Entwicklung von Navigationssoftware für mobile Robotersysteme und Simulation

by

Tristan Schwörer

Matriculation number: 71336

A bachelor thesis submitted in partial fulfillment of the requirements for the degree of the

Bachelor of Engineering (B. Eng.)
in Mechatronics
at Aalen University

Supervisor:
Prof. Dr. Stefan Hörmann (Aalen University)

Submitted on:
April 13th, 2021

Preface

This bachelor thesis is part of the bachelor program in mechatronics at Aalen University and is supposed to take place in the 7$^{\text{th}}$ Semester. It covers the theoretical and practical work between November 2$^{\text{nd}}$ 2020 and April 13$^{\text{th}}$ 2021.

This work took place at the laboratory for mobile roboic systems of the faculty optics and mechatronics at aalen university and was completely independent of any other party and company.

Diese Bachelorarbeit ist fester Bestandteil des Bachelorprogramms Mechatronik an der Hochschule Aalen und soll im siebten Semester statt finden. Die hier dokumentierte Arbeit wurde zwischen dem 2. November 2020 und dem 13. April 2021 realisiert.

Die praktische sowie die theoretische Arbeit fand im Labor für mobile Robotersysteme der Fakultät Optik und Mechatronik an der Hochschule Aalen statt und ist komplett unabhängig von jeglicher anderen dritten Partei oder Firma.

Abstract

This bachelor thesis is about the concept, setup/development and testing of a software stack used for the autonomous navigation in an environment defined by the rules of the carolo cup.

The aim of this stack is lane following and obstacle avoidance based on a sensor data of the environment. This thesis extends the work of Prof. Hörmann who provided the road detection and is supposed to be used by the carolo cup team of university aalen in the future. Even though the robot used in this thesis does not satisfy the rules of the carolo cup the stack should be configurable for different robots aswell.

The robot is equipped with a lidar, a camera, wheel encoders and an imu. The data of these sensors will be filtered and processed using existent ros packages as well as newly developed ones. The resulting data will be fed into the navigation stack that then determines the best route for the robot.

Since there wasn't a driving robot available at the start of this thesis the task of simulating the robot with all of its sensor data and actors has been incorporated into the subject of this work.

Kurzfassung

Diese Bachelor-Thesis handelt von der Erstellung eines Konzepts, dem Aufbau und der Entwicklung eines "Software-Stack" und dessen Testens für die autonome navigation in einer, durch das Regelwerk des carolo cups beschriebenen, Umgebung.

Das Ziel dieses "Software-Stacks" ist, der Spur einer Straße zu folgen und dabei potentiellen Hindernissen auf der Straße auszuweichen. Diese Thesis führt die Arbeit von Prof. Hörmann der die von ihm Entwickelte Spurerkennung zur Verfügung stellte und soll in der Zukunft vom Carolo-Cup Team der Hochschule Aalen verwendet werden können. Obwohl der in dieser Arbeit verwendete Roboter nicht konform zum Regelwerk des Carolo-Cups ist, soll der Stack auch für andere Roboter konfigurierbar sein.

Der Roboter verfügt über einen Lidar, eine Kamera, Rad-Encoder und einen IMU (inertia measurement unit). Die Daten dieser Sensoren werden gefiltert und dann mit bestehenden ros packages und selbst entwickelten aufbereitet. Die resultierenden Daten werden dann an den Navigation Stack übergeben, der dann die beste Rute ermittelt.

Da zu Beginn dieser Arbeit kein vollständig funktionierender Roboter verfügbar war wurde das Teilthema der Simulation des Roboters mitsamt aller seiner Sensoren und Aktoren in das Thema der Thesis aufgenommen.

Acknowledgement

At this point I would like to thank the following people for supporting me during my bachelor thesis:

- **Prof. Dr. Stefan Hörmann** for being my supervisor during this time and for allways helping me with new ideas and approaches.
- **Prof. Dr. Arif Kazi** for being the second supervisor and helping me with ideas regarding the structure of the thesis.

Extending to that i would like to thank my Family and Friends that supported me during this period.

Table of Contents

1. Theoretical Background

This chapter will cover the needed theoretical background about the Gazebo Simulation, the Sensor Plugins, ROS and all of the used ROS packages.

1.1. ROS

ROS (Robot Operating System) is an open Source project developed by the "Open Source Robotics Foundation". Like the name suggests it is an entire Operating System for Robots including Hardware abstraction, low-level device control, implementation of commonly used functionality, communication between processes and package management.
Furthermore it provides tools and libraries to write, build and run code across multiple computers[1].

1.1.1. Packages

This is the main structure for software in ROS. A package can contain many different Nodes, libraries, service etc.. Furthermore it is the smallest possible Structure that can be build by ROS[2].

1.1.2. Nodes

Nodes are processes that perform computation. Since ROS is very fine granular, a system, that controls an entire robot can contain many nodes that are connected using topics. A package can be written with the use of one of the client libraries roscpp or rospy[2].

1.1.3. Plugins

1.1.4. Topics and service

All of the ROS Nodes are connected with a publisher/subscriber like structure. The topic is basically just a name for a certain message.

Not only one node but unlimited many nodes can publish and subscribe to one topic. This generally can be seen like a message bus with not limited connection permissions[2]. Unfortunately the Topic system is not well fitted for request and answers between two nodes, therefore the service structure has been implemented.

A node might offers a service under a certain node and an other node can call that service. Services can have any in- and output that can be specified in a ".srv" file[2].

1.1.5. RVIZ

rviz is a 3D visualization tool offered by default in ROS. It offers functionality to visualize sensor and further geometric data.

1.1.6. REP

REP's (short for ROS enhanced proposals) are guidelines made and maintained by the ros community. It is highly advisable to follow the guidelines as much as possible.
Complying to these guidelines allows external people easier comprehension of the structure of the robot and eliminates misunderstandings.
The most important REP's in this project are REP 103 and REP 105.

REP 103

"This REP provides a reference for the units and coordinate conventions used within ROS"[3]

Coordinate Frame

- **X-Axis** - Forward
- **Y-Axis** - Left
- **Z-Axis** - Up

Units
Units will always be represented in SI Units and their derived units.

The order of preference for rotations

1. Quaternion
2. Rotation matrix
3. fixed axis roll, pitch, yaw
4. Euler angles

[3]

REP 105

"This REP specifies naming conventions and semantic meaning for coordinate frames of mobile platforms used with ROS."[4]

REP103 Applies for all fixed coordinate frames.
Coordinate Frames

- **base_link** is a fixed frame on the robot base. It serves as the reference points for all of hardware mounted on the robot itself like sensors.
- **odom** is a world fixed frame that serves as the reference for the pose of the robot. Since the pose of the robot will drift over time it wont serve as a good long term reference.
  In most cases the odom frame will be computed using localization sensors like wheel odometry, imu's, visual odometry, etc. which leads to a continuous frame.
- **map** is a world fixed coordinate frame that serves as the reference for the odometry frame. It is also the base for a map of the environment such as the ones provided by slam algorithms. The frame is time discrete since it is mostly computed by localization algorithms.

That tree can be extended by an earth frame that would be the reference for the localization of the map in the earth. Which is useful, for long range robot platforms.[4]

1.1.7. TF

In most cases robots that are controlled by ros have a so called tf_tree. This tree is the coordinate frame structure of the robot. In it every sensor and actor has its own coordinate frame.
The structure in most trees of mobile platforms is quite similar which is caused by the REP105 (ROS Enhanced Proposals) this contains a definition of recommended names for the robot frames and their order in the tree. But it should be noted that not every frame that is defined in the norm has to be in every tree. The basic structure mostly starts at a so called fixed frame. This Frame will be the not changing frame in the environment. At moving robots this is often earth, map or odom, while in stationary robots this can even be base_link.

The tree is normally build up like in the following image.
TF2 is the successor of TF and is a very powerful tool in the ROS environment. With it it is possible to transform sensor_msgs and geometry_msgs from one frame in another. Furthermore it offers the possibility to transform old data into the present or at any other point in the past.

URDF and xacro

The robot hardware description consists of one or more URDF(Unified Robot Description Format) based xml file. Its purpose is to define the shape and geometric of every part of the robot.

robot_state_publisher

This package uses the robot hardware description and builds up the tf_tree using static_transform_publishers.

1.2. Gazebo

1.2.1. Plugins

Gazebo offers a wide selection of pre made plugins that can be incorporated into a simulated robot by attaching the plugin to the right tf_frame and configuring its parameters.

1.2.2. Models

1.3. navigation stack

1.3.1. move_base

1.3.2. global_planner

base_global_planner

This is the default global planner of move_base .

1.3.3. local_planner

teb_local_planner

dwa_local_planner

1.3.4. costmap

A costmap is a grid stile map, whose purpose is to store information about obstacles in the surrounding of the robot.
There are two different costmaps, the global costmap and the local costmap. The global costmap is by the global planner to find a collision free path. Whereas the local costmap is used by the local planner for local planning.[5]

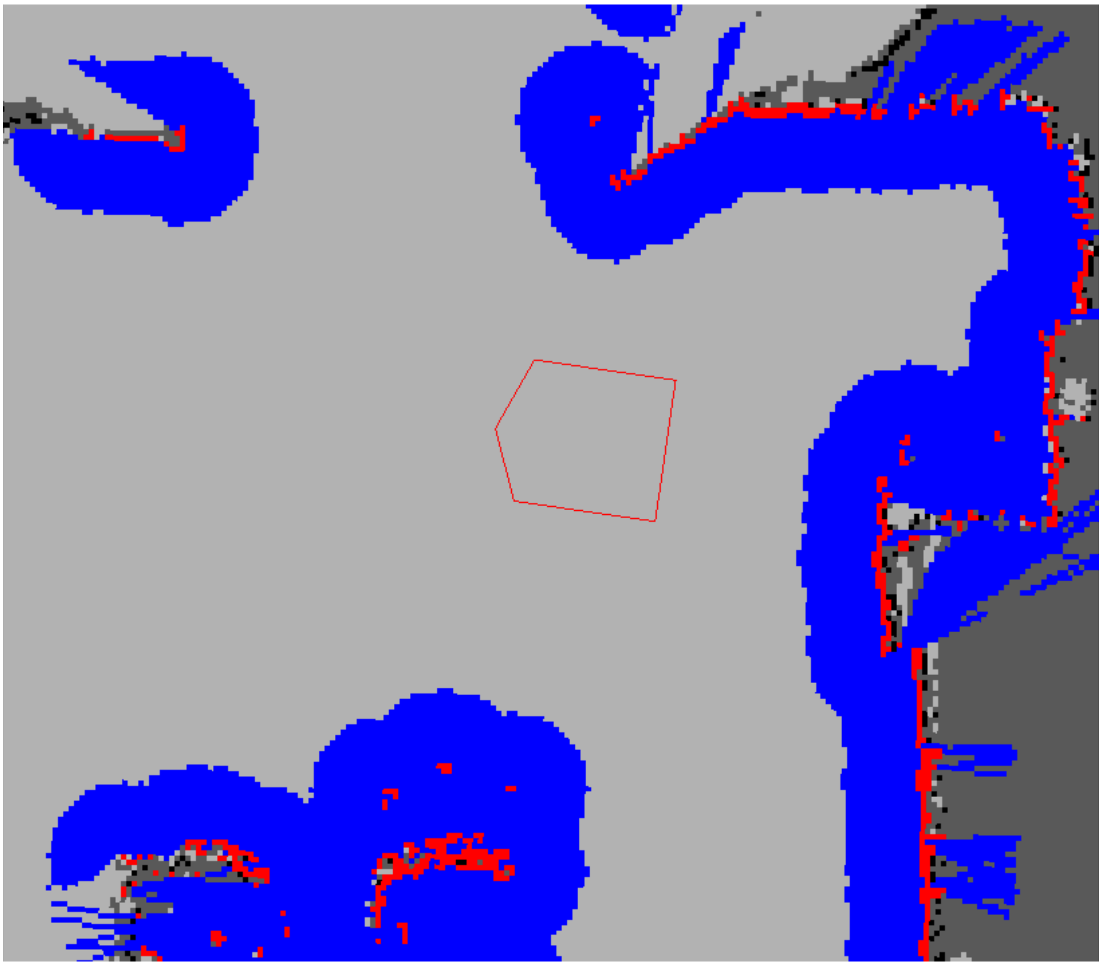


Figure 1.1.: costmap with obstacles and inflation [6]

Cost Values

The values in a costmap can be in the range [0-255], but the underlying structure categorizes them in the following 3 sections:

- lethal obstacle
- free space
- no information

1.3.5. marking and clearing

Obstacles in the costmap can not only be marked, but also cleared by the subscribed data source. For each data source a configuration regarding the clearing and marking permissions is necessary. For clearing the costmap uses a raytracing algorithm, which allows the costmap to handle moving obstacles[6].

