POLITECNICO DI MILANO
School of Industrial and Information Engineering
Master of Science in Automation and Control Engineering

Implementation and comparison in local planners for Ackermann vehicles

AI & R Lab
Laboratorio di Intelligenza Artificiale
e Robotica del Politecnico di Milano

Supervisor: Matteo Matteucci
Co-Supervisor: Alessandro Gabrielli

By: Jordi Ferrer Sánchez
ID number: 875932

Academic Year 2017/2018

Contents

List of Figures

Acronyms

B*	Optimal Timed Elastic Band
DWA	Dynamic Window Approach
Ipopt	Interior Point OPTimizer
MPC	Model Predictive Control
ROS	Robot Operating System
RViz	ROS visualization
TEB	Timed Elastic Bands

Abstract

Motion control plays an important role in autonomous navigation. It can be defined as the compute of motion control inputs for a collision-free path tracking.

To perform the motion control in path planning, a local planner is used. A good local planner should take into consideration the capabilities of the vehicle (kinematic and dynamic constraints), in order to compute feasible movements for the robot.

This work is focused on the local planning problem for non-holonomic vehicles in an (x, y, θ) space. Three local planners using different methods to compute the command velocities are implemented in the Robot Operating System (ROS) framework, widely used in the robotics community.

To fulfill our goal, three ROS nodes are created, the first one uses the Dynamic Windows Approach algorithm to compute the command velocities. The second uses the Timed Elastic Band algorithm. The third uses the Model Predictive Control algorithm.

To deal with the non-holonomic vehicle constraints, an Ackermann kinematic model is used.

Sommario

Il controllo del movimento svolge un ruolo importante nella navigazione autonoma, può essere definito come il calcolo degli input di controllo del movimento per un inseguimento del percorso senza collisioni.

Per eseguire il controllo del movimento nella pianificazione del percorso, viene utilizzato un pianificatore locale. Un buon pianificatore locale dovrebbe prendere in considerazione le capacità del veicolo (vincoli cinematici e dinamici), al fine di calcolare i movimenti fattibili per il robot.

Questo lavoro è incentrato sul problema di pianificazione locale per veicoli non-olonomici in uno spazio (x, y, θ). Tre pianificatori locali che utilizzano metodi diversi per calcolare le velocità di comando sono implementati nel framework Robot Operating System (ROS), oggi ampiamente utilizzato nella comunità di robotica.

Per raggiungere il nostro obiettivo, vengono creati tre nodi ROS, il primo utilizza l'algoritmo Dynamic Windows Approach per calcolare le velocità di comando. Il secondo utilizza l'algoritmo Timed Elastic Band. Il terzo utilizza l'algoritmo Model Predictive Control.

Per affrontare i vincoli non olonomici del veicolo, viene utilizzato un modello cinematico di Ackermann.

Acknowledgements

First of all, I would like to express my gratitude to my family, for their never ending support. Without them, this year and half abroad would not have been possible.

To my couple, to be always there no matter what.

To all my new friends, who introduced me into a new culture and made my stay a great experience. To my old friends, being there no matter the distance.

To my work supervisor, Prof. Matteo Matteucci, to help me everytime I asked and to give me that opportunity. And last but not least, to Alessandro Gabrielli and Gianluca Bardaro for all the support during the development of the tesina.

A mi familia, por su esfuerzo y apoyo incondicional

Chapter 1

Introduction

Autonomous mobile robot navigation is a field that has experienced a huge growth over the last decades. It plays a vital role in self-driving cars, warehouse robots, personal assistant robots, etc. Robots are increasingly operating in indoor environments designed for and shared with people, environment in which it is key for a robot to avoid unexpected obstacles.

In autonomous mobile robot navigation there are three main problems to underline: location, path planning and motion control. The first one lies in finding out where the robot is located in its current environment; The second one is about, given a goal, to perform a path that allows the robot to reach it; The last one formulation consists in computing the proper control inputs to be sent to the robot in order to keep tracking of the given path while it avoids collisions with obstacles.

When talking about path planning, it can be understood as both problems together, to generate the path and to follow it. So when refered to path planning two parts can be distinguished: global planner and local planner.

This work focuses on local planners, in particular on the implementation and adaptation of some planners under the Robot Operating System (ROS), for car-like vehicles, more specifically for Ackermann geometry.

ROS is a framework nowadays widely used by the robotics community. Its main purpose is to make the development of robot software more flexible. It is a collection of tools, libraries and conventions used to interact with different robotic platforms performing complex tasks. The navigation stack of ROS already contains some local planners, as DWA, but it doesn't take into account non-holonomic kinematics as Ackermann.

The planning problem, is not only required to generate plans which are collision-free and feasible, but they might satisfy a constraint in time, length or energy consumption, there the

need to switch to optimal planners taking into account the vehicle kinematics, and the ones provided by ROS do not.

In order to solve that problem, this work implements three local planners. Two already implemented in ROS as Dynamic Window Approach local planner and Timed Elastic Band local planner and one which is not, Model Predictive Control local Planner.

The thesis consists of 5 chapters, the 4 chapters that follow this introduction are outlined below

Chapter 2: this chapter discusses the state of the art. It introduces the basic algorithms used for the local planners and the framework where they are developed. It also introduces the vehicles these planners are designed for.

Chapter 3: this chapter discusses the implementation of the planners, the structure they have, how to handle certain variables and the modifications/adjustments done to work with Ackermann kinematics.

Chapter 4: this chapter contains the different tests performed and the comparisons between the planners.

Chapter 5: this chapter corresponds to the conclusions and future recommendations.

Chapter 2

State of the art

This chapter shows the state of the art on navigation, starting from motion planning history, showing the different algorithms implemented and then introducing ROS Navigation Stack.

2.1 Motion Planning History

Path planning research of autonomous mobile robot has attracted attention since the 1970s, being 1979 the beginning of modern motion planning, when Lozano-Pérez and Wesley [1] introduced the concept of configuration space (C-space).

On the last past decades, motion planning has become more and more significant since robots role on industry and also on our daily lives is getting more importance. Many areas of this problem have been the focus of reaserch and investigation to develop optimal solutions, such as path planning (i.e. to calculate the path the robot must follow to reach a given goal) or motion control (i.e. to calculate the control inputs to send the robot in order to follow the path with free-collision behaviour) by considering robot kinematic and dynamic constraints.

Regarding motion control, and more specifically path motion, many algorithms have been implemented to keep the path tracking, here we present some of them.

2.2 Dynamic Window Approach

The Dynamic Window Approach (DWA) is an online collision avoidance strategy for mobile robots developed by Dieter Fox, Wolfram Burgard, and Sebastian Thrun in 1997 [2]. Unlike other avoidance methods, the dynamic window approach is derived directly from the dynam-

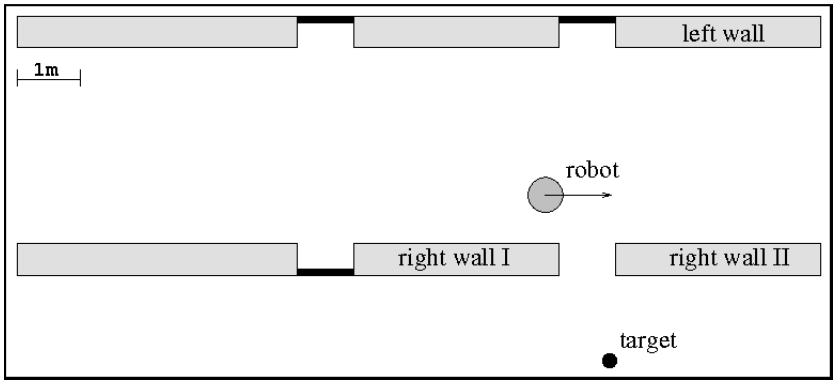

Figure 2.1: Example situation for a mobile robot.

ics of the robot, and is especially designed to deal with the constraints imposed by limited velocities and accelerations of the robot.

In a nutshell, the approach considers periodically only a short time interval when computing the next steering command to avoid the enormous complexity of the general motion planning problem. The approximation of trajectories during such a time interval by circular curvatures results in a two-dimensional search space of translational and rotational velocities.

This search space is reduced to the admissible velocities allowing the robot to stop safely. Due to the limited accelerations of the motors a further restriction is imposed on the velocities: the robot only considers velocities that can be reached within the next time interval. These velocities form the dynamic window which is centered around the current velocities of the robot in the velocity space.

Among the admissible velocities within the dynamic window, the combination of translational and rotational velocity is chosen by maximizing an objective function.

To explain everything in a more understandable way, we consider a robot in the situation seen in Fig. 2.1. In that situation, the velocity space V_s is generated: possible trajectories considering the next instant linear and rotational command velocities, represented in Fig. 2.2, where the pairs (v, ω) inside the gray area are forbiden trajectories (robot will collide if it follows that trajectories), the white area contains the admissible velocities V_a.

Considering $dist(v, \omega)$ the distance to the closest obstacle on the corresponding curvature, and $\dot{v_b}$, $\dot{w_b}$ the accelerations for breakage, we have Eq. 2.1.

$$V_a = \{v, \omega \mid v \leq \sqrt{2 \cdot dist(v, \omega) \cdot \dot{v_b}} \wedge w \leq \sqrt{2 \cdot dist(v, \omega) \cdot \dot{w_b}}\} \tag{2.1}$$

Considering now the velocity space, the robot has a current position (v_a, ω_a) that repre-

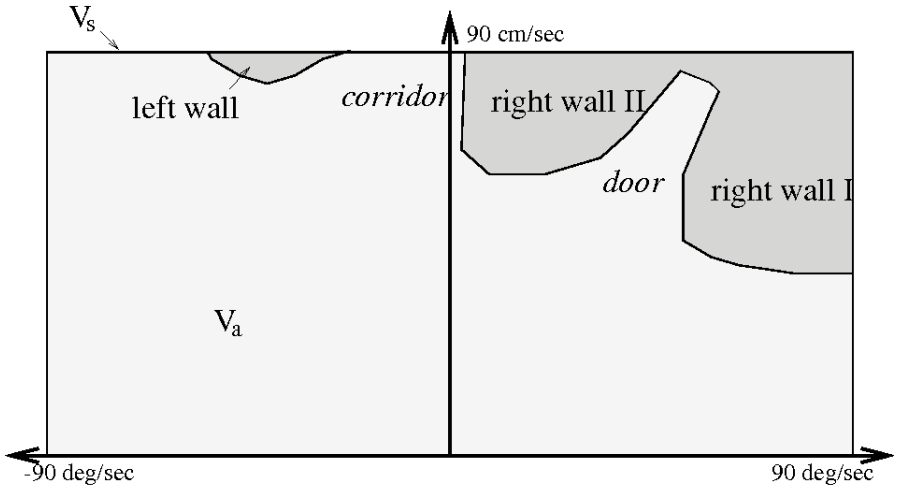

Figure 2.2: Velocity space.

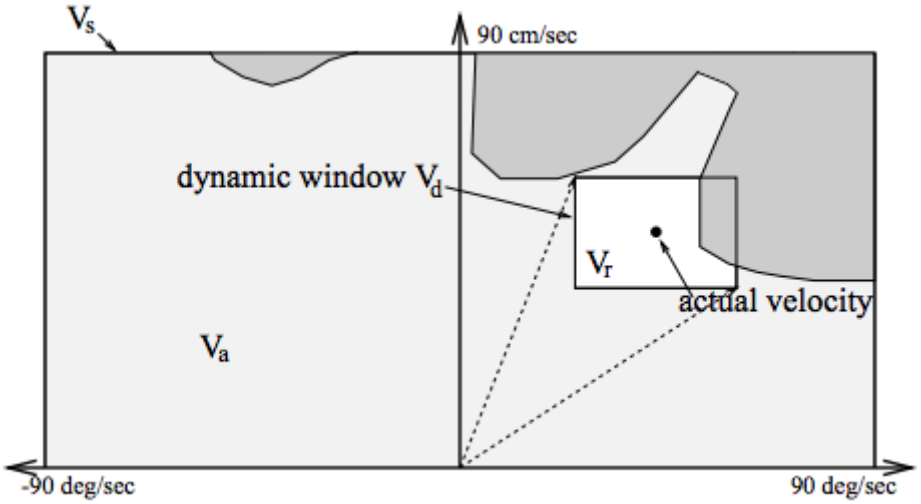

Figure 2.3: Dynamic window.

sents the actual velocities. The next command velocities sent to the robot must be feasible to follow considering the dynamic constraints (i.e. the robot accelerations must allow it to reach the next trajectory in time).

In Fig. 2.3 we see the actual velocity position of the robot, which is the center of the dynamic window V_d. The dynamic window is generated enlarging the position (v_a, ω_a) by the robot accelerations and timestep t (Eq. 2.2). All the pairs (v, ω) inside V_d are trajectories reachable by the robot "instantly".

$$V_d = \{v, \omega \mid v \in [v_a - \dot{v} \cdot t, v_a + \dot{v} \cdot t] \wedge \omega \in [\omega_a - \dot{\omega} \cdot t, \omega_a + \dot{\omega} \cdot t]\} \tag{2.2}$$

Considering now all the restrictions given above, the set of eligible velocities V_r is the intersection of the areas:

$$V_r = V_s \cap V_a \cap V_d \tag{2.3}$$

The resulting eligible velocity space is represented by the white area in Fig. 2.3.

The DWA algorithm written in pseudocode:

Algorithm 1 DWA pseudocode

```
1:  function DWA(robotPose, robotGoal, robotModel)
2:      laserscan ← readScanner()
3:      (v_allowable, w_allowable) ← generateWindow(robotVW, robotModel)
4:      for (each v in v_allowable) do
5:          for (each w in w_allowable) do
6:              dist ← findDist(v, w, laserscan, robotModel)
7:              breakDist ← calculateBreakingDistance(v)
8:              if (dist > breakDist) then
9:                  cost ← costFunction
10:                 if (cost > optimal) then
11:                     best_v ← v
12:                     best_w ← w
13:                     optimal ← cost
14:     return best_v, best_w
```

Among the eligible velocities V_r, a trajectory has to be chosen to send to the robot. In order to make that choice, all the trajectories are evaluated through a cost function. That cost function can depend on many factors, but usually has the following form.

$$F(v, w) = \alpha \cdot heading(v, \omega) + \beta \cdot clearance(v, \omega) + \gamma \cdot velocity(v, w) \tag{2.4}$$

Where α, β and γ are tunnable weights for the following functions.

- $heading(v, \omega)$: measures the alignment of the robot with the target direction (see Fig. 2.4).
- $clearance(v, \omega)$: represents the distance to the closest obstacle that intersects with the curvature.
- $velocity(v, w)$: represents the robot velocity in order to maximize it (if $\gamma \neq 0$).

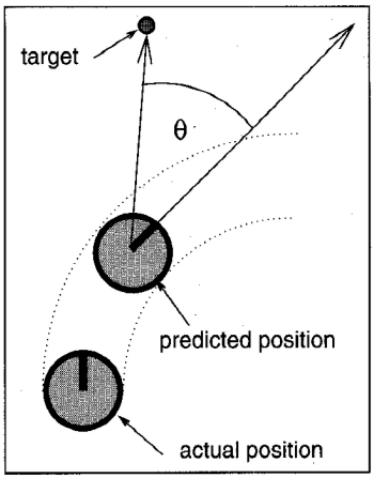

Figure 2.4: Heading of the robot.

2.3 Timed Elastic Band

The Timed Elastic Band (TEB) is an online collision avoidance method for online trajectory optimization. Timed Elastic Band local planner optimizes locally the robot's trajectory minimizing the trajectory execution time (time-optimal objective), separation from obstacles and compliance with kinodynamic constraints such as satisfying maximum velocities and accelerations. It was presented by Christoph Rösmann, Frank Hoffmann, and Torsten Bertram in 2012 [3].

