Athens University of Economics and Business

Degree Thesis — Semester 8, 2014

Racing Robot Path Planning: Algorithmic Design and Experimental Validation

Giorgos Stavrinos 3100171

Supervisors: S. Konstantopoulos,
NCSR Demokritos,
I. Androutsopoulos,
Athens University of
Economics and Business

Contents

Chapter 1

Introduction

Robots that help humans have always been a hot topic for researchers in the areas of Robotics and Informatics. A robot that navigates autonomously fast through obstacles can be really useful in many different situations. If it also searches for the optimal path with respect to time, it can not only offer fast trasportation from one point to another, but also efficiency. There are many emergency, and not only, situations, where time really plays a vital role:

- Rescue operations are a very good example of such emergency cases, where a robot needs to navigate fast and safely from one place to another, while avoiding all the obstacles on its way.
- Factories that need heavy or a big amount of materials transferred from one place to another with efficiency could also find a fast robot-carrier really useful, while at the same time eliminating the possibility of harm to humans from these tasks.

In order to achieve these tasks though, a robot should be able to solve the most vital problem first: finding the optimal path, and following it in the fastest way possible. Solving this major task is the main focus of this thesis, which will be achieved by creating a robot that can race through any track by following the optimal path, trying to achieve the fastest lap time.

Racing robots have been mainly introduced in video games, in the form of racers driven by an Artificial Intelligence algorithm. Even though creating a realistic and effective AI driver is a really difficult task, creating a real physical robot driving autonomously creates a whole new combination of obstacles and ideas to be investigated. A racing robot actually needs to imitate the behaviour of a racing driver in terms of following a path that will allow it to achieve fast laps through different tracks.

Avoiding obstacles and driving between them in the fastest way possible, are only a small part of the problems that need to be solved. The obstacles that have to be overcome in order for a robot to be fast are not only limited to the algorithmic implementation of the path planning process but expand to many researching and practice areas of different sciences, like Robotics, Informatics, and Mathematics.

This thesis is going to investigate and analyse the major factors that make a vehicle fast, and try to implement the optimal behaviour for any situation, with a future goal of using these implementations for robots that can help humans. The structure of this thesis is presented below, based on its chapters:

- Backround: In this first chapter, the literature review takes place, where the state of the art is being investigated. The major methods are selected, which are going to be used in the later implementation of the navigation algorithms.
- Design and Implementation: This chapter analyses the ideas behind each of the implemented algorithms. A total of five path-planning algorithms will be analysed, in search of the fastest one.
- Experimental Methodology and Results: The third chapter of this thesis explains and analyses the methods that were used to evaluate each of the implemented algorithms. Its most important part, though, is the section containing the results of the experiments, which are presented with the help of graphs and tables.
- Conclusions: In the last chapter the achievements of the implementation and experiments are presented, along with future plans based on the current work.

Chapter 2

Background

While reaching its destination is a vital task for an autonomous robot, sometimes it is not enough. The time needed to reach the final point is also important. To achieve a faster point to point navigation, many factors must be considered and analysed. The robot has to follow a path, that will allow it to maintain the highest possible speed, without losing grip or start skidding. Discovering this path generates new problems, though. Evaluating its effectiveness and optimality accurately is the first one, while making the robot always be aware of its current position, is the second vital one. In addition to the above, a formulation of the velocity that provides a measure between highest possible speed and compliance with the path's way points is also needed. In this chapter, I review the relevant literature.

The structure of this chapter based on its sections is the following:

- Racing Line: This section analyses the path the robot must follow, in order to achieve the optimal racing behaviour.
- Velocity Management: This next section explains the necessity of an algorithm to effectively control the robot's speed based on the path described in the previous section.
- Localization: The third section of this chapter investigates the most effective ways a robot should use, in order to know its current position at any moment.
- Robot Navigation: A robot that can localize itself, should not lead to the conclusion that is also able to navigate through its surrounding environment. In this section, the most common navigation algorithms are analysed.
- Honorable Mentions: In this section of the literature review, other important topics on the subject are presented, that were not appropriate for any other section, but also should not be omitted.

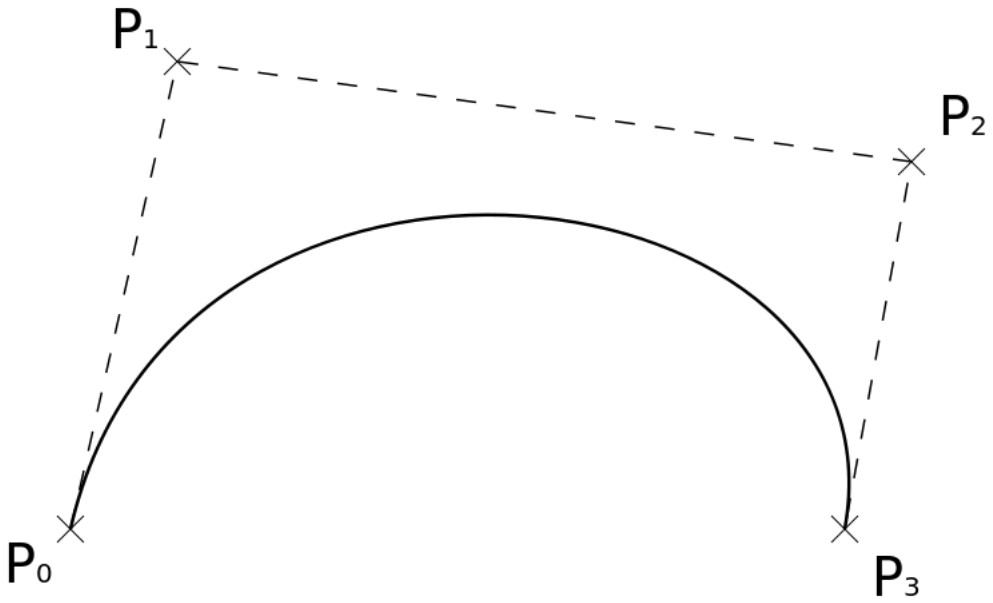

Figure 2.1: An example of Bezier curve given four control points. Taken from *Wikipedia: Bezier Curve* [6]

- Conclusion: In the last section, the basic methods discussed are selected based on their effectiveness, in order to be implemented and validated in the following chapters.

MCP	Grid Search	Difference
70.694	70.694	0
122.544	122.544	0
93.076	93.076	0
25.138	25.138	0
85.686	85.686	0
40.132	39.794	+0.338
33.918	33.890	+0.028
112.984	112.926	+0.058
63.208	63.208	0
76124	76.124	0
75.406	75.406	0

Table 2.1: Data taken from *Searching for the optimal racing line using genetic algorithms* [13]

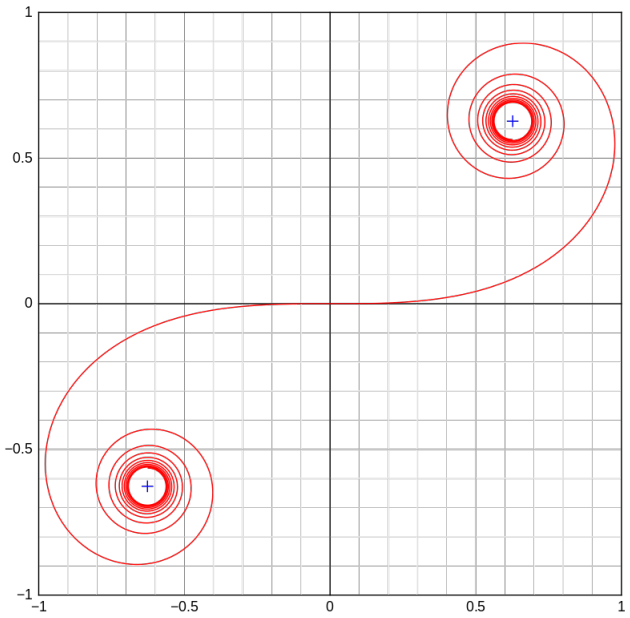

Figure 2.2: Taken from *Wikipedia: Euler Spiral* [7]

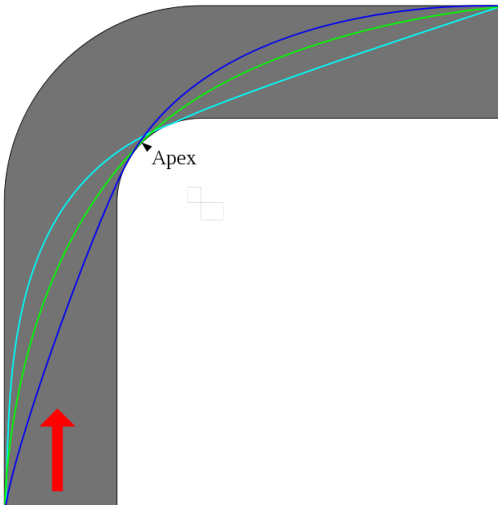

Figure 2.3: Taken from *Wikipedia: Racing Line* [8]

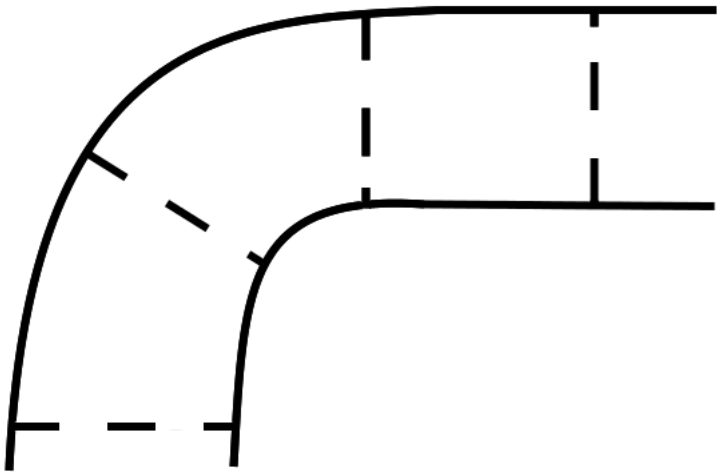

Figure 2.4: The segmentation method applied to a turn

2.1 Racing Line

Formula1 [1]drivers, in order to maintain the highest possible speed around the track, follow a certain, predefined path. This path is known as the *racing line* which allows them to complete the course in the minimum time. Discovering the racing line though, is a difficult and time consuming task. Most racing teams around the world use automated ways to produce the racing line, while others still use professionals to 'draw' the path for the drivers. The manual process of producing the racing line, includes the discovery of the *apex*, also known as the *peak* of each curve of the track, which is the point of the curve that allows the vehicle that goes through it, to follow the straighest line possible. The apex is most of the times the most top point of the curve, and this is the reason of its second name, the peak. The apex of a curve is shown in Figure 2.3.

Many different methods have been implemented to calculate the best racing line, each one using different mathematical methodologies, like Bezier curves [9] and Euler spirals [29]. Bezier curves are used in mathematics to smoothen line segments given their end points. The advantage of using Bezier curves to calculate the racing line is the simplicity of the algorithm, because it is based on a mathematical formula. An example of a Bezier curve is shown in Figure 2.1. This method though, has a big disadvantage: It does not really compute the optimal racing line, but an estimation of it, which in some tracks gets really off. On the other hand, Euler spirals are basically curves that their curvature changes linearly with their length. Although this method creates a good estimation of the optimal racing line, it generates a

new problem which has to be solved: the need to keep only the useful part of the spiral and cut out the end points, so it can be connected with the next Euler spiral to finally form the complete path. A representative example of an Euler spiral is shown in Figure 2.2. The speciality of this method, makes its adaptability extremely low, since when it is included in an algorithm, it must also implement a way to provide only its needed parts and not the whole spiral.

Mathematics provide ways to calculate a good estimation of the racing line, but other ways which are based on algorithms to find the best path based only on its curvature have also been implemented, with optimization between the shortest and the *minimum curvature path (MCP)* [13], or by following strictly the MCP [25]. The minimum curvature path, is a path that is created by focusing solely on the curvature of the path, and trying to minimize it as a whole. In this way, the car-robot following the MCP is able to maintain high speeds even on turns, since their curvature is not big enough to make it brake too much. This method also offers a *preparation* for each turn, making the vehicle following this path able to have the maximum possible speed on the apex of the curve. By finding the shortest path, the minimum distance covered is really implied. This implication leads to the big disadvantage of the implementation of the shortest path: it does not take into account the velocity of the vehicle when creating the path, which creates racing line that cannot be considered fast. By combining the shortest path with the MCP though, the optimal racing line is generated. Computing only the MCP without taking into account the shortest path's way points generates a path with negligible loss regarding the accuracy of the optimal racing line which can substitute the whole optimization between the shortest path and the MCP [25] making the computations faster. Table 2.1 supports this statement, where Grid Search is the the combination of the MCP and the shortest path, while each line represents a different track and each cell shows the lap time using the corresponding path. The last column shows their negligible difference between the lap times of the two paths.