Inflation

Inflation is a process where a occupied cell is inflated by over the distance decreasing cost values for a configurable radius, as pictured in Figure 1.1.

This process is used by the default plugin inflation_layer with the cost distribution pictured in Figure 1.2[6].

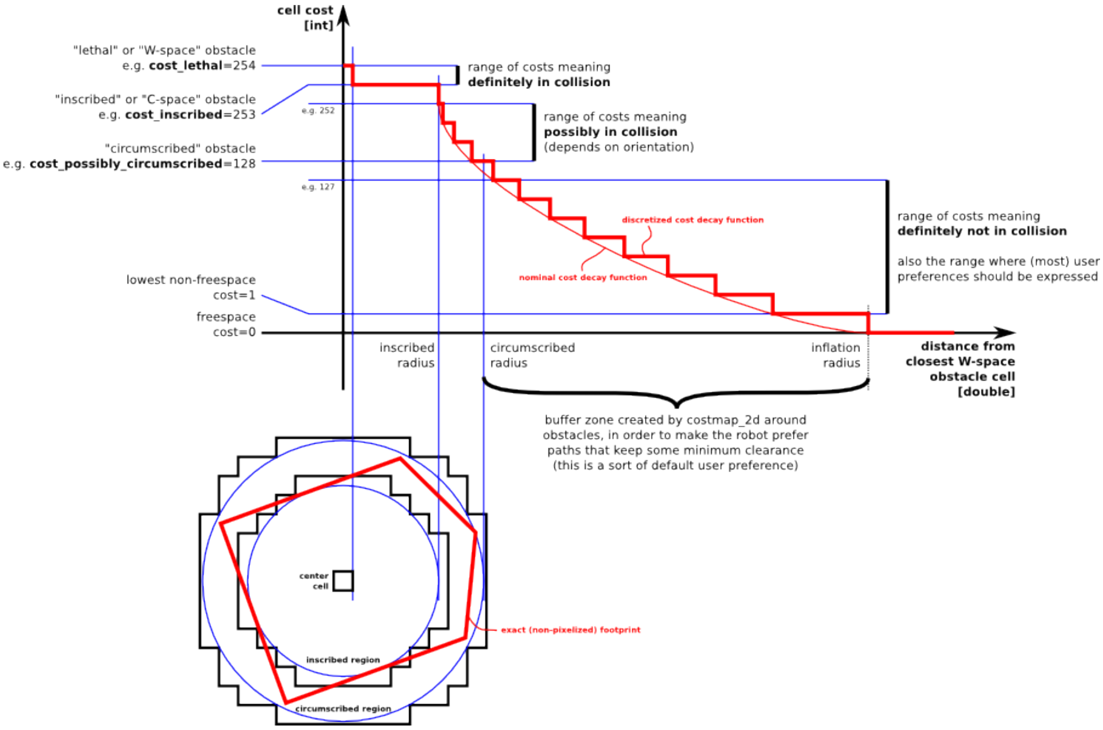


Figure 1.2.: cost distribution and classification [6]

The usage of inflation has two reasons. First it is used to close gaps between two measured obstacles and therefore usefull, if the sensor resolution is relatively corse. The second reason is to prevent the planner from getting too close to obstacles. Therefore the inflation radius is typically set to slightly more than the radius of the robot[6].

layer

The costmap uses a layer structure of plugins that can handle different tasks. The costmap then combines the data from each layer, to produce the final costmap.

By default the costmap_2d package offers the following 3 layers:

- **static map layer** is a layer that converts a prerecorded map into obstacles.
- **obstacle layer** is a plugin that handles input from sensor sources. The plugin marks and raytraces obstacles in 2D. It can handle LaserScans PointCloud and PointCloud2.
- **inflation layer** handles the inflation of lethal obstacles in the costmap.

Furthermore the plugins Social "Costmap Layer" and "Range Sensor Layer" are offered. Using the pluginlib interface and the costmap libraries one can develop custom layers for the costmap to achieve special behaviour of the robot.

1.4. cartographer

Cartographer is a Lidar based SLAM developed by Google. In contrast to gmapping it is based on loop closure to ensure real time mapping even in relatively big environments. Submaps are considered for loop closure, if they are close to each other. A scan matcher tries to find constraints between the submaps and the current scan. When searching for loop closure constraints at a certain rate one can achieve basically instantaneous optimization of the map just by the fact that the current scan is similar to one of the underlying submaps[7].

1.5. Carolo-Cup

The carolo cup is an event hosted by University Braunschweig and is an event in which the teams of many different universities can compete against each other and present their work and progress in the field of autonomous driving.
There are two different levels of difficulty the carolo basic cup and the carolo master cup.

2. Limitations and Requirements

Before diving into the details and developing concepts the guidelines and requirements of the project have to be defined.
This thesis aims for a navigation of a robot in an environment that is similar to the carolo cup but deviates at some parts.

2.1. Robot and Environment

While the robot itself has very strict regulations in the carolo cup theses will not all apply here.

For testing purposes the entire robot with all sensors and the drive controller will be simulated.
The robot that will be used is a differential drive robot from the company Parallax with a diameter of 450mm.

Equally to the carolo cup regulations the lane width is defined by double robot width and will be set to 900mm.
The Robot will be equiped with the following sensors:

- Lidar
- Wheel encoder
- IMU
- Camera

Additional it will feature a motor driver for differential drive steering.

2.2. Software

Generally the software will be developed for ROS-Noetic.
The programming language will be mostly C++ to allow uniformity in the software stack.
Like in the carolo cup the software is not supposed to have any connection to systems outside of the robot.

2.3. Simulation

Since the environment will be simulated the Simulator has to have the following features.

- Sensor plugins with configurable error and ROS interfaces
- Differential drive plugin
- custom models integration
- URDF conversion
- Not too computationally heavy

The simulation will mostly focus on sensor data. That is why sensor plugins with configurable error and a ROS interfaces are needed. Like this the data will be as representative as possible to the real world.

In addition to the sensor plugins the simulator needs to provide a plugin for differential drive steering. This will be the replacement for the motor controller of the real robot.

Custom models is a strict requirement since this thesis focuses on a very specific robot. Furthermore the integration of custom models is necessary to put the robot in different road scenarios.

A URDF conversion plugin is very important like this differences between the simulated robot and the tf-tree in ROS can be avoided and the robot will be defined in one file only.

To get the best correlation between simulation and real world the simulator should be able to run as close to real time as possible. This will make the simulated sensor data way more reliable and puts the nodes of the navigation_stack under the right load.

2.4. Navigation

The navigation is supposed to cover free driving with out obstacles, as well as with static obstacles avoidance. It will not cover dynamic obstacles, road sign detection or driving situations like intersections and parking.

Development of an entire stack exceeds the content of this thesis, so an open source navigation project will be used which needs to satisfy the following requirements.

- Sensor input.
- Goal pose input
- 2D mobile platform support using the conventional drive systems like ackermann and differential

- Path planning in respect to the robots kinematic and shape, as well as the environment detected by the sensors.
- Path planning and navigation in totally unknown environments
- Velocity output as linear and angular velocities

3. Concept

The selection of navigation stacks is quite sparse. Especially considering the defined requirements. In general when it comes to navigation there are only two well documented stacks to choose from.

- navigation_stack
- MoveIt

In contrast to the navigation_stack MoveIt is largely used for the path planning and navigation of industrial robot arms and therefore not suited for this application.
The navigation_stack provides a general setup proposal which seems to be a good starting point for robot navigation, but it has multiple parts that need be modified to adhere to the defined requirements.

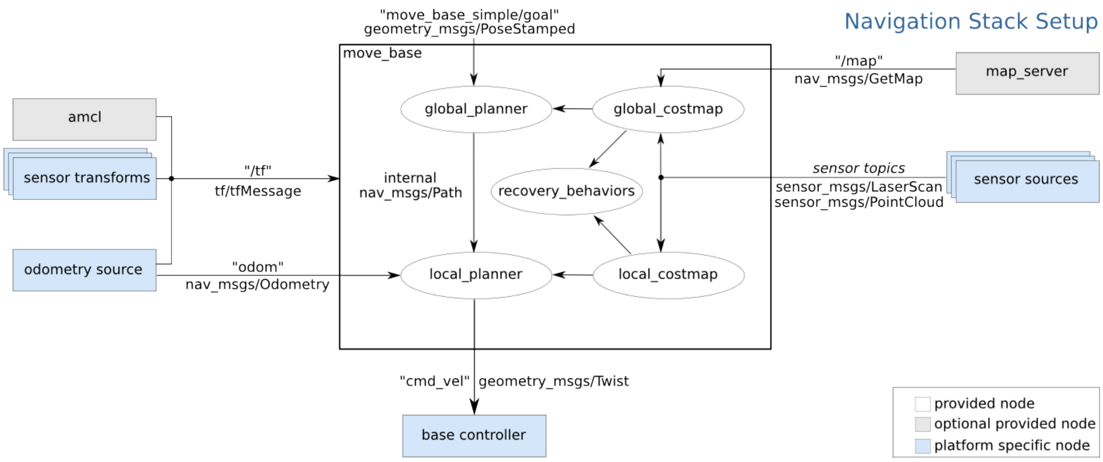


Figure 3.1.: navigation stack setup

3.1. platform specific nodes

Since the platform specific nodes are unique to each robot these will need to be adjusted.

3.1.1. sensor sources

The following sensors will be predefined for this concept:

- lidar

- wheel encoder
- imu
- camera

Since these sensors will certainly have noise we firstly need to add additional filters for the sensor signals.

3.1.2. Odometry source

The odometry is all ways a result of sensor data. Therefore we can make a direct connection between the sensor filters and the odometry input. These Filters will also feature nodes that transform the incoming sensor data into a useable format.

3.1.3. Sensor transforms

To know the position of every sensor relative to the robot and his odometry the tf_tree has to be build. While this can be realized using static transform publishers there is also a cleaner way using the ros package robot_state_publisher. It then requires a robot description in URDF format that specifies the relations between everything mounted on the robot. The transformation between the base of the robot and the odom will be build by the filtered odometry.
The remaining platform specific nodes are all available since the simulated robot will be used and provides a motor controller, as well as all of the sensors and data sources.