A "classic" elastic band is composed of a fixed number n of geometric waypoints or vehicle poses P_i. The set of waypoints is described by

$$Q = \{P_i\}_{i=1...n} \tag{2.5}$$

where each waypoint consists of the tupel

$$P_i = \begin{pmatrix} x_i \\ y_i \\ \theta_i \end{pmatrix} \tag{2.6}$$

The timed elastic band is augmented by time intervals ΔT_i between two consecutive configurations, resulting in a sequence of $n-1$ time differences. In the context of collision the timed elastic bands merely optimizes the location of intermediate waypoints as there are no

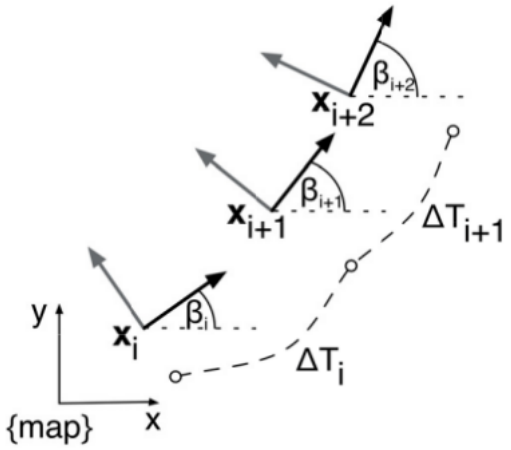

Figure 2.5: TEB: sequences of configurations and time differences.

boundary conditions for the final vehicle state. The set of time intervals is given by.

$$\tau = \{\Delta T_i\}_{i=1...n-1} \tag{2.7}$$

The TEB (Fig. 2.5) consists of the two sets:

$$B := (Q, \tau) \tag{2.8}$$

2.3.1 Cost Function

The optimal band is calculated by minimizing the objective function

$$f(B) = \sum_k \gamma_k \Gamma_k(B) \tag{2.9}$$

which is a weighted sum of multiple objectives and soft penalties for constraint violations. The optimal trajectory $B^\star$ is given by

$$B^\star = \min_B f(B) \tag{2.10}$$

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 (Eq. 2.21) or clearance from obstacles (Eq. 2.13). Sparse constrained optimization algorithms are not readily available in robotic frameworks

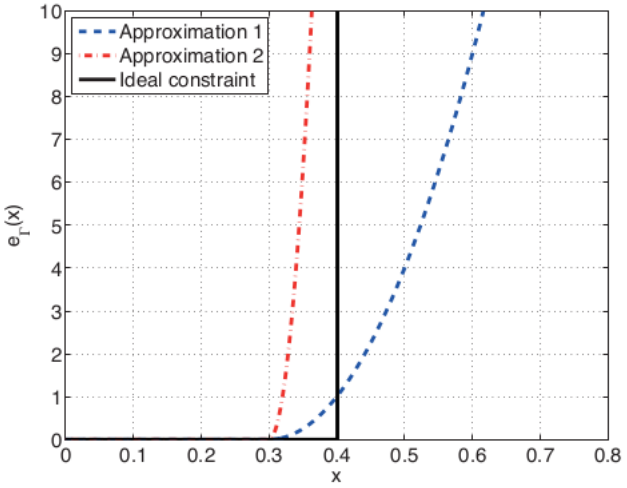

Figure 2.6: Polynomial approximation of constraint.

(e.g. ROS) in a freely usable implementation. Therefore, in the context of "timed elastic band" these constraints are formulated as objectives in terms of a piecewise continuous, differentiable cost function that penalize the violation of a constraint (Eq. 2.11).

$$e_\Gamma(x, x_r, \epsilon, S, n) = \begin{cases} \left(\frac{x-(x_r-\epsilon)}{S}\right)^n & \text{for } x > x_r - \epsilon \\ 0 & \text{for } x \leq x_r - \epsilon \end{cases} \tag{2.11}$$

Where x_r denotes the bound, S expresses the scaling, n the polynomial order and ϵ a small translation of the approximation.

This function approximates the discontinuous step function (see Fig. 2.6), for example it imposes a lower limit on the separation between the ego vehicle and the obstacle. The parameters ϵ, S, n and x_r are chosen such that for all realistic situations the violation of the hard constraint imposes a much higher penalty than the cost of ordinary objective functions.

Fig. 2.6 shows two different realizations of Eq.2.11. Approximation 1 results from parameter-set $n = 2$, $S = 0.1$, $\epsilon = 0.1$ and Approximation 2, which is conspicuously a stronger approximation, result from parameter-set $n = 2$, $S = 0.05$ and $\epsilon = 0.1$. This example shows an approximation of the constraint $x_r = 0.4$.

An obvious advantage of using a multi-objective optimization framework is the modular formulation of objective functions. The possible objective functions employed in the TEB are listed below.

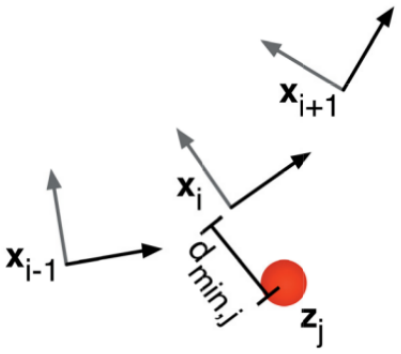

Figure 2.7: Minimal distance between TEB and way point or obstacle.

2.3.1.1 Way points and obstacles

The TEB simultaneously accounts for the attainment of the intermediate way points of the original path and the avoidance of static or dynamic obstacles. Both objective functions are similar with the difference that way points attract the elastic band whereas obstacles repel it. The objective function rests upon the minimal separation $d_{min,j}$ between the TEB and the way point or obstacle z_j (Fig. 2.7). In the case of way points the distance is bounded from above by a maximal target radius $r_{p_{max}}$ (Eq. 2.12) and in case of obstacles it is bounded from below by a minimal distance $r_{o_{min}}$ (Eq. 2.13). These constraints are implemented by the penalty function in Eq. 2.11.

$$\Gamma_{path} = e_\Gamma(d_{min,j}, r_{p_{max}}, \epsilon, S, n) \tag{2.12}$$

$$\Gamma_{ob} = e_\Gamma(-d_{min,j}, -r_{o_{min}}, \epsilon, S, n) \tag{2.13}$$

According to Fig. 2.6, the signs of the separation $d_{min,j}$ and the bound $r_{o_{min}}$ in Eq. 2.13 must be swapped to realize a bounding from below.

Notice, that the gradient of these objective functions can be interpreted as an external force acting on the elastic band.

2.3.1.2 Velocity and acceleration

Dynamic constraints on robot velocity and acceleration are described by similar penalty function as in the case of geometric constraints. Fig. 2.5 shows the structure of TEB. The mean translational and rotational velocities are computed according to the euclidean or angular

distance between two consecutive configurations P_i, P_{i+1} and the time interval ΔT_i for the transition between both poses.

$$v_i = \frac{\sqrt{(x_{i+1} - x_i)^2 + (y_{i+1} - y_i)^2}}{\Delta T_i} \tag{2.14}$$

$$\omega_i = \frac{\beta_{i+1} - \beta_i}{\Delta T_i} \tag{2.15}$$

Due to the vicinity of configurations the euclidean distance is a sufficient approximation of the true length of the circular path between two consecutive poses. The acceleration relates two consecutive mean velocities, thus considers three consecutive configurations with two corresponding time intervals.

$$a_i = \frac{2(v_{i+1} - v_i)}{\Delta T_i + \Delta T_{i+1}} \tag{2.16}$$

For the sake of clarity, the three consecutive configurations are substituted by their two related velocities in Eq. 2.16. The rotational acceleration is computed similarly by considering rotational velocities instead of translational ones. Considering a differential drive mobile robot, the relationship between the wheel velocities and the translational and rotational velocities v_i and ω_i of the robot center point are computed according to:

$$v_{w_r,i} = v_i + \frac{l}{2}\omega_i \tag{2.17}$$

$$v_{w_l,i} = v_i - \frac{l}{2}\omega_i \tag{2.18}$$

in which the parameter l denotes half of the robot wheelbase.

Differentiating Eq. 2.17 and Eq. 2.18 with respect to time leads to the corresponding wheel accelerations. The wheel velocities and acceleration are bounded from above and below according to the manufacturer specifications.

2.3.1.3 Non-holonomic kinematics

Robots with a differential drive only possess two local degrees of freedom. Thus they can only execute motions in the direction of the robot's current heading. This kinematic constraint leads to a smooth path that is composed of arc segments. Thus two adjacent configurations are required to be located on a common arc of constant curvature as illustrated in Fig. 2.8.

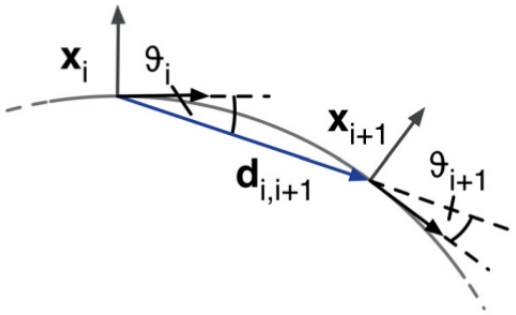

Figure 2.8: Relationship between configurations on a circle for non-holonomic kinematics.

The angle θ_i between the initial configuration P_i and the direction $d_{i,i+1}$ has to be equal to the corresponding angle θ_{i+1} at the final configuration P_{i+1}.

If β_i denotes the absolute orientation of a robot at the i-th configuration the arc condition demands Eq. 2.19.

$$\theta_i = \theta_{i+1} \Leftrightarrow \begin{pmatrix} cos\beta_i \\ sin\beta_i \\ 0 \end{pmatrix} \times d_{i,i+1} = d_{i,i+1} \times \begin{pmatrix} cos\beta_{i+1} \\ sin\beta_{i+1} \\ 0 \end{pmatrix} \tag{2.19}$$

with the direction vector:

$$d_{i,i+1} := \begin{pmatrix} x_{i+1} - x_i \\ y_{i+1} - y_i \\ 0 \end{pmatrix}$$

The corresponding objective function

$$\Gamma_k(P_i, P_{i+1}) = \left\| \left[\begin{pmatrix} cos\beta_i \\ sin\beta_i \\ 0 \end{pmatrix} + \begin{pmatrix} cos\beta_{i+1} \\ sin\beta_{i+1} \\ 0 \end{pmatrix} \right] \times d_{i,i+1} \right\|^2 \tag{2.20}$$

penalizes the quadratic error in the violation of this constraint.

2.3.1.4 Fastest path

Previous "elastic band" approaches obtain the shortest path by internal forces that contract the elastic band. Since our approach considers temporal information the objective of a shortest

path we have the option to replace the objective of a shortest path with that of a fastest path, or to combine those objectives. The objective of a fastest path is easily achieved by minimizing the square of the sum of all time differences.

$$\Gamma_k = (\sum_{i=1}^{n} \Delta T_i)^2 \tag{2.21}$$

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

2.3.2 Homotopy and Homology

TEB local planner can sometimes get stuck in a locally optimal trajectory as it is unable to transit across obstacles. A subset of admissible trajectories of distinctive topologies is optimized in parallel. The local planner is able to switch to the current globally optimal trajectory among the candidate set. Distinctive topologies are obtained by utilizing the concept of homotopy/homology classes.

2.3.2.1 Homotopic trajectories

Two trajectories τ_1 and τ_2 connecting the same start and goal points z_s and z_g respectively, are homotopic if and only if one can be continuously deformed into the other without intersecting any obstacles. The set of all trajectories that are homotopic to each other is denoted as homotopy class. In Fig. 2.9 there can be seen two different homotopy classes.

The closed form generic computation of homotopy classes is difficult. So homotopy classes are substituted by homology classes as they are easier to compute. A homology class defines a set of homologous trajectories in which elements are homologous to each other.

2.3.2.2 Homologous trajectories

Two trajectories τ_1 and τ_2 connecting the same start and goal points z_s and z_g respectively, are homologous if and only if $\tau_1 \sqcup -\tau_2$ forms the complete boundary of a $2D$ manifold embedded in $\mathbb{R}^2$ (i.e. going from start to goal with τ_1 and coming back with τ_2 without surrounding any obstacle).

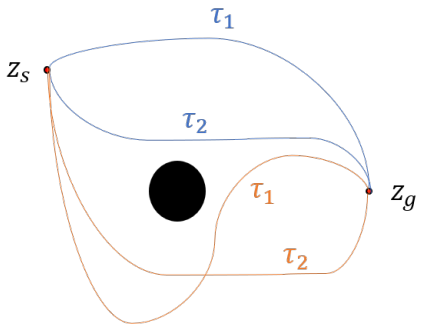

Figure 2.9: Two homotopy classes.

Homotopy implies homology, but the reverse implication does not hold. However, for most practicable mobile robot planning scenarios, both definitions can be considered as equivalent.

Multiple trajectory planning, distinguishing several sets of trajectories to optimize using the concept of homology, is a good solution to avoid local optimums.

2.4 Model Predictive Control

The Model Predictive Control (MPC) is an advanced control method that works in discrete time. From a set of state values, and with respect to a model, it optimizes a problem around an objective and gives a sequence of control signals as outputs. The first set of control values are then used as inputs to the system plant, and after a short period, set as the system time step, the new state values are measured and the process is repeated.

The beginning of MPC was at Shell Oil Company in 1979 where an idea named as "Dynamic Matrix Control" was presented by Cutler and Ramaker ([9], [10]). DMC was the first type of predictive control that could be applied in industry. The idea was to handle multivariable control systems without any constraints and predict future values for linear systems. The idea that the algorithm would predict future plant behavior was discovered to lead to a less aggressive output and a smoother convergence to the target set point.

MPC is based on iterative, finite-horizon optimization of a system model. At time t the current system state is sampled and a cost minimizing control strategy is computed (via a numerical minimization algorithm) for a relatively short time horizon in the future H.

In an MPC problem we can distinguish:

- Internal dynamic model of the system.
- History of past control moves.

- Optimization cost function.

With the dynamic model of the system we can predict the future states of the system in a discrete-time way.

$$x_{k+1} = Ax_k + Bu_k \tag{2.22}$$

$$x_k \in \mathbb{R}^n, u_k \in \mathbb{R}^m \tag{2.23}$$

Using the current state values we predict one step ahead with the time step H.

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k \\ x_{k+2} &= Ax_{k+1} + Bu_{k+1} \\ &\vdots \\ x_{k+H} &= Ax_{k+H-1} + Bu_{k+H-1} \end{aligned}$$

This one-step prediction can then be used recursively to find the n-step prediction by inserting the expression of x_{k+1} in x_{k+2}, x_{k+2} in x_{k+3}, etc.

$$\begin{aligned} x_{k+1} &= Ax_k + Bu_k \\ x_{k+2} &= A^2x_k + ABu_k + Bu_{k+1} \\ &\vdots \\ x_{k+H} &= A^Hx_k + A^{H-1}Bu_k + A^{H-2}Bu_{k+1} + \cdots + ABu_{k+H-2} + Bu_{k+H-1} \end{aligned}$$

The predictions at time k is then expressed as

$$\begin{bmatrix} x_{k+1|k} \\ x_{k+2|k} \\ \vdots \\ x_{k+H|k} \end{bmatrix} = \begin{bmatrix} A \\ A^2 \\ \vdots \\ A^H \end{bmatrix} x_k + \begin{bmatrix} B & 0 & \dots & 0 \\ AB & B & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ A^{H-1}B & A^{H-2}B & \dots & B \end{bmatrix} \begin{bmatrix} u_{k|k} \\ u_{k+1|k} \\ \vdots \\ u_{k+H-1|k} \end{bmatrix} \tag{2.24}$$

where $x_{k+1|k}$ is the one-step prediction at time k.

