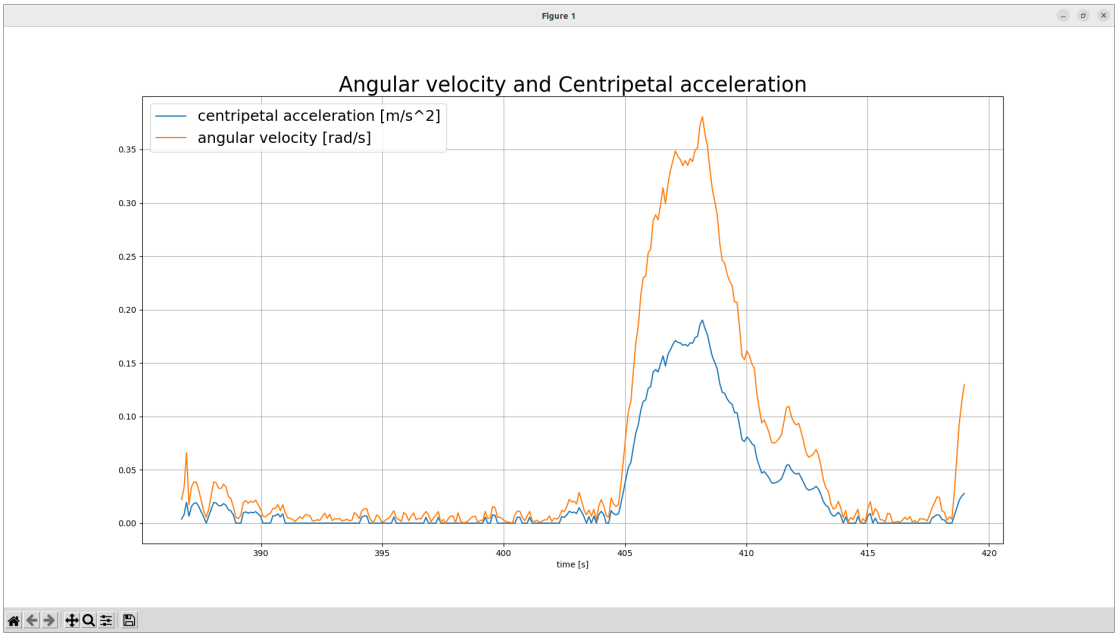


Figure 4.47: Angular velocity and Centripetal acceleration of MPPI in the One Obstacle simulation.

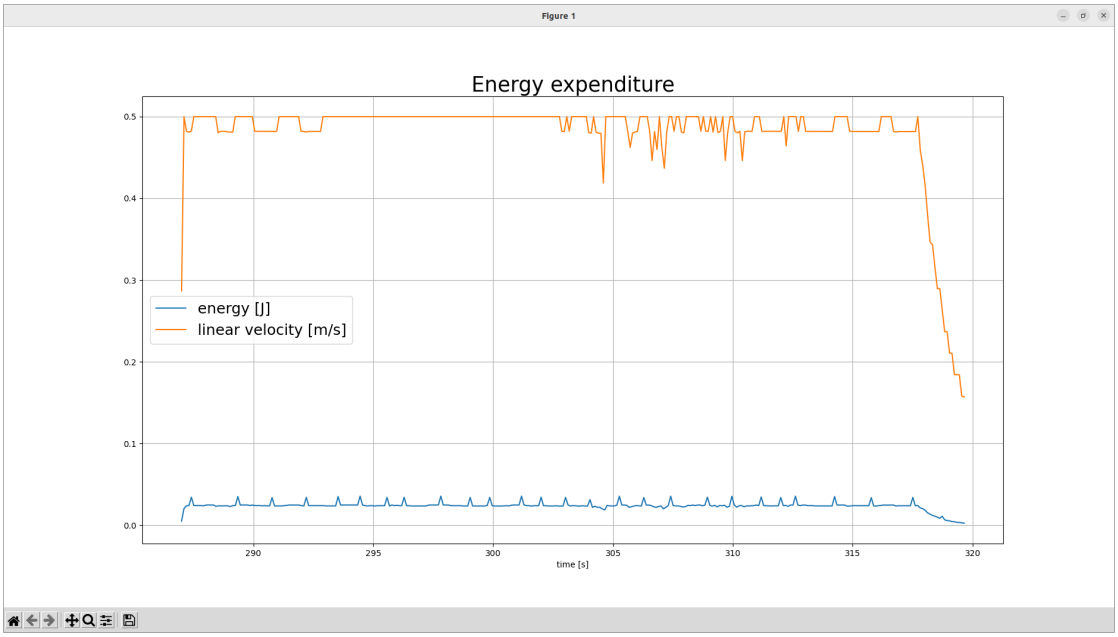


Figure 4.48: Velocity and energy expenditure of DWB in the One Obstacle simulation.

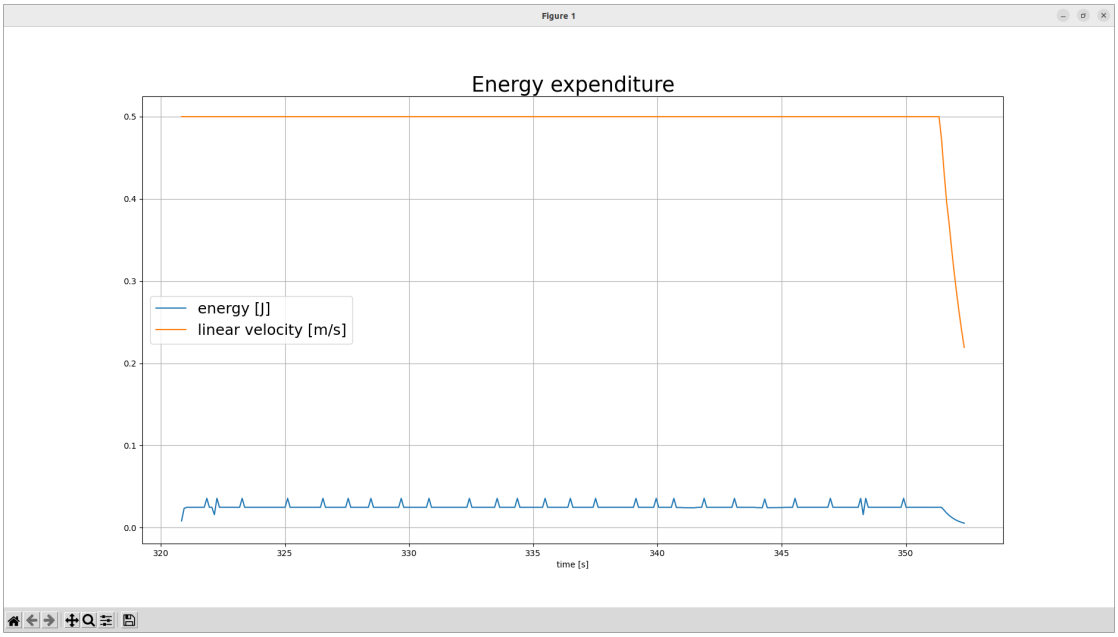


Figure 4.49: Velocity and energy expenditure of RPP in the One Obstacle simulation.

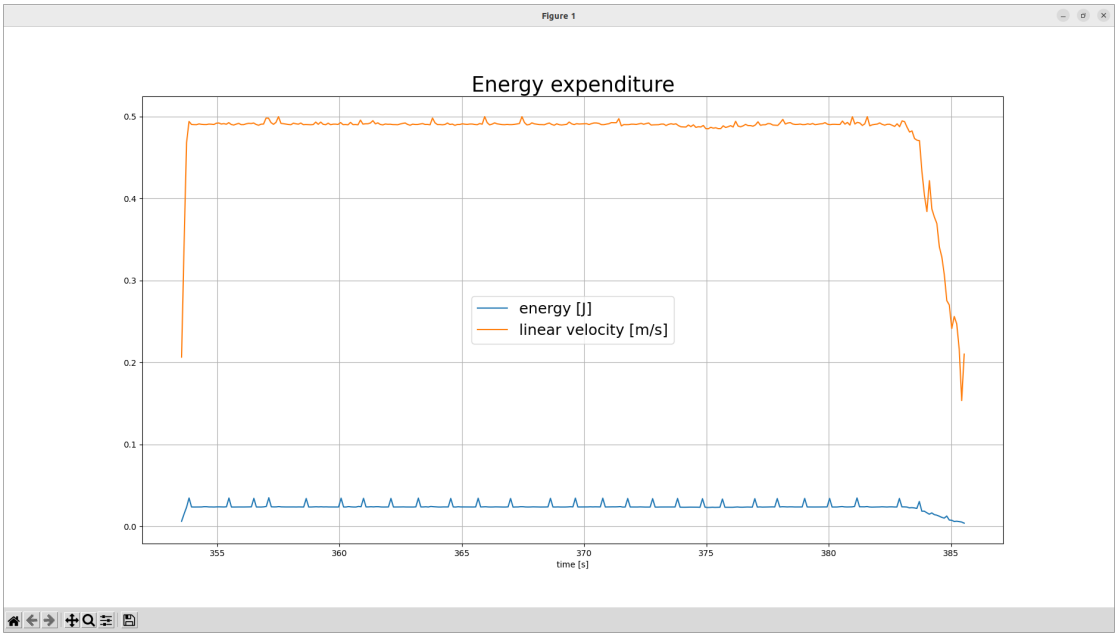


Figure 4.50: Velocity and energy expenditure of TEB in the One Obstacle simulation.