3.2. SLAM

The map is a representation of the robots environment. If this is known from the start it is known, where the robot can go and where not.
In this scenario the robot will always start blind, meaning it will not know anything about its surrounding other than that it can expect a road to be somewhere, which makes the map_server of the navigation_stack and its functionality to convert prerecorded maps redundant.
Adding to that amcl (the localization package of the navigation_stack) will no longer work since it tries to find a position in a predefined map based on the current sensor signals.
Knowing the environment and the position of the robot in it is an important part in robot navigation since it allows to send goals that are relative to the environment and not to the robot. That is why the usage of a SLAM algorithm becomes highly useful. This node supplies both the current map and the position of the robot in it.
The big improvement this concept has from SLAM is, that goals could be extracted from the map, instead of being estimated after the first round on the track.

This node will publish the transform between the map and the odom frame so the position of the robot and every sensor signal can be determined relative to the map frame.

Unfortunately the data that can be fed to the SLAM node is very limited, since it is not guaranteed, that the lidar all ways sees static obstacles. An other data source for the SLAM algorithm could be the points extracted from the road detection. The problem with these is, that they don't have a lot of features in them other than the corridor like point distribution which.

This significantly decreases the reliability of the map which therefore will highly depend on good odometry.
To get the best map both of the data inputs have to be used, which results in the need of a SLAM algorithm with multiple inputs.

3.3. Provided nodes

The provided nodes do not have to be reordered but the recovery behaviors will be removed. These will be incorporated in the incoming goals instead.
To define the tasks of the nodes of move_base they will be sepparated into two sections a global and a local stage, each of the stages consists out of a planner and a costmap.

3.3.1. global stage

The general task of the global planner is like described in the theoretical knowledge to plan a rough path through the grid that will not collide with any obstacle.

In this scenario the global planner will be required to guide the robot on the correct lane as well. This results in two additional requirements:

- the global costmap needs to incorporate cost that set a preference for the right lane but allow the global path to go to the left in case of a blockage
- the global planner has to respect not only lethal but every cost

3.3.2. local stage

The local stage on the other hand has the task of finding a for the kinematic of the robot feasible path that does not collide with objects.
This Path needs to be close to the global path and follow the lane changes dictated by the global stage but it needs to be able to separate itself from the global path if necessary.

3.4. PoseFinder

The job of this node is to extract the pose of a goal from the sensor data or the map (if available).

The requirements of this node will be defined in the "Configuration and testing" section

3.5. sensor filter

Here the entire data processing takes place, which converts the sensor data into a usable format. This block in the concept consists out of the following nodes:

- **road_detection** will extract approximated polynomials for the road markings and the lanes from the camera data.
- **markfreespace** needs to publish points that need to be inflated to generate restricted areas in the costmap. Also produces combined data of the road_detection and the lidar for SLAM.
- **laser_filter** removes error points at the edges of obstacles
- **robot_localization** improves the odometry by fusing wheel encoder data and imu data

The road_detection and the markfreespace nodes are obligatory while the two remaining ones need to be implemented to improve the usability of the data of exactly this setup and might not be needed on a different robot

3.6. Resulting concept

The following simplified schematic is the concept of the navigation stack setup. Not all of the connections between the nodes will be highlighted, to keep schematic simple.

The PoseFinder will first find a goal based on the current state of the map and the sensor data and sends it to the navigation stack. This then uses the filtered sensor data to determine, where the robot is, where it is allowed to go and where not. The cascading planners then determine first a rough, collision free, path on the correct lane and then a path that is possible for the kinematics of the robot. This path then gets converted to velocity commands and sent to the differential drive controller.
This procedure will be repeated at a configurable frequency so the robot will never reach its finish. This is necessary since it is unknown if the goal, in the space that has not been explored yet, is perfectly on the future road, or if it is in an obstacle.

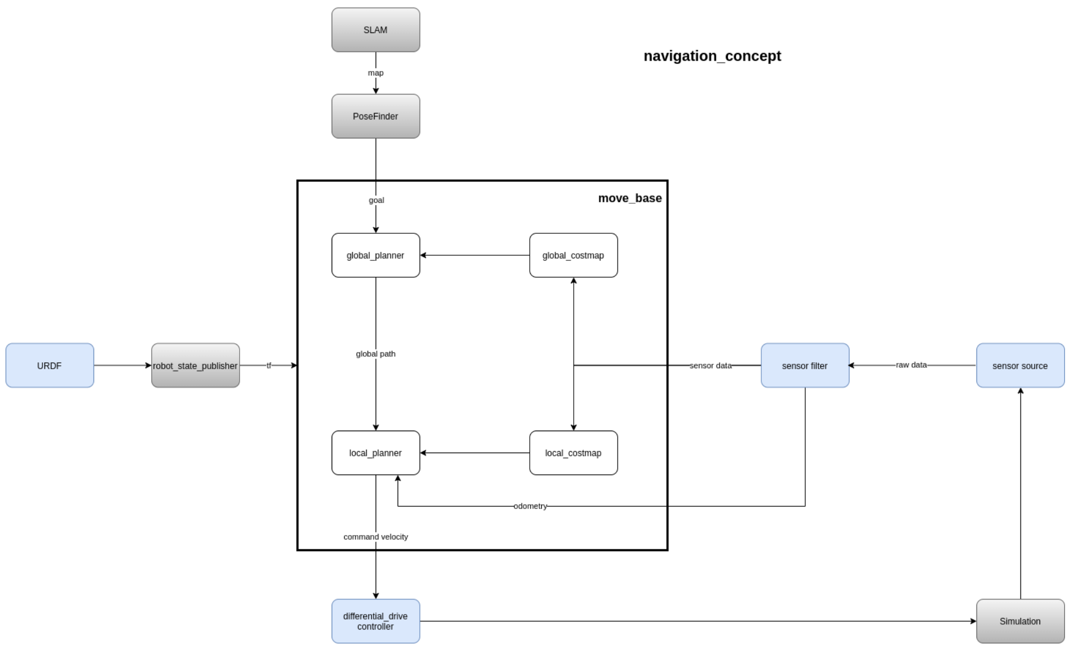


Figure 3.2.: Updated navigation concept

4. Selection

This chapter will cover the package selection and the setup of all nodes in the concept that differ from the recommended setup.
The selection process will mainly focus on the defined requirements.

4.1. Simulation

There are many options, when it comes to robot simulation, which makes a selection mandatory. The chosen Simulator then needs to be configured and equipped with models, sensors and a drive system.

4.1.1. Selection

To begin of the selection process a group of reasonable simulators needs to be collected. The two selected options are Gazebo and V-REP since they are the two most used robotics 3D simulators [8].

Both simulators seem to full fill most of the defined requirements to a certain extend, while Gazebo seems to have a easier installation process and integration into ROS since it is included in the default packages of ROS noetic [9] since it developed by the Open Source Robotics Foundation as the default simulator for ROS.
As well as Gazebo V-REP offers plugins and URDF conversion for custom models but an even bigger selection of mobile robot models. Which unfortunately can only be used as examples since the required models are very specific.
In contrast to V-REP, Gazebo does not contain an integrated model editor.
Based on computational load compared in the Paper of L. Pitonakova et al[10] and the setup differences between both simulators, gazebo will be chosen for this project. But both simulators would have been an excellent choice.

4.1.2. Model

Since Gazebo doesn't have an integrated model editor the freeware blender will be used for the model generation of the environment.
As described in the requirements the robot models will be generated using URDF which will be covered later.

4.2. move_base

As described in the concept the navigation stack will be used in this thesis. Therefore the action move_base will be used to get the robot to a given goal.
Move_base incorporates nav_core and costmap_2d and their internal nodes which need to be selected according to the requirements of the navigation.
The selection of these nodes will be separated into global and local since the individual nodes are highly interconnected and the wanted behavior affects both.

4.2.1. global stage

planner

Since the planner needs to adhere to the nav_core interface [11] the following selection of the documentation can be used as a possible selection.

- global_planner
- navfn
- carrot_planner

Here the choice is fairly easy, since base_global_planner is the successor of navfn. It still supports the the behavior of navfn, but it offers more options, such as A* planning algorithm instead of Dijkstra.

A* and Dijkstra explanation in theoretical

Carrot planner isn't suited for this use-case since it doesn't fullfil the requirement of being able to cover lane changes since it will not generate plans, that go around obstacles. Instead it will generate a straight path to the goal and shortens the path if the goal is behind or in an obstacle[12].

costmap

Since costmap_2d is embedded in move_base the package doesn't need to be selected itself. Instead of the package the layers of the individual costmap need to be selected.

Judging from the required behavior of the global planner the global costmap needs to have information about lethal obstacles, road markings as well as a defined preference for the right lane.

For the road markings and the obstacles a combination of an obstacle layer and an inflation layer will be needed.
This results in lethal cells in the costmap for each point of the road detection and the lidar. These points will still have gaps in between each other which will be filled using the inflation layer that inflates every point with a defined cost distribution.

Unfortunately there is no costmap plugin that can be used to make certain areas more expensive than others without marking them as lethal. This leads to the following concept of a custom costmap layer.
The only information about where the lane is comes from the road detection, which outputs polynomials that approximate the road markings. The aim is to make a gradually increasing cost from the middle of the right lane to the left road marking. Therefore the left road marking needs to be inflated using dynamically adjustable parameters for the cost distribution.
To guarantee that the right lane still has space without cost the right road marking needs to be inflated too. To simplify lane changes in the case of obstacle avoidance these obstacle need to have a cost free zone around them. The plugin should also feature a reset option as a recovery behaviour
All of this results in a layer with the following abilities

- input for points
- input of point individual inflation parameters
- rasterization of the individual cost distribution
- service to reset the cost in the layer

4.2.2. local stage

planner

In contrast to the global planner there are way more options for the local planner node. Like the global planners the local planners will be selected using the options offered in the following description of the nav_core.

- base_local_planner
- dwa_local_planner
- eband_local_planner
- teb_local_planner
- mpc_local_planner

[11]
To choose the local planner the requirements have to be defined first.

Since the global planner is taking care of the obstacles and the roadlanes the local planner has the general task of following the global path and creating a command velocity that is feasible in regards to the kinematics of the robot.

Smooth lane changes are highly wanted in this project. This will help the camera and therefore the road detection to keep seeing the road during a lane swap. To achieve this, the planner is supposed to drive close to the global plan, but if it is more efficient smooth out edges of it.
Furthermore the local planner needs to have a good performance in tight corridor situations, since those will often be caused by obstacles blocking one lane.

During all of this the local planner needs to use the information of lethal obstacles to prevent crashes in its optimized path.

Base and DWA are two planners that are included in the navigation stack. Similar to the global planner selection we choose the successor and therefore DWA, which according to Kaiyu Zeng is the general recommendation for mobile robotics platforms.

ros navigation tuning guide kaiyu zheng cite

In addition to DWA we choose one of the planners with an elastic band approach. The choice between these two is difficult since they both share the same base principle. The decision here is to use teb_local_planner for the further selection since it supports carlike robots and therefore satisfies the requirements for the navigation.

description elastic band and cite teb

choose teb local planner

4.2.3. costmap

The local planner is only supposed to generate velocity commands that lead to no lethal collision. So the local costmap will have the same plugins as the global one without the dynamic inflation layer that is used to guide the robot to the right lane.

4.3. Odometry

4.3.1. Encoder

4.3.2. IMU

4.3.3. Improvement using SLAM

4.4. Laser_Filter

4.5. PoseFinder

As described in the concept the purpose of this node is to determine the pose of the next goal and send it to the move_base action client. Therefore it will need to process the data from the road_detection and if available from the SLAM map and determine a feasible goal.
Since the Robot should never reach a goal the PoseFinder needs to determine new goals at a configurable frequency. Furthermore it needs to predict the path of the road so the

robot has a goal to drive to even if the road detection does not detect any part of the road.

4.5.1. Using current camera data

The easiest way to get new goals would be to take the last point of the polynomial provided by the road detection as the position and calculate the yaw angle of the new goal with the gradient of the last two.

