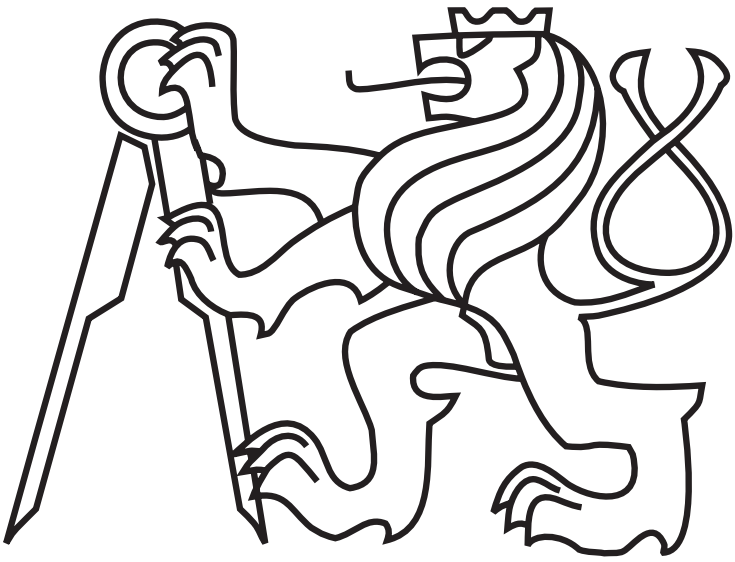
CZECH TECHNICAL UNIVERSITY IN PRAGUE

Faculty of Electrical Engineering

BACHELOR THESIS



David Otgonsuren Rico

Robust Robot Path Planning in Known Map

Department of Cybernetics

Thesis supervisor: **Ing. Tomáš Rouček**

May, 2021

BACHELOR'S THESIS ASSIGNMENT

I. Personal and study details

Student's name: **Otgonsuren Rico David** Personal ID number: **481890**

Faculty / Institute: **Faculty of Electrical Engineering**

Department / Institute: **Department of Cybernetics**

Study program: **Open Informatics**

Specialisation: **Artificial Intelligence and Computer Science**

II. Bachelor's thesis details

Bachelor's thesis title in English:

Robust Robot Path Planning in Known Map

Bachelor's thesis title in Czech:

Robustní plánování ve známé mapě pro robotickou platformu

Guidelines:

Goal of a work is to design and deploy a system for a robot path planning in a given map.
1) Research methods for path and action planning in a known topological map.
2) Get to know the Husky and Phoenix robots and their kinodynamic properties and the software framework for its control and autonomy.
3) Implement a method for automated creation, maintenance and refinement of topological maps.
4) Implement path and action planning method based on a topological map.
5) Integrate the method into ROS and deploy it to a simulated and real robot.
6) Evaluate the performance of your system compared to traditional ROS navigation stack.

Bibliography / sources:

[1] Koenig, Sven, and Maxim Likhachev. "D^* lite." Aaai/iaai 15 (2002)
[2] T.Krajnik, F.Majer, L.Halodova, T.Vintr: Navigation without localisation: reliable teach and repeat based on the convergence theorem. 2018 IROS
[3] Tutorials for systém ROS, http://ros.org.
[4] Yu, J. and LaValle, S., 2013, June. Structure and intractability of optimal multi-robot path planning on graphs. In Proceedings of the AAAI Conference on Artificial Intelligence (Vol. 27, No. 1)
[5] Tutorials for ROS simulator Gazeebo. http://gazebosim.org/tutorials

Name and workplace of bachelor's thesis supervisor:

Ing. Tomáš Rouček, Department of Computer Science, FEE

Name and workplace of second bachelor's thesis supervisor or consultant:

Date of bachelor's thesis assignment: **04.02.2021** Deadline for bachelor thesis submission: **21.05.2021**

Assignment valid until: **30.09.2022**

Ing. Tomáš Rouček	prof. Ing. Tomáš Svoboda, Ph.D.	prof. Mgr. Petr Páta, Ph.D.
Supervisor's signature	Head of department's signature	Dean's signature

III. Assignment receipt

The student acknowledges that the bachelor's thesis is an individual work. The student must produce his thesis without the assistance of others, with the exception of provided consultations. Within the bachelor's thesis, the author must state the names of consultants and include a list of references.

. _____
Date of assignment receipt

_____
Student's signature

Author statement for undergraduate thesis:

I declare that the presented work was developed independently and that I have listed all sources of information used within it in accordance with the methodical instructions for observing the ethical principles in the preparation of university theses.

Prague, date..... .....

Signature

Acknowledgements

Thanks to my family and supervisors for the support. Thanks to Škoda auto a.s. and Lipraco s.r.o. for giving access to work on the Phoenix robot.

Abstract

The goal of this Bachelor thesis is to create a system that can navigate a robot through a known map by camera image and odometry.
Given a goal, the system finds a position closest to an edge on a graph that represents the map. The graph is then modified by adding edges and nodes to accommodate the final and current robot position. The shortest path to given goal is found by the A* algorithm. Robot navigates through the given path using a system that turns the robot and the BearNav navigation system, this system ensures that corrections are made to the traversal of a map by comparing features of camera image previously recorded and currently seen.
After mapping the individual parts of the map with BearNav and creating a file with information about the nodes laying on the map, the robot should be capable of traveling on the map. Experiments were made both in a simulation and real-world. The system was launched on a real robot for the final deployment.

Abstract

Cílem této bakalářské práce je vytvořit systém který dokáže navigovat robota po známé mapě pomocí obrazu kamery a odometrie.
Je-li dán cíl, systém k němu najde nejbližší hranu grafu, který reprezentuje mapu. Graf je poté upraven přidáním hran a vrcholů, které reprezentují cílovou a počáteční pozici robota. Nejkratší cesta od robota k danému cíli je nalezena algoritmem A*. Robot prochází danou cestu pomocí systému na otáčení robota a navigačního systému BearNav, tento systém zajišťuje, že budou provedeny korekce při procházení mapy, pomocí porovnávání významných bodů obrazu které byly nahrány a které byly současně objeveny.
Po manuálním nahráním jednotlivých částí mapy systémem BearNav a poskytnutí informací o jednotlivých uzlech které leží na nahraných hranách, je robot schopen cestovat po mapě. Experimenty byly provedeny jak v simulaci, tak v reálném prostředí. Systém byl nasazen na reálném robotu do produkce.

Contents

List of Figures

1 Introduction

This work implements navigation of a robot in a known map using BearNav. The implementation is then tested both in simulation and real-world scenarios. The only necessary sensor that we have to use is odometry. If we were to rely on odometry only for long-term navigation without any calibration, the error of odometry would continuously build-up, until the robot's travel would not correspond with the path it should take. For a more long-term navigation the odometry needs to be corrected, and in this work it will be done by BearNav that uses a camera image to correct the odometry. In the final implementation more sensors will be use to help the localization of the robot, such as GPS or laser sensors.

The main purpose of the system is to autonomously navigate a robot that carries cars from the production line of a factory to a parking space buffer. The robot created for this task by Lipraco, s.r.o. was designed for the needs of Škoda Auto a.s. Škoda company wants to deploy the system for their production line in Mladá Boleslav.

With BearNav, we can avoid the problem that is created by the drifting of the odometry because it corrects its trajectory based on the current path and a path that it was trained on by comparing features of camera images. There are turning maneuvers that do not consider any corrections in their process. They may increase the error slightly, but BearNav should then visually correct the error created. Thus we should have a reliable system of navigation.

For the navigation the global map is going to be represented as a graph with nodes. To find the optimal path between a starting node and a final node, the A^* algorithm is going to be used due to its completeness, optimality, and efficiency. To prioritize which nodes it should expand while searching, it needs a cost and heuristic function, and in our case, we are going to use Euclidean distance for both.

To test this, we will need to create a non-trivial map where we test multiple subsequent paths. We can check the correctness of the hypothesis by visually comparing the representation of the map and the path that the robot is taking. We can not do this by comparing exact values because the robot will be off the predicted path by some margin, both in real-world or simulation, due to outside effects, noise from sensors, etc.

With a functional navigational system we can compare it to the ROS navigation stack from the ROS library that also implements a 2D navigation. Than decide in which circumstances one or the other would be preferred.

2 State of the art

This section describes various concepts and systems relevant to this thesis. From the principle of robots and motion planning to systems providing interfaces to make communication and implementation easier.

2.1 Robots and motion planning

A robot is a mechanical structure that is capable of interacting with its environment. Robot gains information about the environment from its sensors [5], which range from touch, light, acceleration sensors and more. It then interacts with its environment through a wheeled platform, robotic arm or any other construction. It is controlled by a control system that can be a PC or various logic circuits and micro controllers.
One such implementation is a mobile robot that can move through the environment and not be tied to one physical location, equipped with a variety of possible sensors that in some way represent the space around the robot. These robots can be used for many specific applications which require systems like collision avoidance [6] or localization [7]. Systems that a robot can use to move range from wheeled robots with a diffferential drive, ackermann steering and omni wheeled robots to robots that walk on mechanical legs [8]. Each of these systems comes with its own set of advantages and disadvantages.

A robot with a differential drive [9] has the ability to rotate around the central point of its axis, this is because the wheels on each side have their own motor and by rotating the wheels on each side in different directions and same speed the robot turns around its central point. Differential wheeled robots are used quite widely in the field of robotics and that is because of their simplicity they are easy to program. This system of steering will require that the motors move at exactly the same rate when going on a straight line, otherwise the robot will be turning to one side. An omni wheeled robot [10] can move in all directions of a 2D space, meaning it can move sideways unlike other steering options. This is possible, because of the construction of the mecanum wheels [1] which allow the robot to slide laterally. The wheel has rollers on its circumference which are perpendicular to the direction of the wheel. These wheels will also require additional motors to function properly, common case is two motors per wheel.

Another implementation might be an ackermann steer[11] for a wheeled platform. It is an arrangement of linkages of a car or another vehicle designed to avoid slipping of tires when going along a curved path. The geometrical solution to this is for all wheels to have their axles arranged as radii of circles with a common centre point. Before this each wheel would have their own centre point resulting in unnecessary forces and pressure to the linkage.

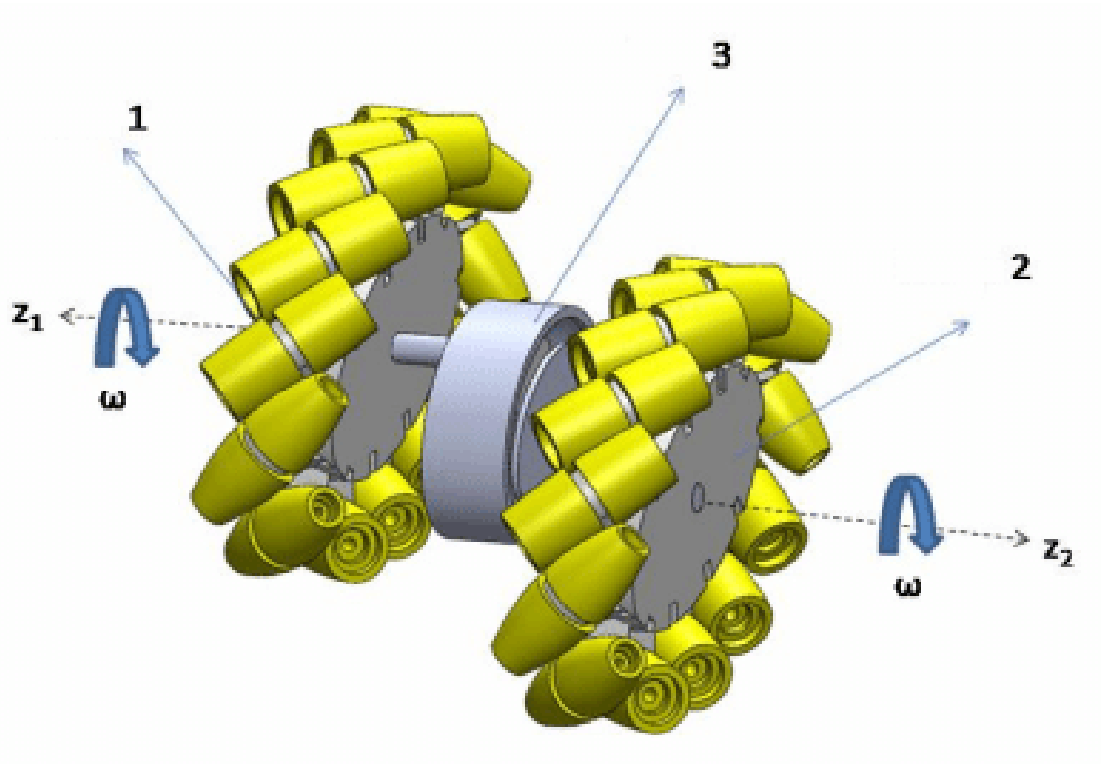


Figure 1: Design of a mecanum wheel [1], vectors represented by numbers show the degrees of freedom, ω and the z vector represent the direction in which the wheel can turn

2.1.1 Odometry

When performing any movement with a robot, information on where such a robot is located and how it is oriented is often helpful. That information might also be relevant not only about the robot but also about any other object in the environment. Odometry [12] provides one such representation. It gives position and orientation in space. Often used in robotics. It estimates the robot's position from its motion sensors relative to velocity measurements over time. One motion sensor commonly used in wheeled robots is a quadrature encoder, an incremental rotary encoder. A rotary encoder is a device that converts an angle of a shaft or an axle to an analog or digital output. If the rotary encoder knows how much the wheels have turned and their circumference, it can estimate its position. The quadrature encoder has two outputs from which it can determine the direction of shaft rotation [13]. Odometry is prone to the build-up of an error over time. During the movement, wheels can slip or slide, diverging the platform from its expected position. Non-smooth surfaces only augment this problem, and a motion sensor like a rotary encoder can not account for that. Accurate measurements and calibration are often needed for odometry to be effective in long-term use or when in difficult terrain. Another method that can be used would incorporate data from other sensors or systems to reliably estimate the position, and these systems could be GPS or a *LiDAR* localization. A robot's position in a free

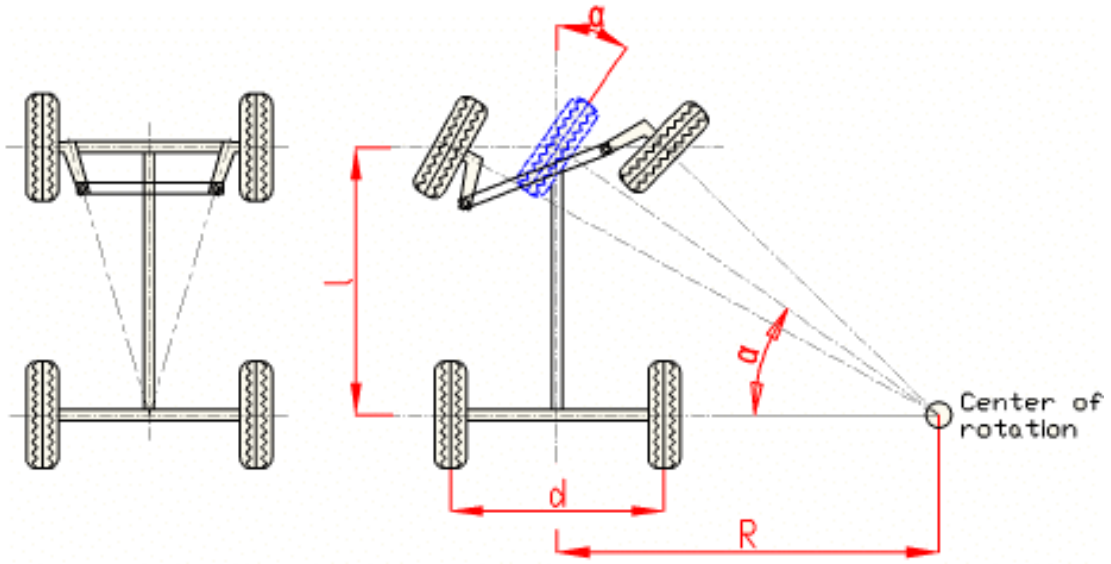


Figure 2: Principle of the ackermann steering [2], d represents the width of the vehicle, R represents the distance between the linkage and the center point, α is the angle between the rear part of the vehicle and the center of the front part

space allows the robot to be used in more complex problems such as path planning. When positions are taken in space and connected to other positions when they can be navigated to them, these points can be represented as nodes and traversals as edges, forming a graph. Forming the basis for a grid-based search of a path for a mobile robot [14]. Many path planning algorithms find the shortest route from the starting configuration into the goal configuration.