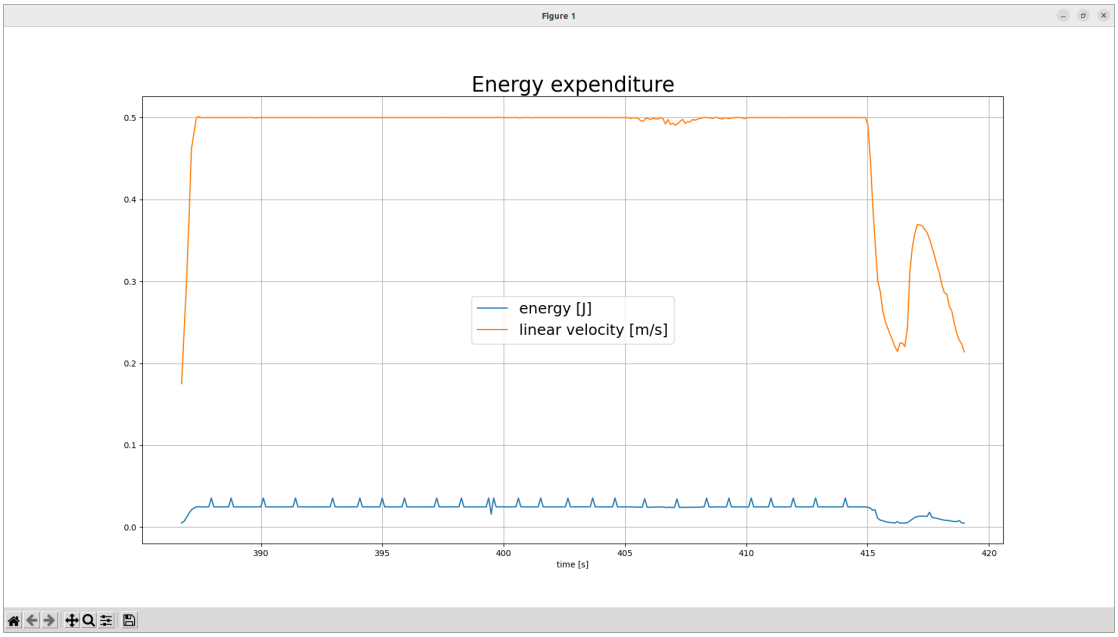


Figure 4.51: Velocity and energy expenditure of MPPI in the One Obstacle simulation.

4.3.1 Observations

In Figure 4.35 it is possible to observe the presence of the *Keepout Filter*. It has been used in order to better customize the chosen path that the robot had to navigate, in particular it allowed the definition of a perfectly linear path.

All of the controllers have successfully completed the tasks at almost same speed and in almost the same time: even though the DWB and the RPP have higher speed and better follows the defined global planned path, the TEB and MPPI have lower speed but cut corners, navigating smaller paths.
For what concerns the energy consumption, which is almost the same for each controller, the TEB and the MPPI controllers have the lowest value.

However, one of the parameters that mostly put in evidence the difference of the controllers' behaviors is the smoothness. The MPPI, that navigates dangerously close to the obstacle, has the lowest value of minimum distance from the obstacle. DWB has the highest average integrated x jerk, because it significantly changes its acceleration in very short periods of time when the robot approaches the sharp curve on the defined path. Moreover, the DWB and the RPP controllers have the highest centripetal acceleration.

In conclusion, the **TEB** controller better manages this specific situation.

4.4 Restricted Area Simulation

4.4.1 Small Restricted area

The Figure show the robot in the simulation environment.

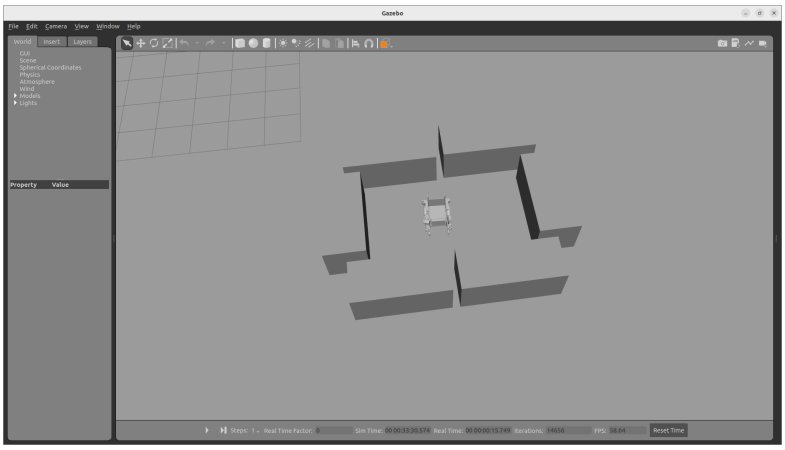


Figure 4.52: Small Restricted Area simulation environment.

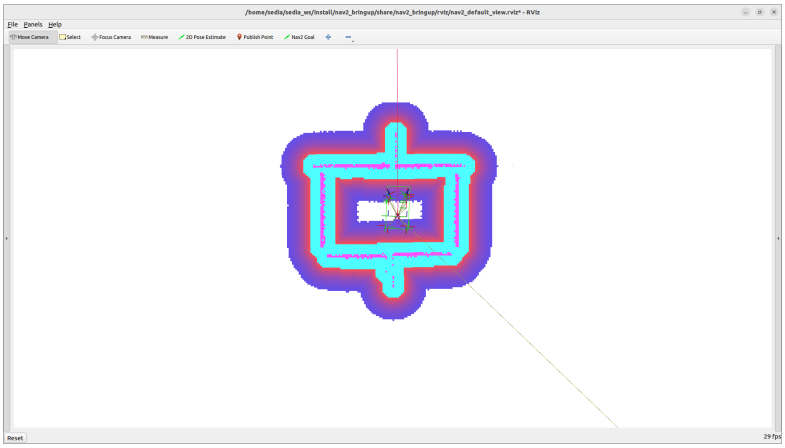


Figure 4.53: Small Restricted Area simulation map.

Table 4.4 displays the results obtained by the simulation.

Controller	**DWB**	**RPP**	**TEB**	**MPPI**
Success rate	0.00%	0.00%	0.00%	0.00%
Avg linear speed (m/s)	0.053/0.5	0.5/0.5	0.172/0.5	0.118/0.5
Avg path length (m)	0.655	0.286	0.616	0.547
Avg time taken (s)	12.462	0.597	6.151	5.358
Min distance from obstacle (m)	0.857	0.856	0.868	0.9
Std distance from obstacle (m)	0.073	0.104	0.105	0.085
Avg integrated x jerk (m^2/s^6)	3.3	0.00	12.727	5.49
Max velocity (m/s)	80.00%	100.00%	68.00%	59.00%
Max local planned path error (m)	7.923	8.294	8.547	8.266
Min local planned path error (m)	0.00	0.014	0.002	0.002
Avg heading error ($°$)	3.829	2.165	47.25	2.012
Std of heading error ($°$)	2.041	0.281	32.891	1.174
Energy Consumption (J)	0.126	0.165	0.1	0.086
Avg Time For Abortion (s)	7.596	0.00	2.456	2.384

Table 4.4: Results of the Small Restricted Area simulation.

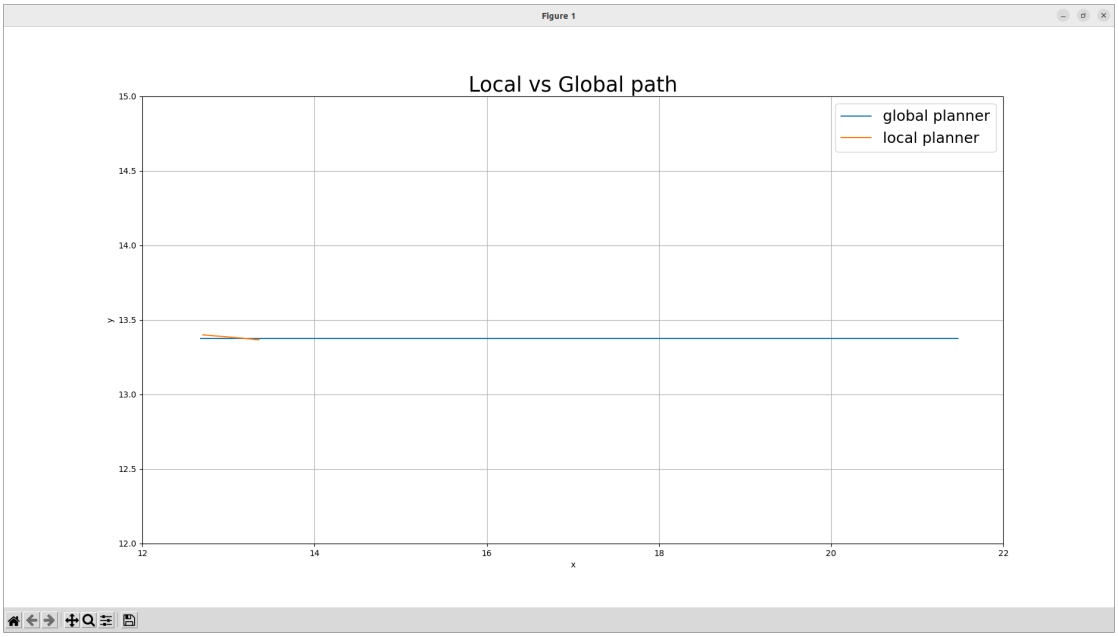


Figure 4.54: Difference between global planned path and DWB in the Small Restricted Area simulation.

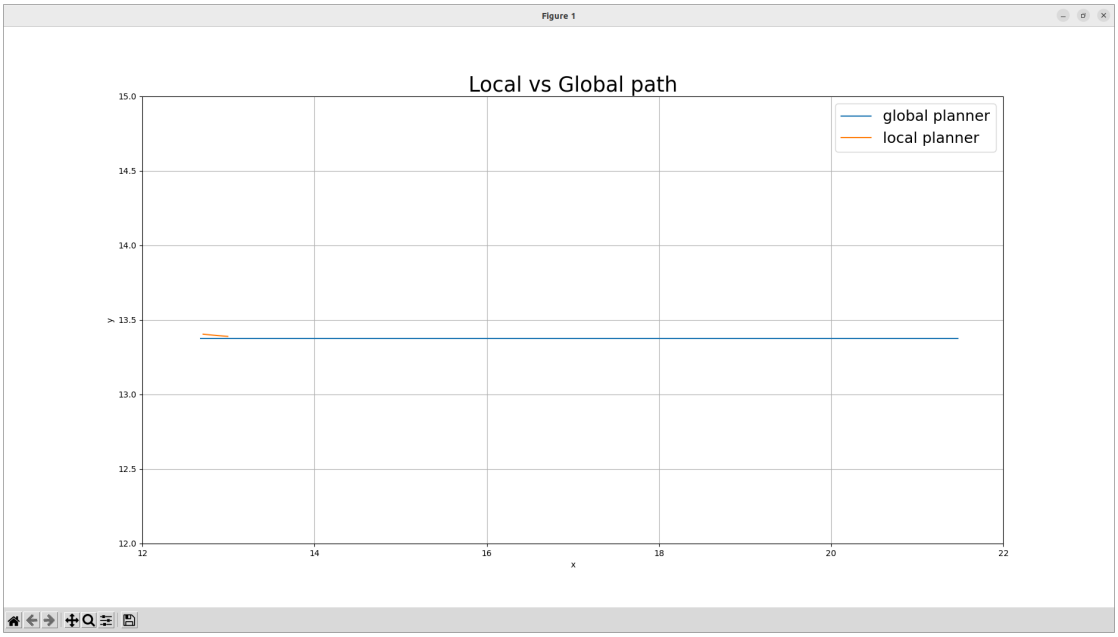


Figure 4.55: Difference between global planned path and RPP in the Small Restricted Area simulation.