It is clear that not only the combination of the MCP and the shortest path is not much better, but sometimes it produces the exact same result. This does not mean the combination created the same racing line with the MCP, but that there were points of the track that the vehicle was going faster or slower in comparison to the pure MCP racing line, that made the final result equal. We will revisit this point later, where the comparison between different paths is held.

Path-finding algorithms have also been implemented, in which many different paths are created, and in the end the most effective one is used [28]. What these algorithms actually do is search for all the possible paths, and

choose the one with the best score that is given to each one of them via an evaluation method. This method is really time consuming and most of the times, due to its enormous amounts of searching loops, has to be limited to a certain depth of search which might return a path really far from the optimal.

To determine the best path, effective evaluation methods are needed. One of the best ways is to compute the traversal time through the track [27]. In other words, after discovering each path via any A* algorithm (basically searching though a tree graph), it is evaluated with a method that returns a score, which determines its effectiveness based on the total time that is needed for the vehicle (robot) to get from the starting point to the finishing point.

In addition to these methods, a system to use the track data has to be implemented. This system is responsible for scanning the track and discovering its borders, to help in the racing line calculation process. A very interesting and effective implementation suggests creating multiple segments of the track [20] which is very convenient, or applying a grid layer over the map, and treating each cell according to its position in relation to the map[24]. The segment based method, suggests dividing the track in a number of segments and picking a point from each segment. Finally by connecting all those points, the racing line is created. This method seems quite logical and easy for a human that can see the track's map and pick the right segments. A computer though can only base its computations on the distance between the segments, making this method useful for really specific problems. An example of this segmentation is shown in Figure 2.4. The dashed lines represents each segment's borders. These segments can gradually become smaller, in order to produce more way points for the path, and make it smoother. On the other hand, the grid layer based method uses each cell of the grid for that purpose, which is more precise, requires many useless computations but it is much more effective in many different problems. Dividing the whole map in cells, results in a path closer to the optimal, because in contradiction with the segmentation method, each way point is selected from a much wider collection of possible points. Having to handle a bigger set of points, creates the need for more computations in order to achieve the desired goal. Finally, this method's extra computations can be limited when handled correctly, to prevent it from taking too much time in comparison to the segmetation based algorithms and offer both fast and precise results.

2.2 Velocity management

A vital factor in racing, is the speed of the vehicle. To control the robot effectively, a lot of mathematics and physics calculations must be considered. The speed on curves is a really different matter than the speed on straight lines, which has to be calculated precisely so that the robot does not go out of track [22]. Kinematics must also be implemented in the velocity calculation algorithm, based mainly on the physics of the vehicle (robot) [26],[23]. A racing robot must keep its speed as high as the racing line allows, without skidding. With the above methods, provided the necessary variables, like downforce of the robot, the friction between the floor and the tires etc, this can be achieved with just a few calculations. On the other hand, the robot that is going to be used cannot reach very high speeds, so downforce and friction may not be factors that can really affect the robot's path.

2.3 Localization

A basic problem in robot navigation, is for the robot to acknowledge where it is heading exactly in relation to the map. Having a sense of where exactly the robot is located is vital, when the goal is to follow the racing line with a further purpose of minimizing the overall time. This is (partially) solved by creating artificial environments, so that the map generated by the robot, is simpler in terms of obstacles and rough edges [30]. This can be achieved by using card boxes that hide edges that confuse the robot, like chair legs, and avoid floor pumps that create noise on the laser scanner. In addition to specially designed environments, tests can be performed using a simulator like Gazebo, which is designed specifically for robotic testing and includes 3D graphics, model and environment importing, in addition to an accurate physics engine that supports sensor data [21]. Having a good map, is not the only thing that is important though. Using probabilistic Monte Carlo models to find the position of itself is one of the best ways possible, provided the odometry from the robot's controller and the laser scan input [17]. With this method, the robot in a fixed frequency, checks its inputs (odometry and laser scan) and determines where it is located on the map, based on the highest probability.

2.4 Robot Navigation

Robot autonomous navigation is a subject that has always been in the foreground of research on robotics. The path that covers the least distance from one point to another, the *shortest path*, is the most commonly used, so we will use this path as our base, along with any other paths that are going to be tested.

The Robot Operating System's basic global planner, which we will discuss in detail later, uses the shortest path as a default too, which is very convenient for our experimentations, since with minor configurations the desired result is going to be achieved. Some other navigation algorithms have been implemented, but most of them are either a dummy implementation, or just an experimentation for future use.

The idea of a real physical robot and not just a computer, planning its path autonomously for racing purposes at least as a beginning, has not been investigated enough, leaving us a wide area to explore, and a lot of obstacles that have to be overcome.

The planning methods, limitations and architecture of an autonomous system are discussed in much more detail later in this thesis, with a focus not only on path planning, but obstacle avoidance too.

2.5 Honorable Mentions

Many of the mentioned papers used racing simulators or were created in order for their AI to take part in a challenge or tournament. Some of these are described below.

2.5.1 The Open Racing Car Simulator (TORCS)

TORCS is an open source video game racing simulator available for every modern operating system (Windows, Mac and Linux) which started being developed in 1997. This simulator is one of the most accurate publicly and freely available, that allows AI implementation for drivers, but also many mechanical configurations. There is an annual competition in TORCS, where programmers from all over the world use their best AI driver to compete with each other.

2.5.2 DARPA Challenge

DARPA challenge is an annual robotic challenge supported by the US Defence Advanced Research Projects Agency, where robots created by researchers race each other on a usually off-road track. DARPA Challenge has two main challenges.

- DARPA Grand (or Urban) Challenge, which is the classic off-road racing challenge.
- DARPA Robotic Challenge, which is an addition to the classic challenge, where robots do not only race each other, but also have to

complete some tasks in order to win, such as open and close valves, remove obstacles from the road, open and close doors, or help people in disaster and emergency scenarios.

2.6 Conclusion

Methods to solve the many different problems of a fast robot have been implemented as solutions for other tasks, which can be used with minor changes to our problem. In the following list, the methods that were concluded as the best are presented:

- Racing Line Calculation: In order to calculate the racing line, the minimum curvature path will be computed, because it is really fast and its loss of the optimal path is negligible. The MCP is so close to the best (optimal) racing line, that in many situations (tracks) it is actually the optimal path [25]. The MCP will be calculated by selecting points on segments within the map that produce it, and connecting those points using natural cubic splines, to avoid discontinuities. [11].
- Racing Line Evaluation: The evaluation of the racing line is not necessary, provided that the method of the MCP calculation is used, because, there is only one MCP for each set of points (track).
- Map Data Management: To determine the track's boundaries the grid layer based algorithm will be used, as it is the only effective way to do this task. In addition to the grid layer, as the algorithm of the calculation of the MCP requires the clusterization of the map, the method based on segments will be used, because it is effective and not intensive in terms of CPU computations [20]. On each segment, one point will be chosen as the best to form the MCP. The segments will gradually increase in number and decrease in size, to achieve a more precise result.
- Velocity Management: The maximum velocity that the vehicle can reach without skidding on each point of the path, will be calculated as

$$Vmax = \sqrt{\mu\rho(g + \frac{F\alpha}{m})}$$

where Vmax is the maximum velocity, μ is the tire-road friction coefficient, ρ is the curvature radius, g is the gravitational acceleration, m is the mass of the robot and Fα the aerodynamic downforce which can be calculated as

$$F\alpha = \frac{1}{2}Ac\rho v^2$$

where A is the surface that receives the air pressure, c is an aerodynamic coefficient, ρ is the air density value and v is the velocity of the robot at the exact previous moment. [13].

Regarding the specific problem to be solved, and the characteristics of the robot that will be used, the friction coefficient μ and the downforce Fα have a negligible impact on the high speed driving while following the racing line. The robot cannot reach high speeds, actually it would be considered slow for a race, as its *motor maximum speed capabilities*, which is the maximum speed the motors can reach, are only a bit faster than the walking speed of a human. With that being said, the downforce can be eliminated in the equation, and the coefficient friction will be considered as μ=1, because the robot's tires do not let it spin on the ground that will be used. So the final equation is in this form:

$$Vmax = \sqrt{\rho g}$$

where it is clear that the curvature of the racing line is the only factor that affects the maximum speed.

- Localization: Both methods described in the localization section will be used to help the robot know its location at any time. A specially constructed space will be used, so that a good map can be created based on the laser scanner input [30]. While autonomously navigating, the robot will probabilistically calculate its position based on Monte Carlo models [17]. All the above, can be achieved using the localization utilities provided by the *Robot Operating System (ROS)* [4], an open source project that is used as a middleware, which offers libraries and drivers for robot applications, under BSD license. In addition to ROS utilities, *Gazebo* simulator [2] which is fully compatible with ROS, will help me create many test environments, to achieve the best results from the algorithms implemented.

In conclusion, making a robot navigate at high speeds and maintain these speeds is a task that requires many different factors to be taken into account, from many different sciences. However, the combination of the knowledge gained from the state of the art with research and experiments, is going to provide the desired goal.

Chapter 3

Design and Implementation

3.1 Introduction

The research, design and implementation of an effective navigation algorithm is a challenging and time consuming task. It requires both knowledge of the already implemented methods and tools, and an innovative-resourceful mind.

The Robot Operating System (ROS) is one of the most robust ways to program and communicate with a robot, since it provides all the necessary modules and drivers. Of course, ROS cannot solve the navigation problem that we are investigating by using only is default nodes. With its solid architecture and some modification to the existing modules, an effective racing-navigating system can be implemented. For this to happen though, the requirements of such a system need to be stated so that it is clear what has to be designed, before the actual way to do it (programmatically and logically).

This chapter covers all the above subjects and its structure, based on its sections, is the following:

- Requirements: This section presents the basic requirements a fast navigating robot must meet in order to be effective.
- Architecture: In this section, the architecture of the tools used is analysed as an attempt to minimize their complexity. A detailed graph of modules and nodes along with the major messages tranferred between them aids to the comprehension of the basic architecture of the system.
- Modified ROS Nodes: In the third section of this chapter, the Robot Operating System nodes that have been modified are presented and analysed, so the reader can understand the idea behind the implemented algorithms.

- Conclusion: The concluding section, summarizes the chapter, and connects it to the next one, which includes the experiments based on the implementations.

3.2 Requirements

A racing car, and also our robot, in order to race through a track has to be able to meet a set of requirements, which are the following:

- Race through a track without any collisions.
- Achieve the fastest lap times.
- Always race through tracks in the optimal way, with respect to speed.

All the above requirements are equally important for a good racing robot but generally for robots that need to go fast from one point to another. Before trying to solve all these problems, we first need to fully understand the architecture of the modules and tools that are going to be used, which are analysed in the following sections.

3.3 Architecture

Figure 3.1 shows the main architecture of the move_base node of ROS [19], which is the basic module that controls the motors of the robot. Move base consists of many nodes, that their use and structure are analysed in the following paragraphs.

The move base node is the controller of all the nodes inside it. The process of communication between these modules is done via messages. Let's first look into the messages move base uses as an input and output. It is clear from Figure 3.1 that move base has five input and just one output messages. The output message is the velocity determined by the module and its nodes, while the input messages are:

- A PoseStamped message that includes the goal point relatively to the map's coordinates.
- Sensor transforms, which are messages from the localization node, regarding the sensors of the robot.
- Odometry messages, which are messages from the localization node too, regarding the motors of the robot.
- A map message which comes from the map_server node, and includes the map that was created through the mapping process or manually by a human.
- Sensor messages, like laser scans or camera data, that are generated from the robot's sensors.

