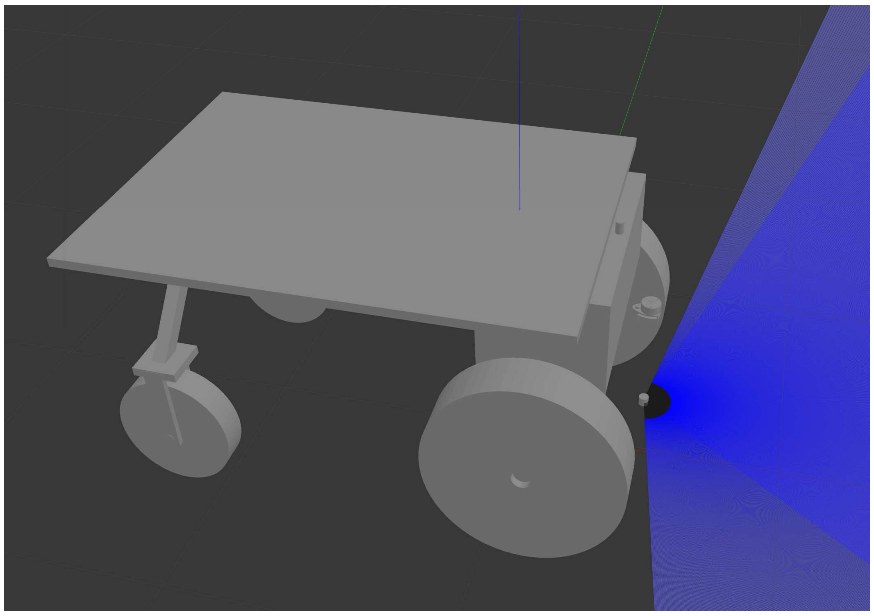
ANT ROBOTICS WAYPOINT NAVIGATION

Author	Roman Hanselmann, Simon Steigmeier, Urban Willi
Coach	Manuel Schlegel, (Christian Bermes)



4. JANUAR 2023

FHGR

Abstract

Definition of Task	This project aims to develop a module for autonomous waypoint navigation and mapping. It will enhance the autonomy of an existing agricultural robot, belonging to Ant Robotics Gmbh.
Goal	The module's task is to localise the robot's own position on a given map. Outside of crop fields, it should automatically find a path to another selected position and autonomous follow this path. The collected data will create a map, which will be saved locally. Since the robot will work in the same environment as humans, it is important that the robot won't crash with humans.
Approach	Different possible solutions were collected and discussed with the client until the most suitable possible solution was found. To prevent unnecessary work and problems, the internet was searched to find possible open source solution that can be partially used in this project. To test the module, a virtual world was created to simulate a more realistic environment.
Essential outcome	In the process of this project a ROS module was created, that allows the client's agriculture robot to plan a path to its destination and drive there autonomously. To test the created software, a virtual world was created to simulate the robot.
Keywords	Waypoint Navigation, Localisation, ROS

Record of changes

Version	Reason for change	Initials	Date
1.0	Creation of this document	RH	19.12.2022
1.1	Adding new .py description	SS	08.01.2023

Table of Contents

Abbreviation

ROS	Robot Operating System
IMU	Inertial measurement unit
LiDAR	Light detection and ranging
GPS	Global position system
GPS - WGS84-format	Position in -> longitude & latitude: `(8.89999984, 49.899999963)`
GPS - UTM-format	Position in zones: `z: 32` `l: U` `x: 492818.42723554146` `y: 5527517.131716844`

1. Introduction

Context

The Client of this project is Ant Robotics. Ant Robotics is a Start-up from Germany that develops and produces support robots for agriculture.

Robot Description

The robots task is to support workers during the harvest of fruits and vegetables, reducing non-productive time spent on transporting crates. It will slowly follows the workers and carry their crates.

Assignment

In this project, an additional software module for this agriculture support robot has to be developed. The module's purpose is to make the robot more autonomous. With this module, the robot should be able to autonomous drive to a selected destination, plan a short path to the destination and avoid obstacles on its way.

Approach

Once the details of the project were clear, it has been researched about the current state of the art of localisation and path finding. After a few possible solutions were found, it has been decided on which one to use. With this information the development of the module could be started. To test the module, a small virtual world has been created, where a virtual model of the robot could drive around.

2. Clarification of the task

In this chapter, the task and the already existing robot from Ant Robotics will be described in more details.

Setting

The ROS module that will be developed during this project, is to make an agriculture support robot more autonomous. The robot's work environment is on fields with crop rows, where human workers pick vegetables or similar products. Normally the human workers have to walk to the end of the field once their crate is full to get a new empty one. Since this is unproductive time, this robot was developed to slowly follow the human workers, carry their crates and enable them to work more productive. Now there is still a human worker needed to manually drive the robot to the next crop row or to an unloading station once it's fully loaded. The module that will be developed during this project shall give enough autonomy to the robot, that it can do these tasks on its own.

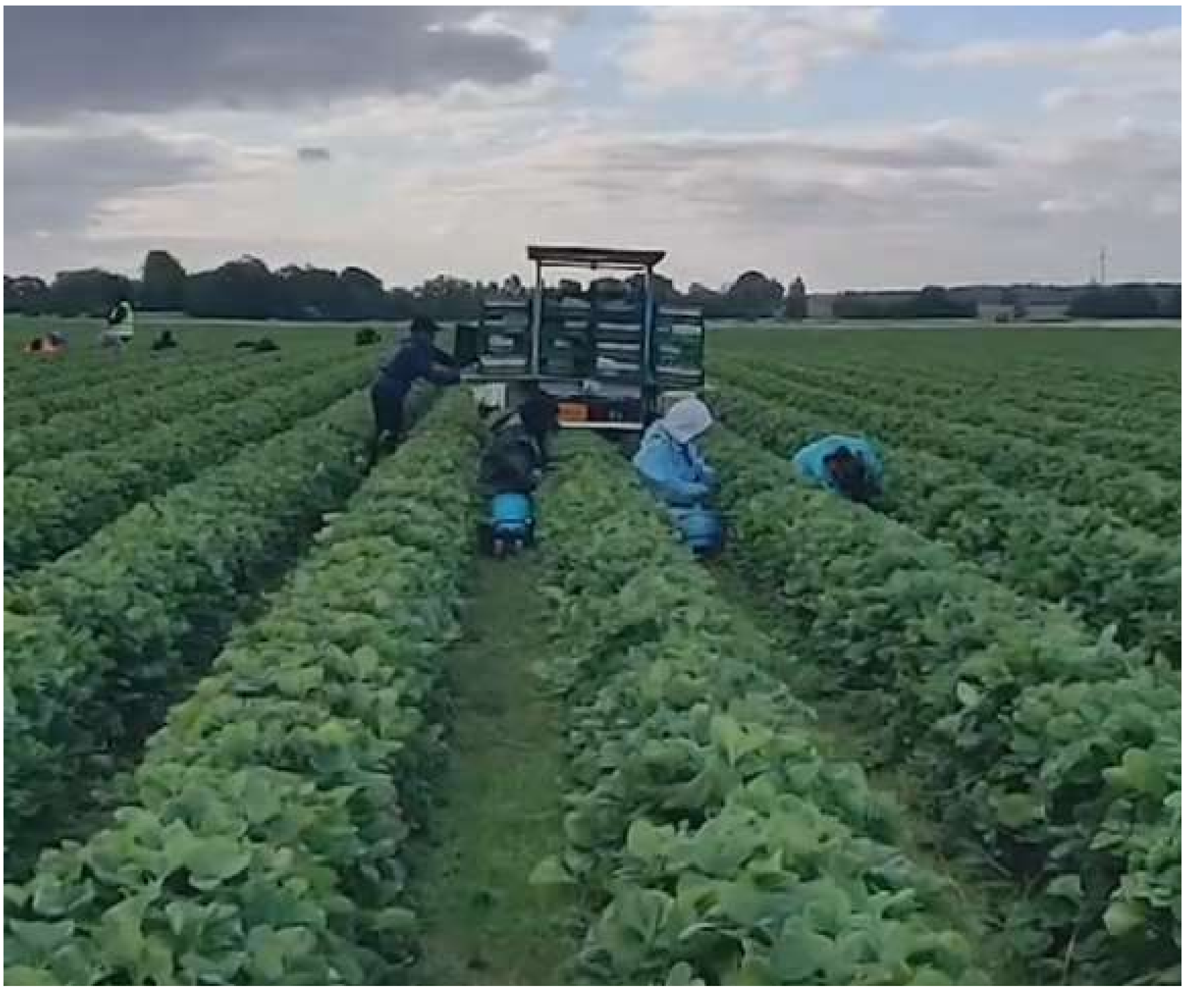


Figure 1: Crop rows with robot

https://www.youtube.com/watch?v=0zSuqwCQMwk

Specifications

Following are all the must specifications listed. **For all specifications, see "Specification_book_signed.pdf" in the attachment**.

Waypoints	Waypoints define the goal for path planning. One goal at a time is defined by a module outside the system boundaries. Waypoints are defined as a coordinate in the world frame.
Position marker	Starting position and important markers for the return journey (transition field - road) can be saved on request.
Path planning	The robot can plan its movement from the current position to the next waypoint.
Driving control	A control algorithm regulates the execution of the planned path.
Obstacle avoidance	On an encounter with an obstacle, alternative paths can be calculated or the robot is stopped if no safe alternatives are found.
Disabling alternative paths	Obstacle avoidance can be turned off, for example inside crop rows. This is a safeguard to prevent damage to plants.
Mapping	A 2D map of fixed size (determined in runtime by another module) is created and saved. It receives obstacle data and crop row positions to add to the map.
Localisation	The robot can localise itself in the map and with the help of positioning data (for example GPS).
Documentation	The contractors will write a report about the project and hand it over at the end of the project. In addition a presentation will be held and a short video is produced showcasing the project functionality.
Legal requirement	The client takes care of the legal requirements. Only legal requirement the contractors have to keep in mind, is the robots current speed limit of 5km/h.
Open source code	Open source code may be used in the project when it makes sense, but usage of any such component must be green-lit by Ant Robotics first.
Modularity	The software project shall be modular in its nature, such that it can use the data provided to it by other ROS packages independent of the data source. It will be activated and commanded by another module.
Simulation testing	The project is software based and shall work on the simulation without the robot or additional hardware.
Programming language	The project should be developed with C++. Alternatively, python can be used for development.
Framework	The module is based on ROS1 Noetic.
Calculations	All calculations and actions are done and saved locally on board.

Table 1: Specifications

System boundaries

In the following illustration you can see the boundary of the already existing part from Ant Robotics (green) and the boundary of the task (red).