2.2 Shortest path graph based search algorithms

The goal of every shortest path algorithm that searches in a graph is to find a valid path such that the sum of the cost of edges traversed is minimal. There are various types of graphs and according to the type different search algorithms are more suitable.

An unweighted graph [15] is a graph where every edge has an uniform cost of traversal. In such a case we can use breadth-first search(BFS) algorithm [16] which takes the starting node as the root node. Using a queue which is a first-in-first-out (FIFO) data structure it stores and removes the nodes. It goes through every successor of the node from the start of the queue and checks if the node is the goal state otherwise they are put at the end of the queue. This process repeats until the goal node is reached. Disadvantage of the BFS algorithm is the exponential memory complexity compared to the depth-first search(DFS) algorithm [17] which has a linear space complexity, but unlike BFS it does not guarantee to find a shortest path but instead it returns any path that leads to the goal state when found.

2.2.1 Undirected graph

An undirected graph [15] is a graph where every edge associated with two vertices can be traversed in any direction. Edges in a directed graph have a starting and endpoint associated with them, specifying a direction of traversal. One commonly used algorithm is Dijkstra's algorithm [18]. It has many variants, but the most common one finds the shortest path from the starting node to all the other nodes. Dijkstra's algorithm uses a minimum priority queue where the nodes are stored and sorted based on the distance traveled. It also contains a set that holds nodes that have not been visited. The algorithm checks unvisited successors of the current node. It then checks their distances and computes a new one based on the current node, these two are compared, and a lower value is set. The current node is marked as visited. It does not return to a previously marked node. This process repeats until all nodes are visited. Since we are often only interested in finding the shortest path from the starting position to one goal state and not every other node, we can use an extension of Dijkstra's algorithm known as the A* algorithm. It uses heuristics in order to speed up its search of the goal state [19] [20]. Such heuristics for two points p and q of dimension n can be an Euclidean distance:

$$d(p,q) = \sqrt{\sum_{i=1}^{n} (q_i - p_i)^2} \tag{1}$$

or a Manhattan distance:

$$d(p,q) = \sum_{i=1}^{n} |q_i - p_i| \tag{2}$$

The heuristics have to be admissible. For a heuristic to be admissible, it can not predict a cost that would overestimate the real cost to the goal. For example, when in a 2D space where we can navigate freely, there does not exist a path that would cost less than a straight line from the node to the goal node. The line represents the Euclidean distance. A* is often used in the field of computer science due to its optimality and completeness. The cost of a node is computed as $f(n) = g(n) + h(n)$. Where $g(n)$ is the cost of traversing from the starting node to the current one and $h(n)$ is the heuristic function that estimates the cost from the current node to the goal node. Unlike the traditional Dijkstra's algorithm, it does not find the shortest paths to every other node from the starting one, but it looks for a goal state. The algorithm has exponential complexity, and its time complexity is polynomial. The implementation of A* algorithm is described in more detail in the system part 2.

2.3 ROS

The Robot Operating System (ROS) [21] [22] is a large framework for writing robot software. It offers a collection of tools and libraries that simplify the task of creating complex and robust robot behavior across a wide variety of robotic platforms.

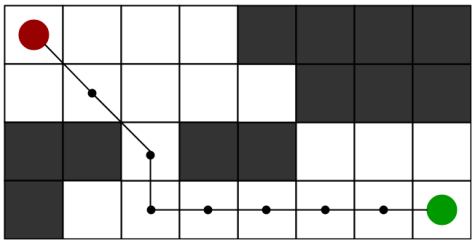


Figure 3: Path found with the A* alg in a grid based search

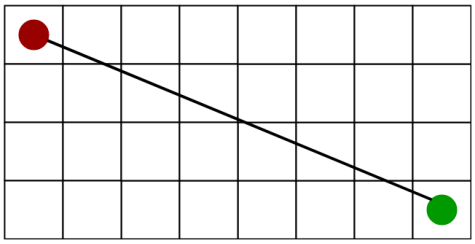


Figure 4: Euclidean distance as an admissible heuristic

One of its core features is message passing, where nodes communicate through the publish/subscribe system, which helps encapsulate individual tasks making them easy to reuse elsewhere. The standard message format defines formats for concepts like poses, transforms, vectors, odometry, paths, and maps.

One of the core features are action servers [23]. They allow for receiving a request from a node to perform a task and then returning a result message to the sender. The *actionlib* package provides initialization of the action server as well as providing a client-server interaction interface making communication with the server simple. During the execution of the task, the action server can publish feedback messages about the task's current status. The goal can also be preempted. All the actions such as goal, feedback, result are defined in the .action file, which can be configured.

RViz enables creating a simple GUI. In *RViz* can choose from a display list which types of object to visualize. Properties of these objects can be altered within the display list. The configuration created in *RViz* can be saved into a .rviz file. When launching *RViz*, this configuration file can be added as an argument, and it will load the saved configuration. Interactive markers [24] is a communication library for *RViz* and other tools. By creating objects and publishing their info on a marker topic, they can be visualized in *RViz*.

Roscpp package provides a client library that enables access to all the ROS topics, parameters, shared libraries, etc. All interactions with the ROS library can be written in C++because of this package, allowing to implement systems using both ROS and the C++standard library [25].
Rqt_reconfigure lets us configure parameters that are latched to nodes by *dynamic_reconfigure*. Providing an easy and simple interface to interact with. Nodes can be chosen from a list, and parameters that can be changed will appear based on the nodes selected.

2.3.1 Navigation stack

The ROS Navigation Stack [22] is a system meant for 2D maps, square or circular robots with a holonomic drive, and a planar laser scanner. A holonomic drive robot is a vehicle that has control in all its degrees of freedom. An omni-wheeled [10] robot is an example of

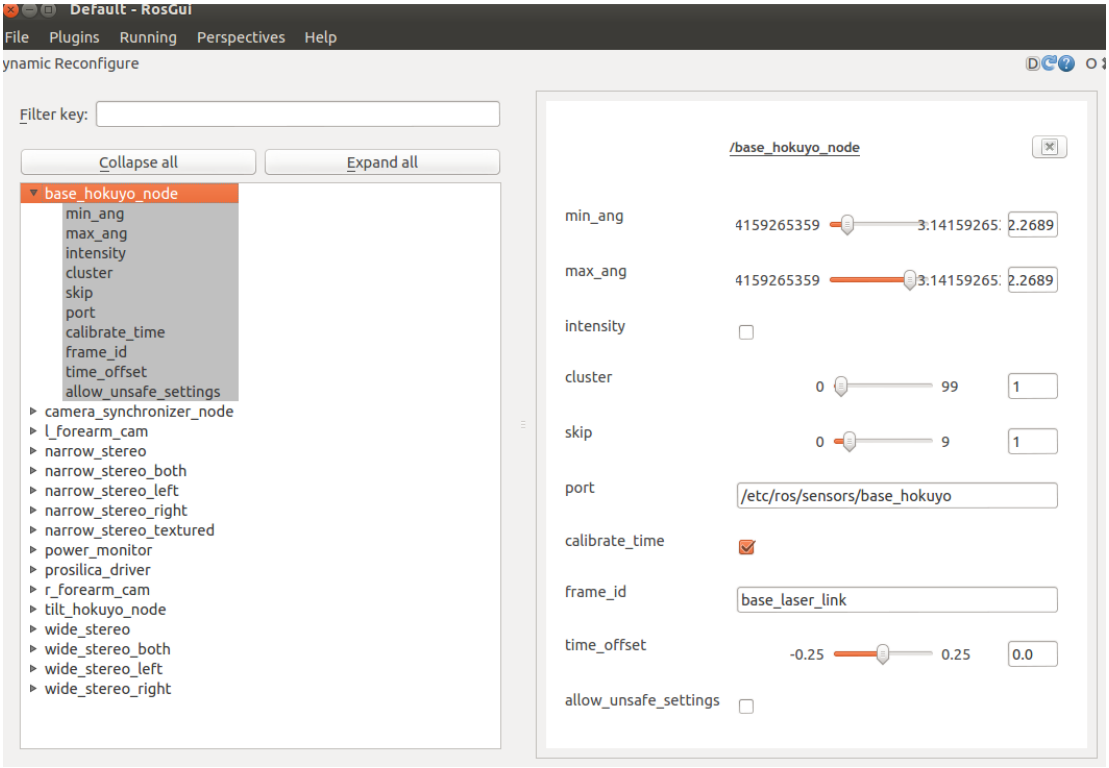


Figure 5: View of the *Rqt_reconfigure* window where nodes are on the left side and their parameters on the right

a holonomic robot because it has three degrees of freedom and can also move in all three degrees [26]. It implements navigation to a goal with a global and local planner. It uses odometry, sensor data, and a goal pose to give safe velocity commands. A pre-requisite to run the navigation stack on a robot is, the robot must be running ROS, have a *tf* transform tree in place [27], and publish sensor data using the correct ROS Message types. The *tf* transform tree is necessary because the localization is done with a laser scanner and it needs to take this data and use it to help the mobile base avoid obstacles in the world. To do this successfully, it needs a way of transforming the laser scan received from the laser frame to the "base_link" frame. Where the base_link represents the center of gravity of the robot. The navigation stack assumes that it can send velocity commands on the "cmd_vel" topic. This means there must be a node subscribing to the "cmd_vel" topic that converts them into motor commands to send to a mobile base. A package must be created that holds all the configuration and launch files for the navigation stack and this package will have necessary dependencies including the *move_base* package which provides a high-level interface with the navigation stack. It is implemented as an action server that waits for a goal that represents a point in the world, and tries to reach it.

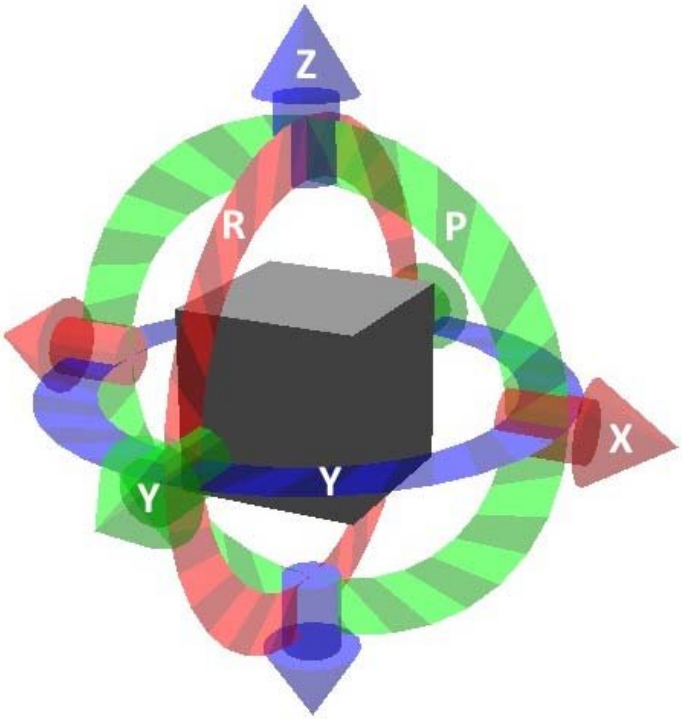


Figure 6: Visualization of an interactive marker [3] that can be rotated in roll, pitch, yaw angles and moved in the x, y, and z axis

2.4 Gazebo

Gazebo [28] simulator offers the ability to accurately and efficiently [29] simulate populations of robots in complex indoor and outdoor environments. One of its features is Simulation Description Format (SDF) [30] which describes objects and environments for robot simulators, visualization, control, and using this format, world and robot models can be created.
The SDF format is an XML format, and an SDF model has several components.
Link is the basic building block of a model, which is meant to represent a physical part of the model. One model may contain many links. Link contains the physical properties, such as collision, visual, inertial, sensor, light.

- Collision defines a geometric shape that is used for collision checking. It needs at least one geometry tag to define the object. Multiple can be used to define one link. Additional tag is the surface tag that specifies friction, contact, and their physical properties.
- Visual tag visualizes parts of the link. The same restrictions apply to visual as to collision. An additional tag is the material tag that specifies the texture and color of

the visualized element.

- Inertial element describes properties like mass and rotational inertia matrix.
- Sensor tag describes the type and properties of the sensor. Light tag define describes the light source. These are not necessary.

Joint is for a description of two kinematic relationships between links. A parent, child relation is always established between two links. Various types of joints specify the constraints of degrees of freedom between those links.

