Budapest University of Technology and Economics
Faculty of Electrical Engineering and Informatics
Department of Automation and Applied Informatics

Development of Navigation Methods in Dynamic Environment for an Intelligent Model Car

Master's Thesis

Author
Soma Veszelovszki

Advisor
Domokos Kiss

December 3, 2019

Contents

HALLGATÓI NYILATKOZAT

Alulírott *Veszelovszki Soma*, szigorló hallgató kijelentem, hogy ezt a diplomatervet meg nem engedett segítség nélkül, saját magam készítettem, csak a megadott forrásokat (szakirodalom, eszközök stb.) használtam fel. Minden olyan részt, melyet szó szerint, vagy azonos értelemben, de átfogalmazva más forrásból átvettem, egyértelműen, a forrás megadásával megjelöltem.

Hozzájárulok, hogy a jelen munkám alapadatait (szerző(k), cím, angol és magyar nyelvű tartalmi kivonat, készítés éve, konzulens(ek) neve) a BME VIK nyilvánosan hozzáférhető elektronikus formában, a munka teljes szövegét pedig az egyetem belső hálózatán keresztül (vagy autentikált felhasználók számára) közzétegye. Kijelentem, hogy a benyújtott munka és annak elektronikus verziója megegyezik. Dékáni engedéllyel titkosított diplomatervek esetén a dolgozat szövege csak 3 év eltelte után válik hozzáférhetővé.

Budapest, 2019. december 3.

Veszelovszki Soma
hallgató

Kivonat

A diplomatervezési feladat egy, a tanszéken futó kutatási projekthez kapcsolódik, amelyben egy intelligens autó tesztplatform létrehozása a cél. A platform alapja egy csökkentett méretű (kb. 1:3 méretarányú) autómodell, melyet egy távirányítható játékautóból alakítunk át. A tesztplatform létrehozásának célja különböző navigációs algoritmusok fejlesztésének és valós környezetben történő tesztelésének elősegítése.

A sikeres navigáció és az ütközések elkerülése érdekében fontos, hogy a jármű valós időben detektálni tudja a környező objektumokat, és megfelelően reagáljon rájuk. Ennek elősegítése érdekében az autóplatformon elhelyezésre került egy-egy vízszintes síkban pásztázó lézeres távolságszkenner (lidar) a jármű elején és hátulján.

A feladat keretében ezekre a szenzorokra építve kell megvalósítani mozgó objektumok felismerését, azok méretének és sebességvektorának becslésével együtt. Fontos, hogy a jármű el tudja különíteni a statikus és a mozgó akadályokat egymástól. További feladat egy olyan akadályelkerülési módszer megvalósítása az autón, amely az objektumok mozgását is figyelembe véve hozza meg a megfelelő navigációs döntést minden időpillanatban.

A megvalósított algoritmusok működését mind szimulált, mind valós környezetben szükséges ellenőrizni. A valós tesztelés történhet a tanszéki járműplatformon, vagy egy kisméretű, egyedileg felépített modellautón is.

Abstract

The theme of the thesis is a sub-task of a research project at the Department of Automation and Applied Informatics, with the purpose of designing an intelligent car testing platform. The platform is based on a decreased-size (around 1:3 scale) remote control car model, that has been modified to support self-driving program control. The platform aims to support the development of different navigation algorithms and helping the testing in real environment.

For a successful navigation and obstacle avoidance, the detection of the surrounding objects and reacting accordingly are essential. For the purpose of detection, two horizontal distance scanning sensors (LIDARs) have been placed on the car - one on the front and one on the back.

Part of the task is to implement the detection of the moving obstacles and estimate their sizes and speed vectors. It is important for the car to be able to separate static and moving objects from each other. The other part is developing an obstacle-avoidance method, implemented as an application on the car, that takes the moving objects into consideration while making navigational decisions at every time step.

The implemented algorithms need to be tested both in simulation and in real environment, that may be executed on the department's vehicle platform or on a small-scale, custom-build model car.

Chapter 1

Introduction

1.1 Self-driving solutions

Nowadays, self-driving cars gain more and more attention, both their technology and their effect on people's daily routines and their lives overall. Several articles are published every year about how these cars will change the way people commute to work, visit their friends or go on a family vacation. These articles often point out the decrease of the number of accidents, an optimized load of traffic and thus a reduced fuel consumption as the major advantages of this technology-to-come.

The release date of these cars, however, is still a matter of question. In 2015, Mark Fields, president and CEO of Ford at the time estimated their first fully autonomous car in 2020. 2 years later, at CES 2017 Nvidia announced that with the partnership of Audi they would develop a self-driving vehicle - also, in market by 2020. Both of these statements are considered too ambitious guesses today, as there is a high probability that we need to wait for at least another decade for reliable fully autonomous vehicles to hit the roads. Claiming that there are no viable signs of these cars in traffic would be a false statement, as there are several companies who have been testing there vehicles on public roads for the last years, but these prototypes are very far from reliable products yet. The company that seems to be ahead of the competition in this race is Tesla. Their self-driving software is already in their products, but it still needs millions of hours of testing and the responsibility is still the driver's if an accident happens.

But why is this delay of release dates? One possible answer is that manufacturers have the tendency to exaggerate when asked about new products, and thus the users' need and the other competitors development speed urged them to make such estimations they could not keep up with. Another theory is that the companies at that time didn't acknowledge how many hours and kilometers of testing is needed to finalize a self-driving product.

However, the spreading of autonomous vehicles is blocked by several legislative and technological obstacles. As this thesis describes an engineering problem an its solution, I will reflect on the technological blockers that both car manufacturers and other self-driving software developer companies need to face. Just to list some of these problems, we can name reliable object detection, error insensitive, robust decision-making, fast, well-tuned physical control, and to meet the safety and quality requirements set by the market, determinant scheduling and redundancy throughout the whole software are also essential.

Almost all L5 level self-driving applications can be divided into the same sub-tasks, which are perception, environment building, trajectory planning and control.

Perception (or sensor layer) consists of managing a sensor setup that is able to provide the car with the necessary information about its environment and its own state to create a realistic model of the world. One of the hardest question in this area of engineering is how many and what type of sensors to use. It is especially important for automotive manufacturers, for reducing the price of a sensor set on a car makes manufacturing more profitable. Typical sensor setups include 4 front and 4 rear radars for short-range measurements (typically parking purposes), long-range radars facing forward and backward, IMUs[1] and GPS receivers. Camera-based solutions (like the one Budapest-based AImotive is developing) may also need front- and rear-facing mono- and/or stereo-cameras and fish-eye cameras on the side. Other approaches use LIDARs placed on top of the car (e.g. Uber and Waymo). The price of these laser scanners are sometimes higher than the car they are applied on. While trying to keep them as minimal as possible, manufacturers must create sensor setups that will be fit for the needs of future fully autonomous programs, which may not be estimated currently today. After the setup has been decided on, the perception layer is still in need of an enormous amount of engineering, as it is responsible for handling these sensors - calibrating them, reading and filtering their values, and forwarding them to the upper-level components of the chain.

Using the filtered, but still raw output of the sensor handling components, these softwares need to make a realistic and fine-grained map of the world surrounding the vehicle. This environment-building sub-task requires various different, preferably independent input sensors, and it implements the fusion of these data. While its inputs are raw camera images, 2D or 3D LIDAR scans and radar measurements, its output is a 2D or 3D (depending on the application) map of the world, possibly published in a standard format for further evaluation. Among many problems this step has to solve are the different measurement frequencies of the sensors (cameras used in seld-driving solutions typically record video at 30 frame/sec, while automotive LIDARs rotate at maximum 20 Hz), the sensor noise that still remains after filtering, and also contradicting measurements coming from different sensors.

The built world model is used by trajectory-planning that calculates an optimal path for the target destination. This path should avoid any obstacles that may cause a collision. This step requires detailed information about the physical dimensions and kinematics of the objects in the map, and also about the driven vehicle's capabilities (such as maximum acceleration or maximum wheel angle) - note, that these values are strictly limited according to automotive standards and safety measurements.

Once the desired trajectory has been calculated, the control module has the task to actually *drive* the car. It controls the actuators, and its aim is to follow the path as strictly as possible. This steps needs a well-detailed picture of the controlled car's characteristics, which can be obtained via in-depth identification.

All important players in the automotive industry are developing some kind of self-driving, using different software architectures defined by the above modules.

1.2 The *vr-car* project

The Department of Automation and Applied Informatics at the Budapest University of Technology and Economics also has a team developing a self-driving solution - however, in

[1] Inertial measurement units are electronic devices that measure and report a body's specific force, angular rate, and sometimes the orientation of the body, using a combination of accelerometers, gyroscopes, and sometimes magnetometers. (source: Wikipedia)

a much smaller scale. The *vr-car* project is based on model car of around 1:3 scale, which is applied with several computers, sensors and actuators (see figures 1.1 and 1.2) in order to make it fit for testing self-driving algorithms. As my diploma project I participated in this endeavour by implementing a mapping algorithm that separates static and moving objects, and a local trajectory planner that controls the car while avoiding collisions with these obstacles. Please note that the hardware components and software modules that the *vr-car* project embraces and are mentioned in the following introduction are not my work. The modules developed by me (the mapping and the local planner nodes) are described later, starting from section 2. But before explaining my project in detail, let me introduce the *vr-car* solution, as a frame for my diploma project.

The original car is a child's toy - a car equipped with a DC motor and a steering servo, with an operational steering wheel and throttle pedal. Its size is roughly 100cm*60cm, perfect for applying modifications that enable the car to become a self-driving algorithm test vehicle.

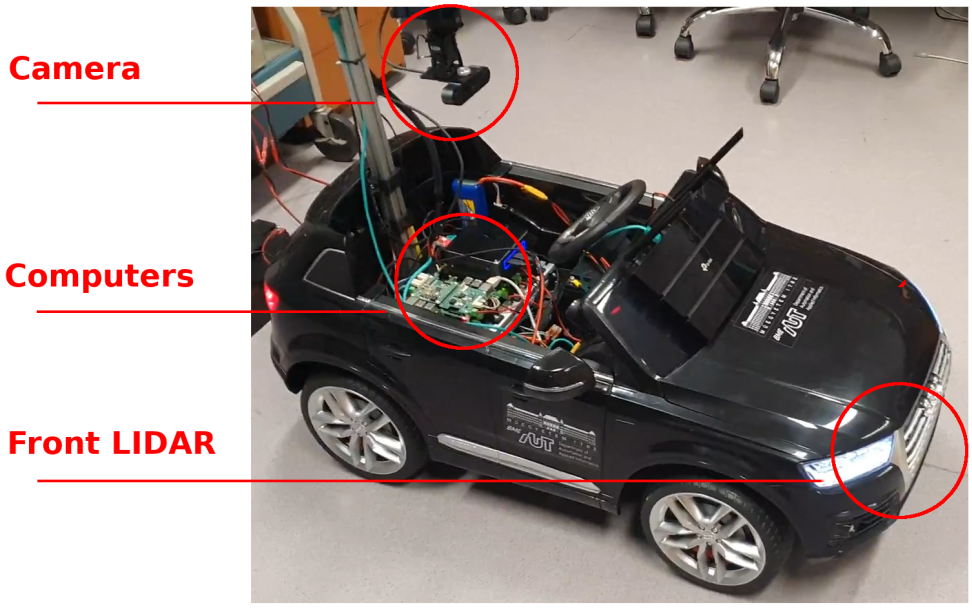


Figure 1.1: The car setup (front side)

Amongst others, the car is equipped with a front, a rear and a top LIDAR, a camera, inertial sensors, ultrasonic distance sensors, several Raspberry Pi-s and an Intel NUC computer.

The vehicle supports the development and testing of multiple smaller projects (mapping and navigation algorithms, control modules, etc.), all contributing to the car becoming more and more able to drive itself. For these sub-projects, the above mentioned sensors are all needed, later I will explain them in detail. These sensors and computers provide a good hardware base for the project, but they are not all that's necessary. A project of this complexity cannot be maintained without a reliable and robust software framework to build upon. This framework is ROS.

1.2.1 About ROS

> The Robot Operating System (ROS) is a **set of software libraries** and tools that help you build robot applications. From drivers to state-of-the-art algo-

Figure 1.2: The car setup (rear side)

> rithms, and with powerful developer tools, ROS has what you need for your next robotics project. And it's all **open source**.

The above description is quoted[2] from their website. Short as it may be, their description does point out that ROS is a software framework for robotic applications. Wikipedia also has a brief and comprehensible article about ROS, in which it states that:

> Although ROS is **not an operating system**, it provides services designed for a heterogeneous computer cluster such as hardware abstraction, low-level device control, implementation of commonly used functionality, message-passing between processes, and package management. Running sets of ROS-based processes are represented in a **graph architecture** where processing takes place in nodes that may receive, post and multiplex sensor data, control, state, planning, actuator, and other messages. Despite the importance of reactivity and low latency in robot control, ROS itself is not a real-time OS (RTOS). It is possible, however, to integrate ROS with real-time code.

So what is ROS exactly? It is a framework that helps building robot applications. It is basically a large set of software components with an own build system. It is not an operating system, therefore it needs one under it. Although it may be installed on Debian and Windows 10 as well, its main target OS is Ubuntu. For this reason, the chosen OS for the computers in the project became Ubuntu (versions Artful and Bionic are supported).

All ROS applications have a graph architecture, meaning that they are considered as a set of separate processes (nodes) that are connected via messages. These message are

[2] As the short descriptions on ROS pages are pretty straightforward, I am going to quote from them whenever possible.

published on topics, which are message bus names. The standard defines some popular message types, but supports custom messages as well. The communication through messages is based on the publish-subscribe model, a publisher provides some data on a topic in the form of messages that all subscribed nodes can read. The implementation of the message-passing, the used networks layer and the transmission and reception of the messages are hidden from the application. ROS handles the delivery between different processes and even different hosts.

What ROS guarantees is a stable build system and architecture that helps developers create multi-process applications relatively easy, with reliable message-passing over a configurable transport layer (TCP by default). However, these features would not be enough for developers to consider ROS a relevant candidate for a robotic application framework, easy-to-integrate tools and a detailed documentation are also necessary. Fortunately, ROS does include these.

Its documentation is comprehensible and covers most topics that developers may need to look into. Besides that, a large series of tutorials are also available on their website about getting started with ROS and building simple applications, and also about deeper topics that give major insight to the reader about how ROS tools and features can be used effectively. These tutorials usually demonstrate given features using example applications that are also available for installing.

1.2.2 Useful ROS tools

ROS comes with numerous tools that are plug-and-play compatible any applications, due to that fact that ROS apps are just a set of independent nodes - once opened, these tools become nodes in the ROS graph and they can communicate with the other nodes via messages.

Most of these tools enable the developer to monitor the ROS graph, the existing nodes and the connections among them. Let me mention a few of these programs that proved themselves elementary during working with ROS.

The first one is rqt_graph, which is GUI tool for visualizing the ROS graph - the nodes and the message connections among them. Figure 1.4 shows part of the *vr-car* ROS graph represented using rqt_graph. The nodes are shown in the ellipses, and the arrows among them are the topics, on which the messages are sent.

A useful command-line tool is rostopic, which may be used to list, search and monitor message topics. For example executing *rostopic list* while the *vr-car* project is running produces the following output:

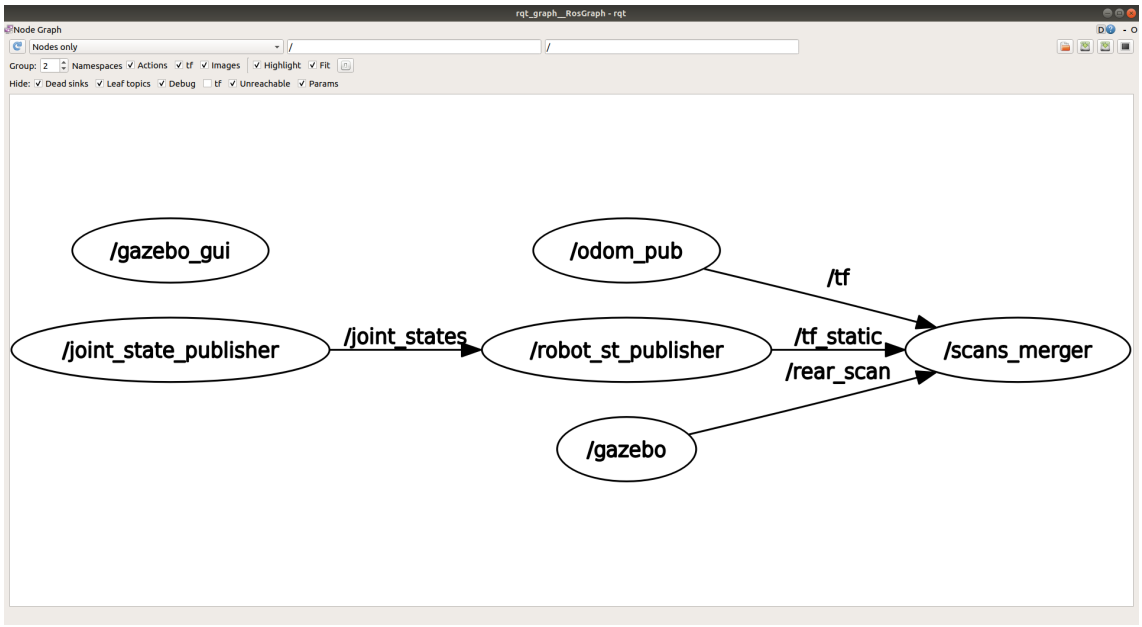


Figure 1.3: Visualizing the ROS graph using rqt_graph

```
soma@Soma-Ubuntu:~$ rostopic list
/Global_planner_wrapper/plan
/camera1/VRcar/camera
/camera1/VRcar/camera/compressed
/camera1/VRcar/camera/compressed/parameter_descriptions
/camera1/VRcar/camera/compressed/parameter_updates
/camera1/VRcar/camera/compressedDepth
/camera1/VRcar/camera/compressedDepth/parameter_descriptions
/camera1/VRcar/camera/compressedDepth/parameter_updates
/camera1/VRcar/camera/theora
/camera1/VRcar/camera/theora/parameter_descriptions
/camera1/VRcar/camera/theora/parameter_updates
/camera1/camera_info
/camera1/parameter_descriptions
/camera1/parameter_updates
/clicked_point
/clock
/front_scan
/gazebo/link_states
/gazebo/model_states
/gazebo/parameter_descriptions
/gazebo/parameter_updates
/gazebo/set_link_state
/gazebo/set_model_state
/globalTrajectory
/initialpose
/joint_states
/lidar_pcl
/lidar_scan
/move_base/global_costmap/costmap
/move_base/global_costmap/costmap_updates
/move_base/local_costmap/costmap
/move_base/local_costmap/costmap_updates
/move_base_simple/goal
/obstacles
/odom
/particlecloud
/path_planner/path_visualization
/path_planner/path_visualization_array
/rear_scan
/rosout
/rosout_agg
/static_grid
/static_grid_updates
/tf
/tf_static
/vrcar/filtered_control
```

For robotic applications, coordinate transformations and frame tracking in time are essential. Fortunately, ROS has an off-the-shelf solution for coordinate frame handling, this program is called tf. Its documentation briefly explains that

> tf is a package that lets the user keep track of multiple coordinate frames over time. tf maintains the relationship between coordinate frames in a tree structure buffered in time, and lets the user transform points, vectors, etc between any two coordinate frames at any desired point in time.

And for visualizing the tf tree, rqt may be used.

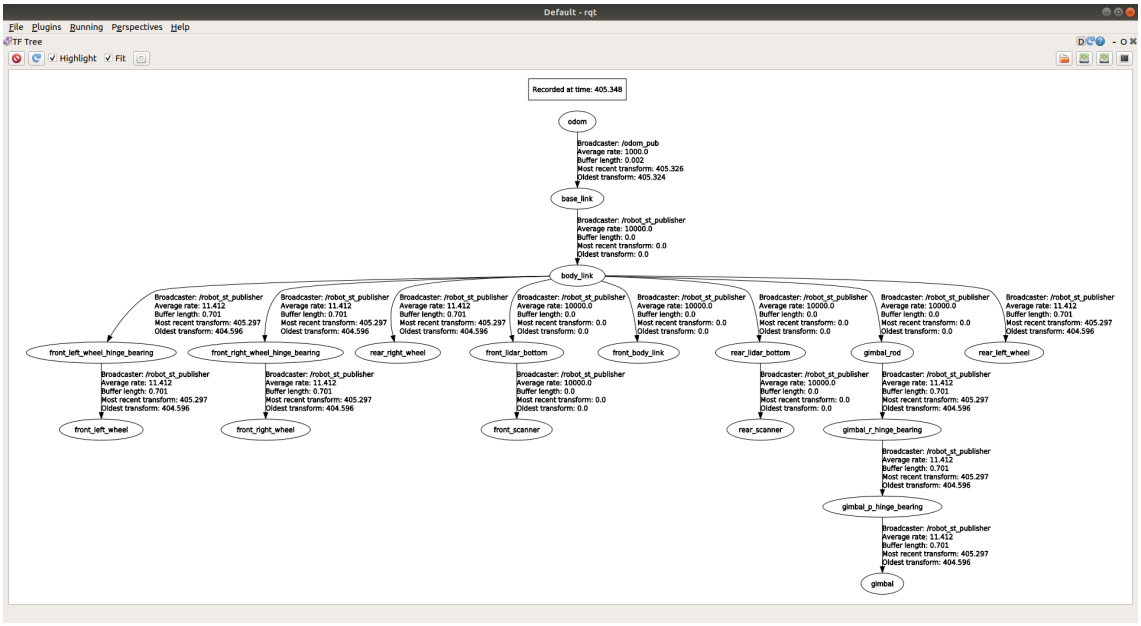


Figure 1.4: Visualizing the tf tree using rqt

In many cases, algorithm testing consists of various iterations, and it is handy to provide the program the same inputs over and over. For this purpose, ROS has a tool called rosbag, which is able to record and play back messages. This way, the whole ROS graph may be simulated by replaying the node outputs. Debugging algorithms is much easier using rosbag recordings, than having to recreate the same situations running the actual nodes.