Intuitively it can be seen that the prediction structure is built such that the decision variables $u_{k|k} \dots u_{k+H-1|k}$ controls the state trajectory, which is important in trajectory tracking when we often desire specific predictions.

The deductions seen above are for linear systems, anyway it is possible to apply the same

procedure to non-linear systems, linearizing arround the current state of the system.

The problem objective is expressed as a quadratic program. A basic problem formulation looks like

$$\begin{aligned} &\text{minimize} \quad \sum_{k=1}^{H} x^T Q x + \Delta u^T R \Delta u \\ &\text{subject to} \\ &\qquad x_{k+1} = Ax_k + Bu_k, k = 1, \ldots, H-1 \\ &\qquad x \in \mathbb{R}^n, u \in \mathbb{R}^m \\ &\qquad u_k \in \mathbb{U} \\ &\qquad x_k \in \mathbb{X} \end{aligned} \tag{2.25}$$

where the objective is minimized over the horizon H with respect to the model, the variable constraints ($\mathbb{X}$) and possibly control limitations ($\mathbb{U}$). An MPC works in stages and using the model it predicts plant behavior several steps in the future. The objective is subject to several iterations depending on the horizon set for the optimization. The resulting output of the optimization is a sequence of control signals and predictions.

In path following applications the output control trajectory will correspond to predictions of the vehicles position at a given time step. Only the first control signals u_1 will be sent to the plant and the controller will perform another optimization, where a new control sequence is calculated. In the new optimization the horizon has therefore moved one step into the future.

Algorithm 2 Basic MPC control loop

1: Measure x_k
2: Solve the problem formulation (2.25) with x_k as initial state, where u_k is calculated
3: Set u_k as input to the plant
4: Repeat steps 1-3

2.5 ROS

The Robot Operating System (ROS) has been developed by Willow Garage and Stanford University as a part of STAIR project, as a free and open-source robotic middleware for the large-scale development of complex robotic systems. It is an open-source, meta-operating system for robots. It provides the services like an operating system, including hardware abstraction, low-level device control, message-passing between processes, and package man-

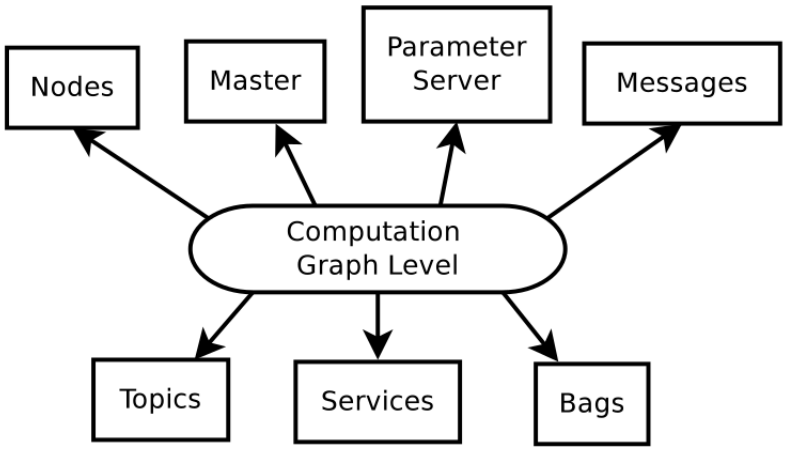

Figure 2.10: ROS Computation Graph Level.

agement [6]. Nowadays it is widely accepted and used in the robotics community. Its main goal is to make the multiple components of a robotics system easy to develop and share so they can work on other robots with minimal changes.

ROS has three levels of concepts: the Filesystem level, the Computation Graph level, and the Community level. For our purposes we require to understand the Computation Graph Level, which is the peer-to-peer network of ROS processes. Its basic elements are shown on Fig. 2.10 and explained below.

- **Nodes:** Nodes are processes that perform computation; ROS is designed to be modular, therefore a robot system is conformed by many nodes, which separates the code and functionalities, making the system simpler. They are implemented with the use of a ROS client library as roscpp or rospy.
- **Master:** The ROS master provides naming registration and lookup to the rest of the Computation Graph services. The role of the master is to enable individual ROS nodes to locate each other. Once these nodes have found each other, they communicate in a peer-to-peer fashion.
- **Parameter Server:** The parameter server allows data to be stored in a central location. With the parameter server it is possible to configure the nodes while running or to change the working parameters of a node.
- **Messages:** Nodes communicate with each other by passing messages. A message contains data that provides information to other nodes, it is a data structure comprising typed fields.

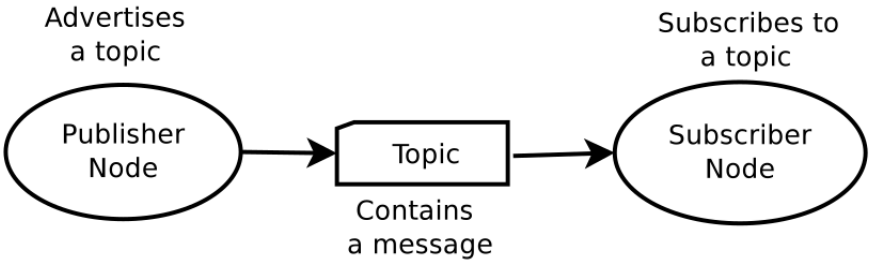

Figure 2.11: ROS Communication between Publisher and Suscriber Nodes.

- **Topics:** Messages are routed via a transport system with publish / subscribe semantics. A node sends out a message by publishing it to a given topic. The topic is a name that is used to identify the content of the message, therefore a node that is interested in a certain kind of data will subscribe to the appropriate topic. When a message is received on a topic, a user callback function is called, this function is in charge of managing the obtained data; the performed action can even be the simple storage of the data.
- **Services:** When a topic is published, the data is sent in a many-to-many fashion (asynchronous), which means a topic can have one or more publishers / subscribers. This model is not appropriate for request/reply interactions, instead it is done via a Service, which is defined by a request message and a reply message in a one-to-many fashion (synchronous), there can be only one server and one or more clients. A node offers a service under a name and a client uses the service by sending the request message and awaiting the reply.
- **Bags:** Bags are the primary mechanism for data logging in ROS, they allow to record datasets, visualize it, label it, and store for future use. The store data can be played back and use as if a ROS node was sending it.

Fig. 2.11 shows how the communication happens with a publisher and a subscriber node, the publisher advertises a message in a topic and the other node will subscribe to the topic to obtain the message.

In ROS, there is also the possibility to use Timers, they let you schedule a callback to happen at a specific rate; timers are not to be intended as real-time threads. To use a timer the user has to start it (can be done during a callback of a topic for instance); once the timer has been started a callback function is executed, when the operation is finished the timer must be stopped so that it can be started again later on.

Another important aspect about ROS is that it comes with a 3D visualizer called RViz (ROS visualization), the idea behind RViz is to provide visualization displays for different

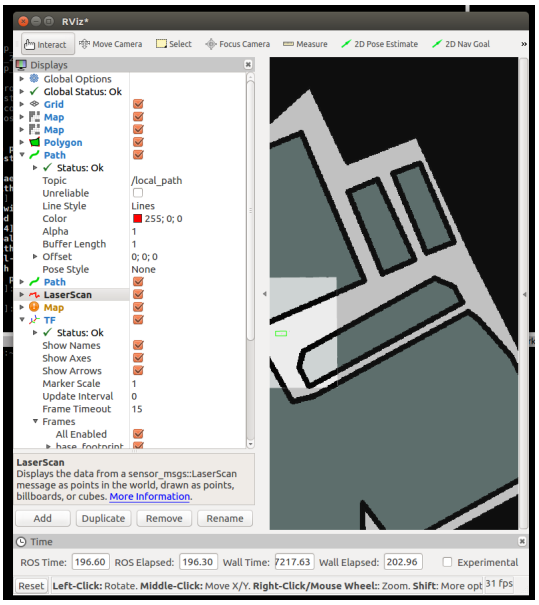

Figure 2.12: Rviz framework.

ROS messages, it is easy to use, the user only needs to select a display either by the message type, or search also by the available topic names. For navigation several displays can be used as Map, Path, Pose, Odometry, TF, etc. Each display has some options that can be setup like the color, and the topic to subscribe. The visualizer also allows to save a configuration to be able to use it for a specific application, so there is no need to setup the visualizer every time. Fig. 2.12 shows an example for setting up RViz displays.

TF is especially important for the navigation implementation. It lets to keep track of multiple coordinate frames over time (see Fig, 2.13). TF maintains the relationship between coordinate frames in a tree structure buffered in time, and lets to transform points, vectors, etc, between any two coordinate frames at any desired point in time.

2.5.1 Stage

Stage is a robot simulator. It provides a virtual world populated by mobile robots and sensors, along with various objects for the robots to sense and manipulate.

There are three ways to use Stage:

- The Stage program: a standalone robot simulation program that loads your robot control program from a library that you provide.
- The Stage plugin for Player (libstageplugin) - provides a population of virtual robots

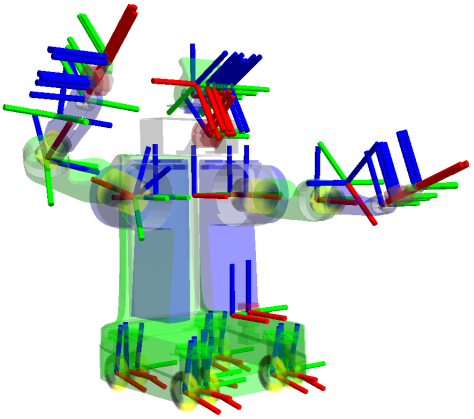

Figure 2.13: TF keeping track of several frames.

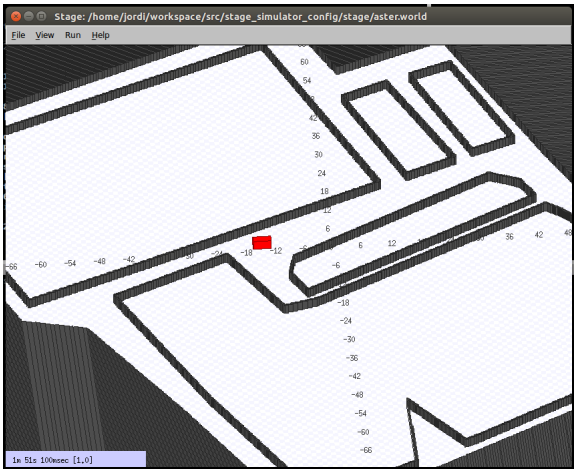

Figure 2.14: Stage simulator.

for the popular Player networked robot interface system.

- Write your own simulator: the "libstage" C++ library makes it easy to create, run and customize a Stage simulation from inside your own programs.

Stage provides fairly simple, computationally cheap models of lots of devices rather than attempting to emulate any device with great fidelity. This design is intended to be useful compromise between conventional high-fidelity robot simulations, the minimal simulations described by Jakobi [11], and the grid-world simulations common in artificial life research [12]. Stage is intended to be just realistic enough to enable users to move controllers between Stage robots and real robots, while still being fast enough to simulate large populations. Fig. 2.14 shows a Stage framework example.

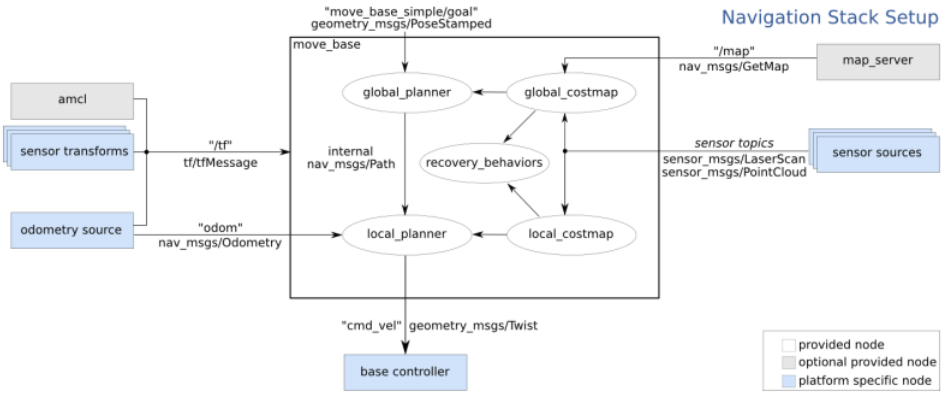

Figure 2.15: Navigation Stack node design.

2.5.2 Navigation Stack

For mobile robot navigation, several tasks are required to solve three main problems: mapping, localization and path planning. ROS has a set of useful resources for a robot to navigate through a known, partially known, or unknown environment; using these the robot is capable of planning and tracking a path while it deviates from obstacles that appear on its path throughout the course. These resources are found on the Navigation Stack [5].

The Navigation Stack Setup can be seen on Fig. 2.15. The move base node subscribes to odometry, sensor data, and goal position messages and produces velocity commands to be send to the mobile base. It can also subscribe to a map, so it can generate a better path with known information of the environment; the Adaptive Monte Carlo Localization Algorithm is done to performe the localization part. The Navigation Stack main component is the move base package, it is divided into global and local path planning modules, and it maintains a global and local costmap, used by the respective planners, which keep the information of the obstacles in the environment in the form of an occupancy grid. The global costmap is initialized with a static map, if one is available, and then it can be updated with data coming from the sensors. For the other hand, the local costmap is initialized from the global costmap, taking the local area of the robot (see Fig. 2.16).

For any global or local planner to be used by the move base package, they have to ad- here to some interfaces defined in the nav core package. The global planner must adhere to the nav core::BaseGlobalPlanner interface and the local planner to the nav core::BaseLocalPlanner interface; also they must be added as plugins to ROS in order to work with the navigation stack.

The current interfaces provided by the Navigation Stack are listed below [7]:

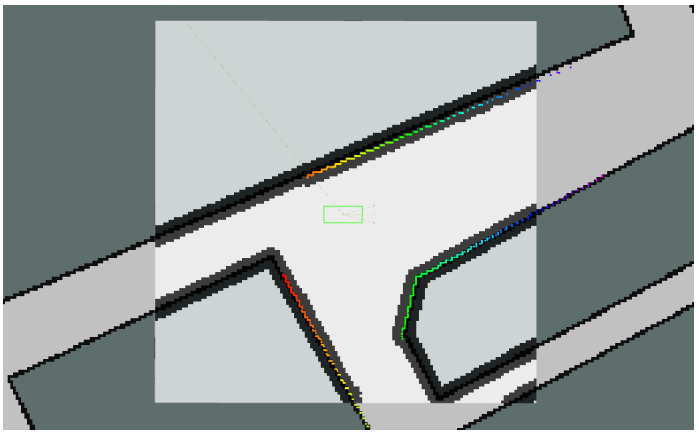

Figure 2.16: Local costmap.

- **Global Planners**
  - **navfn:** Grid-based global planner that uses a navigation function to compute a path for a robot.
  - **global_planner:** Fast interpolated global planner built as a replacement to the older navfn. This one can use the Dijkstra algorithm or the A*, or if required it can also behave as the old navfn. The behavior is set through the parameters.
  - **carrot_planner:** Simple global planner that takes a user-specified goal point and attempts to move the robot as close to it as possible, even when that goal point is in an obstacle.
- **Local Planners**
  - **base_local_planner:** Provides implementations of the Dynamic Window and Trajectory Rollout approaches to local control.
  - **dwa_local_planner:** Modular DWA implementation with a lot cleaner and easier interface to understand and more flexible y axis variables for holonomic robots than base_local_planner's DWA.
  - **eband_local_planner:** Implements the Elastic Band method on the SE2 manifold.
  - **teb_local_planner:** Implements the Timed-Elastic-Band method for online trajectory optimization.