- Fixed joint connects two links rigidly since it does not have any degrees of freedom
- Continuous joint is a hinge joint that rotates on a single axis with a continuous range of motion
- Revolute is a hinge joint that rotates on a single axis with a fixed range of motion
- Revolute2 which is two revolute joints connected in series
- Ball joint, which is a ball and socket joint, has three degrees of freedom
- Universal, similar to a ball joint but constrains one degree of freedom
- Prismatic is a sliding joint that slides along an axis with a limited range specified by upper and lower limits

A plugin includes a specified library that is meant to control the model, sensor, and light. Moving the model can be handled by a drive plugin, camera plugin for a camera image, and so on. The library can be a custom library or any from the *gazebo_plugins* library of plugins. The plugin can be easily connected to ROS by latching it onto topics or nodes, making it invaluable when using ROS and Gazebo simultaneously.

2.5 BearNav

The BearNav [4] [31] navigation system is divided into two steps: teach and repeat. In the learning phase, the robot is guided by an operator, extracts features from its camera image, and stores its traveled distance and velocity. It sets its velocity to distance traveled during the repeat phase and corrects it by comparing current extracted features and features stored. The BearNav system is implemented as a ROS package that, when launched, starts multiple nodes.

- First node is the *odometry monitor* node that continuously records and publishes the distance traveled. It has parameters that can be changed in the *rqt_reconfigure* node, which specify at what distance it should periodically save or load the velocity information and features extracted.
- *Joy node* provides an interface from a generic joystick to ROS. Making it possible to maneuver the robot using the joystick.
- The *feature extraction* node is subscribed to the camera image, which extracts up to a maximum number of features to be reached if possible. The number of features that the node should try to reach is customizable.
- *Mapper* node saves the path profile like angular and linear velocity and features extracted into a .yaml file used in the learning phase when guided by an operator.
- *Map preprocessor* node loads a saved map and prepares it for use in the navigation phase.
- *Navigator* node autonomously navigates the robot through a path by comparing currently extracted features from the *feature extraction* node and features loaded from the *map preprocessor*. It adjusts the loaded values of linear and angular velocity based on this comparison.

Then some nodes are launched but are not part of the BearNav system. Action clients are used as simple GUI for communication with the *mapper*, *navigation* and *map preprocessor* node. *Rqt_image_view* is a helpful visualization tool that lets us see the camera image, features extracted on it, comparison of the features when navigating. Lastly *rqt_graph* node which is a node that provides the ROS computation graph visualization. We can see each node, what it is subscribed to and to which nodes it publishes, names of the topics.

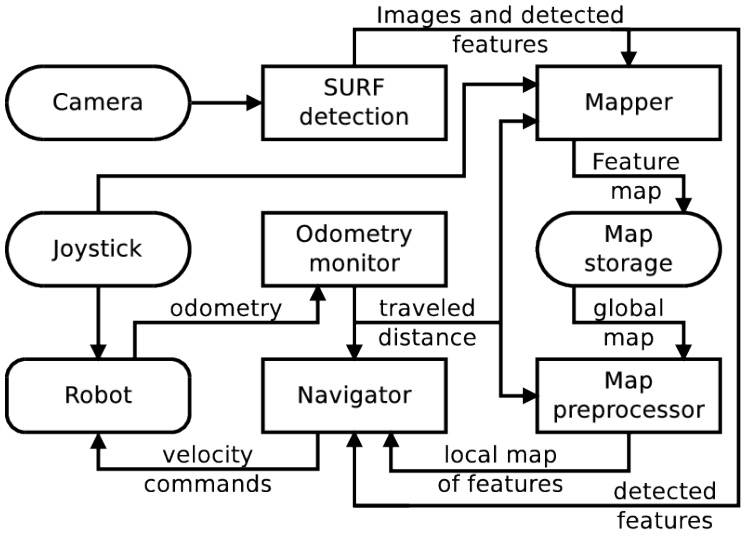


Figure 7: Overview of the communication between individual parts of the BearNav system [4]

3 System description

The system that takes care of finding the path and communicating with BearNav comprises five modules working together. All the system's main parts are implemented as action servers from the *actionlib* package. The distance module returns information about the two closest edges to a given point. The A* module configures a graph based on the given goal and robot's position. It then finds the shortest path in the modified graph and returns the path found. The turn robot module turns a robot to the desired heading. The navigator module goes through a path given by the A* module and calls an appropriate action to go from one node to another. The map module shows a graph, robot model in *RViz* and provides an interactive box that publishes its position as a goal to the navigator module displayed. See figure 13 for a representation of interactions and overview of the system. Lastly there is a description of the simulation in *Gazebo* and its integration and creation.

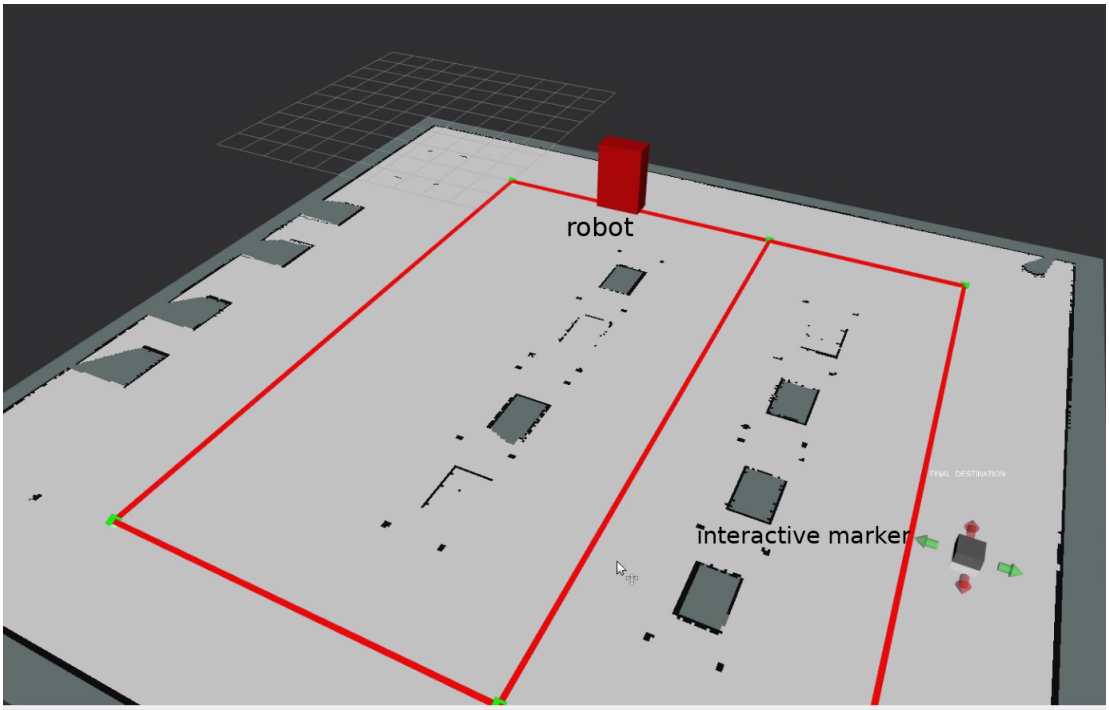


Figure 8: View of a robot navigating through a graph to a point designated by an interactive marker. Edges of the graph are represented as red lines and nodes as yellow boxes. Environment is meant to simulate a warehouse. View is from the *RViz* application.

3.1 Distance module

The distance module is an action server that loads a file containing information about the map segments. It waits until it is called with x, y parameters representing a point

and returns information about up to two closest segments to the given point. Previously it returned the closest segment and not up to two. Why it does so is described in the A* module in 3.2.

Each segment is represented as a set of two 2D points and their identification numbers that correspond to the node's identification. All the segments are loaded from a .txt file containing their information. In the form of segment id, x position of the first point, y position of the first point, and analogically for the second point. When all the info is loaded, the initialization of the distance action server is made. See 5 for an example.

(a) admissible (b) non-admissible

Figure 9: Example of an admissible and non-admissible segment represented by two points creating a line p, line n perpendicular to s and passing through the goal point. The segment is admissible when the intersection is between the two points

The distance action server then waits until it receives a goal. The goal is a 2D point. The nearest segment to the goal holds the closest distance to the goal from the line represented by the segment, and the intersection must also lie between the two points and on the line. The intersection is a point that lies on the line created by the segment and a line perpendicular to it and also passing through the goal node. When a goal is received, the algorithm goes through every segment and checks if the segment is a viable candidate and if the distance found to the goal is closer than that of the previous viable candidates. See figures in 9 to check which segments are viable, and which are not. If true, the result is updated accordingly. Algorithm in 1 shows the implementation in pseudocode.

Algorithm 1: get nearest segment(edge)

```
Data: goal, segments
Result: return segment closest to the goal point
shortest_len = MAX_FLOAT_VAL;
second_shortest_len = MAX_FLOAT_VAL;

foreach segment e of the segments do
    intersection_viable, len2line = lengths_and_intersection(goal, e);
    if intersection viable and len2line < shortest_len then
        //move current best option to second best res[1] = res[0];
        res[0] = x;
        shortest_len = len2line;
    else if intersection viable and len2line < second_shortest_len then
        res[1] = x;
        second_shortest_len = len2line;
end

return result;
```

The function that returns if the intersection lies on the segment line and between the two points that define it as well as the closest distance to the point relies on a geometrical solution.
There are lines defined as

$$s : y = p \cdot x + q \tag{3}$$

$$n : y = -\frac{1}{p} \cdot x + c \tag{4}$$

where s is a line that passes through the two points in the segment, n is a line perpendicular to s and passing through the goal. The slope of n is a negative reciprocal of the slope of s. For the lines to be perpendicular in 2D space a simple rule will help. When we want to get a perpendicular vector to a vector (a, b) we simply exchange the two values and multiply one by -1. Getting $(b, -a)$. When we multiply the line n by p, we get

$$p \cdot y = -x + p \cdot c \tag{5}$$

and by comparing the slopes of the x, y values of with line s 3, we see we did exactly the same.
First we want to get values p, q that define s 3 from points in the segment represented as $(x_1, y_1), (x_2, y_2)$. We put the values of each point into the line definition getting two equations with two unknowns, which can be solved as such:

$$y_1 = p \cdot x_1 + q \tag{6}$$

$$y_2 = p \cdot x_2 + q \tag{7}$$

We can now subtract the equation 7 from the equation 6 leaving only one variable in the resulting equation

$$y_1 - y_2 = p \cdot (x_1 - x_2) \tag{8}$$

By dividing the right side of the equation 8 by $(x_1 - x_2)$ the parameter p is defined

$$p = \frac{y_1 - y_2}{x_1 - x_2} \tag{9}$$

Parameter q can now be defined from equation 6, because parameter p is already obtained in 9

$$q = y_1 - p \cdot x_1 \tag{10}$$

To get the parameter c in line n from the goal point and parameters of s. By inserting the values of goal into n and using newly found parameter p we can express c from line n 4 as:

$$c = -x_{goal} - p \cdot y_{goal} \tag{11}$$

Then with the defined lines we can find the their intersection. So we want both definitions of the lines 3 and 4 to be true for the same x and y. Again we have two linear equations with two unknowns. This is one of the ways it can be solved:

$$y = p \cdot x + q \tag{12}$$

$$y = -\frac{1}{p} \cdot x + c \tag{13}$$

Multiply equation 13 with p

$$-x - p \cdot y + p \cdot c = 0 \tag{14}$$

Substitute y in equation 14 with y in equation 12

$$-x - p^2 \cdot x - p \cdot q + p \cdot c = 0 \tag{15}$$

In equation 15 there is only one variable, lets define the variable x from the rest of parameters in equation 15

$$x = \frac{p \cdot (c - q)}{1 - p^2} \tag{16}$$

equation 12 now contains one variable when x is defined as in equation 16

$$y = p \cdot x + q \tag{17}$$

Having the point that lies on both lines we can check if it is out of bounds for the segment. If it is we return false for the viable option, if not then we calculate the euclidean distance

between the goal and intersection. Euclidean distance of two points which is defined as

$$d = \sqrt{(x_1 - x_2)^2 + (y_1 - y_2)^2} \tag{18}$$

In this case we return true for the viable option and the shortest distance from segment to goal.

3.2 A* module

A* module is an action server that loads a file containing information about nodes on the map, such as their distance on a BearNav map, global position, and their successors, and what type of navigation to use to get to them. It is subscribed to the odometry topic, which publishes the robot's position and saves the received position because it will be necessary for creating a starting node for the pathfinding algorithm. Action server waits until it is called with x, y parameters representing the goal point we want to get to. First, it calls the distance action server with the robot position and gets two segments that are closest to it. We choose the segment that corresponds with the robot's orientation, and we add the starting node to the graph. We call the distance action server again, but now with the goal point and we get to possibilities where to put our final node. Then the A* algorithm[19] is ran twice. Once for every possibility of the final node. This due to the fact that we don't know in which direction we will arrive, making both choices possible final destinations. If the distance action server returns only one possibility of the final node, then the A* algorithm runs only once. Finally, the two paths are compared, and the shortest one is returned.

When the A* node is launched, first, the information about the nodes is loaded from a .txt file. Nodes of the graph have a structure containing an identification number, x,y positions, map name, the distance of the node from the beginning of the map. Additional information about the nodes must be known, if a node is in a closed or open list, node's cost function and heuristic cost function value and id of the parent node, these are necessary for the A* algorithm to work. Lastly, information about the successors of a node must be known, including their id, type of navigation to be used, and optionally the map name if the navigation type is BearNav. After the information is loaded, the action server is initialized.
Action server waits for a goal which is a 2D point. When received action client for the distance server is initialized. Then the distance action server is then called through the client with parameters of robot position, and up to two edges closest to the robot are returned.