The tools mentioned above were programs that help developers to monitor or interfere with the ROS graph and node communications. However, there are standalone applications, too, that are, in a way, independent from the ROS graph overall, they work like regular graph nodes that have certain inputs and outputs.

One of most essential standalone program is rviz, a 3D visualization tool. rviz supports a wide range of elements that it can display (camera images, maps, markers, point clouds, robot models, etc.), and the developers also have the option to implement custom display plugins for their own types.

For vehicle applications such as the *vr-car* project, the typical displayed types are robot models, odometry, trajectories and maps. Just imagining the simplest car driving project we can, displaying the car's position and orientation could be quite useful. Unsurprisingly, rviz supports the visualization of vehicle odometry (see Odometry display type), and according to its documentation:

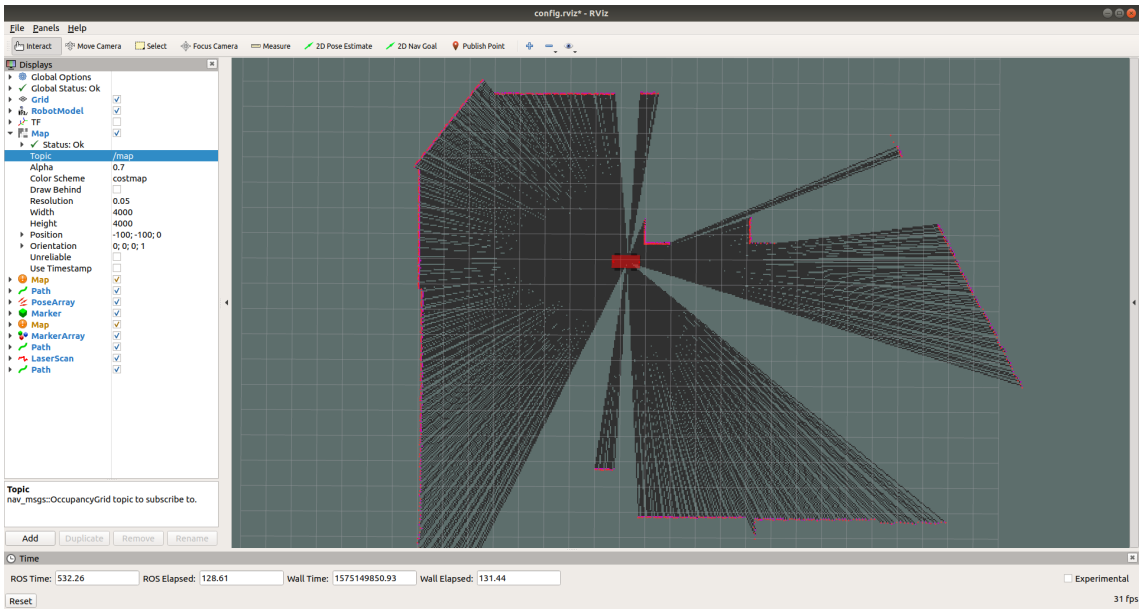


Figure 1.5: rviz

> The Odometry display accumulates a nav_msgs/Odometry message over time, showing them as arrows.

So one of the nodes in the project's graph needs to publish the odometry messages, and no further coding is needed, rviz may be opened and odometry will be displayed.

Testing in real life, with the real target host may be difficult and expensive. My diploma project is a perfect example for this, because the test car that operates as the host for the ROS nodes was very rarely accessible for me, as I was developing the project from home. Therefore, having to test my algorithm with the actual *vr-car* vehicle would have been a struggle. Fortunately, Gazebo, which is an open-source 3D robotics simulator, can be connected to ROS easily. With the help of this software, I was able to simulate the car (alongside the sensors and actuator fixed on it) on my computer. What Gazebo provides according to their website is

> a robust physics engine, high-quality graphics, and convenient programmatic and graphical interfaces.

Simulating in Gazebo is relatively more difficult than visualizing in rviz. At every simulation start, Gazebo needs to be given a world describing the simulation environment, and objects to place in the world (e.g. in the simulation represented in figure 1.6 the world consists of the boxes, while the object placed in the world is the red car).

The simulated world is defined by an SDF file. Quoting from theirwebsite:

> SDF is an XML format that describes objects and environments for robot simulators, visualization, and control. Originally developed as part of the Gazebo robot simulator, SDF was designed with scientific robot applications in mind. Over the years, SDF has become a stable, robust, and extensible format capable of describing all aspects of robots, static and dynamic objects, lighting, terrain, and even physics.

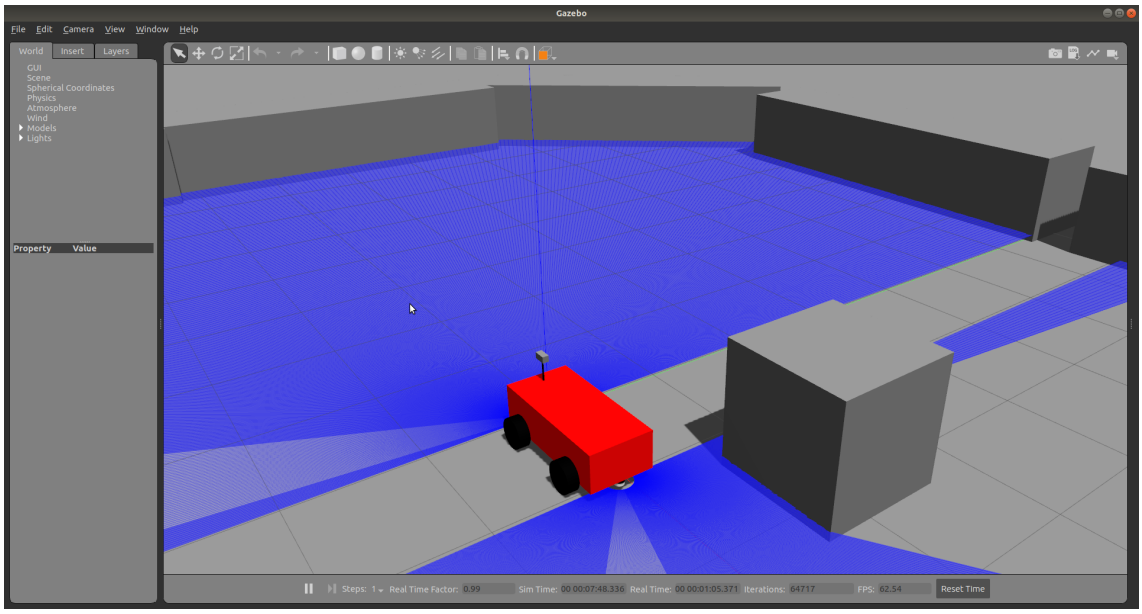


Figure 1.6: Gazebo

And the objects are defined by Unified Robot Description Format (URDF) files. These are also XML descriptors that define the structure of an object by links and joints. A basic URDF file containing a single cylinder looks like the following (the example is taken from one of the URDF tutorials provided by ROS):

```
<?xml version="1.0"?>
<robot name="mokifej_foni">
  <link name="base_link">
    <visual>
      <geometry>
        <cylinder length="0.6" radius="0.2"/>
      </geometry>
    </visual>
  </link>
</robot>
```

Both the SDF world and the URDF models can be opened, edited and saved in Gazebo, thus making the creation and update of these files easy. These models were already created by another member of the *vr-car* project, so I was able to simulate the car without any difficulty or extra work, which was a great help during the development of my algorithms.

In the next sections I am going to explain the ROS graph and the message-passing between nodes using a simplified model of the *vr-car* project, while also describing the features of the application.

1.2.3 Remote drive

The first (and probably simplest to comprehend) feature of the car is the ability to control it remotely, using either a PC keyboard, a joystick or a set of racing wheel and pedals.

In order to make this possible, the car definitely needed a node that controls the car's actuators according to an input actuation. For simplicity, let's assume that this input actuation (a ROS message) contains a target speed and wheel angle that the module has to handle as a reference. Having direct control of the car's motors (the accelerator DC

motor and the steering servo) and of the necessary sensors (e.g. a rotary encoder for measuring speed) the control is manageable.

Besides this control module, the graph contains another node, that reads driver interactions (key strokes, joystick state or steering wheel and pedal positions), converts these to driver commands (target actuations) and publishes the correspondent messages that the control node listens to. Note, that these input nodes are implemented separately for the input types - there is a module for keyboard input, one for handling the joystick and a third one for reading the racing wheel and pedal data[3].

With one input module being active, the graph now contains two processes - the input and the control nodes. These two nodes are communicating through a message that contains speed and wheel angle reference values.

1.2.4 *vr-drive*

Another feature of the car is that the user drive it using one of the input methods listed above while 'seeing what the car sees'. This is implemented by reading the video stream from the camera fixed on the vehicle and transferring it to the user's PC. The user can see the images via an Oculus Rift VR headset. To make it more realistic, the camera on the car is moved in a way that it follows the user's head movements.

1.2.5 2D mapping

As figures 1.1 and 1.2 show, the car is equipped with a front and a rear 2-dimensional, 360 degree LIDAR. The sensors are hard to see because they are fixed under the chassis. The reason for them being hidden is that they are meant to detect low-height objects as well, therefore they could not have been placed on the top of the car, which would be the usual placement of LIDARs. But as the front LIDAR cannot see backward effectively because it is covered by the wheels (and the same goes for the rear LIDAR looking forward), both LIDARs are needed in order to have a 360 degree view angle. But because the two LIDARs are placed on two different locations on the car, creating one map from their measured values is not trivial. To solve this issue, a special node called *scans_merger* has been added to project, with the task to merge the scan data from the front and rear LIDARs, thus creating a joined scan.

This scan needs to be evaluated and and kept track over time, while the car is moving, in order to create a map from them. This algorithm could have been initialized as part of the project, but a popular ROS package, gmapping is able to do that outstandingly[4].

1.2.6 3D mapping

3D mapping is implemented using the front LIDAR, which is also a 2-dimensional, 360 degree sensor, but it is placed on a gimbal, and is tilted forward and backward with 0.5 Hz frequency. Because of the tilting movement, the LIDAR scans a different plane in every step. And merging these planes creates a 3D map (see figure 1.7).

[3]The racing wheel and the throttle and brake pedals are actually handled on a separate Windows host, and their positions are forwarded to another host running ROS.

[4]The advantages and drawbacks of gmapping will be discussed in section 2.

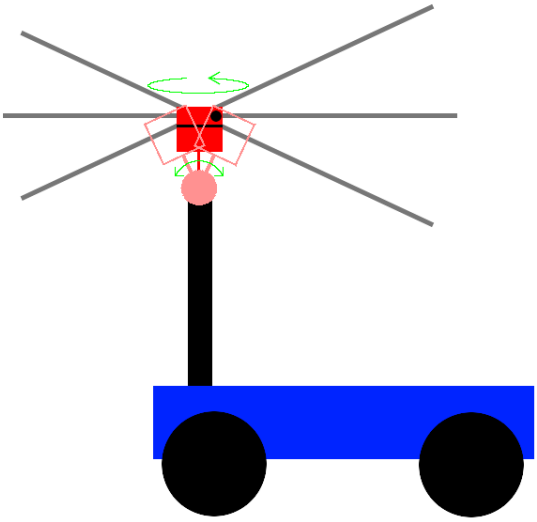


Figure 1.7: Creating a 3D map by tilting a 2D LIDAR

1.2.7 Navigation

The project has a goal to make the car able to navigate from its actual position to a desired target position in the map. In order to do that, it needs a reliable map of its environment. As I mentioned earlier, gmapping produces a good-quality map, therefore the navigation algorithm uses that as its base. Looking at the navigation node as a black box (other members of the project have implemented and are still working on it, its internal logic is not relevant for my diploma project), it receives a destination point in a ROS message, and produces a path in another. Both the input and output messages are standard ROS messages that I will explain in detail later. For providing the destination point for the algorithm, there exists several possibilities, but one of the most straightforward method is through rviz. The visualization tool is not only capable of displaying content, but it has a feature that the user can select a '2D Nav Goal' in the view, and the selected point will be sent out on a specific topic into the graph. A very effective and elegant way is to combine the map drawing with this feature for creating an interface that enables the user to decide where the next destination point should be, by clicking somewhere in the map. Needless to say, giving a target position for the navigation node is implemented using this method.

Now that the main features of the *vr-car* has been listed and briefly explained, let me introduce what my part of the project was, and how it fits into the existing graph. So far, the graph consists of the nodes represented in figure 1.8. Note, that this image is a simplified version of the actual *vr-car* graph, it does not contain all the existing nodes, and some nodes have been merged into one for the sake of simplicity.

1.2.8 Avoiding moving obstacles

My task as a diploma project was to design, implement and tune two nodes, that embed themselves into the existing graph.

The first one is a mapping node. While there are numerous mapping implementations available in ROS (see gmapping for one example), none of them handles moving objects

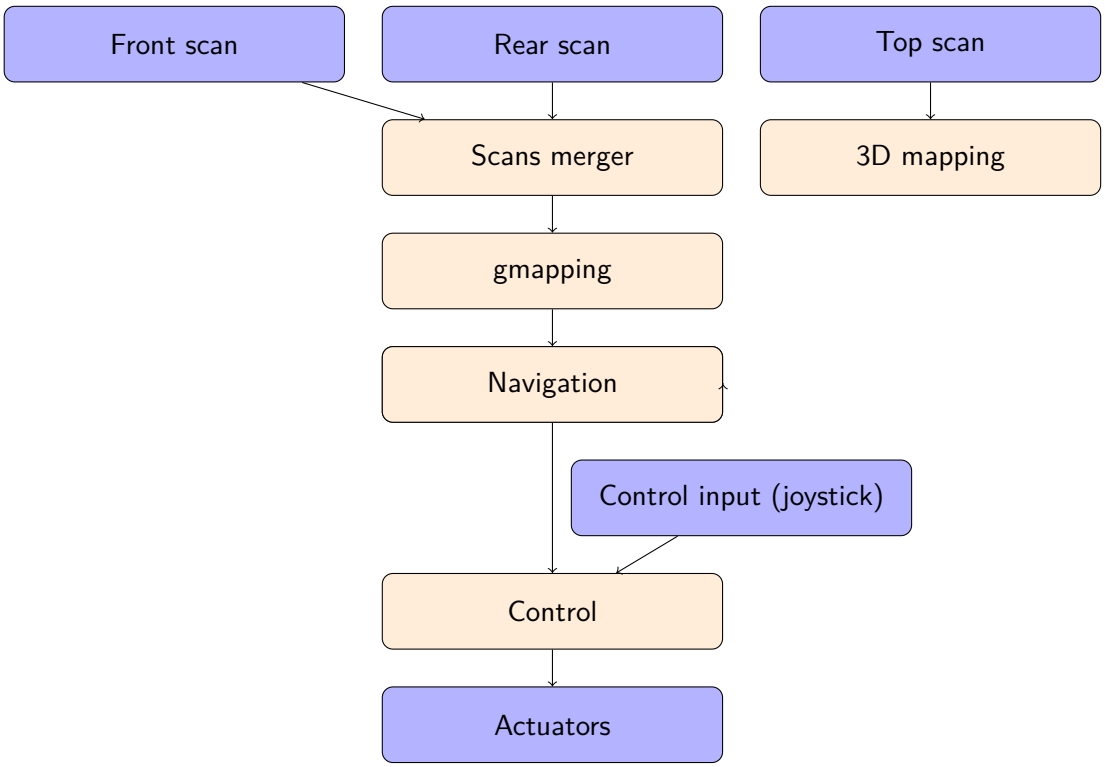


Figure 1.8: The original *vr-car* graph

correctly (I am providing a detailed reasoning in section 2). So the goal of the mapping node was to separate the static points from the moving objects in the map.

The other node was a local trajectory planner node, that (based on the mapping nodes output of static and dynamic objects) performs local obstacle avoidance considering not only the position but also the velocities of the surrounding obstacles.

The *vr-car* graph with the two additional nodes is represented in figure 1.9. The mapping nodes uses the raw front and rear LIDAR scans as its inputs for the map creation. The local trajectory planner node cuts the connection between the Navigation and the Control nodes, because it overrides the pre-calculated path based on the static and dynamic obstacles in the car's environment.

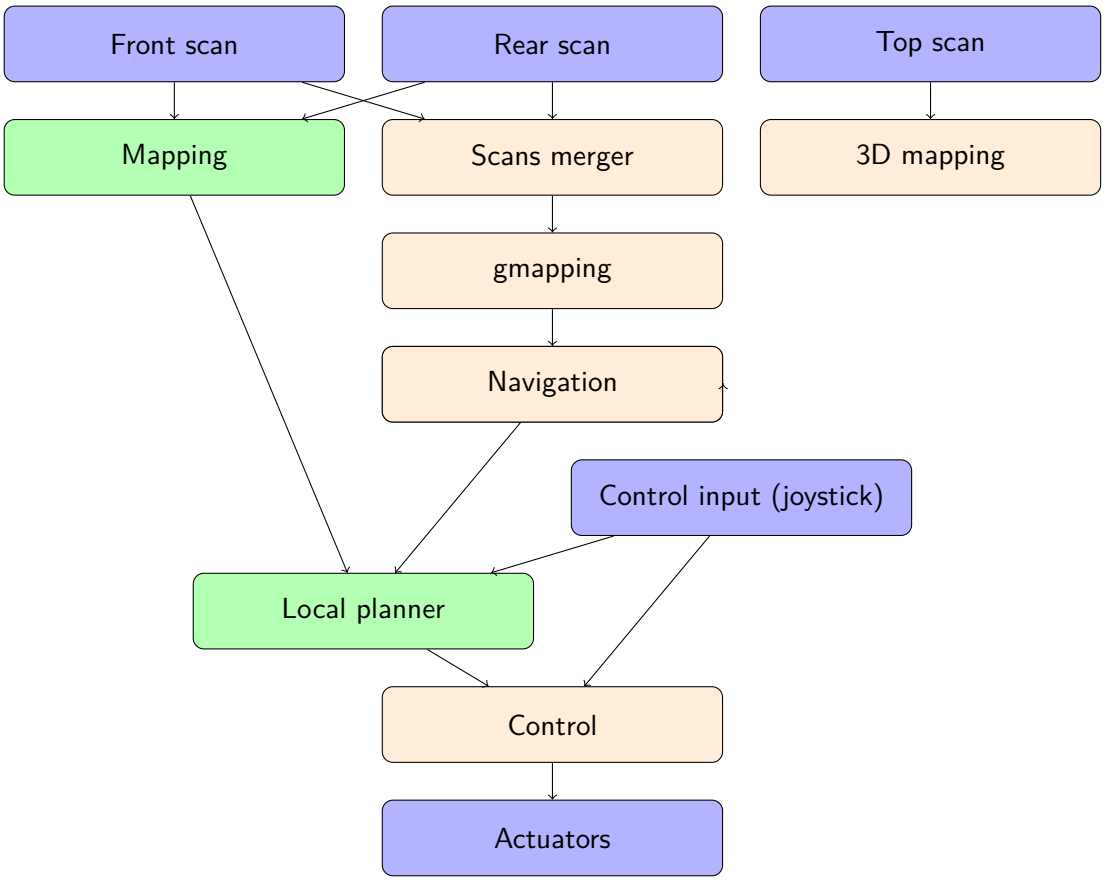


Figure 1.9: The modified *vr-car* graph

Chapter 2

Mapping

2.1 Existing implementations

This chapter describes the process of building a separate static and a dynamic map (containing non-moving and moving obstacles, respectively) based on radial distance measurements (in this case provided by a LIDAR). Isolating the static and the moving obstacles from each other has its difficulties but also its advantages. First of all, the implemented local track planner calculates safer, more optimized paths if it is informed of both the static and the dynamic obstacles along the path. Secondly, SLAM (Simultaneous localization and mapping) algorithms work better if the their input contains only static objects in the space, because they build an internal map of the world. Passing detections of moving obstacles to a SLAM algorithm may lead to worse localization quality, as they may ruin this map.