One of the problems of using the navigation stack is that is not very flexible, or easy to configure. The current implementations of some local planners in ROS implement the dynamics and kinematics for holonomic robots, but doesn't take into account all possible

non-holonomic robots.

Next section introduces the concept of holonomic and non-holonomic robots, the Ackermann vehicle and its kinematic constraints.

2.6 Car-Like Vehicles

As we mentioned before, this thesis focuses in motion control for non-holonomic systems, more precisely for Ackermann vehicles. In this section the Ackermann vehicle is described to explain the kinematic constraints we take into account to generate the command velocities.

We say a robot is holonomic if all the constraints that it is subjected to are integrable into positional constraints of the form:

$$f(q_1, q_2, ..., q_n, t) = 0$$

Where the variables q_i are the system coordinates. When a system contains constraints that cannot be written in this form, it is to be nonholonomic. Said in other words, a holonomic system is when the number of controllable degrees of freedom is equal to the total degrees of freedom.

Nonholonomic systems are characterized by constraint equations involving time derivatives of the system configuration variables. This usually happens when a system has less controls than configuration variables. A car-like vehicle has two controls, from the driver's point of view the controls are the accelerator and the steering wheel (linear and angular velocities), but it moves in a 3-dimensional space (x, y, θ), thus not every path in the configuration space is a feasible path for the vehicle [4].

We represent this problem as the parallel parking problem, a motion planning problem in mechanics to determine the path a car must take in order to parallel park into a parking space. It shows clearly why car-like vehicles are non-holonomic (i.e. controllable degrees of freedom is less than the total degrees of freedom). The front wheels of a car are permitted to turn, but the rear wheels must stay aligned. When a car is initially adjacent to a parking space, to move into the space it would need to move in a direction perpendicular to the allowed path of motion of the rear wheels.

A well known car-like vehicle is the Ackermann vehicle, which has the Ackermann steering geometry, named after Rudolph Ackermann (1764–1834) who pantented it in 1818. The motion of an Ackermann based robot, like a car, can be described on first approximation by

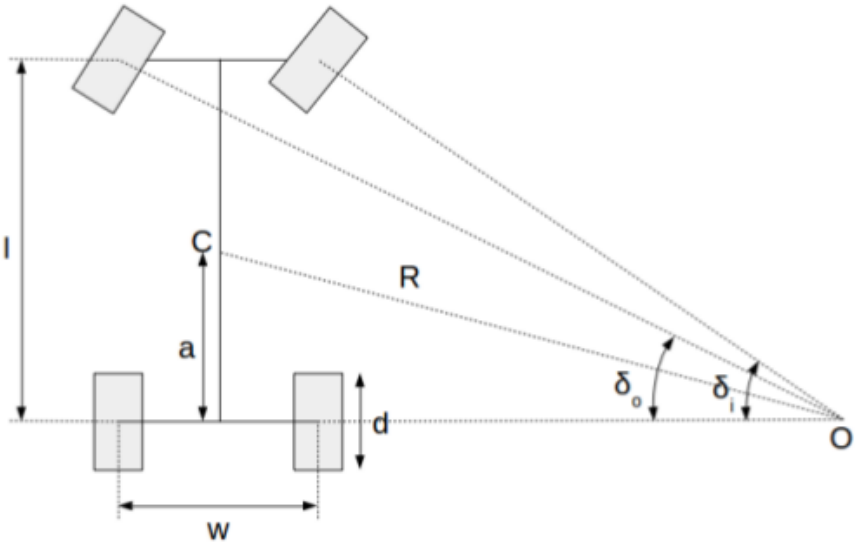

Figure 2.17: Sketch of an ackermann based robot.

the translational speed and the steering angle, and also by the steering speed.

Fig. 2.17 shows a sketch of an ackermann architecture. The important parameters of this kind of configurations are:

- **wheel base (l):** is the distance between the front and rear axles of the robot.
- **track (W):** is the distance between the left and right wheels
- **wheel diameter (d):** diameter of the robot wheel.
- **center of mass (C):** position of the center of mass of the robot, which is considered the point of rotation of the robot.

The inner and outer front wheels have different steering angles to avoid slipping (δ_i and δ_o respectively), and the resulting steering angle for the whole robot (taken at the center of mass) can be computed as:

$$\delta = \frac{cot(\delta_i) + cot(\delta_o)}{2} \tag{2.26}$$

Also, the rear wheels have different angular speeds when the robot is turning (ω_i for the inner wheel and ω_o for the outer one), and the equivalent translational speed for the whole robot (taken at the center of mass) can be computed as:

$$v = d\pi\frac{\omega_i + \omega_o}{2} \tag{2.27}$$

The kinematic model of the vehicle is composed by the following tree differential equations.

$$\begin{cases} \dot{x} = & v \cdot cos(\theta) \\ \dot{y} = & v \cdot sin(\theta) \\ \dot{\theta} = & v\frac{tan(\delta)}{l} \end{cases} \tag{2.28}$$

x Cartesian position along the x-axis

y Cartesian position along the y-axis

θ Orientation of the vehicle

δ Steering angle

v Linear velocity

We name the angle δ the Ackermann steering angle.

The position of an object in a two dimensional space is completely defined by the position of its center of mass (x, y) and its orientation θ, and a trajectory in a two dimensional space is given by a temporal sequence of these variables. Therefore, in order to estimate the trajectory the robot will follow given the translational speed and the steering angle, first it is necessary to compute the turn radius of the center of mass:

$$R = d\sqrt{a^2 + \frac{l^2}{tan^2(\delta)}} \tag{2.29}$$

The distance traveled by a vehicle in a given amount of time Δt for a given translational speed v can be computed as the circular arc. However, the angle of the circular sector is unknown a priori, so the distance is approximated by:

$$L = v\Delta t \tag{2.30}$$

The error of this approximation increases with the speed of the robot and the interval of time considered. However, for small periods and relatively low speeds this error can be ignored.

Chapter 3

Local Planners Implementation

This chapter shows the implementation of the three planners introduced before.

3.1 Implementation Motivations

As previously mentioned, the scope of this work is to implement three local planners making use of ROS framework. `move_base` is a structured node which provides the opportunity to solve navigation problem on robots in an easy way, but, on the other hand, its structure is rigid. It depends on plugins, which are dynamically loadable classes that are loaded from a runtime library. For example the DWA or TEB ROS local planners can dynamically be loaded and used by the navigation stack, but most part of the variables and parameters can not be accessed. However, the main reason is not all planners support Ackermann kinematics, so to be able to apply navigation to a robot with Akermann geometry the decision made was to implement the global and local planners by our own.

3.2 Local Planners Frame

The local planners implemented are stored in a ROS package, the files are distributed according the following structure.

```
package
├── cfg/
├── include/
├── launch/
├── src/
├── CMakeLists.txt
└── package.xml
```

The computation of the command velocities depends on the local planner, but the main structure of the nodes is hold in a similar way: how the costmap is stored and updated, how the global plan is recived and stored, etc.

3.2.1 Costmap

The local planner needs the local costmap to know the environment arround the robot and perform all the calculations properly. By means of the `costmap_2d` package [13], the costmap can be stored in a 2D or 3D occupancy grid and updated properly by sensors data.

To store the costmap a subscriber must be declared first. It is suscribed to the topic where the local costmap data is published: `"/local/costmap/costmap"`. Everytime new data is published to the topic, the callback function `costmap_cb_callback` is called. It receives the data in a message of the type `nav_msgs::OccupancyGrid ::ConstPtr`. It stores the information of the costmap, which must be sent to the local planner. To do so a cost translation must be done, while the message recived stores the costmap with values between $-1 : 100$ the local planner asks for a costmap with values in the range $0 : 255$.

`message`	`costmap`	**state**
0	0	no obstacle
99	253	inscribed obstacle
100	254	lethal cost
-1	255	unknown

Values 0/0 (no obstacle), 100/254 (lethal cost/obstacle) and $-1 : 255$ (unknown) are clear. But the values between 1/1 and 99/253 represent the distance from an obstacle to the robot.

To avoid collisions, the costmap is inflated. An inflation is the process of propagating cost values out from occupied cells that decrease with distance. So as seen in Fig. 3.1, the obstacles are inflated. All cells at a distance equal or less than the robot inscribed radius are for sure making the robot collide (inscribed obstacle); all cells at more distance, but less than the circumscribed radius are possible colliding placements, depending on robot orientation. The cells not in collision danger but close enought to an obstacle are given a value grater than 0, in order to make the path move away from obstacles when evaluating the cost function.

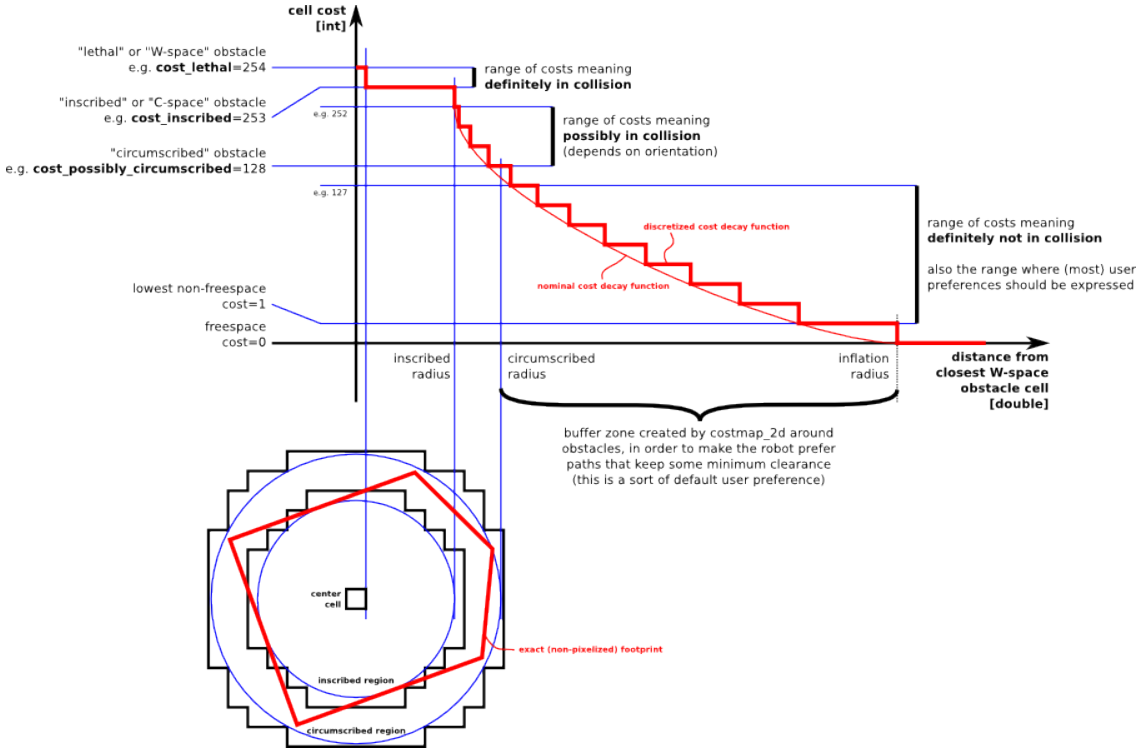

Figure 3.1: Inflation values cost.

3.2.2 Path

The path must be received and stored in order to track it. To do so the subscriber must be declared first. Like the costmap subscriber, it is subscribed to the topic `"/path"`, where data is published. The function called when new data is published to this topic is `path_cb_callback`.

The data received is a message of the type `nav_msgs::Path::ConstPtr`, which is stored in `current_path`. To send then the path to the local planner the path is converted to type `std::vector<geometry_msgs::PoseStamped>`, which is a vector of positions without directions, except the last one, which is the goal we want to reach with a certain orientation.

3.2.3 Odometry

The position, orientation and velocity of the robot are obtained through the odometry topic `/odom`. It is indispensable to have this information, which gives us the actual state of the robot. In our case, an Ackermann vehicle, the position, the steering and the linear velocity.

3.2.4 Footprint

An important information to compute several calculations is the robot footprint (i.e. the area the robot occupy). As this information is constant, at least in our robot, it is uploaded from parameters and stored.

The footprint can be represented by several geometric objects: “point”, “circular”, “line”, “two_circles”, “polygon”. In our case it is represented has a rectangular polygon, with 4 points.

3.3 Dynamic Window Approach

The ROS DWA local planner dwa_local_planner [14] doesn't support the Ackermann kinematics, so the implementation performed must take them into account when generationg the possible trajectories.

3.3.1 Structure

3.3.1.1 Dependencies

The package implemented has dependencies on the following ROS packages.

base_local_planner is the basic local planner to overwrite.

ackermann_local_planner contain Ackermann kinematics.

roscpp is a client library to quickly interface with ROS Topics, Services, and Parameters.

map_msgs defines messages commonly used in mapping packages.

ros_base is a metapackage which extends ros_core and includes other basic non-robot tools like actionlib, dynamic reconfigure, nodelets, and pluginlib.

nav_msgs defines the common messages used to interact with the navigation stack.

nav_core provides the BaseGlobalPlanner, BaseLocalPlanner, and RecoveryBehavior interfaces.

geometry_msgs provides messages for common geometric primitives such as points, vectors, and poses. These primitives are designed to provide a common data type and facilitate interoperability throughout the system.

costmap_2d provides an implementation of a 2D costmap that takes in sensor data from the world, builds a 2D or 3D occupancy grid. This package also provides support for map_server based initialization of a costmap, rolling window based costmaps, and parameter based subscription to and configuration of sensor topics.

The dependencies on this packages are defined on `CMakeLists.txt` and `package.xml` files.

From the ackermann_local_planner we get the necessary interface for the local planner. The ackermann_local_planner depends on base_local_planner as well, inheriting important functions listed here (see the BaseLocalPlanner class in the nav_core package for more details):

- **computeVelocityCommands:** This function is periodically called to get a new velocity command for the robot. This function returns true if a feasible motion command has been found and false otherwise.
- **initialize:** This function is called at construction time to initialize all the necessary parameters of the local planner. It returns either false or true depending on whether the initialization failed or not respectively.
- **isGoalReached:** This function is periodically called to check whether the target position has been reached (it returns true) or not (it returns false).
- **setPlan:** This function is called once for each new navigation target or when re-planning is necessary, immediately after the makePlan function of the global planner returns a valid plan. This function return true or false depending on whether the new global plan could be set properly or not, respectively.

3.3.1.2 Files

On DWA local planner package we can distinguish the following files:

- **cfg/dwa_params.yaml:** It contains all the parametes needed for the local planner.
- **include/costmap_utils.h:** To save the costmap recived from a Publisher.
- **include/Dwa_Planner_configuration.h:** All the variables and parameters are declared.
- **include/dwa_planner.h:** The velocity comands are calculated and sent.

- **include/local_planner_utils.h:** Functions used by the other files.
- **launch/local_dwa.launch:** To launch out local planner node.
- **src/Dwa_Planner.cpp:** Subscribers, Publishers and callback functions declared.

3.3.1.3 Implementation

Two class objects are used form the package ackermann_local_planner:

- **AckermannPlannerUtil** which is a helper class implementing infrastructure code many local planner implementations may need. It was modified in order to not use it as a plugin. Main functions used are:
  - `void reconfigure_callback`
    Sets class parameters.
  - `void initialize`
    Initializes the object to call other functions.
  - `bool get_goal`
    Gets global plan goal.
  - `bool set_plan`
    Sets the path to follow.
  - `bool get_local_plan`
    Gets the local plan to follow.