To add a node representing the robot's current position and an edge that connects to the graph, we need to calculate which of the endpoints the robot is more closely oriented to. The graph is an oriented graph because we cannot get from one node to another with one type of action and go back to the previous node with the same action, whether it is turning a robot 90 degrees or going through a mapped line with BearNav. We need their individual

angles to calculate the angle between the robot's orientation and the point relative to the robot's position. We get the final position α like so:

$$\alpha = |\varphi - \gamma| \tag{19}$$

φ in radians is the angle between the axis of the space and the robot heading, γ in radians

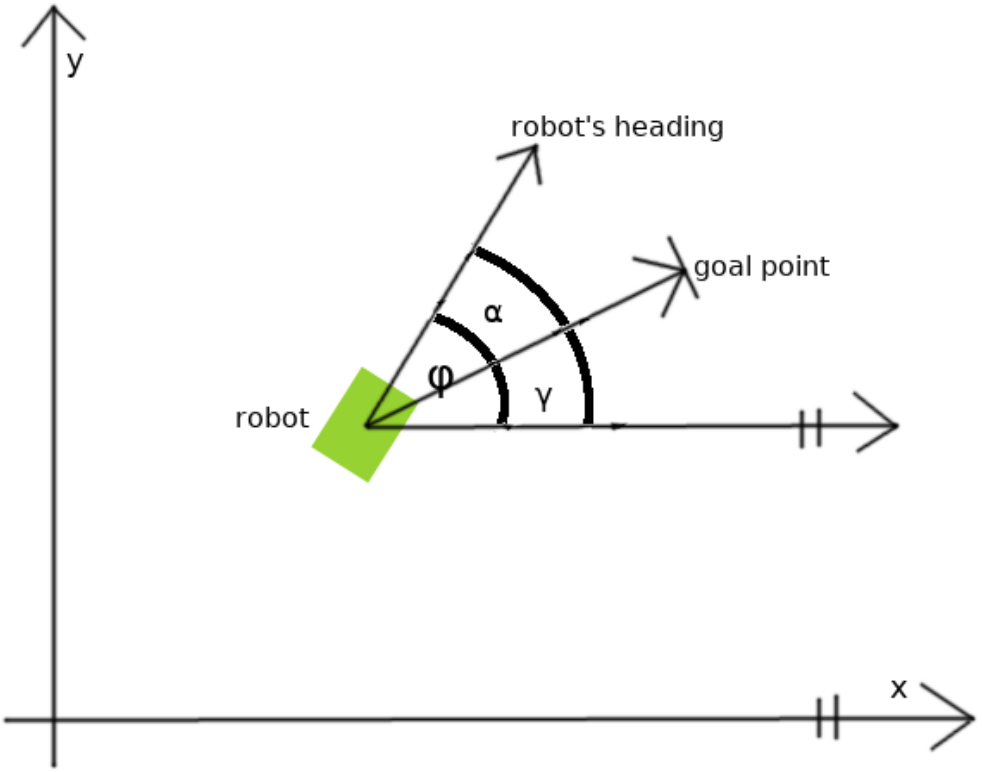


Figure 10: Example of the three different angles in space

is the angle arctan of the robot position and point. For a visualization of the angles see figure 10. To get φ, we need to get this angle from the robot position. Robot's orientation is defined in quaternions [32]. Quaternions are an extension of complex numbers. They are represented in the form $a + bi + cj + dk$ where a, b and c are real numbers and i, j, and k are basic quaternions. For the basic quaternions a requirement must apply that satisfies

$$i^2 = j^2 = k^2 = ijk = -1 \tag{20}$$

They are used in many applications to represent the rotation of an object. While Euler angles are easy to understand and interpret, they suffer from ambiguity. While quaternions might be more challenging to interpret, they do not suffer from ambiguity. To get our angle, we will use the ROS package *tf*, which is a package that helps keep track of coordinate frames and relations between them. This package provides a function that translates

quaternions into Euler angles [32]. Euler angles describe the orientation of an object in a 3D space. It consists of three angles roll, pitch, and yaw, each representing a rotation about a different axis. Yaw in the Euler angles corresponds to our φ. To get γ we will need to look at the robot position (x_2, y_2) and point (x_1, y_1) and create two lines of length:

$$x = x_1 - x_2 \tag{21}$$

$$y = y_1 - y_2 \tag{22}$$

These lengths represent two legs of a right triangle and where the distance between the two points is the hypotenuse.

$$\tan\gamma = \frac{y}{x} \tag{23}$$

By using the arctan function on both sides of equation 23, the left side becomes $\arctan\tan\gamma$ which becomes only γ and gets the result for γ

$$\gamma = \arctan\frac{y}{x} \tag{24}$$

We can now substitute φ and γ back into 19 and getting α. By doing this for both points, we can compare which of the angles is the smallest, and for that point, we will decide that the robot is oriented it's way. With this information, we can add the starting node and edge and connect them to the graph. The starting node represents the robot's position, and the edge represents a successor to the chosen point connecting it to the graph. The navigation type for the edge is BearNav, and the map corresponds to the map of the chosen point.

Then the distance action server is called again now with the parameters of the goal point. The distance server returns the two closest edges to the goal point. We will create an end node that is a successor to the starting point for each edge and has the coordinates of the intersection. We do not know beforehand which of these will lead to the shortest path, so we will consider both of these end nodes as goals for the A* algorithm.

3.2.1 A* algorithm

The A* algorithm is a well-known pathfinding algorithm. In our case, we use Euclidean distance as our heuristic and the successor cost function between the positions of the nodes. The algorithm tries to find a path with the smallest cost to the goal node, and it does so by maintaining a tree of paths and extending it from nodes with the lowest cost until the final goal node is reached. The algorithm is implemented in 2.

Algorithm 2: A* algorithm

```
Data: start, goal
Result: finds shortest path from start to goal node
list open_list;
list closed_list;
put start node in open_list;
while open_list is not empty do
    curr_node = take a node from the open list with the lowest cost;
    set costs f(n) = g(n) + h(n);
    if curr_node is goal then
        break;
    foreach successor e of curr_node do
        successor_current_cost=g(curr_node)+w(curr_node, e);
        if e is in open_list then
            if g(e) < successor_current_cost then
                continue;
        else if e is in closed_list then
            if g(e) < successor_current_cost then
                continue;
            move e into the open_list;
        else
            move e into the open_list;
            set the heuristic distance of e;
        end
        g(e) = successor_current_cost;
        set the parent of e as curr_node;
    end
    put curr_node into the closed_list;
end
```

Since there can be up to two possible goals, we run the A* for every modified graph by the possible goal. We compare the costs and choose the one with the lower cost. Since we have set the parents of the nodes when searching for the goal. We can backtrack from the goal and find our path. We add to the path all the necessary information about the types of navigation and names of the maps. This path is returned.

3.3 Turn robot module

Turn robot module is an action server. It is subscribed to odometry, and it gets the angle of rotation of the robot from the odometry message, which is then converted to degrees and saved. The action server waits until it is called with parameters of angles it should rotate and direction, clockwise or counter-clockwise. While the robot is not turned in the desired

position, it publishes to the topic that controls the robot's linear and angular speed so that it matches the rotation direction. Control of the robot's linear and angular speeds is handled by the ROS publisher to command velocity topic. The robot's driver is subscribed to this topic and moves the robot accordingly to the message. When finished, it stops the robot and returns that it succeeded.

The subscription to odometry continuously updates the angle in degrees of the robot. The robot's orientation is in quaternions, so we use the ROS *tf* package to convert the quaternion to radians [27]. Since the goal of the action server is in degrees, we have to convert the angle from radians into degrees because the angle is in radians from $-\pi$ to π. We use the conversion:

$$angle = \frac{angle \cdot 180}{\pi} \tag{25}$$

if the angle in equation 25 is lower than zero than we add to it the value 360.0 keeping the range in degrees from 0.0 to 360.0

$$bounded\_angle = angle + 360 \tag{26}$$

The action server, when called with a goal in the form of a boolean when true, means clockwise turning, else counter-clockwise. And an angle in degrees, which represents the number of degrees to turn. First, it takes the robot's current angle and based on the type of turning, it either adds or subtracts from the current angle giving us the desired angle. When adding or subtracting, our desired angle can go below 0 or above 360 degrees. If below zero, we add 360. Above 360, we subtract 360 to correct our desired angle. Since the angle is a floating-point number checking by equality if the current angle is equal to the desired angle would be unwise because, most likely, we would never get the exact value. We set up an interval around the desired angle which, when the current angle lies in it, we will consider as satisfactory. The robot turns in the desired direction until the current angle lies in the selected interval. When it does, the robot stops, and the result that the action server succeeded is returned.

3.4 Navigator module

The navigator module is an action server. It waits until it is called with parameters x, y representing the point to which it should navigate to. First, it calls the A* action server and waits for it to return the path. Then it goes through every node in the path and either calls the BearNav or turn robot action server depending on which node is associated with what type of navigation.

The navigator node, when launched, initializes a service client, which sets the distance on a BearNav map. This is necessary when going through multiple maps because if the distance were not reset, it would keep its final position from a previous map and take it as its starting position for the following map. This was one of the problems encountered

when implementing the navigation. Next, the action server is initialized. After the action server receives a goal as a 2D point, multiple action clients are created. First, the clients for BearNav are created. *Map preprocessor* client that loads a map given its name. *Navigator* client for BearNav, different from the current navigator, when given starting distance and final distance will go through the map previously loaded by the map processor. The A* client will return the shortest path from a given graph.

Finally, the turn robot client rotates a robot by an angle and direction. For every initialization of the client, we wait until their action server is created. We then call A* action server with our goal. We wait until the result is given. It includes all the necessary information to call BearNav or turning of the robot. The path is in order, so we go through every part of the result, check which type of navigation we need to use, call the correct action servers, and wait until they finish their action. In the end, the result is returned as a boolean set to true, representing the action successfully completed. When every action of the path is being performed, simultaneously the module publishes to a topic *distance_traveled* where the message shows the percentage of total distance traveled, current action that is being performed, from which node this action started, at which it is supposed to end at and a complete path that is being traversed.

3.5 Map module

Lastly, the map module provides a visualization of the graph in *RViz* and a simple GUI that enables the user to start the whole system. It loads the data about nodes and visualizes them in *RViz* with how they are connected as well a movable box that, when clicked, will send a goal message to the navigator action server as a desired destination for the robot.

The node also has to be launched with *RViz* and its setup. The setup has the name of the topics the map module will use, from markers representing the graph, the interactive box, and optionally the robot model if provided. It uses the ROS visualization msgs package to communicate with *RViz* and specify what shapes to use. First, we load the info about the nodes then we create an interactive marker server. Information about the graph is published once, and it appears in *RViz* where edges are red lines and nodes are green boxes. A few more functions need to be added to make an interactive box. First, creates the box, specifying its shape and color. Then two functions enable the movement of the box on the x and y axis through arrows that appear beside the box. Button control makes the box clickable and, finally, a function that receives feedback on what actions have been done on the box. If the action is a button click, it publishes its current position to the navigator action server, making it a simple GUI that launches the system.

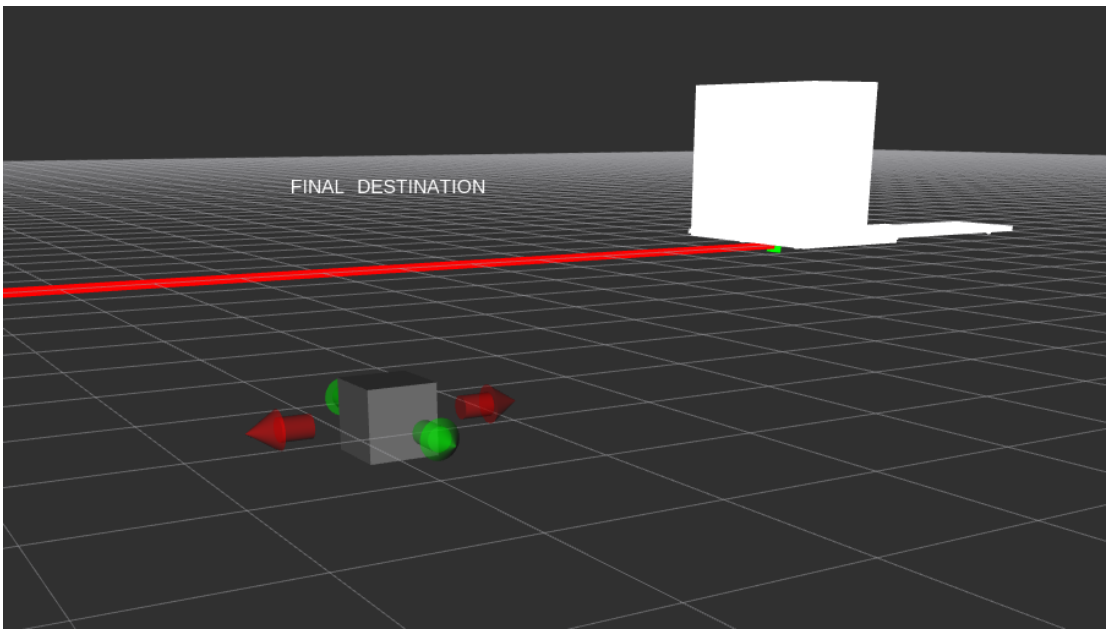


Figure 11: Interactive marker in *RViz* with the robot model

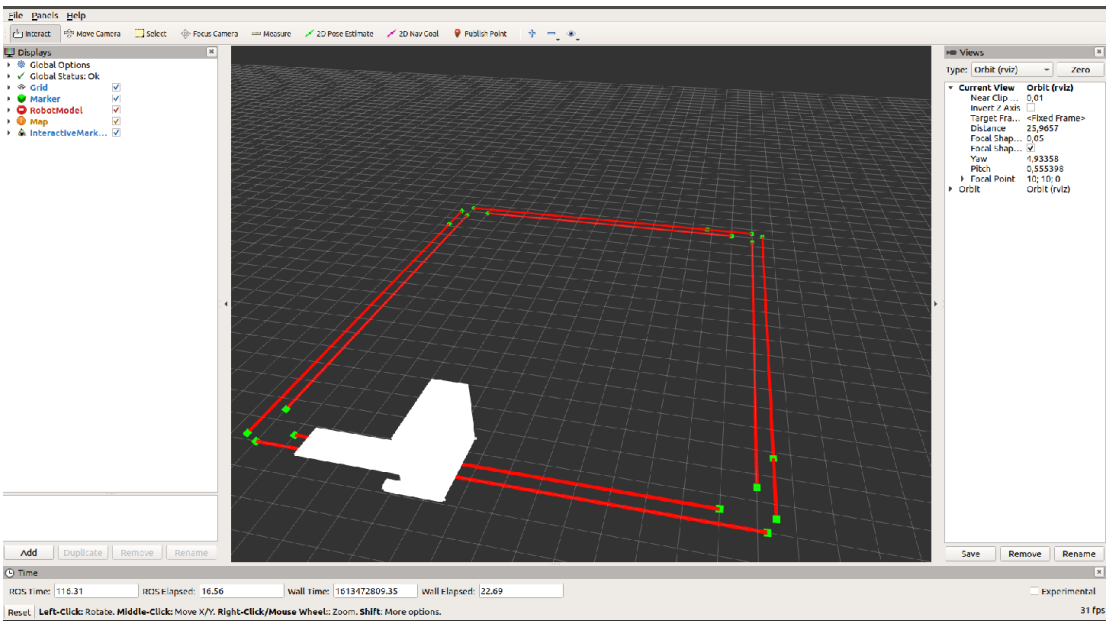


Figure 12: Top down view of segments and robot model in *RViz*

3.6 Gazebo

To test the created system we need an environment in which it can interact with, one such implementation can be a simulation created by the Gazebo simulator. All the files that will be mentioned in this section are .xml files unless said otherwise. First, when launching