As a first sub-task of the mapping project, I checked if any implementation is available already, that can handle dynamic objects. The only possible candidate was gmapping, which is a popular ROS package, used in a wide variety of applications that require map-building and localization. Moreover, the *vr-car* project already used gmapping, when I started working on it. Its SLAM algorithm takes LIDAR measurements as its input and generates an occupancy grid (a 2D map) of the car's environment. By using this map it is able to make corrections to the car's odometry-based position and orientation, which is usually inaccurate. I tried out the package, and the result maps were promising, the generated map was insensitive to the car's longitudinal movements and its rotations. But unfortunately, gmapping's SLAM does not support dynamic objects. See 2.6.1 for further details. Therefore, gmapping could not be used as the producer of the static map. But that didn't mean it couldn't be used for its second feature, localization. Note, that the mapping implementation I made is not a SLAM algorithm, it is not able to make corrections to the car's pose. So for that purpose, I still needed the help of gmapping, which proved to be very reliable at localization. But the static map-building needed to be implemented internally.

2.2 Method

This chapter describes the process of building a separate static and a dynamic map (containing non-moving and moving obstacles, respectively) based on radial distance measurements (in this case provided by a LIDAR). Isolating the static and the moving obstacles

from each other has its difficulties but also its advantages. First of all, the implemented local track planner calculates safer, more optimized paths if it is informed of both the static and the dynamic obstacles along the path. Secondly, SLAM (Simultaneous localization and mapping) algorithms work better if the their input contains only static objects in the space, because they build an internal map of the world. Passing detections of moving obstacles to a SLAM algorithm may lead to worse localization quality, as they may ruin this map.

As a first sub-task of the mapping project, I checked if any implementation is available already, that can handle dynamic objects. The only possible candidate was gmapping, which is a popular ROS package, used in a wide variety of applications that require map-building and localization. Its SLAM algorithm takes LIDAR measurements as its input and generates an occupancy grid (a 2D map) of the car's environment. By using this map it is able to make corrections to the car's odometry-based position and orientation, which is usually inaccurate. I tried out the package, and the result maps were promising, the generated map was insensitive to the car's longitudinal movements and its rotations. But unfortunately, gmapping's SLAM does not support dynamic objects. See 2.6.1 for further details. Therefore, gmapping could not be used as the producer of the static map. But that didn't mean it couldn't be used for its second feature, localization. Note, that the mapping implementation I made is not a SLAM algorithm, it is not able to make corrections to the car's pose. So for that purpose, I still needed the help of gmapping, which proved to be very reliable at localization. But the static map-building needed to be implemented internally.

In order to create two disjunct maps, one static and one dynamic, the key element of the process is the separation of the moving and non-moving obstacles of the measured points. After determining these two disjunct set of points, the maps can be converted to any desired or required format. Static maps are usually published as occupancy grids, while dynamic obstacles need to present information about their speed vector. Occupancy grids do not group the grid points according to their probabilities, therefore they do not know about the obstacles' borders and areas[1]. Dynamic obstacles however can be either represented as separate points with their own speed vectors, or groups of points, each groups having one speed vector. I chose the latter representation, thus publishing groups that contain a set of points (all the points, ideally) of the same obstacle. This way there is a one-to-one relationship between moving obstacles and groups. The separation and grouping methods of the mapping process is shown on diagram 2.1.

The diagram consists of 3 sub-graphs - these are also marked on the diagram. The first, and most important is the separation and grouping of dynamic points. The second sub-graph describes the additional calculations that need to be done for the dynamic obstacles, and consists the documentation for the message structure defining these dynamic obstacles. And the third one is about the static map that is built from the static points. The sections under 2 explain these sub-graphs in detail.

[1]In this project, 2D LIDARs were used, therefore the measured objects were seen as 2D shapes that have areas, not 3D objects that have volumes

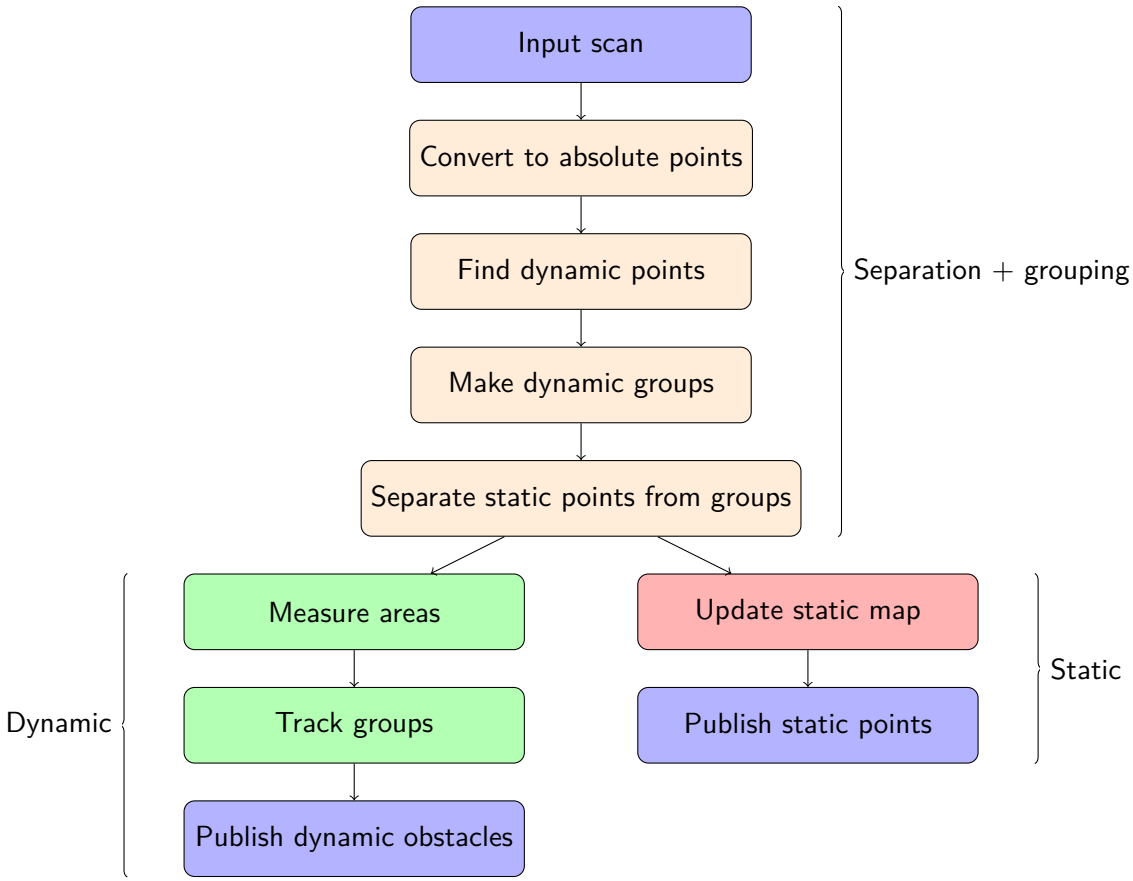


Figure 2.1: Mapping method

2.3 Input parameters

The following input parameters can manipulate the mapping node:

- **LIDAR_MAX_DIST** [meter] The maximum measurable distance by the LIDAR(s). Depends on the LIDAR's characteristics.
- **MAP_RES** [meter] The resolution of the built static map. Increasing this this value helps building a better quality map, but also increases the node's resource need.
- **MAP_SIZE** [meter] The size of the built static map. Increasing this this value helps building a better quality map, but also increases the node's resource need.
- **SINGLE_LIDAR** [boolean] Defines if the node's input is one LIDAR scan, or a separate front and rear scan that needs to be handled separately.

2.4 Separation and grouping

This section describes the method of separating dynamic and static points of the input scan. This step is essential in the pipeline of creating two disjunct maps.

2.4.1 Input scan

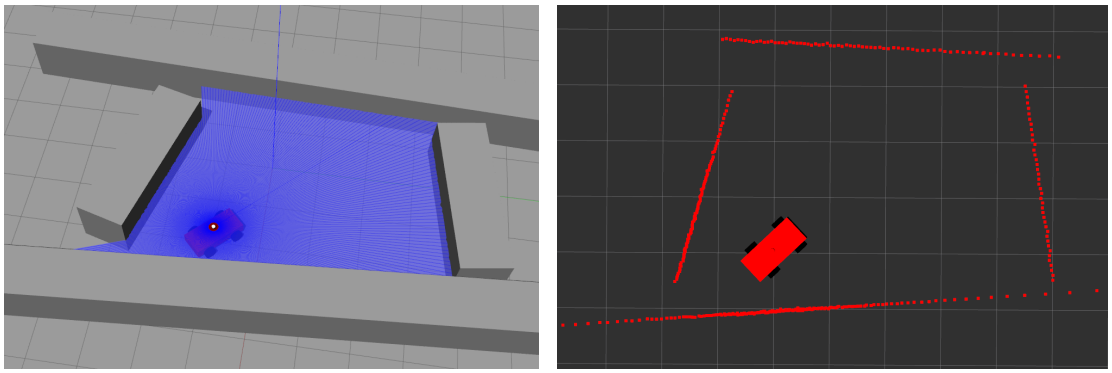
The input of the mapping algorithm is a 2D LIDAR scan, consisting of radial distance measurements. Two types of LIDARs were used in the project, RPLidar A1 and RPLidar A2. Both types have the following specifications:

Scan rate	10 Hz
Sample rate	8000 samples/sec
Distance resolution	0.2 centimeters
Angular resolution	1°
Detection range	12 meters

The devices proved to be reliable, and for a project of this volume, their frequencies, resolutions and ranges were adequate. Their output is a ROS message of type *sensor_msgs/LaserScan*, which has the following structure:

```
std_msgs/Header header
float32 angle_min
float32 angle_max
float32 angle_increment
float32 time_increment
float32 scan_time
float32 range_min
float32 range_max
float32[] ranges
float32[] intensities
```

Among numerous other things, the message contains an array of radial distances (*ranges*), that are the measurements themselves. The messages can be visualized easily using rviz.



(a) Gazebo simulation **(b)** rviz visualization

Figure 2.2: LIDAR scan

2.4.2 Absolute points

For static map-building, absolute points[2] are needed in the space, and static-dynamic point separation also uses absolute points, so firstly, these radial distances need to be transformed. However, the internal static map and the static-dynamic separation use different coordinate systems. In order to understand the need for this, let's take a look at figure 2.3.

[2] Absolute points are not relative to the car, but to a fix base point.

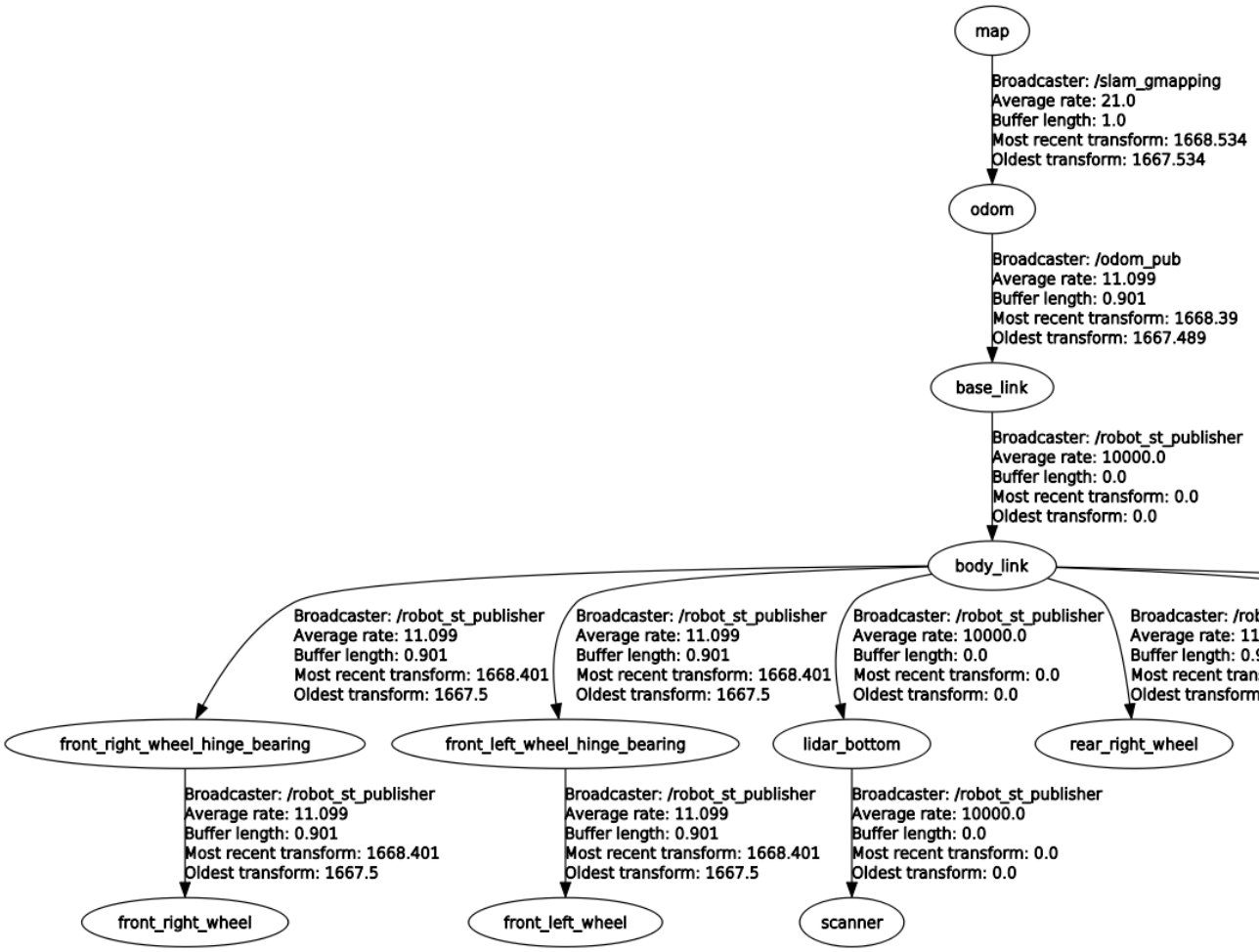


Figure 2.3: Coordinate frames

As the figure shows, there are 3 coordinate frames that are important to the current case. The *odom* frame is the car odometry's coordinate system. It is calculated from values measured on the vehicle, such as servo position and speed. The odometry is also published as a message of format (*nav_msgs/Odometry*), that has the following structure:

```
std_msgs/Header header
string child_frame_id
geometry_msgs/PoseWithCovariance pose
geometry_msgs/TwistWithCovariance twist
```

The message contains the car's position and orientation in *pose* and its linear and angular speed in *speed*.

The *scanner* frame is the LIDAR's coordinate system. The transformation between *odom* and *scanner*[3] is basically the pose of the LIDAR, relative to the car. The *map* frame is the output of gmapping's localization. Basically, gmapping takes the car odometry and the LIDAR scans as its input, and corrects the odometry using SLAM. As a result the difference between frames *map* and *odom* will increase with time.

The internal static map in my implementation uses this corrected *map* frame as the base for its points, so that its error is minimized. But unfortunately, the static-dynamic separation

[3]The implementation supports one and two LIDARs as well. In case of the two-LIDAR setup, instead of the *front_scanner* there is a *rear_scanner* and a *scanner* frame.

method cannot use this frame. The reason is that this frame does not get updated on every new scan, but with a much slower frequency. Therefore there are 'jumps' in the *map* frame, which would cause false dynamic point detections (see 2.4.3). To avoid this undesired situation, the static-dynamic separation uses the *odom* frame as its base, which is more continuous than the *map* frame.

Therefore, two transformations are needed for each input point. The transformations are calculated using tf, which maintains the relationship between coordinate frames in a tree structure (see figure 2.3) buffered in time, and makes the transformation of points, vectors, etc possible at any desired point in time.

2.4.3 Dynamic points

The next step is similar to a creating a subtraction image in image processing, where subtracting two images, taken from the same position but at a different time, results in an image that amplifies the movements between the snapshots. Previous methods[3] that I studied before starting my implementation are also based on this step. The aim of this algorithm unit is basically the same: selecting the points from the input that are likely to be part of a moving mass. This is done by finding the points among the current measurements that have not been present in the previous ones.

This step is best explainable in practice. Let's assume that the position and orientation of the LIDAR is fix, and one object (e.g. a car) in the detected area is moving. Figure 2.4 shows this situation. On the image, the pale contour represents the previous pose of the object, and the current state is blue.

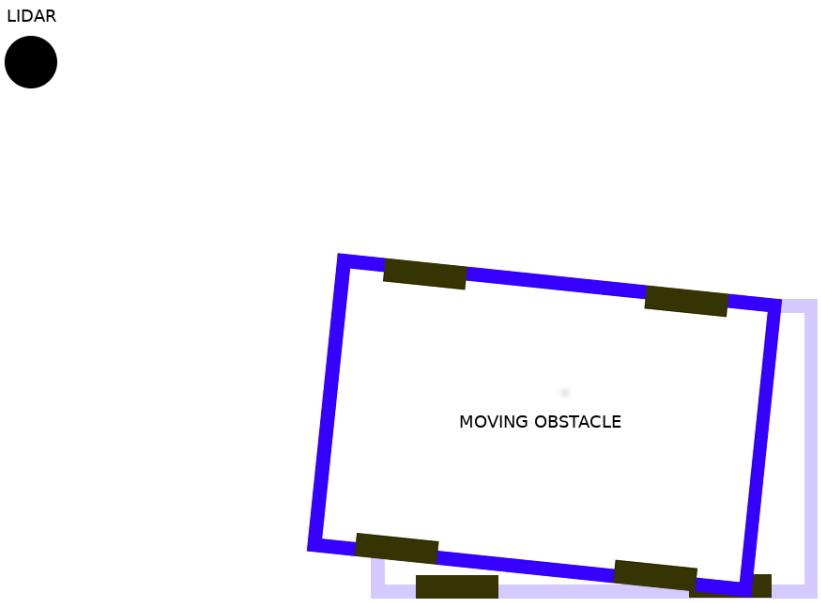


Figure 2.4: The obstacle is moving

Figure 2.5 shows how the LIDAR detects the moving object in two different time-snaps. I marked the points corresponding to the current position with red color, and the ones of the previous measurement are pale red. As it is suggested in the figure, the measurements are far from ideal, the detected points are noisy.

To understand the method of dynamic point detection, I removed the LIDAR, its virtual rays and the obstacle's contour from the image, thus leaving leaving only the measured

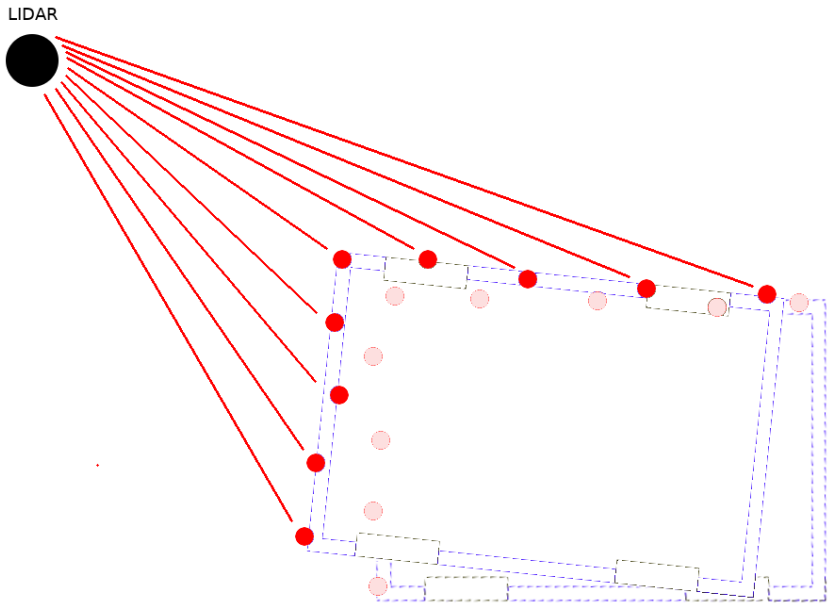


Figure 2.5: The detected points of the moving obstacle

points of the two time-snap. The result, which is basically a time-buffered array of 2D points, is presented on figure 2.6. The algorithm iterates through each point in the current measurement, and checks if any point in the previous measurements[4] is within its compliance radius[5].