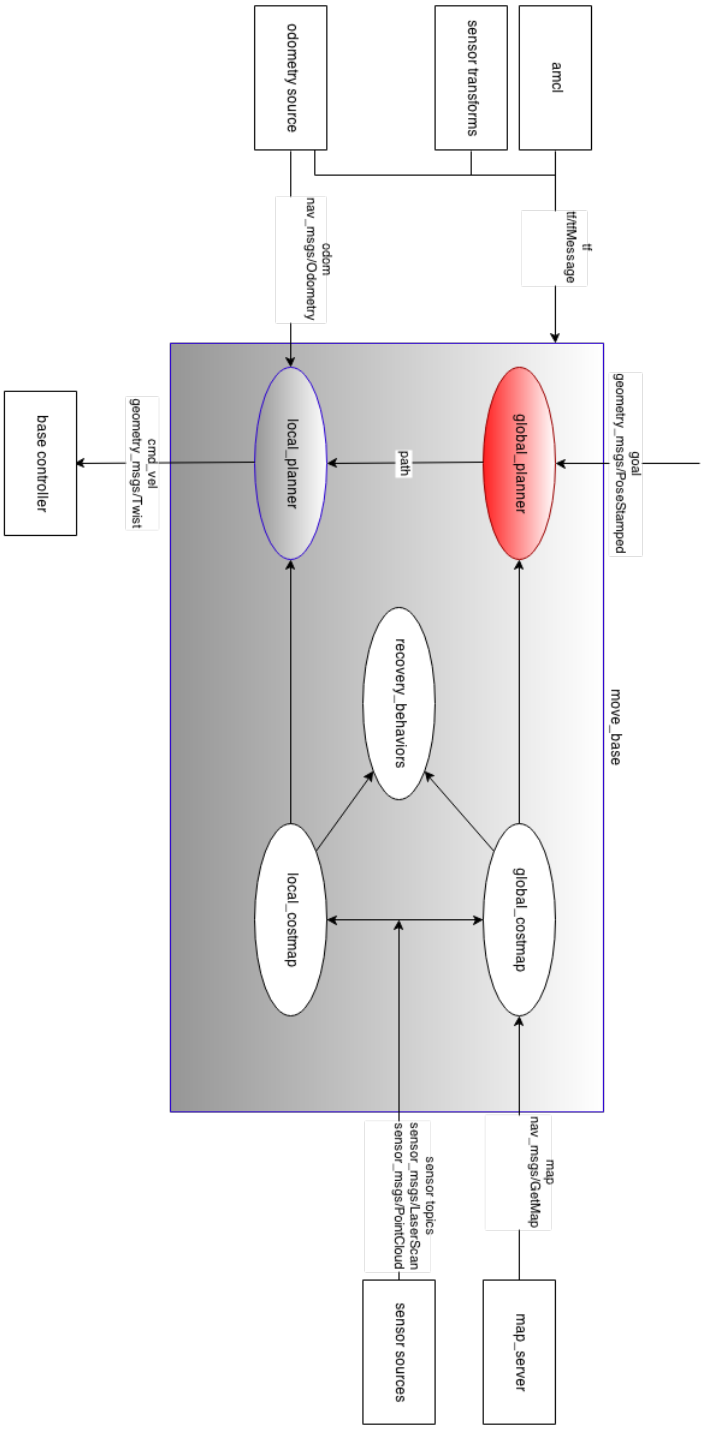

Figure 3.1: Based on move_base ROS wiki page [19]. Shapes with black outline are used without changes, blue outlined and grey-gradient nodes are used with minor changes, and red outlined with red gradient nodes are used with heavy changes.

The move base module consists of five nodes:

- The global_costmap node, that gets the map and sensor data as inputs, and creates a costmap of the given track. In general the costmap is created based on the map and the size of the robot, by assigning values to each map cell that resembles the probability of the robot to crash. These values are inside the space [0,255] where 0 is a free-safe cell, and 255 is a cell inside an obstacle. Figure 3.2 contributes to a better understanding of the costmap value designation.

- The local_costmap node that gets only sensor data as an input, and creates an on-the-go costmap, based only on what the available sensors scan. The local costmap is used for local planning, and basically for obstacle avoidance.

- The recovery_behaviours node, that gets the global and local costmaps as inputs, and tries to unstuck the robot in hazardous cases, like crashes.

- The global_planner node, that creates a plan based on the starting position of the robot and the global costmap. It generates a path that disregards obstacles added after the mapping process.

- The local_planner node, uses the global planner's path and computes the maximum velocity of the motors based on the next points of the path, which is the output of the whole move base module.

What is interesting for the fast navigation problem and its solution though, is mainly the global and local planner in addition to the global costmap. The local costmap holds no interest, because there are no plans for obstacle avoidance outside of the obstacles found during the mapping process.

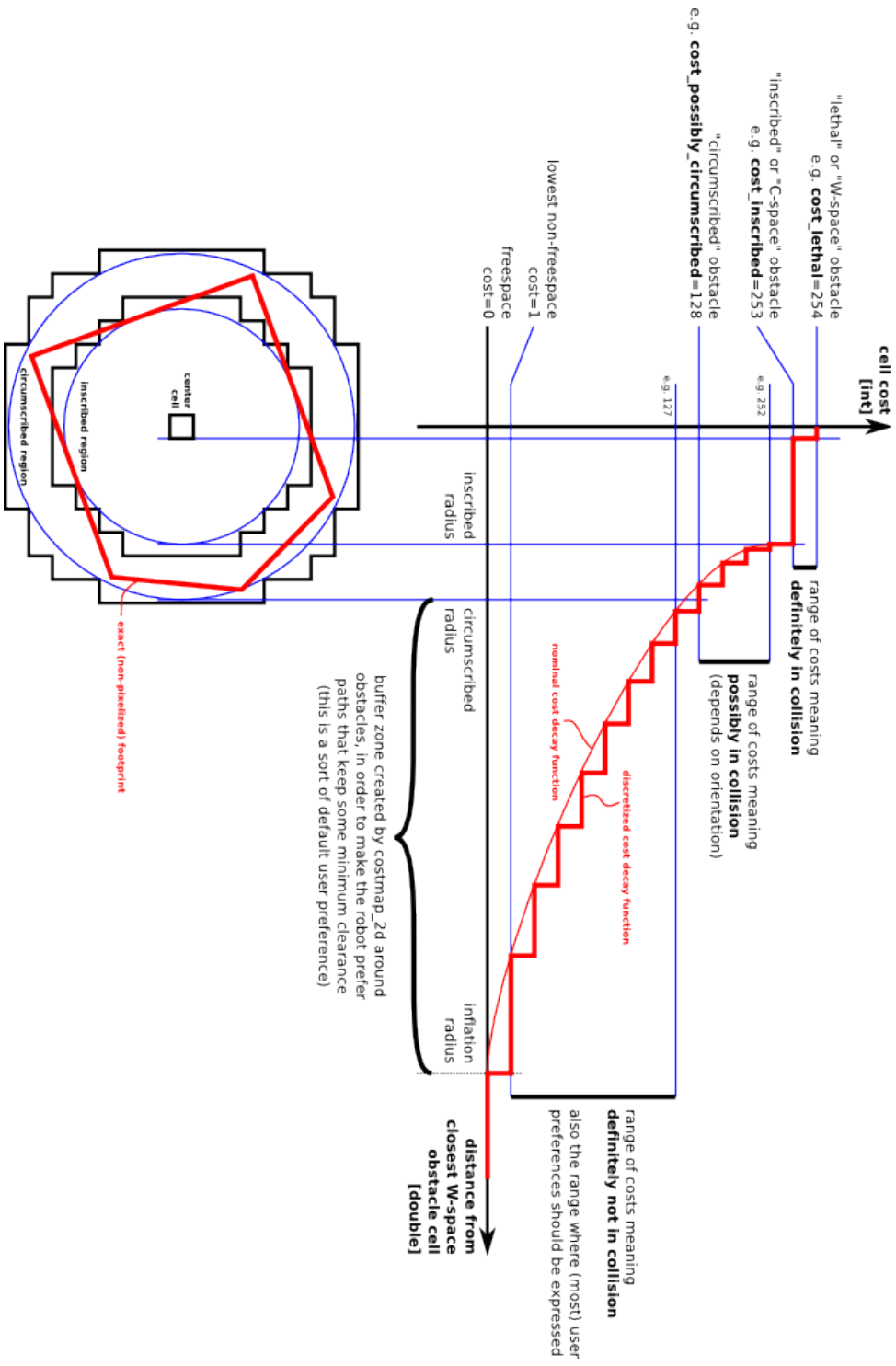

Figure 3.2: Costmap values inflation taken from the official costmap 2D ROS wiki page [10].

3.4 Modified ROS Nodes

The design of the final algorithms is simple, yet useful and innovative. In the following sections the modifications of each node are presented.

3.4.1 Global Costmap

The global_costmap node is used without any modification, because as it is described in more detail in the following section, it is used as it is produced by this node, and it is modified inside our global planner.

3.4.2 Local Planner

As a local planner a modified version of the TrajectoryPlannerROS is used, which is also provided by ROS [12].
Many times, races require the drivers to go through the track more than once, known as laps, so we need our robot to be able to do that too. The global planner passes a vector of points (PoseStamped messages) to the local planner, so a very easy solution to this problem, is to just append the vector with itself as many times, as the laps that are needed. This solution has the disadvantage of the memory needed to save such a big vector, but generally the presentation of the robot's racing skills will be limited to a maximum of 5 laps, so that is not really a problem specifically for our situation.

3.4.3 Global Planner

The implementation of the global planner was based on the NavFn algorithm [18] that comes with ROS. In short, NavFn uses Dijkstra's algorithm, to calculate the shortest path between two points, while avoiding obstacles like walls, using the costmap that is created upon the initialization of the navigation node of the robot.

Now that our robot can actually race more than one lap, given a plan, let's see how the plan is calculated. The problem with NavFn's basic usage, is that in our specific problem, the starting and ending points are the same, so there is no actual shortest path for it to calculate. Even if a point a little bit behind the robot is added, NavFn will find the path that connects the two points so the robot will just turn 180 degrees and go a bit forward. What is needed is a way to tell the robot to ignore the path behind it, and find the shortest path that connects the two points by driving through the entire track. For this purpose, we need to modify the costmap, so that an invisible wall is created behind the robot, between the starting and ending point.

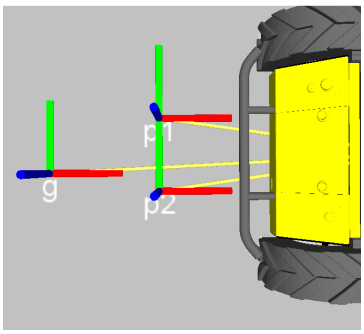

Figure 3.3: The three invisible transformation points

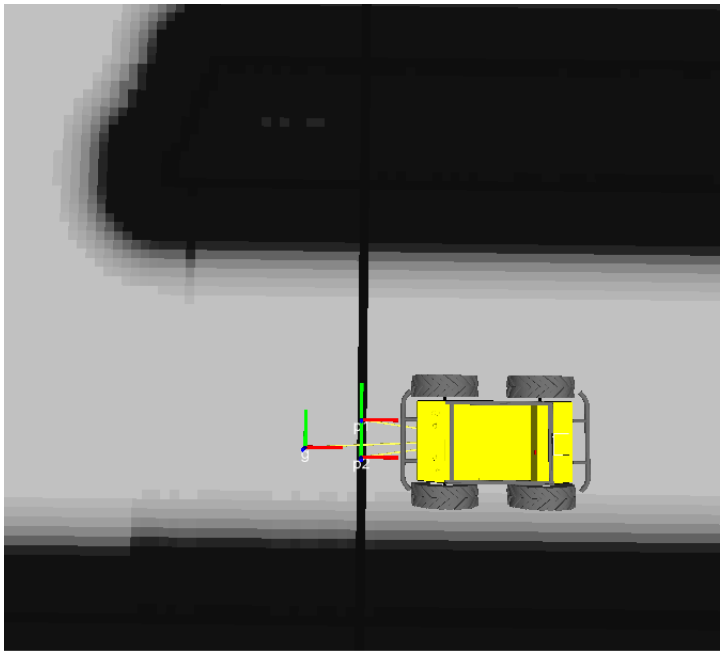

Figure 3.4: The invisible wall created based on p1 and p2

In order to create this invisible wall though, two points that define a line segment that cross both track's borders are needed as well as a point to be used as an ending point. In order to achieve this, three invisible transformation points (tf) were created, that always follow the robot. In Figure 3.3, those points are shown, where those three points are shown, at the back of the robot, where p1 and p2 are used to define the line segment which when extended until it crosses both track's borders, will be used as an invisible wall, and g is the goal point. Now that the needed points are in place, the modification of the costmap for the invisible wall is needed.