  AckermannPlanner which is the class implementing the local planner.
  - `AckermannPlanner`
    Constructor for the planner, it has AckermannPlannerUtil as input.
  - `~AckermannPlanner`
    Destructor for the planner.
  - `void reconfigure`
    Reconfigures the trajectory planner.
  - `bool check_trajectory`
    Check if a trajectory is legal for a position/velocity pair.
  - `base_local_planner::Trajectory find_best_path`
    Given the current position and velocity of the robot, find the best trajectory to exectue.

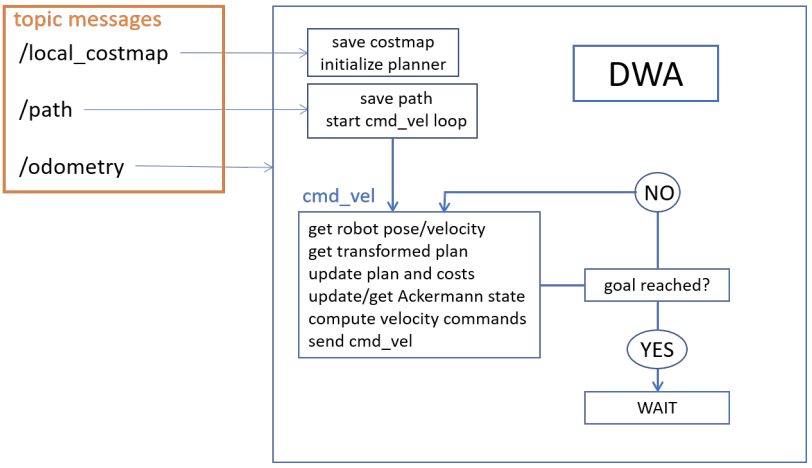

Figure 3.2: DWA control flow.

- `void update_plan_and_local_costs`
  Take in a new global plan for the local planner to follow, and adjust local costmaps.
- `bool set_plan`
  Sets new plan and resets state.

In Fig. 3.2 the flow of the DWA node implemented is shown.

When the node is initialized, it waits for the '/local_costmap' topic to be published. Once that happens, the costmap is stored and the `AckermannPlannerUtil` and `AckermannPlanner` objects are initialized. When the global planner publishes the '/path' topic, the message is stored and the loop to genertate the command velocities starts.

Inside the command velocity loop, the robot position and velocity are deduced from '/odometry' topic. The transformed plan (local path) is actualized (i.e. prune the points already passed and take in the new ones inside the local window). The local cost and plan are then updated from laser information (i.e. obstacles updated and, in case some of them make the current plan infeasible, the plan is updated). The Ackermann state is a structure containing the robot transversal velocity, steer angle and steer velocity current state, which is updated before computing the command velocities. Once they are computed, they are sent to robot servos and the loop starts again if goal has not been reached.

DWA Implementation

As seen in section 2.2 the procedure to generate the command velocities with the DWA algorithm are: to find the window of feasible motion commands in state space, to generate a set

of trajectories within this window, to evaluate that set through a cost function.

Window generation

The first step is to find out which are the maximum translational velocity and steering angle (i.e. the dynamic window) that can be achievable in the given time interval, taking into account that the final translational and steering velocities must be 0. After finding the boundaries of the dynamic window, a finite number of samples will be taken in each of the two dimensions of the window in order to generate all the trajectory candidates.

Given that the number of samples is finite, and in general small in order to reduce the overall computational complexity of the algorithm, it makes no sense to use a fixed time interval to compute the dynamic window because, when the robot is close to a goal, only a small subset of all the computed trajectories will be feasible to reach the goal (those with small translational speeds).

Therefore, a variable time interval is used in terms of the distance to the goal d and the current translational speed v_t of the robot, as shown in Eq. 3.1.

$$T_{sim} = \frac{d}{v_t} \tag{3.1}$$

To compute the boundaries of the dynamic window, the current state of the robot, in terms of current translational velocity and current steering angle and velocity, is needed. Also the dynamic parameters of the robot are needed, that is the maximum translational acceleration and deceleration and the maximum steering velocity, acceleration and deceleration.

If the time required to accelerate to the maximum velocity T_{acc} and to decelerate to a complete stop T_{dec} are smaller than the desired time internal, the boundaries of the dynamic window are the maximum and minimum velocity. However, if the given time interval is not enough to accelerate to the maximum velocity, one or both of the boundaries of the dynamic window must be reduced.

To compute the maximum and minimum velocities:

$$\begin{aligned} v_{max} &= \begin{cases} \frac{T_{sim} acc_{max}}{2} + \frac{v_i}{2} & \text{if } T_{acc} + T_{dec} \geq T_{sim} \\ v_{max} & \text{if } T_{acc} + T_{dec} < T_{sim} \end{cases} \\ v_{min} &= \begin{cases} -\frac{T_{sim} acc_{max}}{2} + \frac{v_i}{2} & \text{if } T_{acc} + T_{dec} \geq T_{sim} \\ v_{min} & \text{if } T_{acc} + T_{dec} < T_{sim} \end{cases} \end{aligned} \tag{3.2}$$

For the steering angle window boundaries, a similar procedure is followed.

Trajectory generation

Once the boundaries of the dynamic window have been computed as explained, it is time to generate a set of trajectory candidates to be evaluated.

The boundaries define a two dimensional subspace with all the feasible values of translational speed and steering angle. Since it is not computationally feasible to evaluate all the possible candidate pairs, an uniform sampling is performed in both dimensions, and a reduced set of candidate pair is generated.

For each pair of steering angle and translational speeds, the resulting trajectory is generated for the desired time interval using the kinematic and dynamic constraints of the robot. Each trajectory is then evaluated with a set of cost functions. The usual costs functions used to evaluate each trajectory are:

- **oscillation:** This cost function penalizes trajectories that would change the motion direction in order to avoid oscillations.
- **obstacles:** This cost function eliminates the trajectories that would collide with an obstacle, either an static one from the map or a dynamic one detected by the sensors.
- **path:** This cost function evaluates the trajectory in terms of how close it is to the planned path.
- **goal:** This cost function evaluates the trajectory in terms of how close the final position reached by the trajectory is to the global (or local) goal.

Each cost function assign a cost to the trajectory, and the total cost assigned to it is the weighted sum of all these costs. Some of the cost functions may discard the trajectory without assigning any cost (trajectories that would collide with an obstacle for example).

Heading cost function

Due to the motion limitations introduced by the kinematic constraints of an ackermann based robot, it is useful to introduce a new cost function to be evaluated. This cost function compares the heading of the robot in several points along the candidate trajectory with the heading of the desired path, and assigns a cost proportional to the angular difference in all evaluated points (the greater the error, the greater the cost).

Once the two closest points are found, a vector representing the slope of each curve is generated by using the current and the previous points, v_{seg} for the global path segment and v_{traj} for the candidate trajectory. With these two vectors, the heading difference for a single point is computed.

$$\Delta\theta = atan2\left(\frac{v_{seg} \times v_{traj}}{v_{seg} \cdot v_{traj}}\right) \tag{3.3}$$

The heading differences at all evaluation points are accumulated and then multiplied by a scale factor.

3.4 Timed Elastic Band

The ROS TEB local planner teb_local_planner [15] supports Ackermann kinematics, the main modification needed is to not perform the local planner as a plugin.

3.4.1 Structure

3.4.1.1 Dependencies

The package implemented has dependencies on the following ROS packages.

base_local_planner is the basic local planner to overwrite.

teb_local_planner_ack is the teb_local_planner package modified in order to use its functions without the need to implement it as a plugin.

roscpp is a client library to quickly interface with ROS Topics, Services, and Parameters.

map_msgs defines messages commonly used in mapping packages.

ros_base is a metapackage which extends ros_core and includes other basic non-robot tools like actionlib, dynamic reconfigure, nodelets, and pluginlib.

nav_msgs defines the common messages used to interact with the navigation stack.

geometry_msgs provides messages for common geometric primitives such as points, vectors, and poses. These primitives are designed to provide a common data type and facilitate interoperability throughout the system.

costmap_2d provides an implementation of a 2D costmap that takes in sensor data from the world, builds a 2D or 3D occupancy grid. This package also provides support for map_server based initialization of a costmap, rolling window based costmaps, and parameter based subscription to and configuration of sensor topics.

The dependencies on this packages are defined on `CMakeLists.txt` and `package.xml` files.

From the teb_local_planner_ack we get the necessary interface for the local planner. The teb_local_planner_ack depends on base_local_planner as well.

3.4.1.2 Files

On TEB local planner package we can distinguish the following files:

- **cfg/teb_params.yaml:** It contains all the parametes needed for the local planner.
- **include/costmap_utils.h:** To save the costmap recived from a Publisher.
- **include/Teb_Planner_configuration.h:** All the variables and parameters are declared.
- **include/teb_planner.h:** The velocity comands are calculated and sent.
- **include/teb_planner_utils.h:** Functions used by the other files.
- **launch/local_teb.launch:** To launch out local planner node.
- **src/Teb_Planner.cpp:** Subscribers, Publishers and callback functions declared.

3.4.1.3 Implementation

One class object is used from the package teb_local_planner_ack:

- **TebLocalPlanner** which is the class implementing the local planner. Main functions used are:

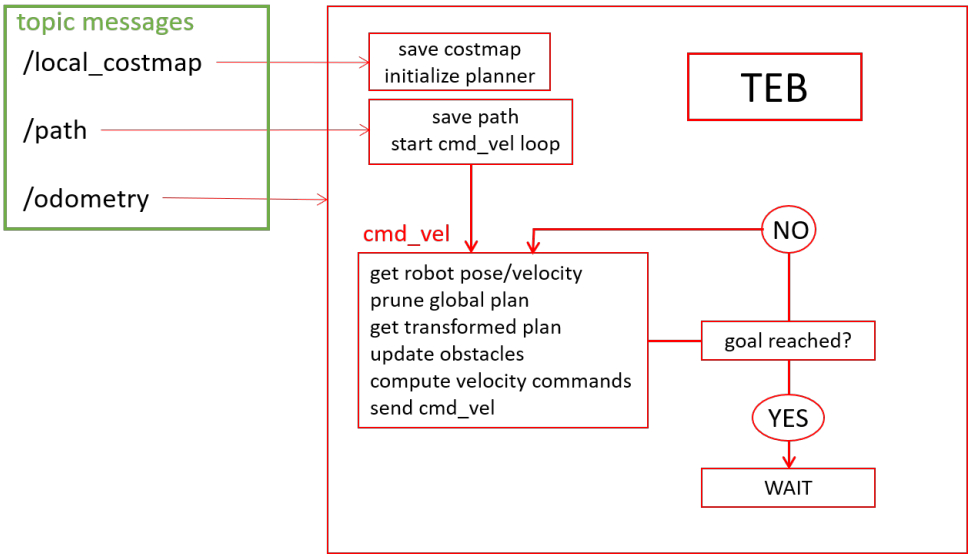

Figure 3.3: TEB control flow.

- `TebLocalPlanner`
  Constructor of the planner.
- `~TebLocalPlanner`
  Destructor of the planner.
- `void initialize`
  Initializes the teb planner.
- `bool setPlan`
  Set the plan that the teb local planner is following.
- `bool computeVelocityCommands`
  Check if the goal pose has been achieved.
- `bool isGoalReached`
  Check if the goal pose has been achieved.

In Fig. 3.3 the flow of the TEB node implemented is shown.

When the node is initialized, it waits for the '/local_costmap' topic to be published. Once that happens, the costmap is stored and the `TebLocalPlanner` object is initialized. When the global planner publishes the 'path' topic, the message is stored and the loop to genertate the command velocities starts.

Inside the command velocity loop, the robot position and velocity are deduced from '/odometry' topic. The transformed plan (local path) is actualized (i.e. prune the points

already passed and take in the new ones inside the local window). The obstacles are then updated from laser information. Then the planner is called, which generates the timed elastic band and optimizes it. The velocity commands are then obtained and sent to robot. The loop starts again if goal has not been reached.

TEB Implementation

Fig. 3.4 shows the control flow of the implemented TEB. In the initialization phase an initial path is recived and transformed to a timed elastic band. At each iteration, the algorithm dynamically adds new configurations or deletes previous ones in order to adjust the spatial and temporal resolution to the remaining trajectory length or planning horizon.

The optimization problem is transformed into a hyper-graph and solved with large scale optimization algorithms for sparse systems which are contained in the "g2o-framework" [16].

The required hyper-graph is a graph in which the amount of connected nodes of one single edge is not limited. Therefore an edge can connect more than two nodes. The TEB problem (Eq. 2.9 and 2.10) can be transformed into a hyper-graph that has configurations and time differences as nodes. They are connected with edges representing given objective functions Γ_k or constraint functions. Fig. 3.5 shows an example hyper-graph with two configurations, one time difference and a point shaped obstacle. The velocity bounding objective function requires the mean velocity which relates to the euclidean distance between two configurations and the required travel time. Hence it forms an edge connecting those states of B. The obstacle requires one edge which is connected to the nearest configuration. The node representing the obstacle is fixed (double circle), thus its parameters (position) cannot be changed by optimization algorithms.

After verifying the optimized TEB, control variables v and δ can be calculated to directly command the robot drive system. Before every new iteration, the re-initialization-phase checks new and changing way-points which can be useful if way-points are received after analyzing short-range camera or laser-scan data.

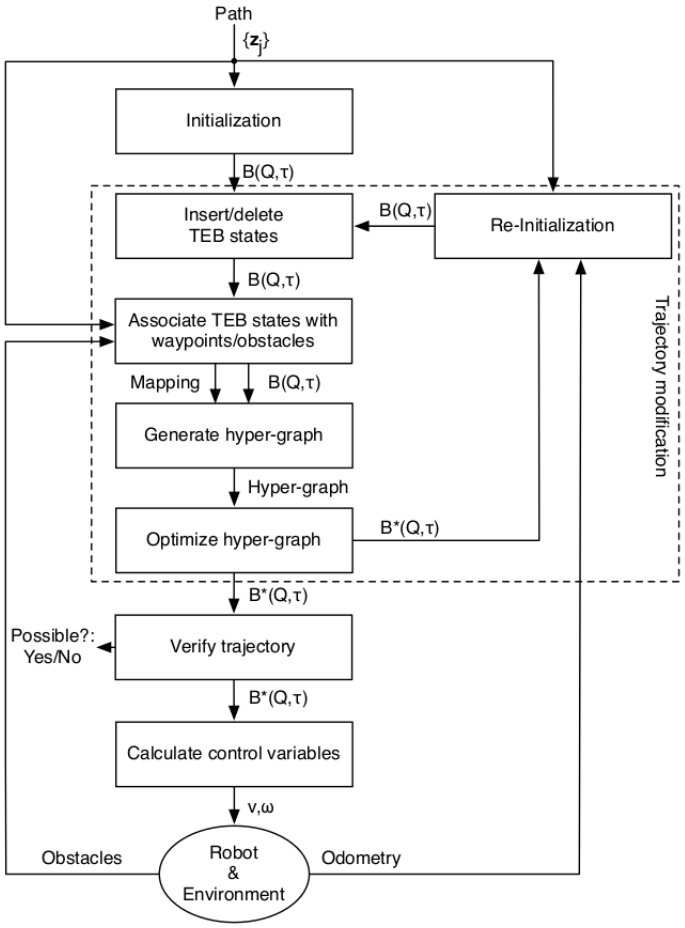

Figure 3.4: Control flow of TEB.

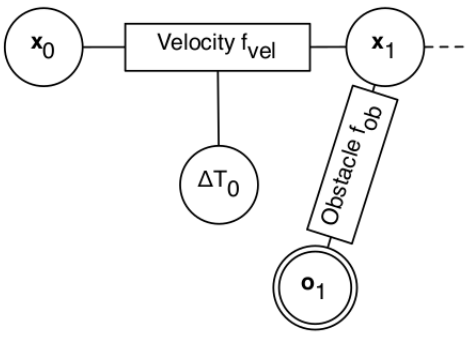

Figure 3.5: Velocity and obstacle objective function formulated as a hyper-graph.