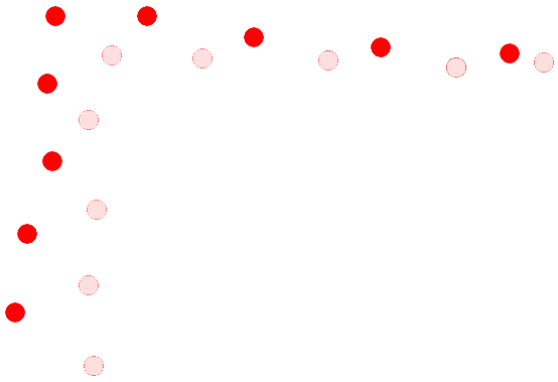


Figure 2.6: The detected points of the two time-snaps

The measured points with their compliance radiuses are presented in 2.7. The possible dynamic points, that have no points from the previous measurement within their radius, are marked with blue, and their compliance radius with yellow. The possible static points are marked with red, along with their radiuses. Note, that these points are *possible* dynamic and static points. The final classification will be preceded be multiple filtering mechanisms and point grouping, but this is the base for finding dynamic objects.

[4]The number of measurements to 'look up' is configurable.

[5]The compliance radius is a maximum allowed distance between measurements representing the same physical point but measured in different time-snaps. If the distance between two measurements is greater than this value, the measurements are assumed to represent different points of the space. The compliance radius is calculated for each measurement separately, and it is proportional to the distance of the LIDAR and the measured point.

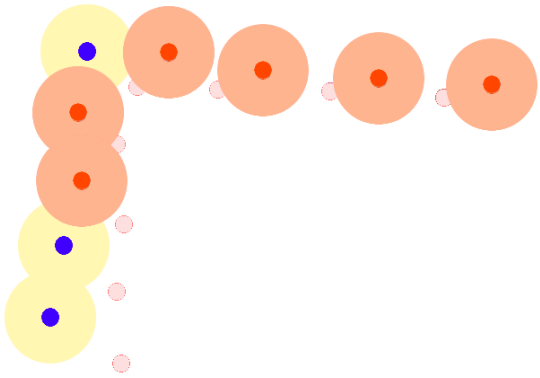


Figure 2.7: The compliance radiuses and the dynamic points

Several conclusions can be drew from 2.7. The one, and probably most important is that with right parameterizing (number of 'look-up' measurements, compliance radius, etc) this method amplifies the positive[6] and negative[7] changes of the scans. The second remark is that the method does not detect all the points of a moving object. However, due to measurement noise, false detections may happen, and non-moving points may be marked as dynamic.

Therefore, the result of this separation step needs to be filtered. I implemented a filtering step that only keeps those samples in the set of dynamic points, of which the left and right neighbour in the scan has also been marked dynamic. It can be easily compared to erosion in image processing, which is able to remove peaks from the image. Let's take a look at 2.8. The possible dynamic points are marked with blue. The filtering mechanism iterates through each of them, and check both neighbours. As it can be easily read from the figure, point A has two neighbours that have been marked static, therefore it will be removed from the set of dynamic points. However, points B and C both have dynamic neighbours (each other), so they will not be filtered out. This algorithm removes the single-size false detections from the result of the separation.

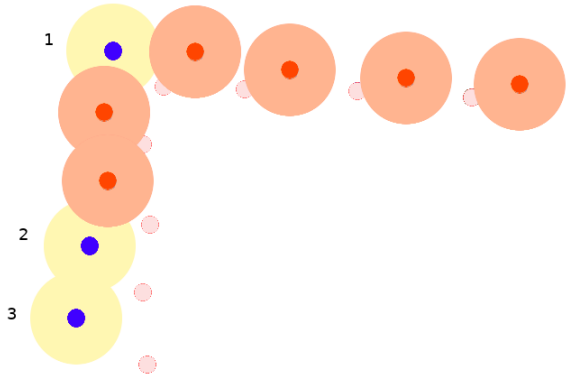


Figure 2.8: Point A will be filtered out

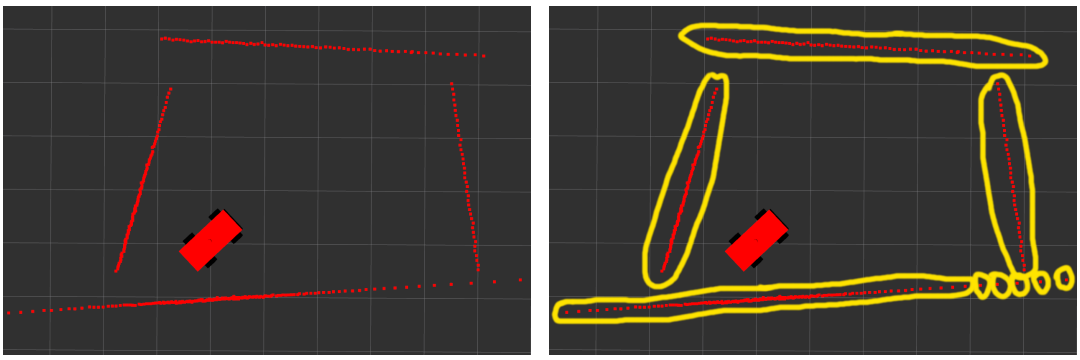
[6] Previously a distant background was detected, but now a closer object appears.

[7] Previously a close object was detected, but now it disappears, and the distant background becomes visible.

2.4.4 Grouping

After the filtering, we can safely assumed that the elements in the dynamic point set are in fact points of a moving obstacle. The next step is grouping the corresponding dynamic points, and also adding those points to these groups that were previously marked as static, but they are part of a moving group. For example in 2.8, after the separation and filtering, only points B and C were marked dynamic, but actually, all the points in the image are points of the same object, that is moving. The grouping algorithm needs to be able to add these other points to the group as well.

The grouping algorithm is based on the supposition that coherent points are close to each other, while the distance between points of separate objects is larger. With a few exceptions, this statement holds its stand, by practice it proved to be a reliable base for grouping. The algorithm itself is quite simple, it separates the list of measurements (including the static and dynamic points as well) into subsets - groups. A new group is started when the distance between two adjacent points is larger than the permitted maximum within a group. This maximum is an empirical constant, with experiments of valid use cases. The result of the step is shown in 2.9b.



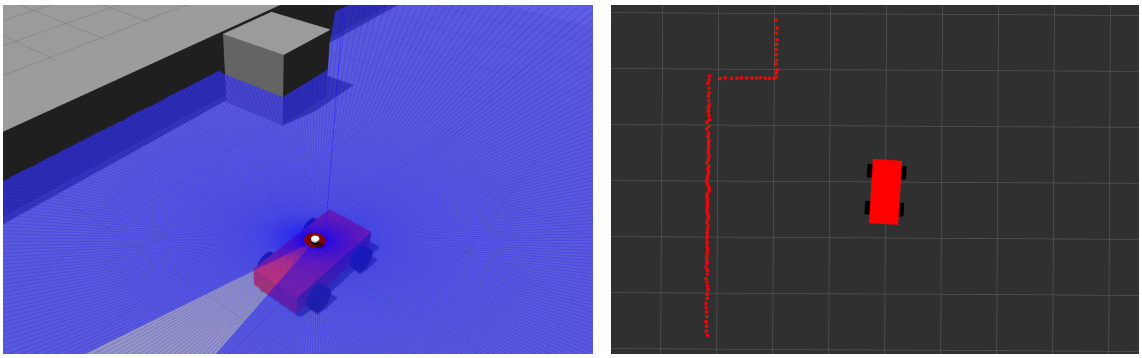
(a) Gazebo simulation **(b)** Groups

Figure 2.9: Distance-based grouping

As it is clearly visible on the image, far points that are correspondent to the same object may be separated from each other, and grouped one by one - see the bottom right corner of the picture. However, the further an object is from the LIDAR, the less important it is for the local trajectory planner, which needs to avoid close obstacles. But still, they are not valid groups, so as the last step of the grouping mechanism, these single-point groups will be removed from the list of groups. But before doing that, there is still one special case that needs to be handled by the grouping unit - the *wall following car*. Let's imagine the situation represented in 2.10. The detected obstacle (represented by the box) is moving alongside the wall, and within the maximum group point distance. Therefore, the grouping algorithm will join the groups. This wouldn't be a problem if the box wasn't moving, but as soon as it changes its position, grouping it together with the wall would ruin the mass center and speed vector calculation, described in 2.5.2.

To avoid this problem, the algorithm detects if an obstacle is unusually large, and cuts off long 'tails' from the group. Figure 2.11 explains this step in practice.

Let's assume that the algorithm iterates through the points visualized in the figure from the right. The distance between points C and D is larger that the allowed maximum group point distance, therefore point C starts a new group. Until the program reaches point A, all the points are added to this group, because the distance between adjacent



(a) Gazebo simulation **(b)** Lidar scan

Figure 2.10: The *wall following car*

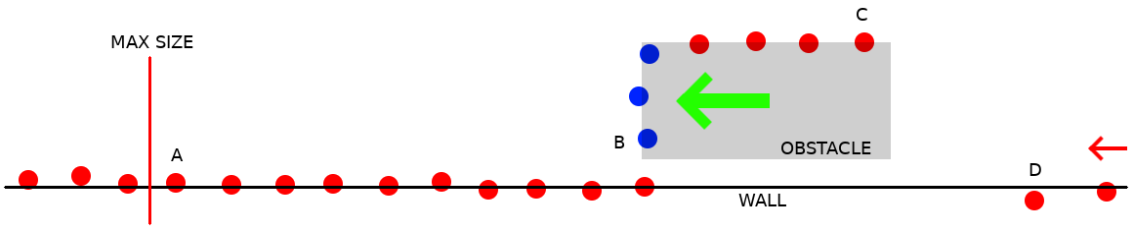


Figure 2.11: Points on the left of B will be cut off

points is always lower than the maximum group point distance, but after point A, the algorithm detects that the group is larger than the permitted limit. So the iteration turns around, and all the points are getting removed from the group, until the first dynamic point is reached - in the current example this would be point B. As a result, only the points between (and including) point B and point C will be grouped together.

As an undesired side-effect, this step may also remove some of those points from the group that are in fact parts of the moving object but were marked as static, because the removal only stops at the first *dynamic* point.

2.4.5 Separation of static points and dynamic groups

The previous step has successfully sorted the measured points into groups, but it is undecided yet if these groups are moving or not. This decision is made by checking the number of dynamic points in each group, which must exceed the required minimum in order to the group being marked as dynamic.

This was the last step of the separation of the groups, from this point, static points and dynamic groups will be handled differently.

2.5 Dynamic obstacles

After the dynamic groups have been separated from the static points, additional information needs to be calculated for them, so that the local track planner algorithm can use their data for its obstacle-avoidance feature. 3 pieces of information are needed for each object, its position, size and speed vector. All of these are calculated from the object's

mass center, which is the average of the group points. This mass center is considered to be the object's position in space.

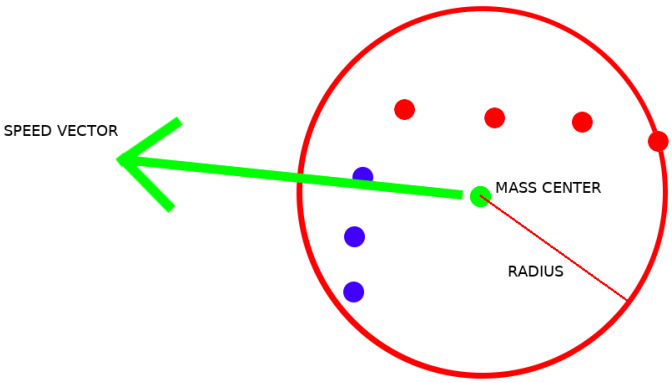


Figure 2.12: The calculated information for dynamic obstacles

2.5.1 Areas

First, the groups are substituted by bounding circles. The circle's radius is the distance between the farthest group point and the mass center. This simplification results in easier path calculation in the local planner, but also drops every information known about the obstacle's shape. For long objects, for instance, this method provides a very unrealistic image of the object, because the bounding circle is much larger than the obstacle itself. But for usual shapes such as cars, the loss that the representation causes is less than the processing time we save with the simplification.

2.5.2 Tracking

Speed vectors are calculated from the change of the mass center between measurements. But in order to be able to check the mass center of a group in any previous measurement, object tracking is needed. The tracking is done by estimating each obstacles' current position based on their previous positions and speed vectors, and finding the best fit among the current groups. Filtering is also needed for such false detections when a previously detected object is not recognized for a few cycles. Without filtering, these obstacles would disappear from the list of groups for a period of time and then they would reappear, but tracking would fail. Therefore the algorithm keeps these objects alive for a given number of measurement periods, updating their positions with their last known speed vector in every cycle.

After the objects have been tracked, their speed vectors can be calculated. A filter is applied here, as well, for smoother obstacle paths.

2.5.3 Publishing dynamic obstacles

Dynamic obstacles - as outputs of the mapping node - are published on several topics, responsible for containing information for further processing. The array of dynamic objects is published on the *dynamic_objects* topic of type *environment_builder/DynamicObjectArray* (an array of *environment_builder/DynamicObject*s, which is a custom type for representing the detected obstacles in the map. It's format is the following:

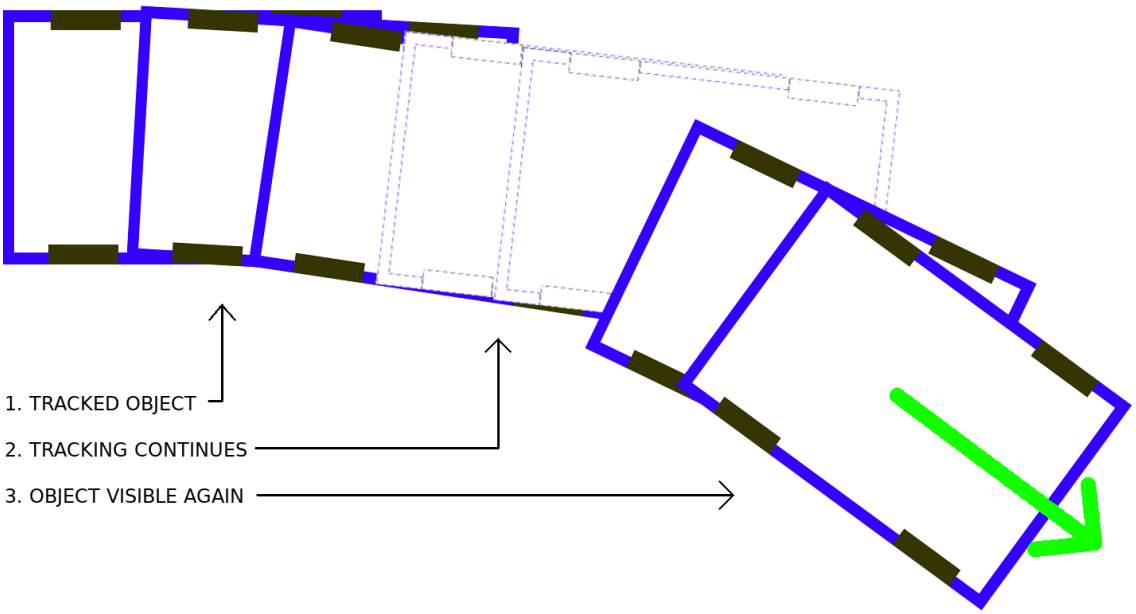


Figure 2.13: Object tracking

```
float32 radius
geometry_msgs/Pose pose
geometry_msgs/Twist twist
```

And the structure of *environment_builder/DynamicObjectArray* is simply:

```
DynamicObject[] objects
```

The node also publishes messages that are useful for visualization and diagnostics or debugging. On topic *obstacles*, visualization messages of type *visualization_msgs::MarkerArray* are published. This type of message contains an array of *visualization_msgs::Marker*s, which look like the following:

```
uint8 ARROW=0
uint8 CUBE=1
uint8 SPHERE=2
uint8 CYLINDER=3
uint8 LINE_STRIP=4
uint8 LINE_LIST=5
uint8 CUBE_LIST=6
uint8 SPHERE_LIST=7
uint8 POINTS=8
uint8 TEXT_VIEW_FACING=9
uint8 MESH_RESOURCE=10
uint8 TRIANGLE_LIST=11
uint8 ADD=0
uint8 MODIFY=0
uint8 DELETE=2
uint8 DELETEALL=3
std_msgs/Header header
string ns
int32 id
int32 type
int32 action
geometry_msgs/Pose pose
geometry_msgs/Vector3 scale
std_msgs/ColorRGBA color
duration lifetime
bool frame_locked
geometry_msgs/Point[] points
std_msgs/ColorRGBA[] colors
string text
string mesh_resource
bool mesh_use_embedded_materials
```

With its *type* set to either *SPHERE* or *ARROW*, these markers represent the areas and speed vectors of the objects, and are easily visualizable in rviz.

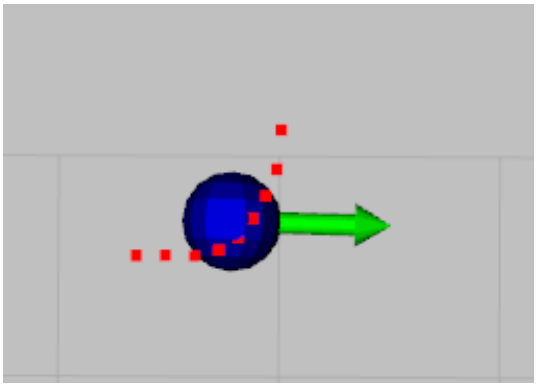


Figure 2.14: Visualizing moving objects

And on topic *diff_grid*, *nav_msgs/OccupancyGrid*s are published. This is topic is purely for diagnostic and debugging purposes. It represents the dynamics of the map compared to the previous measurement, using an occupancy grid. This means that the occupancy grid will have high values where something changed in the map, an will store low values where the map is still. The algorithm that searches for dynamic objects uses this grid as its basis. The structure of the message is as follows:

```
std_msgs/Header header
nav_msgs/MapMetaData info
int8[] data
```

Where *data* stores the actual grid values and *info* is a *nav_msgs/MapMetaData*, storing meta information about the grid:

```
time map_load_time
float32 resolution
uint32 width
uint32 height
geometry_msgs/Pose origin
```

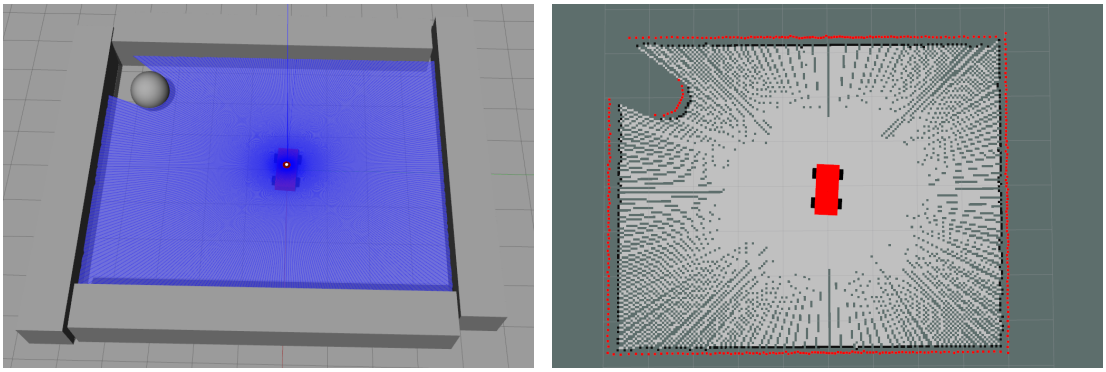
2.6 Static points

Static points used for two separate purposes. The first one is building a static map, the second is localization with the help of gmapping.

2.6.1 Static map

After separating the dynamic groups from the scan points, only the static points are left. These points do not change their positions with time[8], so they can be added to the static map.

For static map-building, gmapping has already been mentioned as a candidate, but unfortunately, its algorithm does not recognize changes in the environment. In practice, if an object has been detected and placed in its map, it will not be erased from there, even after the object has moved away. Figures 2.15 and 2.16 demonstrate the problem.



(a) Gazebo simulation **(b)** Map created by gmapping

Figure 2.15: Initial state - ball is still

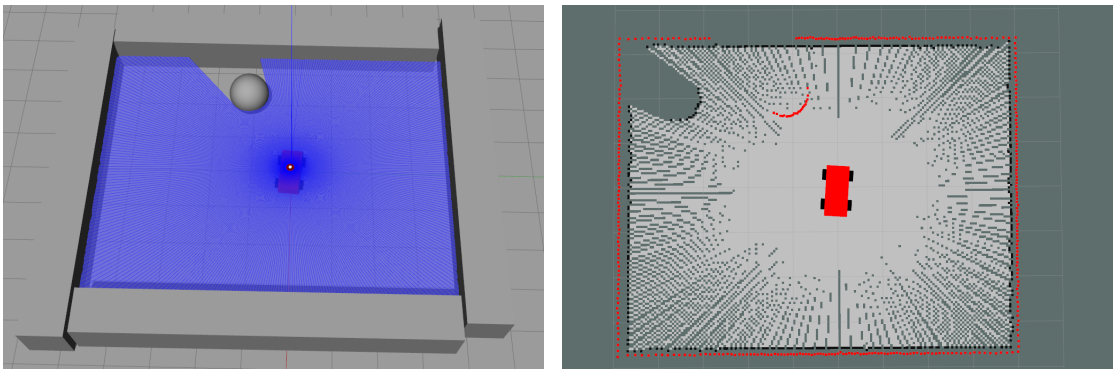
2.15 shows a situation where the ball is standing still. The mapping algorithm of gmapping successfully creates a map (in the form of an occupancy grid[9]) that marks the place of the ball as occupied. However, when the ball starts moving (see 2.16), the map does not change, because the algorithm cannot handle moving obstacles.