The implementation of the invisible wall was based on the graph shown in Figure 3.2. The goal was to find the cells that the line segment defined by the two invisible tf points crosses, and assign them a value of 255 so that it blocks the robot's path. Figure 3.4, which is a screenshot from rviz, the visualization tool of ROS [14], shows the costmap of a track in a specific point, and the invisible wall created in the initialization process of the modified NavFn global planner.
All the above made the NavFn algorithm able to find the shortest path for any track, while also taking into account the direction the robot is facing, since during the initialization of the global planner the costmap is modified to create the invisible wall behind the robot.

In addition to the NavFn (shortest path) algorithm, four more paths were implemented. The idea behind these paths is that the shortest path in many cases is not the fastest one, especially for robots that are able to achieve high speeds. Our robot has a motor maximum speed of 0.6m/s which is in no way considered high, so the shortest path seems to be the best choice. The four new paths were designed in order to achieve better lap times than the shortest path with our slow robot. The first three are actually very similar and gradually closer and more similar to the shortest path. The first path, is the *middle path* (*MP*), which is the path that is always in the middle of the track. The second one is the path that goes through the middle of the middle path (*MP@50*), and the third path's points are always in the middle of the previous path and the shortest path (*MP@25*). Obviously, from those three paths, the only useful ones are the last two, since the middle path was used just as a reference for the other two. The forth and last path implemented is an approximation of the minimum curvature path (MCP) that was discussed in the Background section. In order to implement the new paths, since we already had some points from NavFn's shortest path, we used those points and just geometrically moved them inside the track. At first, for every point computed by NavFn, a search sequence for the corresponding track borders was used, and then that point was moved to the middle of the track using their Euclidean distance. This method though, not only took too much time to be a feasible way, but most importantly had a lot

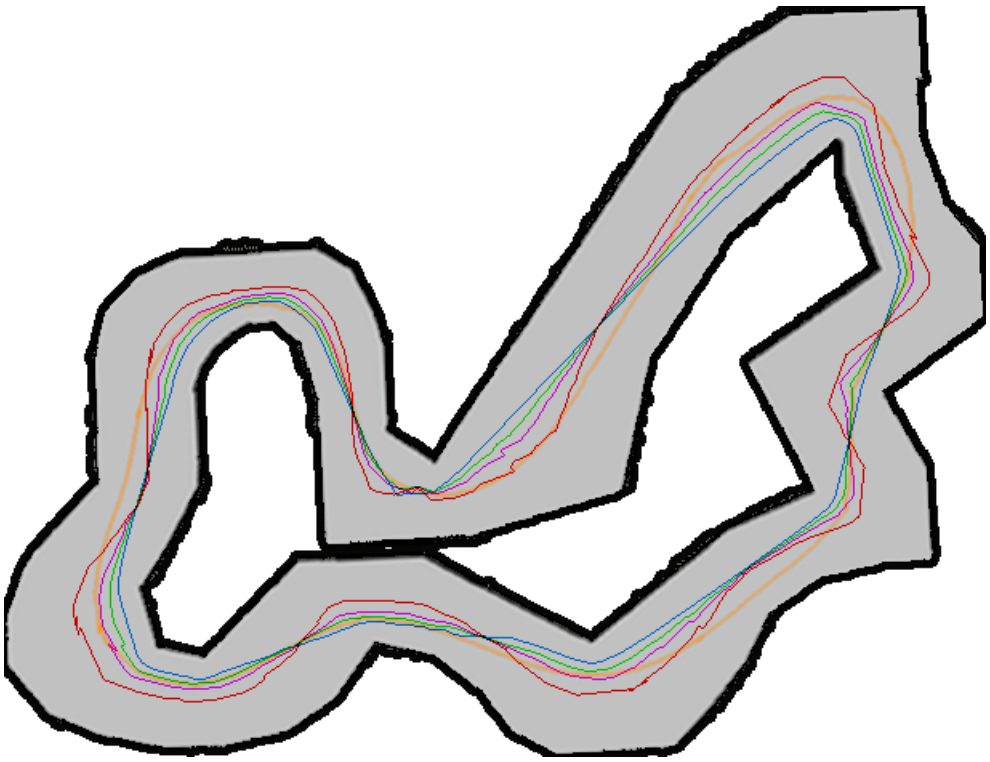

Figure 3.5: The five paths in Track1

of noise (edges throughout the path) as each point was calculated separately.

As a solution to the said problem, every fifteen points calculated from NavFn are used to define the middle path, so the process is a lot faster, and the path almost completely smooth. After the calculation of the middle path, the other two paths are defined by just moving the middle path's points. MP@50's points are in the middle (half of the Euclidean distance) of the middle path and the shortest path. Following the same logic, MP@25's points are in the middle of the MP@50 and the shortest path.

In addition to the above methodology, to calculate the MCP, the track was divided into segments where one point for each segment had to be chosen as best, while recursively increasing the number of segments to make the path smoother, in accordance to the K1999 algorithm [16].

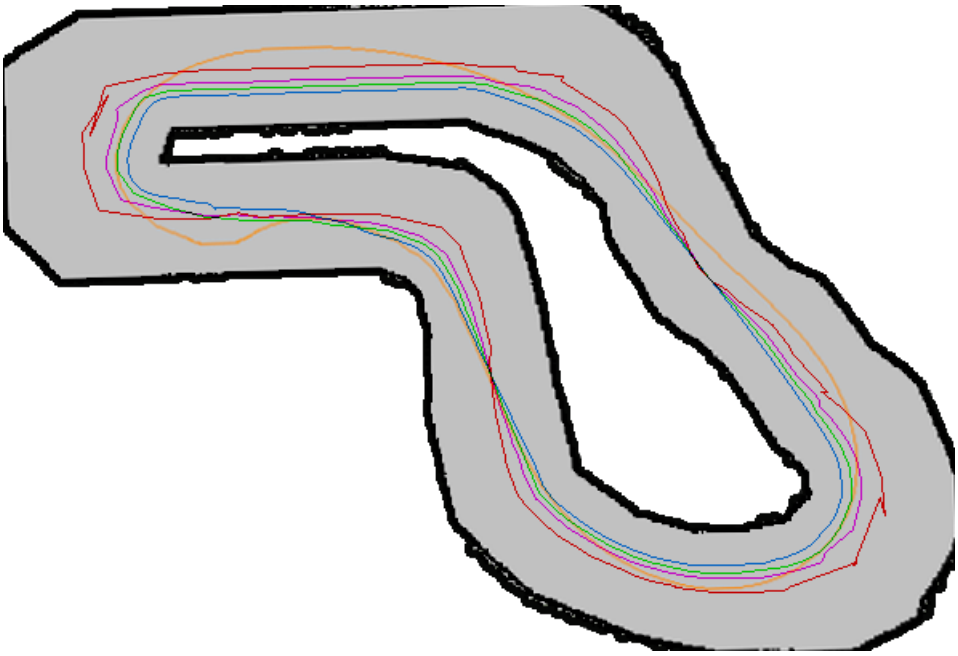

Figure 3.6: The five paths in Track2

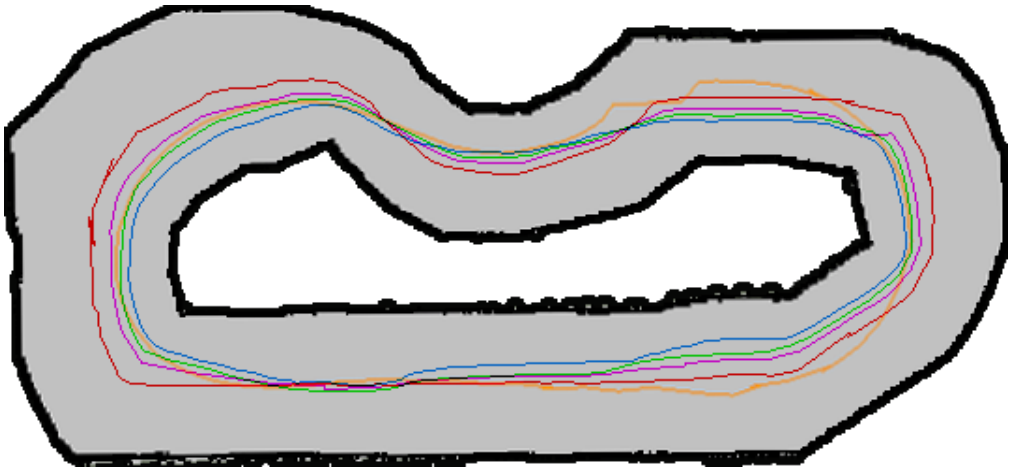

Figure 3.7: The five paths in Track3

Figures 3.5, 3.6 and 3.7 show the five different paths, each one with a different color, in three different test tracks that were created in Gazebo simulator. The shortest path is in blue color, MP in red, MP@50 in pink, MP@25 in green, and MCP in orange.

3.5 Conclusion

In conclusion, the design and implementation of a fast navigating system requires the modification of the move base module of ROS, and more specifically the modification of the global and local planner nodes. Five paths were implemented each one with its advantages and disadvantages, that have to be investigated through experiments, which mainly include the comparison between each algorithm's lap time on different tracks.

Chapter 4

Experimental Methodology and Results

4.1 Introduction

The implementation of a collection of paths for the robot to follow is actually half of the work that should be done in order to find the best path for our robot. The next stage is the experiments with which a path is going to be chosen as best according to our needs and data. The experiment stage is based on the Gazebo simulator, and was run in two different machines for maximum portability.
More details on this subject are presented in the next sections, analysing the process of experimentation, and presenting results.

4.2 Experiments on Gazebo Simulator

4.2.1 Experimentation Setup

Experiments on Gazebo simulator using ROS is the best way to test your code without involving a real robot. With the parameters set correctly, Gazebo is able to offer a very accurate simulation of controlling a mobile platform's motors and of receiving input from different sensors in addition to gravity, friction and collision simulation.

The tests that were run on the various tracks were simple yet effective for our cause. Lap times were recorded for each track, both clockwise and counter-clockwise, so that we have two different results from the same track, since the entrances, exits, and the turns themselves are mirrored. This makes each track produce two different lap times from the same algorithm, one for the clockwise and one for the counter-clockwise lap. For the MCP, this happens because the algorith takes into account the curves of the

tracks, which change according to the direction of the race. The rest of the algorithms, which do not use in any way the turns of the track, produce different results because the braking points are different since the entrance and the exit of each turn are not the same when the direction of the race changes. The experiments also included the tweaking of controller and local planner parameters, in order to achieve the most accurate movement from the simulated robot, Husky [15].

For the simulated experiments it was found that a controller frequency, which is the frequency that the motor controller is changing its command to the motors so that the robot keeps following the path, of approximately 25Hz is ideal, depending on the CPU of the machine that is running the system. In addition to the controller frequency, a really vital parameter is the simulation time (sim_time), which was set to 1.5-2.5 seconds, which is the time that local planner will look forward to plan the velocity. A higher number for the sim_time parameter means better prediction of the ideal velocity, but the computations made are too many even for a powerful CPU to handle while also having to control the motor controller at 25Hz and run path planning or other algorithms at the same time. The rest of the parameters should reflect the specifications of the motor system that is being simulated. As a reference, the settings that were used for our experiments, are shown in Tables 4.8 to 4.11. In the base local planner parameters, the controller frequency is set at 40Hz because the robot's maximum velocity is set at 1 m/s.

For the simulated experiments, two different machines were used. One with a much weaker CPU performance, and one with above average performance. This separation of the experiments was done, because each machine actually needs a different combination of parameters in order to use its CPU in the most effective way. For example, while the machine with the powerful CPU could run a simulation with a controller frequency of 40Hz, the less powerful one, had a controller frequency of 20-25Hz, and a bit bigger prediction window, to compensate with the lower controller frequency, and avoid collisions.

Table 4.12 shows the different values for the major parameters of the base local planner for both of those machines. This table presents the fact that as the motor maximum speed increases (max_vel_x) the controller frequency and prediction window parameters need a higher value too. These values should not be higher than a certain limit though, which is mostly defined by the CPU's capabilities, because the system will fail to update its commands in time, and collide with obstacles. A balance between the prediction_window and the controller_frequency must be discovered which varies from system to sytem.

4.2.2 Track Design

Three tracks were created in Gazebo's built-in 3D building editor, for experimental purposes. The tracks should not offer an advantage to an algorithm over the others, and instead try to imitate the synthesis of real world tracks by following their standards, with 'S' turns, 'U' turns etc.

The most representative example of a biased case is a NASCAR-like oval track which can be seen in Figure 4.15. In this figure, the shortest path is the inner boundary of the track, making this racing line extremely slow compared to the MCP which takes advantage of the whole track width, to minimize its curvature, and maximize the vehicle's speed.