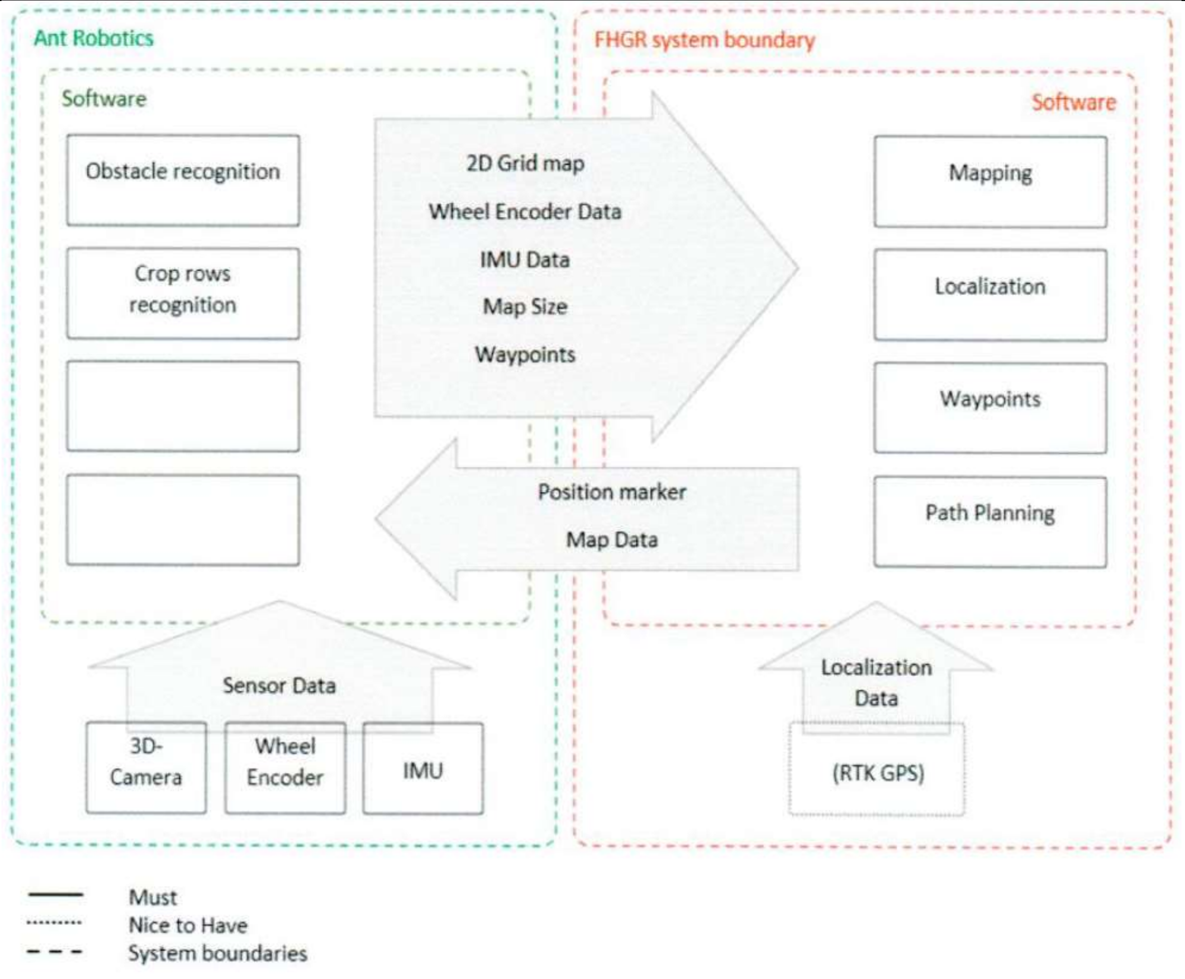


Figure 2: System boundaries

Preliminary work

The client already built a working robot before this project started. The robot has following sensors on board:

- Wheel encoder
- IMU
- 3D camera
- 2D LiDAR

This robot is already able to do certain tasks on its own (also see 2. Clarification of the task).

For easier testing, the client also created a virtual model of the robot to use in a virtual environment and made it accessible for this project.

The client also gave many useful tips regarding programming ROS.

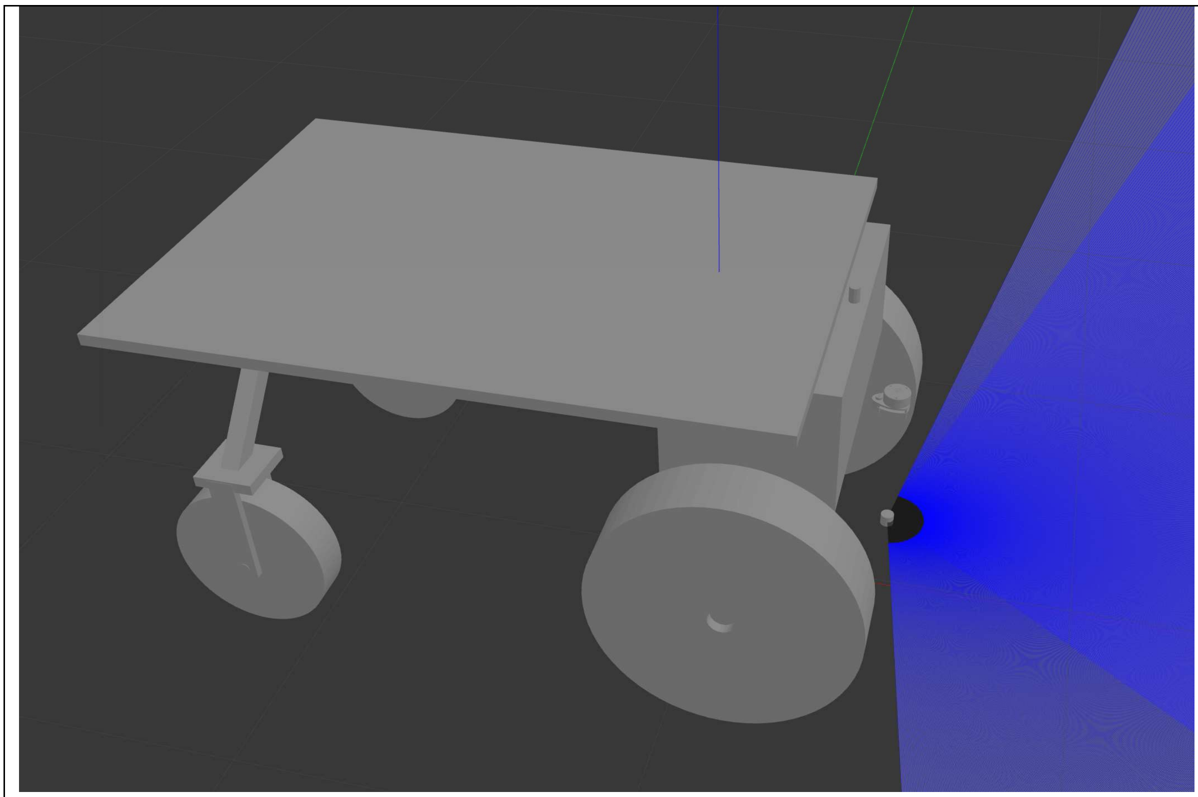


Figure 3: Virtual robot model

Project plan

The roadmap for the project can be found in the “Project_status_v4.pdf” file in the attachments. The roadmap shows what task is planned in which week of the year. In week 51, a one-week buffer was added for holidays and to compensate eventual delays.

3. State of the art

In this chapter, the results of the state of the art research will be shown. Different possible solutions were found and later it was agreed on one concept. For more detailed information about the concept see “Concept_desicion.pdf” in the attachments.

Mapping

For mapping, two possible solutions were found:

- 2D grid map

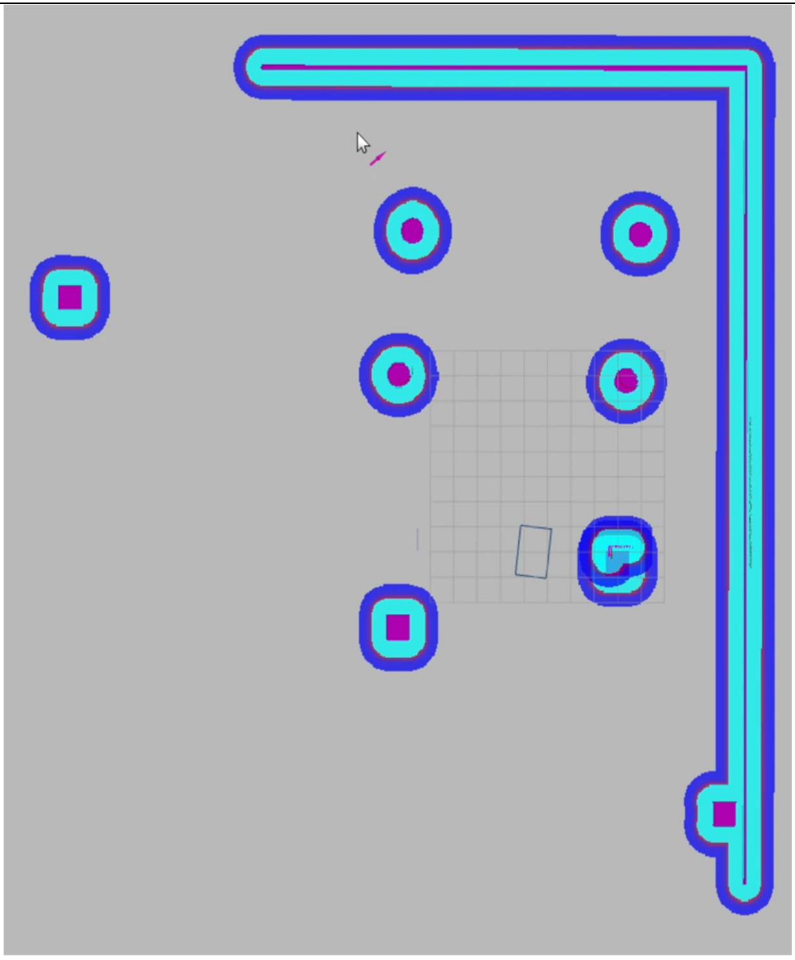


Figure 4: 2D grid map

https://risc.readthedocs.io/_images/ros_map.jpg

- 3D grid map

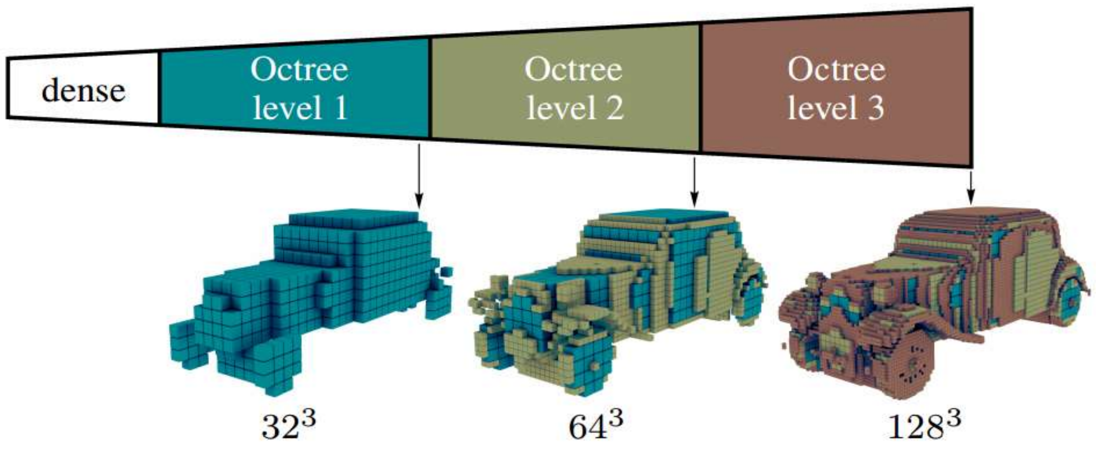


Figure 5: 3D grid map

https://wiki.ros.org/ccny_rgbd/keyframe_mapper?action=AttachFile&do=get&target=3rooms_octomap.png

Since the robot is big and is supposed to work on a more or less flat field, there is no real benefit of having a third dimension. On top of that, a 3D grid map needs a lot more storage space especially on large fields. With this information, it was decided to use a 2D grid map.