Due to the above problem, static map-building needed to be implemented internally, which is basically a 2D occupancy grid. The grid points' possible values are the following:

- Occupied (value: 100)
- Unknown (value: 50)
- Free (value: 0)

[8] Except those obstacles that start moving only after they have already been detected as sets of static points. But these objects do not ruin the quality of the map, either.

[9] An occupancy grid represents a map in an evenly spaced field of probability values representing if the grid points are occupied by an obstacle



(a) Gazebo simulation **(b)** Map created by gmapping

Figure 2.16: Moving state - ball has changed position

I chose the most straight-forward way of map-building - each static measurement introduces an occupied point and a free ray in the map. The map's size and resolution can be set as input parameters (see MAP_SIZE and MAP_RES in 3.3). 2.17 visualizes the situation. The green and red grids are the car's current position and the measured point.

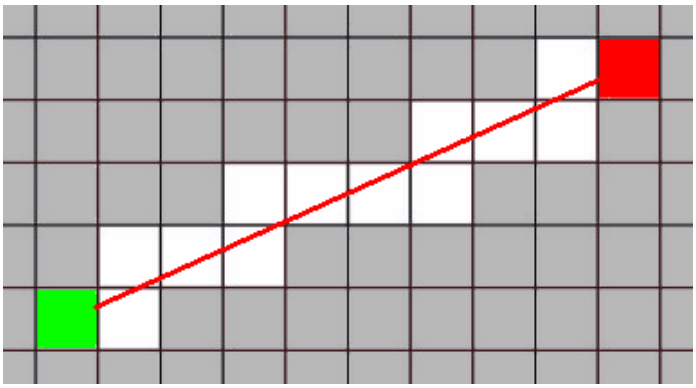


Figure 2.17: Introducing a free ray

2.6.2 Publishing static points

The built static map is published on topic *static_grid*, which is a nav_msgs/OccupancyGrid, just like the previously introduced *diff_grid* topic's messages.

With the publishing of the maps the mapping node has finished its task, and the focus now shifts to the motion planning node which calculates actuations that drive the car to the destination without hitting any static or dynamic objects.

Chapter 3

Motion planning

3.1 Existing local planner algorithms

After building a map of the static and dynamic objects surrounding the vehicle, the next step is motion planning, which consists of two sub-tasks - global trajectory planning and local obstacle avoidance. As a result of global planning, a trajectory is created that - without taking the moving obstacles into consideration - leads the car to the target configuration, without making it collide with any static objects in the way. Local obstacle avoidance takes this trajectory as its input, and updates the car's actuators (acceleration and steering) to follow this trajectory, while also preventing collisions with dynamic objects.

As a part of my diploma project, I implemented the latter, a local obstacle avoidance algorithm, that relies on the previously created static and dynamic maps, and gets its input trajectory from a global planner node[1]. According to [6], there exists a wide range of local planner algorithms, and they can easily be grouped by their complexity:

- Velocity-based methods, most of the times combined with the dynamic window approach
- Predictive and probability-based methods
- Complex methods based on visual detection and AI
- Other methods

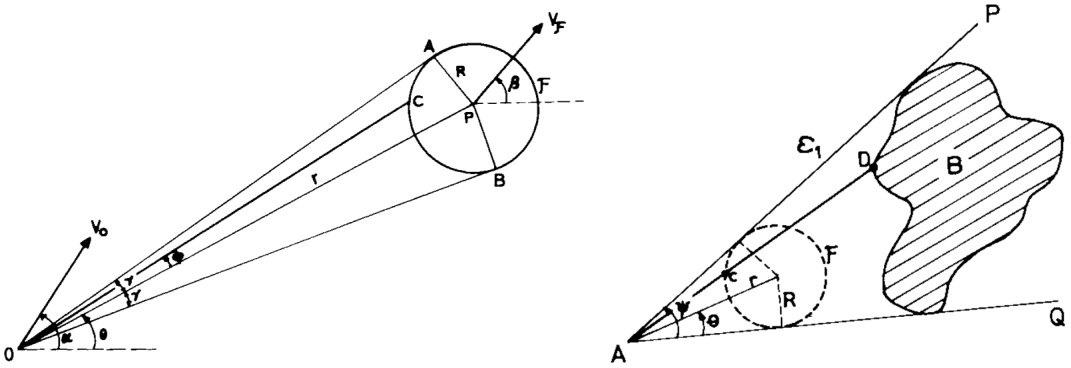
Out of these methods, velocity-based algorithms are by far the most popular, due to their being relatively easy to comprehend and implement and because of their low hardware requirements. As the mapping node and the local planner that I designed need to run on a processor with limited resources, I also chose a velocity-based obstacle avoidance method, using velocity obstacles, with dynamic windowing.

After reading briefly about several motion planning algorithms, I found two different approaches that I thought was worth looking into more deeply.

One was the collision cone approach explained in [2]. This algorithm creates a cone that is designed in a way that it encloses the vehicle and also the obstacle it needs to avoid (see

[1] Implementing a global motion planner was not part of my diploma project, but is considered a necessary condition for the local planner to work properly.

figure 3.1). The cone defines how probable a collision is to happen using a given speed relative to the other object.



(a) Between point and circle **(b)** Between point and irregular geometry

Figure 3.1: Collision cones

The other method was to use velocity obstacles (described in [4] and in [5]). These methods are based on the following statement: *Assuming a given a given vehicle configuration (position, orientation, speed and wheel angle), the time until collision with a given static or dynamic object in the map can be calculated.* Therefore, velocity obstacle-based methods calculate this collision time for all surrounding objects and for all vehicle velocities. thus creating a velocity obstacle map, or forbidden velocity map (also referred to as dynamic velocity space in [6]) that defines how safe given velocities are. Another implementation is explained in [1], where the above mentioned two approaches are mixed while creating a virtual plane that enables the car to navigate in a dynamic environment.

3.2 Method

The planner algorithm I implemented uses velocity obstacles to determine which configurations of the car are safe. Using the dynamic window approach is advised in this step, as it filters out non-reachable velocities before collision-time calculations, thus decreasing the execution time of the algorithm.

Given this velocity obstacle map, the algorithm has a basic knowledge of the reachable velocities and their level of safety. These safety levels provide a good starting point for further calculations, trajectory and collision estimations. My algorithm converts the forbidden velocity map's collision times to safety factors, but also takes the destination point's position and the target speed into consideration when selecting the next actuator outputs. The next graph shows all the sub-tasks of the local trajectory planner algorithm.

The diagram consists of 4 sub-graphs - these are also marked on the diagram with different colors. The orange sub-graph describes the environment-independent sub-tasks of the algorithm - finding the next destination point, updating the target actuation and the dynamic window. The yellow one is about the method of reducing the size of the static map by filtering out those points that are not needed for the trajectory planning. The red side sub-graph shows that for dynamic objects, trajectory calculation is also necessary besides filtering. And the last one (populating the bottom area of the graph, marked with green colour) contains the main planning logic. The sections under 3 explain these sub-graphs in detail.

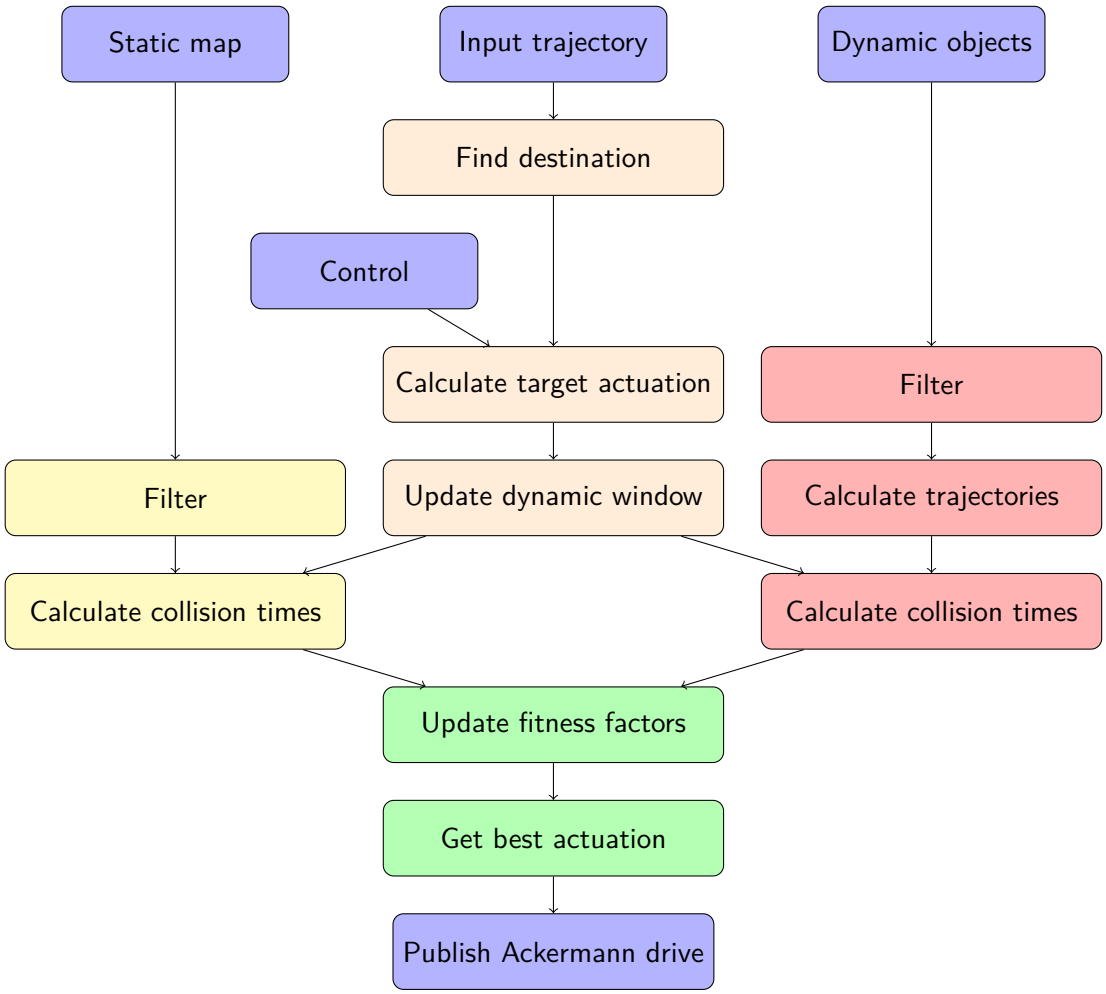


Figure 3.2: Motion planning method

3.3 Input parameters

The following input parameters can manipulate the motion planning node:

- **CAR_FRONT_REAR_WHEEL_AXIS_DIST** [meter] The distance between the front and rear wheels axles of the car.
- **CAR_LENGTH** [meter] The length of the car.
- **CAR_WIDTH** [meter] The width of the car.
- **COLLISION_CHECK_TIME_STEP** [millisecond] The resolution of the iterative collision check.
- **DYNAMIC_WINDOW_ANGLE_RESOLUTION** [radian] The wheel angle resolution of the dynamic window.
- **DYNAMIC_WINDOW_SPEED_RESOLUTION** [meter/sec] The speed resolution of the dynamic window.
- **MAX_ACCELERATION** [meter/sec2] The maximum permitted acceleration.
- **MAX_DYNAMIC_OBSTACLE_DISTANCE** [meter] The radius of the dynamic obstacle filter.
- **MAX_SPEED_BWD** [meter/sec] The maximum permitted backward speed.
- **MAX_SPEED_FWD** [meter/sec] The maximum permitted forward speed.
- **MAX_STATIC_OBSTACLE_DISTANCE** [meter] The radius of the static

3.4 Target actuation and dynamic window

In every loop (at each new incoming map data), the target actuation and the dynamic window are updated. The target actuation describes the desired speed and steering wheel angle for the next step, therefore the desired linear and angular speed vectors. The dynamic window is independent from the target actuation, but depends on the current actuation. It contains a set of speeds and wheel angles that are reachable by the car within the next step. The width and height of the window (the maximum speed and wheel angle change in one step) are defined by the mechanical characteristics of the car. So basically, the aim of this sub-task is to provide the algorithm a target actuation, and a set of reachable actuations.

When I implemented the target actuation update mechanism, that leads the car according to the input trajectory, I had several options to choose from. The first option was to split the trajectory into line-segments, and implement and tune a steering angle controller (e.g. a PD controller) to follow this line. I dropped this idea to prevent the re-tuning that would have been needed for each target vehicles (simulation, real-life test vehicles). Another option was the 'invisible string' the 'dog bone' method, where a target destination point is always in front of the car, and the car is always trying to reach this point, as if the point was pulling it by a string. The target actuation's wheel angle defines a trajectory curve, that goes through the destination point. If the destination point is always on the trajectory, and in front of car (within the right distance range), the car is going to follow the trajectory. I chose the latter (the 'invisible string') method.

3.4.1 Finding the next destination point

This method requires a destination point in each iteration. This point is preferably on the trajectory, in front of the car, 'pulling' the car to the right direction. The destination point search start with finding the point in the trajectory nearest to the car's current position. The destination point needs to be in front of the car, but its distance from the car (which is nearly equal to its distance from the previously found nearest point) is not trivial (see figure 3.3).

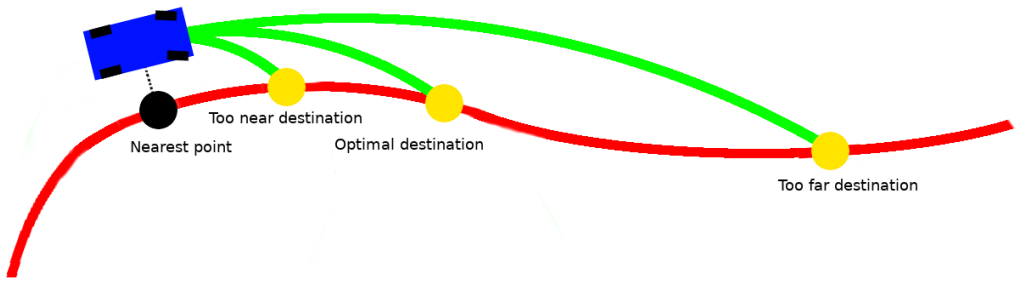


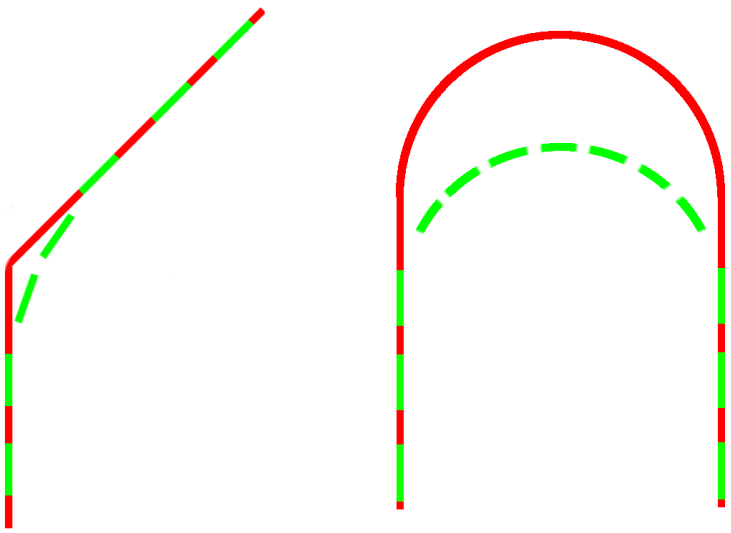
Figure 3.3: Updating the destination point

Choosing a too near destination will result in the car starting to oscillate around the target trajectory with an increasing amplitude, until a point where the angular difference between the car and the trajectory will be too large for the car to find the path again. This situation is displayed in figure 3.4.

However, the destination point must not be chosen too far, either, because it straightens the curves of the trajectory, rounds its sharp peaks, and adds an offset to the long circular

Figure 3.4: Oscillation around the target trajectory

sections. Figure 3.5 demonstrates these effects. The effect shown on the left side might even be advantageous, but the right-hand side effect is certainly not.



(a) Rounding sharp peaks **(b)** Rounding curves

Figure 3.5: Trajectory rounding

After empirical testing, the optimal distance of the destination point and the nearest trajectory point (marked with yellow and black dots in figure 3.3) proved to be around 4 times the distance between the car's front and rear axles (see CAR_FRONT_REAR_WHEEL_AXIS_DIST in 3.3). Note, that this value largely depends on the dynamic window's maximum angular velocity limit and the car's mechanical characteristics (e.g. using a faster steering servo or a 4-wheel steering car would reduce the optimal distance). In order to follow the straight segments of the target path before curves, I implemented the length of the invisible string (the distance in front of the nearest trajectory point) to change dynamically, according to the car's current distance from the trajectory. This tweak suppressed the rounding effect mentioned above, but not entirely. This is now considered a limitation of the algorithm's efficiency.

3.4.2 Calculating the target actuation

After the destination point has been selected, the next step is to calculate the target actuation - the speed and steering angle pair that would lead the car to the destination point. First, the target wheel angle is calculated using linear algebra (see figure 3.6). The target speed calculation is based on two factors. The first is the target speed limit, which we expect the car to keep, given no extraordinary circumstances. This should be set to a safe speed that the car can keep up during its travel (see MAX_SPEED_FWD and MAX_SPEED_BWD in 3.3). The actual target speed will never exceed this limit. But

the target steering angle may require this target speed to be lower, because the car can manage to reach higher speeds safely when going straight, than when in a sharp bend.

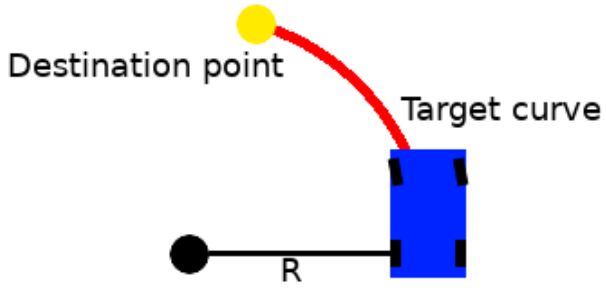


Figure 3.6: Calculating the target steering angle

3.4.3 External target actuation

For easier testing, the node also supports external target actuations, meaning that the target actuation is not calculated from a trajectory, but is directly provided by another node. The external actuation can be set on topic */vrcar/filtered_control_orig*, which is a topic of *ackermann_msgs/AckermannDrive* messages, that have the following layout:

```
float32 steering_angle
float32 steering_angle_velocity
float32 speed
float32 acceleration
float32 jerk
```

Among other fields (which are currently not used by the project) it has a *speed* and a *steering_angle* field - given in m/s and radians. These two fields are used as the target speed and wheel angle.

The program chooses between the target actuation modes dynamically, and the external target actuation has greater priority. So when messages are arriving on the external target actuation topic, they are the ones the algorithm uses, but when these messages cease to arrive, the node automatically switches to trajectory follow mode and calculates the next target actuation according to the given path.

3.4.4 Updating the dynamic window

In every iteration, the dynamic window must also be updated, which is independent from the target actuation, but depends on the current actuation, which is obtained by reading the odometry message published another node in the car. I have already described the structure of the *nav_msgs/Odometry* message earlier in 2.4.2.

The 'width' and 'height' of the window (which is defined by the maximum speed and wheel angle change in one iteration) mostly depends on the car's physical, mechanical and other characteristics (motor-wheel dead-time, maximum motor torque, speed controller's parameters, maximum angular velocity of the steering servo), but the environment's characteristics (air resistance, friction) also influence them. The maximum speed and wheel angle change are settable parameters (see MAX_ACCELERATION and MAX_WHEEL_ANGULAR_VELOCITY in 3.3). The dynamic window is represented

in figure 3.7. The red arrow shows the current actuation (speed and wheel angle pair). The dynamic window itself appears on the image as the grey area under the arrow heads. Every actuation within the dynamic window's area is reachable within the next step. When the current actuation is not near a hard limit, it is the center of the dynamic window.