The tracks made for NASCAR are created in order to stress the drivers' endurance, and offer a good show for the spectators, that includes high speeds, and crashes. On the other hand, Formula1 tracks are focused on corners, and require precise driving skills from the drivers, in order to completely avoid collisions, since the single-seated vehicles of Formula1 are much more fragile than the NASCAR cars.
The tracks are carefully designed to demonstrate and evaluate the different algorithms on realistic and technical situations. The segmentation of the tracks follows the standards of real-life Formula1 racing, where the tracks are divided into sectors, based on their turns. Figure 4.16 shows, between other track characteristics, the sectors of Monza, the track of the Italian Grand Prix. Comparing the segmentation of Monza and our test tracks, Track1 and Track2, we can see that there is a same logic behind their division in sectors: each sector has a major turn and its entrance or exit. For example, in Monza, 'Parabolica', its most known turn, is included in Sector 3. Sector 3 has only one more turn which can be considered as a breaking point between sector 2 and 3, as well as the entrance to the huge straight that leads to 'Parabolica'. The same thing happens in sector 3 of Track1, where the main top turn is the major turn of that sector, but in this same sector the previous turn is included, with an identical logic, as in Monza's third sector.

The exact same idea is implemented for all the turns of the created tracks, like Track1's second sector where an 'S' turn is included along with its entrance. Track2 breaks its 'S' turn in half and treats it as a minor turn, because the two 'U' turns are more important for this track, and the 'S' turn imitates the use of 'Variante Ascari' in Monza, dividing the two sectors.

By following the Formula1 standards and adapting them into the created test tracks, the requirements of the unbiased tracks has been met, making the results from the experiments on them as valid as possible.

4.2.3 Results

In Tables 4.1 and 4.2 the lap times for the four paths (excluding MCP, to find the best of the other four and then make a comparison with the MCP) are shown for the three tracks clockwise and counter-clockwise respectively. From those two tables we can see that the shortest path is the best of the other three with the MP@75 coming second. With that being said, we come to the conclusion that the minimum curvature path should be compared with the shortest path.

Tables 4.3 and 4.4 show the lap times in the same three tracks while the robot was racing in the high-end computer. The last two tables clearly show that the shortest path is actually faster than the minimum curvature path for a robot that its motor maximum speed is 0.7 m/s.

To investigate this phenomenon a bit further, the maximum speed of the motors of the simulated robot was gradually boosted by 0.1 m/s from 0.7 to 1.0 m/s. The goal here was to find a value for the motor's maximum speed that beyond it, the difference between the MCP and SP becomes bigger, with the MCP being the faster one.

In order to accurately compute the lap times with the new motor maximum speeds, controller frequency had to be changed to higher values, reaching 40Hz when the motor maximum speed was set to 1.0 m/s. The results for these metrics can be found in tables 4.5 to 4.7. For a better understanding of the results, Figures 4.1 to 4.3 present the change in the Lap Times (in seconds) when the motor maximum speed of the robot increases (in m/s). In Figure 4.4 the influence of the motor maximum speed to the average speed is presented, by collecting data from all tracks. From all these plots, it is clear that above 0.8 m/s, MCP starts becoming a faster path, but their difference becomes more and more appreciable while the motor maximum speed increases.

Another interesting approach to this matter is the investigation of the tracks' lap times, sector by sector. Figures 4.5 and 4.6 present the sectors for Track1 and Track2 respectively (Track3 is omitted due to its almost symmetrical shape which makes its segmentation pointless), while Figures 4.7 to 4.11 show the different sector times achieved when the motor maximum speed changes to higher values. Continuing with the comparison between the shortest and the minimum curvature path, Figure 4.8 is quite interesting, as the MCP seems to be slower even after 0.9 m/s.

Although this result seems quite unexpected at first, it is actually normal for a robot to race through those two paths in almost the same time,

because not only the sector is quite short, but also the paths are identical in many points, and when they are not, the robot is going out of its path when following the SP, due to its high speed, and unintendedly creates a different path that is closer to the MCP, just to come back to the SP, when the motor controller is activated again. As it was mentioned earlier, the controller frequency is the parameter that controls this activation of the motor controller. The commands sent to the motors are updated based on the controller_frequency value, and during the time between two activations, the last command is sent to the motor controllers repeatedly, causing the behaviour described above.

This inability of the robot to follow the path precisely is a problem that cannot be completely avoided, and the higher the speed, the more observable this problem is. Although the problem can be partially solved by raising the motor controller frequency, this solution is limited by the CPU of the system. In addition, even though the shortest path gets less and less precise to the global plan, it is still safe to state that beyond a certain motor maximum speed the MCP is faster. This idea is supported by the fact that the SP is actually faster when it is not precisely followed, because the vehicle 'cheats' its way through tight and 'edgy' way points.

Racing teams and drivers are based on the velocity in the apex of the most difficult turns for each track, to determine if a lap was good in terms of, not only time, but technique too. In Figures 4.12 to 4.14 the velocity in the peak of each turn is shown both for the shortest path and the minimum curvature path. The motor maximum speed is set at 1 m/s, and in most cases the MCP is a lot faster than the SP, due to its preparation for the turn, which minimizes the curve, and maximizes the velocity in the most important point of the turn, the peak.

Since the k1999 algorithm, that produces an estimation of the MCP, is considered the best path to follow for a racing vehicle, being a car or a robot, the finding that the shortest path can under certain circumstances be faster, is quite interesting. For this reason, with all the previous experiments, it was discovered that for the same tracks that were used, a motor maximum speed of 0.9 m/s, makes the MCP again the fastest path. We can safely state that the MCP is not the best path for a slow robot that needs to achieve the best possible lap times. The selection of the algorithm that produces the path that the robot will follow should be chosen by taking into consideration its motor's capabilities.

0.7 m/s	Track1	Track2	Track3
MP	07:57:231	04:04:960	03:07:079
MP@50	06:54:597	03:55:434	03:01:328
MP@75	07:00:090	03:52:816	02:59:587
SP	06:41:160	03:49:880	02:55:124

Table 4.1: Clockwise lap times

0.7 m/s	Track1	Track2	Track3
MP	07:27:275	04:06:566	03:02:945
MP@50	06:56:129	03:57:583	03:00:570
MP@75	07:01:090	03:46:262	02:59:981
SP	06:36:586	03:45:711	02:59:591

Table 4.2: Counter-clockwise lap times

0.7 m/s	Track1	Track2	Track3
MCP	02:10:199	01:10:630	00:58:800
SP	02:01:871	01:05:892	00:54:192

Table 4.3: Clockwise lap times MCP-SP comparison

0.7 m/s	Track1	Track2	Track3
MCP	02:16:212	01:11:370	00:59:971
SP	02:06:859	01:05:187	00:56:812

Table 4.4: Counter-clockwise lap times MCP-SP comparison

0.8 m/s	Track1	Track2	Track3
MCP	01:52:566	01:07:677	00:51:615
SP	01:46:682	01:04:848	00:51:132

Table 4.5: Clockwise lap times MCP-SP comparison

0.9 m/s	Track1	Track2	Track3
MCP	01:42:197	00:53:246	00:45:499
SP	01:43:682	00:58:361	00:48:674

Table 4.6: Clockwise lap times MCP-SP comparison

1.0 m/s	Track1	Track2	Track3
MCP	01:39:106	00:51:827	00:44:774
SP	01:42:917	00:57:909	00:48:031

Table 4.7: Clockwise lap times MCP-SP comparison

controller_frequency	40.0
max_vel_x	1.0
min_vel_x	0.05
max_rotational_vel	0.75
min_in_place_rotational_vel	1.0
acc_lim_th	0.75
acc_lim_x	0.50
acc_lim_y	0.50
holonomic_robot	false
yaw_goal_tolerance	0.8
xy_goal_tolerance	0.15
goal_distance_bias	0.01
path_distance_bias	10
sim_time	2.5
heading_lookahead	0.4
oscillation_reset_dist	0.25
vx_samples	6
vtheta_samples	20
dwa	false

Table 4.8: Base local planner parameters

global_frame	/odom
robot_base_frame	/base_link
update_frequency	5.0
publish_frequency	5.0
static_map	false
rolling_window	true
width	4.0
height	4.0
resolution	0.1
transform_tolerance	0.5

Table 4.9: Local costmap parameters

global_frame	/map
robot_base_frame	/base_link
update_frequency	3.0
publish_frequency	0.0
static_map	true
transform_tolerance	0.5

Table 4.10: Global costmap parameters

obstacle_range	2.5
raytrace_range	3.0
footprint	[[-0.5, -0.33], [-0.5, 0.33], [0.5, 0.33], [0.5, -0.33]]
footprint_padding	0.01
inflation_radius	0.6
observation_sources	scan
scan	{data_type: LaserScan, topic: /scan, marking: true, clearing: true}

Table 4.11: Costmap common parameters

System CPU	controller_frequency	max_vel_x	prediction_window
Intel(R) Core(TM) i7-4790 @ 3.60GHz	25	0.7	2.5
AMD A10-5745M @ 2.90GHz	15	0.7	1.5
Intel(R) Core(TM) i7-4790 @ 3.60GHz	25	0.8	2.5
AMD A10-5745M @ 2.90GHz	20	0.8	1.5
Intel(R) Core(TM) i7-4790 @ 3.60GHz	30	0.9	2.5
AMD A10-5745M @ 2.90GHz	25	0.9	2.0
Intel(R) Core(TM) i7-4790 @ 3.60GHz	40	1.0	2.5
AMD A10-5745M @ 2.90GHz	25	1.0	2.5

Table 4.12: High-end computer's in comparison to a medium-end computer's base local planner parameters

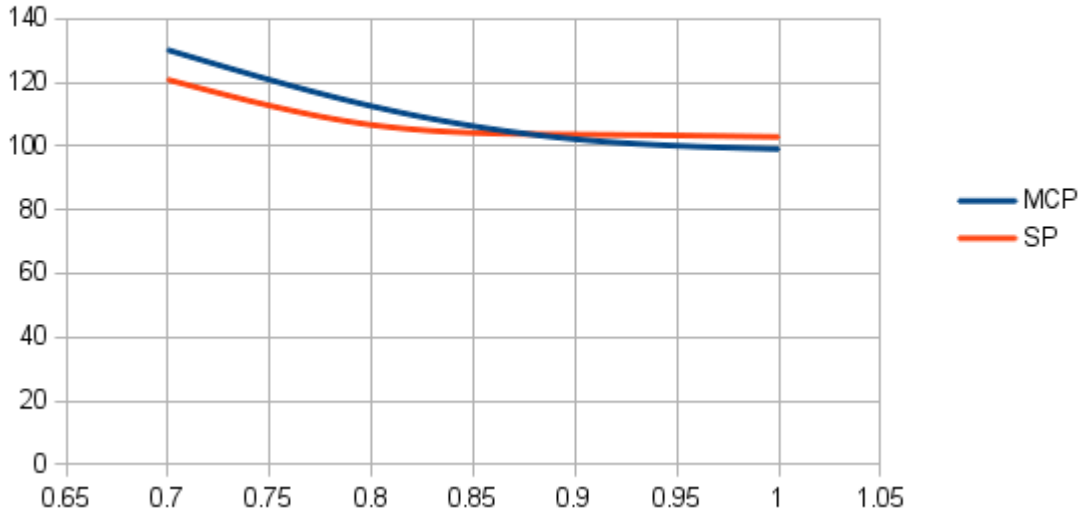

Figure 4.1: Track1 Lap Times/Motor Maximum Speed (seconds / m/s)