Localisation

For localisation, two possible solutions were found:

- GPS only
- GPS and IMU data

For localisation on an open field, some kind of GPS is the most obvious solution. But soon the question came up if GPS alone is robust enough for autonomous driving and what would happen if the robot loses the GPS signal for a moment. Luckily the robot already has a IMU integrated. The IMU itself is very inaccurate for autonomous driving, especially on an uneven underground. But it should be good enough to just keep its path until the robot gets the GPS signal again. That's why it was decided to use GPS and IMU data.

Path smoothing

For path smoothing, two possible solutions were found:

- Path smoothing algorithm + PID controller
- Pure pursuit controller + local planer (real time path adaptation)

Because the robot hasn't a narrow turning radius, a path smoothing element is needed. Both of the two found possible solutions would have a similar result. But since the first solution needs more steps to get a similar result, it was decided to use the Pure pursuit controller.

Object detection

For object detection, two possible solutions were found:

- 2D LiDAR only
- 2D LiDAR + 3D camera

The robot has currently mounted a 3D camera, on its top facing down, to detect crop rows and obstacles in front of the robot. Because of its small field of view, it was already decided beforehand to add a 2D LiDAR and increase its sight with that. The problem that occurred here is, that the 2D LiDAR can't be mounted to low, because plants might interfere with the sensor. But because of that it won't be able to detect the crop rows and crates. That's why it is necessary to combine these two sensors. To do so, it is necessary to transform the 3D camera data to a 2D point cloud and merge it together with the 2D LiDAR point cloud.

Because the client already has a ROS module that reads the 3D camera data, it was decided that the client will integrate the 2D LiDAR into the already existing module. The module, which will be developed in this project, will receive the already transformed 2D point clouds from the existing module.

Path finding

For path finding, two possible solutions were found:

- Dijkstra algorithm
- A* algorithm

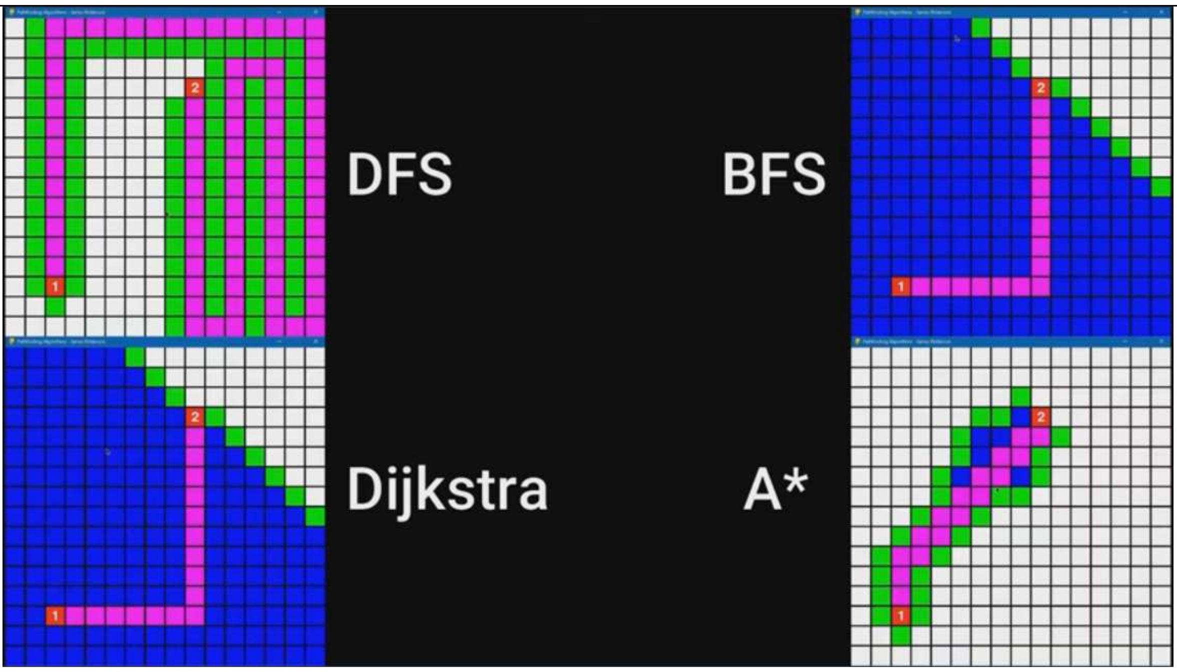


Figure 6: Different path finding algorithms - open area

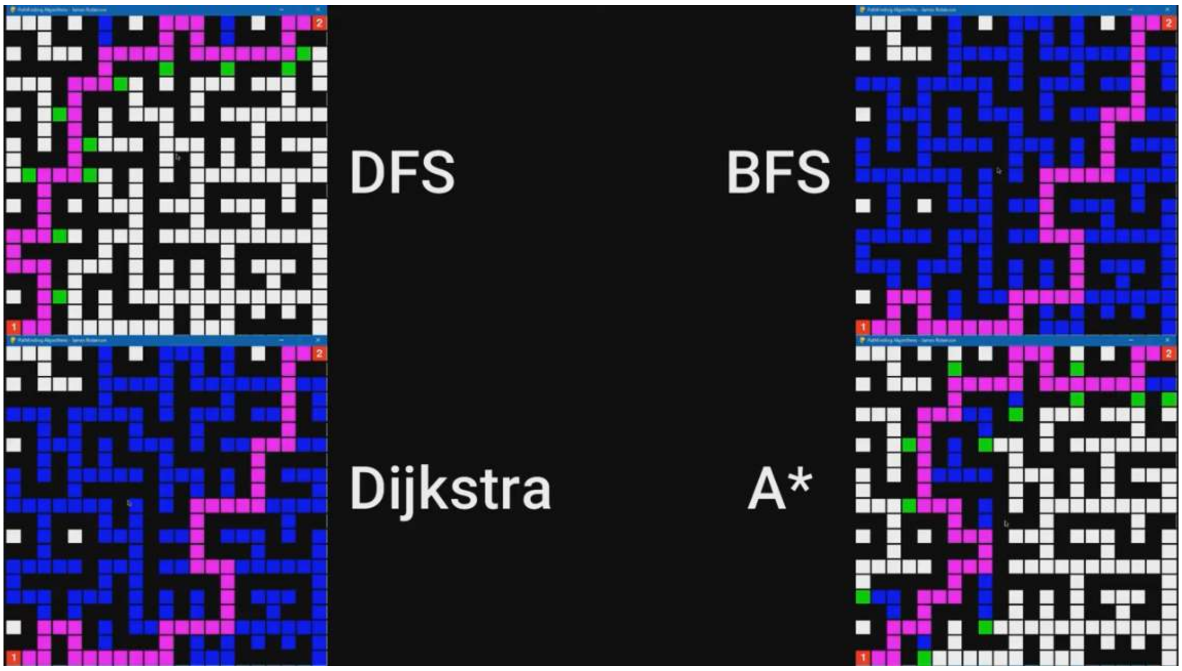


Figure 7: Different path finding algorithms – labyrinth

https://www.youtube.com/watch?v=aW9kZcJx64o

The Depth First Search and the Broad First Search algorithms are old algorithms that are either slow or won't provide the shortest path.

A* is the most common path finding algorithm nowadays. It finds the shortest path (in rare cases it’s not the very shortest but still a very short path) and it’s also very fast to do so.

The Dijkstra algorithm is mainly used when there are areas on your map that need to be weighted. For example, when a specific area is dangerous to drive through and should be avoided or when a specific area is easier to drive through than other parts of the map. The Dijkstra algorithm is also able to plan a path through multiple destinations instead of just one.

The robot however will work on a map that can be unknown previously. Therefore, weightings would have to be added manually once the map is discovered or the robot would have to add them ongoing while driving around the map. Adding weightings manually is not a good solution for this project since the customers who will use the robot, can’t be expected to any knowledge about robots. This could bother the customers because additional work is required and could lead to problems caused by incorrectly operating the robot. Furthermore adding weightings once the map is fully discovered probably means that the work is already done anyways.

Since it’s also not necessary to plan a path for multiple destinations at once, it was decided to use the A* algorithm.

4. Concept

In this chapter is a short summary of the decisions that were made together with the clients. Followed by a description of the concept for this project.

Decision summary

Problem	Solution	Reason for the decision
Mapping	2D grid map	A 2D grid map was chosen because it's enough for this project and there's no benefit of adding a third dimension.
Localisation	GPS and IMU	The GPS data are used to localise the robot and the IMU data will be used to calculate the current position, in case the GPS signal is lost.
Path finding	A* algorithm	The A* algorithm was chosen because it is the fastest algorithm up to date and also delivers the shortest path.
Path smoothing	Pure pursuit controller + local planer (real time path adaptation)	This solution was chosen because the other possible solution might have caused more work for a similar result.
Object detection	2D Point cloud multiple sources	There are two sensors for object detection available on the robot. A 2D LiDAR and a 3D camera. The 2D LiDAR can't be mounted to low, because plants on the ground might interfere with it. To still detect obstacles on the ground the data from the 3D camera are needed. The data will be transformed by the client and isn't part of this project. For this project the finished 2D point clouds will be available.

Table 2: Decision summary

Concept description

The robot is supposed to start in an unknown environment. To be able to work in an unknown environment, it got 3D camera and a 2D LiDAR sensor to sense obstacles in front of it. The robot already transforms these data to 2D point clouds and makes them available for it's modules. These data can be used to continuously create a 2D grid map. The map data can be saved locally.

The module should enable the robot to autonomously drive to a selected destination. To do that, a path from the current location of the robot to its destination is needed. To accomplish that, the A* algorithm can be used to find the shortest path to its destination. Because of the robot's turning radius, the path has to be smoothed with the pure pursuit controller.