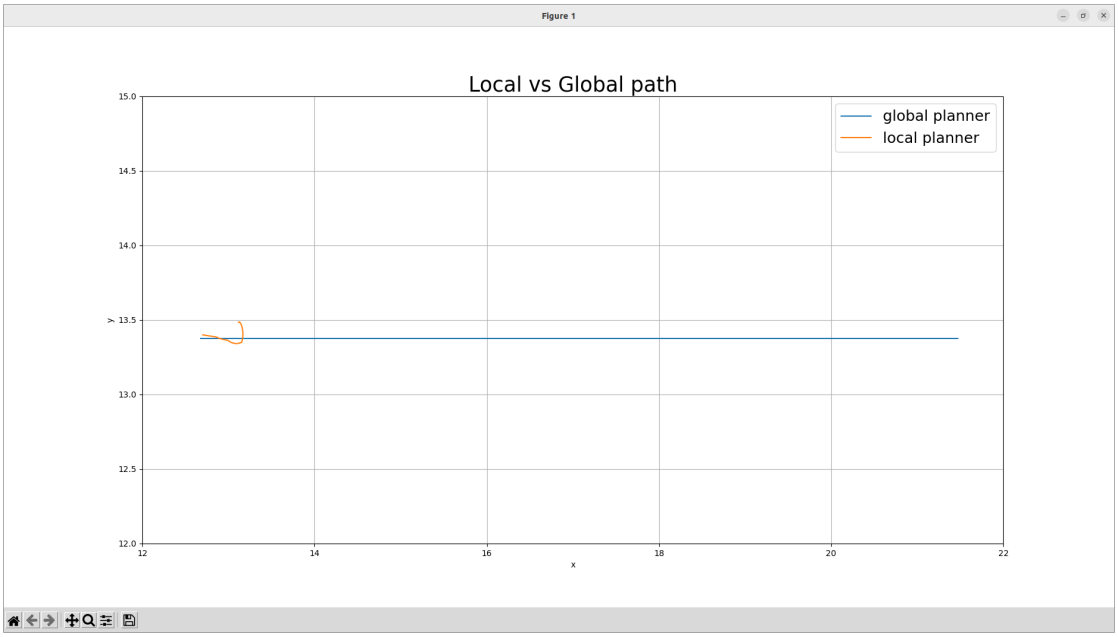


Figure 4.56: Difference between global planned path and TEB in the Small Restricted Area simulation.

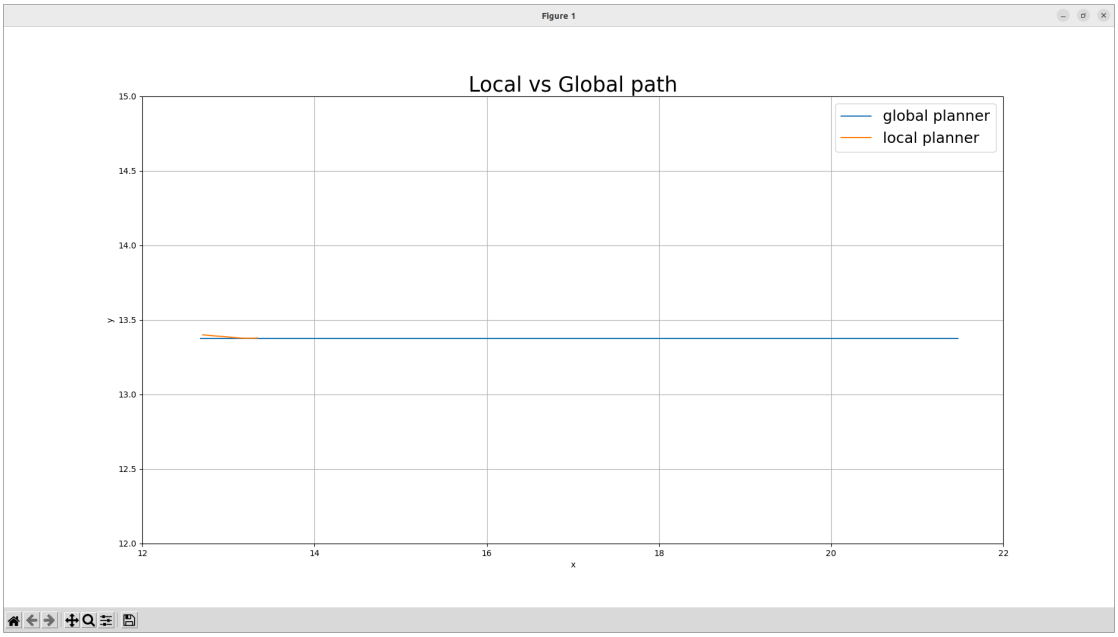


Figure 4.57: Difference between global planned path and MPPI in the Small Restricted Area simulation.

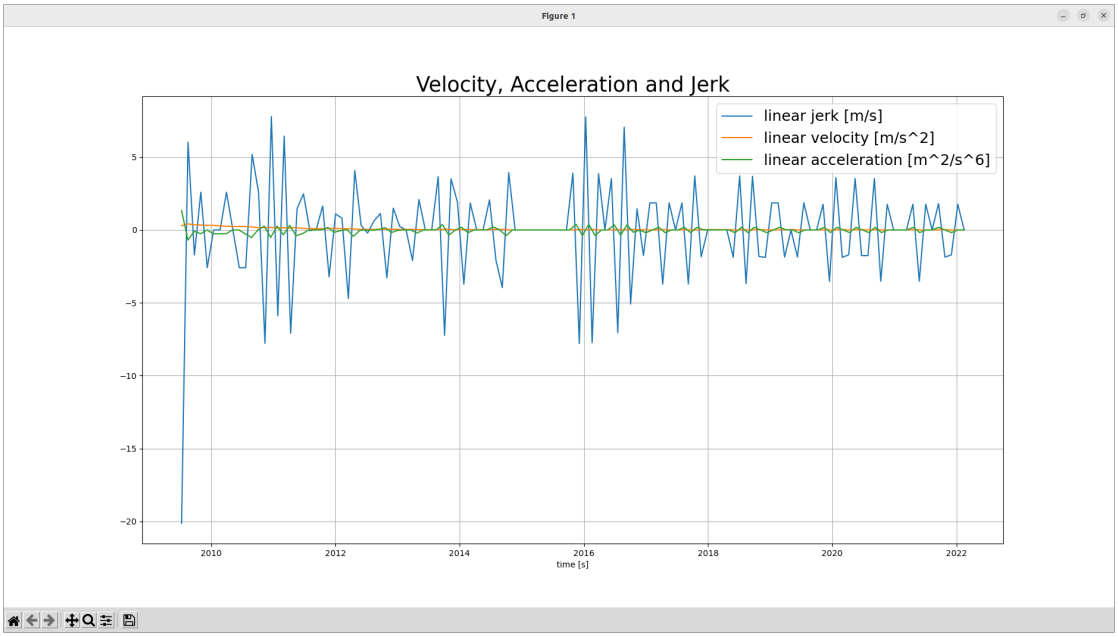


Figure 4.58: Velocity, acceleration and jerk of DWB in the Small Restricted Area simulation.

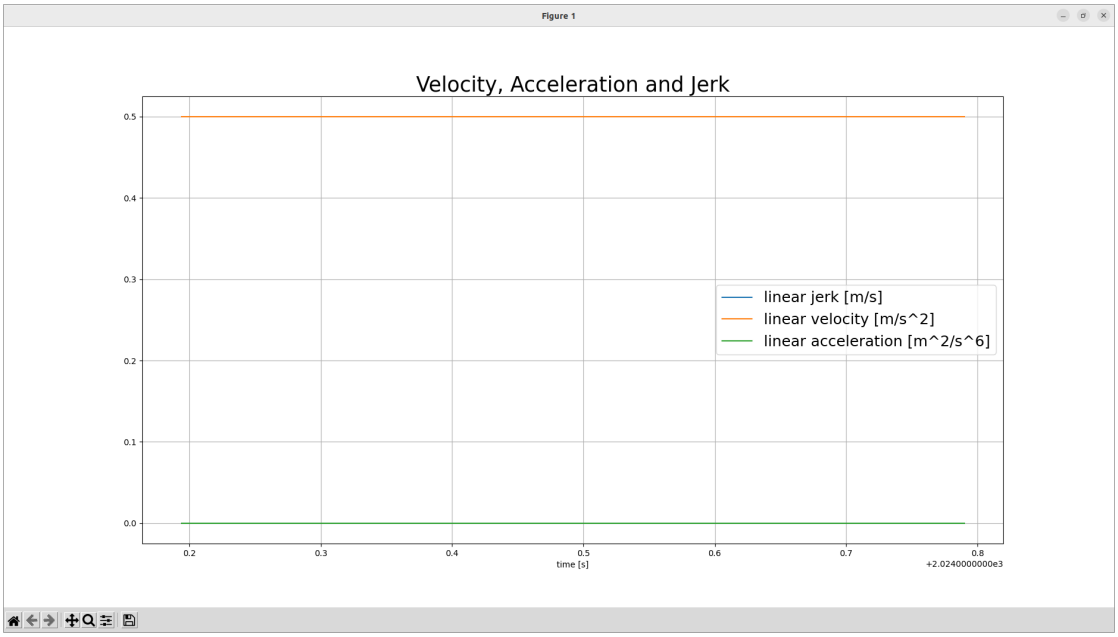


Figure 4.59: Velocity, acceleration and jerk of RPP in the Small Restricted Area simulation.