While this is a logical approach in an ideal scenario, it certainly will not work in a realistic one since we can't assure a continuous data stream from the road detection.

This is mostly caused by the camera not always seeing enough of the road which can for example be caused by the following reasons:

- obstacle covering the markings
- while driving a corner the camera isn't always pointed tangential to the curvature
- steering during obstacle avoidance
- noise in camera data

This suggest that a prediction for the possible upcoming road is needed. To a small extend this is provided by the polynomials of the road detection, but the have a restricted domain, after which the error will rapidly increase.

4.5.2. Approximations

Since the road mostly consists of circles of varying radii and origins, it is self evident, that using the polynomials in their restricted domain to represent a section of a circle will give a better estimate.
This circle can be calculated using the following linear least-square approach:

In theory this approximation should work for almost straight sections as well, but the radius and the origin will trend to infinity and caused by the camera noise and the inaccuracy of the road detection the representation of the road will get worse and worse.

This leads to the following linear least square approach for straight lines:

Switching between the result of the two approximations will result in the best result for both scenarios. This switching will be triggered by the radii of the approximated circles exceeding a configurable threshold.

The next step is a goal extraction from the chosen approximation. This goal will be calculated at a given distance from the robot origin on the approximated route. In contrast to the approximated lines, the circles have a defined angle of trustworthiness. Otherwise small circles would cause the goal to be way of the prediction. The orientation of the goal will then be determined using the differentiation of the function.

With the calculated points on either both circles or on both lines the mean can be calculated and represents the new goal for the robot.

4.5.3. goal from map

If the robot is running a SLAM algorithm during the first round the map should be finished once the robot passes its start point. Then the approximation is unnecessary since extracting goals directly from the slam map is more efficient and provides goals that will always be on the road. Furthermore the distance, at which goals will be found can be increased, which results in higher speeds and/or lower planner frequencies.

To find a goal in the SLAM map a circle rasterization algorithm will be used.

This algorithm finds every cell on a circular path around the robot and its associated value in the map. The values outside of a given FOV (field of view) can be eliminated. Then the remaining values with a larger likelihood than a configurable threshold will be reduced to one point by taking the mean value of all of them.

The orientation of the goal is then determined by using the approximation algorithms.

4.6. MarkFreeSpace

The purpose of this node is to provide data to the SLAM algorithm as well as to the costmaps. This data consists from the points on the polynomials of the road detection in combination with the filtered points of the laser scan.

For SLAM and the obstacle layer of the costmap the node simply transforms the data into the same frame and casts it into the form of a sensor_msgs::PointCloud2.

The data of the dynamic cost layer has to contain more information. It is in the form of a sensor_msgs::PointCloud and contains channel values for point individual inflation radius, min-cost and max-cost. By giving the Layer point individual inflation parameters the node provides information about the cells on the left road marking that need to have inflated cost values and information about the points on the right road marking that should have a clean zone around them. Furthermore the node provides points of obstacles that are on the road and clears the inflated cost around it. Which will allow the global planner to make a plan for passing the obstacle.

4.7. SLAM

There are numerous Lidar based SLAM packages available for ROS, but with the defined restriction of being able to use both, the points extracted from the road detection, as well as the lidar data, most of the lidar based SLAM algorithms wont work since they only accept one input of the type sensor_msgs::LaserScan.
This rules out the popular options for Lidar based SLAM like gmapping and HectorSLAM and the well documented package Google Cartographer will be used based on its flexibility in regards to robot configurations.
This Package comes with an advantage of being based on loop closure which might be usefull when driving rounds in a circuit stile environment. The robot will drive the same route over and over again and thus the map could get more and more reliable over time even though the data is very self simillar and not great for SLAM.

Cartographer accepts numerous different input types including both PointCloud2 and LaserScan sensor messages. Additionally Cartographer can use provided odometry, aswell as IMU data to improve the result.

5. Configuration and Testing

This chapter will contain the configuration of all nodes of the concept. In addition the newly developed nodes have to be tested as well as the entire navigation concept.
The structure of this chapter is to address the outer nodes of the navigation concept first and then move to the inner nodes.

5.1. URDF and Robot State Publisher

5.2. Gazebo

5.2.1. Plugins

5.3. Filter

5.3.1. road_detection

5.3.2. laser_filter

The Lidar sensor is simulated with realistic noise and errors. This also introduces the well known edge errors in Laser distance measurement. Here the beam is split by an edge and the averaged measurement results in a measurement way behind the obstacle like further described in the paper of Klapa about the "Edge effect and its impact upon the accuracy of 2D and 3D modelling using laser scanning" [13].

When using a lidar to project obstacles in the costmap this error produces a lot of point. like lethal error obstacles that significately hinder navigation as visible in Figure 5.1.
To remove these error measurements the ROS package "laser_filters" can be used. Among many different filters plugins that can be constructed into a filter chain it features the filter plugin "ScanShadowFilter" that is developed to remove the veiling effect around obstacles cause by the edge effect [14].

5.3.3. robot_localization

When looking at the odometry published by the differential drive plugin in Figure 5.2 we notice, that it has rotational error.

The following nodes require odometry:

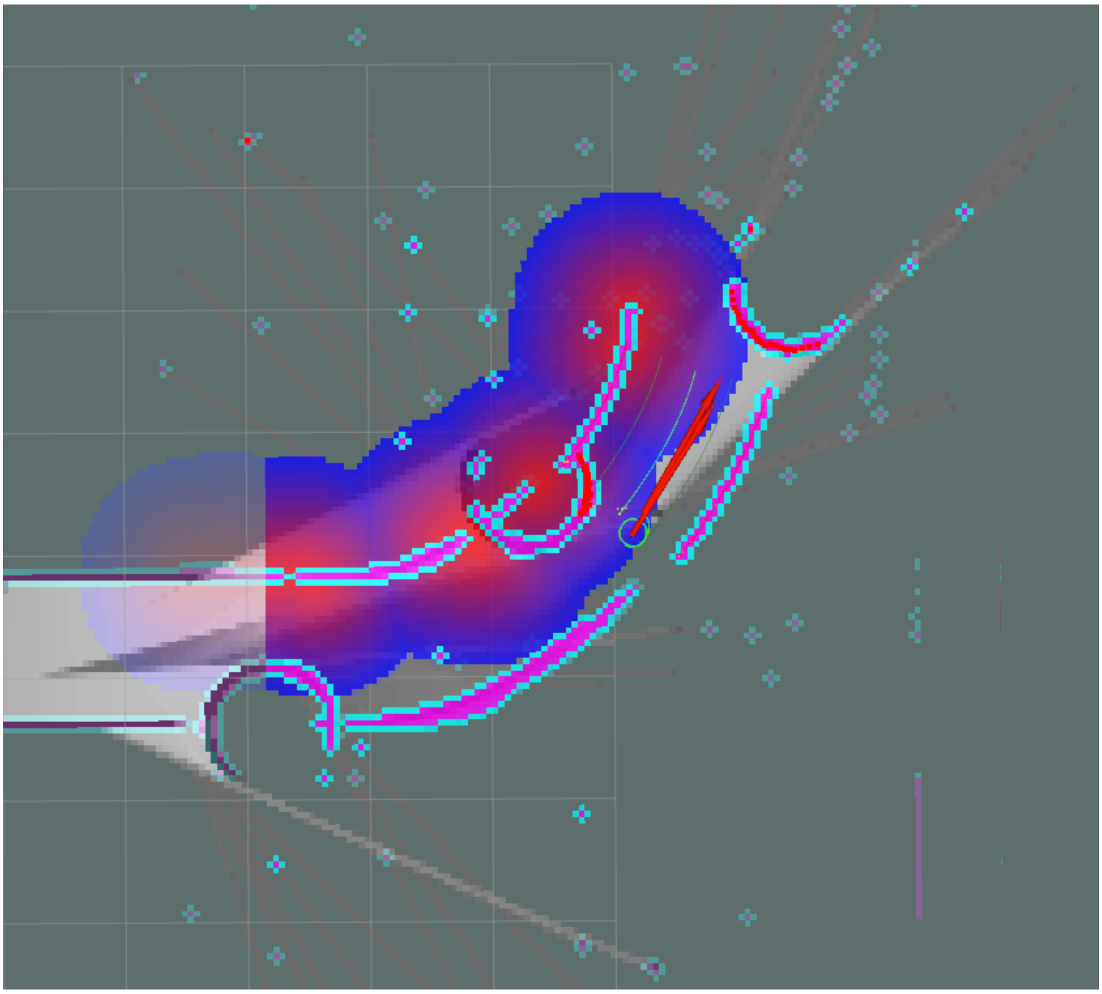


Figure 5.1.: Unfiltered Lidar data in costmap

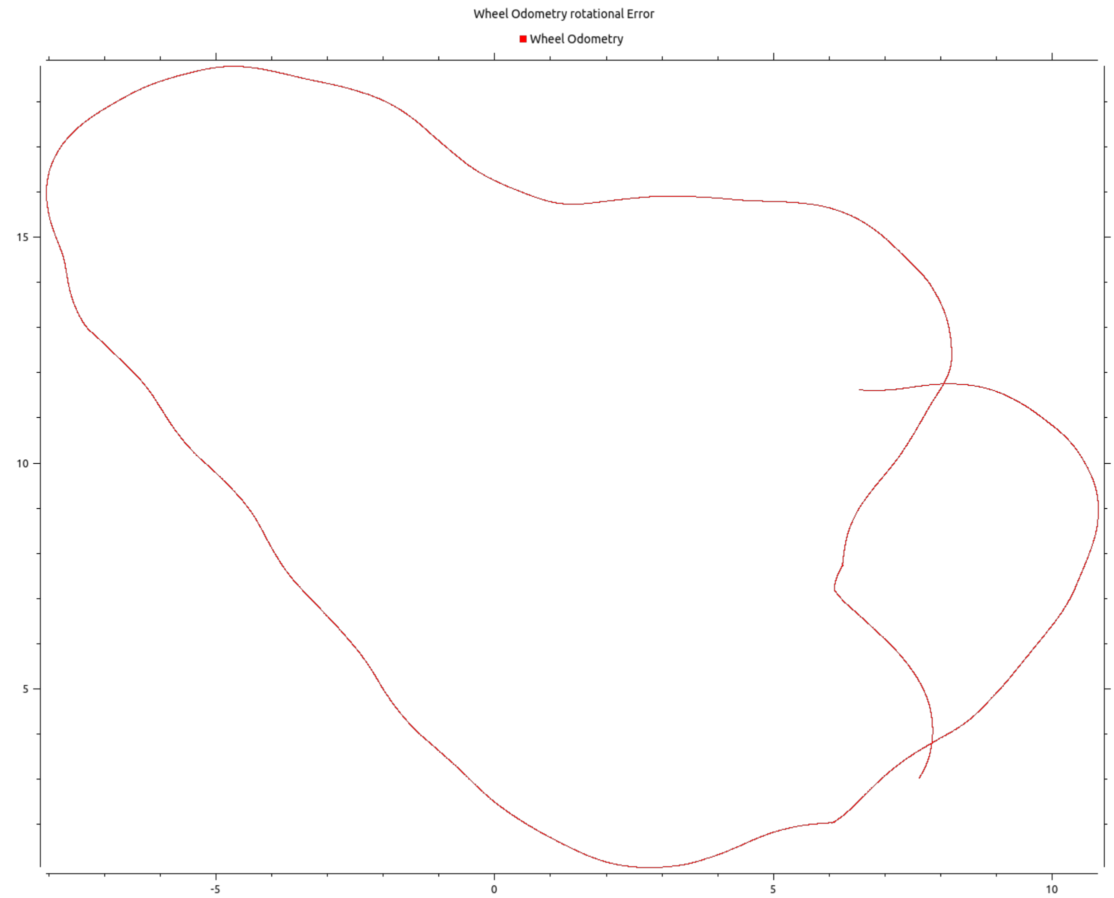


Figure 5.2.: Odometry from wheel encoder

- cartographer
- move_base
- posefinder

To improve the odometry the ROS package robot_localization can be used. It provides an extended kalman filter for the fusion of sensor data for odometry.