In case that new obstacles appear while the robot drives through an unknown area or in case the already discovered area changes, the module needs a local planer to adjust its path if necessary.

5. Realisation

Robot and Software Description

Physical robot
The robot is an agricultural vehicle used for transporting crates. The software module shall not depend heavily on the physical properties of the robot, so that it can be used on different robots. Physical constraints that were taken in account is the non-holomorphic nature, the robot has a minimum turning radius.

Sensors available on the robot
The sensors which are mounted or will be mounted to the robot are the input sources for our system [figure 8]. The implementation of the sensors into the ROS environment is outside the system boundaries of our module and is done by Ant Robotics. For the development of our module, all sensors were simulated in ROS.

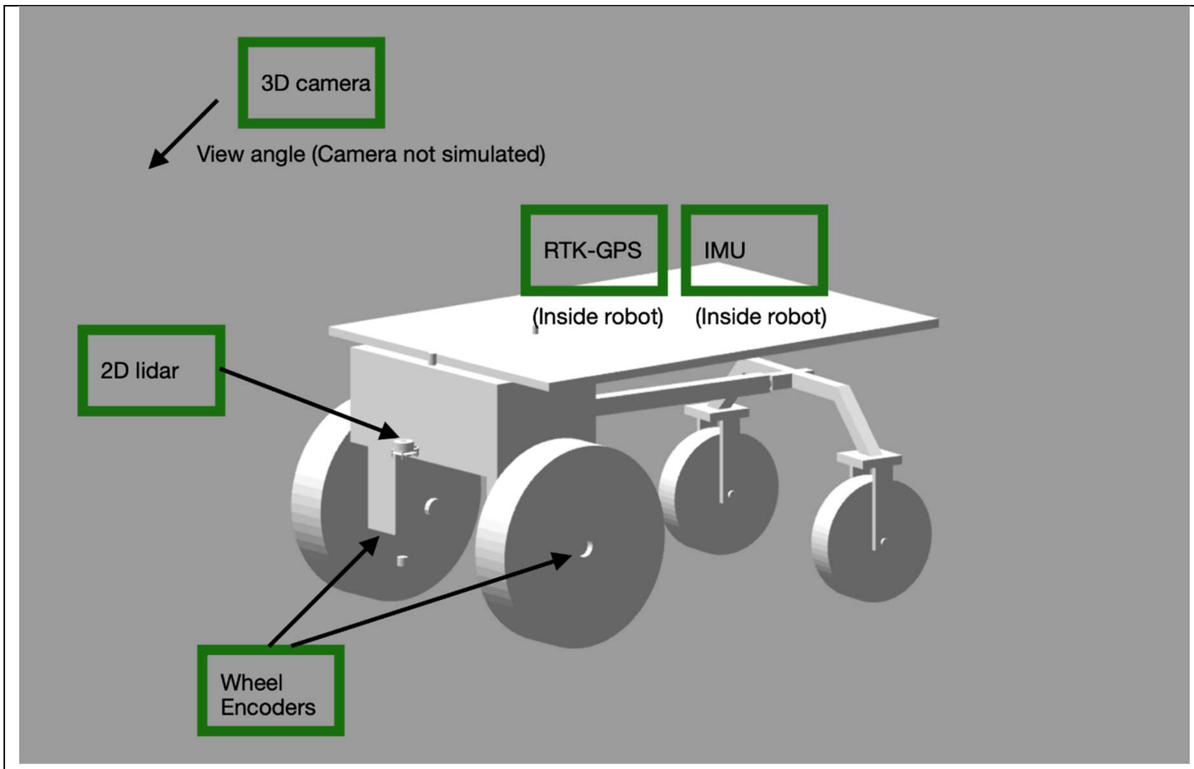


Figure 8: Sensors available on the robot

These following sensors were included:

- RTK-GPS
  For absolute positioning information. An RTK-GPS has a ground station near the operating robot, which improves accuracy.

- 3D camera
  The 3d camera provides colour images and a 3D point cloud matching each pixel. It may be used to identify crates lying on the ground. The image processing and 3D point cloud analysis is not part of our module. The input to the module from the camera was modelled as a second lidar scanner close to ground level.

- 2D lidar scanner
  Lidar scanners produce distance measurements on a (usually) horizontal plane around them. The lidar scanner is mounted in the front of the robot and does not cover the rear. The data produced is used for obstacle avoidance and for mapping [figure 9].

- Inertial Measurement Unit
  The IMU produces acceleration data and turning velocities of itself. These can be used for position estimation via integration.

- Wheel encoders
  The wheel encoders measure how much each wheel has turned. With this data, odometry can be calculated, which is a position estimation based on the revolutions of the left and right wheel and the wheel diameter.

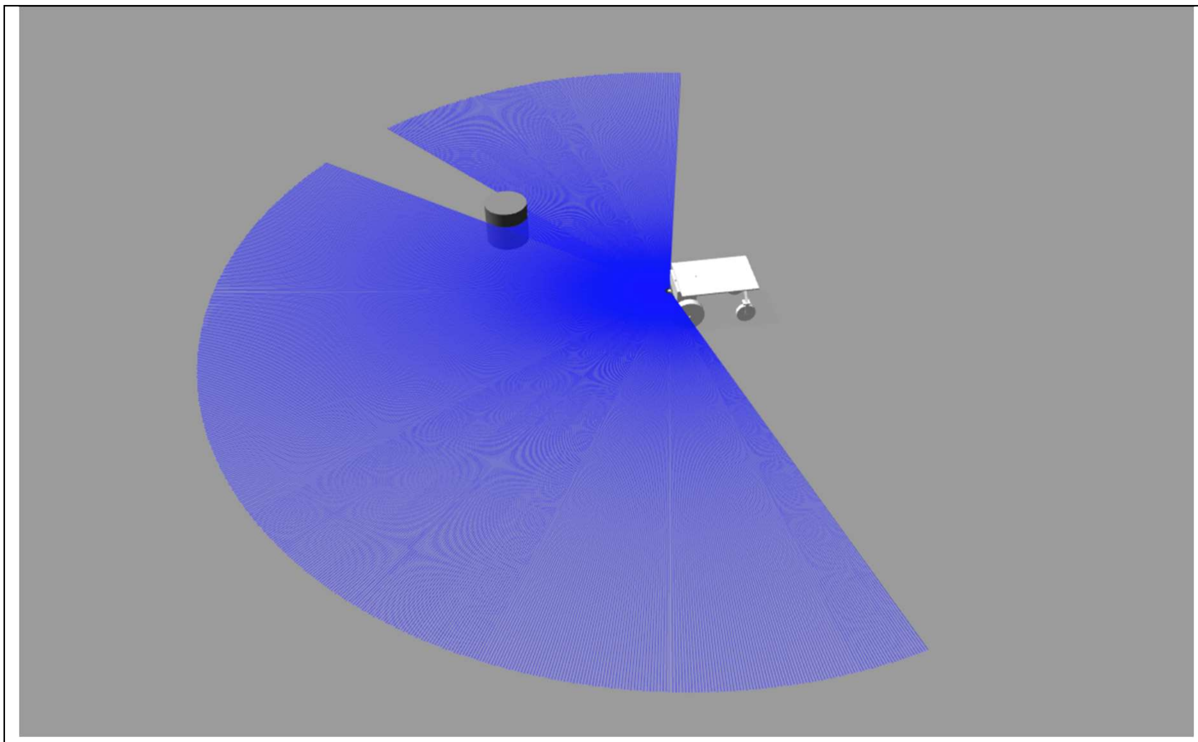


Figure 9: The Lidar in action

System architecture of ROS

ROS is a modular framework. Several open-source packages were used, as well as the standard installation of ROS of course. Not all packages contained in ROS can be mentioned here, a selection of the high-level packages was made [figure 10]. A full list of all ROS packages is in the attachments.

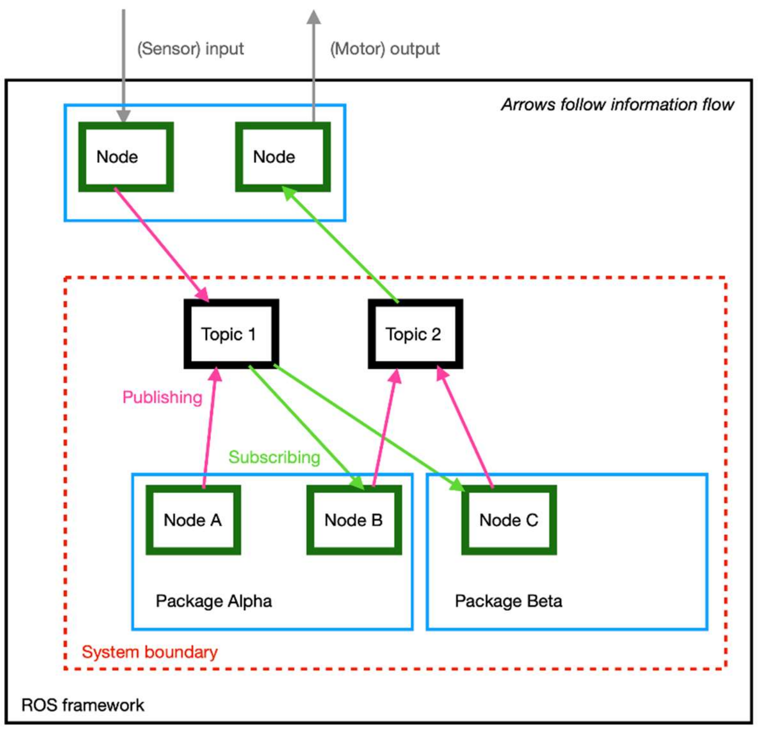


Figure 10: Graphic of the Ros Architecture

ROS provides a framework for information exchange, between different programs called nodes. Information is exchanged on channels called topics, to which any given node can subscribe (listen to) or publish onto. A publication is called a message which can contain different standardised information, for example position, time or error messages.

The software will be presented following the information flow starting from the sensors and ending with the control algorithms and outputs of the system.

GPS transformation

The GPS data needs to be transformed from the global WGS84-format (latitude and longitude) to the coordinate system of the robot. A node from the open-source robot_localization package is run for this purpose.