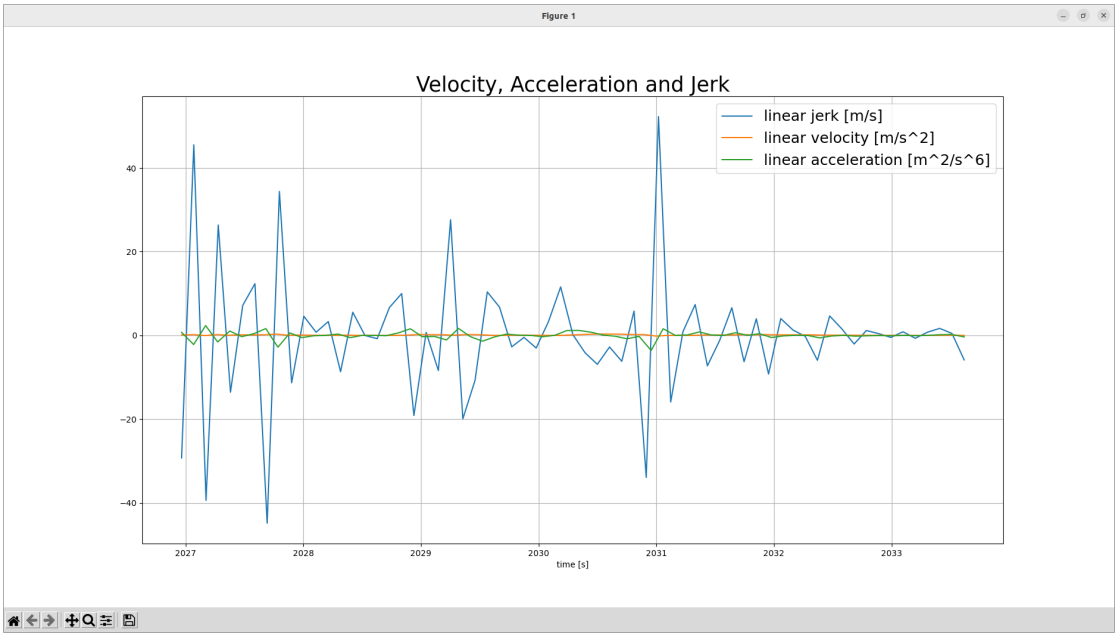


Figure 4.60: Velocity, acceleration and jerk of TEB in the Small Restricted Area simulation.

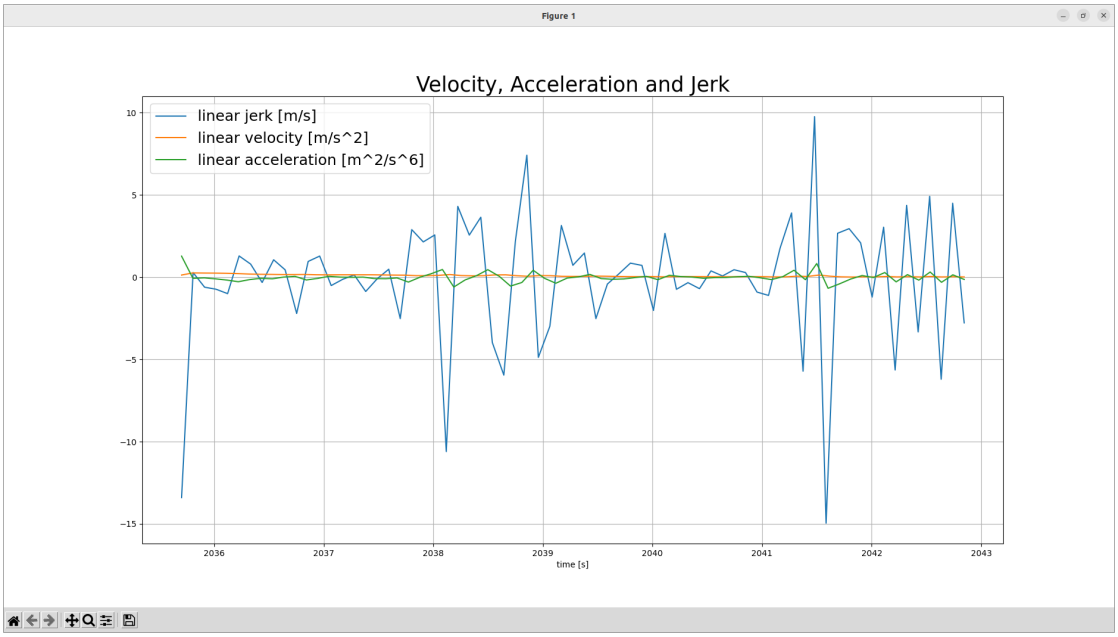


Figure 4.61: Velocity, acceleration and jerk of MPPI in the Small Restricted Area simulation.

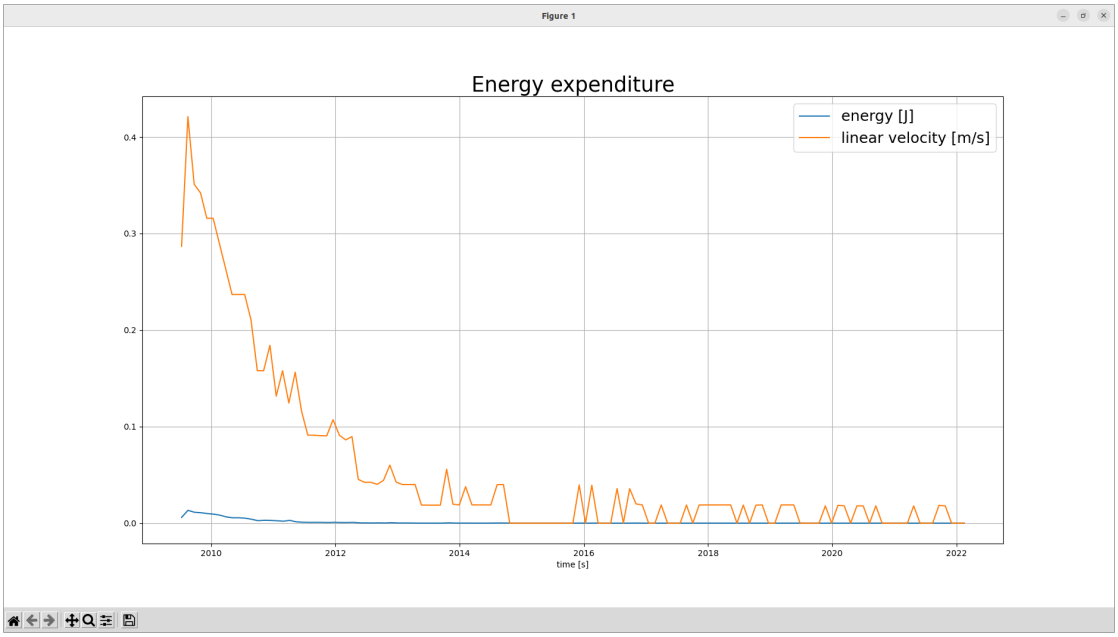


Figure 4.62: Velocity and energy expenditure of DWB in the Small Restricted Area simulation.

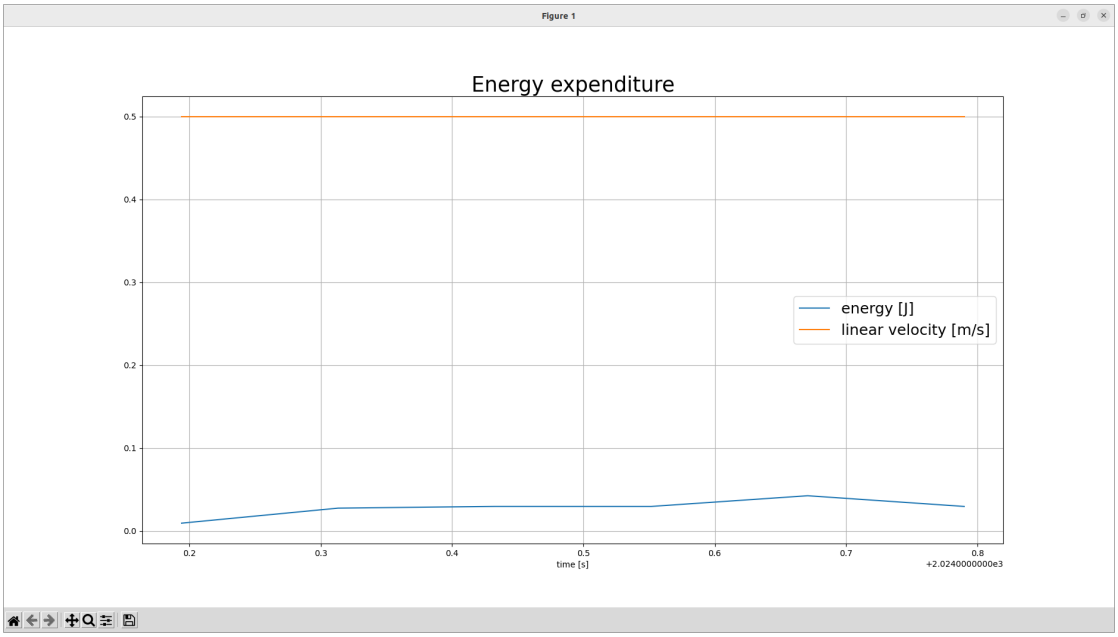


Figure 4.63: Velocity and energy expenditure of RPP in the Small Restricted Area simulation.

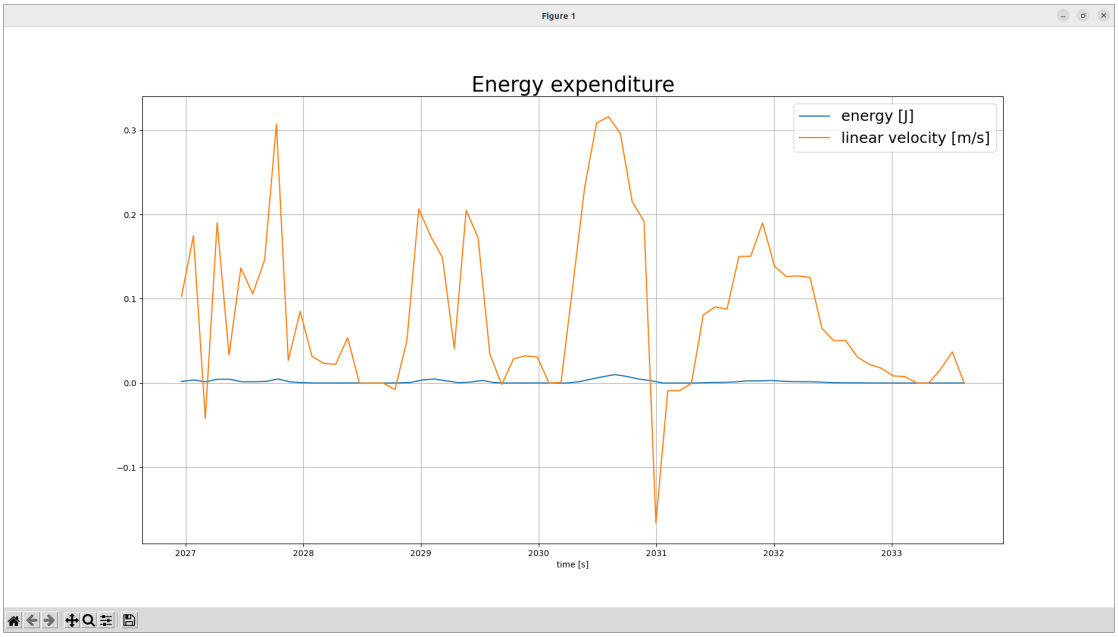


Figure 4.64: Velocity and energy expenditure of TEB in the Small Restricted Area simulation.