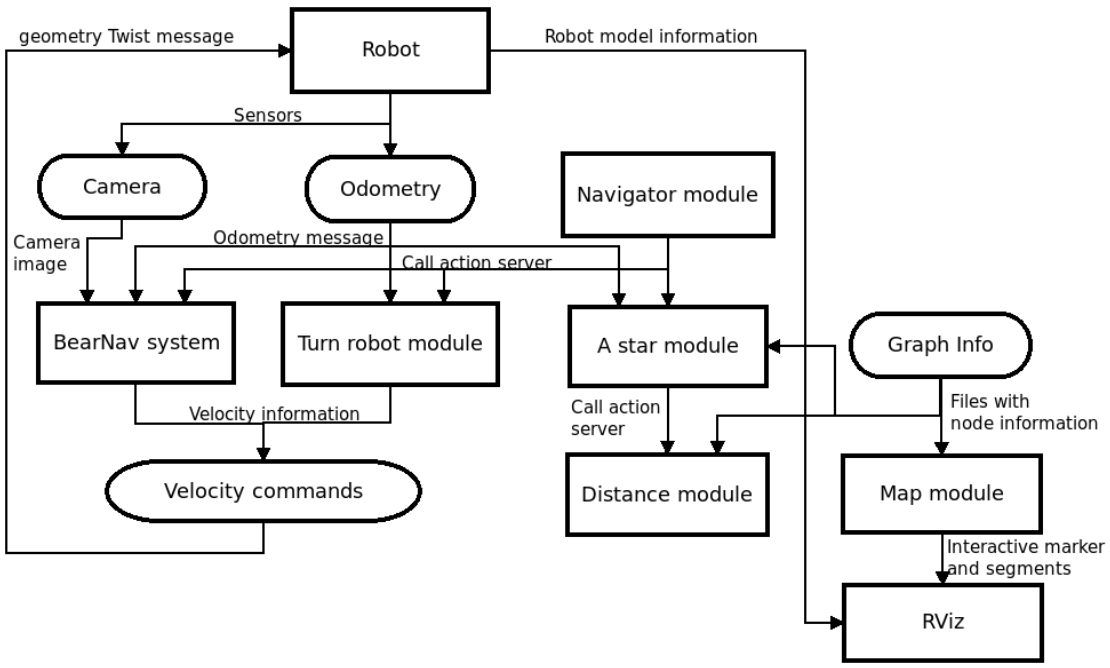


Figure 13: Diagram of the system parts and their interactions

Gazebo, we need a .world file. Without one, we would spawn a default world where we would only have a ground plane and a light source. The file is formatted by Simulator Description Format(SDF). Only a single model should be defined in the world file for simple reuse. In our world file, there is a ground plane and a light source. Additionally, there are links representing walls in the simulation. They have specified their geometric shape, the position of both visual and collision elements. Most importantly, the walls have URI and name tag that references to a script describing texture properties based on the name chosen. Various pictures are available that are meant to simulate the outside environment with different types of lighting. Gazebo has some textures available like a wooden palette, tiles, etc. Still, since these were not ideal for BearNav since it could not extract many features from such a simple texture, we chose high-resolution pictures instead of the simple textures. Additional simple geometric objects were placed within the walls to test the collision and functionality of various sensors.

Separate from the world file, other models can be put into the simulation during runtime. We separate the world file and the files where the robot is defined so that we can use and change other world files and the robot files independently, avoiding unnecessary copying. Another model that can be launched separately is supposed to represent an undercarriage of a car. The model consists of five links. First is a box that is the middle part, and four others are cylinders rotated and shaped to resemble wheels.

Lastly, the robot that can be put in the simulation is modeled after the Phoenix robot described in the 4.2. Three different files were used for the description of the robot. *Xacro* file contains the description of links and joints that were used. The front and the tail of

the robot are made up of boxes that are held together with fixed joints. The same goes for wheels. The camera plugin needs its own link, so one small box was put in the front center of the robot. Laser plugins also need their links, so two small boxes were put at the front bottom and rear bottom of the robot. The whole model is set as dynamic, which makes Gazebo ignore the physical elements of the model. This was done because trying to replicate real parameters did not make the robot behave accordingly in the simulation. And fine-tuning all the parameters from weight, friction, inertia, and others would be very time-consuming and possibly for naught, especially when these parameters were not provided by the robot manufacturer and the robot's construction changed several times during the duration of the project. Specs file assigns values to variables.

The .specs file is included in the *xacro* file so it can easily access parameters. Gazebo file includes information about plugins used in the *xacro* file. The drive plugin that controls the movement of the robot. This is a custom-written plugin that replicates ackermann steering see in 2.1. The camera plugin is taken from the gazebo library of plugins. It is a depth camera, see 4.2 for more detail. Many parameters can be used to configure the camera, such as color scheme, the field of view, frame rate, and more. The laser plugin is also taken from the gazebo library. Its parameters are what angle it should take samples from and at what distance, how many samples are meant to be taken from this angle. We need to describe this plugin for every link that will be used as a laser sensor since the plugin needs to know for which link it is specified. With these files, the robot is ready to be used in a Gazebo simulation.

4 Robot models

Description of robot models used in the simulation and real-world scenarios. The first section describes the Wild Thumper robot model that was used in real-world scenarios. The robot was used instead of the Husky robot because of the pandemic situation and availability. The following section describes the Phoenix robot model used in both simulation and real-world scenarios as well as the one meant for final deployment.

4.1 Wild Thumper

This platform was used due to access restrictions and was borrowed from the school for testing at home. The robot is equipped with six wheels. Each pair of wheels is controlled by a *Robo Claw 2x7A MotorController* [33]. It can control each wheel individually. When launched, it subscribes to the *cmd_vel* topic, where its messages represent the linear and angular velocity of the robot. Also, the node publishes to the *odom* topic where the message represents the robot's position by odometry. The motor controller has a USB interface that allows the node that controls the motor to be launched from a connected PC. Parameters can be set when launching the node, such as maximum speed of the wheels in meters per second or width from one wheel to another, which is also necessary to set correctly, so the odometry values are as accurate as possible. The controller does accept not only orders like acceleration, speed, and distance but also limiting the voltage and current of the motor.

On top of the robot's construction, a PC is mounted. *Intel NUC* was chosen due to its small size and range of voltage that it can be powered by. Range from 12V to 19V is ideal when powered by a battery that gradually decreases voltage as it is being used. The specifications of the *Intel NUC* are Intel Core i5-7260U processor, 8GB RAM, an 80GB SSD. Operating System is Ubuntu 18.04, which was chosen because it is free of charge and supports the *ROS melodic* distribution used in the system section. A monitor, keyboard, and mouse can be connected to the *NUC*, but this would not be practical during the movement of the robot but is useful when setting up the PC. It should be connected to a local Wi-Fi network. Every time the PC is turned on, it should automatically connect to the network if possible. When connected, we can access the PC on the robot with our own when connected to the same network. Using the Secure Shell Protocol (SSH), we can change files, launch nodes on the robot's PC from our own. The *NUC* is connected to the previously mentioned *Robo Claw Controllers* and additionally a camera and a *rplidar a3* sensor, which is a *LiDAR* sensor. *LiDAR* [34] is a device that determines a distance by targeting an object with a laser, and based on the time until the light is reflected back, the distance is estimated. *LiDAR* stands for laser imaging, detection, and ranging.

The camera is mounted at the front and top of the construction. It is a standard camera with a Full HD image. The camera is also protected from external effects by a metal construction covering it from the sides and top. The laser sensor has an angle of 360^o from which it can take samples from. It is placed on top of the *Intel NUC*, so no part of the robot

would interfere with the samples taken from the laser sensor. Then two batteries come with the robot. Both are lithium batteries which have good capacity relative to their weight. The battery with four cells and a nominal voltage of 14,8V, and a capacity 6750mAh is the one that powers the PC. The second battery with two cells has a nominal voltage of 7,4V and a capacity of 5200mAh and powers the motor controllers. A great part of testing was done on the Wild Thumper robot, since the simulation environment can not completely accommodate for every effect in a real-world scenario.

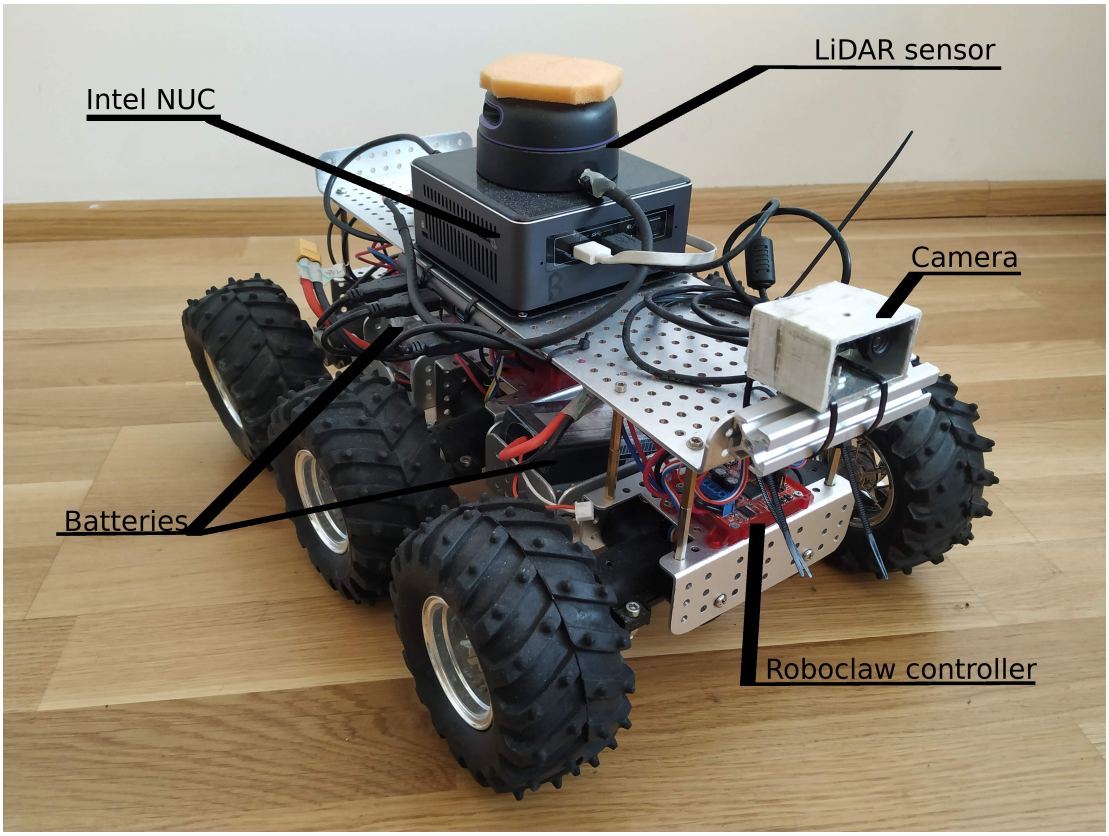


Figure 14: View of the Wild Thumper robot

4.2 Phoenix

Phoenix robot was constructed by the company Lipraco, Ltd. The robot is designed for autonomous navigation within a parking space where its goal is to transport cars from the production line of a factory to a parking place buffer. The robot was designed specifically for the needs of the Škoda Auto a.s. factory, where the aim is to test if automation of a task which is driving a newly constructed car from the factory plant onto a designated parking space in the vicinity.

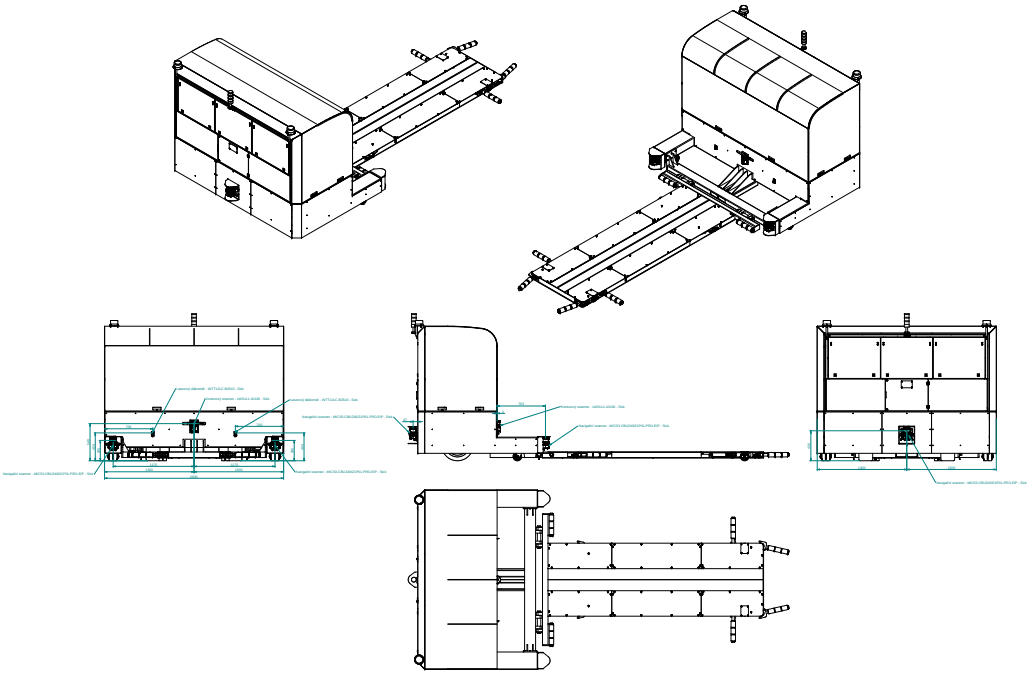


Figure 15: Diagram of the Phoenix robot

It used to have a lead-acid battery but it was changed to lithium battery because the lithium batteries weighed less. These batteries allow the robot to run for approximately eight hours straight without needing to be charged again. Phoenix robot has 5.02 meters in length, 2.5 meters in width, and 2 meters in height. Its current speed is 2 m/s aiming for 40 km/h in the final deployment. The robot consists of two main parts. The front part, which has about 1.2 meters in length, has the full height and width of the model. In the lower part of the front body, the motors are located, which control two large wheels situated in the middle of the front body. In the upper part is the battery of the robot as well as the controller. The robot has its own WiFi and PC that's connected to it. The PC is *Intel NUC* with Intel Core i7 generation 8, chosen for its small size and hardware capabilities. When connected to the WiFi with a personal computer, it is possible to connect to the PC on the robot through an SSH connection. From which we can give commands to the robot as well as receive information from its sensors.