Sensor Fusion

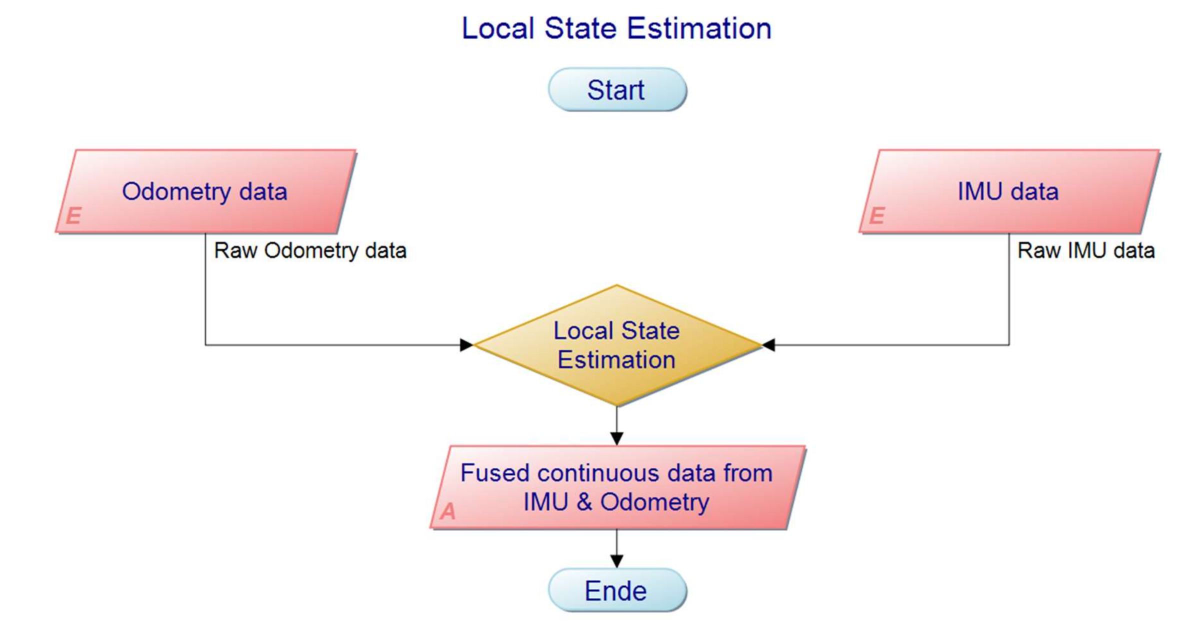


Figure 11: Local State Estimation

To achieve a more reliable result, the robot averages all information that is available and outputs a probabilistic position estimation. For the calculations an extended Kalman filter was used. The robot_localization package available as open-source is an implementation of such an extended kalman filter and was used in this module. Two position estimations are calculated.

The first node, called the local ekf node [figure 11], fuses the odometry data and the imu data. Both sources are continuous, and the result is a continuous position estimation. However, the error accumulates with time and makes the estimation drift. This is due to the relative nature of the sensor data, both the IMU and the wheel encoders only measure changes in position.

The second node is called the global ekf node and tries to remedy the drift by integrating the absolute GPS data [figure 12].

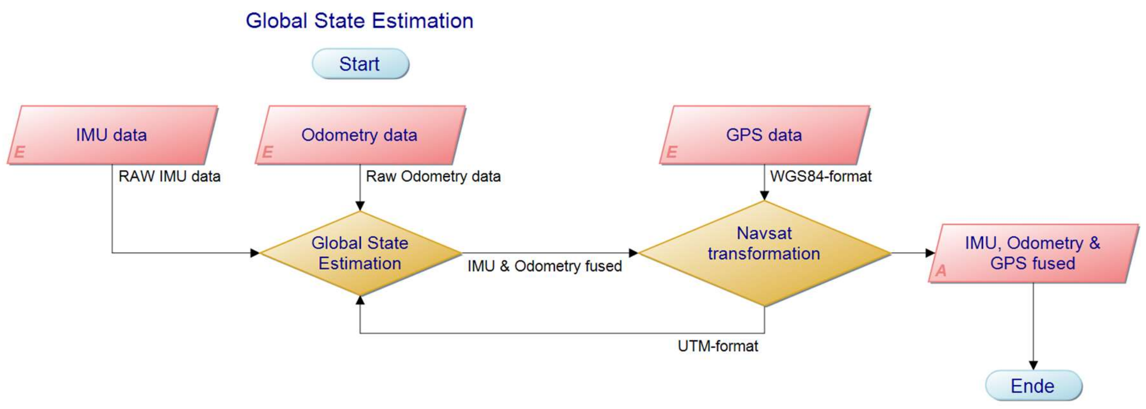


Figure 12: Global State Estimation:

It fuses the GPS data with the same inputs as the first node, the odometry and IMU data. This time, the output is no longer continuous as the GPS data can jump from one measurement to the next. This can be detrimental for map building and matching scans, which is the reason a node with only continuous data is kept in parallel.

Physical robot model and frames

The CAD model and its setup in Gazebo, the simulation engine of ROS, was provided by the company [figure 13].

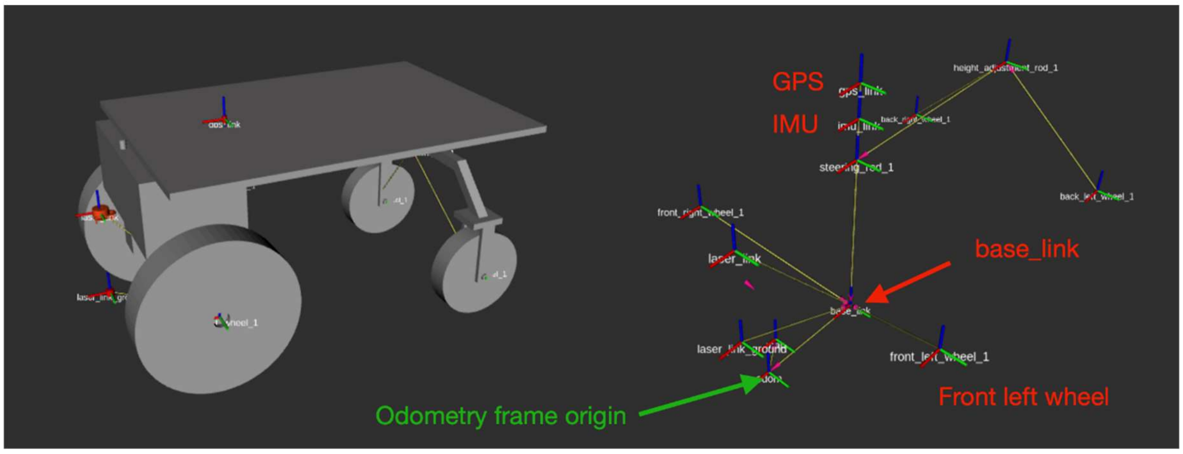


Figure 13: CAD robot model and the different frames

Before diving into the mapping and localization, it makes sense to take a step back and use the physical model of the robot as the example for introducing frames.

The heart of the simulated robot is the base_link. This is the reference piece for all parts mounted to the robot and it moves with the robot. Wheels, sensors and so on are all defined in their position in reference to the base_link.

The base_link is called a reference frame. Analog to the base_link, there exists a frame (coordinate system) for the odometry and a third for the map. The odometry frame describes the location of the robot in relation to its start point. The map coordinate system should be fixed in relation to the real world and describes the absolute position of the robot.

The transformation between different frames can either be static or dynamic. A static transformation describes the position of the Lidar sensor in relation to the base_link, it never changes. A dynamic transformation describes the position of the robot in the map frame.

Localization

The heart of the localization process is the open-source gmapping package. It takes as input the fused odometry data from the global ekf node and the laser scans from the lidar. Once integrated, it will also take into account the data from the 3d camera, transformed into planar laser scan message. It tries to transform each incoming scan into the odometry frame and match it to the already existing map data. It outputs the estimated transformation from the odometry frame to the world frame, as well as continually updating the map it is constructing. On this map it can also clear obstacles that are no longer in place [figure 14]

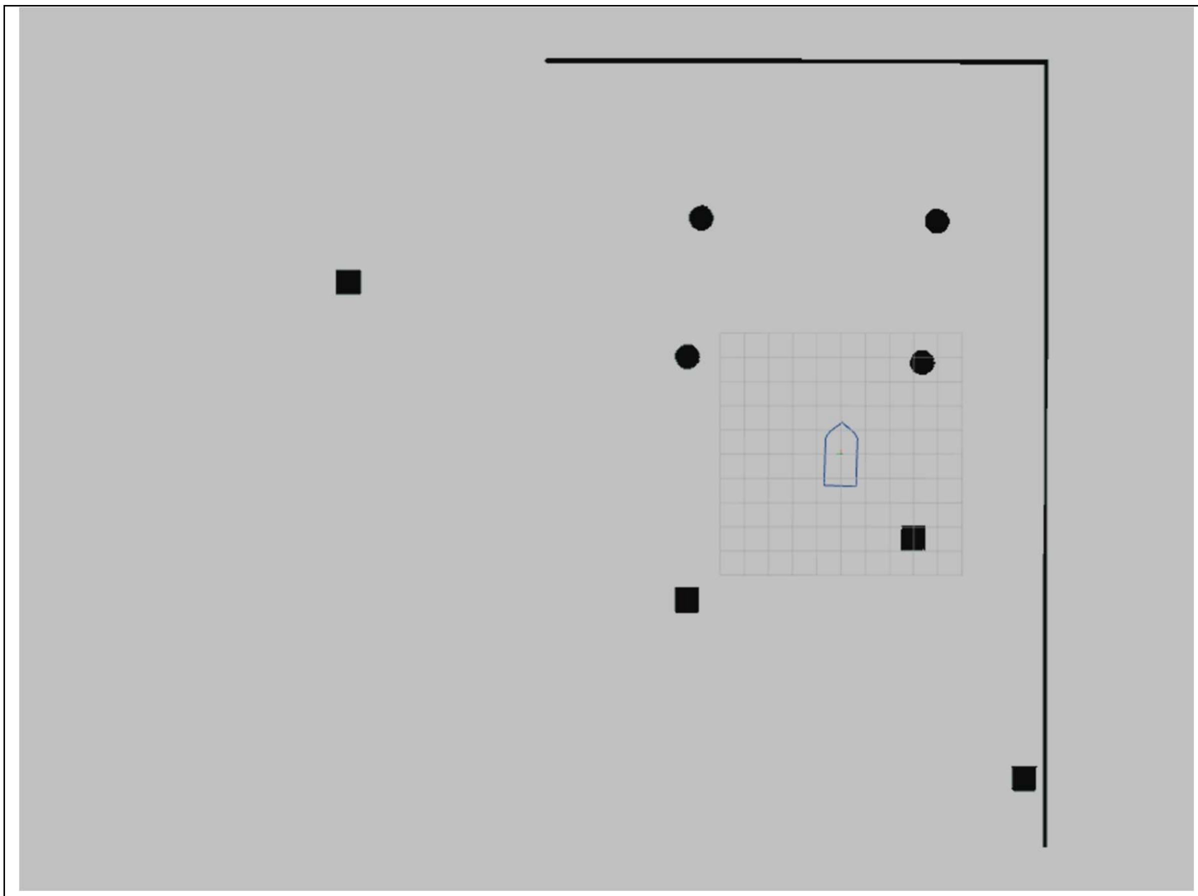


Figure 14: The static MAP

The constructed map can be saved and later reloaded with the help of the open-source map_server node. If the robot is driving in an already known map (without updating it), the open-source amcl node is used, which implements an adaptive Monte Carlo localization approach.