3.5 Model Predictive Control

ROS doesn't include any local planner solving the motion problem by MPC, to implement this planner an MPC problem has been set out and solved.

3.5.1 Structure

3.5.1.1 Dependencies

The package implemented has dependencies on the following ROS packages.

base_local_planner is the basic local planner to overwrite.

ackermann_local_planner contain Ackermann kinematics.

ifopt_ipopt provides a unified Eigen-based interface to use Ipopt (Nonlinear Programming solver) [17].

roscpp is a client library to quickly interface with ROS Topics, Services, and Parameters.

map_msgs defines messages commonly used in mapping packages.

ros_base is a metapackage which extends ros_core and includes other basic non-robot tools like actionlib, dynamic reconfigure, nodelets, and pluginlib.

nav_msgs defines the common messages used to interact with the navigation stack.

geometry_msgs provides messages for common geometric primitives such as points, vectors, and poses. These primitives are designed to provide a common data type and facilitate interoperability throughout the system.

costmap_2d provides an implementation of a 2D costmap that takes in sensor data from the world, builds a 2D or 3D occupancy grid. This package also provides support for map_server based initialization of a costmap, rolling window based costmaps, and parameter based subscription to and configuration of sensor topics.

The dependencies on this packages are defined on `CMakeLists.txt` and `package.xml` files.

3.5.1.2 Files

On MPC local planner package we can distinguish the following files:

- **cfg/mpc_params.yaml:** It contains all the parametes needed for the local planner.
- **include/costmap_utils.h:** To save the costmap recived from a Publisher.
- **include/MPC_Planner_configuration.h:** All the variables and parameters are declared.
- **include/mpc_planner.h:** The velocity comands are calculated and sent.
- **include/mpc_planner_utils.h:** Functions used by the other files.
- **include/mpc_utils.h:** Auxiliary functions used to set out MPC problem.
- **include/MPC.h:** MPC problem class.
- **launch/local_mpc.launch:** To launch out local planner node.
- **src/MPC_Planner.cpp:** Subscribers, Publishers and callback functions declared.
- **src/MPC.cpp:** MPC functions defined here.

3.5.1.3 Implementation

In Fig. 3.6 the flow of the MPC node implemented is shown.

When the node is initialized, it waits for the 'local_costmap' topic to be published. Once that happens, the costmap is stored and the `AckermannPlannerUtil` object is initialized in order to keep tracking of the transformed plan. When the global planner publishes the '/path' topic, the message is stored and the loop to genertate the command velocities starts.

Inside the command velocity loop, the robot position and velocity are deduced from '/odometry' topic. The transformed plan (local path) is actualized (i.e. prune the points already passed and take in the new ones inside the local window). Then the MPC problem is setted out and solved. The velocity commands are then obtained and sent to robot. The loop starts again if goal has not been reached.

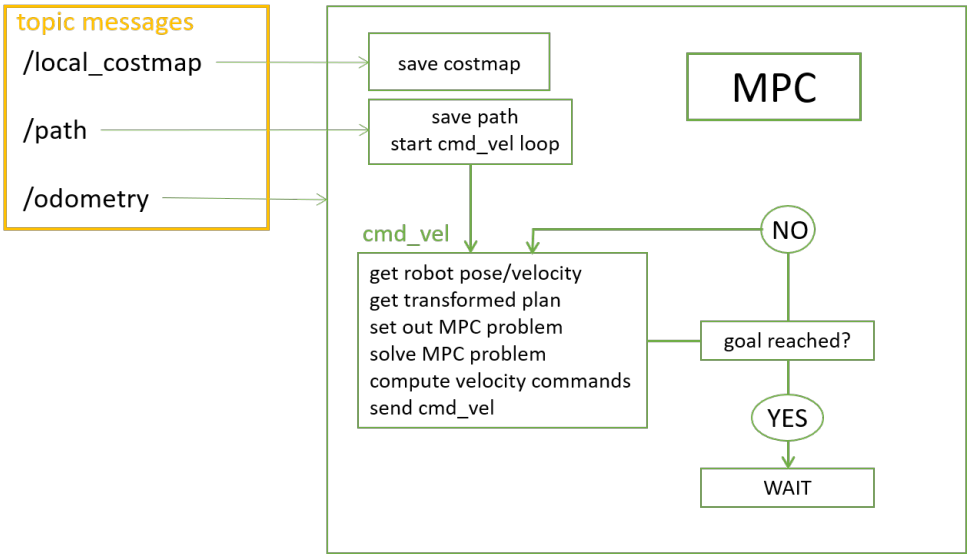

Figure 3.6: MPC control flow.

MPC problem Implementation

As said previously, the MPC problem is solved making use of the Ipopt solver. To do so first it is needed to prepare the problem.

The state of the system $\hat{x}_k$ is

$$\hat{x}_k = \begin{bmatrix} x_k \\ y_k \\ \theta_k \\ v_k \\ cte_k \\ e\theta_k \end{bmatrix}$$

Where cte_k is the cross-track error (roughly the distance from the track waypoints) and $e\theta_k$ is the orientation angle error, both to be minimized in order to keep tracking of the path.

The input of the system will be

$$u_k = \begin{bmatrix} \delta_k \\ a_k \end{bmatrix}$$

Where δ_k and a_k are the steer angle and acceleration respectively.

The kinematic model to predict further states is

$$\begin{aligned} x_{t+1} &= x_t + v_t \cdot cos(\theta_t) \cdot dt \\ y_{t+1} &= y_t + v_t \cdot sin(\theta_t) \cdot dt \\ \theta_{t+1} &= \theta_t + \frac{v_t}{L} \cdot \delta_t \cdot dt \\ v_{t+1} &= v_t + a_t \cdot dt \\ cte_{t+1} &= ydes_t - y_t + (v_t \cdot sin(e\theta_t) \cdot dt) \\ e\theta_{t+1} &= \theta_t - \theta des_t + \frac{v_t}{L} \cdot \delta_t \cdot dt \end{aligned} \tag{3.4}$$

Deduced from (2.28), where for calculation simplifications δ_t is written instead of $tan(\delta_t)$ and $ydes_t$ and θdes_t are the y position and θ orientation desired at time t.

At every iteration to calculate the velocity commands a new MPC problem has to be set out, the procedure is as follows.

The first H points of the transformed plan (local path) are stored into p, where H is the MPC horizon. Then a transformation between frames is applied in order to simplify calculations (see Fig. 3.7):

$$\begin{aligned} r_x &= (p_x - x) \cdot cos(\theta) - (p_y - y) \cdot sin(-\theta) \\ r_y &= (p_x - x) \cdot sin(-\theta) + (p_y - y) \cdot cos(\theta) \end{aligned} \tag{3.5}$$

A translation and a rotation is applied to p to obtain the reference path r, seen from robot frame (i.e. we can consider zero the current x, y and θ of the robot state).

So the initial state of the system is

$$\hat{x}_0 = \begin{bmatrix} 0 \\ 0 \\ 0 \\ v_0 \\ cte_0 \\ e\theta_0 \end{bmatrix}$$

The path to follow r is then approximated by a polynomial f of order 3.

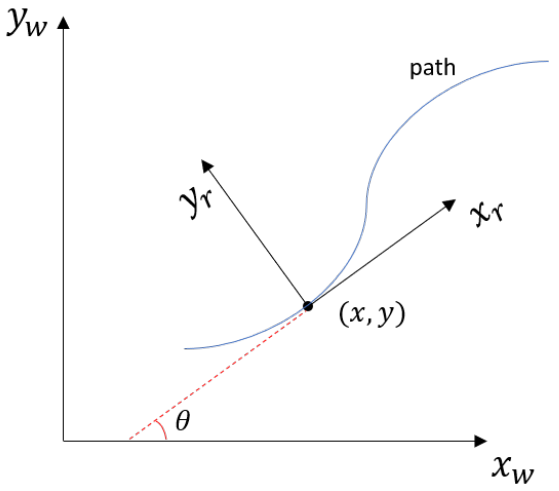

Figure 3.7: Change from World frame to robot frame.

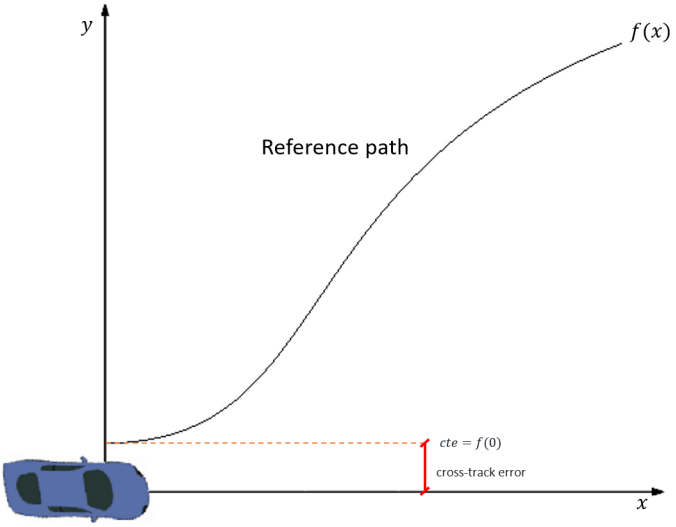

Figure 3.8: Cross-track error approximation.

$$f(x) = k_3x^3 + k_2x^2 + k_1x + k_0 = y \tag{3.6}$$

From which we have the following relation.

$$cte_0 = f(0) = k_0 \tag{3.7}$$

As seen in Fig. 3.8.

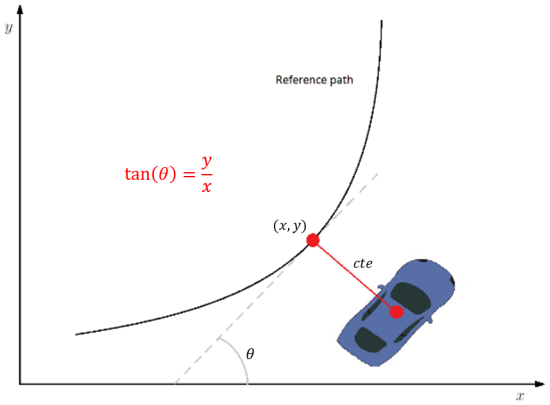

Figure 3.9: Orientation angle error approximation.

If f is derived from respect to x, the result is

$$\frac{\partial f(x)}{\partial x} = 3k_3x^2 + 2k_2x + k_1 = \frac{\partial y}{\partial x} \tag{3.8}$$

From Fig. 3.9 can be deduced then the relation.

$$\begin{aligned} \theta &= atan\left(\frac{y}{x}\right) \\ e\theta_0 &= \theta_0 - \theta des_0 = -atan\left(\frac{\partial y}{\partial x}\right)\Big|_0 = -atan(k_1) \end{aligned} \tag{3.9}$$

Once the initial state is obtained, and the predictions calculated with (3.4) model, the function to minimize will be

$$F(x) = \sum_{k=0}^{H}(cte_k + e\theta_k + \|v_k - v_{ref}\|) + \sum_{k=0}^{H-1}(\|\delta_k\| + \|a_k\|) + \sum_{k=0}^{H-2}(\|\delta_{k+1} - \delta_k\| + \|a_{k+1} - a_k\|) \tag{3.10}$$

Where the first part make the robot to keep tracking of the path with a reference velocity, the second part makes to keep the tracking with the minimum possible steer angle and acceleration, and the third part avoids robot oscillation as the difference between consecutive inputs is minimized (i.e. the changing of inputs must be smooth).

The variables are subject to constraints depending on the vehicle kinematic and dynamic limits, as steer angle and acceleration limits, maximum velocity, etc.

Chapter 4

Results

In this section we present some tests performed with the three local planners. The first tests we make the robot to track different paths without unkown obstacles, to compare the performance following the path. On the last one we add unknown obstacles, to compare the performance avoiding collisions.

4.1 Environment

The map where the experiments are carried out is a map of a real environment. It is $203 \times 217m^2$, with a resolution of $0.25m$, meaning that the grid is conformed by 812×868 cells, to which we refer as Aster map.

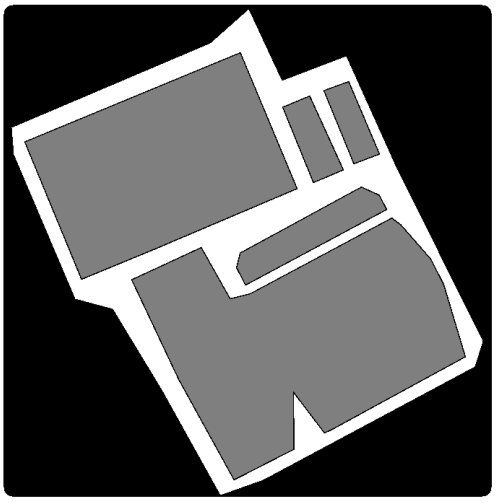

Figure 4.1: Aster environment.

4.2 Simulator

For the simulations we used Stage in ROS, the robot simulator introduced previously.

For its usage first the world must be defined, it contains:

- window: the size, center, rotation and scale of the space must be defined.
- floorplan: for the floorplan a path to the bitmap to be used is specified, as well as its size, and position in the window.
- robot: the model of the robot to be used.

In Stage the mobile robot base models are called position model, our simulated robot was created as a position model and named carlike_robot. The carlike_robot contains three laser models situated in front of the vehicle, on the left and on the right corners. For localization we used a *gps*, since the simulator returns the true global position. To define the kinematics, Stage has three option: *diff*, *omni*, and *car*, the last one was selected as it accounts for the Ackerman vehicle kinematics, it has velocity and steering angle, plus the wheelbase value is set. All the parameters setted are based on a real robot.

Fig. 4.2 shows the view of the simulator using the Aster map. The red box represents the robot mobile base, while the green parts represent the data from the laser sensors.

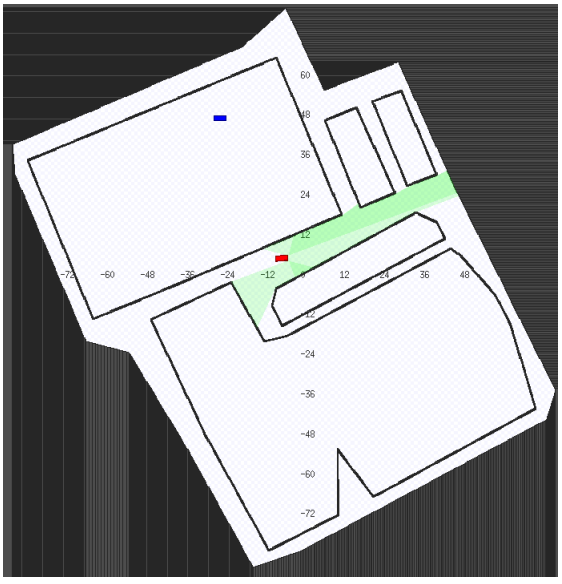

Figure 4.2: Stage Simulator.

4.3 Tests Setup

For setting up the different tests we have written two nodes, `test_local_plan_pub` and `test_local_plan_sub`, the first one publishes the path message, while the second node subscribes to the results data, to be stored and analyzed later on. For the publisher node, a timer and a publisher, for the path, have been defined. The different paths are stored in a rosbag file and sent by the publisher node. The timer activates a flag publish. When the flag is set, the path is charged by the node and published, afterwards, the node stops.

The most important results for us are the computing time and the distance to path and obstacles, the planner nodes are publishing these values inside a `std_msgs::Float64MultiArray` message in a topic called `/results`.

When the path is received by the local planners, they keep track of it. When the robot reaches the goal, the results are published on the topic `/results`. When this happens, the subscriber node takes the information and saves it into a rosbag file.