To counter the rotational error the IMU and the encoder-odometry will be fused.

The IMU provides the following data:

- orientation
- angular velocity
- linear acceleration

Based on the knowledge about the usecase of the IMU it does not make a lot of sense to fuse all of the data of the IMU and the kalman filter needs to be configured accordingly.

At first the measurements for three dimensional use can be removed, which are:

- pitch angle and velocity

- roll angle and velocity
- linear acceleration in z

The IMU is used for localization purposes, therefore the linear acceleration data are not interesting, since they would need to be integrated twice to be used for the pose. This would amplify every little error in acceleration over time making the odometry unreliable over time.

Accordingly the only things fused from the imu are the yaw orientation and velocity.

Just like the IMU data the integration of the wheel-odometry data has to be discussed which consist only from linear and angular velocity.
Here the most interesting part is the y velocity since the robot is relying on differential drive steering and therefore not able to have y accelerations other than drift.

In contrast to the acceleration values of the IMU the y velocity will be included since according to Tom Moore [15] it will give certainty that the robot has not moved in the y direction. Obviously the x and yaw velocity has to be included aswell. The position component of the wheel-odometry on the other hand will not be used, based on the fact that the position is already derived from the velocitys this would include the same data twice.

Unfortunately this does not solve the problem of the odometry correction yet. As visible in Figure 5.3 the odometry of the extended kalman filter has large jumps in it compared to the wheel-odometry. When observing it in real time the ekf odometry starts to drift and jumps back after a certain amount of time.
Looking at the linear velocities of both the ekf and the wheel odometry in Figure 5.4 it is noticeable that the ekf filter does estimate a continuous acceleration, whereas the velocity of the wheel encoders actually decreases.

To fix this a logical approach is to include more data about the robots movement. Fortunately robot_localization has an input for command velocities such as velocities produced by move base.
It is very important to set the control timeout to a value that is larger than the cycle time of move_base. Otherwise this will lead to translational offsets caused by too low estimate for the velocities like shown in Figure 5.7.
After the inclusion of the command velocity the acceleration limits can be set equal to the limits in the local planner.
When observing both the pose and velocities again it is noticeable, that the odometry has drastically improved as pictured in Figure 5.6, equally the velocities stay closer together as pictured in **??**.