Navigation

The Navigation is based on the open source navigation package. It creates a global [figure 16]. and a local cost map [figure 15]. For the global cost map, an existing map that is loaded can be used as the basis. The cost map is where the robot keeps track of obstacles and they are used for path planning. To ensure that obstacles are cleared when navigating, an inflation layer is applied to the cost map to increase the size of all obstacles before applying the path finding algorithm.

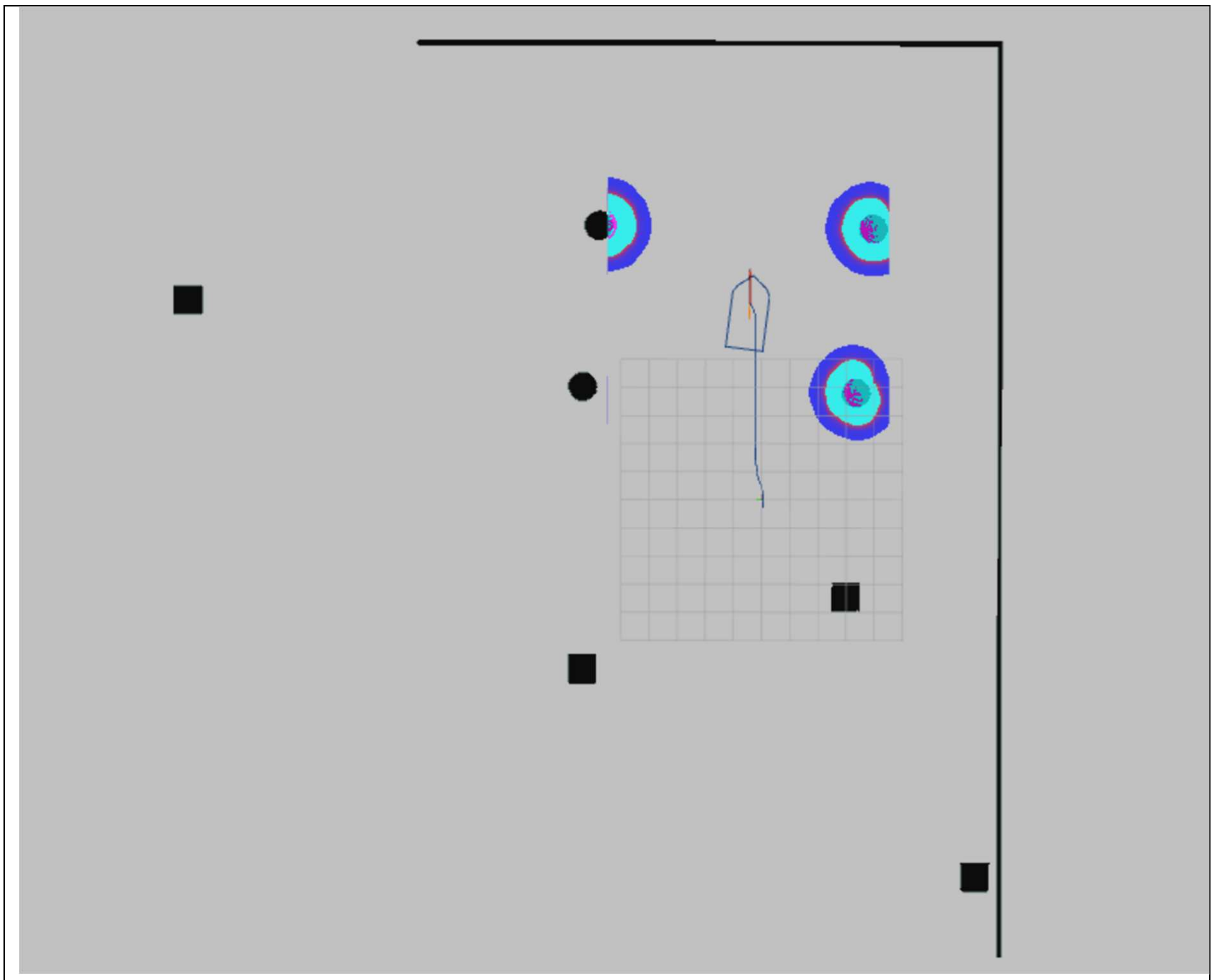


Figure 15: Local Cost map

The global cost map is updated more slowly than the smaller local cost map. The global cost map spans the whole known map and is used for the global path planning to the navigation goal. In that way, known obstacles can be avoided even if they are far away.

The local cost map is smaller and serves as the basis for the local path planner. It reacts more quickly and also takes into account the physical possibilities of the robot, such as the minimum turning radius. For the local path planner, teb_local_planner was chosen as it supports non-holomorphic robots with Ackermann drive and also supports dynamic obstacle input from a separate node.

The navigation node sends a message with linear and angular speed to the external ROS node controlling the motor drivers.

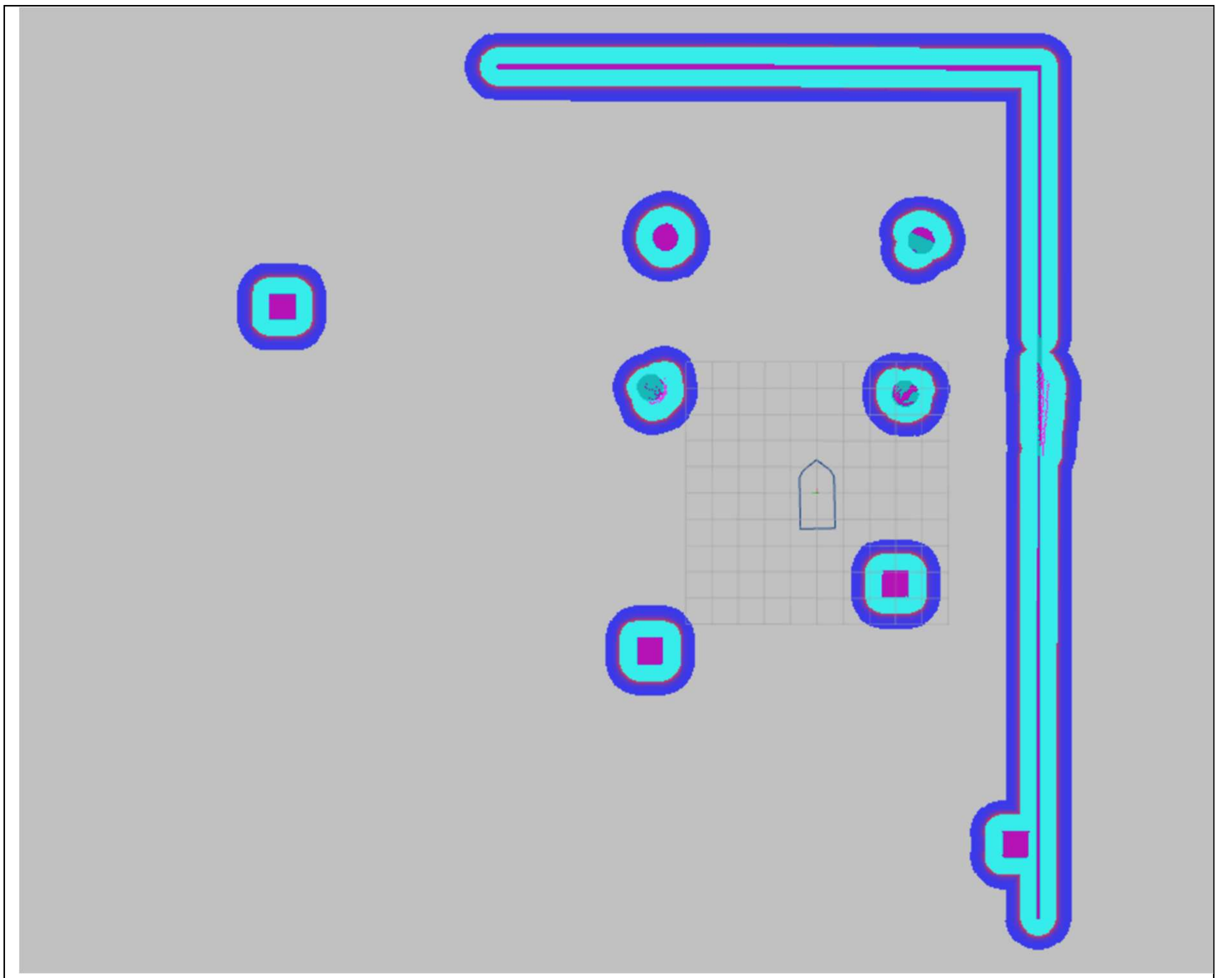


Figure 16: Global cost map

The Human Machine Interface
To get in touch with the robot we have written a Python file as a Human-Machine-Interface. This allows us to get information from the robot and also send some commands to the robot. In the next few sections, we will give you a little overview of the structure and sub functions from the Python file.

Note, if you are interested in more details check the attachment or the README.md in the fhgr_waypoints folder.

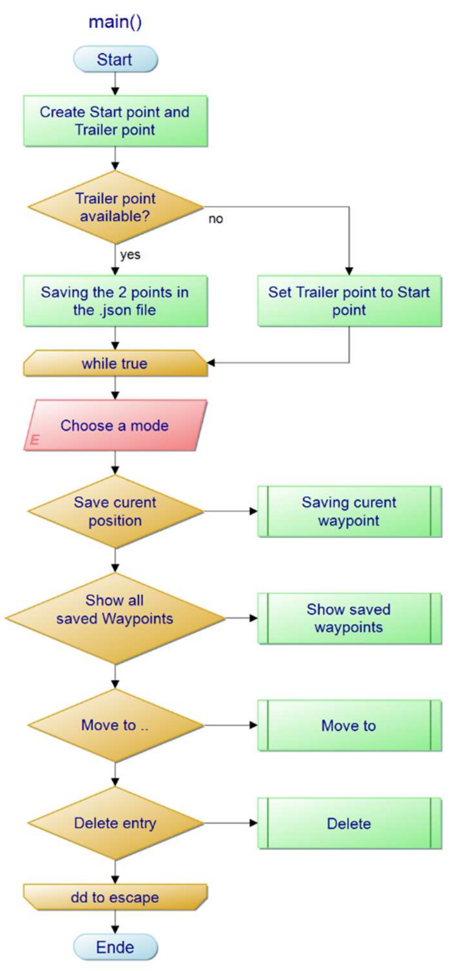


Figure 17: The main() function

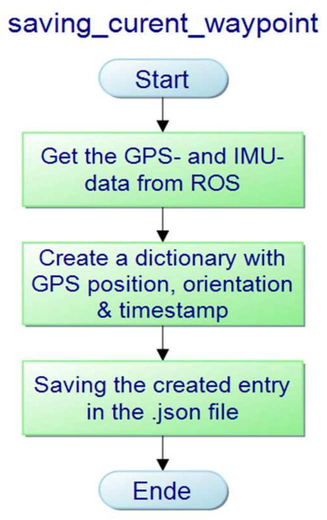
Main()
In the [Figure 17] you see the main menu routine. At the beginning, the program creates the Start point, in order to do that, the current global position, the robots orientation and the current time will be saved as a dictionary. If a Trailer point is available he will also be saved in a dictionary. In case that the Trailer point is not available, he will be set equal to the Start point. After creating those dictionaries they will be saved in a .json file.