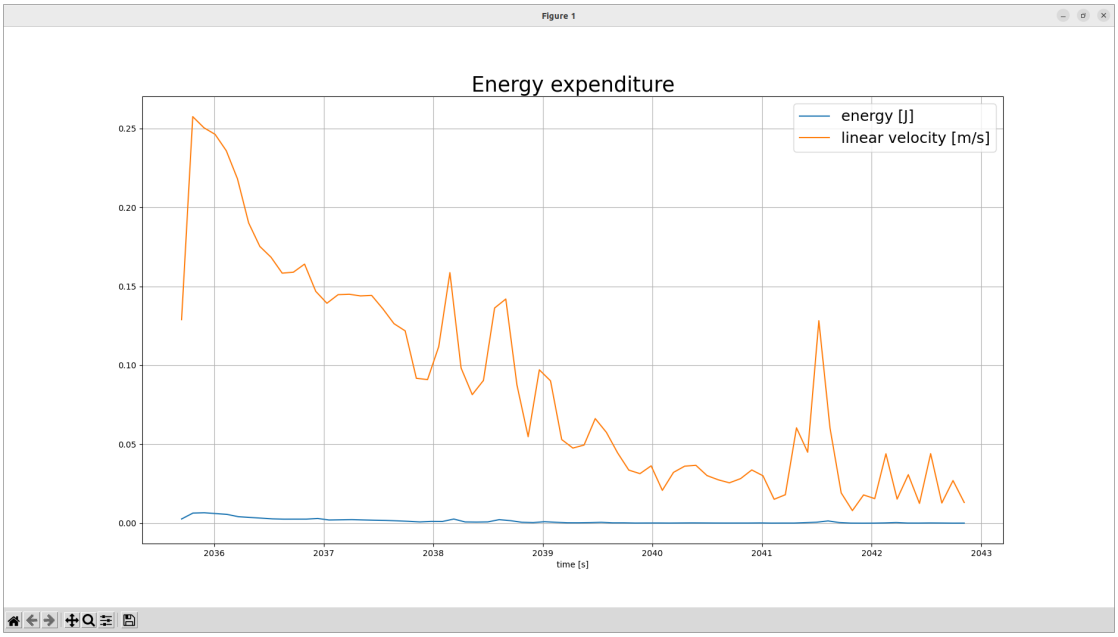


Figure 4.65: Velocity and energy expenditure of MPPI in the Small Restricted Area simulation.

4.4.2 Large Restricted Area Simulation

The Figures 4.66 and 4.67 show the robot in the simulation environment.

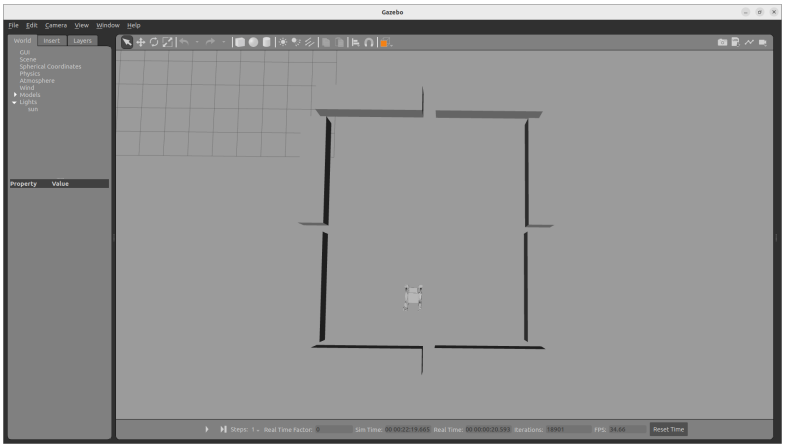


Figure 4.66: Large Restricted Area simulation environment.

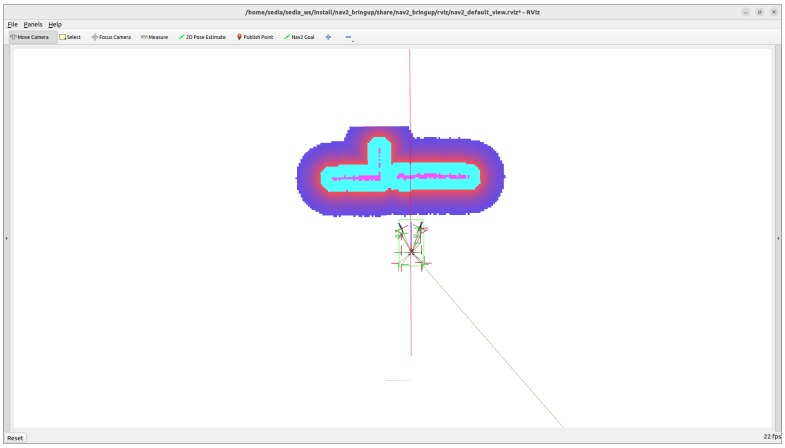


Figure 4.67: Large Restricted Area simulation map.

The results provided by the Large Restricted Area simulation are shown in Table 4.5.

Controller	**DWB**	**RPP**	**TEB**	**MPPI**
Success rate	0.00%	0.00%	0.00%	0.00%
Avg linear speed (*m/s*)	0.209/0.5	0.5/0.5	0.341/0.5	0.317/0.5
Avg path length (*m*)	3.817	3.458	3.567	3.694
Avg time taken (*s*)	18.226	6.905	11.267	11.81
Min distance from obstacle (*m*)	0.944	1.326	1.24	1.071
Std distance from obstacle (*m*)	0.507	0.47	0.495	0.538
Avg integrated x jerk (m^2/s^6)	2.446	0.00	8.181	3.272
Max velocity (*m/s*)	100.00%	100.00%	100.00%	100.00%
Max local planned path error (*m*)	4.762	5.152	5.899	5.064
Min local planned path error (*m*)	0.00	0.00	0.00	0.00
Avg heading error (°)	8.036	0.622	32.148	2.265
Std of heading error (°)	6.218	0.076	23.411	1.075
Energy Consumption (*J*)	1.682	1.767	1.543	1.57
Avg Time For Abortion (*s*)	7.058	0.00	2.648	0.37

Table 4.5: Results of the Large Restricted Area simulation

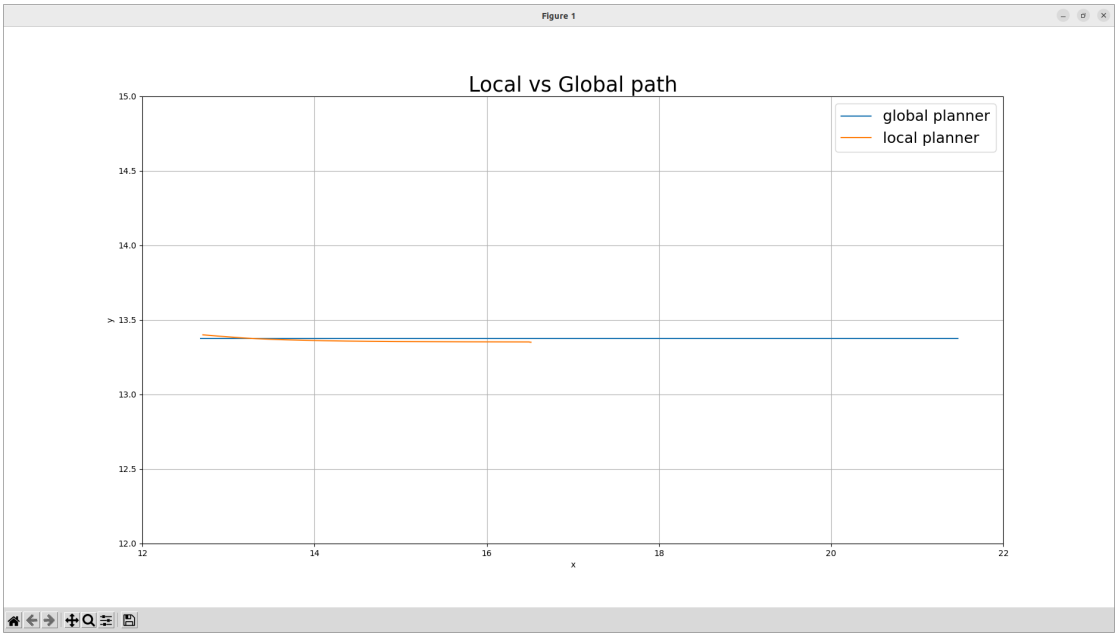


Figure 4.68: Difference between global planned path and DWB in the Large Restricted Area simulation.

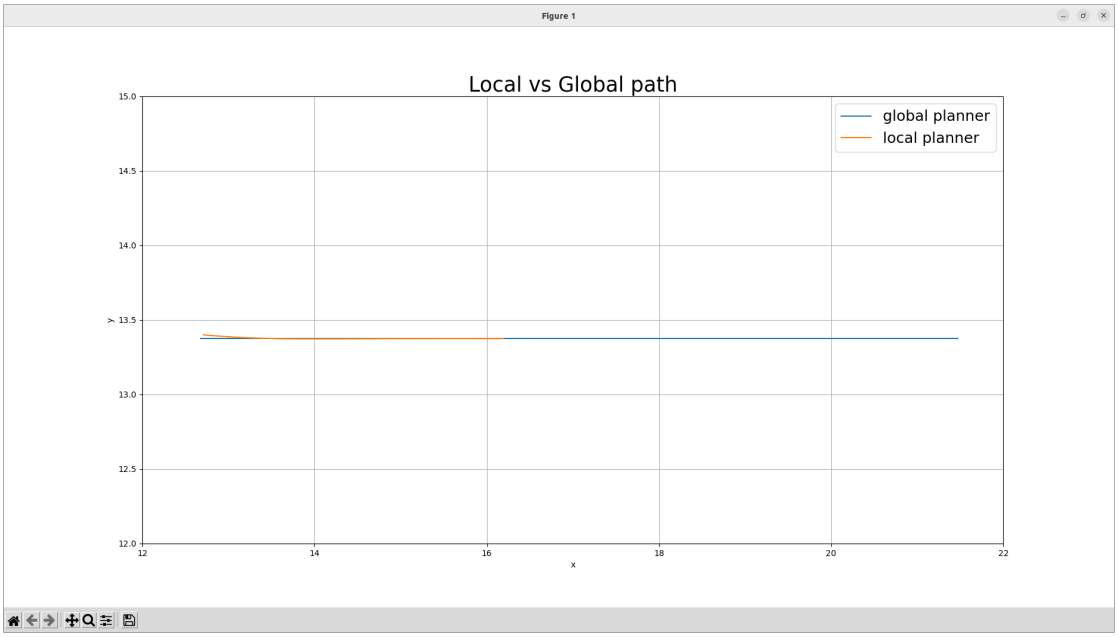


Figure 4.69: Difference between global planned path and RPP in the Large Restricted Area simulation.