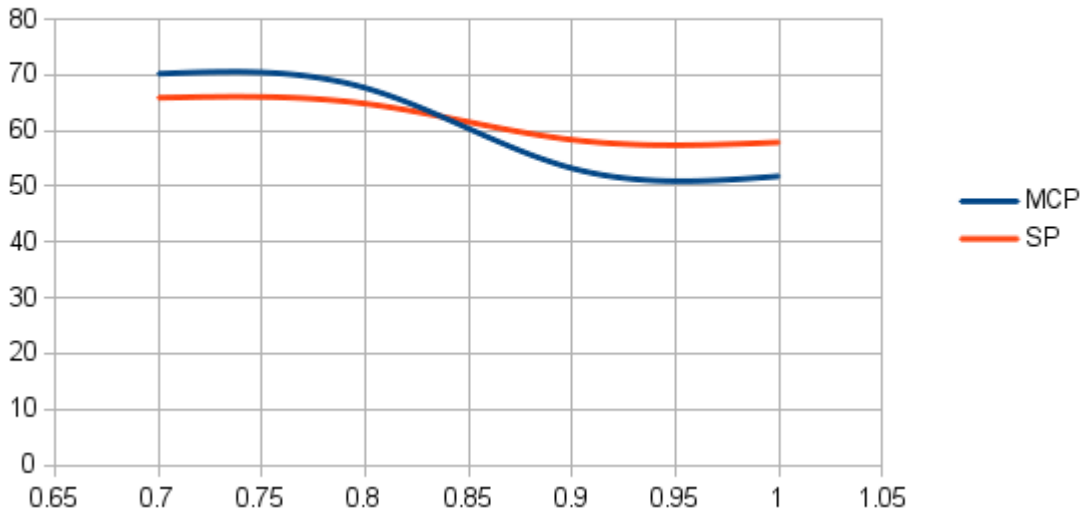

Figure 4.2: Track2 Lap Times/Motor Maximum Speed (seconds / m/s)

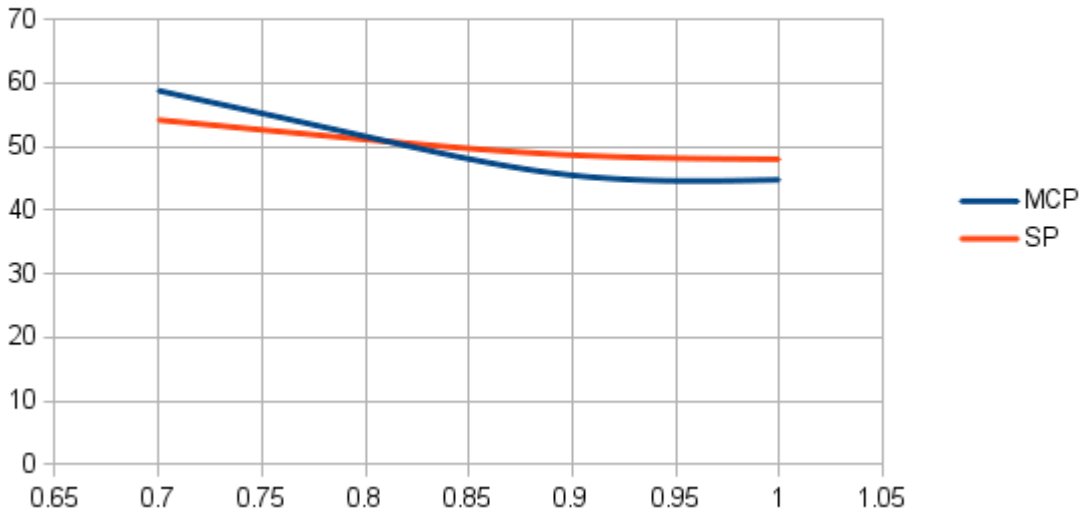

Figure 4.3: Track3 Lap Times/Motor Maximum Speed (seconds / m/s)

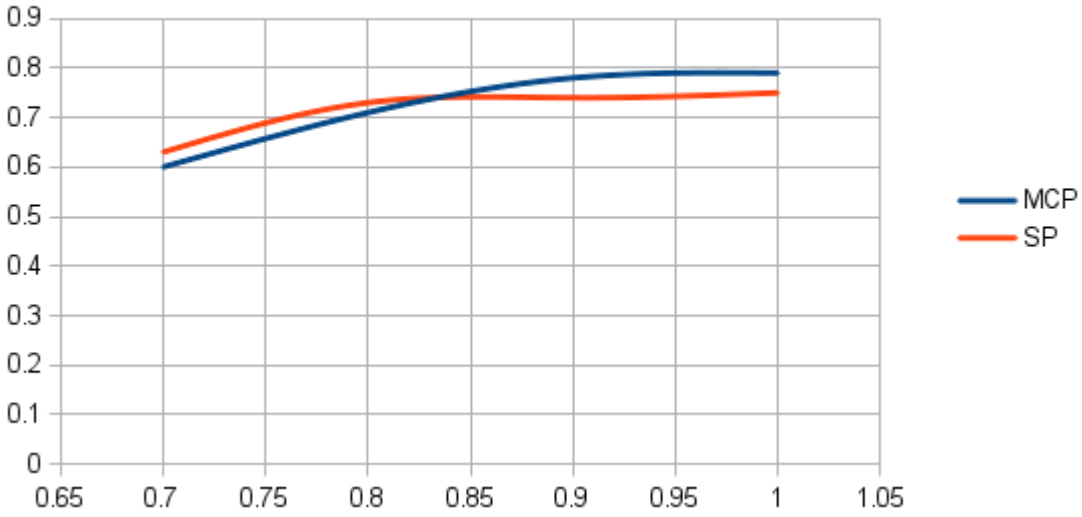

Figure 4.4: All tracks Average Speed/Motor Maximum Speed (m/s / m/s)

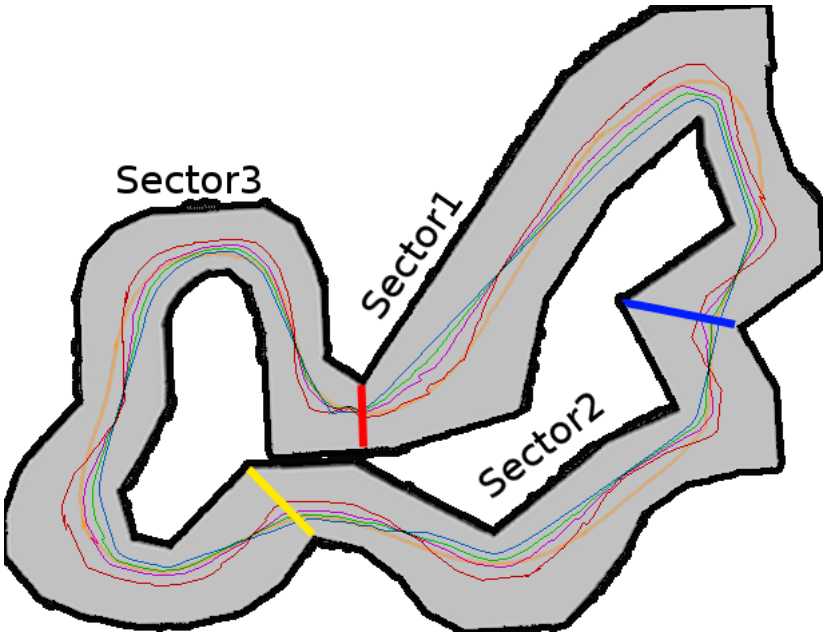

Figure 4.5: The three sectors in Track1
Sector1 Red-Blue
Sector2 Blue-Yellow
Sector3 Yellow-Red

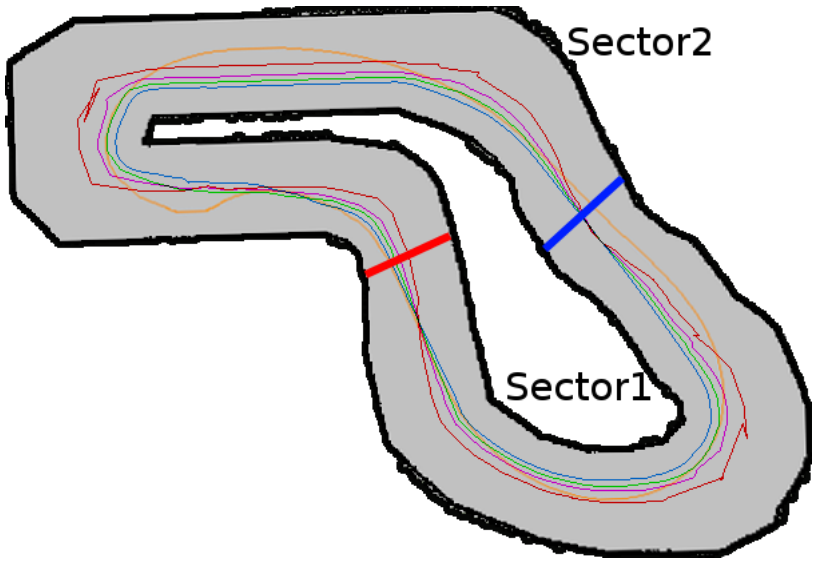

Figure 4.6: The two sectors in Track2
Sector1 Bottom area between red and blue
Sector2 Top area between red and blue

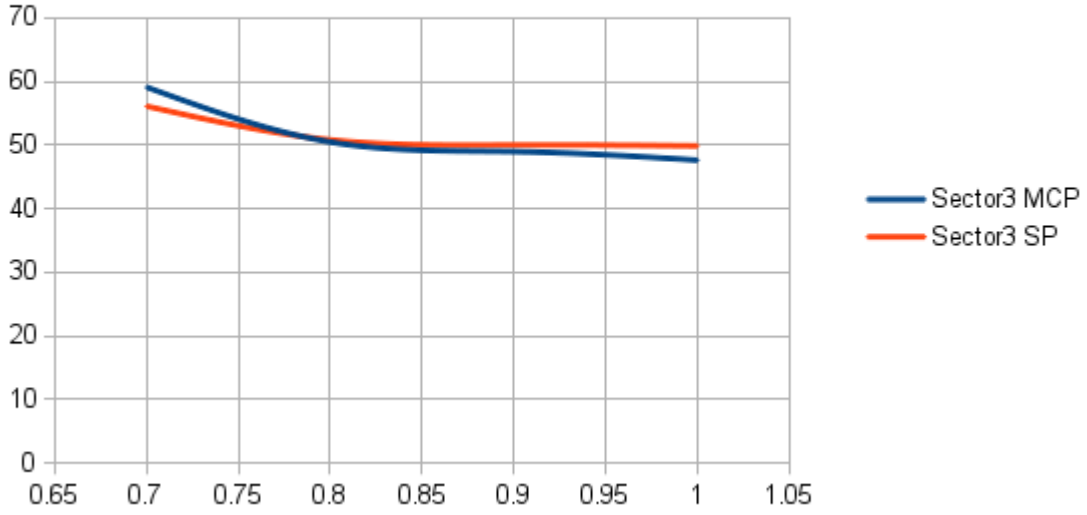

Figure 4.7: Track1 Sector3 Times/Motor Maximum Speed (seconds / m/s)

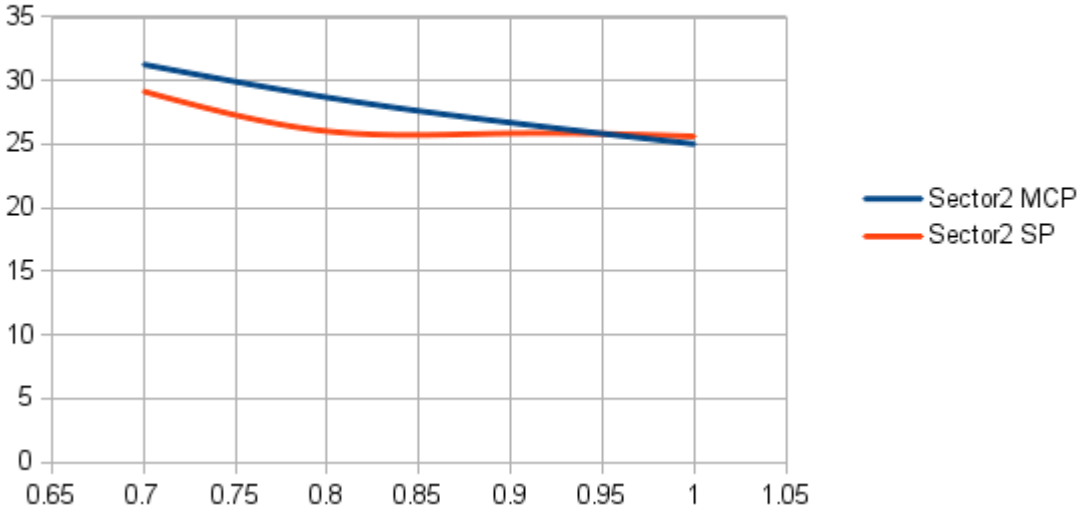

Figure 4.8: Track1 Sector2 Times/Motor Maximum Speed (seconds / m/s)