To plot the information we have used MATLAB, since it has the possibility to easily open and parse rosbag files.

4.4 Path tracking

4.4.1 Test 1

In this Test the robot must follow the path in Fig. 4.3.

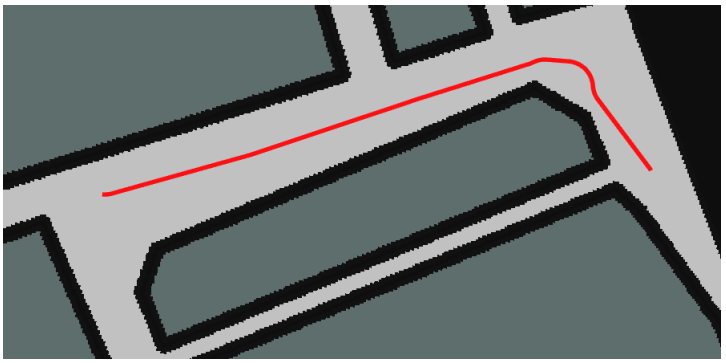

Figure 4.3: Path to follow in Test 1.

In Fig. 4.4 we evaluate the distance from the robot to the path for the 3 different planners.

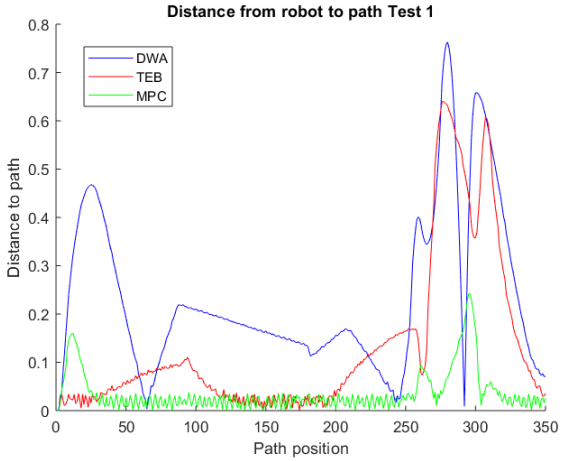

Figure 4.4: Distance from robot to path in Test 1.

It can be seen the planner with more distanc to path is the DWA (Fig. 4.5), while the one following path closely is the MPC. Anyway, the oscillation shown up in the MPC planner is much higher, which means it has a less robust performance. the TEB planner keeps tacking closer than DWA but still it has a more oscillatory behavior.

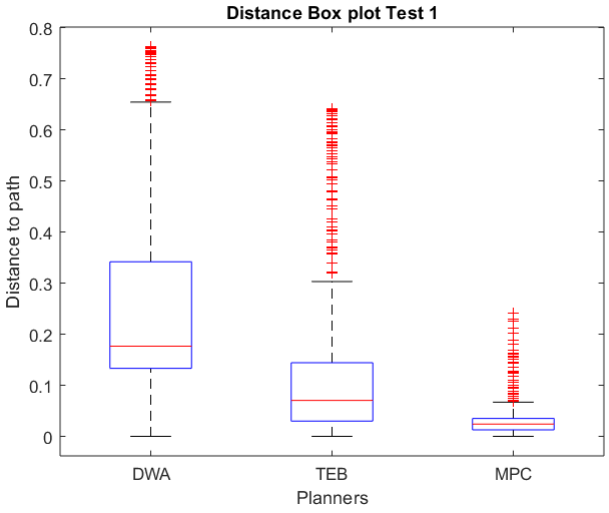

Figure 4.5: Distance Box plot Test 1.

Regarding the time complexity, looking at Fig. 4.6 it can be seen DWA is the planner that takes more loops to arrive to the goal, but the planner taking more time to do so is TEB. It is necessary to emphasize that TEB is the only planner not showing a linear relation between loops and calculation time, the point of slope change is the time in which the robot faces the turn.

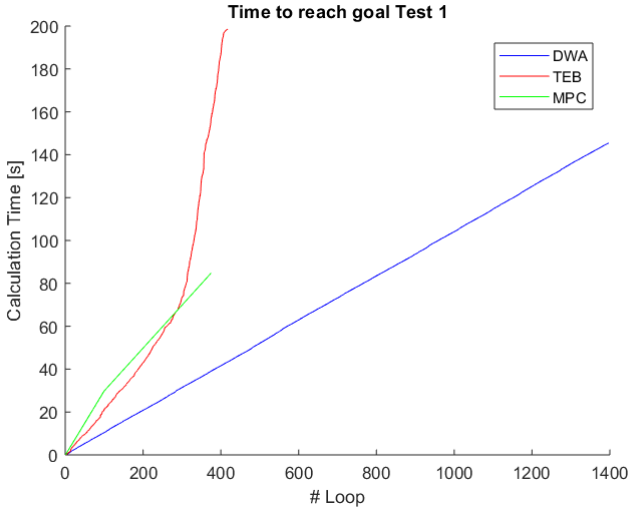

Figure 4.6: Time complexity Test 1.

The last comparison is between the velocity commands sent to the robot (Fig. 4.7). In the y direction velocity it is 0, as an Ackermann vehicle can not move sideways. In the x direction MPC and TEB are able to perform the tracking in higher velocity, the TEB reduces velocity when facing the turn, which is properly way to drive. In rotation velocity, all local planners behave similiarly, though MPC has a more oscillatory behaviour.

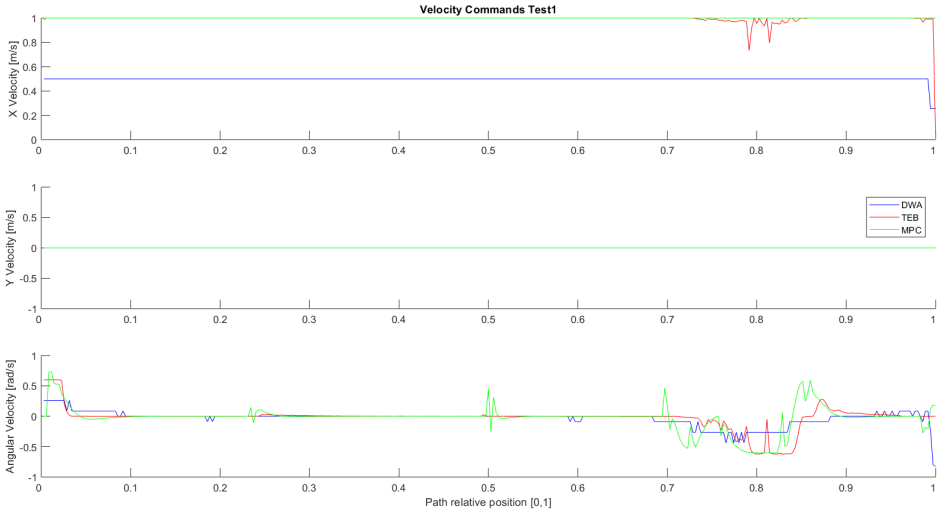

Figure 4.7: Velocity Commands Test 1.

4.4.2 Test 2

In this Test the robot must follow the path in Fig. 4.8.

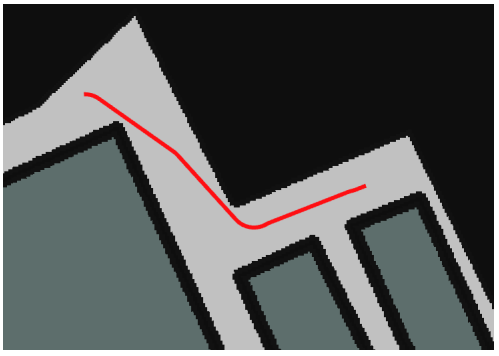

Figure 4.8: Path to follow in Test 2.

In Fig. 4.9 we evaluate the distance from the robot to the path for the three different planners. We can observe that DWA and TEB planners, when facing a turn, cross the global path. As it is an absolute distance, it can be seen because of the 'M' shape on the graphic with the middle point at zero value (arround 250). Instead, the MPC planner keeps on the same side of the global path while performing the turn. MPC is again the planner with the most oscillatory behaviour.

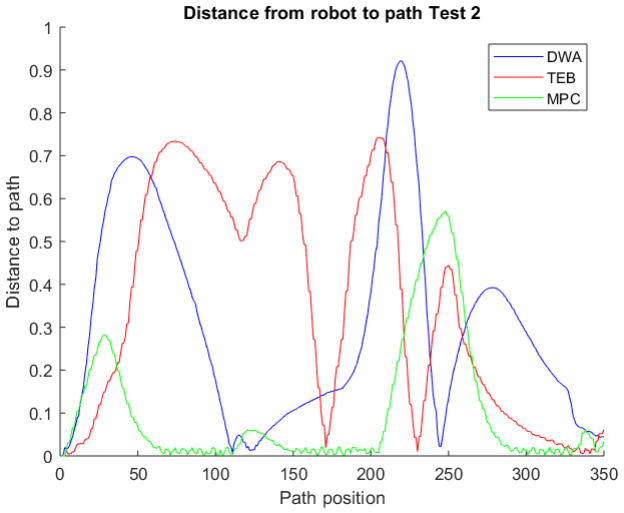

Figure 4.9: Distance from robot to path in Test 2.

In Fig. 4.10 can be seen TEB is the planner keeping more distance from path while following it. Still MPC is the planner with more erratic behaviour.

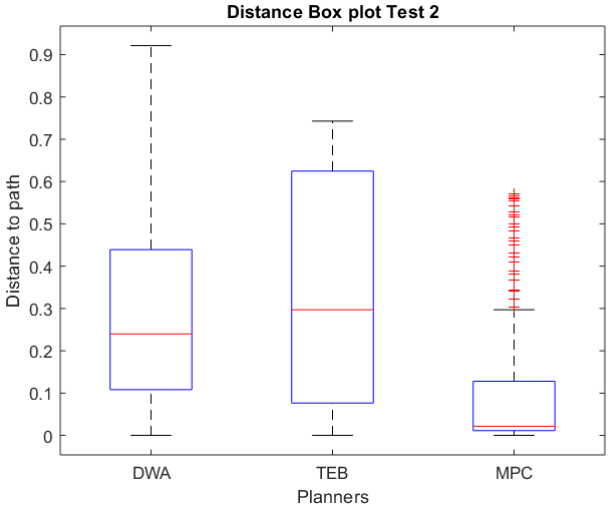

Figure 4.10: Distance Box plot Test 2.

At Fig. 4.11 the results are similar to the ones obtained in the previous Test.

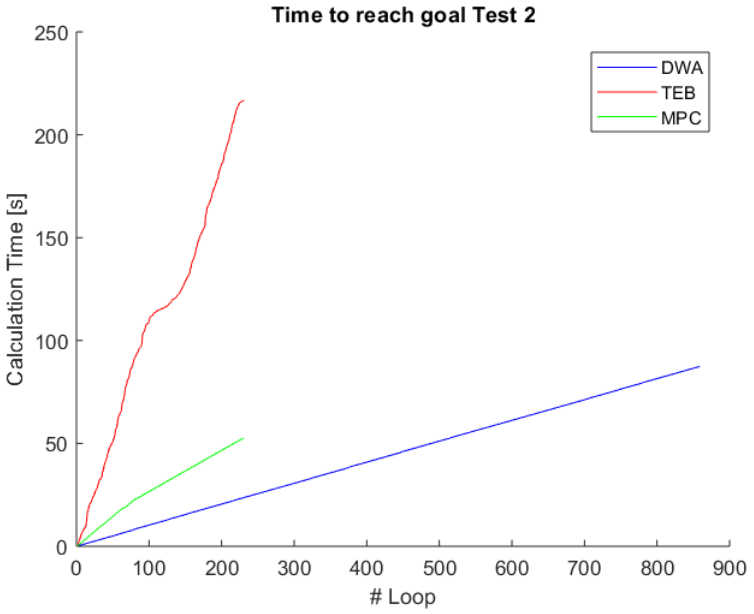

Figure 4.11: Time complexity Test 2.

At Fig. 4.12 we can see DWA is the planner with more stable velocity along the tracking, but its also the one having more problems to reach the goal, as seen on the rotational velocity graphic. TEB performs more pronunciated turns.

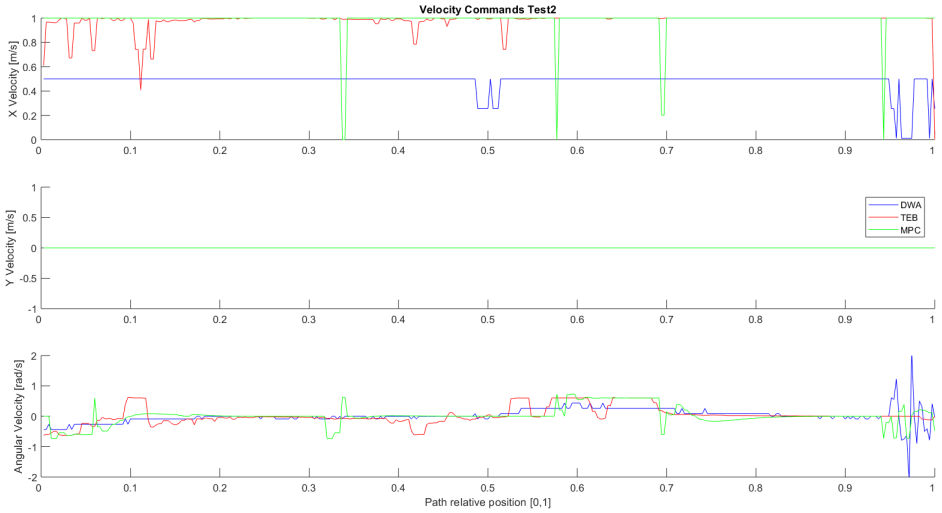

Figure 4.12: Velocity Commands Test 2.

4.4.3 Test 3

In this Test the robot must follow the path in Fig. 4.13.

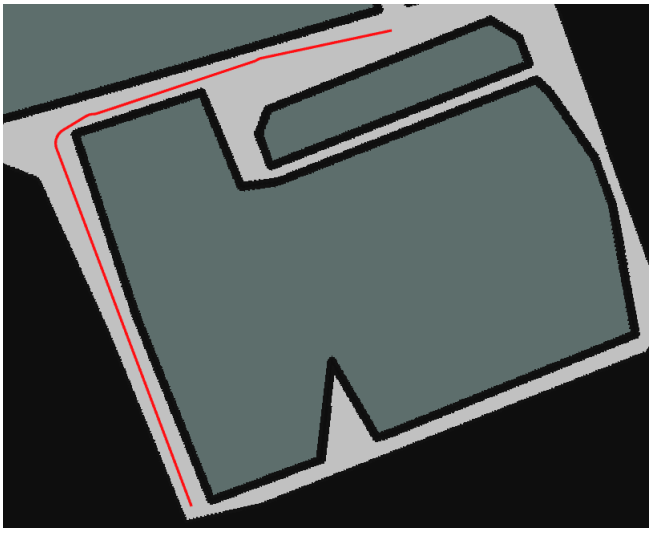

Figure 4.13: Path to follow in Test 3.

In Fig. 4.14 we evaluate the distance from the robot to the path for the 3 different planners. The most robust behaviour is DWA,, while MPC has oscillatory performance.