You can choose between several options which are displayed to a command prompt, like:

- Save the current positon
- Show all saved waypoints
- Move to a user given goal
- Or delete a waypoint entry

Saving
If you want to save the current robot position, you can run the saving_curent_waypoint function, which is explained in the [figure 18].

Therefore, the program will get the x- and y-GPS-Coordinates and the robots orientation from ROS. This information and the current timestamp will be saved in a dictionary. The whole dictionary gets saved in a .json-file. And the main() menu will be displayed again



saving_curent_waypoint
Start
Get the GPS- and IMU-data from ROS
Create a dictionary with GPS position, orientation & timestamp
Saving the created entry in the .json file
Ende

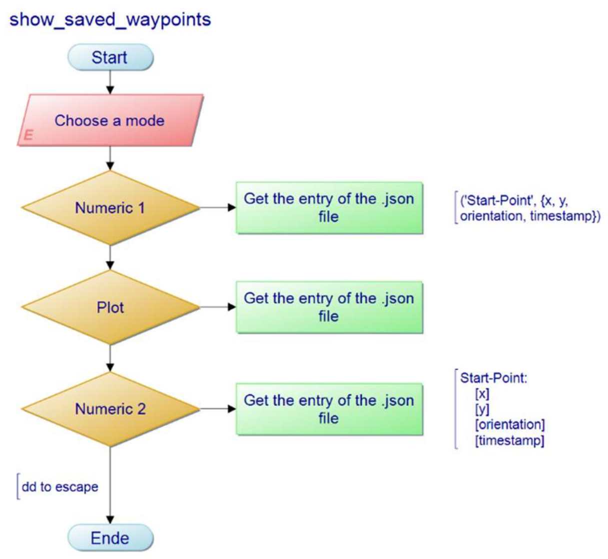
Figure 18: Saving current waypoint

Show saved waypoints

The [figure 19] get an overview of the three different output methods of the saved waypoints.

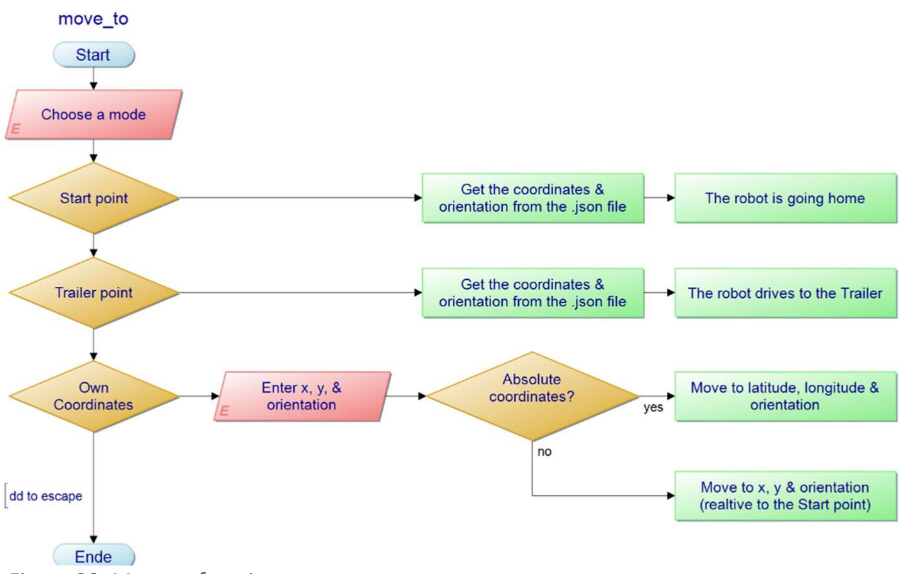
You can choose between:

- Two different numeric console outputs, which are showed in the [figure 19]
- Or one plot output.



show_saved_waypoints
Start
Choose a mode
Numeric 1
Get the entry of the .json file
('Start-Point', {x, y, orientation, timestamp})
Plot
Get the entry of the .json file
Numeric 2
Get the entry of the .json file
Start-Point:
[x]
[y]
[orientation]
[timestamp]
dd to escape
Ende

Figure 19: Show all saved waypoints



move_to
Start
Choose a mode
Start point
Get the coordinates & orientation from the .json file
The robot is going home
Trailer point
Get the coordinates & orientation from the .json file
The robot drives to the Trailer
Own Coordinates
Enter x, y, & orientation
Absolute coordinates?
yes
Move to latitude, longitude & orientation
no
Move to x, y & orientation (realtive to the Start point)
dd to escape
Ende

Figure 20: Move to function

The information of the waypoints is stored in the .json file. If you choose the plot output, it will display the waypoints with the relative distance to the start-point by default. If you are interested in the global position of the waypoints, you can also do that, and you will see the waypoints with the latitude and longitude values.

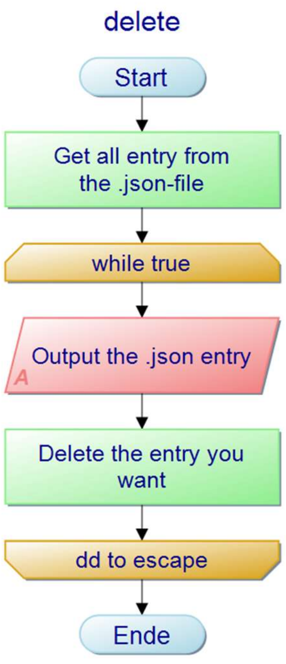


Figure 21: Delete an entry

Move
To move the robot, you can have a look at the [figure 20]. There you have three options:

- You can let the robot move to his Start point
- Move to the Trailer
- Or give him your own x- and y-coordinates and an orientation

If you want to let the robot move to user-given-coordinates you have the choice between relative or absolute coordinates. The relative coordinates are relative to the robots start-point and the absolute coordinates are written in latitude and longitude. The orientation is sent with Euler-angle in degrees around the z-axis (0 +- 180)

Delete
In order to delete a waypoint entry, you can call the function delete() in [figure 21]

The program gets all saved waypoints from the .json file and prints them to the screen. You can now easily choose the number of the waypoint you would like to delete.

6. Testing

In this chapter the testing process and the virtual environment are described.

Virtual environment

One of the specifications for this project is that this project is software based and can be tested in a simulation without the real robot or additional hardware. See “Specification_book_signed.pdf” in the attachment for more details about this specification.

The client already created a virtual model of the robot and made it accessible for this project. But to actually be able to test the module with the virtual model, it was necessary to create a small virtual world that contains objects, similar to what the real robot would encounter.

At first a minimalistic world was created to resemble small strawberry field and a place to unload the crates. This world was created to keep the processing power and the loading time low for easier development.

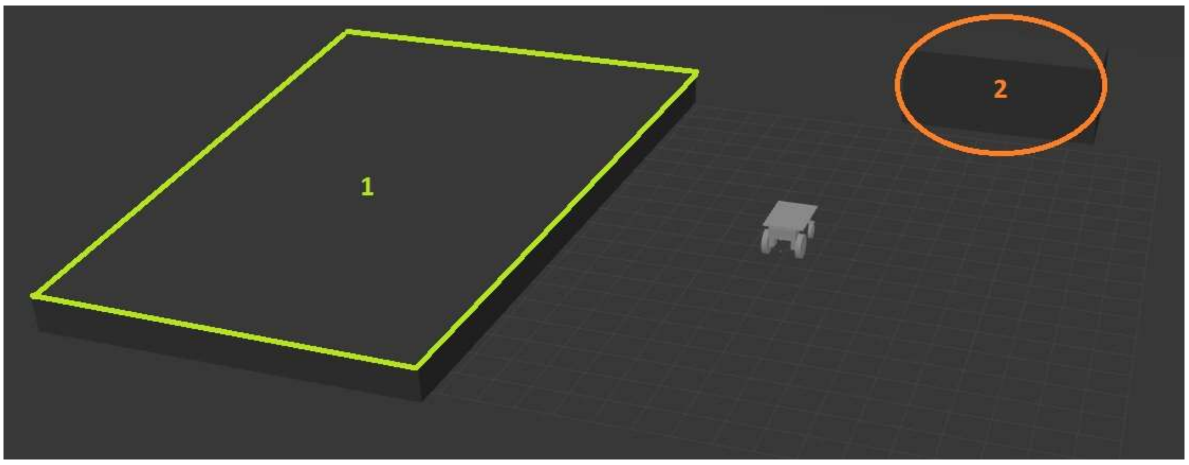


Figure 22: Minimalistic field

1. Minimalistic strawberry field
2. Unloading place

Based on this empty world, another world was created. This world contains parts of possible environments.

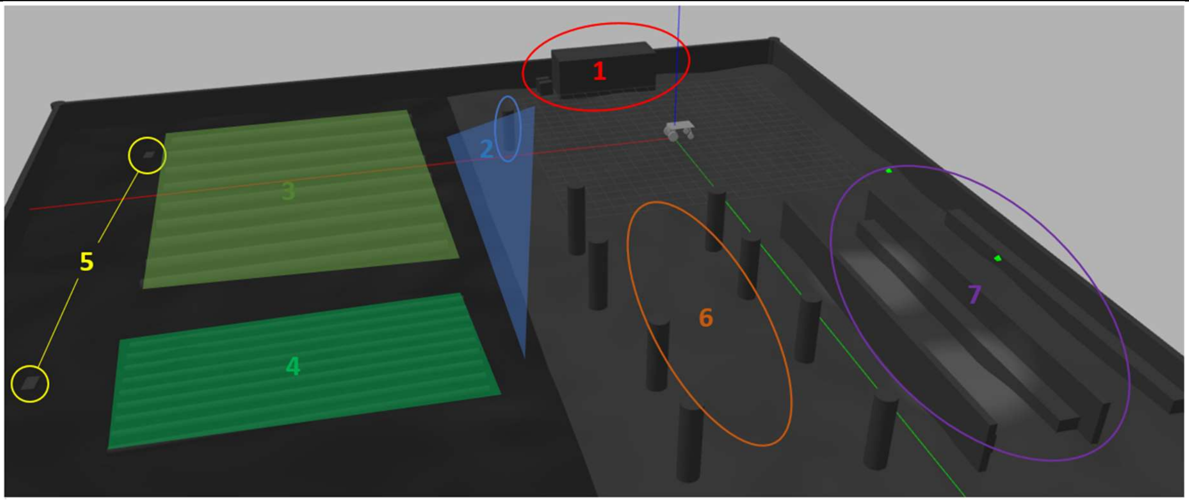


Figure 23: Filled field

3. Place to unload creates
4. Moving object (simulating a human slowly walking around the blue area)
5. Wide crop rows (wide enough for the robot to drive between the rows)
6. Narrow crop rows (narrow enough for the robot to drive with the rows between its wheels)
7. Crates (randomly placed crates that might be an obstacle on a real field)
8. Tree rows (trees could be an obstacle on a real field)
9. Indoor rows (with and without additional light sources)