At the bottom of the front body, we have two *LiDAR* sensors [34] at the rear. There is one *LiDAR* in front of the body in the lower part at the middle. Each sensor has a radius of 180^o. These sensors are used for obstacle avoidance, car pickup, and car detection. In the front center, we have a camera at the height of 1.1 meters. It is a Realsense D-435, which is an RGB-D camera [35], D stands for depth. This camera is mainly used for navigation with the BearNav system. By emitting a constant light(most usually infra-red), it can gather

from the image the distance to objects by deciding the volume of reflection from emitted light. This might be used for obstacle avoidance in the future. There is one additional camera attached on a hand from the rear of the main body. This camera is used to identify the car to be loaded or dropped off via a bar code behind the front screen of the car when the ramp is underneath the car. For compliance reasons, the robot has yellow lights, industrial semaphore on the top of the robot as well as a siren, which can activate, for example, in cases such as a low battery, motor malfunction, crash, etc. A kill switch in the shape of a big red button is at the front, and another is on the controller connected to the robot, which acts as a security measure when in unexpected or dangerous behavior.

The back part of the robot is a ramp on four small wheels that are not connected to any motor. The wheels at the front of the ramp have two degrees of freedom in which they can rotate and have approximately 20cm in diameter. Wheels that are in the back of the ramp are significantly smaller (10cm in diameter) than the rest and have only one degree of freedom in which they can rotate, which is the direction that the robot is facing. The ramp is very close to the ground because this part is meant to fit under a car when it is about to be picked up or dropped off. The ramp has four pairs of hydraulic levers. When the ramp is under a vehicle, the levers can move close to each other, and by applying force to each wheel of the car, the car is then lifted. To drop off a car, the levers simply go apart from each other until the car rests on the ground and the robot moves away with the ramp. Due to its size, its own weight, and the weight of the possible payload, the robot has relatively low maneuverability.

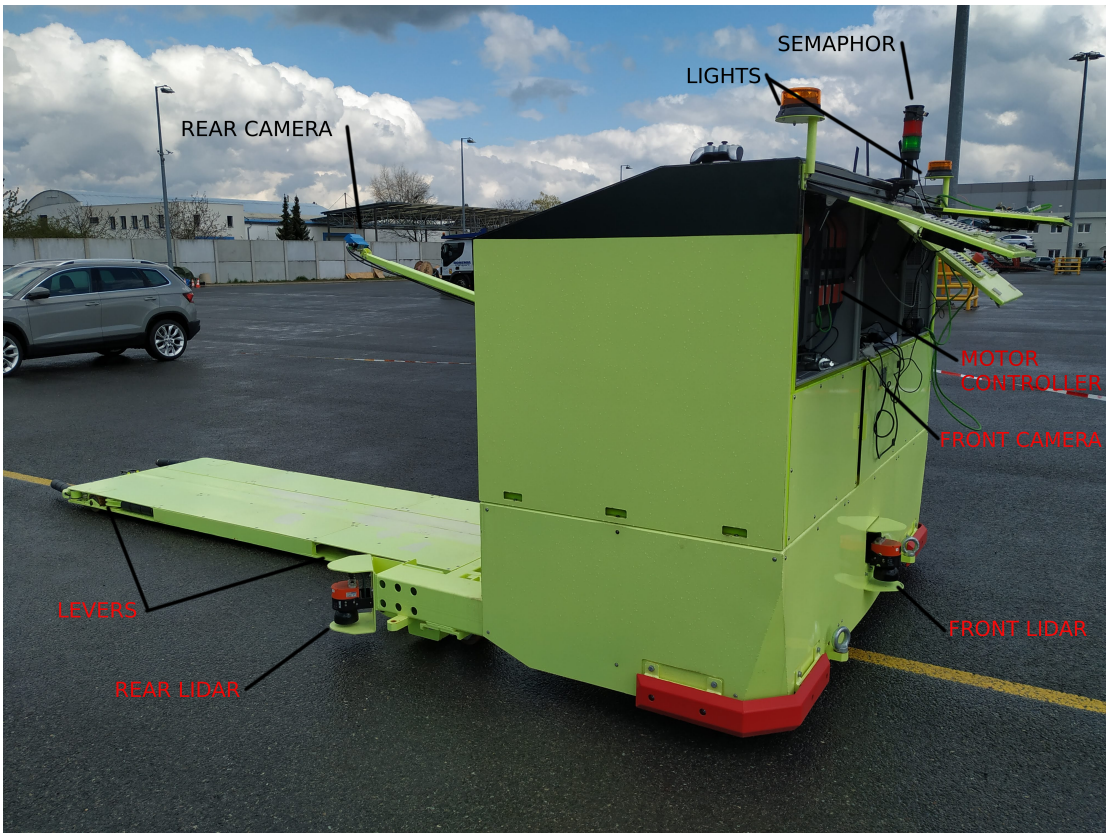


Figure 16: View of the Phoenix robot with sensors

5 Testing sites

This section provides a description of scenarios in which the system was tested and configured. The first scenario describes the environment in a simulation. The second describes a closed space, real world scenario. We can record any ROS topics that are being published and save them into a *rosbag*. This data can be used to replicate some scenarios or analyze them.

Whether they are in a simulation or real-world, testing sites share some information that has to be included in both alike. We need to teach the BearNav maps and save them so that they can be replicated in the repat phase. BearNav maps are saved in the *YAML* format [36]. Information in these map files contain features extracted at a certain distance, the velocity at a distance. When launching BearNav, a parameter can be set that specifies the path from which to load maps and where to save them. A file containing the information about nodes is also necessary to run the system that navigates the robot through the shortest path. This file is used to automatically create the representation of a graph. The information that has to be known is the global position, distance on the BearNav node, the id of successors, and what type of navigation to use to get to them. The information about the nodes of the graph is saved in a .txt file. The data structure for the file is as:

```
xPos yPos distanceFromStartOfMap numberOfSuccessors
nameOfMap
idSuccessor navigationType
idSuccessor navigationTypeBearNav
nameOfMap
-
next node
```

The node's id corresponds to its position in the file. Position and distance are floating-point numbers. Id, a number of successors, and navigation type are integers. The name of a map is a String. The first two lines describe the node itself. Then, there are lines for every successor, where we specify its id and navigation type. If the navigation type is BearNav, then one more line is added where the BearNav map name is written. For the visualization of edges and action server that finds the closest edges to a point, another .txt file has to be present. This file contains information about edges mapped by BearNav. This file is setup like:

```
idEdge x1Pos y1Pos id1 x2Pos y2Pos id2
another edge
```

On every line we have an edge id and two points that define the edge.

5.1 Simulation testing site

The simulation was created using *Gazebo* a widely used platform for robotic simulations. We have a default world that is built on. This default world has a ground plane and a light source. This is the starting point in creating the simulation. Then simple objects were placed using the gazebo model editor. *Gazebo* model editor can be accessed when launching *Gazebo* from the GUI, simple shapes such as boxes, cylinders, and balls can be placed into the world, also any models from the *Gazebo* library, which range from postboxes to 1:1 parks. A custom model added to the model path can then be inserted into the world from the model editor. Walls were added where their textures are chosen in the script for wall textures, and these textures are various pictures. Walls form a square-like shape that has approximately $3600m^2$ of surface. Walls were created in the *Gazebo* building editor. The building editor can be accessed similarly to the model editor. In the building editor, we specify our layout and then generate it into an SDF file. In the SDF file, we can then change any other properties, such as height or textures.
When these are present only thing is the robot model itself. The robot model was also configured in the SDF format. After which robot model it was modeled see the section 4.2. The configuration of the model is in section 3.6. When we add this model to the simulation, we have a robot that can interact with the created environment.

During this process, there have been some issues encountered. Since we wanted to use custom pictures as textures instead of the preset ones because of greater detail and variety. A custom script was created to substitute for the preset textures. When launched, the textures would appear in the simulation as expected, but when we were looking at the image that the robot's camera was seeing, the textures were not shown. Instead, we only saw Gazebo's default grey color. The issue was resolved by adding the path to the walls in an environment variable because the camera saw only textures from the default path. Another issue was that often the robot was unresponsive to velocity commands. This problem arose because of relaunching of the simulation. Some processes were not killed and continued to run and then interfered with the newly launched simulation. By correctly setting up the drive plugin and killing all the processes, this issue is mostly solved.

5.2 Closed space, real-world testing site

This section will cover the environment of a closed space for the Wild Thumper 4.1. Due to the Covid-19 pandemic as of writing of this thesis, there have been limitations and difficulties in choosing a place to freely perform experiments with a robot, whether the space would be at university or anywhere else. So for the purpose of the experiment, the Wild Thumper robot has been lent to me to perform the experiments at home. The robot mapped several *BearNav* maps in the apartment, each a few meters in length. The configuration of *BearNav* was such that the detector of the features was *FAST* and descriptor *BRIEF*. The features were saved every meter. The number of features to be extracted from the image was

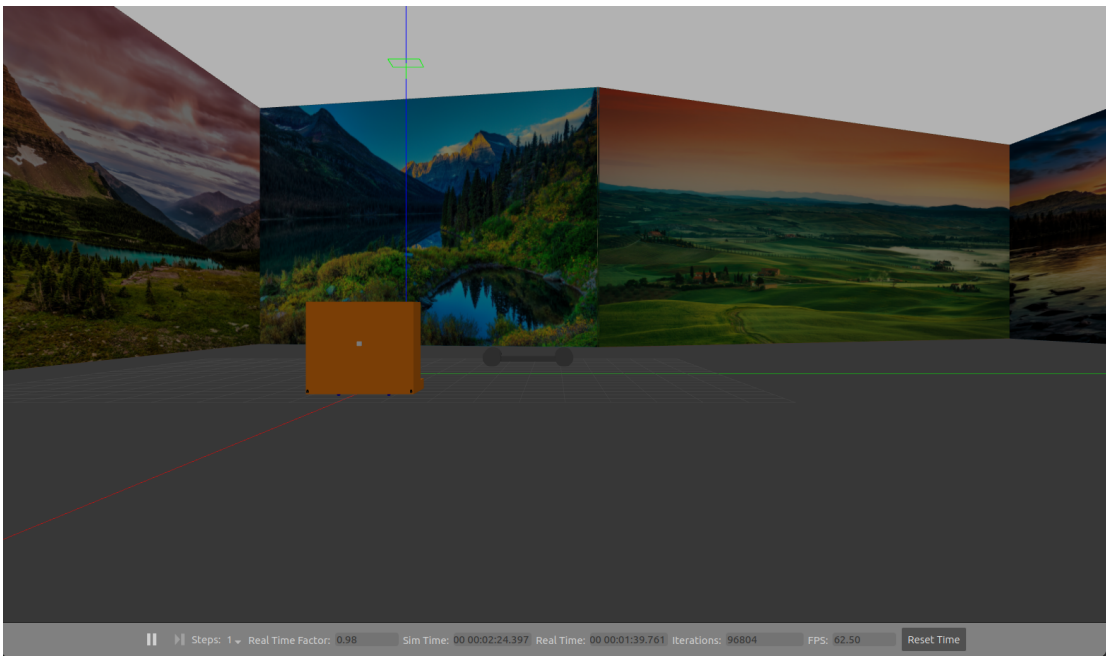


Figure 17: View of the robot in the simulation

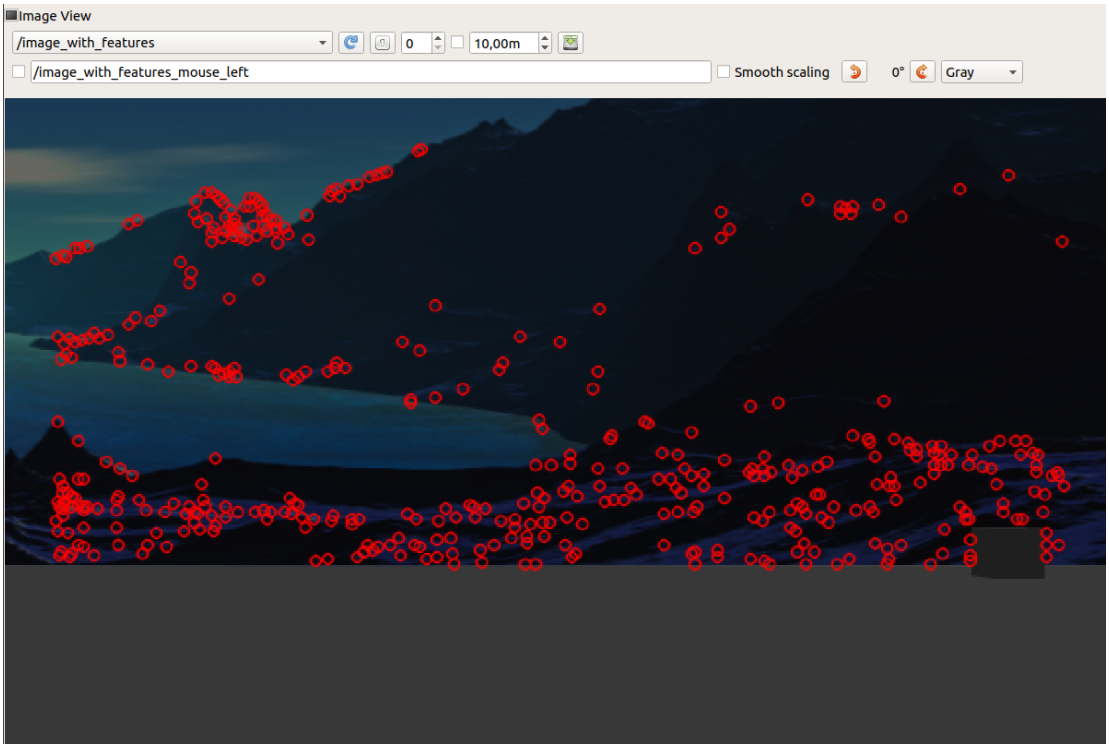


Figure 18: Camera view from the robot and features extracted in the simulation

set to 500. Then the description of the graph that was represented by the maps was created in two .txt files. Since the robot had different steering than the robot in the simulation, the robot in the simulation has ackermann steering[11]. In contrast, the Wild Thumper has a differential drive[9], the turn robot module had to be adjusted appropriately. For the Wild Thumper robot, it means it can turn on the spot because its central point is in the middle of the robot. The turn robot module was adjusted so that only the angular velocity was necessary for turning the robot, and the linear could be set to zero.