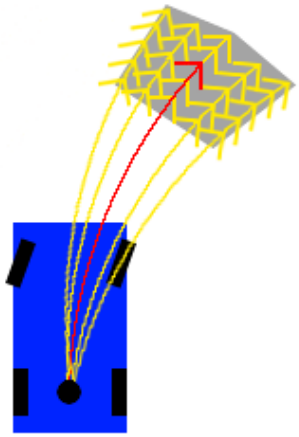


Figure 3.7: The dynamic window

However, the size of the dynamic window may reduce when its size is limited by the hard limits of speed and wheel angle. A perfect example for this situation is when the steering angle is at its maximum at either left or right direction (see MAX_WHEEL_ANGLE in 3.3), in which case changing the angle further to this direction is impossible. This results in the dynamic window getting halved - as only its original right or left side is reachable, depending on the current steering direction. It is trivial that in these cases such as the previous example, the current actuation is no longer in the center of the dynamic window. Another important characteristics of the dynamic window is its resolution - both speed and angular velocity, which are settable parameters (see DYNAMIC_WINDOW_SPEED_RESOLUTION and DYNAMIC_WINDOW_ANGLE_RESOLUTION in 3.3). Increasing the resolution leads to smoother control, but also increases the node's resource need and runtime.

The managing the dynamic window internally, the motion planning node also publishes the dynamic window's possible actuations in a *visualization_msgs::MarkerArray* on topic *available_velocities*, so that they can be viewed on-the-fly in rviz.

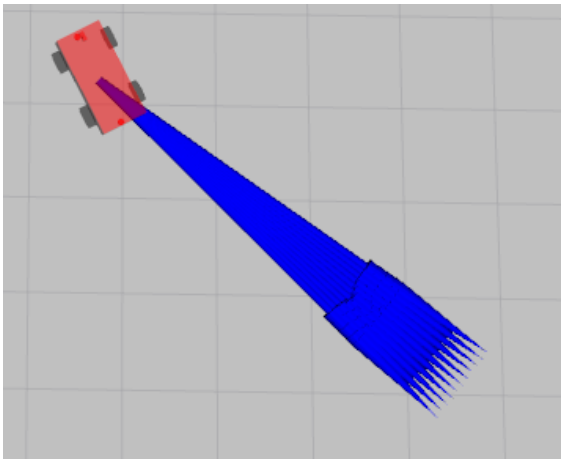


Figure 3.8: Available velocities in rviz

3.5 Static velocity obstacle map

Static points in the map are received from the mapping node in *nav_msgs/OccupancyGrid* messages on topic *static_grid* (see (see 2.6.2)), and they populate a very large percent of the car's surroundings, so handling them efficiently can boost up the algorithms runtime performance. Therefore, they are not joined with the dynamic points, but filtered separately, and the collision times are calculated in a way that is optimized for non-moving objects to gain performance.

3.5.1 Filtering static points

Separating the static points from the dynamic objects has both advantages and disadvantages for local motion planning. The most important advantage is that the collision check with static points is much simpler and faster than with dynamic obstacles. A great disadvantage, however, is that the static map contains lots of points, and running a collision check on all of them would cause the algorithm to be unacceptably slow. To prevent this undesired behaviour, a preparatory filter is applied to the map, removing those points from the map, that are too far from the car's current position to be worth keeping. Depending on the environment's characteristics, this step may reduce the run-time of the static collision times drastically - even to zero, if all the static points are far from the car (e.g. in a large room).

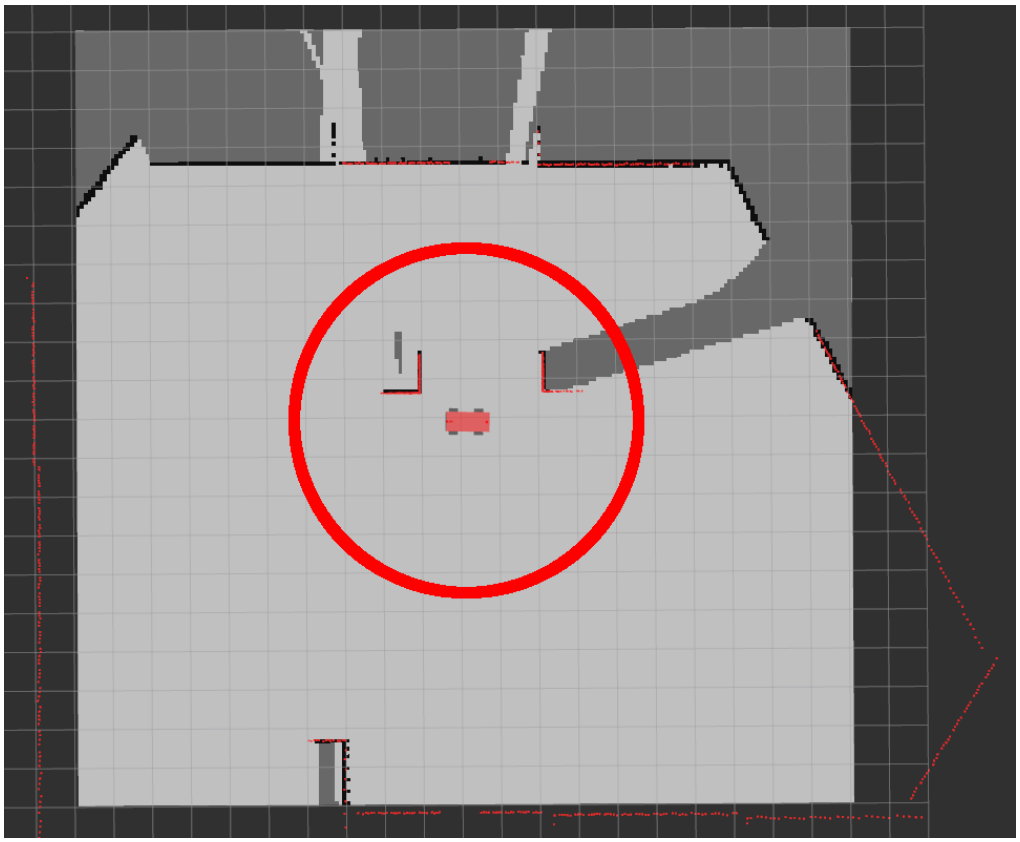


Figure 3.9: Static filtering

Figure 3.9 shows that the mapped environment of the car is a much larger territory than what the motion planning algorithm needs to check for collisions. The circle marks the area that is kept after the filtering. In this example, this area contains the two square-shaped

obstacles, all the other points in the static map are thrown. The size of the area worth keeping always depends on the actual application and the car's dynamic characteristics. But as a general rule of thumb:

$$R = \frac{v_{max}^2}{a_{max}}$$

v_{max}	The maximum speed
a_{max}	The maximum deceleration
R	The radius of the kept area

is a good guess. The maximum speed and deceleration and settable parameters (see MAX_SPEED_FWD, MAX_SPEED_BWD and MAX_ACCELERATION in 3.3). Optimization is possible using the car's speed vector, and filtering out those points that are very far from the current direction.

3.5.2 Static collision times

Collision times need to be changed for all surrounding obstacles in order to be able to build the dynamic velocity obstacle map. Due to the fact that the number of static points is usually high compared to the dynamic points, the collision times are calculated separately, so that the calculation can be optimized for static objects. Collision check needs to be executed for all the actuations within the dynamic window. The algorithm is the following.

First the car's expected movement for the actuation is predicted with a curve. When the wheel angle is very close to 0, this curve can be approximated with a straight line, in these cases the algorithm is simplified. However, now let us consider a scenario when the angle is not 0, and the trajectory will in fact be a curve of finite radius. Figure 3.10 shows this scenario.

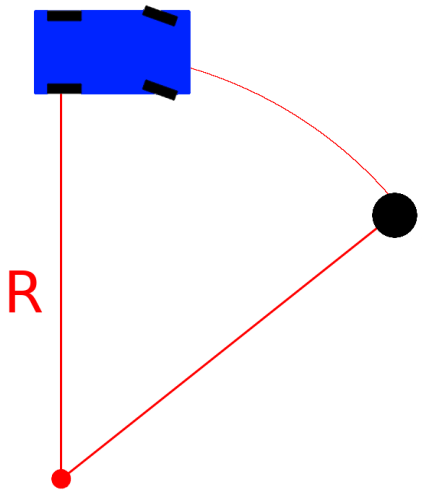


Figure 3.10: The car's expected trajectory is a curve

In order to get the collision time, first the algorithm needs to calculate the collision distance. This calculation is based on linear algebra. But in order to be able apply mathematical laws on the model, the objects need to be converted to simple geometrical forms. This simplification is represented in figure 3.11.

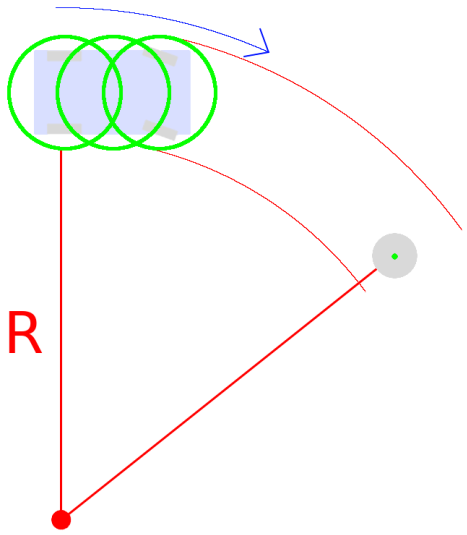


Figure 3.11: Simplifying object shapes

As it is clearly visible on the image, the car is approximated with 3 circles, fixed on the car's axles and its center, and the obstacle is simplified to one circle. In order for easier calculations, the obstacle is not handled as a circle of given radius, but as a point, and its radius is added to the radiuses of the car's circles. We also need to take into consideration, that the static map is always noisy, and we can never expect measurements to be noiseless, either. Therefore, a third value is added to the radiuses of the car's circles, which is the minimal distance that needs to be kept from static objects. Therefore the car circle radius is as follows:

$$r = \frac{w_{car}}{2} + r_{obs} + d_{min}$$

w_{car}	The width of the car
r_{obs}	The obstacle radius
d_{min}	The minimal kept distance
r	The modified car circle radius

The angular difference between the car and the obstacle (denoted γ) can be calculated easily using linear algebra (see figure 3.12).

After the angle is obtained, the collision distance and time can be easily calculated using the following equations:

$$s_{coll} = R \cdot \gamma$$

$$t_{coll} = s_{coll} \cdot v_{car}$$

R	The radius of the trajectory curve
γ	The angular difference
s_{coll}	The distance until collision
v_{car}	The car speed
t_{coll}	The time until collision

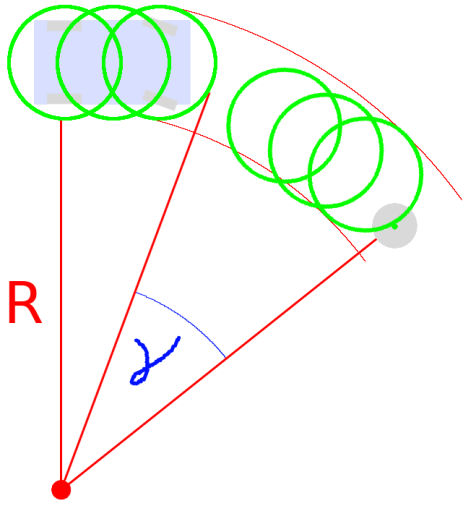


Figure 3.12: Angular difference between the car and the obstacle

Using the method described above, the collision times for the static obstacles in the map can be calculated. Doing this for all possible actuations will provide a static velocity obstacle map, which later can be used to grade actuations.

3.6 Dynamic velocity obstacle map

Dynamic obstacles are also obtained from the mapping node, on topic *dynamic_objects* (see 2.5.3). Handling dynamic obstacles is more difficult than handling static points, the algorithm, however, evaluating the moving objects is more straight-forward than static method using linear algebra. The reason for the dynamic algorithm being easier to comprehend is that it uses an iterative approach instead of linear algebra. (Linear algebra with multiple objects moving in non-straight paths is not trivial.)

3.6.1 Filtering dynamic objects

This first step when evaluating dynamic objects in the map is similar to static case - removing the too far obstacles. Trajectory modelling and collision check may be resource and time heavy, so handling as few dynamic objects as possible is crucial. The size of the area worth keeping depends on the actual application and the dynamic characteristics of the car and the other moving objects. But as a general rule of thumb:

$$R = 2 \cdot \frac{v_{max}^2}{a_{max}}$$

v_{max}	The maximum speed
a_{max}	The maximum deceleration
R	The radius of the kept area

is a good guess (note that the radius is twice as large as in the static case). Some optimizations using the car's speed vector can be made in this case, too, but keeping in mind that dynamic objects can approach from behind as well is important.

3.6.2 Trajectory calculation

Now that the irrelevant dynamic objects have been filtered out, the next step is to calculate the kept obstacles' trajectories. Trajectory calculation is done using an iterative method - for each obstacle, with a given time step (see COLLISION_CHECK_TIME_STEP in 3.3) the algorithm start iterating, updates the obstacle's position and orientation in each step and saves into an array of configurations - this array will become the trajectory itself. *But how many steps should be saved?* may be an interesting question, to which the answer is also: it depends. It depends on how the application logic uses these trajectories. For my application, the needed number of steps is:

$$N = \frac{3 \cdot \frac{v_{max}}{a_{max}}}{T_{step}}$$

v_{max}	The maximum speed
a_{max}	The maximum deceleration
T_{step}	The trajectory simulation time step
N	The number of needed steps

After the trajectories for all moving obstacles have calculates, the algorithm can move forward, and check for collisions with these objects.

3.6.3 Dynamic collision times

Calculating the collision times with the dynamic obstacles is not based on linear algebra, like in the static case, but is executed in an iterative way. For all possible actuations (actuations that are present in the dynamic window), a trajectory is calculated - the same way that it has already been calculated for the other dynamic objects in the map. Using this trajectory and the pre-calculated other trajectories the algorithm iterates through time, updates the objects' configurations in each step according to their trajectories, and checks collisions. The iteration's time step is the configurable value that has been used before (see COLLISION_CHECK_TIME_STEP in 3.3). The simulation time (until the iterations go) is:

$$N = \frac{3 \cdot \frac{v}{a_{max}}}{T_{step}}$$

v	The speed of the current actuation
a_{max}	The maximum deceleration
T_{step}	The collision check simulation time step
N	The number of needed steps

Given that:

$$v <= v_{max}$$

v_{max}	The maximum speed

the following statement is always true:

$$N <= \frac{3 \cdot \frac{v_{max}}{a_{max}}}{T_{step}}$$

which was the number trajectory simulation time steps for the surrounding obstacles (see 3.6.2). Therefore, it is provided that all objects in the environment have a pre-calculated trajectory for the whole collision check simulation interval.

3.7 Velocity obstacle map evaluation

Let me summarize briefly what the algorithm has done so far.
In each iterations, the algorithm:

1. found a new destination point and updated the target actuation
2. updated the dynamic window of actuations
3. received a static map and an array of dynamic obstacles from the mapping node
4. filtered both in order to keep only the objects relevant for local motion planning
5. calculated trajectories for all dynamic obstacles
6. checked all available actuations for collisions with any of the static or dynamic objects
7. built a velocity obstacle map using these collision times

Now the next (and final) step is to evaluate the velocity obstacle map, and find the best actuation. But defining what 'the best' actuation is is not trivial. Let's look at the scenario represented in figure 3.13.

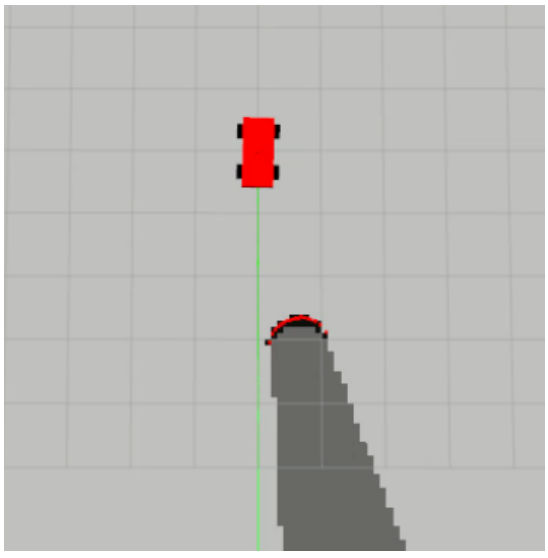


Figure 3.13: What is 'the best' actuation?

The target trajectory is a straight line (marked with the green line in the rviz visualization, published by the motion planning node on topic *global_trajectory*). But there is an obstacle in the way. Without the obstacle being in the picture, one could state without hesitation, the the target actuation is always the one the keeps the car on the line. But once the object is there, this is no longer the case. Defining the best actuation to be the one that

keeps car in direction would lead to collision, obviously, the car needs to steer to the right to avoid that. But this raises even more questions. *Should the car steer to the right long before the object? Is it enough to turn the wheels in the last moment? Are we sure we are able to steer away from collision, isn't an emergency brake a better solution?* I answered these questions and solved the problem of finding the best actuation with the introduction of fitness factors.

3.7.1 Fitness factors

A fitness factor is a number in the [0.0 1.0] interval describing how good (how fit) the given actuation is according to an expectation or requirement. I introduced three fitness factors in the algorithm.

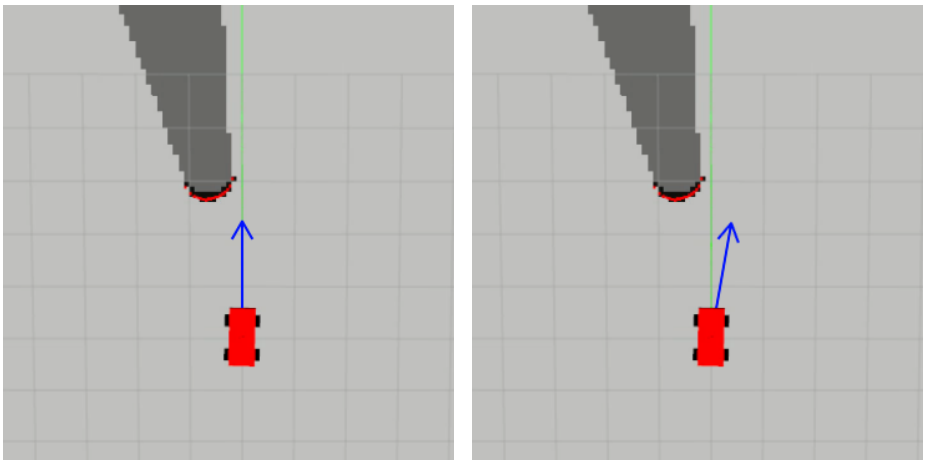
Speed factor
Describes how near the actuation's speed is to the target speed.
1.0: The actuation's speed is exactly the same as the target speed.
0.5: The actuation's speed is halfway between the target speed and the worst possible candidate.
0.0: The actuation's speed is the maximum permitted speed to the opposite direction (worst possible candidate).

Direction factor
Describes how near the actuation's wheel angle is to the target wheel angle.
1.0: The actuation's wheel angle is exactly the same as the target wheel angle.
0.5: The actuation's wheel angle is halfway between the target wheel angle and the worst possible candidate.
0.0: The actuation's wheel angle is the maximum permitted wheel angle to the opposite direction (worst possible candidate).

Safety factor
Describes how safe the actuation is, meaning how improbable a collision is.
1.0: Using the actuation, a collision within time interval will not happen.
0.5: The actuation leads to a collision, but the obstacle is farther than the car's brake distance.
0.0: The actuation leads to an unavoidable collision.

Let me explain these factors with two simple examples, regarding the same scenario (shown above in figure figure 3.13). The following images show two possible actuations in a situation (both are within the dynamic window). And based on their speed and wheel angles, the car's current configuration and the obstacle's position, the fitness factors can be calculated for both actuations.

Figure 3.14 shows the two example actuations. As a precondition, the target speed and wheel angle must be defined. The target speed is always defined by the maximum allowed speed (see MAX_SPEED_FWD and MAX_SPEED_BWD in 3.3) and the target wheel angle. For this is only an example, let me assume that the car is in perfect orientation and arbitrarily set the target speed to MAX_SPEED_FWD, and the target wheel angle to zero. Let us assume that we have already calculated the collision times for each possible actuation.