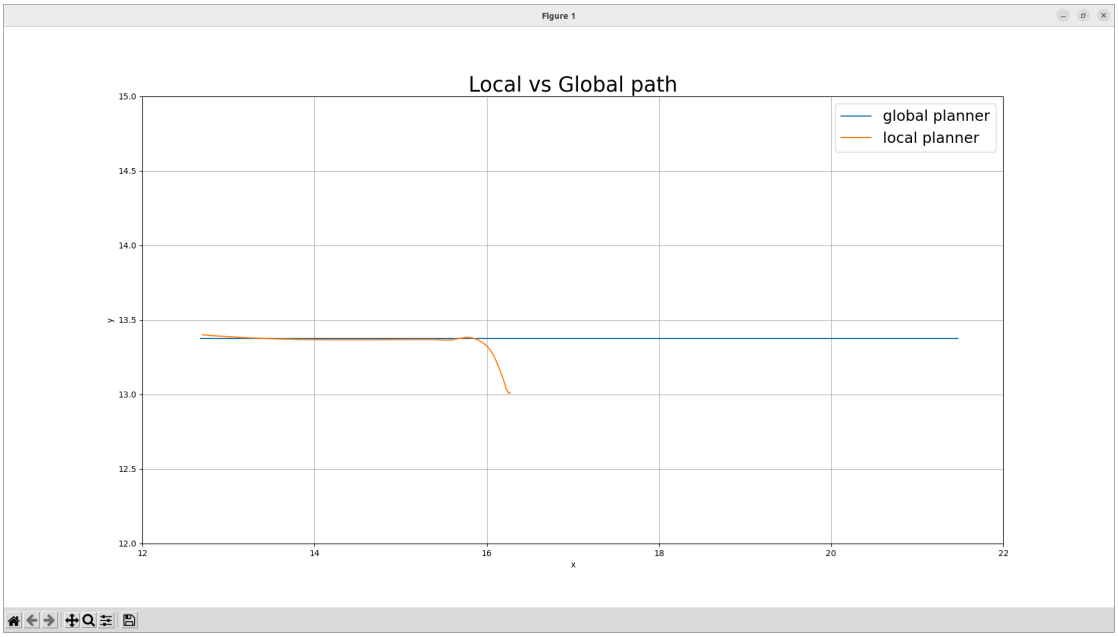


Figure 4.70: Difference between global planned path and TEB in the Large Restricted Area simulation.

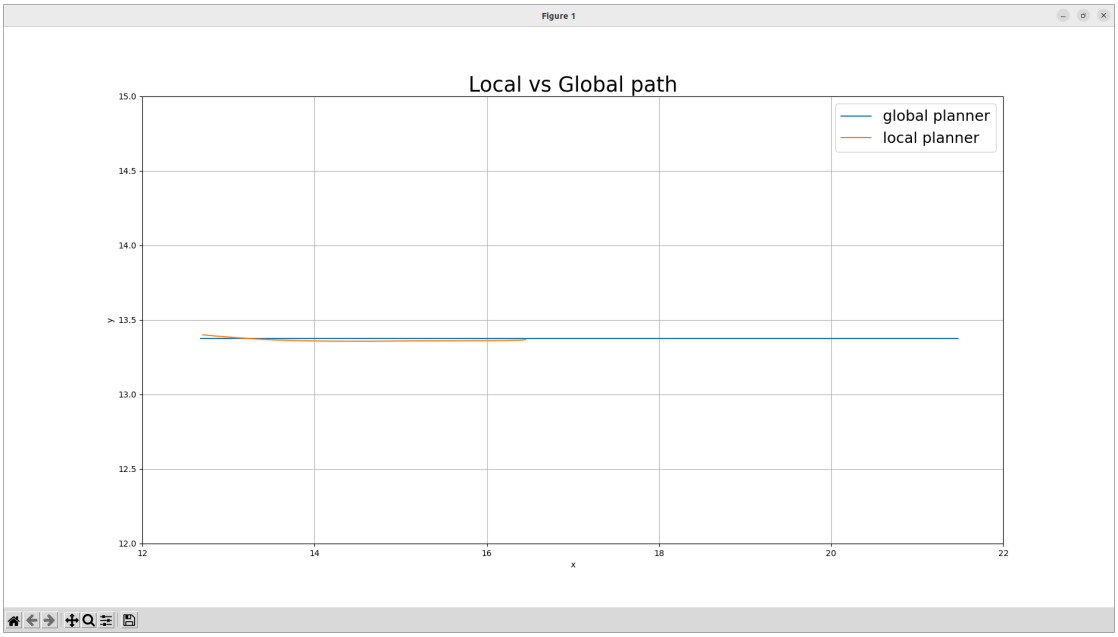


Figure 4.71: Difference between global planned path and MPPI in the Large Restricted Area simulation.

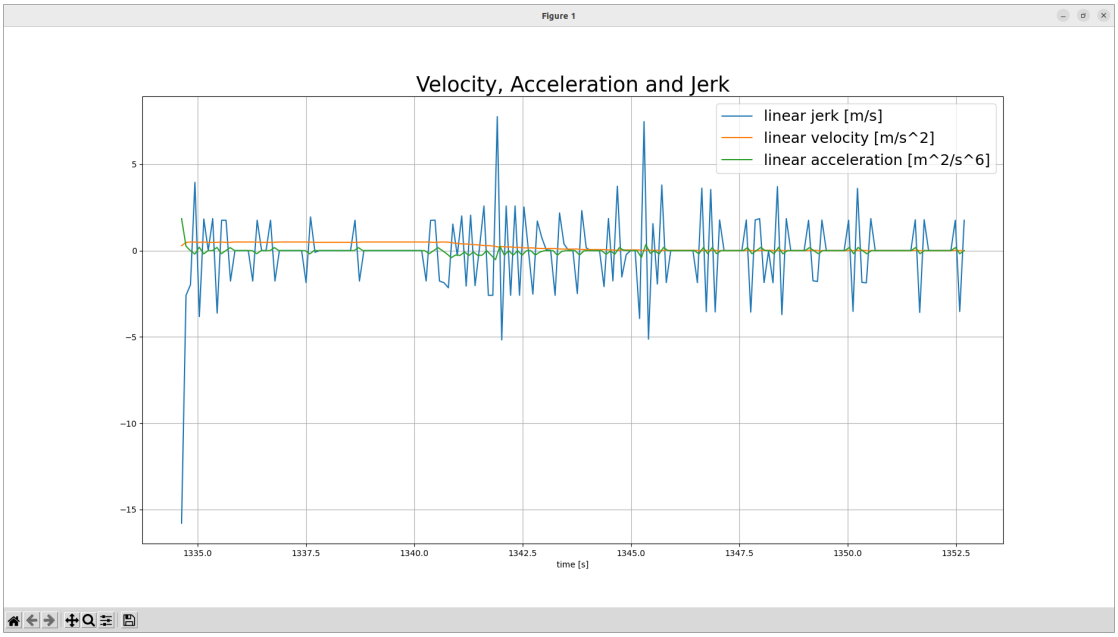


Figure 4.72: Velocity, acceleration and jerk of DWB in the Large Restricted Area simulation.

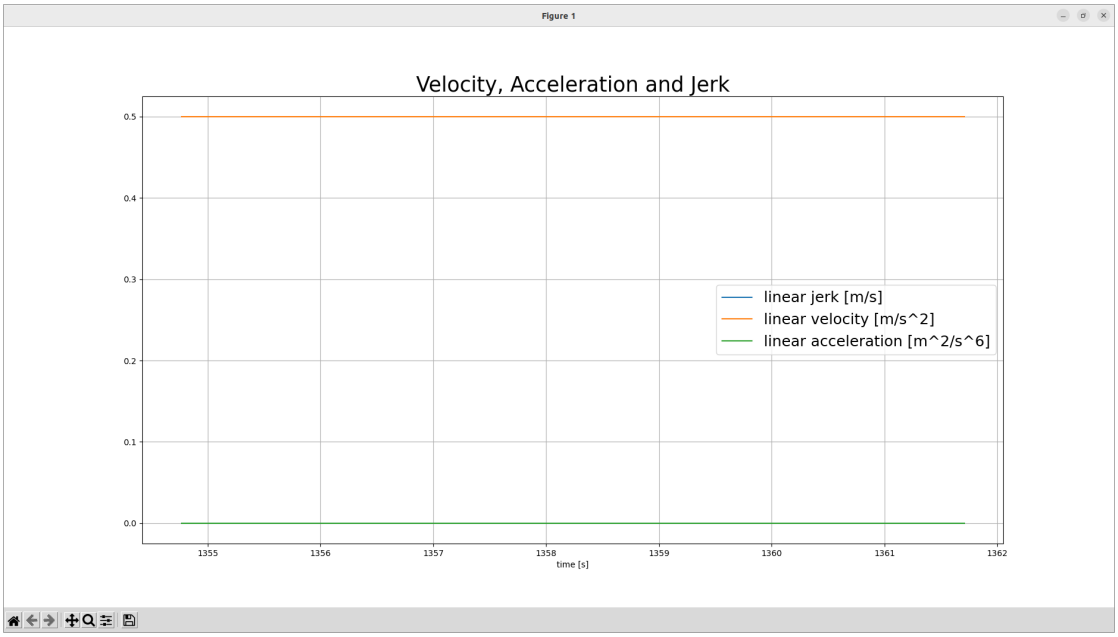


Figure 4.73: Velocity, acceleration and jerk of RPP in the Large Restricted Area simulation.