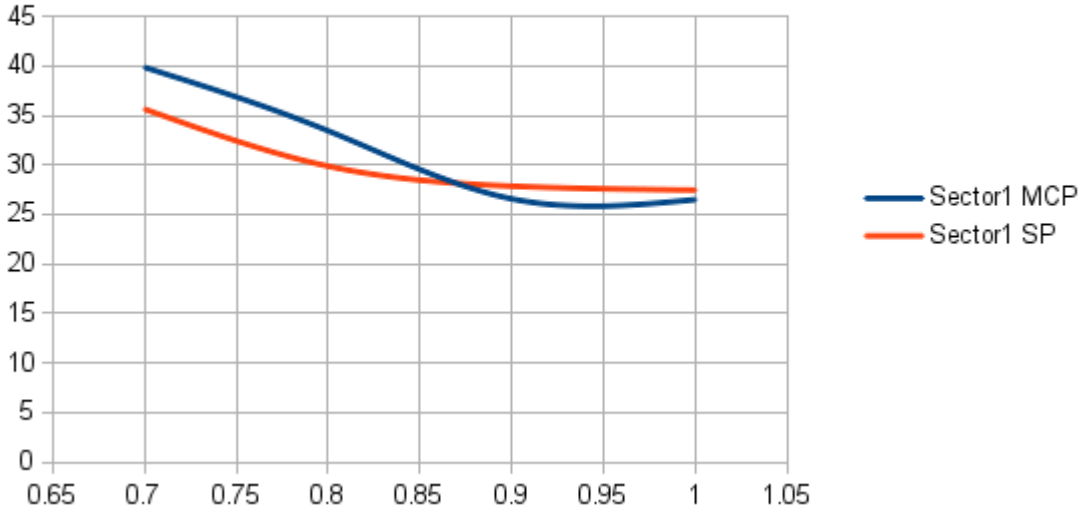

Figure 4.9: Track1 Sector1 Times/Motor Maximum Speed (seconds / m/s)

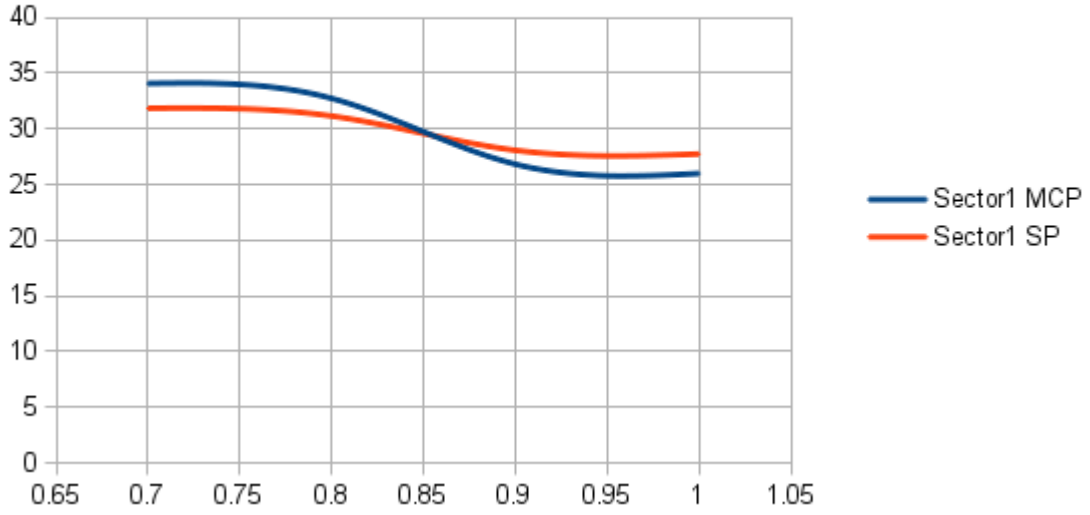

Figure 4.10: Track2 Sector1 Times/Motor Maximum Speed (seconds / m/s)

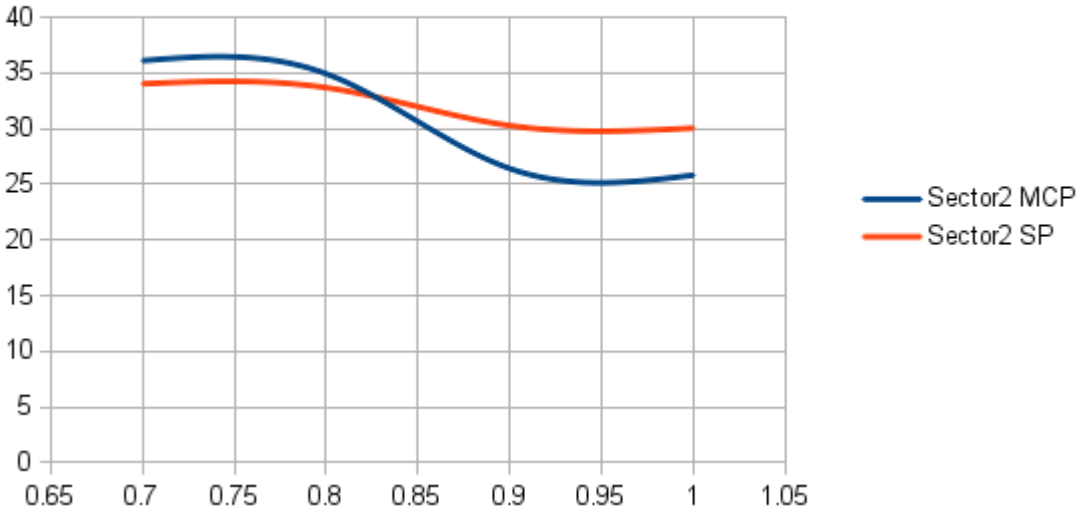

Figure 4.11: Track2 Sector2 Times/Motor Maximum Speed (seconds / m/s)

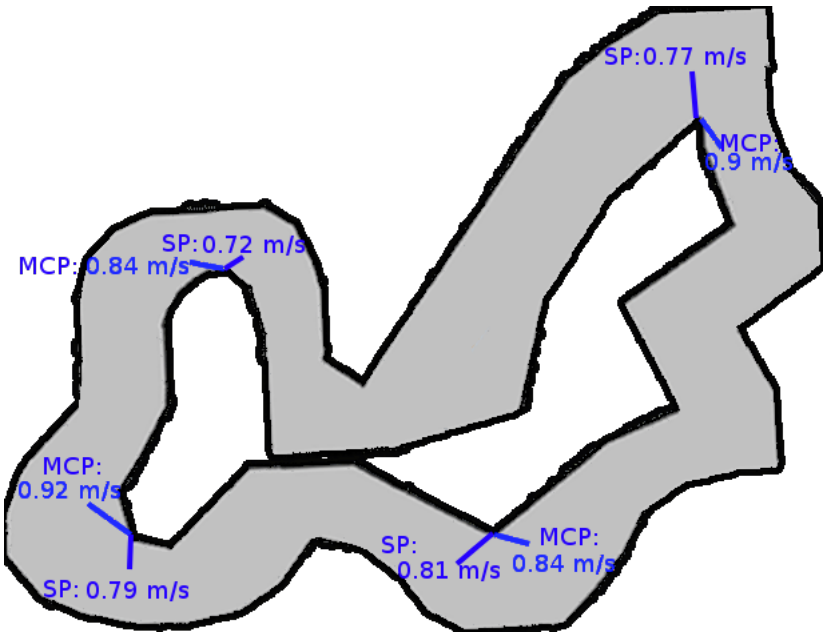

Figure 4.12: Current speed at the basic turns of Track1

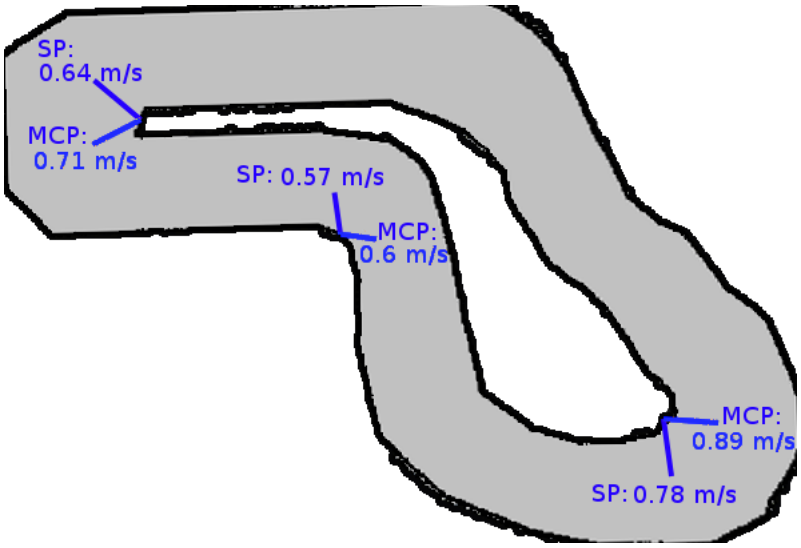

Figure 4.13: Current speed at the basic turns of Track2

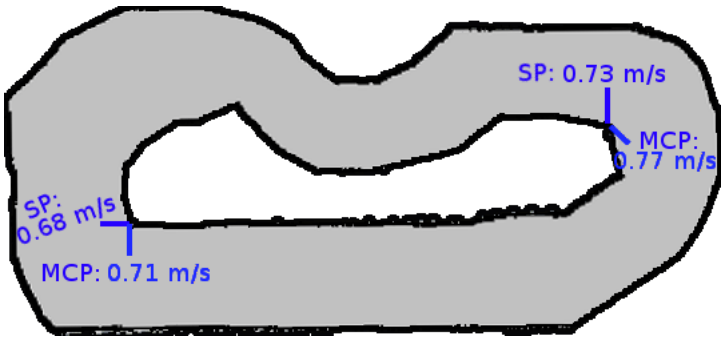

Figure 4.14: Current speed at the basic turns of Track3

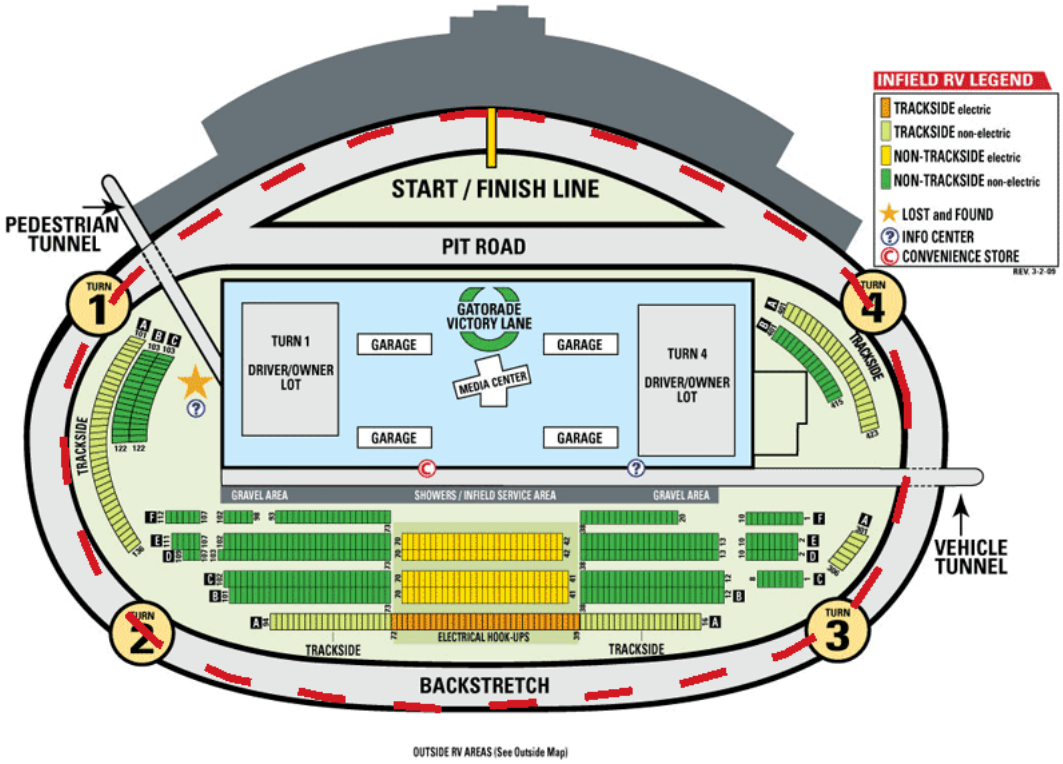

Figure 4.15: Official NASCAR oval track named Daytona. This image is taken from the official fan NASCAR website[3]. The red dashed line has been added to show the MCP in this track.