(a) Straight actuation **(b)** Curved actuation

Figure 3.14: Possible actuations

During factor calculations, the following acronyms are used:

MAX_SPEED_FWD	The maximum permitted speed
MAX_WHEEL_ANGLE	The maximum permitted wheel angle
T_{check}	The collision time check interval
v_{target}	The target speed
α_{target}	The target wheel angle
v_i	The speed of actuation i
α_i	The wheel angle of actuation i
$t_{coll,i}$	The collision time using actuation i
$F_{speed,i}$	The speed factor for actuation i
$F_{dir,i}$	The direction factor for actuation i
$F_{safety,i}$	The safety factor for actuation i

$$MAX\_SPEED\_FWD = 1.5\frac{m}{s}$$

$$MAX\_WHEEL\_ANGLE = 20deg$$

$$T_{check} = 3s$$

$$v_{target} = MAX\_SPEED\_FWD$$

$$\alpha_{target} = 0deg$$

Factor's for actuation i are calculated using the following equations:

$$F_{speed,i} = 1 - \frac{abs(v_i - v_{target})}{MAX\_SPEED\_FWD - MAX\_SPEED\_BWD}$$

$$F_{dir,i} = 1 - \frac{abs(\alpha_i - \alpha_{target})}{2 \cdot MAX\_WHEEL\_ANGLE}$$

$$F_{safety,i} = \frac{t_{coll,i}}{T_{check}}$$

The factor calculations for the two examples are the following:
Straight actuation (figure 3.14a)

$$v_1 = 1.5\frac{m}{s}$$

$$\alpha_1 = 0deg$$

$$t_{coll,1} = 1.5s$$

The fitness factors are the following (calculated by the equations listed above):

$$F_{speed,1} = 1.00$$

$$F_{dir,1} = 1.00$$

$$F_{safety,1} = 0.50$$

As it is clearly visible on the picture, and also from the results, the speed and direction factors are perfect, but the safety factor is low, because the leads to a collision in the near future. Now let us examine the other actuation.
Curved actuation (figure 3.14b)

$$v_2 = 1.5\frac{m}{s}$$

$$\alpha_2 = -10deg$$

$$t_{coll,2} = 4.5s$$

Note that the collision time of the actuation is always is the [0 T_{check}] interval. The result of the collision time check method is T_{check} when the actuation does not cause a collision. The fitness factors are the following (calculated by the equations listed above):

$$F_{speed,2} = 1.00$$

$$F_{dir,2} = 0.75$$

$$F_{safety,2} = 1.00$$

Now that the speed, direction and safety factors have been calculated for all possible actuations, the only sub-task left is to find the best actuation.

3.7.2 Best actuation

The best possible actuation is always the one with the highest fitness factors. The factors of an actuation are summarized to one number (which will then become *the* fitness of the actuation) the following way:

$$F_i = \frac{F_{speed,i} \cdot w_{speed} + F_{dir,i} \cdot w_{dir} + F_{safety,i} \cdot w_{safety}}{w_{speed} + w_{dir} + w_{safety}}$$

F_i	The fitness of actuation i
$F_{speed,i}$	The speed factor for actuation i
$F_{dir,i}$	The direction factor for actuation i
$F_{safety,i}$	The safety factor for actuation i
w_{speed}	The weight of the speed factor (see W_SPEED_FACTOR in 3.3)
w_{dir}	The weight of the direction factor (see W_DIRECTION_FACTOR in 3.3)
w_{safety}	The weight of the safety factor (see W_SAFETY_FACTOR in 3.3)

In the implementation, the weights are set as follows. (Further tuning is possible and advised for more production-ready implementations.)

$$W\_SPEED\_FACTOR = 2.00$$

$$W\_DIRECTION\_FACTOR = 1.00$$

$$W\_SAFETY\_FACTOR = 4.00$$

As an example, let us continue the previous example with the straight and the curved actuations.

Straight actuation (figure 3.14a)

$$F_{speed,1} = 1.00$$

$$F_{dir,1} = 1.00$$

$$F_{safety,1} = 0.50$$

An the final fitness is:

$$F_1 = \frac{1.00 \cdot 2.00 + 1.00 \cdot 1.00 + 0.50 \cdot 4.00}{2.00 + 1.00 + 4.00} = 0.71$$

Curved actuation (figure 3.14b)

$$F_{speed,2} = 1.00$$

$$F_{dir,2} = 0.75$$

$$F_{safety,2} = 1.00$$

An the final fitness is:

$$F_2 = \frac{1.00 \cdot 2.00 + 0.75 \cdot 1.00 + 1.00 \cdot 4.00}{2.00 + 1.00 + 4.00} = 0.96$$

The results show that out these two actuations, the second one (the curve) would have been selected by the algorithm. This is mainly because safety has a much higher weight than direction when summarizing factors. Obviously, this has not been a complete actuation search because the example consisted of only two actuations, while the the dynamic window can consist of tens or even hundreds of possible alternatives. As I mentioned earlier, the width and the height of the dynamic window depends on the car's and the environments characteristics, while its resolution can be set in the parameters (see DYNAMIC_WINDOW_ANGLE_RESOLUTION and DYNAMIC_WINDOW_SPEED_RESOLUTION in 3.3).

3.7.3 Publishing Ackermann driving control

The hard part of the node's work is done, the best possible actuation has been found. The very last task is to publish this desired speed and wheel angle pair to the other ROS nodes that control the car.

This publication happens on topic */vrcar/manual_control*, using ackermann_msgs/AckermannDrive messages.

The actuator nodes will use these messages to propagate the requested values to the DC motor and the steering servo, and thus, the control pipe has reached its end.

Chapter 4

Test results

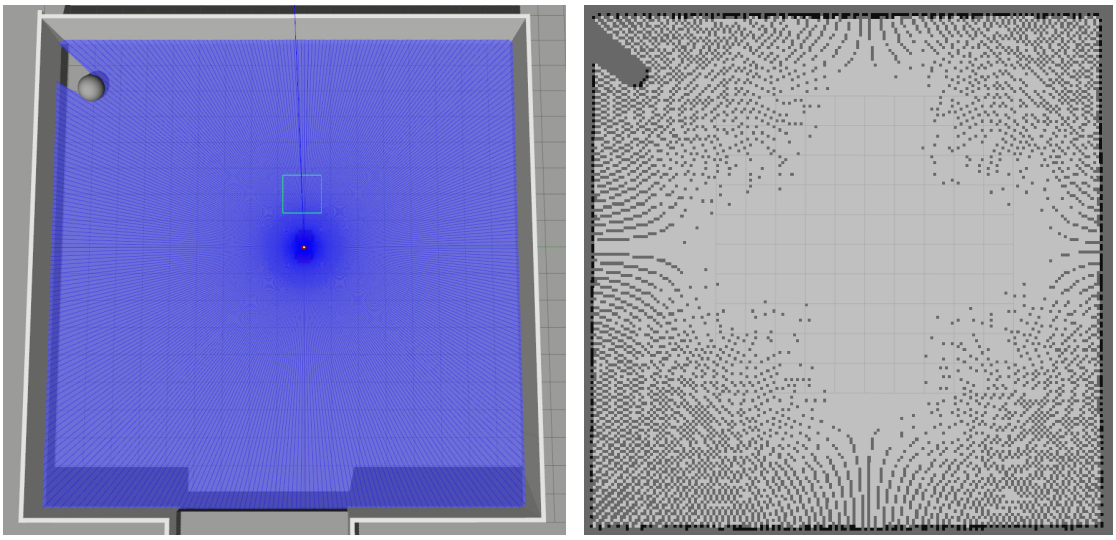
Now that the algorithms have been written about in detail, let me explain how I tested the nodes both in simulator and in real life.

4.1 Mapping results

The aim of the mapping node was to find the moving objects in the environment, based on LIDAR scans, and to separate them from the static parts of the map. The algorithm converted the LIDAR measurements into absolute map points, than created a differential grid - basically a map, with the high values not marking objects like a regular map, but marking movements. The algorithm found the possible dynamic point candidates based on this grid. After the candidates have been selected, the algorithm created groups among the measured points, based on the positions. Relying on these groups being valid, and the assumption that points in the same group must be still or moving in the same direction, speed vectors were calculated for the groups. The groups with speed larger than a minimum value (which may be caused by measurement noise) became the moving objects, and the points inside these groups were marked dynamic. Every other measurement was treated non-moving, and therefore placed in the static map. The static map building is as straightforward as it can be - every static measurement creates a free ray in the map, starting from the position of the LIDAR and ending at the measured point.

The results of the mapping and the separation of static and dynamic points are presented by an example with a valid use-case. The scenario is the following. The car is standing in a closed room, the distance from the walls are smaller than the LIDAR's range. There is one other object in the room, in the simulation, this is a ball, its size similar to the car's. The ball is standing in one place at first, but after a while, it start moving in a straight path with constant speed. Finally, the car starts moving as well - first in a straight path, then turning.

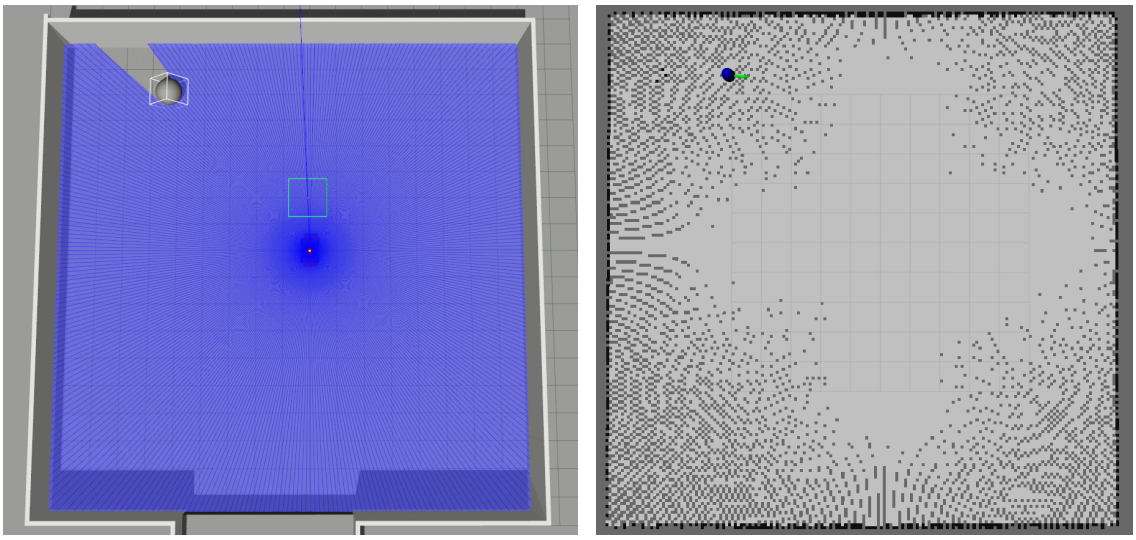
4.1 presents the initial situation of the demonstration. The left image shows the simulation - the car is standing in the middle of the room, the object in the top left corner is not moving yet. On the right image, the created static map is visible. The number of unknown (grey) grid points is large at this state of the simulation, it will dramatically decrease once the car starts moving. This effect is caused by the LIDAR's finite angular resolution and ray characteristics. The black grid points represent statically occupied positions. All the walls are marked static, and the ball as well.



(a) Gazebo simulation **(b)** rviz visualization

Figure 4.1: Initial state - ball is still

Once the ball starts moving (4.2), it stops being marked as static, but is handled as a moving object. The static block at the top left corner has been removed, and the ball is displayed using the dynamic visualization toolset - a sphere and an arrow, representing the ball's area and speed vector.

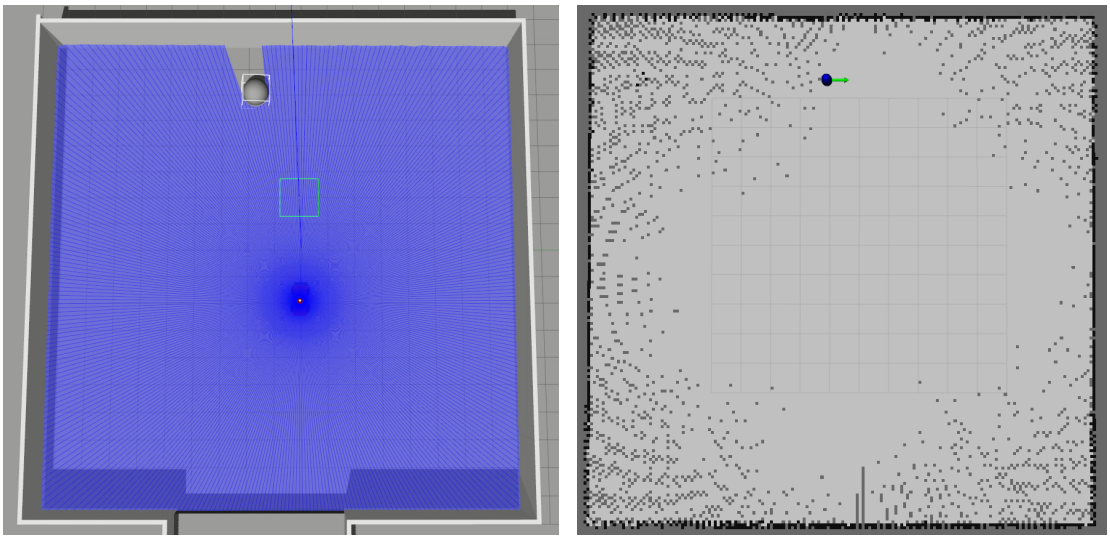


(a) Gazebo simulation **(b)** rviz visualization

Figure 4.2: The ball is moving

When the car starts moving, the map immediately starts to clear out. If we compare 4.2 and 4.3, it is clear that the latter contains less unknown grid points.

Due to the map implementation being very simple, and updating each map point without handling its history, the mapping node can only be used with certain limitations regarding obstacle speed and the car's angular velocity. But the main task for the mapping algorithm was not to create a static map with the best possible quality, but to separate the static points from the dynamic objects.



(a) Gazebo simulation **(b)** rviz visualization

Figure 4.3: The car is moving

4.2 Motion planning results

The task of the motion planning node was to calculate and publish actuations based on the static and dynamic maps provided by the mapping node, that lead the car towards the destination, following the required trajectory, without any collisions. The algorithm first filters the static and dynamic input maps to decrease the number of environment points. Then it calculates trajectories for the dynamic obstacles. Independently from the maps, it sets the target actuation according to the target trajectory or an external control command, and also updates the dynamic window, based on the current actuation. When all these steps are finished, it calculates the collision times for all the actuations in the dynamic window and all objects in the car's environment, and builds the velocity obstacle map. Using this map and the target actuation, fitness factors are calculated for all available actuations. Summarizing and evaluating these factors, the best possible actuation is determined. This actuation is then published to the actuator nodes.

As the motion planning examples also demonstrate the map building implicitly, I recorded several measurements of different kinds of scenarios, and I selected a set of them that represent the algorithms advantages and deficiencies. Most of the scenarios I tested in simulation.

As a 0th step, I tested if the algorithm follows a straight, then a curved trajectory with no obstacles in the way. These tests were needed to check if actuation handling and the target position update mechanism work as expected. After these features were tested successfully, the testing procedure could move forward to moving objects in the trajectories.

The first example is still a quite simple one. The car is given a straight trajectory, but there is a static obstacle in its way. The expected behaviour is to bypass the object while staying as close to the trajectory as possible, and when the car has moved past the obstacle, it should keep to the trajectory again.

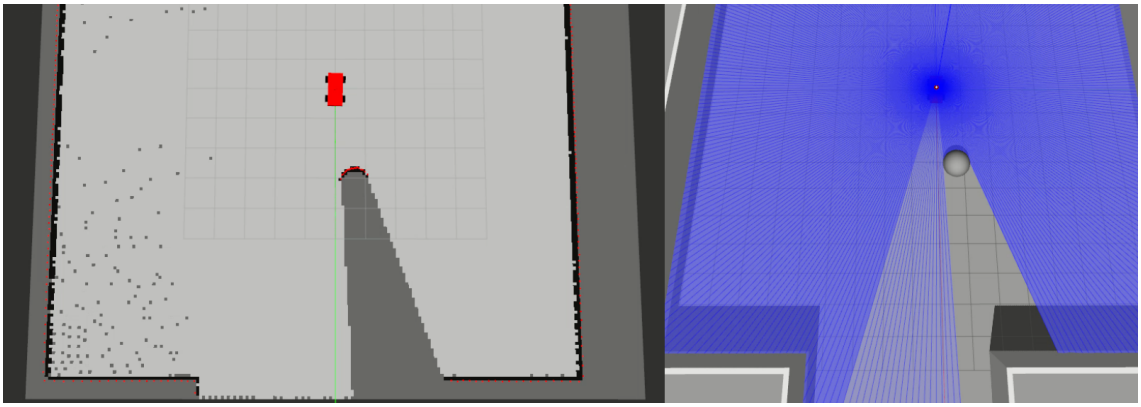
The simulation images always display the Gazebo simulation on the right and the rviz visualization on the left. Let me briefly summarize how the objects are displayed in rviz. The car has red colour, the target trajectory is marked by the green line. The LIDAR

scan consists of the red dots and the map points are either white (free), grey (unknown) or black (occupied). Therefore, static objects appear as black masses in the map. Each dynamic object is marked with a blue circle representing its position and size, and a green arrow for its speed vector.

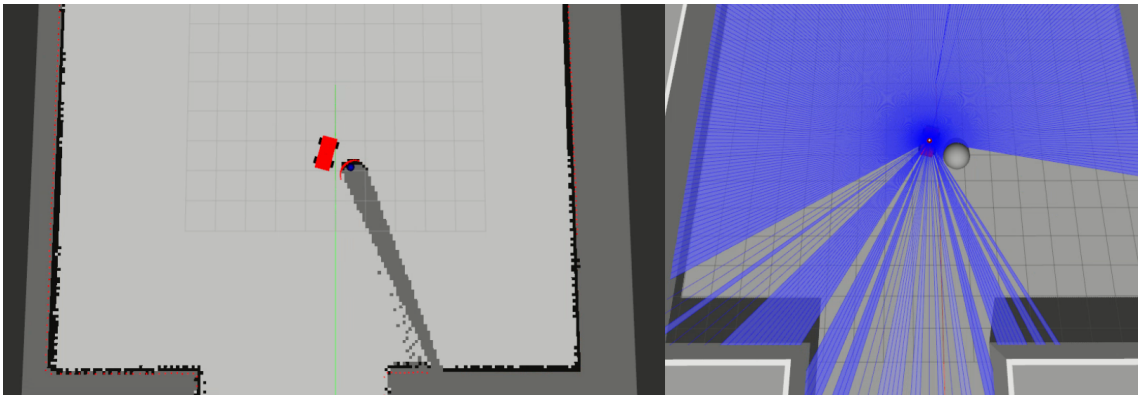
As the images **??**-**??** show, the car successfully avoids the object and then keeps to the trajectory again.

The second example is similar to the first one, but has one major difference: the object to avoid is not static but is moving opposite the car, with a speed nearly equal to the car's.

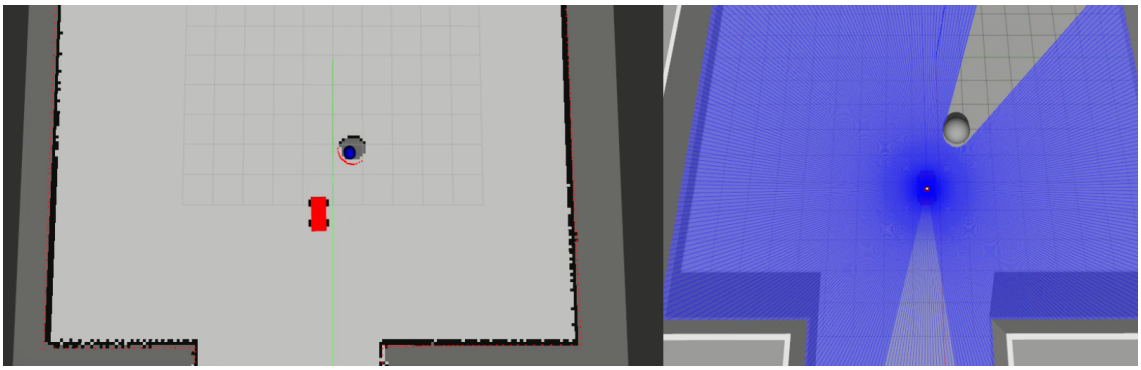
Figures **??**-**??** represent the scenario. Comparing images **??** and **??**, it clearly shows that in the latter case (when the object is moving opposite the car), the car starts the obstacle avoidance movement a lot earlier. That is because the object is successfully detected as dynamic (marked by a blue circle and a green speed vector in the rviz visualization).