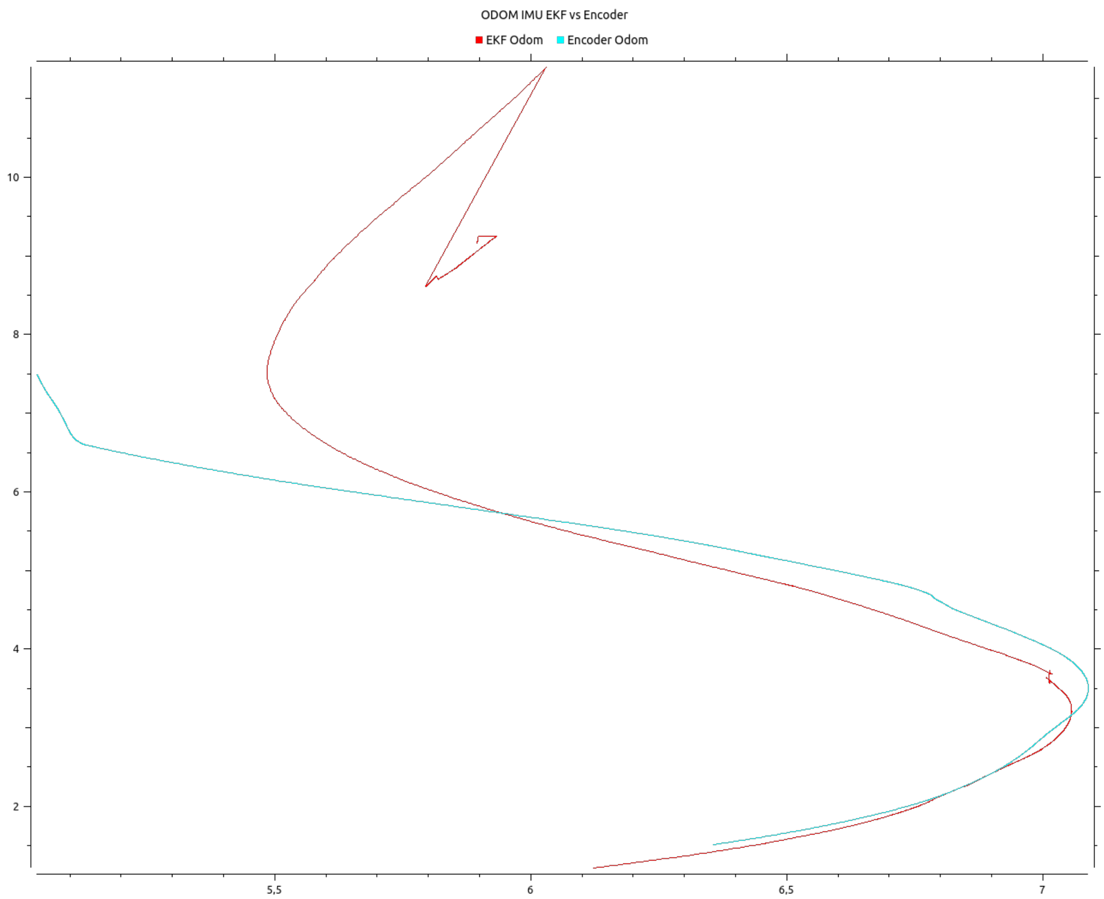


Figure 5.3.: pose comparison wheel odom + IMU

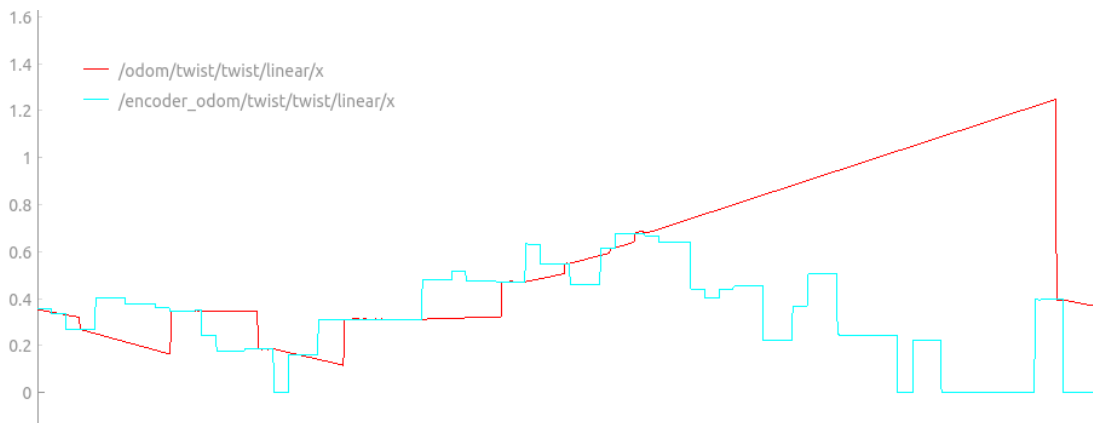


Figure 5.4.: velocity comparison wheel odom + IMU

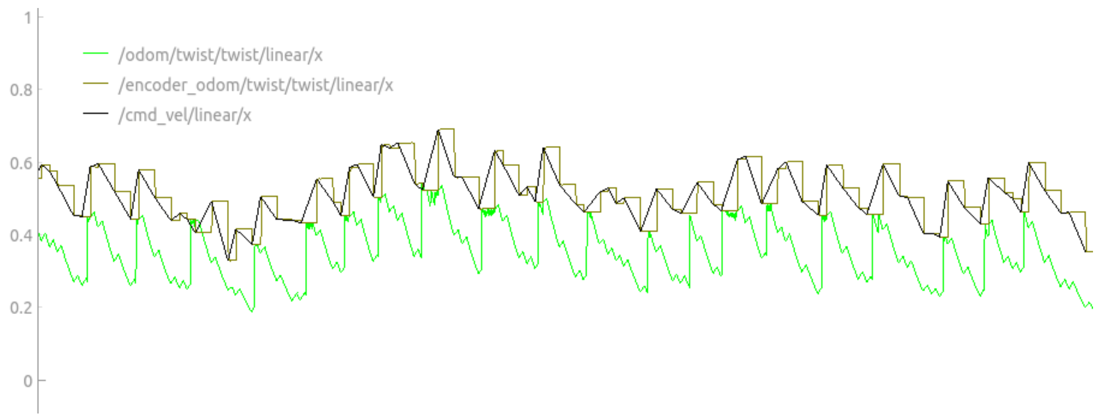


Figure 5.5.: velocity offset caused by too low control timout

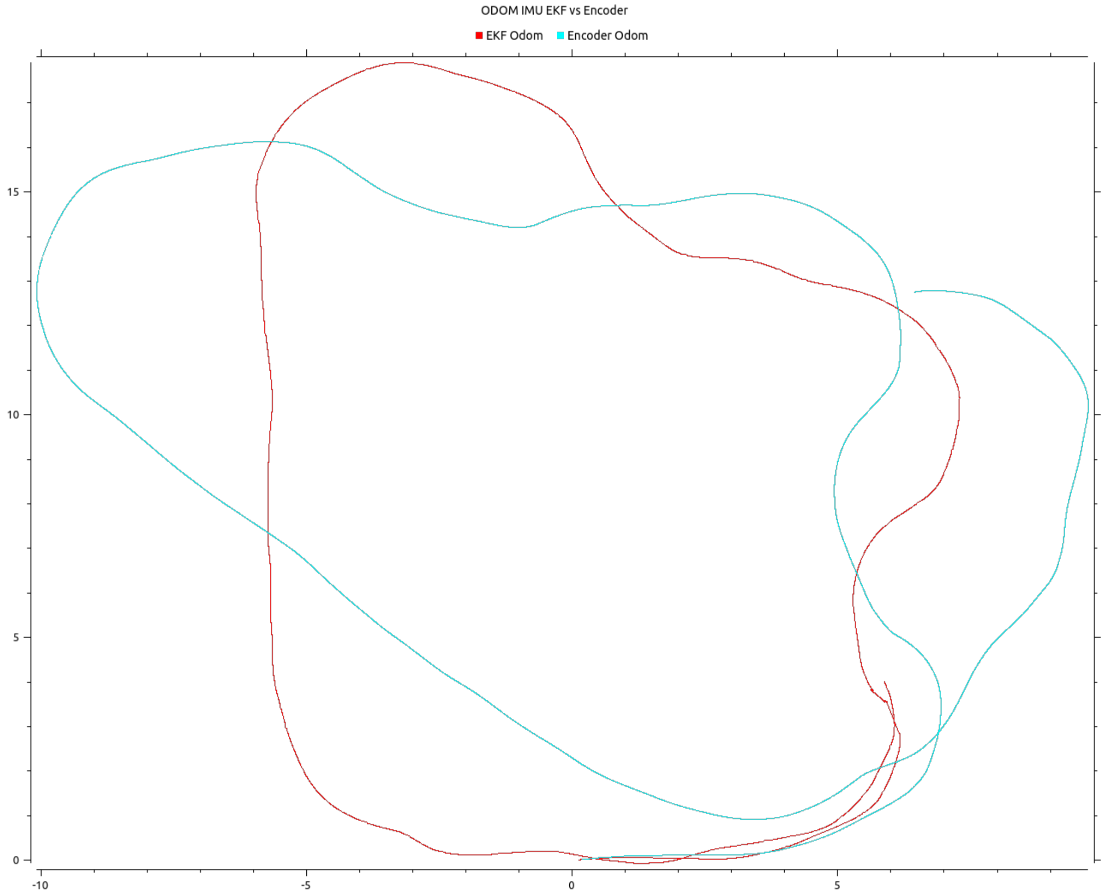


Figure 5.6.: Odometry comparison wheel odom + IMU + cmd_vel

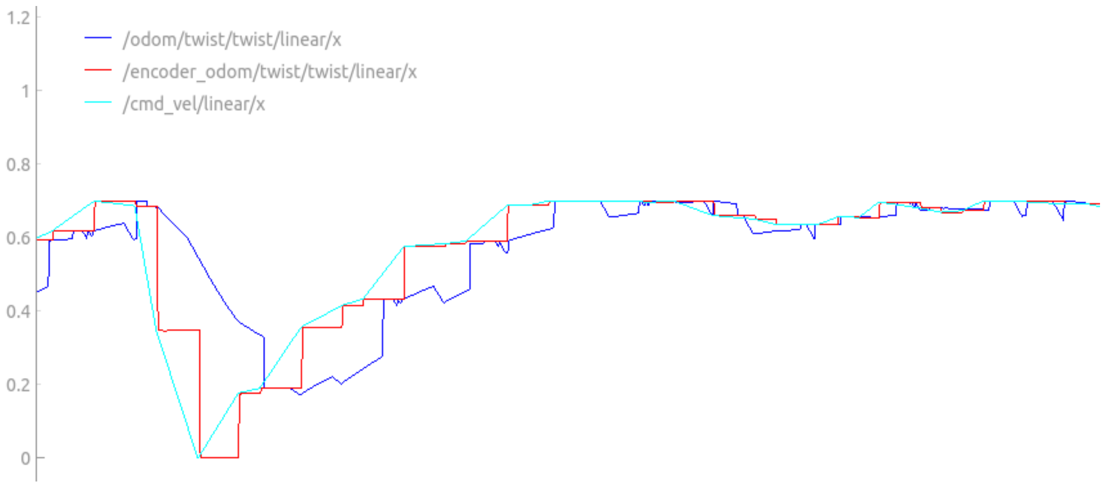


Figure 5.7.: Velocity comparison with cmd_vel

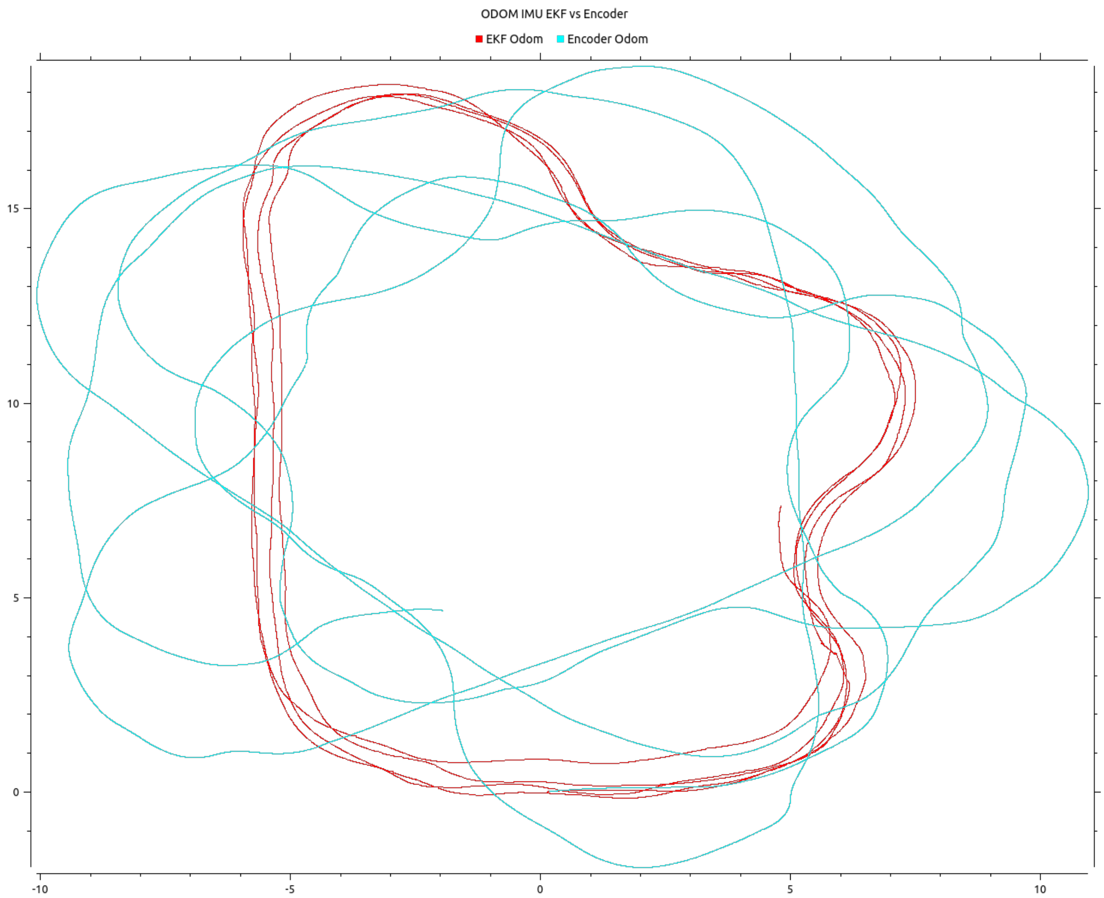


Figure 5.8.: Odom comparison multiple rounds

Even after four rounds the odometry has not gained a large error in both translational and rotational as pictured in Figure 5.8, furthermore the difference between the original odometry of the wheel encoders to the odometry from robot localization is quite remarkable.
Finally after the rotantional and translational errors are marginal the scale of the odometry needs to be checked. To isolate the different errors from the scaling error a circular track is build with a radius of 10 meter. Since the turning radius is constant this isolates the rotational error which can be seen at the graph of the wheel odometry in Figure 5.9.
The radius has to be chosen that high to display the potential error better.

Since the robot is driving on the lane and not on the middle road marking the expected radius is 10,45 meter. When evaluating the EKF odom in Figure 5.9 we see that the scale is very precise.

5.3.4. markfreespace

5.4. Cartographer

The goal of this node is to produce a map that gets more reliable over time.

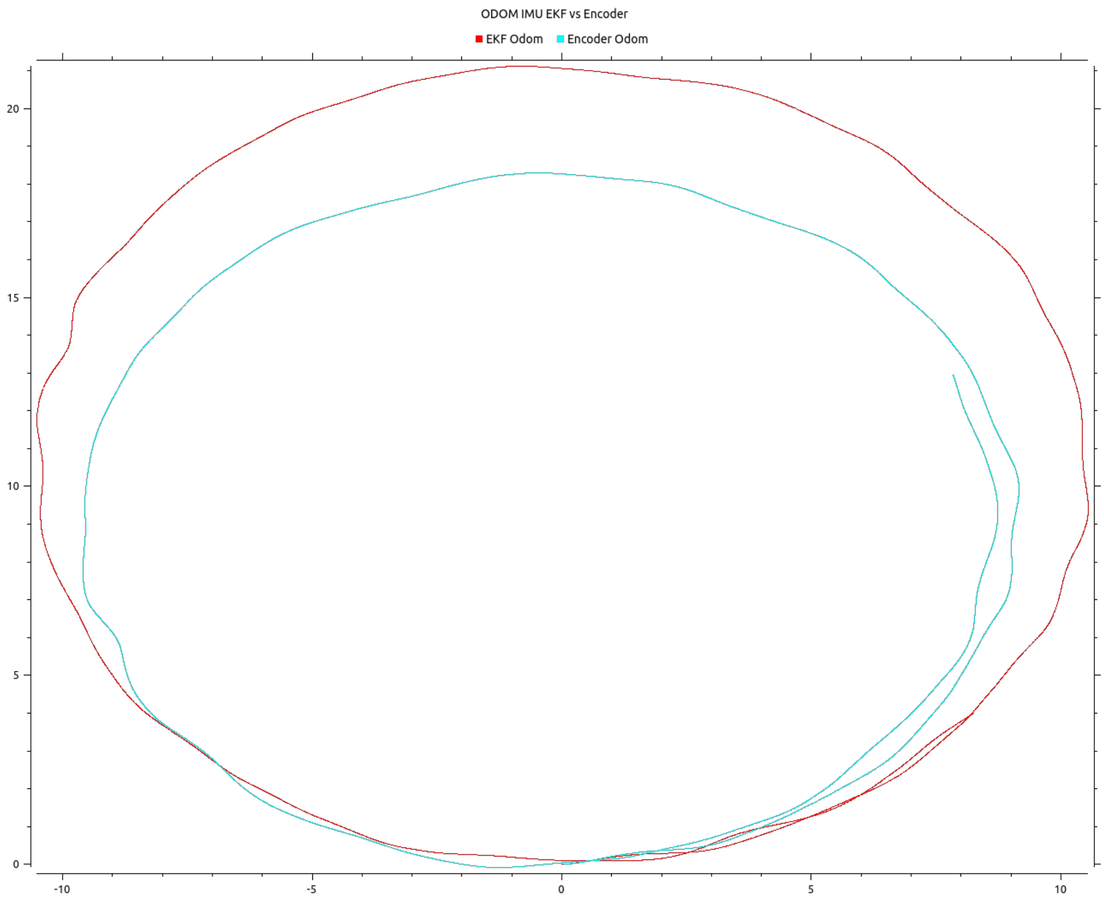


Figure 5.9.: Odom comparison circular track

Unfortunately the data available for Cartographer is very self simillar, meaning a straight road will allways look the same and therefore does not have sufficient features for proper loop closure. In contrast to the points from the road detection the lidar can actually supply such features and will therefore be a good improvement for the resulting map.

But it is not guaranteed that the lidar will even sees anything the SLAM algorithm has to work with the points of the road detection only aswell.
The basic configuration of cartographer is purely based on the setup of the robot. In this case cartographer is supposed to use lidar and the points of the road detection at the same time. To reduce the amount of times one of the sensor doesn't see anything these will need to be merged in the markfreespace node and cartographer will receive one PointCloud2 only.
To improve the map further the odometry supplied by the robot_localization package is used as an input, aswell as the IMU.
Furthermore cartographer will be set to 2d map building.

5.4.1. Tuning

With the basic configuration cartographer is not able to provide a reliable map and a tuning procedure has to be performed. Here the general recommendation of the tuning guide should be followed, which states to tune local SLAM first and disabling global

slam while doing so [16]. To tune the local SLAM the parameter of the tracjectory builder have to be adjusted.
The trajectory builder contains a scan-matcher, which will compare incoming sensor data a tries to align it with each other as good as possible. This behavior can be tuned by configuring the size of the linear and angular search windows and the weight for the rotation and translation of the incoming scans.
One more important setting is the size of the sub maps. These can be adjusted by the amount of scans they contain. Since the submaps will consist out of the scan matched obstacles it is important to set the size of the submaps not to high, if the incoming data will be very self simillar. Otherwise the scanmatcher will combine too many scans while shifting them over each other since they look so simillar. This results in both rotational and translational error.
As soon as the local SLAM produces a reliable result after multiple rounds of the robot the global SLAM can be activated and tuned.