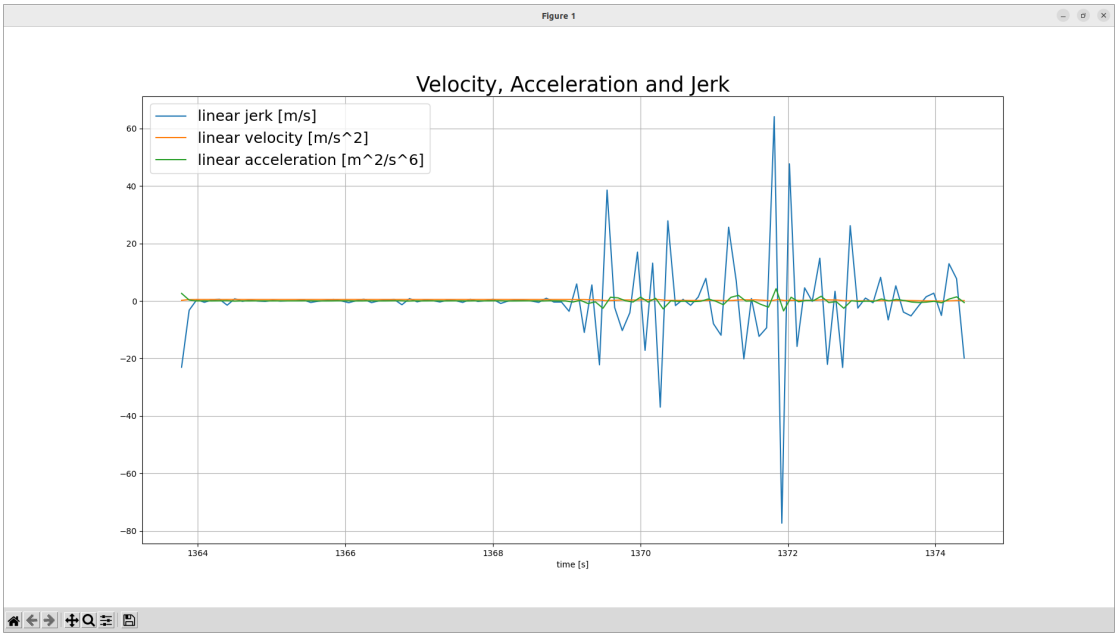


Figure 4.74: Velocity, acceleration and jerk of TEB in the Large Restricted Area simulation.

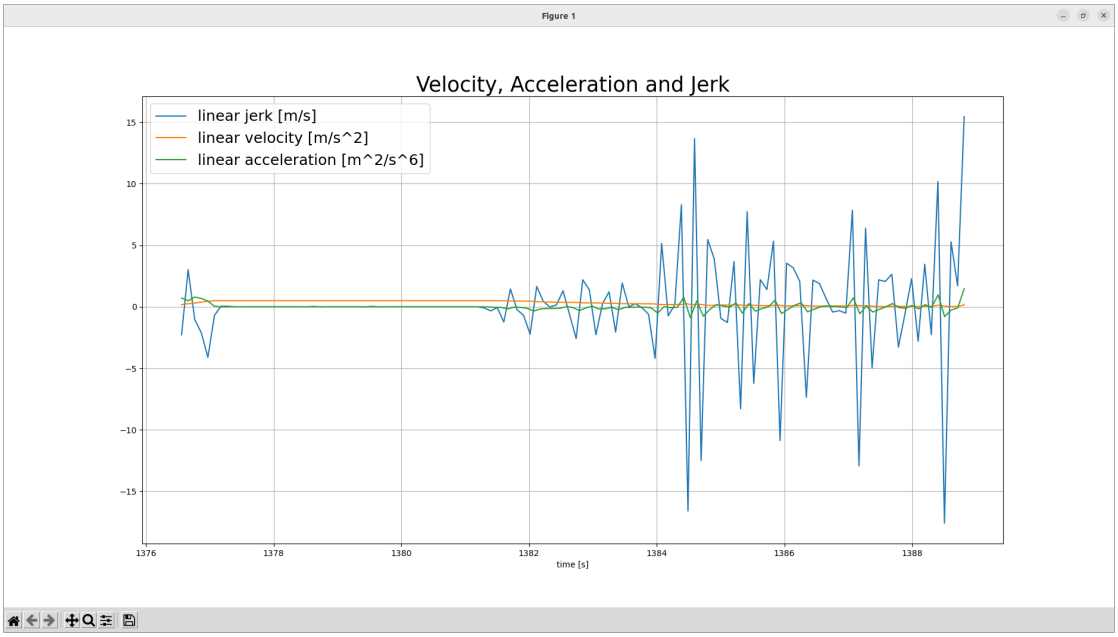


Figure 4.75: Velocity, acceleration and jerk of MPPI in the Large Restricted Area simulation.

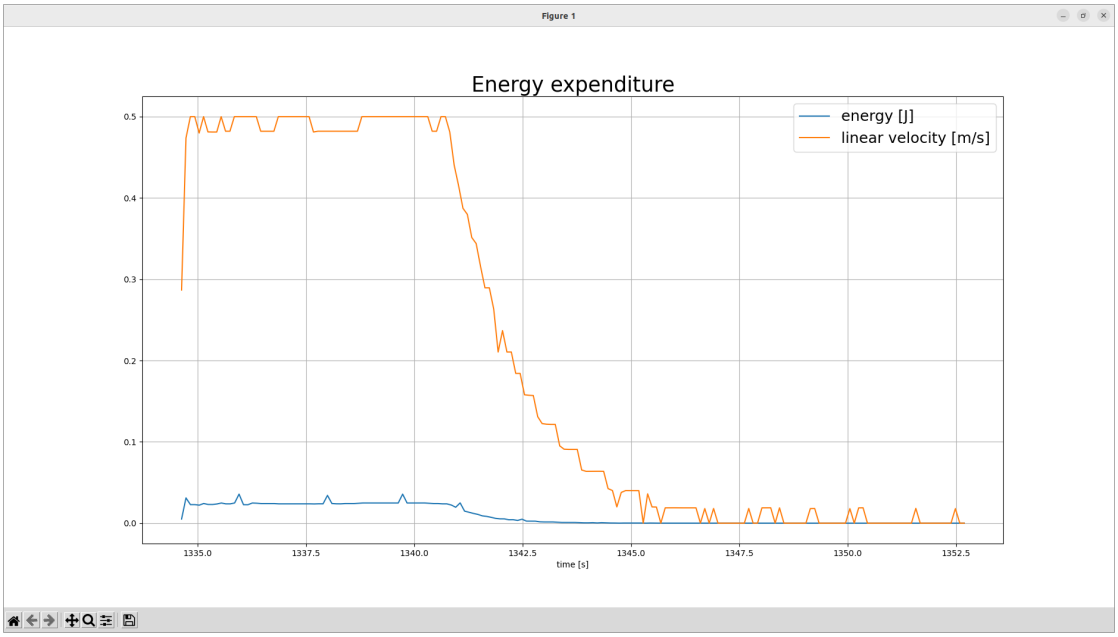


Figure 4.76: Velocity and energy expenditure of DWB in the Large Restricted Area simulation.

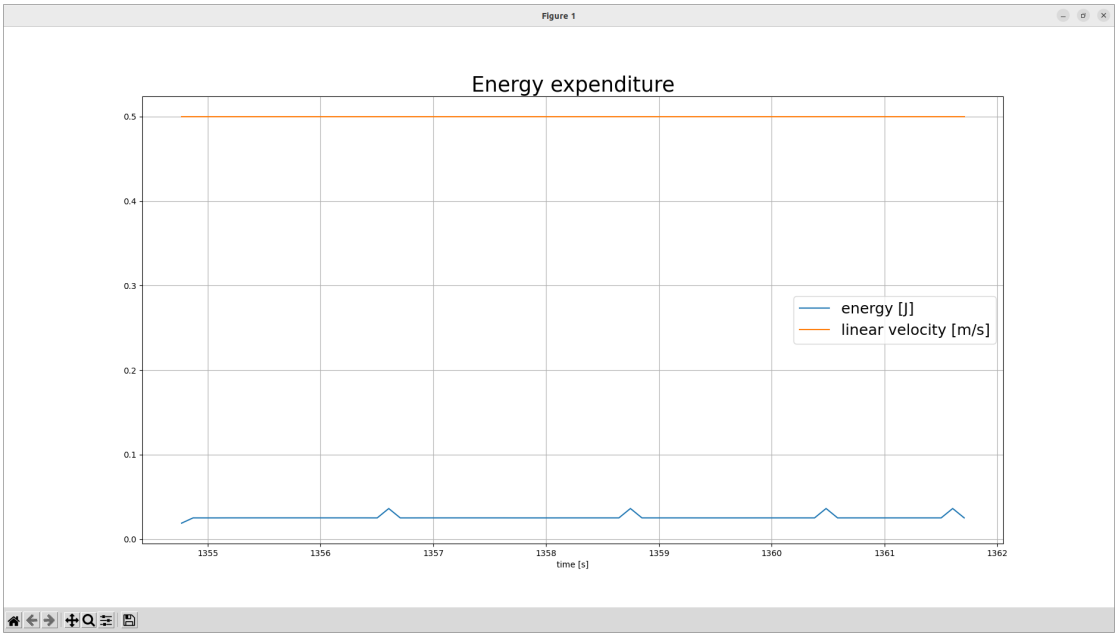


Figure 4.77: Velocity and energy expenditure of RPP in the Large Restricted Area simulation.

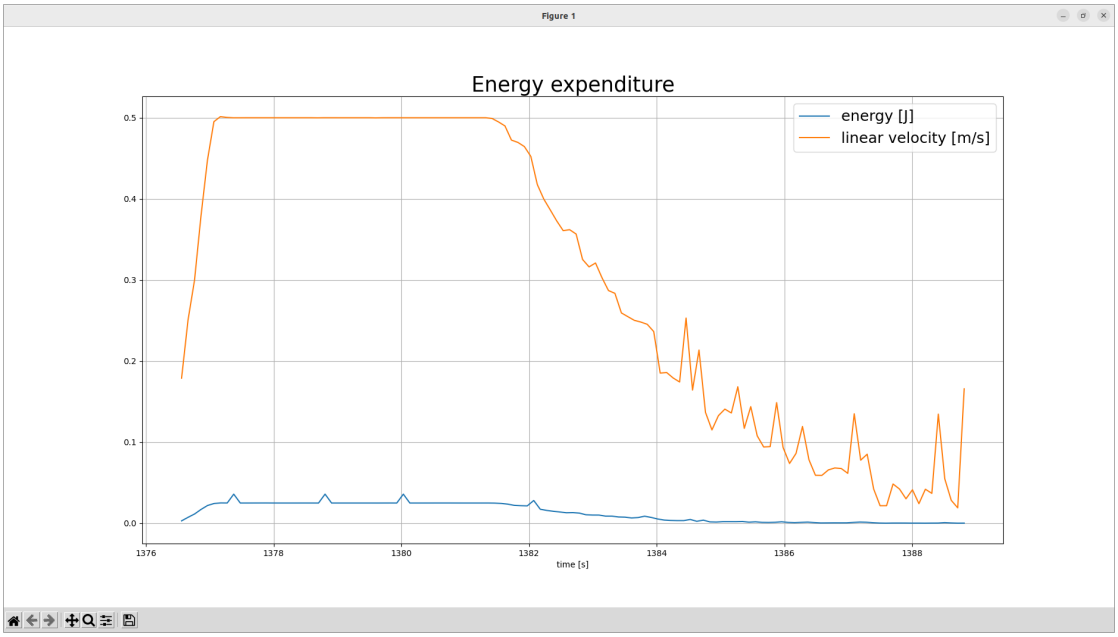


Figure 4.78: Velocity and energy expenditure of MPPI in the Large Restricted Area simulation.