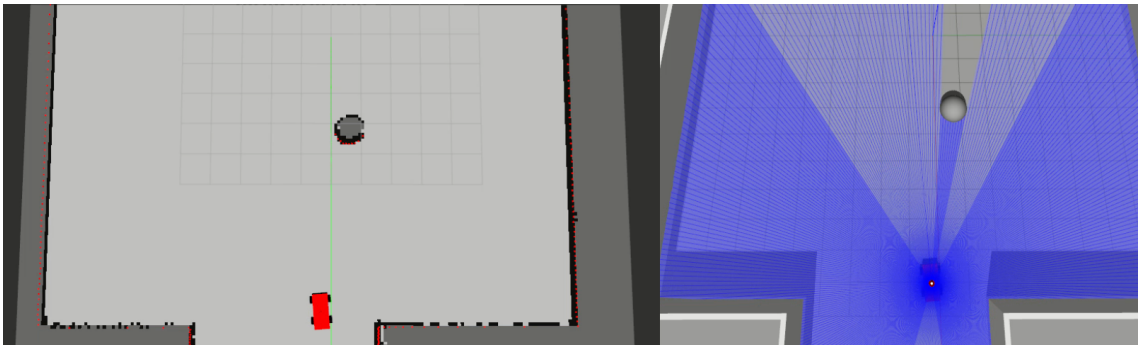
(a)



(b)

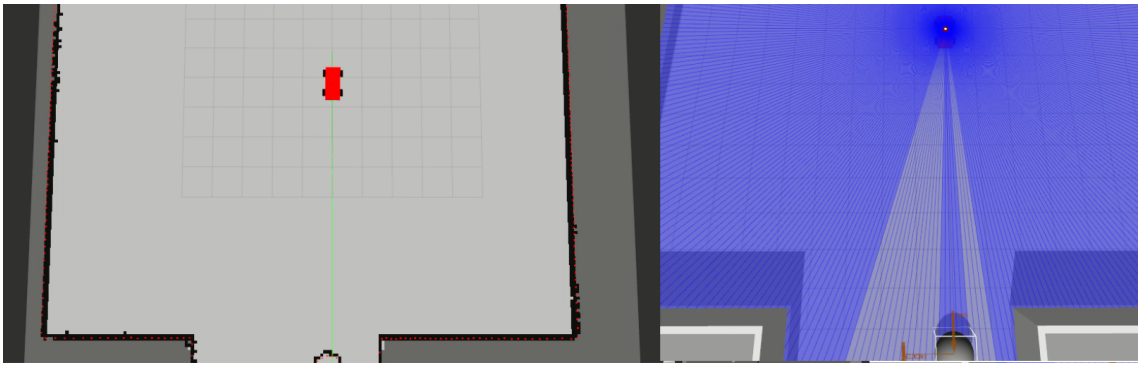


(c)

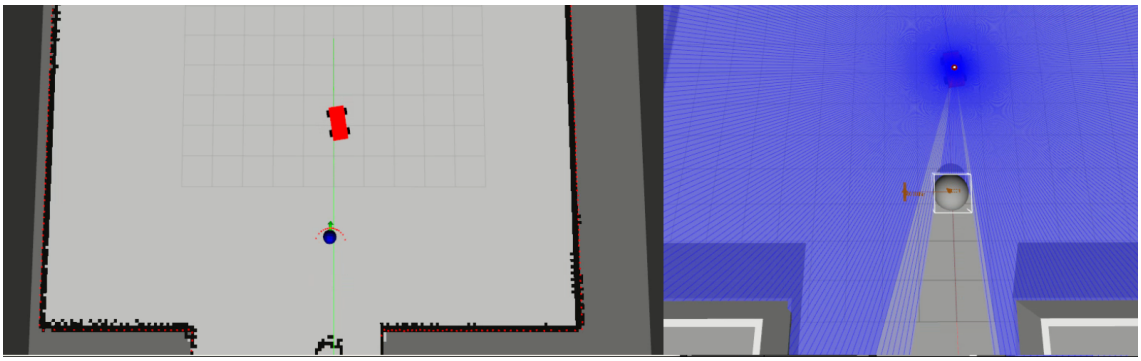


(d)

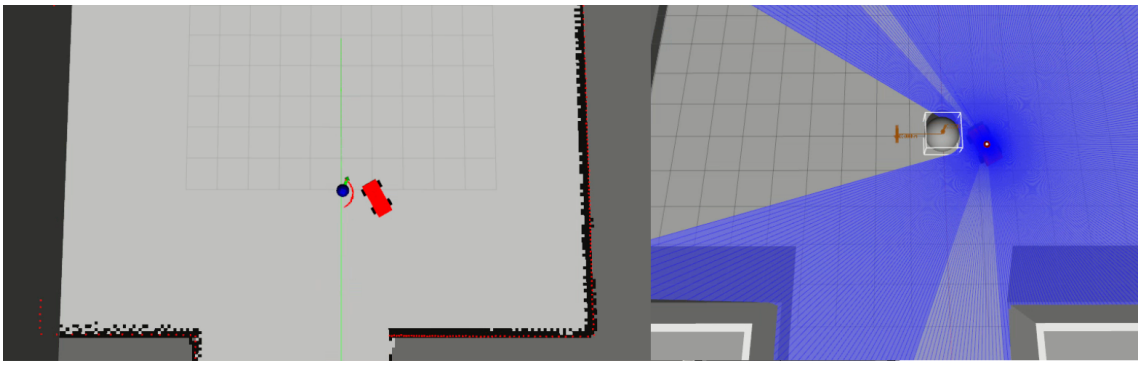
Figure 4.4: Straight trajectory, static obstacle



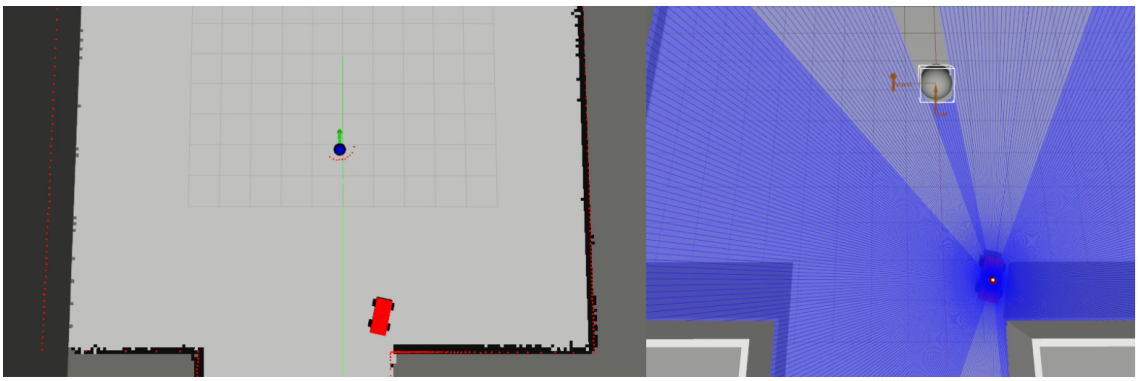
(a)



(b)

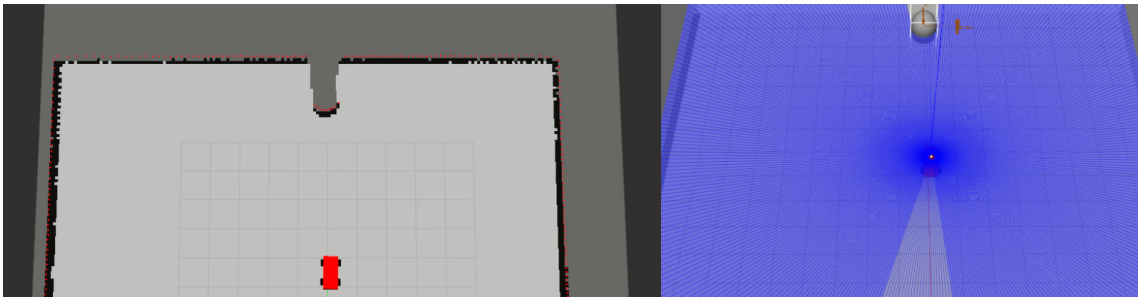


(c)

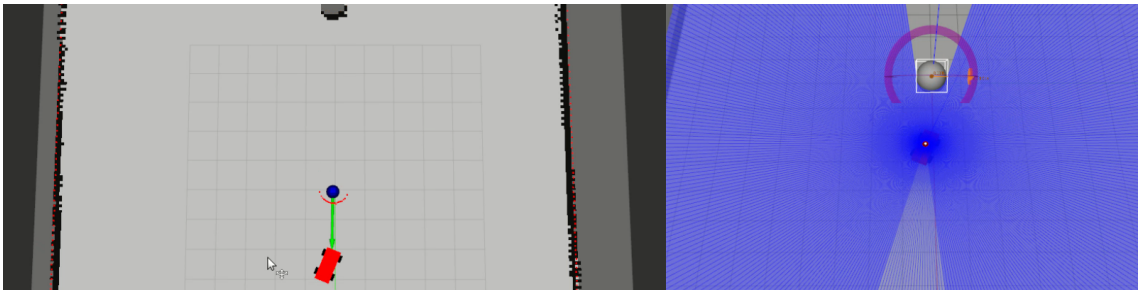


(d)

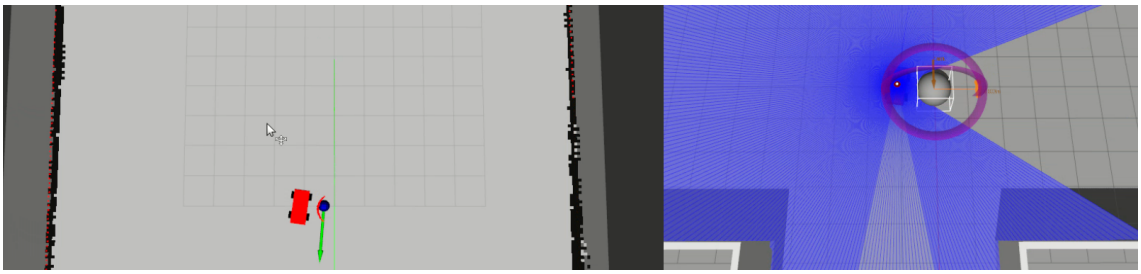
Figure 4.5: Straight trajectory, dynamic obstacle



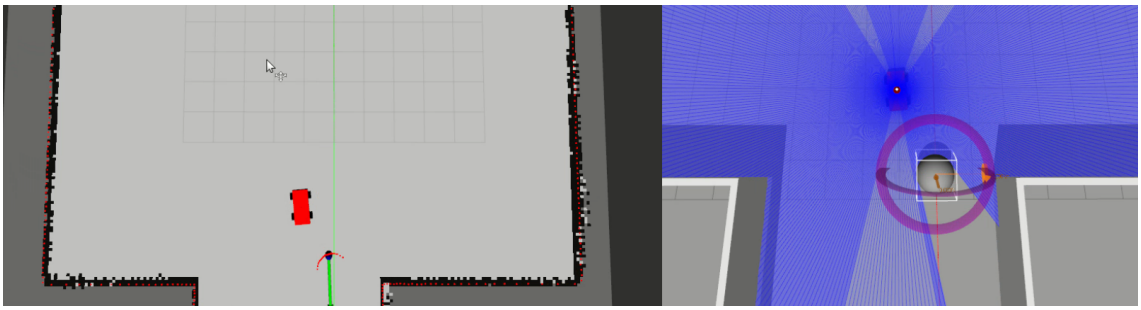
(a)



(b)

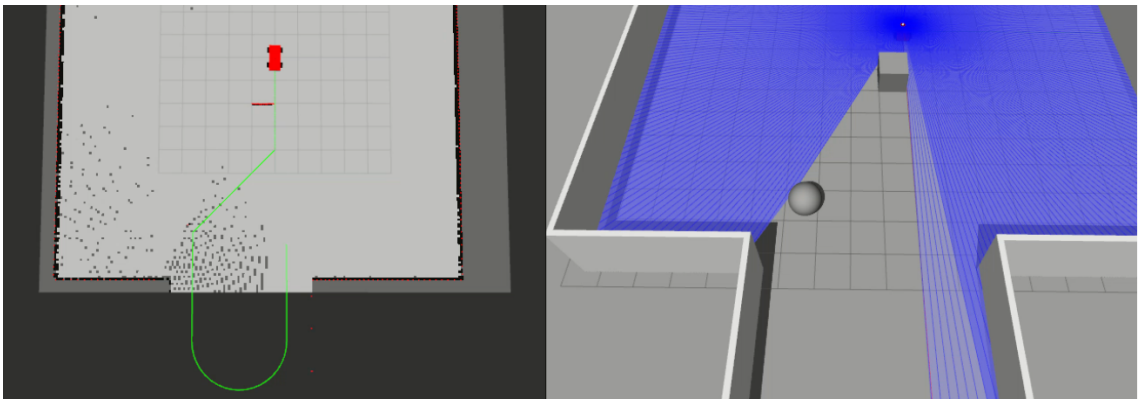


(c)

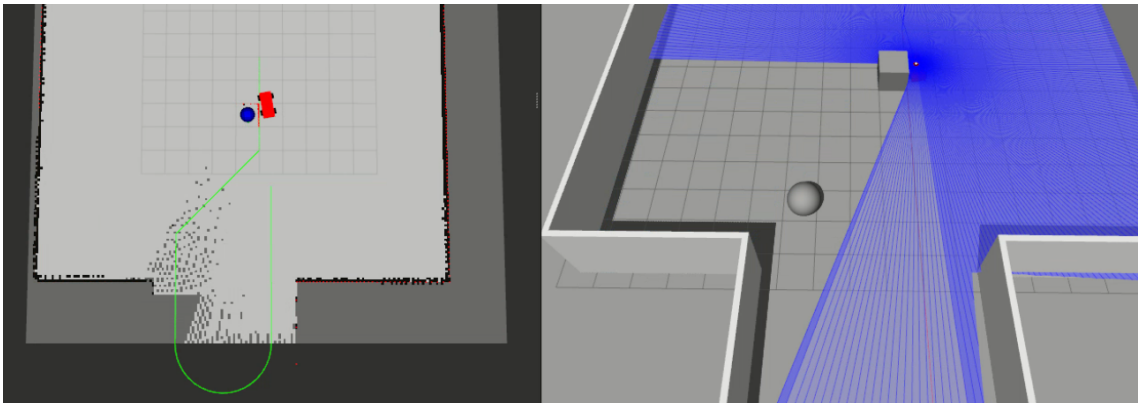


(d)

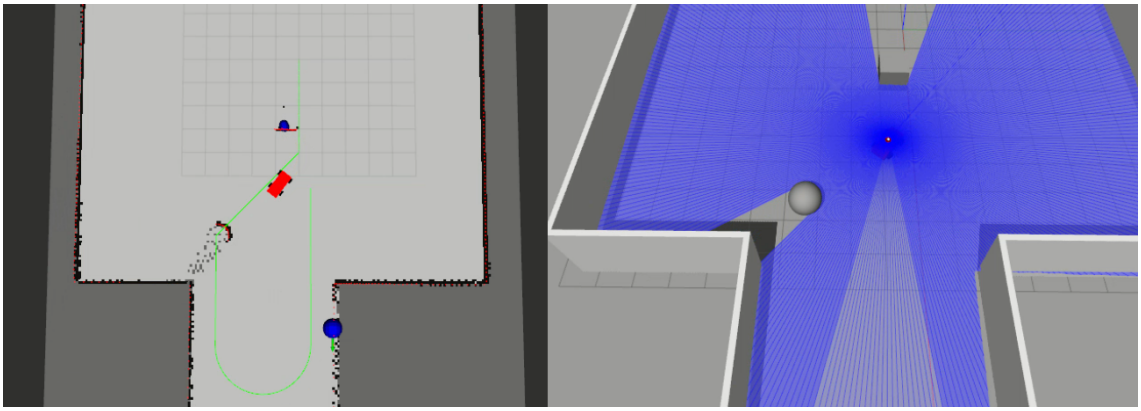
Figure 4.6: Straight trajectory, faster obstacle from behind



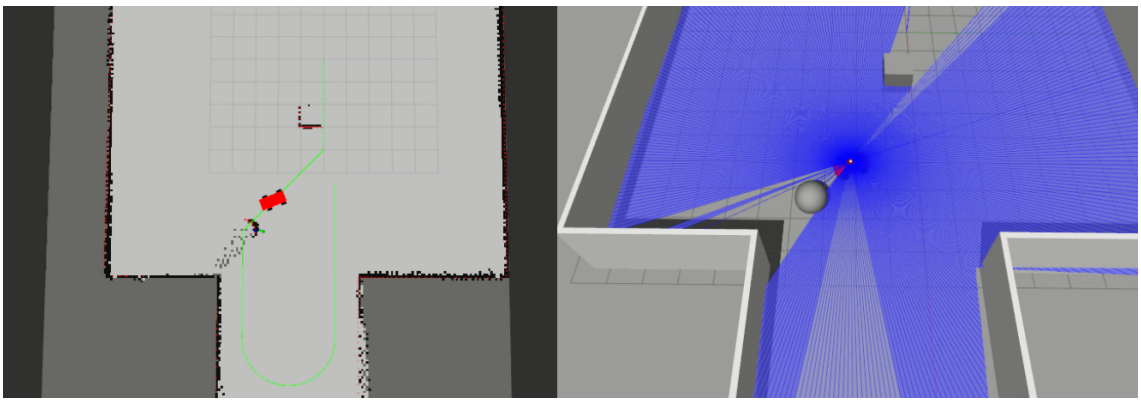
(a)



(b)



(c)



(d)

Figure 4.7: Curved trajectory, 2 static obstacles

Chapter 5

Conclusion

As a conclusion I am going to evaluate whether the mapping and the motion planning node have achieved the expected results. The main task of the project was to create two ROS nodes that can be integrated into an existing system. The tasks of these two nodes were to separate the static and dynamic objects in the map, and calculate actuations that avoid these obstacles.

TODO

5.1 Interesting findings and morals

During the implementation of the mapping and planning algorithms I had to face a few problems that I thought was worth mentioning, either because they were surprisingly hard to overcome or just because they were unexpected or interesting.

5.1.1 Symmetrical map

The first one is rather a tip than a finding - a tip for myself in the future and anyone who is planning on writing a mapping program. *Don't use a symmetrical map, with the car starting in its center!* Build an asymmetrical map or place the car somewhere but the center. It took me at least 3 hours of debugging to find where one of the absolute point angle calculations was wrong, that cause the map to rotate by 180 degrees. But for quite a long time, this bug didn't event come to my attention, because the map was symmetrical, and therefore was insensitive to a 180 degree rotation.

5.1.2 The closing ball

The second finding is an unexpected mapping problem that I didn't event notice until it ruined the local trajectory planning. It happened when a round object[1] is moving in a straight line, and passes near the car. Because of its round shape, only a half of the object is seen by the LIDAR at every time point. But the half that is visible is changing as the object is passing the car (kind of like the Sun making different parts of the Moon visible, as it's rotating relative to the Earth). But the speed vector calculation is based on the movement of the objects' mass center, which is calculated by the visible points. Therefore,

[1] In the simulator I used balls as obstacles, because applying a still force to them made them move with a linear speed.

as the visible points select a different half of the object, the mass center changes, and alters the object's speed vector.

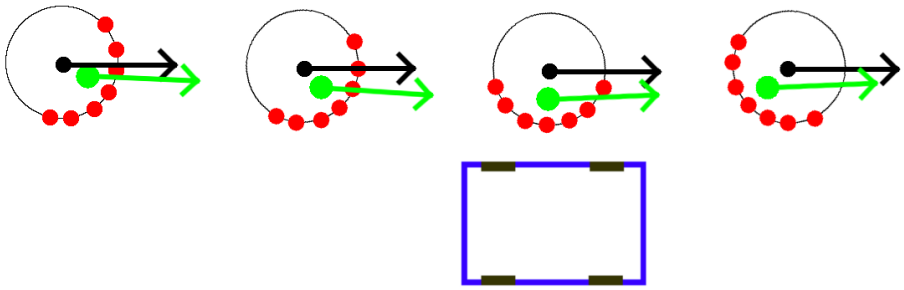


Figure 5.1: The closing ball

In figure **??** this effect can be viewed. The ball is moving in a straight line, its speed vectors in every step are marked with the black arrows. The visible points (marked with red dots) are shifting to the side of the ball as it passes the car. As a result, the calculated speed vectors (green arrows) tend to 'bend' towards the car when the obstacles is getting closer.

The effect of the speed vectors changing does not seem relevant, but during motion planning, the ball is seen as an object that is moving forward, but suddenly, it starts to approach the car by turning more and more towards it, making all actuations get marked as unsafe.

The problems has not been solved entirely, but its effect has been reduced by changing the mass center calculation method in the mapping node.

Bibliography

[1] Fethi Belkhouche. Reactive path planning in a dynamic environment. (1):1–10, 6 2009. DOI: 10.1109/TRO.2009.2022441.

[2] Animesh Chakravarthy and Debasish Ghose. Obstacle avoidance in a dynamic environment: A collision cone approach. (1):1–13, 9 1998. DOI: 10.1109/3468.709600.

[3] Baifan Chen, Zixing Cai, Zheng Xiao, Jinxia Yu, and Limei Liu. Real-time detection of dynamic obstacle using laser radar. (1):1–5, 11 2008. DOI: 10.1109/ICYCS.2008.357.

[4] Bruno Damas and José Santos-Victor. Avoiding moving obstacles: the forbidden velocity map. (1):1–6, 10 2009. DOI: 10.1109/IROS.2009.5354210.

[5] Paolo Fiorini and Zvi Shiller. Motion planning in dynamic environments using velocity obstacles. (1):1–13, 7 1998. DOI: 10.1177/027836499801700706.

[6] M. G. Mohanan and Ambuja Salgoankar. A survey of robotic motion planning in dynamic environments. (1):1–15, 10 2017. DOI: 10.1016/j.robot.2017.10.011.