With these preparations, the robot was ready to perform experiments. Soon the dataset had to be modified because the features that were periodically saved every meter were too far apart, resulting in the robot going from its path too far before it could be compared to the next saved feature. The mapping had to be done again with new parameters. New maps had features saved every 0.25m. This change increased the reliability of traversal.

5.3 Open space, real-world testing

This dataset is from the Škoda Auto a.s. company's parking space in Mlada Boleslav. A space with three Škoda cars and the Phoenix 4.2 robot was assigned where we could operate. After getting access to the facility, we could prepare the setting for the experiment. Since the space was relatively small, only two *BearNav* maps could be mapped. These were mapped using the default configuration of *BearNav* which was *FAST* for the detector of features and *BRIEF* as the descriptor. Features, velocity commands of the robot were saved every meter. One map that went along the parked cars was 42m long, and the second that was perpendicular to the other one, and it was 16m long. These maps had to be represented as a graph in two .txt files. The turn robot module had to be replaced by a different turning action server because the curve created by the turning would be too large. It was replaced by an action that was implemented in the state machine of the driver. This action turned the robot around a center point which was at the end of the ramp. This is due to the fact that the turn robot module was tuned for the specific vehicle and in future this node will most likely be used.

Rosbags have been saved from every traversal performed. Setting up even such a simple environment was quite lengthy because of the robot's size and maneuverability. The robot can reset its odometry through a ROS service call which is useful because when launching a robot from a global starting position, we do not have to turn on and off the robot for it to correspond.

6 Results

This section describes the results of testing parts of the system as well as the testing of the system as a whole on different testing sites. The results are from the simulation and two real-world testing sites.

6.1 System modules

The modules were tested one by one during the creation of the system. The first module tested was the distance server action server described in 3.1. To test this module, various files containing information about edges in a graph were used. At first, the module returned correct details on the closest edge to a given point, such as the intersection coordinates, length from the given point to the intersection. An edge was correctly classified as the nearest edge in some cases, but the information returned did not correspond to the values expected. The problem was due to the fact the algorithm did not work when the line represented by the edge was either vertical or horizontal in the 2D space because the algorithm worked with the line as a function. When the line was vertical, it was not a function. In contrast, when the line was horizontal, the perpendicular line to it that was meant to be found and used to find the intersection was again vertical. This was solved by adding special cases when the line processed was vertical or horizontal. With these changes, the module worked correctly. It was modified later to return not the closest edge but up to two closest edges, and a parameter could be set as the maximum distance that an edge to a given point can have, but this was done for the needs of another module.

The second module was the A^* action server described in the 3.2. It loads information about nodes, their positions, distances on *BearNav* maps, successors and creates a graph from them. Same as the distance action server, this did not need any other environment apart from the information about the nodes to be tested. While the visual representation of the graph might be helpful, it is not necessary to test this module. The action server relied on the odometry of the robot to assess its starting position, but for testing the algorithm without the robot, simply publishing a message with its intended position on the odometry topic is enough. Then the goal can be called with the position to navigate to. At first, the graph was modified so that the starting node was created from the robot's position and its successors were both nodes connected to the closest edge found by the distance server module. The goal node was constructed from the given position with no successors, but the first node's successors on the closest edge to the given position were modified by adding the goal node to them. While this was a valid graph and the shortest path could be found between the start and goal nodes, it was not accurate when it had to represent a robot's path. The problem was in initializing the starting and goal nodes. Assuming that the robot could navigate to both nodes of the closest edge was wrong because it did not consider the robot's orientation. Since we assume the robot navigates on straight oriented edges aside from the turning edges, we need to find up to two closest edges and decide which of their end positions the robot is more closely oriented to. This found position is set as

the starting node's successor, avoiding a path that the robot could not navigate. Now we have a path that the robot can navigate to but might not be necessarily the shortest one. This is because when choosing the goal node and how it is connected to the graph, we only check the closest edge. Still, we do not expect that another edge is oriented differently and slightly more far away from the given point, but still satisfying the constraints might lead to another goal state. With two goal states, we run the A* algorithm twice for each version of the graph with its goal node and compare the lengths of each path and choose the shortest one. With this, the module reliably finds the shortest paths for any of the graphs given.

6.2 Simulation results

In a simulation with a robot that could be driven by commands, the turn robot module could be tested. With this module, there weren't any significant problems. The linear and angular velocity parameters that were sent periodically during the turning maneuver had to be tuned. If the linear velocity was too great, the maneuver might take too much space, and if too low, the maneuver would take too long. Similarly, with angular velocity, where if set too high, the robot would behave unnaturally, and if too low, the maneuver would take too much space and time. The parameters were tuned in few iterations by hand. At first, the turn robot action server only turned the robot left or right but was modified to accept direction (clockwise, anti-clockwise) and angle in degrees to turn in that direction. This change was meant to help the module be better for further reuse.

With a mobile robot, mapped *BearNav* maps, configured turning, and .txt files that represent a graph, we tested the navigator module that depends on every other main module, as well as the *BearNav* action servers. The system was launched on a square-like graph in the simulation, see figure 19. The first problem that was encountered was that when given a goal to navigate to a path has been found. The robot moved accordingly through the first mapped edge than when it had to go through a second edge or every other subsequent edge, and it immediately returned that it successfully navigated through it even though it stayed at the very beginning of the edge. The problem was replicated in simpler scenarios. The problem lays in improper initialization of ROS service variable, this *rosservice* call was meant to call an action that would set a distance on which the robot was located. Since the service was not appropriately called, the robot thought it was at the end distance of the previously traversed edge. Suppose the next edge was the same in size or lower it then automatically returned that it completed the traversal. With this fixed, the robot could navigate freely through the graph without any significant problems. Some problems prevailed, such as when launching the simulation, the robot would sometimes not respond to velocity commands.

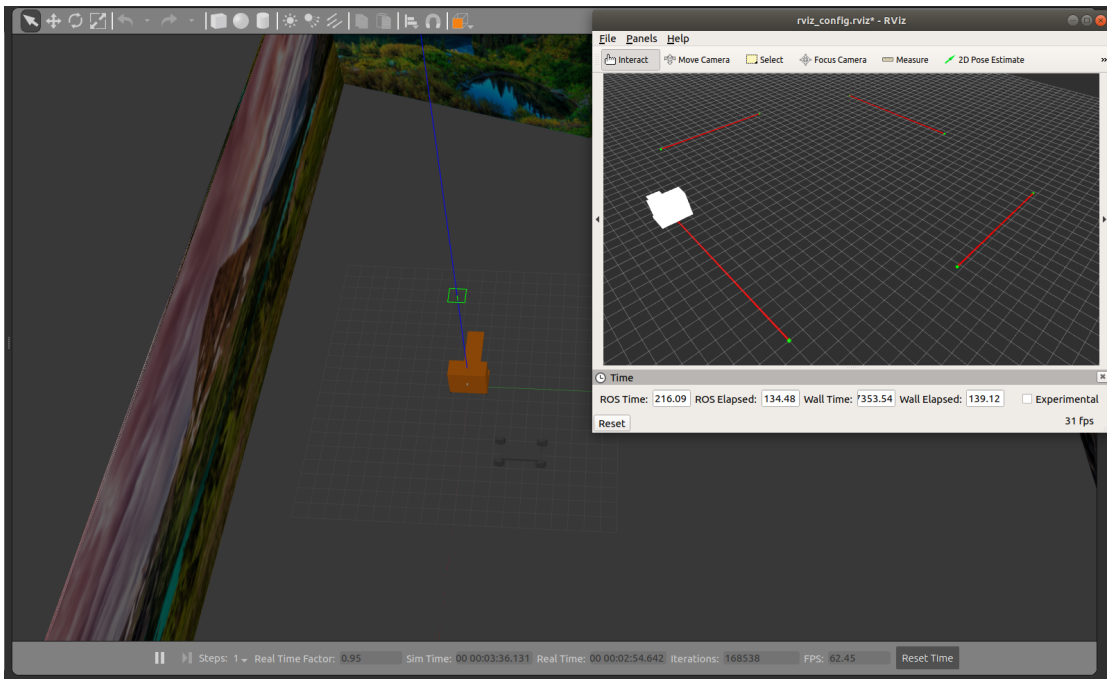


Figure 19: Top down view of the *Gazebo* simulation and *RViz* window with the visualization of edges

6.3 Closed environment results

Having the Wild Thumper robot, mapped edges in the environment, files containing information about the graph, and the turn robot module configured for the robot where the linear velocity is set to zero, so it turns on the spot. See figure 20 for a view of the graph created for this space. When launching the system, the first problem was that the robot diverged too much from its path, so new *BearNav* maps had to be created that saved found features more often. After that, the same problem that appeared in the previous experiment arose. It did not traverse the second edge of the path or any after that, but this was resolved in the simulation experiments. When this problem was fixed, the environment changed substantially, and the mapped edges did not longer correctly represent the reality. New edges would have to be mapped, and the graph changed accordingly. This was decided to be too time-consuming and was not done because of the outside effects. The issue was not related to the system. Videos have been recorded of the robot traversing a simple path on a single edge.

6.4 Škoda results

For this experiment, the Phoenix robot 4.2 was ready, as well as two mapped edges and a turning defined in the robot's driver. While testing, the path returned did not correspond to an expected one. This was due to not checking a particular case when the line was

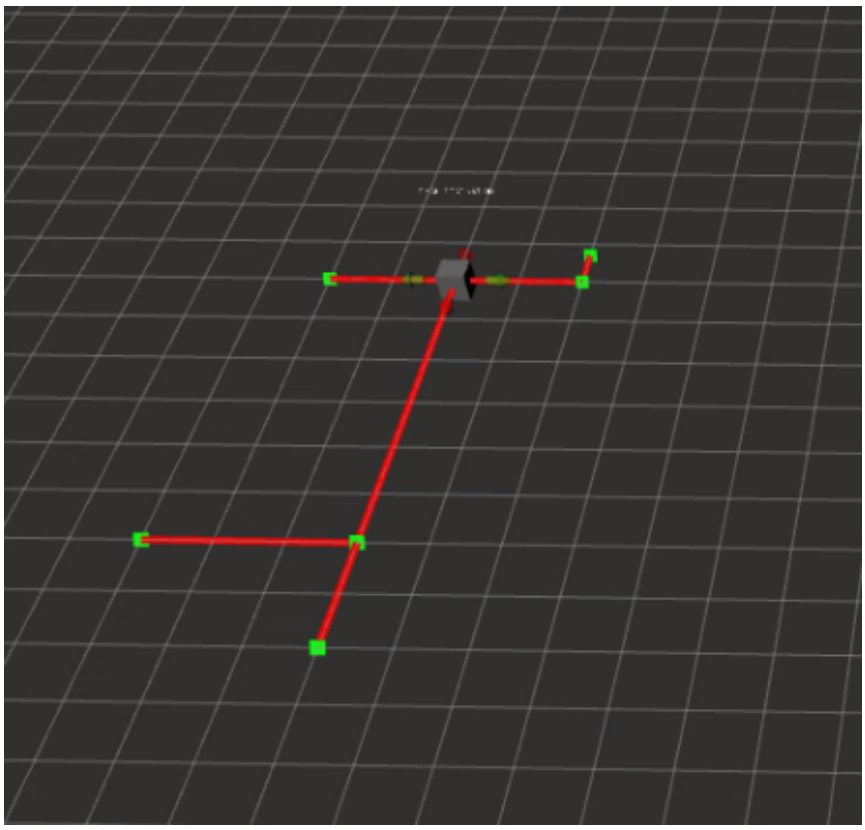


Figure 20: View of the graph for the closed environment from *RViz*

horizontal in the 2D space. The issue was resolved on the spot by checking the specific case and dealing with it appropriately in the code of the distance server module. Sometimes not all necessary components were launched, resulting in resetting the experiment. This was costly in time since the robot had to be returned to the starting position manually with a controller and having to reset specific parameters. The robot was used without its safety precautions like collision avoidance and safety stops since they were no yet implemented. After correctly initializing every part of the system, the robot then successfully navigated to a given point, traversing an edge, making a turning maneuver, and finally traversing the last edge to the point that was nearest to the given point. This traversal was done twice. It was recorded on video from various angles, and *rosbags* containing every topic published have been saved as well.