The global SLAM has two options of combining the submaps the loop-closure, which will check, if the robot was at this spot already, and a scanmatcher, which will try to match the submaps to the current scan. For both of these the weights can be adjusted individually and like in the local SLAM the size of the window for translation and rotation can be adjusted. Reducing the window size to a minimum is aswell important when dealing with self simillar data, so the submaps will not be shifted on top of each other. The size should be chosen so the global SLAM can still correct errors of the individual submaps With these values and the submaps the global planner calculates constraints between the maps which will be valued between 0 and 1. These constraints can be blocked with a threshold value that will block constraints with a smaller value. like this the computation time can be drastically reduced and only the important constraints will be processed. Furthermore the weighting of the Pose of the odometry and the local SLAM can be adjusted, which can be usefull with bad odometry.

5.4.2. Testing

Testing of the SLAM will mostly consist out of a Black-Box Test meaning the only things that will be observed is the output and therefore the map. The SLAM will be tested in the following cases:

- Data purely from the road detection.
- Data from road detection and lidar scan with obstacles on the side of the road.
- Data from road detection and lidar scan with obstacles on the road.
- Long duration test with both road detection and lidar scan with obstacles on the side of the road.

During all of these test the navigation will solely work with the predicted goals since it is yet unsure if the SLAM map is even usable. Furthermore the same tuning will be used for all tests.

Data purely from localization:
The reason for this test is to check if the SLAM algorithm can handle Mapping with as least information as possible. This will make loop-closure difficult and cartographer has to work with the self simillar data only.
The aim is, that the robot can drive multiple rounds, on the track and cartographer produces an optimized map with little unmatched submaps and well connected road markings.
As long as the localization in the map works nicely the distortion in the map is less important and will not be tested.

The following pictures contain the SLAM map after the 1^{st},3^{rd} and 5^{th} round.

include pictures of test

Data purely from localization and lidar scan with obstacles on the side of the road:

Data purely from localization and lidar scan with obstacles on the road:
This test is meant to be the worst case for the SLAM algorithm during navigation. The purpose is to check how cartographer handles data loss during obstacle avoidance and lane swapping. The obstacles will be placed in two corners and therefore in the edge case where the camera has the worst chance of seeing the road because of the steering angle, aswell as on the straight section of the road to cover the case where the camera can not see the road during merging on an other lane. Furthermore the obstacles will be placed far enough apart so the lidar has only vision on one at the same time.

Long duration test with both road detection and lidar scan with obstacles on the side of the road:
Cartographer seems to not merge old submaps but process all of them allways, which will progressively increase computational load. Since the SLAM is supposed to be used during mapping this can become important with a lot of sensor inputs that offer constraint potential (lidar data) and long runtime. The same setup as in the 2^{nd} test will be used but the focus is on the moment, at which cartographer cannot optimize in real time caused by too many submaps and constraints.

Discussion of the test results
As proven by the first two tests, cartographer is well tuned and the map would be useable for goal extraction. The submaps are well alligned and the map has no huge translational or rotational offset.

The 3^{rd} and 4^{th} test on the other hand display the limitations of the slam algorithm. When obstacles are located on the right lane the allignment of the submaps fails and a lot of submaps cant be attached to the rest of the map. After passing the obstacle the map gets better again, which would imply that the map could be good enough for goal extraction, if no obstacle is near the robot.
The 4^{th} test proves that cartographer is not usable in SLAM mode during long time navigation an a circuit. This is caused that too many submaps are close to each other that share features for a constraint.
Based on these test results it is not feasible to use the SLAM map for navigation with obstacles on the road, since the map is just not reliable enough and it is not certain of there are obstacles on the road or not.

5.5. PoseFinder

5.6. Costmaps

Based on the fact that both planners are responsible for different tasks the configuration of the individual costmaps need to fulfill different tasks too. The requirements of the two planners will be compared in order to determine the configuration of their individual costmap.

As described in the theoretical knowledge of this thesis the costmap are structured in layers. This means that the data can be evaluated by different plugins before it will be combined into the real costmap.

It makes sense to first take a look at the general behaviour, that both planners share. In this case it is obstacle avoidance. This means that the lethal obstacles need to be marked in both costmaps.

To implement this the provided plugin obstacle_layer can be used. It will take incoming sensor data (sensor_msgs::PointCloud or sensor_msgs::LaserScan) and mark the points in the costmap.

Since the data here comes form the road detection and a lidar and both have a certain resolution it is unsure, if the result of a scanned obstacle in the costmap is actually a closed line or just points, since this highly relies on the resolution of the sensors and the costmap.

To fill theses gaps in the costmap the provided plugin inflation_layer can be used. It will inflate only the lethal obstacles in the costmap with a configurable cost distribution.

This setup is already enough for the local costmap, where as the global planner needs to fulfill the quest of changing lanes if necessary but all-ways preferring the right lane. For this a custom plugin will be needed that makes the transition to the left lane more expensive but still possible.
The final layer plugin setup of the costmaps results in the following:
global costmap

- obstacle layer
- inflation layer
- dynamic cost layer

local costmap

- obstacle layer
- inflation layer

The costmaps will both have the same size that should allways be larger than the distance, at which the goalfinder searches for goals.
Since both costmaps are rolling window costmaps they will reference the continuos frame "odom" and move with the frame "base_footprint".
Counterintuitively the robot radius needs to be set smaller than the robot actually is. Otherwise the slightly moving obstacles collide with the footprint and the global planner can not produce a valid path that the local planner can follow.
This footprint is only considered by the global planner, which is only required to provide a rough path. The local planner has its own setting for a footprint, therefore the obstacle avoidance will still work as expected.
The last remaining settings are the resolutions and the frequencies of the costmaps, which are chosen based on the performance of the navigation and the computational load.

5.6.1. dynamic_cost_layer

Enhancing to the plugins provided in the navigation stack this layer handles inflation of cells with a configurable cost decay and radius. While this seems to be similar to the provided inflation layer, this offers way more flexibility since it will inflate specific points by their individual radius and cost distribution and not just every lethal one by one fixed distribution.

This behavior can be used to inflate the left road marking in the global costmap to force the global plan on the right side of the road. The plugin can also be used to inflate cells with zero cost, which is useful to guarantee a cost free right lane or to give some free space around obstacles located on the road.

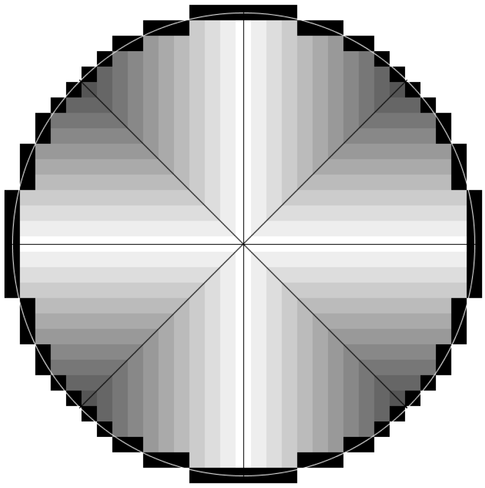


Figure 5.10.: modified bresenham rasterization with efficient surface filling

The layer receives a message of type sensor_msgs::PointCloud on a configurable topic. This PointCloud is expected to feature Channel Values for the inflation radius, the maximal and the minimal cost for each individual point.

Since we can't assume that the incoming points will be in the frame of the costmap the points in the costmap have to be transformed into the right frame using tf2.

To minimize the computation load a bresenham based algorithm for the circle rasterization will be used.[17] Now the point symmetry around the cell can be used to further minimize the computational load and only $\frac{1}{8}$th of the circle has to be computed. The rasterization process can be described by the following image.

Adding to the typical behavior of the bresenham rasterization the area of the circle will be filled using the point symmetry and by skipping overlapping points of the lines like in Figure 5.10. Here it is visible, that every row with the same color is only calculated once and then projected in all eight octants. The Black cells on the perimeter are the rasterized cells of the bresenham algorithm.
The cells within the circles perimeter are filled with the cost specified for that point. For this the following linear decaying 1$^{\text{st}}$ degree function will be used which requires the computation of the distance of the rasterized cell to the center of the circle.

$$cost(distance) = maxcost - distance * \frac{maxcost - mincost}{radius}$$

with:

$$distance = \sqrt{cell.x^2 + cell.y^2}$$

Since this will still require the usage of a square root for each cell in the circle This will be optimized as well.

The goal here is to use a function that contains only the squared distance, which still represents a decaying trend. This requirement rules out every function with an odd degree, as well as all functions with an x offset. This leaves all functions with an even degree from which we choose the 2nd degree function to reduce square operators. The comparison between the two functions can be seen in the picture below.

$$cost(distance) = maxcost - distance^2 * \frac{maxcost - mincost}{radius^2}$$

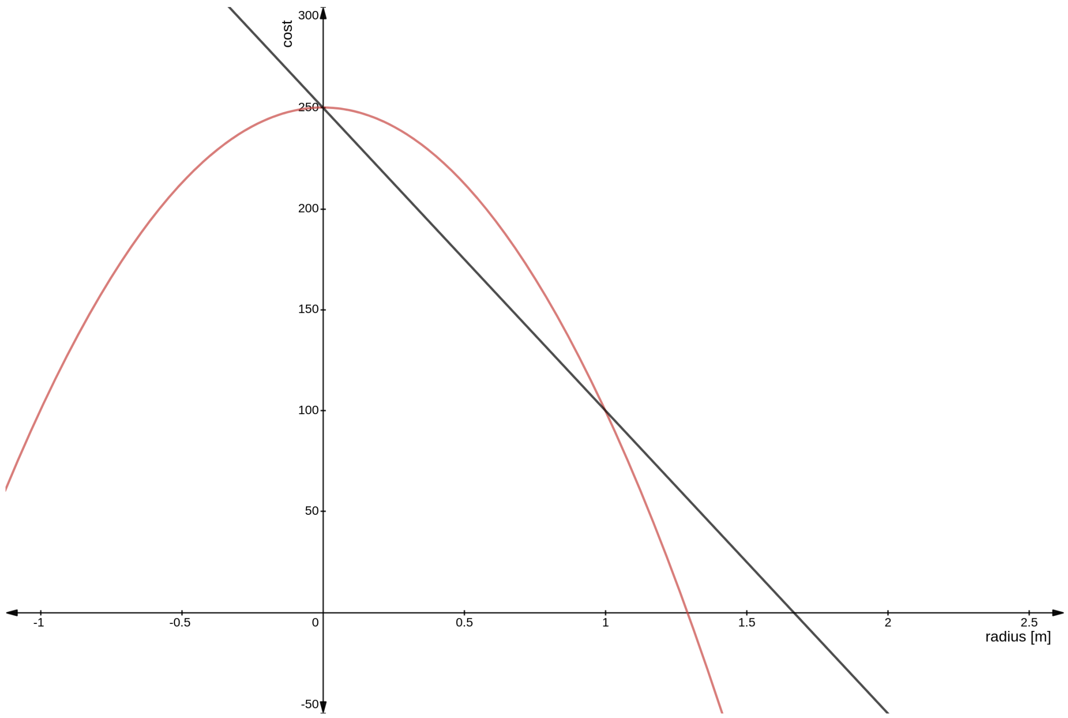


Figure 5.11.: cost distribution comparison with maxcost=250 mincost=100 radius=1

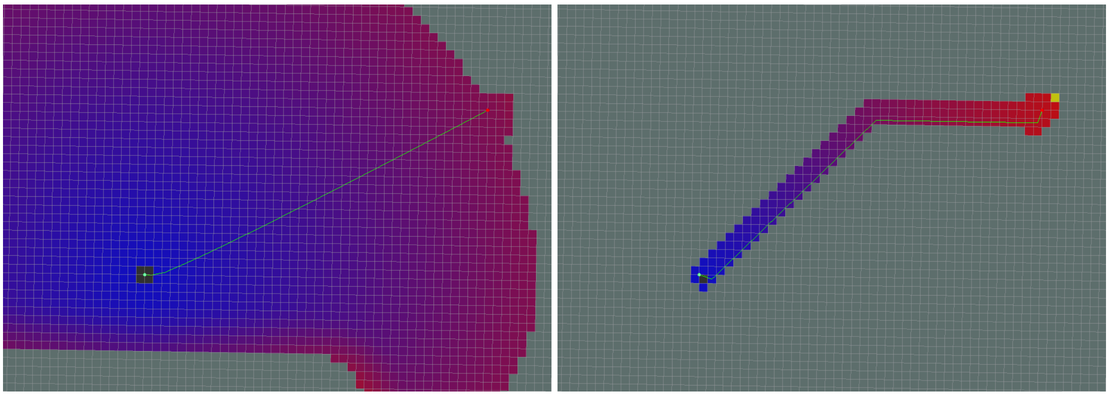
5.7. Planners

5.7.1. global_planner

There is not much room for configuration, when it comes to the global_planner of the navigation_stack. Probably the most important step for computation load is the choice of a planning algorithm.
The two algorithms that are offered by the global_planner node are Dijkstra and A*.