For testing our project outcome, we oriented ourselves on the specification book. The short outcome is listed in [table 3] listed in short as follows:

Nr	Status	Keyword	Description
1	pass	Waypoints	Waypoints define the goal for path planning. One goal at a time is defined by a module outside the system boundaries. Waypoints are defined as a coordinate in the world frame.
2	pass	Position marker	Starting position and important markers for the return journey (transition field - road) can be saved on request.
3	pass	Path planning	The robot can plan its movement from the current position to the next waypoint.
4	pass	Driving control	A control algorithm regulates the execution of the planned path.
5	pass	Obstacle avoidance	On an encounter with an obstacle, alternative paths can be calculated or the robot is stopped if no safe alternatives are found.
6	pass	Disabling alternative paths	Obstacle avoidance can be turned off, for example inside crop rows. This is a safeguard to prevent damage to plants.
7	pass	Mapping	A 2D map of fixed size (determined in runtime by another module) is created and saved. It receives obstacle data and crop row positions to add to the map.
8	(pass)	Localization	The robot can localize itself in the map and with the help of positioning data (for example GPS).
9	pass	Documentation	The contractors will write a report about the project and hand it over at the end of the project. In addition a presentation will be held and a short video is produced showcasing the project functionality.
10	pass	Legal requirement	The client takes care of the legal requirements. Only legal requirement the contractors have to keep in mind, is the robots current speed limit of 5km/h.
11	pass	Open source code	Open source code may be used in the project when it makes sense, but usage of any such component must be green-lit by Ant Robotics first.
12	pass	Modularity	The software project shall be modular in its nature, such that it can use the data provided to it by other ROS packages independent of the data source. It will be activated and commanded by another module.
13	partial	Simulation testing	The project is software based and shall work on the simulation without the robot or additional hardware.
14	pass	Programming language	The project should be developed with C++. Alternatively, python can be used for development.
15	pass	Framework	The module is based on ROS1 Noetic.
16	pass	Calculations	All calculations and actions are done and saved locally on board.

Table 3: Short listed outcome

The [table 4] shows the testing documentation with explanations:

Nr	Status	Keyword	Description
1	pass	Waypoints	With the Python file as Human machine interface we can load defined waypoints from a .json file and save them. Sending waypoint goals is also possible
2	pass	Position marker	This is done with the Python file with the 'Saving Waypoint' function
3	pass	Path planning	For this we have a local and global planning algorithms from a Ros package
4	pass	Driving control	The Ros package 'teb_local_planer' takes this task, for fine tuning there is a handy simulation where you can put obstacles in the calculated way of the robot and see, how he will change his path to the goal.
5	pass	Obstacle avoidance	Therefor we simulated a Lidar, which is able to detect incoming obstacles. Afterwards the local- and global-planning algorithms from a Ros package take over to calculate the way around the obstacle
6	pass	Disabling alternative paths	During the crop rows the Ant Robotics algorithm taking over the steering, our package is then reduced to map-only-mode
7	pass	Mapping	This task is done by the Ros module map-server. To save the created world a shell-command is necessary
8	(pass)	Localization	With the 'robot_localization' package we were able to fuse the Odometry-, IMU- and GPS-data. But it's buggy
9	pass	Documentation	This is achieved while you are reading
10	pass	Legal requirement	-
11	pass	Open source code	Done by using Ros packages and write other code by our self
12	pass	Modularity	To achieve this we created our own Ros package. The Python file is ready to take some other data for localization
13	partial	Simulation testing	Due to the fact, that our robot is not able to move properly the subject testing could not been executed.
14	pass	Programming language	For the human machine interface, we decide to write it in Python. It makes the plotting easier and also the understanding of the code itself. Additionally its not a time critical task
15	pass	Framework	Only ROS 1 noetic in use
16	pass	Calculations	All calculations and actions are done and saved locally on board.

Table 4: Outcome with explanations

For testing we had multiple runs in the simulation where we could see if the implemented packages and functions run properly. If not, there were the option to fine tune the parameters. Also, the different functions and requirements flow into each other. So, you can let the robot move to a point where under the hood the path planning, the map server, the lidar and the localization algorithms is running together.

As you can see, the testing in isolated conditions was mostly successful. BUT we had a major challenge to deal with, which is the fact, that our robot is not able to move properly. Unfortunately, we were running out of time, and we were faced to finish the programming or implementation of the Ros packages and start to write the documentary and create the Movie. There for we were not able to write a good and reproducible testing routine.

Another thing to point out is the enormous computing power that the simulation needs. The fans of our Laptops where going insane but the output was quite poor. So we could not really load the self-made-real-looking-world and let the robot explore in it. The simulation was quite jerky.

How good is the outcome?

The table below [table 5] shows on the left colourful how the project outcome works. In the Status cell you will see if the outcome is stable or not.

	Status	Keyword	Description
🟩	stable	Waypoints	The Waypoints are stable and are saved in a .json file, which allows you to access it for further use
🟩	stable	Position marker	Also, in the .json file with the timestamp you can also understand, when the waypoint was saved
🟥	unstable	Localization	Here we have a bug into the sensor fusion part, but we were not jet able to localize it. Because the modules have a strongly dependency under each other's the following tasks are not really stable.
🟥	(stable)	Simulation testing	In isolated condition the simulation testing works, but when it comes to test the project outcome it fails, because the robot is not able to drive stable
🟨	(stable)	Path planning	If there is a navigation goal send, the local and the global path planning works well
🟥	unstable	Driving control	The planed path is quite good executed, but sometimes the robot drives backwards
🟨	(stable)	Obstacle avoidance	The Lidar is working well and detecting the static objects, it can happen, that the robot
🟩	stable	Disabling alternative paths	This feature is working good and stable
🟩	(stable)	Mapping	The map is created, but if the robot begins to lose himself in the simulation the output of her is not usable
🟨	stable	Documentation	Was written shortly before submission, could have been done better, if there were more time left
🟩	stable	Legal requirement	The robot drives not faster than 5 Km/h
🟩	stable	Open source code	Ros packages or self-written code
🟩	stable	Modularity	If you want to implement an other position system (for example for indoor navigation) you will need to implement this into the Python-Human-Machine-Interface
🟩	stable	Programming language	Ros and Python and also a good code documentation and description
🟩	stable	Framework	-
🟩	stable	Calculations	-

Table 5: How good is the project outcome

7. Conclusion

- ROS is not just a python-file with a few functions, it is an operation-System. So it is quite hard to get into it.
- We have learned very much about ROS and how the basic Models are working together.
- It was also a good training for our upcoming bachelor thesis to know how to face a big project like this one was.
- As a team we three had a good time together and our communication under each other was pretty good.
- Creating the virtual environment was way more time consuming than expected. Reason for that was the appearance of multiple bugs. The first bug would mess up the whole map once it was saved, closed and reopened. After time wasting problem finding, a possible cause was found. If the ctrl+z command was used while creating the map instead of the undo button, this bug would occur when the map gets opened the next time. Another bug is that the objects on the map would fall through the ground. Cause for this is, that the density of the ground sometimes gets set to 0 and can be solved by setting it to a higher value. There was no solution found to stop the bug from appearing. The last bug is, that textures on newly imported models wont show or get messed up and look very strange. There was no solution found for this bug and the newly imported models are all grey.
- At the end of the project, we were not able to keep up with our time plan, because the implementation of the different ROS packages turned out to be more difficult than expected.
- A good Documentation of the ROS packages is kind of hard to get. There is quite few information about the theoretical aspect of the ROS packages on the internet but when it comes to the implementation in your own project, you are mostly quite lost.
- Unfortunately, we were running out of time, and we were faced to finish the programming or implementation of the ROS packages and start to write the documentary and create the Movie. There for we were not able to write a good and reproducible testing routine.

8. Further development

- Fixing the robots behaviour not to drive backwards. Also make the forward path planning smarter, in order to the escape behaviour. For example, if you let the robot drive in a corner he will be trapped, because he should not drive backwards.
- Get a solution for the drift in the State Estimation. In our logic the global state estimation and the GPS should be close together and the local state estimation is allowed to drift. However, in our print of the different state estimations the global state estimation is the one who drifts away. We have made a plot function to visualize this issue [figure 24]
  We already calculated the transformation between the different frames without any changes. It is likely, that the problem is caused by a wrong setting in the different simulated sensors.

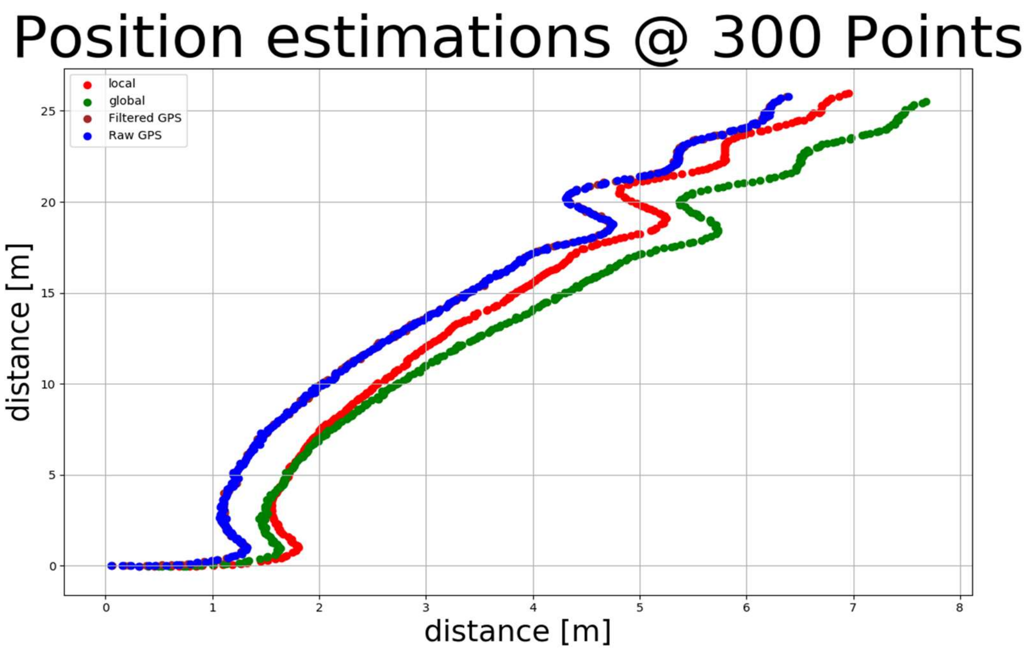


Figure 24: Plot from the different State estimations

9. Register

All illustrations without any source data were made by the creators of this document.

Figures

Tables

Attachment

Concept_desicion.pdf

Project_status_v4.pdf

Specification_book_signed.pdf

README.pdf