While repeatedly using the navigator on the same line when the robot finished traversing, another error was encountered. The robot did not move when the navigation action server was given another command to continue elsewhere. When the navigation action server called a *BearNav* action server to traverse an edge, it set the robot's position on the edge through a *rosservice* call, and then it set the distance to be traveled, which was equal to the goal distance minus the distance set. This was wrong as the distance to be selected was meant to be only the goal distance. The *BearNav* action server would itself then recognize

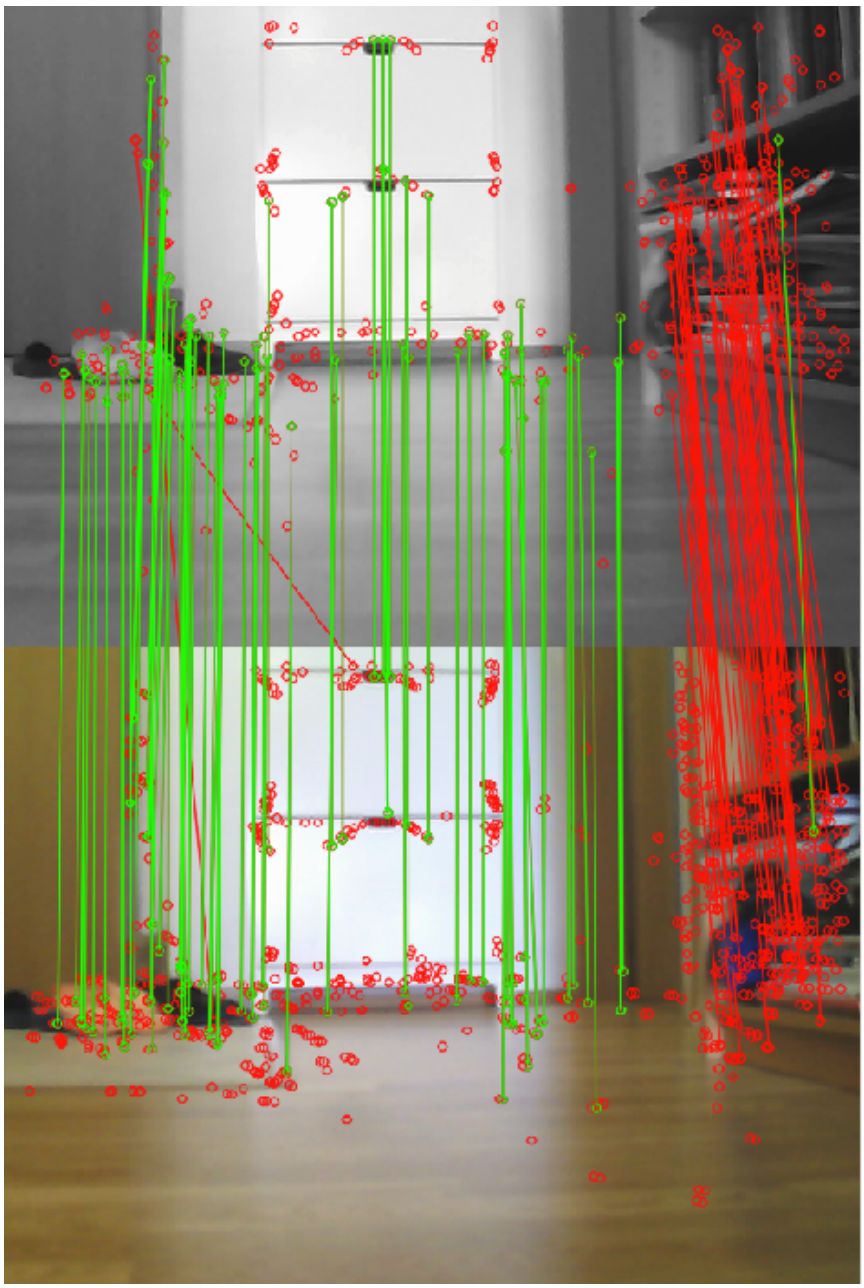


Figure 21: Comparison of navigation matches when traversing an edge

it needs to travel the difference between the distance between the goal and the starting position. This caused that when it should traverse an edge from a starting distance other than zero, it would not go to the desired distance, and if the set distance were greater than the final goal distance, the robot would not move at all. By removing the subtraction of the starting distance, the issue was fixed, and the robot could be navigated without further problems. Since the Phoenix robot is meant for the final deployment of the Škoda auto

project, we can conclude this experiment as a success.

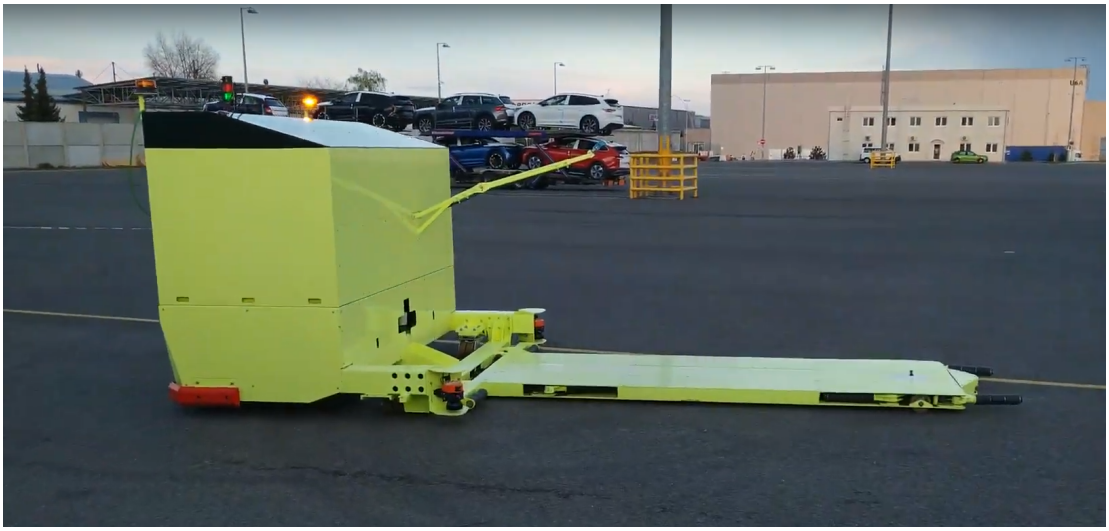


Figure 22: View of the robot during the experiment

6.5 Comparison of the system and the ROS navigation stack

The system implemented and the ROS navigation stack both realize navigation in a 2D map and rely on odometry for localization. Navigation stack tries to find any valid path [22] while this work's system finds the shortest path. The navigation system additionally uses laser sensors for localization and obstacle avoidance, whereas the system in this work uses a camera image for navigation and does not have any obstacle or collision avoidance implemented, even though the robots tested had laser sensors. The system does not need any laser sensors because they were not a concern of this work, but they can be used for better localization in the future. This system is possible to use on mobile robots not only with a differential or a holonomic drive, but also robots with ackermann steering or any other kind of drive that accepts the geometry Twist message and can turn or drive straight. Navigation stack is only for robots with a differential or holonomic drive and additionally it is recommended that robot has a circular or a square like shape. Pre-requisite to run the navigation stack are prepared configuration and launch files in a package created by the user. The system of this work requires to add information about the nodes of a graph in a file, map edges that are meant to be traversed with BearNav, and configure the turning module for the specific robot.

7 Conclusion

Even though there were problems, I have managed to perform successful experiments in the simulation, real-world scenarios, and tasks that the Phoenix robot was made for. Experiments consisted of traversing different paths on various graphs and comparing visually if the robot is taking the correct sequence of actions.

Problems were encountered during the process of creating the simulation, such as making accurate representations of models and sensors that they would be similar to their real-world counterparts, making sure that when the simulation is launched, it always responds to every vital request. When creating the system and all its modules it consists of, assuring proper communication between all the parts also proved to be quite challenging. Not handling special cases that were needed to be checked and dealt with appropriately also happened. These issues were found from the creating process up to the performing of experiments. All of the above problems were resolved.

With this, I can assume that my assumption in the introduction was correct, and I managed to create a system that reliably navigates a robot through an environment that is represented as a graph, and the edges are mapped BearNav maps. The BearNav system corrects the traversal, forcing the robot to go through a trained path, thus removing the buildup of the error in odometry even when launched in succession for the found path between the starting and end position. The system can be configured for any system that can navigate straight lines and not just BearNav.

The system implemented in this work can be chosen over the navigation stack, if the robot's drive is any other than differential or holonomic, the robot does not have a laser sensor and has a camera, needs to find the shortest path or the implementation of obstacle avoidance is not necessary.

There are still things to be done that could improve the system, optimizing the source code for better performance. Replacing the A^* search algorithm with a D^* search algorithm which generally performs better than traditional A^* [37]. The representation of a graph could be converted from the .txt file to a more readable and intuitive format like *.YAML*. For the graph to be created from the files, the information about the vertices have to be added manually, so a graphical interface where this graph could be created would make the system more user-friendly.

References

[1] Florentina Adascalitei and Ioan Doroftei. Practical applications for mobile robots based on mecanum wheels - a systematic survey. *Romanian Review Precision Mechanics, Optics and Mechatronics*, pages 21–29, 01 2011.

[2] J. Hrbáček, T. Ripel, and Jiri Krejsa. Ackermann mobile robot chassis with independent rear wheel drives. pages T5–46, 10 2010.

[3] Michael Mortimer, B. Horan, Matthew Joordens, and Alex Stojcevski. Searching baxter's urdf robot joint and link tree for active serial chains. *2015 10th System of Systems Engineering Conference, SoSE 2015*, pages 428–433, 07 2015.

[4] Tomas Krajnik, Filip Majer, Lucie Halodová, Jan Bayer, Tomas Vintr, and Jan Faigl. Navigation without localisation: reliable teach and repeat based on the convergence theorem. *arXiv preprint arXiv:1711.05348*, 2017.

[5] HR Everett. *Sensors for mobile robots*. CRC Press, 1995.

[6] Dieter Fox, Wolfram Burgard, and Sebastian Thrun. The dynamic window approach to collision avoidance. *IEEE Robotics & Automation Magazine*, 4(1):23–33, 1997.

[7] David Filliat and Jean-Arcady Meyer. Map-based navigation in mobile robots:: I. a review of localization strategies. *Cognitive Systems Research*, 4(4):243–282, 2003.

[8] Pedro Gregorio, Mojtaba Ahmadi, and Martin Buehler. Design, control, and energetics of an electrically actuated legged robot. *IEEE Transactions on Systems, Man, and Cybernetics, Part B (Cybernetics)*, 27(4):626–634, 1997.

[9] Se-gon Roh and Hyouk Ryeol Choi. Differential-drive in-pipe robot for moving inside urban gas pipelines. *IEEE transactions on robotics*, 21(1):1–17, 2005.

[10] A Kilin, P Bozek, Yury Karavaev, A Klekovkin, and V Shestakov. Experimental investigations of a highly maneuverable mobile omniwheel robot. *International Journal of Advanced Robotic Systems*, 14(6):1729881417744570, 2017.

[11] Wm C Mitchell, Allan Staniforth, and Ian Scott. Analysis of ackermann steering geometry. Technical report, SAE Technical Paper, 2006.

[12] Johann Borenstein and Liqiang Feng. Measurement and correction of systematic odometry errors in mobile robots. *IEEE Transactions on robotics and automation*, 12(6):869–880, 1996.

[13] Kok Kiong Tan, Huixing X Zhou, and Tong Heng Lee. New interpolation method for quadrature encoder signals. *IEEE Transactions on Instrumentation and Measurement*, 51(5):1073–1079, 2002.

[14] Peter Yap. Grid-based path-finding. In *Conference of the canadian society for computational studies of intelligence*, pages 44–55. Springer, 2002.

[15] Douglas Brent West et al. *Introduction to graph theory*, volume 2. Prentice hall Upper Saddle River, 2001.

[16] Scott Beamer, Krste Asanovic, and David Patterson. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis*, pages 1–10. IEEE, 2012.

[17] Robert Tarjan. Depth-first search and linear graph algorithms. *SIAM journal on computing*, 1(2):146–160, 1972.

[18] Donald B Johnson. A note on dijkstra's shortest path algorithm. *Journal of the ACM (JACM)*, 20(3):385–388, 1973.

[19] Peter E Hart, Nils J Nilsson, and Bertram Raphael. A formal basis for the heuristic determination of minimum cost paths. *IEEE transactions on Systems Science and Cybernetics*, 4(2):100–107, 1968.

[20] František Duchoň, Andrej Babinec, Martin Kajan, Peter Beňo, Martin Florek, Tomáš Fico, and Ladislav Jurišica. Path planning with modified a star algorithm for a mobile robot. *Procedia Engineering*, 96:59–69, 2014.

[21] Official robot operating system website. `https://www.ros.org/`. Accessed: 2020-02-01.

[22] Morgan Quigley, Ken Conley, Brian Gerkey, Josh Faust, Tully Foote, Jeremy Leibs, Rob Wheeler, Andrew Y Ng, et al. Ros: an open-source robot operating system. In *ICRA workshop on open source software*, volume 3, page 5. Kobe, Japan, 2009.

[23] Morgan Quigley, Brian Gerkey, and William D Smart. *Programming Robots with ROS: a practical introduction to the Robot Operating System.* " O'Reilly Media, Inc.", 2015.

[24] David Gossow, Adam Leeper, Dave Hershberger, and Matei Ciocarlie. Interactive markers: 3-d user interfaces for ros applications [ros topics]. *IEEE Robotics & Automation Magazine*, 18(4):14–15, 2011.

[25] Margaret A Ellis and Bjarne Stroustrup. *The annotated C++ reference manual.* Addison-Wesley, 1990.

[26] Raul Rojas and Alexander Gloye Förster. Holonomic control of a robot with an omnidirectional drive. *KI-Künstliche Intelligenz*, 20(2):12–17, 2006.

[27] Tully Foote. tf: The transform library. In *2013 IEEE Conference on Technologies for Practical Robot Applications (TePRA)*, pages 1–6. IEEE, 2013.

[28] Official gazebo simulator website. `http://gazebosim.org/`. Accessed: 2020-02-01.

[29] Nathan Koenig and Andrew Howard. Design and use paradigms for gazebo, an open-source multi-robot simulator. In *2004 IEEE/RSJ International Conference on Intelligent Robots and Systems (IROS)(IEEE Cat. No. 04CH37566)*, volume 3, pages 2149–2154. IEEE, 2004.

[30] Zandra B Rivera, Marco C De Simone, and Domenico Guida. Unmanned ground vehicle modelling in gazebo/ros-based environments. *Machines*, 7(2):42, 2019.

[31] Tomáš Krajník, Jan Faigl, Vojtěch Vonásek, Karel Košnar, Miroslav Kulich, and Libor Přeučil. Simple yet stable bearing-only navigation. *Journal of Field Robotics*, 27(5):511–533, 2010.

[32] James Diebel. Representing attitude: Euler angles, unit quaternions, and rotation vectors. *Matrix*, 58(15-16):1–35, 2006.

[33] Official website of the robo claw motor controller manufacturer. `https://www.basicmicro.com/`. Accessed: 2020-02-01.

[34] Frederick G Fernald. Analysis of atmospheric lidar observations: some comments. *Applied optics*, 23(5):652–653, 1984.

[35] Felix Endres, Jürgen Hess, Jürgen Sturm, Daniel Cremers, and Wolfram Burgard. 3-d mapping with an rgb-d camera. *IEEE transactions on robotics*, 30(1):177–187, 2013.

[36] Oren Ben-Kiki, Clark Evans, and Brian Ingerson. Yaml ain't markup language (yaml™) version 1.1. *Working Draft 2008-05*, 11, 2009.

[37] Masoud Nosrati, Ronak Karimi, and Hojat Allah Hasanvand. Investigation of the*(star) search algorithms: Characteristics, methods and approaches. *World Applied Programming*, 2(4):251–256, 2012.

Appendix

Content of Attachments

In table 1 are listed names of all files attached and their contents.

File names	Description
source.zip	source codes, action, launch, configuration files
skoda_exp.mp4	video of the experiment performed at Škoda
sim_video1.mp4	video of the experiment done in simulation
sim_video2.ogv	video of early stage navigation in simulation
closed_env.mp4	video of an edge traversal in a closed environment

Table 1: Contents of the attachments