4.4.3 Observations

The success rate is 0% for each controller, hence none of them successfully accomplished the tasks.
All of the controllers navigate at low speed, despite the RPP that navigates at the maximum velocity (0.5 m/s).

The maximum planned path error represents the distance between the last pose of the robot, where it aborts its task, and the actual goal. It is interesting to notice that, overall, all the controllers return the same value, despite the average heading error: the TEB controller has the highest heading error, because, when it is in close proximity to the obstacle, it makes pointless turns and tries repeatedly to find a path that may allow the completion of the task (even though it is not feasible). Also, this particular behavior is put in evidence by the fact that the TEB controller has a relatively high average time taken and very high average heading error, but small average time for abortion. Moreover, it has the highest average integrated jerk, hence it is the less smooth controller. All of these features make clear that it deals in the worst way with this specific situation.

It can be noticed that the DWB is the slowest above all the controllers and takes more time to complete, as well as abort, the task. Also, it stops at the minimum

possible distance between the obstacle and the robot, because when it tries to re-plan the path it decelerates a lot and slightly turns in close proximity of the obstacle.

The MPPI slowly navigates towards the obstacle until it reaches its minimum distance from it. It behaves similarly to the DWB, but it takes much less time to abort the impossible task and, while re-planning a new feasible path, it rotates less nearby the obstacle.

Finally, the RPP navigates at maximum speed until the robot reaches the point where the distance between itself and the obstacle is minimum. At that point the robot's speed will be 0 m/s. This means that this controller does not try to re-plan, thus avoiding robot's pointless rotations. This feature is cleared by the time spent by the controller to abort the impossible task taken into account, that, in this case, is 0 s.

In conclusion, the **RPP** manage better this condition.

Chapter 5

Conclusion

The automated benchmark that has been developed for this project is a tool that may be useful not only to Alba Robot, but also for several other users exploiting the controllers provided by Nav2. The developed code can precisely define the characteristics of each controller, according to quantitative statistical data based on multiple simulations.
For this project the controllers have been observed in 5 different situations (Empty World, Static Obstacles, One Obstacle, Small and Large Restricted Areas) in order to evaluate their smoothness, obstacle avoidance capability, path following capability and kinematics. Thanks to the usage of Gazebo and Rviz, it has been possible to carefully observe the behavior of the controllers. Moreover, the implemented Python code allowed the execution of this accurate analysis in an automatic and smart way.

The **DWB** controller seems to be the less smooth above all of the ones taken into account, because it has the highest jerk and centripetal acceleration in every use case. However, it follows the global planned path almost perfectly and avoid obstacles keeping safe distances from them. It also deals quite well with impossible tasks, because, when it goes in the close proximity of the constraint, it stops and makes very small rotations on the current pose.

The **RPP** is the only controller that is able to reach the maximum velocity in every condition. It has the best behavior in terms of path following: it has always the minimum difference between the global and the local planned and the minimum heading error. Moreover, it aborts impossible tasks in no-time, making the robot just stop, without rotating, when it is close to the constraint. Nonetheless, it has a high centripetal acceleration, which means that when making turns it does not navigate smoothly.

The **TEB** is the best controller in terms of obstacle avoidance, because it cuts corners but keeping safe distance from obstacles. Overall, it deals quite well with all the use cases, except for the Restricted Area one, because it endlessly try to generate new possibly feasible paths.

The **MPPI** controller avoid obstacles in the most dangerous way, because it keeps less distance from them cutting corners more than any others. However, it seems to be the smoothest above all the controllers taken into account, since it always has the smallest maximum centripetal acceleration and average linear jerk. Moreover, in case of impossible tasks, it makes the robot stop at very close proximity to the constraint and slightly rotate nearby it. These rotations are even smaller than the ones executed by the DWB.

In order to optimize the performance in these use cases, it is possible to improve the tuning of each controller. However, the results obtained by this analysis provide an interesting overview on each controller. This project, in fact, is the first step for the creation of a brand-new customized controller, that will be able to perfectly deal with any situation.
As demonstrated by the analysis, this new controller should merge the good qualities of the ones that have been observed. In details, it is possible to create a new plugins-based controller, which should include plugins concerning:

- constraints
- goal angle
- goal position
- costs on the costmap
- alignment with the path
- orientation of the robot on the path
- forward navigation preference
- avoidance of unnecessary twisting
- energy consumption
- sharp curves
- linear jerk
- maximum velocity

The customized controller should behave like the TEB, for what concerns obstacle avoidance, the RPP, for path following and the MPPI, for smoothness.

Bibliography

[1] Camera dei deputati. *La disabilità*. June 2015. URL: https://www.camera.it/temiap/2015/06/15/OCD177-1390.pdf (cit. on p. 1).

[2] Shrestha, B.P., Millonig, A., Hounsell, and N.B. et al. «Review of Public Transport Needs of Older People in European Context». In: 10 (Nov. 2016), pp. 343–361 (cit. on p. 2).

[3] Rozafa Basha. «Disability and Public Space – Case Studies of Prishtina and Prizren». In: *International Journal of Contemporary Architecture "The New ARCH"* 2 (Dec. 2015), pp. 54–66. DOI: 10.14621/tna.20150406 (cit. on p. 2).

[4] Rubio F, Valero F, and Llopis-Albert C. «A review of mobile robots: Concepts, methods, theoretical framework, and applications». In: *International Journal of Advanced Robotic Systems* 16(2) (Apr. 2019) (cit. on p. 6).

[5] Matijevics, Istvn, Simon, and Janos. «Behavior Trees in Robotics and AI: An Introduction». In: Aug. 2017. ISBN: 9781138593732. DOI: 10.1201/9780429489105 (cit. on p. 6).

[6] M. Quigley, B. Gerkey, K. Conley, J. Faust, T. Foote, J. Leibs, E. Berger, R. Wheeler, and A. Ng. «ROS: an open-source Robot Operating System». In: *IEEE International Conference on Robotics and Automation Workshop on Open Source Software*. 2009 (cit. on p. 7).

[7] Steve Macenski, Alberto Soragna, Michael Carroll, and Zhenpeng Ge. «Impact of ROS 2 Node Composition in Robotic Systems». In: *IEEE Robotics and Automation Letters* 8.7 (2023), pp. 3996–4003. DOI: 10.1109/LRA.2023.3279614 (cit. on p. 7).

[8] Steven Macenski, Tully Foote, Brian Gerkey, Chris Lalancette, and William Woodall. «Robot Operating System 2: Design, architecture, and uses in the wild». In: *Science Robotics* 7.66 (2022), eabm6074. URL: https://www.science.org/doi/abs/10.1126/scirobotics.abm6074 (cit. on p. 7).

[9] Geoffrey Biggs and Tully Foote. *Managed nodes*. June 2015. URL: https://design.ros2.org/articles/node_lifecycle.html (cit. on p. 12).

[10] Michele Colledanchise and Petter Ögren. «Improving Greenhouse's Automation and Data Acquisition with Mobile Robot Controlled System via Wireless Sensor Network». In: Dec. 2010. ISBN: 978-953-307-321-7. DOI: 10.5772/13401 (cit. on p. 14).

[11] Synopsys. *What is LiDAR?* URL: https://www.synopsys.com/glossary/what-is-lidar.html (cit. on p. 24).

[12] Nav2. *Navigation Plugins*. URL: https://navigation.ros.org/plugins/index.html (cit. on p. 25).

[13] D. Fox, W. Burgard, and S. Thrun. «The dynamic window approach to collision avoidance». In: *IEEE Robotics Automation Magazine* 4.1 (1997), pp. 23–33. DOI: 10.1109/100.580977 (cit. on p. 27).

[14] Steve Macenski, Shrijit Singh, Francisco Martin, and Jonatan Gines. «Regulated Pure Pursuit for Robot Path Tracking». In: *Autonomous Robots* (2023) (cit. on p. 31).

[15] Coulter RC. «Implementation of the Pure Pursuit Path Tracking Algorithm». In: (Jan. 1992) (cit. on p. 33).

[16] Campbell SF. «Steering control of an autonomous ground vehicle with application to the DARPA Urban Challenge». In: *International Journal of Contemporary Architecture "The New ARCH"* (2007) (cit. on p. 34).

[17] Christoph Roesmann, Wendelin Feiten, Thomas Woesch, Frank Hoffmann, and Torsten Bertram. «Trajectory modification considering dynamic constraints of autonomous robots». In: *ROBOTIK 2012; 7th German Conference on Robotics*. 2012, pp. 1–6 (cit. on p. 38).

[18] Grady Williams, Paul Drews, Brian Goldfain, James M. Rehg, and Evangelos A. Theodorou. «Aggressive driving with model predictive path integral control». In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1433–1440. DOI: 10.1109/ICRA.2016.7487277 (cit. on p. 44).

[19] Bent E. Sørenson. «FINANCIAL CALCULUS: AN INTRODUCTION TO DERIVATIVE PRICING: Martin Baxter and Andrew Rennie, Cambridge University Press, 1996». In: *Econometric Theory* 14.3 (1998), pp. 365–368. DOI: 10.1017/S0266466698143050 (cit. on p. 45).

[20] Wikipedia contributors. *Softmax function — Wikipedia, The Free Encyclopedia*. [Online; accessed 28-February-2024]. 2024. URL: https://en.wikipedia.org/w/index.php?title=Softmax_function&oldid=1206683487 (cit. on p. 46).

[21] Grady Williams, Andrew Aldrich, and Evangelos Theodorou. *Model Predictive Path Integral Control using Covariance Variable Importance Sampling*. 2015. arXiv: 1509.01149 [cs.SY] (cit. on p. 50).

[22] Abhijat Biswas, Allan Wang, Gustavo Silvera, Aaron Steinfeld, and Henny Admoni. «SocNavBench: A Grounded Simulation Testing Framework for Evaluating Social Navigation». In: *J. Hum.-Robot Interact.* 11.3 (July 2022). DOI: 10.1145/3476413. URL: https://doi.org/10.1145/3476413 (cit. on pp. 55, 56).