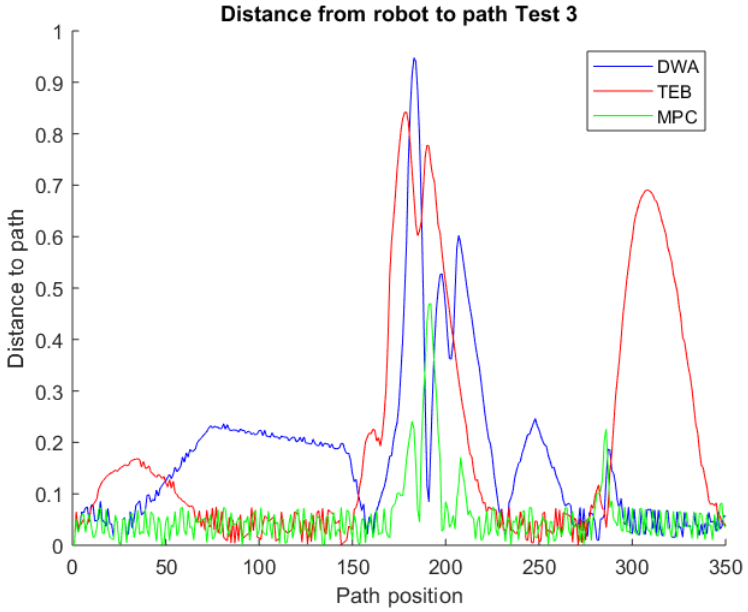

Figure 4.14: Distance from robot to path in Test 3.

In Fig. 4.15 by other hand it can be seen MPC is the planner varying less the mean distance along all the path.

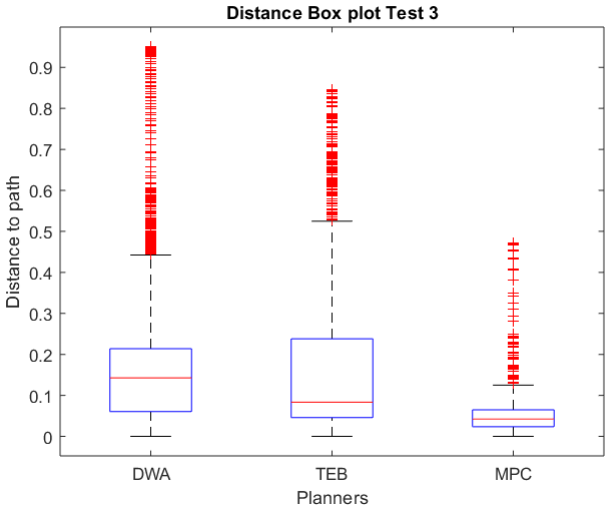

Figure 4.15: Distance Box plot Test 3.

At Fig. 4.11 the results are different from the ones obtained in previous Tests. In this case the planner to take more time to reach the goal is DWA.

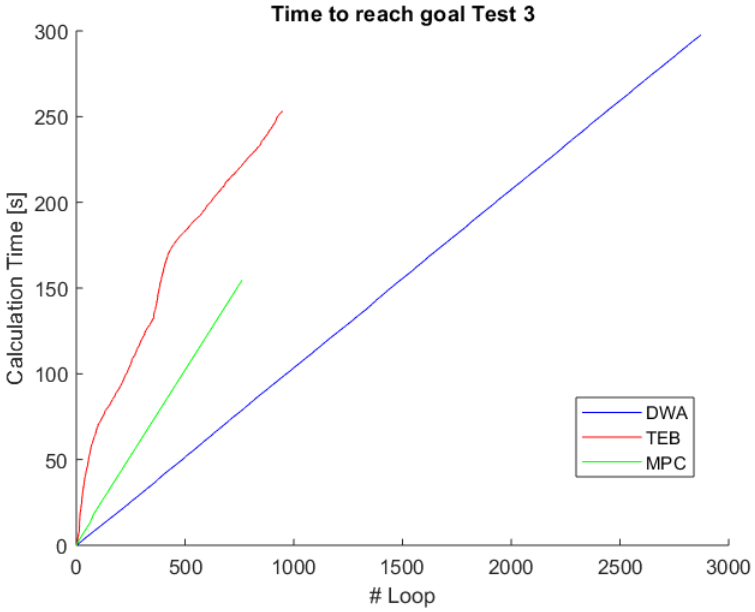

Figure 4.16: Time complexity Test 3.

At Fig. 4.17 we can see DWA is the planner with more stable velocity along the tracking,

while MPC oscillates along all the path.

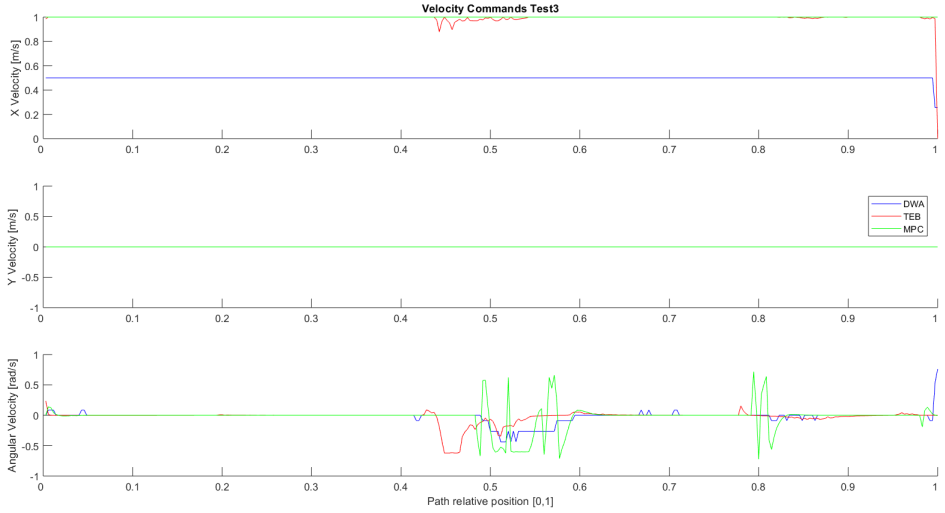

Figure 4.17: Velocity Commands Test 3.

4.4.4 TEB with/without homotopy class

In the time complexity Figures from last Tests, it can be seen that the TEB local planner doesn't have a linear behaviour. The reason is because of the parallel planning in distinctive topologies (homotopy class). This requires much more CPU resources, since multiple trajectories are optimized at once, thus the time complexity increases and, as explained, not in a linear way (as the amount of different homotopic classes depends on the environment).

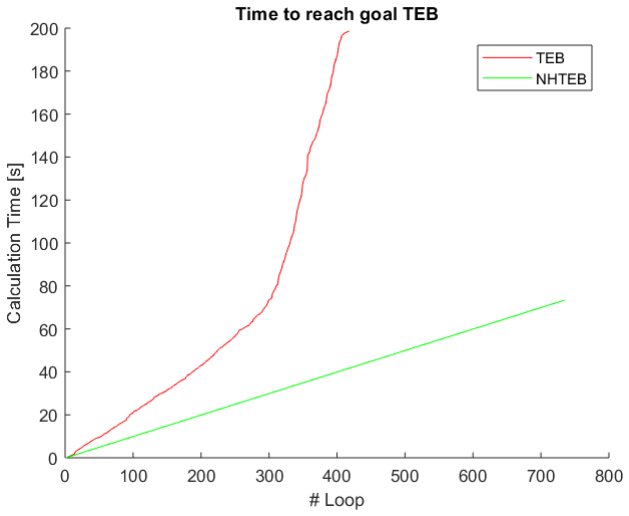

Figure 4.18: Time complexity Path 1 TEB with/without homotopy class.

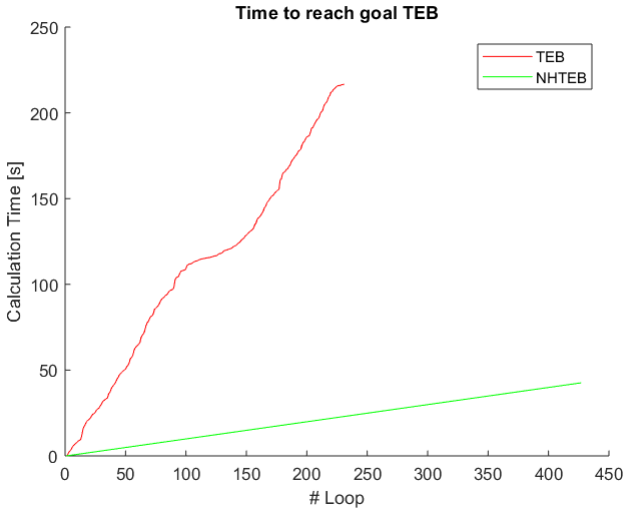

Figure 4.19: Time complexity Path 2 TEB with/without homotopy class.

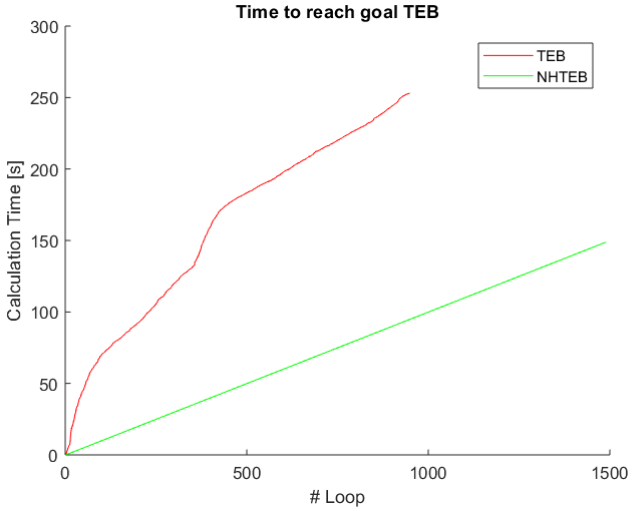

Figure 4.20: Time complexity Path 3 TEB with/without homotopy class.

From Fig. 4.18 to 4.20 it can be seen disabling the parallel planning increases the planner performance in time complexity terms. Specifically, the efficiency increases from 3 to 10 times.

4.5 Obstacle avoidance

This Test compares the distance the DWA and TEB planners hold from an unknown obstacle (i.e. not previously known from global costmap). The MPC planner is not included as it is not able to avoid unknown obstacles.

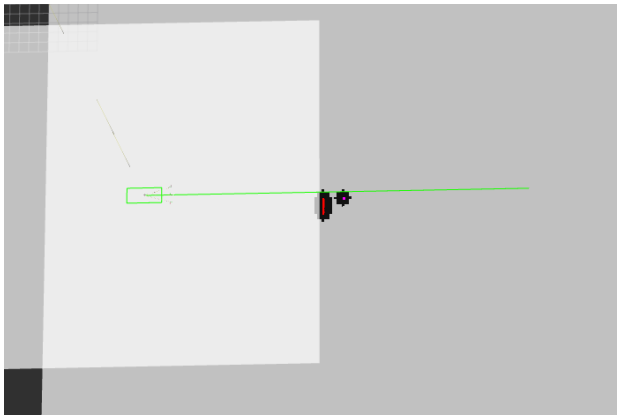

Figure 4.21: Path to track.

To compare them, both planners are supposed to follow a straight path (Fig. 4.21), where in the middle there is an obstacle they must avoid. The distance both planners keep along the path is shown in Fig. 4.22.

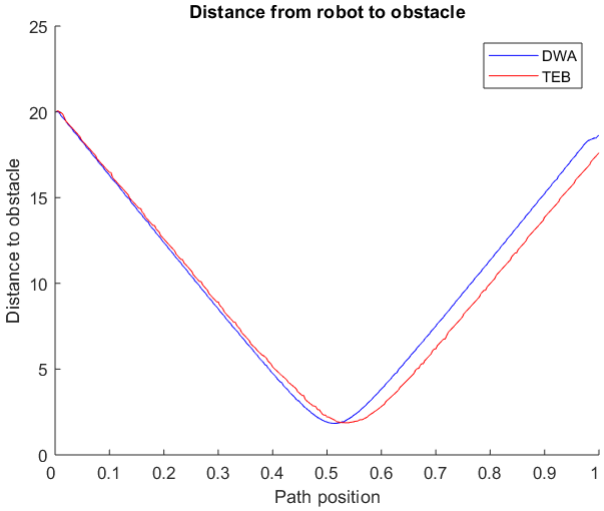

Figure 4.22: Distance to obstacle.

Both planners perform a similar trajectory, avoiding the obstacle. The minimum distance of DWA planner to the obstacle is 1.83 while for TEB planner it is 1.88.

Chapter 5

Conclusions and Future Work

In this work we have implemented three local planners for autonomous vehicles, the first based on the Dynamic Window Approach algorithm, the second on the Timed Elastic Band and the third on the Model Predictive Control.

Even if MPC is the local planner showing less distance to path on all tests performed, its behaviour is more oscillatory than the others and, together with the fact that it is not able to avoid unknown obstacles, it is not a planner that can be used on real robots yet.

Both TEB and DWA have a good performance keeping track of the path and avoiding obstacles, still, DWA is better computationally speaking if TEB uses parallel planning. By the way, TEB without parallel planning has better performance than DWA.

The Ackermann kinematics incorporation to three planners has been successful, generating movements that vehicles with that geometry can perform.

The future work to do should focus on the recovery behaviour, in order to restart the planning if the robot is stuck or the behaviour is too oscillatory. It can be improved also the communication global/local planners, to have a better performance. In the last place, the MPC local planner should be modified in order to be able to avoid unknown obstacles.

Bibliography

[1] T. Lozano-Pérez, and M. A. Wesley. “An algorithm for planning collision-free paths among polyhedral obstacles,” *Commun. ACM*, vol. 22, no. 10, pp. 560–570, Oct. 1979.

[2] D. Fox, W. Burgard, and S. Thrun (1997). "The dynamic window approach to collision avoidance". Robotics & Automation Magazine, IEEE. 4 (1): 23–33.

[3] C. Rösmann, W. Feiten, T. Wösch, F. Hoffmann, and T. Bertram (2012) "Trajectory modification considering dynamic constraints of autonomous robots". Proc. 7th German Conference on Robotics, Germany, Munich, pp 74–79.

[4] J.-P. Laumond, S. Sekhavat, and F. Lamiraux. "Guidelines in Nonholonomic Motion Planning for Mobile Robots", 04 2006, vol. 299.

[5] A. Koubaa, "Robot Operating System (ROS): The Complete Reference (Volume 1)", ser. Studies in Computational Intelligence. Springer, 2016, vol. 625.

[6] Robot operating system. Accessed on 07.01.2018. [Online]. Available: http://wiki.ros.org/ROS/Introduction

[7] nav_core. Accessed on 18.01.2018. [Online]. Available: http://wiki.ros.org/nav_core

[8] J.K. Fredlund, and K.S. Sulejmanovic. "Autonomous driving using Model Predictive Control methods", Lund University, Department of Automatic Control, Box 118 SE-221 00 LUND, 2017.

[9] D. Dougherty and D. Cooper. "A practical multiple model adaptive strategy for single-loop mpc", 2002.

[10] J. Löfberg. "Minimax approaches to robust model predictive control", 2003.

[11] N. Jakobi (1997) "Evolutionary Robotics and the Radical Envelope of Noise Hypothesis", Adaptive Behavior Volume 6, Issue 2. pp.325 - 368.

[12] S. Wilson (1985) "Knowledge Growth in an Artificial Animal", Proceedings of the First International Conference on Genetic Agorithms and Their Applications. Hillsdale, New Jersey. pp.16-23.

[13] costmap_2d. Accessed on 15.01.2018. [Online]. Available: http://wiki.ros.org/costmap_2d

[14] dwa_local_planner. Accessed on 23.01.2018. [Online]. Available: http://wiki.ros.org/dwa_local_planner

[15] teb_local_planner. Accessed on 17.02.2018. [Online]. Available: http://wiki.ros.org/teb_local_planner

[16] R. Kümmerle. et al. g2o: "A General Framework for Graph Optimization", Proc. of the IEEE Int. Conf. on Robotics and Automation (ICRA), May 2011.

[17] Ipopt solver. Accessed on 06.03.2018. [Online]. Available: https://projects.coin-or.org/Ipopt