Dijkstra does only consider the cost to the start node and determines the cost to get from the start node to every other node, until it finds the goal, from which it then can backtrack the shortest/cheapest path.[18].

A* is based on the dijkstra algorithm but has one main difference. It considers not only the cost from the start to the current cell in the grid, but aswell a heuristic parameter, which is a general guess for the cost from the cell to the finish. The heuristic parameter is mostly bound to the euclidean distance between the cell in the grid and the finish. Like this the parameter will never over estimate the actual cost of the path[18].



(a) Dijkstra (b) A*

Figure 5.12.: planning algorithm comparison (grey cells are not observed)[19]

It is obvious, that A* is much more efficient in this use case since the robot will mostly go straight or in a slight curve. Therefore A* will be chosen for the global planner.

Another important setting is necessary since the global costmap is used as a rolling window costmap. The global_planner will be default outline the global costmap with lethal cost to prevent the global planner to plan outside of a fixed costmap. This behaviour results in artifacts, after the map has moved together with the robot which hinder the planners from finding a path.

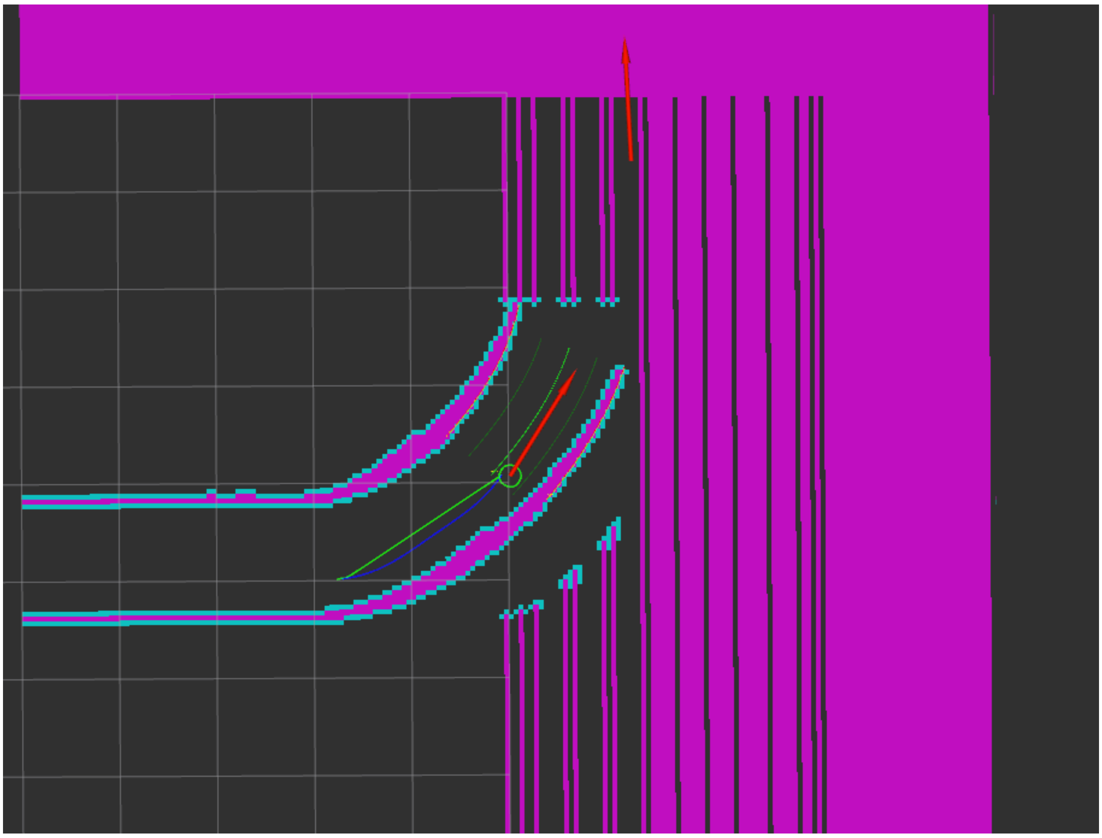


Figure 5.13.: global planner border error

To prevent this behavior the not documented parameter "outline_map" has to be set to false. The default value of this parameter is therefore changed in the forked version of the navigation stack.
To make planning easier the parameter "cost_factor" can be reduced. This parameter multiplies the cost of every cell in the costmap before planning which would make the gradually decreasing cost of the dynamic_cost_layer redundant.

5.7.2. teb_local_planner

Like the global planner the local planner has to be configured to comply to the tasks the local planner has.
A good starting point for the configuration are the example configurations in the repository of the developer of the planner[20]. They offer base configurations for both differntial drive and car like, aswell as omnidirectional robots.

Unfortunately teb_local_planner only considers lethal cost and without any configuration would follow the global path very loosely resulting in e.g. cutting corners.

To give the planner a tendency to follow the global planner closer the option "viapoint" can be used. This allows to set points at a configurable distance to each other on the global path, that attract the local path and therefore pull the local path and the robot to the correct lane. The attraction of the local path through the via points is then tuned so the robot drives roughly in the middle of the right lane, when no obstacle is on the road, but still can separate itself from the global path, when avoiding obstacles.

The parameter “max_global_plan_lookahead_dist” controls how much of the global plan is actually considered by the local planner. This parameter highly influences the computational load when setting the distance to larger values. Using shorted distances results in oscillations while driving straight, which is caused by the jumping global path.

6. Results and Discussion

7. Conclusion

7.1. Personal conclusion

8. Outlook

9. List of Figures

10. List of Tables

11. References

[1] ROS contributors. *ROS Introduction*. http://wiki.ros.org/ROS/Introduction. Online; Accessed: 2021-03-12. 2021.

[2] ROS contributors. *ROS Concepts*. http://wiki.ros.org/ROS/Concepts. Online; Accessed: 2021-03-12. 2021.

[3] Mike Purvis Tully Foote. *Standard Units of Measure and Coordinate Conventions*. https://www.ros.org/reps/rep-0103.html. Accessed: 2021-02-20. Oct. 7, 2010.

[4] Wim Meeussen. *Coordinate Frames for Mobile Platforms*. https://www.ros.org/reps/rep-0105.html. Accessed: 2021-02-20. Oct. 27, 2010.

[5] ROS contributors. *Navigation RobotSetup*. http://wiki.ros.org/navigation/Tutorials/RobotSetup. Online; Accessed: 2021-03-12. 2021.

[6] ROS contributors. *costmap_2d*. http://wiki.ros.org/costmap_2d. Online; Accessed: 2021-03-12. 2021.

[7] Wolfgang Hess et al. "Real-Time Loop Closure in 2D LIDAR SLAM". In: *2016 IEEE International Conference on Robotics and Automation (ICRA)*. 2016, pp. 1271–1278.

[8] M. Santos Pessoa de Melo et al. "Analysis and Comparison of Robotics 3D Simulators". In: *2019 21st Symposium on Virtual and Augmented Reality (SVR)*. 2019, pp. 242–251. DOI: 10.1109/SVR.2019.00049.

[9] ROS contributors. *ROS Index*. https://index.ros.org/packages/#noetic. Online, Accessed: 2021-02-26. Feb. 26, 2021.

[10] Lenka Pitonakova et al. "Feature and performance comparison of the V-REP, Gazebo and ARGoS robot simulators". In: (2018).

[11] *nav_core documentation*. http://wiki.ros.org/nav_core?distro=noetic. Accessed: 2021-03-01.

[12] ROS contributors. *carrot planner*. http://wiki.ros.org/carrot_planner. Online; Accessed: 2021-03-04. 2021.

[13] Przemyslaw Klapa and Bartosz Mitka. "Edge effect and its impact upon the accuracy of 2D and 3D modelling using laser scanning". In: 1 (Jan. 2017), pp. 25–33. DOI: 10.15576/GLL/2017.1.25.

[14] ROS Contributors. *laser_filters*. http://wiki.ros.org/laser_filters. Online; Accessed: 2021-03-17. 2021.

[15] Tom Moore. *robot_localization*. http://docs.ros.org/en/melodic/api/robot_localization/html/configuring_robot_localization.html. Online; Accessed: 2021-03-15. 2016.

[16] Google Cartographer contributors. *SLAM tuning*. https://google-cartographer-ros.readthedocs.io/en/latest/tuning.html. Online; Accessed: 2021-03-09. 2020.

[17] Kang Zhang Leen Ammeraal. *Computer Graphics for Java Programmers*. ger. 3rd ed. Springer, Cham. ISBN: 9783319633565. URL: https://doi.org/10.1007/978-3-319-63357-2.

[18] Kai-Uwe Sattler and Gunter Saake. *Algorithmen und Datenstrukturen*. ger. 6th ed. dpunkt.verlag. ISBN: 9783969100660. URL: https://content-select.com/de/portal/media/view/5f4a4a42-330c-4397-9629-0695b0dd2d03.

[19] ROS contributors. *global_planner*. http://wiki.ros.org/global_planner. Online; Accessed: 2021-03-10. 2021.

[20] Cristoph Rösmann and contributors. *teb_local_planner_tutorials*. https://github.com/rst-tu-dortmund/teb_local_planner_tutorials. Online; Accessed: 2021-03-11. 2021.

Appendix

A. Additional Topics

B. Source Code

Eidesstattliche Erklärung

Name: Schwörer **Vorname:** Tristan
Matrikel-Nr.: 71336 **Studiengang:** Mechatronik

Hiermit versichere ich, **Tristan Schwörer**, an Eides statt, dass ich die vorliegende Bachelorarbeit
an der **Hochschule Aalen**
mit dem Titel **„Entwicklung von Navigationssoftware für mobile Robotersysteme und Simulation“**
selbständig und ohne fremde Hilfe verfasst und keine anderen als die angegebenen Hilfsmittel benutzt habe. Die Stellen der Arbeit, die dem Wortlaut oder dem Sinne nach anderen Werken entnommen wurden, sind in jedem Fall unter Angabe der Quelle kenntlich gemacht.

Ich habe die Bedeutung der eidesstattlichen Versicherung und prüfungsrechtlichen Folgen (§23 Abs. 3 des allg. Teils der Bachelor-SPO der Hochschule Aalen) sowie die strafrechtlichen Folgen (siehe unten) einer unrichtigen oder unvollständigen eidesstattlichen Versicherung zur Kenntnis genommen.

Auszug aus dem Strafgesetzbuch (StGB)

§156 StGB Falsche Versicherung an Eides Statt Wer von einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

Ort, Datum Unterschrift