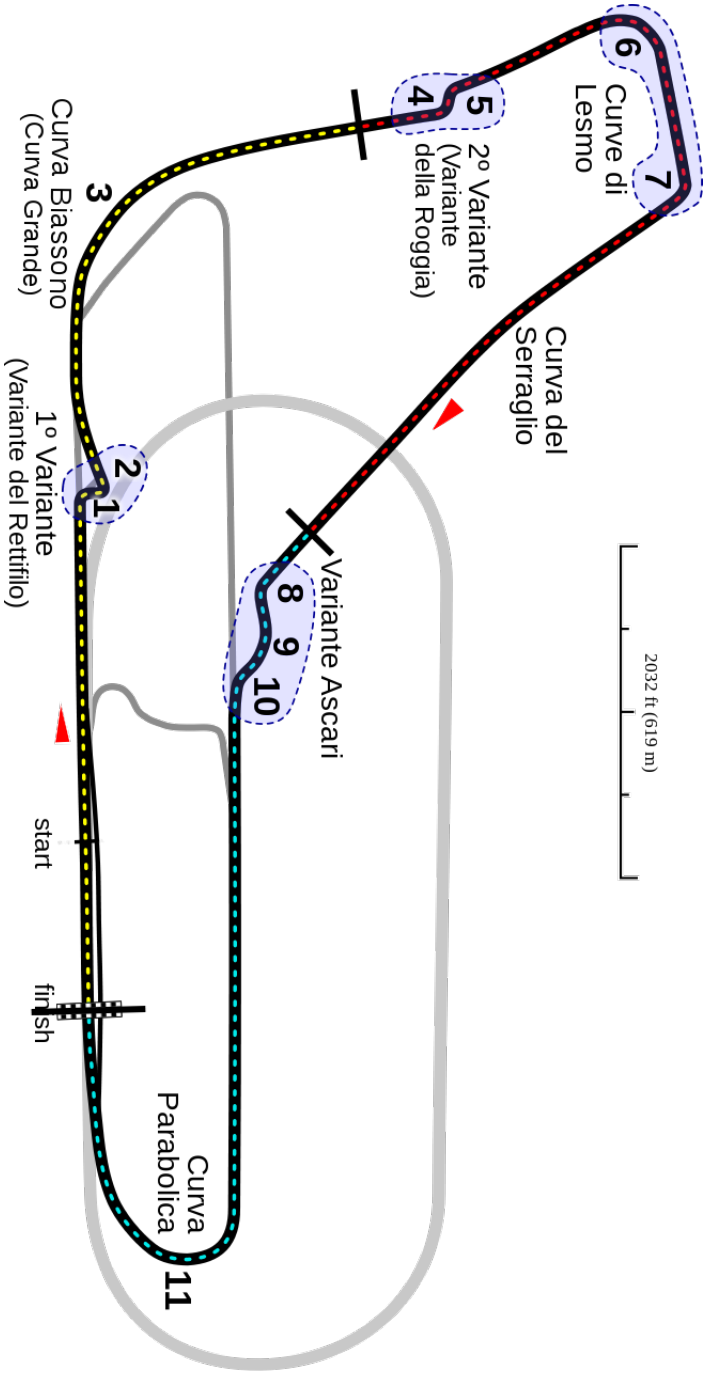

Figure 4.16: Official Formula1 track segmentation at Monza. This image is taken from Wikipedia: Autodromo Nazionale Monza [5]. Three thick line segments have been added to make the sectors a bit more clear.

4.3 Conclusion

To conclude, the experiments on the simulated robot gave us some really interesting results, with the most important being the finding that the MCP is not always the best path for a robot to follow.

This statement is really important since the minimum curvature path has been 'deified' by almost every racing line researcher. A path should not be categorised as good or optimal, without investigating other parameters that can affect its effectiveness. With that being said, MCP should be treated as a good alternative, but not the only path to be followed. This result is based on another conclusion that was made through experimentation that states that the motor maximum speed capabilities is a major factor in the selection of the best path. There are many situations, where the SP can actually be much faster than the MCP, as it was proven in the results of the experiments discussed in this chapter. The proven 'defeasibility' of the MCP and the motor maximum speed, should enable the robot programmer to choose the optimal algorithm to implement according to the given system.

Chapter 5

Conclusions

By designing algorithms for a fast robot and testing each one of them to find the optimal with respect to lap times, this thesis took a step towards the future goal of creating robust path planning strategies that are optimal with respect to speed, for many different applications.

The interesting result that connects the minimum curvature path and the motor maximum speed of the robot, which under certain circumstances makes the shortest path faster, can be used in the future for further path optimization. An interesting implementation that benefits from this is an algorithm that divides the route between the starting and the ending points into sub-routes, where the optimal path in each sub-route can be chosen between the SP and the MCP, making the final path consist of a combination of shortest and minimum curvature paths.

The implemented algorithms and results from this thesis, can be used in the future as a base for the creation of an emergency robot, that navigates fast and efficiently. The future implementation could include the following:

- The ability of the robot to navigate through tracks while having extra obstacles or other robots to block its path. This new algorithm, can be expanded to select between different behaviours like:
  - Avoiding the obstacles and create a new path just to replace the occupied space.
  - Overtaking other robots, by accelerating, and creating a temporary path to clear the overtake, and then return to the planned path.
  - Cut through newly created paths generated by a combination of obstacles.
- The ability of the robot to navigate through a maze, instead of just a track, where the possible direction of the path is only one. In this

implementation, the robot should be able to choose the optimal path in terms of speed and overall time needed to complete the maze, instead of just covered distance. This can be achieved by creating an optimizer of the path created, by evaluating all or part of the possible paths leading from the starting point, to the goal point.

- Another interesting approach is the implementation of an optimal navigating system for areas that are not known for the robot. This means that the robot will not have a map of the area it is currently navigating, so it has to use dynamic path planning algorithms. Some ideas that could implement and extend the capabilities of such dynamic algorithms are the following:
  - Make educated guesses about the partially known curve that is in front of it, with a goal of achieving maximum speeds without colliding.
  - Take risks or drive safely, according to the different circumstances. For example, if the robot must be on time at all costs, and even some seconds have hazardous consequences, crashing will not cause any more harm to the already faulty situation, so the robot has to take a risk, and drive fast without taking all the necessary precautions, actually imitating a human driver on an emergency.
  - As described earlier, an algorithm that could also dynamically switch between path planning algorithms would be interesting. For example, as it was also shown from the results of the experiments, since the shortest path and the minimum curvature path can both be fast in different circumstances, an optimal global plan, could include parts of both those implementations.
- The optimal navigation of the robot is not the only problem that can make it fast though. In situations where more than one robots are used, the navigation becomes more complicated since robots' path may sometimes cross. For this problem, a communicating system between the global path planning algorithms of the robots could be implemented, which could even create 'areas of actions' for each one of them.

All the above implementations regarding the future plans, should use the experimental methods used in this thesis, which proved precise and effective. Their evaluation should use the time-based validation, as well as comparison between other implementations, for a better overview of their effectiveness and robustness.

Bibliography

[1] Formula1 official. http://www.formula1.com.

[2] Gazebo simulator. http://gazebosim.org/.

[3] Nascar official fanpage. http://nascarthreesixty.com/.

[4] Robot operating system. http://www.ros.org/.

[5] Wikipedia, autodromo nazionale monza. http://en.wikipedia.org/wiki/Autodromo_Nazionale_Monza.

[6] Wikipedia, bezier curve. http://en.wikipedia.org/wiki/B%C3%A9zier_curve.

[7] Wikipedia, euler spiral. http://en.wikipedia.org/wiki/Euler_spiral.

[8] Wikipedia, racing line. http://en.wikipedia.org/wiki/Racing_line.

[9] Matteo Botta, Vincenzo Gautieri, Daniele Loiacono, and Pier Luca Lanzi. Evolving the optimal racing line in a high-end racing game. In *Computational Intelligence and Games (CIG), 2012 IEEE Conference on*, pages 108–115. IEEE, 2012.

[10] Paul Bovbel. ROS costmap 2d. http://wiki.ros.org/costmap_2d#Inflation. Last edited: 2014-08-11.

[11] F Braghin, F Cheli, S Melzi, and E Sabbioni. Race driver model. *Computers & Structures*, 86(13):1503–1516, 2008.

[12] Daniel Callander. ROS trajectory planner. http://wiki.ros.org/base_local_planner#TrajectoryPlannerROS. Last edited: 2014-03-24.

[13] Luigi Cardamone, Daniele Loiacono, Pier Luca Lanzi, and Alessandro Pietro Bardelli. Searching for the optimal racing line using genetic algorithms. In *Computational Intelligence and Games (CIG), 2010 IEEE Symposium on*, pages 388–394. IEEE, 2010.

[14] Benjamin Chretien. ROS rviz: 3d visualization tool. `http://wiki.ros.org/rviz`.

[15] ClearpathRobotics. ROS husky robot. `http://wiki.ros.org/Robots/Husky`.

[16] Rémi Coulom. *Reinforcement Learning Using Neural Networks, with Applications to Motor Control*. PhD thesis, Institut National Polytechnique de Grenoble, 2002.

[17] Frank Dellaert, Dieter Fox, Wolfram Burgard, and Sebastian Thrun. Monte carlo localization for mobile robots. In *Robotics and Automation, 1999. Proceedings. 1999 IEEE International Conference on*, volume 2, pages 1322–1328. IEEE, 1999.

[18] Christian Dornhege. ROS navfn. `http://wiki.ros.org/navfn`. Last edited: 2013-02-16.

[19] Pooyan Fazli. ROS move base. `http://wiki.ros.org/move_base`. Last edited: 2014-08-01.

[20] Thomas Gustafsson. Computing the ideal racing line using optimal control. Master's thesis, Linköpings Universitet, SE-581 83 Linköping, 2008.

[21] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *Intelligent Robots and Systems, 2004.(IROS 2004). Proceedings. 2004 IEEE/RSJ International Conference on*, volume 3, pages 2149–2154. IEEE, 2004.

[22] Gilles Reymond, Andras Kemeny, Jacques Droulez, and Alain Berthoz. Role of lateral acceleration in curve driving: Driver model and experiments on a real vehicle and a driving simulator. *Human Factors: The Journal of the Human Factors and Ergonomics Society*, 43(3):483–495, 2001.

[23] Roland Stelzer and Tobias Pröll. Autonomous sailboat navigation for short course racing. *Robotics and autonomous systems*, 56(7):604–614, 2008.

[24] Sebastian Thrun, A. Buecken, W. Burgard, Dieter Fox, T. Froehlinghaus, D. Henning, T. Hofmann, M. Krell, and T. Schmidt. Map learning and high-speed navigation in rhino. In D. Kortenkamp, R.P. Bonasso, and R. Murphy, editors, *AI-based Mobile Robots: Case Studies of Successful Robot Systems*. MIT Press, 1998.

[25] M.E. Tipping, M.A. Hatton, and R. Herbrich. Racing line optimization, March 12 2013. US Patent 8,393,944.

[26] Christopher Urmson, Joshua Anhalt, J. Andrew (Drew) Bagnell, Christopher R. Baker, Robert E Bittner, John M Dolan, David Duggins, David Ferguson , Tugrul Galatali, Hartmut Geyer, Michele Gittleman, Sam Harbaugh, Martial Hebert, Thomas Howard, Alonzo Kelly, David Kohanbash, Maxim Likhachev, Nick Miller, Kevin Peterson, Ragunathan Rajkumar, Paul Rybski, Bryan Salesky, Sebastian Scherer, Young-Woo Seo, Reid Simmons, Sanjiv Singh, Jarrod M Snider, Anthony (Tony) Stentz, William (Red) L. Whittaker, and Jason Ziglar. Tartan racing: A multi-modal approach to the darpa urban challenge. Technical Report CMU-RI-TR-, Robotics Institute, http://archive.darpa.mil/grandchallenge/, April 2007.

[27] Bart van de Poel. Raceline optimization, 2009.

[28] Jung-Ying Wang and Yong-Bin Lin. Game ai: Simulating car racing game by applying pathfinding algorithms. *International Journal of Machine Learning and Computing*, 2(1):13–18, 2012.

[29] Ying Xiong et al. *Racing line optimization.* PhD thesis, Massachusetts Institute of Technology, 2010.

[30] Safdar Zaman, Wolfgang Slany, and Gerald Steinbauer. Ros-based mapping, localization and autonomous navigation using a pioneer 3-dx robot and their relevant issues. In *Electronics, Communications and Photonics Conference (SIECPC), 2011 Saudi International*, pages 1–5. IEEE, 2011.